VILNIUS UNIVERSITY

Gintaras Vaira

GENETIC ALGORITHM FOR VEHICLE ROUTING PROBLEM

Doctoral Dissertation

Technological Sciences, Informatics Engineering (07 T)

Vilnius, 2014

The dissertation has been prepared during the period 2009 – 2013 at Vilnius University.

Scientific supervisor:
assoc. prof. dr. Olga Kurasova (Vilnius University, Technological Sciences, Informatics Engineering – 07 T).

VILNIAUS UNIVERSITETAS

Gintaras Vaira

GENETINIS ALGORITMAS TRANSPORTO MARŠRUTŲ SUDARYMO UŽDAVINIAMS SPRĘSTI

Daktaro disertacija

Technologijos mokslai, informatikos inžinerija (07 T)

Vilnius, 2014

Disertacija rengta 2009 – 2013 metais Vilniaus universitete.

Mokslinė vadovė:

doc. dr. Olga Kurasova (Vilniaus universitetas, technologijos mokslai, informatikos inžinerija – 07 T).

Acknowledgments

First of all, I would like to express my thanks to my scientific supervisor Assoc. Prof. Dr. Olga Kurasova for the continuous support and guidance throughout the process of this dissertation research, for her patience, motivation and valuable scientific advice. I want to thank Prof. Dr. Habil. Gintautas Dzemyda for encouraging me to do my doctoral studies.

I greatly appreciate the time and effort of the reviewers Prof. Dr. Habil. Artūras Kaklauskas and Prof. Dr. Julius Žilinskas who carefully read the initial version of this thesis and provided with valuable comments and advice that helped me to improve the quality of the work.

Special thanks to Vita Jurevičiūtė, Kristina Pociuvienė and Janina Kazlauskaitė who helped me with the proofread of my papers and this thesis.

I would like to thank the Lithuanian State Studies Foundation for the financial support for the doctoral studies.

Also, I wish to thank my parents, brothers, all relatives and friends for all their support during the preparation of this thesis. Most of all I wish to thank my wife Lina for the patience, love and support during this challenging period of my life.

Finally, I would like to express my thanks to all the people who have been in one way or another involved in the preparation of this thesis.

Abstract

In recent years, a vehicle routing problem (VRP) attracts much attention due to the increased interest in various geographical solutions and technologies as well as their usage in logistics and transportation. Many researches on different heuristic approaches can be found for the solution of the vehicle routing problem, where specific situations and constraints are analyzed. In this research we investigate genetic algorithm approaches for solving vehicle routing problem with different constraints. Due to stochastic characteristics, genetic algorithms generate solutions in the whole search space including the infeasible space. For a constrained problem, the feasible search space is smaller than the whole search space. Having constraints in the problem definition, the aim is to find the solution that does not violate any constraint. Such solution is called a feasible solution or feasible individual. The common genetic algorithm approaches involve additional repair and improvement methods that are designed for a specific constraint to keep the generated solutions in the feasible search space. The usage of the repair and improvement methods designed for specific constraints or genetic operators specially designed for a specific problem can produce an inadequate result when they are applied to different problems. In this thesis we propose a genetic algorithm based on a random insertion heuristics for the vehicle routing problem with constraints. The random insertion heuristic is used to construct initial solutions and to reconstruct the existing ones. The process of random insertion preserves stochastic characteristics of the genetic algorithm and preserves feasibility of generated individuals. The defined crossover and mutation operators incorporate random insertion heuristics, analyze individuals and select which parts should be preserved and which should be reconstructed. The second population increases the probability that the solution, obtained in the mutation process, will survive in the first population, thus increasing a diversity in the population and the probability to find the global optimum. The proposed operators are not designed to a certain specific problem and can be applied to different problems. The proposed algorithm can be applied for the rich vehicle routing problem. No additional repair or improvement methods are used that could be a problem for extending scheme with a new constraint handling.

Contents

List of Figures

List of Tables

Glossary

Chromosome – the literal string encoded form of solutions that the classical genetic algorithm paradigm deals with.

Crossover operator **–** recombination operator in genetic algorithm, where new solution in new generation is created by taking into account more than one solution from previous generation.

Dijkstra's algorithm – breadth first search algorithm for search of the shortest path in the graph proposed by Dijkstra (1959).

Decoding – transformation of the chromosome to the solution.

Encoding – the representation of the solution as a chromosome.

Feasible search space – set of all possible feasible solutions.

Feasible solution – a solution that satisfies all the constraints defined in the problem.

Generation **–** the population in certain iteration of the genetic algorithm.

Genetic algorithm – search heuristic that is based on ideas of evolution theory (Holland, 1975). A genetic algorithm works with the population and usually has following components: representation, fitness function evaluation, initialization, selection, recombination (crossover and mutation), termination.

Genetic operator – one of the recombination operators (crossover or mutation) used in the genetic algorithm.

Individual – the single solution in genetic algorithms, where encoding/decoding is bypassed.

Insertion heuristic – construction heuristic where solution is created by inserting elements one by one by evaluating certain functions to select the element and the place in the solution for insertion.

Mutation operator – recombination operator in genetic algorithm, where new solution is created from the single solution by changing some characteristics within it.

Population – the set of the solutions in the genetic algorithm.

Rich vehicle routing problems – the family of the extended vehicle routing problems that includes several or all aspect of real-life vehicle routing (Hartl et al., 2006).

Vehicle routing problem – general name given for a class of problems, in which a set of vehicles service a set of customers.

Abbreviations and Acronyms

BCRC	Best cost route crossover operator proposed in (Ombuki et al., 2006).
CAX	Common arcs crossover operator proposed in this research.
CNX	Common nodes crossover operator proposed in this research.
CVRP	Capacitated vehicle routing problem.
GA	Genetic algorithm.
LCIS	Longest common increasing subsequence.
LCSX	Longest common sequence crossover operator proposed in this research.
LNS	Large neighborhood search.
LRX	The crossover operator used in (Alvarenga et al., 2005), in this thesis it is called largest route crossover.
PDPTW	Pick-up and deliver vehicle routing problem with time windows.
RBX	The crossover operator proposed in (Potvin and Bengio, 1996) that is called a route-based crossover.
RVRP	Rich vehicle routing problem.
TSP	Traveling Salesman Problem.
VRP	Vehicle Routing Problem.
VRPPD	Vehicle routing problem with pick-up and deliveries.
VRPTW	Vehicle routing problem with time windows.

Symbols

A_r	Set of arcs in the solution x_r.
$a_{ri} \in A_r$	Single arc in the set A_r.
C	Set of constraints.
$C_c \in \mathrm{C}$	Set of capacity constraints.
$C_{pd} \in \mathrm{C}$	Set of pick-up and delivery constraints.
$C_{tw} \in \mathrm{C}$	Set of time window constraints.
$f(x)$	Objective function.
$f_c(x)$	Function that evaluates single constraint violation.
$f_d(x)$	Objective function to minimize total travel distance.
$f_v(x)$	Objective function to minimize number of vehicles (routes).
$f_f(y)$	Objective function for feasible solution.
$f_u(x)$	Objective function for infeasible solution.
$F_c(x)$	Function that evaluates violation of all constraints.
$f(a_i, t_m)$	Evaluation function of task t_m insertion in arc a_i.
$G=(N, E)$	Graph that defines visiting nodes and arcs of the VRP.
$G_r^T=(T_r, A_r)$	Graph that defines routing solution x_r.
$G^r=(N^r, E^r)$	Graph of the road network.
$h_c(n, a)$	Evaluation function of node n insertion in arc a.
k	Number of customers in the problem or number of the tasks in the problem.
$l(n_i, n_j)$	Distance between node n_i and n_j.
$l_s(n_v^r)$	Distance to the starting node in shortest-path calculation.
M	Set of request in VRP.

N	Set of the nodes in VRP.
$n \in N$	Single node in VRP.
n_0	Depot node.
$Nh(x)$	Neighborhood of the solution x.
N^r	Node list of the road network.
$n_i^r \in N^r$	Single node in the road graph.
$P(n_v^r)$	Shortest-path from starting node to the node n_v^r.
$P^{\vert}(n_u^r)$	Shortest-path from node n_u^r to the end node.
Q	The priority queue Q of labeled nodes in shortest-path calculation.
$r=(n_1, n_2,...)$	Defines the route of nodes.
S	Full search space.
S_F	Feasible search space.
S_U	Infeasible search space.
$s_i=(t_{i1}, t_{i2},...)$	Single sequence of tasks in the VRP solution x.
$T=\{t_1, t_2,...\}$	Set of tasks in the VRP.
$t_i \in T$	Single task in the VRP.
$t_a(n_j)$	Time of the arrival at the node n_j.
U_i	Set of nodes that left unserviced in solution x_i.
V	Set of the vehicles in VRP.
x_i	Single solution of VRP.

Introduction

Research context and motivation

The vehicle routing problem (VRP) is a well known combinatorial problem that attracts researchers to investigate it by applying the existing and newly created optimization algorithms. Traditionally, the VRP is defined as a routing problem with a single depot, a set of customers, multiple vehicles and the objective to minimize the total cost while servicing every customer. A set of constraints can be defined for the VRP. In literature we can find different kinds of vehicle routing problems (VRPs) that are grouped according to the specific constraints. The well known constrained VRPs are as follows: VRP with capacity limitations (CVRP), where vehicles are limited by the carrying capacity; VRP with time windows (VRPTW), where a customer can be serviced within a defined time frame or time frames; VRP with multiple depots (MDVRP), where goods can be delivered to a customer from a set of depots; VRP with pick-up and delivery (VRPPD), where rules are defined to visit pick-up places and later to deliver goods to the drop-off location. Many researches on different heuristic approaches can be found for the solution of the above mentioned problems.

In recent years, VRP attracts much attention due to the increased interest in various geographical solutions and technologies as well as their usage in logistics and transportation. More and more logistic companies are trying to organize deliveries of goods better by enabling various today's proposed technologies. They can be various logistic systems coupled with widely used positioning systems, etc. The important part in reduction of transportation costs is a better organization of routes by solving a vehicle routing problem. For example, a better organization of fleet routes in various distribution areas – delivery of post, supply delivery to markets, fuel delivery to gasoline stations, etc. – can save fuel, money and/or time that can be used for servicing new customers. Also, a better organization of routes in business deliveries can affect ecological aspects by reducing pollution that is important problem of these days.

Problem statement

In literature we can find algorithms that are designed for one or another VRP, where the algorithms are designed to deal with a specific subject or specific constraints. Although mentioned VRP variants mimic some real world situations, these situations do not reflect the whole problem. The mentioned VRPs are criticized for being too focused on specific models that involve non-realistic assumptions. Real-world VRP with various constraints generalizes traditional VRP and is usually called a rich vehicle routing problem (RVRP). Solving RVRP has been a challenging today's task.

A number of different exact and heuristic methods have been studied to solve the VRP that is known to be NP-hard. Although the exact methods give the optimal solution, their computation time considerably increases with the increasing size of the problem. Various heuristic methods exist for solving problems that are known to be NP-hard. Local searches and heuristic approaches may be sensitive to the given data sets (i.e., constraints) or require additional training data during the learning process. Also hybrid combinations of various algorithms are designed while seeking for higher efficiency in the computation.

Metaheuristic is another approach for solving a complex problem that may be too difficult or time-consuming for other techniques. One of the metaheuristics that are investigated for solving VRP is a genetic algorithm (GA). Genetic algorithms are based on ideas of evolution theory. The main principle here is that only the fittest entities survive. Genetic algorithms work with individuals, sometimes also called chromosomes, each representing a possible solution to a given problem. GA typically works with the initial population of solutions; together with each new generation GA creates a new potential offsprings, based on the selected individuals from the previous generation using a set of stochastic transition operators (crossover and mutation). The iterative process of generations and evaluation of individuals continues until a sufficient stopping criterion is met.

The standard genetic algorithm has limitations in the constrained environment. Due to a stochastic characteristic, genetic algorithms can continue very long until the acceptable solution has been found for a constrained problem. For a constrained problem, the feasible search space is smaller than the whole search space and genetic algorithm operators generate solutions in the whole search space including the infeasible space. The common approaches for constraint handling in genetic algorithms involve additional repair and improvement methods that are designed for a specific constraint to keep the generated solutions in the feasible search space. The repair of one constraint can involve the violation of another constraint. Such approaches can produce an inadequate result when they are applied to different problems and are hardly extendable with new constraints. Specialized algorithms usually are hardly applicable to RVRP.

Tasks and objectives of the research

The objective of the thesis is to design a new genetic algorithm for vehicle routing problem that handles constraints in genetic operators and that can be efficiently applied for solving rich vehicle routing problem.

In order to achieve the objective, the following tasks are stated:

- To study existing genetic algorithms for solving vehicle routing problems.

- To analyze approaches in genetic algorithms for dealing with constraints in vehicle routing problems and investigate search intensification approaches in genetic algorithm operators.
- To analyze the existing formulations of rich vehicle routing problem and detail them.
- To propose a new genetic algorithm for rich vehicle routing problem, where genetic operators handle constraints in solutions in each iteration.
- To investigate Dijkstra's shortest path algorithm speed up techniques in order to efficiently apply the proposed genetic algorithm to the real vehicle routing problems taking into account the road network.
- To evaluate the proposed genetic algorithm by applying it on public available benchmark instances and compare it with other known genetic algorithms.

Practical significance of the results

The practical significance of the thesis is as follows:

- The proposed genetic algorithm can be applied to real-world vehicle routing problem more flexibly and in such way reduce costs for various companies that deal with delivery by reducing overall traveling path and/or traveling time. The algorithm also can be applied to dynamic re-computation of the VRP depending on new data (new requests came from customers; some accident happened for one of the vehicles during the delivery; etc.).
- The proposed algorithm can be applied to any problem that can be defined as a graph and which solution depends on the sequence of the elements.
- A part of the research was used in the project “Algorithm for optimizing the route between N points and algorithm for fixing deviations and

mathematical averaging of fuel level data from transport means (fuel filling pouring off)" based on the agreement between "Institute of Mathematics and Informatics", JSC "AKTKC – Apsaugos centras" and "Agency for Science, Innovation and Technology" for achievement of innovation voucher (agreement No 31V-79, 2010-07-28).

Research methods

Both the exploratory research and systematic review have been used to collect and summarize the results of other researches. Experimental research and generalization method have been used to evaluate the proposed methods and algorithms in comparison with the obtained results in other researches.

Defensive propositions

1. Insertion heuristic in genetic algorithms is suitable not only to produce initial solutions, but also can be incorporated in genetic operators for constraint handling, i.e. for generation of feasible partial solution in each iteration by evaluating constraints. By repeatedly applying destroy method and random insertion heuristic, diversification is enabled in the population and, by dealing only with feasible solutions, infeasible search space is not examined, thus avoiding unnecessary computation and increasing overall computation speed.
2. The crossover operators that preserve common sequence from two parent solutions can intensify the search towards the optimal solution. In contrast to traditional crossover approaches, where offspring solutions are constructed from the parts of parent solutions, new crossovers define the degree of the destruction by preserving the parts that are common in both parent solutions, thus preserving the parts that have a higher probability to be optimally constructed than the other ones.

3. A genetic algorithm, based on the feasible reinsertion approach in genetic operators, on crossovers preserving common parts, and on second population in mutation operator, produces similar or better solutions than other genetic algorithms in short computation time. The usage of the second population in the mutation operator increases diversification in the population and overall efficiency of the genetic algorithm. Overall genetic algorithm is applicable to the rich vehicle routing problem.

Proposed solutions and contributions of the scientific novelty

- Operators of the genetic algorithm, that involves the destroy and reconstruct approach of large neighborhood search (LNS) usage in crossover and mutation operators, are proposed, where random insertion heuristic in genetic operators is used as a reconstruction method. Insertion heuristic is adjusted with evaluation of constraints to avoid generation in infeasible search space, thus speeding-up the computation. Random insertion heuristic preserves stochastic characteristics of the genetic algorithm thus involving the diversification in the population.
- New crossover operators that are based on the search of common parts in the parent solutions for generation of the offspring are proposed. In contrast to traditional crossover approaches, where offspring solutions are constructed from parts taken from the parents, the proposed crossovers identify and preserve parts of the solution that are common in both parents, thus intensifying a search towards the optimal solution. The usage of the longest common increasing sequence (LCIS) search in crossover operator preserves the sequence of elements from parent solutions, where the sequence is important characteristic of vehicle routing problem solutions.

- The genetic algorithm is proposed that involves insertion heuristic, feasibility preservation, a search of common parts in the crossover operators and the second population used in the mutation operator. Solutions obtained in the second population remain competitive in the main population: they have a higher probability to be selected for reproduction and involve the diversification in the population. The proposed algorithm produces solution in short time and solutions are better or equal to results obtained by other genetic algorithms. The advantage of a new developed genetic algorithm is that it can be applied to rich vehicle routing problem and the formulation of the rich vehicle routing problem is also defined in this research.

Approbation of the research

The main results of the thesis were presented at the following international conferences:

- 9th Conference on Databases and Information Systems, DB&IS 2010, July 5 - 7, 2010 – Riga, Latvia;
- 1st International Conference of EURO Working Group on Vehicle Routing and Logistic Optimization (VeRoLog 2012), June 18 - 20, 2012 – Bologna, Italy;
- 25th European Conference on Operation Research (EURO-2012), July 8 - 11, 2012 – Vilnius, Lithuania;
- 26th European Conference on Operation Research (EURO-2013), July 1 - 4, 2013 – Rome, Italy;
- 2nd International Conference of EURO Working Group on Vehicle Routing and Logistic Optimization (VeRoLog 2013), July 7 - 10, 2013 – Southampton, United Kingdom.

List of Publications

Articles in the reviewed scientific periodical publications:

G. Vaira, O. Kurasova. Parallel Bidirectional Dijkstra's Shortest Path Algorithm. *Databases and Information Systems VI*, Volume 224 of Frontiers in Artificial Intelligence and Applications, p. 422–435, IOS Press, 2011, ISSN 0922-6389 (print), ISSN 1879-8314 (online).

G. Vaira, O. Kurasova. Genetic algorithms and VRP: the behaviour of a crossover operator. *Baltic Journal of Modern Computing*, 1(3–4), p. 161–185, 2013, ISSN 2255-8942 (print), ISSN 2255-8950 (online).

G. Vaira, O. Kurasova. Genetic Algorithm for VRP with Constraints based on Feasible Insertion. *Informatica*, 25(1), p. 155–184, 2014, ISSN 0868-4952.

Articles in other editions:

G. Vaira, O. Kurasova. Modified bidirectional shortest path Dijkstra's algorithm based on the parallel computation. In *Proceedings of the 9th International Baltic Conference on Databases and Information Systems (Baltic DB&IS 2010) (J. Barzdins, M. Kirikova (eds.)),* p. 205-217 Riga: University of Latvia Press, 2010, ISBN 978-9984-45-199-2.

G. Vaira, O. Kurasova. Feasible Insertion Genetic Algorithm for VRP with Constraints. In *Proceedings of 2nd International conference of EURO Working Group on Vehicle Routing and Logistic Optimization (VeRoLog 2013)*, p. 96, 2013a, Southampton, United Kingdom.

Outline of the dissertation

The text of the thesis consists of introduction, 3 main chapters, conclusions and references. Each chapter is provided with the summary (except introduction and conclusions). The total scope of this thesis is 140 pages, 26 figures and 19 tables.

Introduction describes research context and motivation, presents the statement of the problem, discusses tasks and objectives of the research, methodology of research, presents practical significance of results, scientific novelty, defending propositions and approbation of obtained results.

Chapter 1 provides overview of vehicle routing problems and solutions, reviews genetic algorithms for solving vehicle routing problems in details.

Chapter 2 describes the proposed genetic algorithm for vehicle routing problem.

Chapter 3 provides experimental evaluation of the proposed algorithms.

Conclusions present the main conclusions of the thesis.

Chapter 1
Vehicle routing problem: a review

The chapter is organized as follows. **Section 1.1** describes a vehicle routing problem and constraints. **Section 1.2** reviews heuristic approaches for solving VRP. Main genetic algorithms principles are discussed in **Section 1.3**. **Section 1.4** analyzes genetic algorithm application for VRP and the common feasibility handling approaches. **Section 1.5** investigates usage of insertion heuristics in GA and crossover operators that deal with feasible solutions. **Section 1.6** reviews shortest path problem and Dijkstra's algorithm speed-up techniques. **Section 1.7** summarizes this chapter.

1.1. Vehicle routing problem

Vehicle routing problem (VRP) is a general name given for a class of problems, in which a set of vehicles service a set of customers. This statement was first defined by Dantzig and Ramser (1959). VRP is a generalization of a *traveling salesman problem* (TSP), where only one traveler is taken into account. The TSP is defined as a set of cities, where a single traveler needs to visit all of them and return to the starting city. The objective of the TSP is to find the shortest route.

The vehicle routing problem typically is described as a graph $G = (N, E)$ and a set of homogeneous vehicles $V = \{v_1, \ldots, v_t\}$, where t is the number of vehicles. The graph G consists of the nodes $N = \{n_0, n_1, \ldots, n_k\}$, where n_0 is a depot and $N \backslash \{n_0\}$ are k customers that need to be serviced, and edges $E = \{e_{ij}\}$, where $i \neq j$, $0 \leq i \leq k$, $0 \leq j \leq k$, $e_{ij} = (n_i, n_j)$. Each vehicle that services customers starts the travel from the depot and finishes it in the depot as well. The objective of the typical VRP is to find the solution, at first, minimizing the total vehicle number required, and secondly, minimizing the length of the total traveled path (Dantzig and Ramser, 1959; Jih et al., 1996; Potvin and Bengio, 1996; Tan et al., 2001; Jung and Moon, 2002; Ombuki et al., 2002; Jih and Hsu, 2004; Alvarenga et al., 2005; Ombuki et al., 2006; Yeun et al., 2008). For the set E, the cost matrix D is defined, where d_{ij} is the cost of the edge $e_{ij}=(n_i, n_j)$, and $d_{ii} = 0$. Usually the VRP is treated as symmetric, where $d_{ij} = d_{ji}$. In the real world problem, the cost matrix is asymmetric and needs to be calculated from geographic data by using the shortest path algorithms. Moreover, if a vehicle set is not homogeneous, some roads can be forbidden for certain vehicles and allowed for others. The different shortest path can exist for a different vehicle type, so a different matrix needs to be calculated for all the different vehicle types. A review of various speed-up techniques for the shortest path problem can be found in Section 1.6.

Various constraints can be added to the VRP. The defined constraints usually refer to real life situations. Let us define a single constraint $c \in C$, where C is a set of all constraints that should not be violated in the final solution.

The most known constraints for the VRP are capacity constraints and time window constraints. The capacity constraints $C_c \subseteq C$ are carriage limitations applied to each vehicle. A *capacitated vehicle routing problem* (CVRP) is usually defined with equal capacities for all vehicles. However, in real life vehicle fleet with different capacities can be used to solve the delivery problem.

Time window constraints $C_{tw} \subseteq C$ define time frames when a customer can be serviced. The problem dealing with time windows constraints is called *vehicle routing problem with time windows* (VRPTW). Single-sided and double-sided windows are specified in terms of time frames that are widely considered in literature. However, real life situations can give a multiple time frame representation, where a customer can be serviced in one of the defined time frames:

- $[t_1, \infty)$ defines a time frame constraint when a vehicle has to arrive no earlier than the time t_1. If a vehicle comes too early, it has to wait until time t_1.
- $[0, t_2]$ defines a time frame constraint when a vehicle has to arrive to a customer no later than the time t_2.
- $[t_1, t_2]$ defines a double sided time frame constraint, where $t_1 \leq t_2$. The constraint includes the limitation from both previously defined constraints.
- $[0, t_1] \cup [t_2, t_3] \cup [t_4, \infty)$ defines multiple time frames, where $t_{i-1} \leq t_i$. The multiple time frame constraints can include any of previously defined time window constraint. However, the single constraint from the group needs to be satisfied.

The time window constraint can be added to the depot node to define the overall traveling time limit for a single vehicle. The maximum number of vehicles can be treated as an additional constraint $c_v \in C$, where c_v defines the limit of vehicles in the solution.

Real situations can give another type of constraints where goods need not only to be brought from a depot to a customer, but also to be picked up from a number of customers and brought back to depot or to any other customer. This problem is known as a *vehicle routing problem with pick-up and deliveries* (VRPPD). The set $C_{pd} \subseteq C$ defines pick-up and delivery constraints within the problem, where each $c \in C_{pd}$ is a constraint that defines the delivery of a certain amount of goods from the starting node n_s to the target node n_t. In Figure 1, the filled circle represents a depot, the empty circles represent customers, the dotted lines represent possible pick-up and delivery constraints, and the solid lines represent a possible routing solution for two vehicles. Combination of VRPPD and VRPTW is called *pick-up and delivery problem with time windows* (PDPTW) (Li and Lim, 2003; Ropke and Pisinger, 2006).

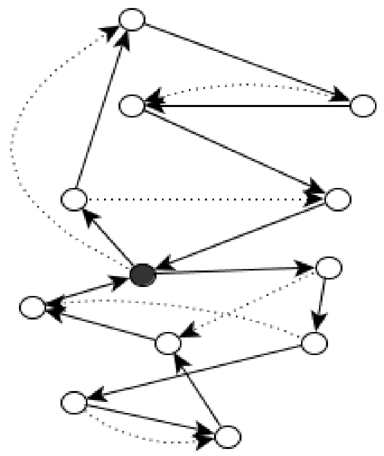

Fig. 1. Pick-up and delivery problem

Particular mathematical formulations can be found for each various VRP extensions, where in each formulation the constraints evaluation is included in the objective function (Yeun et al., 2008). In this thesis we define it in general way. Let us define the function $F_c(x)$ that evaluates violation of constraints in the solution x and $f_c(x)$ that evaluates violation of the single constraint $c \in C$:

$$F_c(x) = \sum_{c \in C} f_c(x)$$

$$f_c(x) = \begin{cases} 0 & \text{if constraint is satisfied} \\ z, \text{ where } z \in \mathbb{R}, z > 0 & \text{otherwise} \end{cases}$$

The objective of the traditional VRP is to find a solution x that satisfies the equation $F_c(x) = 0$ and minimizes the functions $f_v(x)$ and $f_d(x)$ in the defined order, where $f_v(x)$ evaluates the vehicle number in the solution and $f_d(x)$ identifies the total travel path:

$$f_v(x) = |\{r_1, \ldots, r_q\}| = q$$

$$f_d(x) = \sum_{j=1}^{q} d_l(r_j)$$

$$r_j = (n_{j_1}, \ldots, n_{j_m}), n_{j_i} \neq n_0, \; r_j - \text{single route with } m \text{ nodes}$$

$$x = \{r_1, \ldots, r_q\}, q \leq c_v$$

$$N_j = \{\forall n_{j_i} \in r_j\}, \quad |N_j| = m$$

$$N_1 \cap \ldots \cap N_q = \varnothing, \{n_0\} \cup N_1 \cup \ldots \cup N_q = N$$

$$d_l(r_j) = l(n_0, n_{j_1}) + \sum_{i=2}^{m} l(n_{j_{i-1}}, n_{j_i}) + l(n_{j_m}, n_0)$$

$$l(n_{j_{i-1}}, n_{j_i}) - \text{distance between nodes}$$

Beside VRP with mentioned constraints, other VRP extensions are analyzed in the literature:

- *Multiple depot VRP* (MDVRP). In this problem multiple depots exist from where vehicles could start traveling and where they could end up (Cordeau et al., 2001). In *open VRP* (OVRP) vehicles do not require to return to the depot (Brandão, 2004).
- *Split delivery VRP* (SDVRP). Customer can be serviced by more than one vehicle, if it reduces overall cost (Archetti et al., 2006; Archetti and Speranza, 2012). Such situations are also investigated in *multiple commodities VRP* where different types of vehicle need to be used for delivering different types of goods to the customer (Archetti and Speranza, 2012).
- *VRP with satellite facilities* (VRPSF). Additional satellite facilities exist in the graph where vehicles can be replenished with goods instead of coming to the depot (Bard et al., 1998). In *two-echelon VRP* (2E-VRP) two routing levels are defined, where the first level addresses depot-to-satellite delivery and satellite-to-customer delivery is addressed in the second level (Crainic et al., 2010).

- *Periodic VRP* (PVRP). In periodic VRP the horizon of defined number of days is given and visiting frequency within defined time range is defined for each customer. Solution of such problem is a set of routes for each day that satisfy all the frequency and delivery constraints within all days (Cordeau et al., 2001; Baptista et al., 2002)
- *VRP with backhauls* (VRPB). In this problem some customers require deliveries (linehauls) and some of them can return commodities back to the depot (backhauls). Backhauls typically go after linehauls. Such problem is treated as a special case of VRPPD problem (Ropke and Pisinger, 2006).
- *VRP with time deadlines* (VRPTD). Such problem is similar to VRPTW except that there is not lower bound for time, thus not requiring to pay attention to wait times (Thangiah et al., 1993). In *VRP with soft time windows* (VRPSTW) service is allowed after time window but with additional penalty cost (Toth and Vigo, 2002). Another VRP that deals with time is called *time-dependent VRP* (TDVRP), where the time of day and specific events related to the real-world situations (i.e. rush hours) are included in the problem (Ichoua et al., 2003).
- *Heterogeneous fleet VRP* (HVRP). This problem includes non-homogeneous fleet where different vehicles include different characteristics (Gendreau et al., 1999).
- *Green VRP* (G-VRP). In this problem vehicles that are used in the delivery are powered with alternative fuel. Typically that are electrical vehicles. Such vehicles require often refuel due to fuel capacity limitations. Limitations arise in the problem because of time required for refueling and availability of the refueling stations (Erdogan and Miller-Hooks, 2012). In paper (Schneider et al., 2012) the problem is called *electric VRP* (EVRP).

- In a *dial-a-ride problem* (DARP) a transportation of users is considered, where desired departure or arrival time and maximum transportation duration is defined for users (Cordeau and Laporte, 2003).

There are a number of other VRP extensions that differ depending on included data, constraints and objectives. Various combinations of mentioned VRP exist to specific real-world problems (i.e. delivery of goods, waste collection, blood collection and delivery, post delivery, etc.). There is also research that combines VRP with freight loading problem, i.e. in paper (Iori et al., 2007) two-dimensional rectangular loading surface is considered, where constraint is defined for sequential loading and unloading (2L-CVRP).

The *generalized VRP* (GVRP) defines an extension of VRP, where a set of customers is partitioned into clusters and the limitation is to visit single cluster only once (Bektas et al., 2011). There are attempts to describe *rich vehicle routing problem* (RVRP) that include most of the mentioned constraints and situations and also other real-world constraints and situations (Toth and Vigo, 2002; Hartl et al., 2006; Hasle and Kloster, 2007; Rizzoli et al., 2007; Pisinger and Ropke, 2009). Typically it is descriptive formulations or summarized real-world constraints (Drexl, 2012).

1.2. Heuristics for VRP

The vehicle routing problem has got much attention in recent years. Due to usefulness in real life and innovation in the transportation sector as well as logistics, VRP continues to draw researchers' attention. A number of different exact and heuristic methods have been studied to solve the VRP that is known to be NP-hard. Although exact methods give the optimal solution, their computation time considerably increases with the increasing size of the problem.

Branch and bound (B&B). Branch and bound is an optimization technique which search of all possible solutions while discarding (pruning) a large number of non-promising solutions by estimating upper and lower bounds of the quantity to be optimized. Depth first strategy is used to search the tree, where nodes whose objective value are lower/higher than the current best are not explored. Algorithm requires branching operator for splitting solutions set into the smaller ones and bounding operator for computing lower/ higher bound for the objective function to be be optimized. *Branch and cut* (B&C) is a B&B technique, where search space is reduced by adding new constraints (cuts). Branch and bound algorithm is suitable to solve VRP of small instances with only few nodes (Toth and Vigo, 2001; Toth and Vigo, 2002; Lysgaard et al., 2004; Yeun et al., 2008; Bektas et al., 2011; Vidal et al., 2013).

Constructive heuristics are methods that start from the empty solution and iteratively extend it until the full solution is constructed. Construction heuristics that are typically used for solving VRP are as follows:

- *Savings algorithm*;
- *Route-first cluster-second*;
- *Cluster-first route-second*;
- *Insertion heuristics*;

Savings algorithm. One of the constructive heuristics is *savings algorithm* proposed by Clark and Wright (1964) (Laporte et al., 1999; Cordeau et al., 2005; Vidal et al., 2013). Algorithm starts with the initial solution where all nodes are visited by separate route from depot. The algorithms search and merge two routes by maximizing the saving cost, where cost typically is a distance. Merge is possible, if merged route remains feasible.

Route-first cluster-second. The construction starts from the initial route that visits all the nodes. The route is then split into several routes starting from the depot (Laporte et al., 1999; Vidal et al., 2013).

Cluster-first route-second. In contrast to “route-first cluster-second” approach, the nodes are firstly added to clusters and then routes are optimized in each cluster. The clusters are created by solving *generalized assignment problem* (GAP). *Sweep* algorithm is proposed by Gillet and Miller (1974) (Laporte et al., 1999; Vidal et al., 2013). Algorithm inserts new nodes to route by going circularly around the depot in each step increasing the angle. Nodes are inserted at the end of the route, if insertion is feasible and if no insertion found, new route is started. Afterwards each route is optimized.

Insertion heuristics. Insertion heuristics are popular methods for solving a variety of vehicle routing and scheduling problems. Insertion heuristics were first introduced for a traveling salesman problem (TSP) and belong to a group of route construction algorithms (Rosenkrantz et al., 1977; Campbell and Savelsbergh, 2004). The main principle of insertion heuristics is to start from a single node that is usually called a seed node and that forms the initial route from the depot. Other nodes are inserted one by one evaluating certain functions to select a node and the place in the route for insertion. The well-known insertion heuristic approaches used in TSP are categorized by the methods used for the node selection to be inserted: random insertion, the nearest insertion, the farthest insertion and the cheapest insertion. For the farthest and the nearest insertion each next node is selected for insertion according to the distance to the already constructed route where the functions for maximization and minimization are defined respectively. The node is inserted by evaluating the cost function $c(n_i, n_k, n_j) = l(n_i, n_k) + l(n_k, n_j) - l(n_i, n_j)$, where n_i, n_j are the nodes in the current constructed route, n_k is the node to be inserted, and $l(n_i, n_j)$ is the distance function. In the random insertion heuristic a node is randomly selected from a set of nodes that are still not included in any route. The place in the route where a randomly selected node has to be inserted is determined by minimizing the same cost function

$c(n_i, n_k, n_j)$. In contrast to the random insertion heuristic, the cheapest insertion heuristic selects the node for insertion by minimizing the defined function for all nodes and all places in the route (Solomon, 1987).

Solomon (1987) has proposed three types of insertion heuristics. The most successful of them is called I1. The first route is initialized with the seed node which is the farthest one from the depot. Nodes are inserted into the first route until reaching the limit of capacity constraints. If still there are unrouted nodes, a new route is created and the insertion process is repeated until all the nodes are inserted. Two subsequently defined criteria $C_1(n_i, n_u, n_j)$ and $C_2(n_i, n_u, n_j)$ are used to select the node n_u for insertion between the nodes n_i and n_j. The first function determines detour and delay values. The second function generalizes a regret measure over all routes to estimate what could be lost later if the node is not immediately inserted in its best place. The criterion function C_1 depends on the coefficients $(\alpha_1, \alpha_2, \mu)$ and the overall insertion method efficiency depends on them (Potvin and Dubé, 1994). In (Potvin and Rousseau, 1993), the authors have proposed a parallel version of insertion heuristic I1.

Local-improvement heuristics. The main principle of the l*ocal search* (LS) improvement heuristics is as follows. For solution $x \in S$ (where S is a search space), a neighborhood in the search space can be defined as $Nh(x) \subseteq S$, where $Nh(x)$ is a function that maps the solution x to a set of solutions by applying the defined moves (perturbations). Local search is an iterative process that takes the initial solution x and, in each iteration, searches for the improved solution x' in the neighborhood of x. The search stops at solution x'' when the improved solution is not found in neighborhood $Nh(x'')$. Such a search approach finds a local optimum and is called *Hill Climbing* (HC). It is a popular method used in other algorithms for improvement of solutions. An example of neighborhood *Nh*(*x*) can be one of the following local-improvement approaches (Laporte et al., 1999; Vidal et al., 2013):

- *2-opt*, where the 2-opt neighborhood is a set of solutions that can be obtained by removing two edges in solution *x* and adding new ones to reconnect the route. 3-opt generalizes 2-opt neighborhood and here 3 edges are replaced by new ones. In λ-opt, where λ edges are replaced by new ones, two previous improvements are generalized.
- *Lin-Kernighan*. In the algorithm proposed by Lin and Kernighan (1973), λ value is changed during search (Laporte et al., 1999; Helsgaun, 2000; Vidal et al., 2013).
- *Shift* (also called *re-locate*). In *shift* neighborhood, one node is moved from one route to another. In *swap* (also called *exchange*) neighborhood two nodes are exchanged between routes.

In addition to the mentioned local-improvement methods other local search methods exist: Or-opt, cross, etc. (Laporte et al., 1999; Cordeau et al., 2005; Vidal et al., 2013).

Local searches and heuristic approaches often produce a near optimal solution within a reasonable computation time. These methods may be sensitive to data sets given or require additional training data during the learning process.

Metaheuristic is another approach for solving a complex problem that may be too difficult or time-consuming by traditional techniques. Some of the metaheuristics that are applied to the VRP are following:

- *Simulated annealing* (SA). Simulated annealing approach mimics the annealing process in metallurgy. In order to escape the local optimum, the probability of accepting deteriorated move for the solution depends on the so called “temperature”. The higher temperature, the higher probability to accept degraded solution. Temperature parameter is evolved during the search, thus imitating the cooling process in metallurgy (Černý, 1985; Misevičius, 2003; Cordeau et al., 2005; Vidal et al., 2013).

- *Tabu search* (TS). The idea of the *tabu search* is to prevent a move in the search that was already performed during specified amount of last iterations. In such approach restrictions are stored in memory so called *tabu list*. Application of *tabu search* prevents cycling in search and allows moving the search to unexplored search space. (Cordeau et al., 2001; Brandão, 2004; Archetti et al., 2006; Yeun et al., 2008; Vidal et al., 2013).
- *Ant colony optimization* (ACO). This approach is inspired by the behavior of the ants. In the nature initially each ant wanders randomly and when the food is found, the ant returns to the colony by laying down pheromone trails. When other ants find the path with pheromone trails they choose to go by that path with higher probability comparing to go randomly. By the time pheromone trails evaporate, so longer paths will evaporate more than shorter ones because of time needed to travel down the path and back again. Evaporation technique of the pheromone trails lead to optimization of the path length (Rizzoli et al., 2007; Yeun et al., 2008; Jančauskas et al., 2012; Vidal et al., 2013).
- *Large neighborhood search*. The large neighborhood search (LNS) heuristic belongs to the class of heuristics known as a *very large scale neighborhood search* (VLSN) (Pisinger and Ropke, 2009; Vidal et al., 2013). In the large neighborhood search, the neighborhood is defined as $Nh(x) = r(d(x))$, where neighborhood solutions can be found by applying, at first, the destroy function $d(.)$ and then the reconstruction function $r(.)$. The large neighborhood search maintains two solutions: the best solution found x^b and the current solution x is used that takes

part in the exploration of the neighborhood. If x is found such that $f_a(x) < f_a(x^b)$, where $f_a(x)$ is the acceptance criteria function, x^b is replaced with a new solution: $x^b = x$. In *adaptive large neighborhood search* (ALNS) the neighborhoods are applied depending on their performance in previous iterations (Pisinger and Ropke, 2009).

- *Genetic algorithm.* It is a population based algorithm that follows the idea of biological evolution and natural selection where the fittest individuals survive. The genetic algorithm is described in details in Section 1.3.

There are also other approaches beside mentioned ones that are designed to solve one or another specific VRP. In (Drexl, 2012) the author explains the gap between models analyzed in theory and the practical applications of the algorithms. In literature we can find researches on algorithms for RVPR, i.e. in (Pisinger and Ropke, 2009) ALNS is proposed for solving general vehicle routing problem. Hasle and Kloster (2007) have proposed the approach for solving RVRP is based on regret insertion and *variable neighborhood descent* (VND) approach. VND belongs to VLSN algorithm group and is similar to already mentioned ALNS approach. Difference is that VND switches to another neighborhood only when search of the current neighborhood is trapped in the local optimum (Pisinger and Ropke, 2009). In (Rizzoli et al., 2007) ant colony optimization algorithm is proposed for solving real-world vehicle routing problem.

In literature various hybrids of previously mentioned algorithms can be found. There are also approaches to design parallel algorithms for solving vehicle routing problems. A survey of different approaches for solving various VRP can be found in (Yeun et al., 2008; Vidal et al., 2013). In (Dzemyda and Sakalauskas, 2011) we can find a survey of heuristic methods for solving problems that are known to be NP-hard.

Various global optimization algorithms are also investigated by researchers from Lithuania: J. Mockus, A. Žilinskas, J. Žilinskas, G. Dzemyda, L. Sakalauskas, A. Misevičius. It is worth to mention the researches that involve investigations of the genetic algorithms: Misevičius and Kilda (2005); Žilinskas and Žilinskas (2007); Misevičius (2009), Žilinskas (2008), Redondo et al. (2012), Lančinskas et al. (2013). There are also doctoral dissertations prepared: Felinskas (2007) investigated different heuristic methods, including genetic algorithms, for optimization of resource-constrained project schedules; Šešok (2008) investigated the usage of genetic algorithm for optimization of topology of truss structures, where additional improvement step in genetic algorithm is proposed to use to find better solutions; Lančinskas (2013) investigated parallelization of random search global optimization algorithms, where strategies are proposed for modification and parallelization of *non-dominated sorting genetic algorithm* (NSGA); GA with distribution strategy has been suggested and investigated by Mačiūnas (2013) for the optimization of mechanical properties of grillages;

In this research the focus is given only to genetic algorithm approaches for solving general vehicle routing problem. Genetic algorithms have been successfully applied to solve many combinatorial problems as well as to the VRP. The standard genetic algorithm has limitations in the constrained environment. However, it is able to incorporate other techniques within its framework to produce a hybrid that provides better efficiency (Yeun et al., 2008).

1.3. Genetic algorithm

Genetic algorithms are based on ideas of evolution theory (Holland, 1975). The main principle here is that only the fittest entities survive (Reid, 2000; Jung and Moon, 2002; Lukasiewycz et al., 2008a). A genetic algorithm can be divided into several sub-parts that are used in this algorithm:

representation, fitness function evaluation, initialization, selection, recombination (crossover and mutation), termination. The whole process of genetic algorithm is described in Figure 2.

1. The initial population is created, where each individual is expressed via defined representation;
2. The fitness function is evaluated for the initial population;
3. The subset of the population (so-called parents) is selected that will be used in recombination operators to generate offspring;
4. The crossover operator is applied to parents to create new offspring;
5. The mutation operator is applied with a certain probability;
6. The fitness function is evaluated and the individuals with the worst fitness value are removed;
7. If the stopping criterion is not met, go to Step 3.

Fig. 2. Steps of a genetic algorithm

Representation. The classical genetic algorithm paradigm deals with the solutions encoded as a literal string, called chromosomes. A chromosome is the representation of a single solution of the problem and requires additional encoding/decoding steps to be defined in the algorithm. Genetic algorithm approaches can be divided into two sets: algorithms that are applied to the VRP represented as a chromosome, and algorithms that skip the encoding/decoding step. In genetic algorithms, where encoding/decoding is bypassed, a single solution is usually called individual. The TSP problem has a single constraint – all cities should be visited. The solutions of the TSP problem are vectors of the nodes, where each solution starts always from the same node and the direction is not important:

$$x_{tsp} = (n_{i_1}, n_{i_2}, \ldots, n_{i_k})$$

A single solution within a problem can be defined as $x_{tsp} \in S_{tsp}$, where S_{tsp} is the whole search space of the TSP and $|S_{tsp}| = (k - 1)!/2$. Each TSP solution can be easily encoded as a queue of indexes (chromosome):

$$i_1, i_2, \ldots, i_k$$

Such encoding can be useful for any problem that can be expressed as TSP, for example in computer wiring, scheduling of jobs on a single machine, etc. (Deep and Adane, 2011). However, it is worth mentioning that such a representation does not hold any additional information.

Population and initialization. In initialization, the initial set of chromosomes, also called as the initial population, is created. The size of initial population is important for the overall genetic algorithm. A small size of the initial population can lead to finding of a local optimum only, while a larger initial population gives a higher probability that the global optimum will be found, however, the computation time increases (Reid, 2000). While the TSP is defined as a complete graph, usually the initialization is done by randomly selecting a node and assigning it to the route.

Evaluation and selection for reproduction. The selection operator is used to identify chromosomes which will be used in reproduction and will survive in the next generation. Different techniques can be used in selection operators, however, usually a natural selection process is simulated, where the "strongest" individuals are used in reproduction. One of the method for selection is called *roulette wheel*. The name explains the method: a wheel is divided into parts according to the fitness of the individuals in the population, where better individuals get a larger part of the wheel and the worst individuals get a small part of the wheel. So, the probability to be selected is directly proportional to the fitness value. When the wheel is spinning, a pin on the wheel will most probably point to a better individual. The individuals with a higher fitness value have a higher probability to be selected for reproduction, and vice versa (Golberg and Deb, 1991; Zhong et al., 2005).

The second method for selection is called *ranking*. In the *ranking* method, all individuals in population are sorted according to the fitness value $f(x)$ to assign ranks, where the individual with a better fitness value gets a higher rank. In TSP, usually $f(x)$ defines the length of the total path traveled, however, this function can include additional characteristics and measurements in order to keep individuals in the population. If in the *roulette wheel* method the fitness value is used when assigning a probability to be selected, in the *ranking* method individuals are selected proportionally to the rank (Golberg and Deb, 1991; Zhong et al., 2005)

Another method for selection is called *tournament selection*. This method uses characteristics from the *ranking* method, but, in contrast to it, *tournament selection* ranks only a subgroup of individuals. At first, two subgroups from a population are selected. Each subgroup must contain at least two individuals. The individuals are ranked within a group like in the *ranking* selection operator. The best individual from each group is selected for reproduction, and the worst individuals are chosen to leave the population. To generate l new offsprings in each iteration, assuming that two new offsprings will be generated from two selected parents, l subgroups have to be selected from the population (Golberg and Deb, 1991; Alvarenga et al., 2005; Zhong et al., 2005).

Recombination. An important part of the genetic algorithm is recombination operators. The crossover operator simulates the reproduction between two individuals, where the created offsprings inherit some characteristics from parent individuals. Many crossover and mutation operators exist that operate with a chromosome encoded as a literal line of symbols or numbers. The list of common crossovers used for solving TSP as well as for solving VRP is as follows (Blanton and Wainwright, 1993; Jih et al., 1996; Ombuki et al., 2002; Tan et al., 2006; Kumar et al., 2012):

- Partially matched crossover (PMX),
- Cycle crossover (CX),
- Ordered Crossover (OX),
- Uniform Crossover (UX),
- Uniform Order Crossover (UOX),
- Edge Assembly Crossover (EAX),
- Merge crossovers (MX1, MX2), etc.

Crossovers listed above produce an encoded chromosome or chromosomes as a result that need to be decoded for evaluation.

Binary string:	0 1 1 0 1 1 0 0
1st parent:	1 2 3 4 5 6 7 8
2nd parent:	3 5 1 8 4 7 2 6
Intermediate offsprings	
1st offspring:	– 5 1 – 4 7 – –
2nd offspring:	1 – – 4 – – 7 8
Generated offsprings:	
1st offspring	2 5 1 3 4 7 6 8
2nd offspring	1 3 5 4 2 6 7 8

Fig. 3. Uniform order crossover (UOX)

In (Jih et al., 1996) we can find a review, where the Uniform Order Crossover is mentioned as a good approach for solving VRP. It is an analogue of the Uniform Crossover translated into an order-based form:

1) a binary string of the same length as parent chromosomes is generated;
2) the first intermediate offspring preserves nodes from the second parent, where the generated string contains "1";

3) permute nodes from the second parent where a binary string contains "0" in the same order as they appear in the first parent;
4) fill these permuted elements in the gaps of the first intermediate offspring;
5) switch the parents and perform the steps 2 – 4 to create the second offspring (Jih et al., 1996).

Figure 3 provides an example of a uniform order crossover.

A mutation operator is used with intention to prevent getting stuck in the local optimum and increase a probability to find the global optimum (Hong et al., 2002). In the mutation operator, a new offspring is created from the single solution by changing some characteristics within it. In the genetic algorithm, crossover and mutation operators are applied by a predefined probability. We can find the values for these probabilities proposed in (Srinivas and Patnaik, 1994; Hong et al., 2002). In the adaptive probability approach, the probabilities are adjusted during computation depending on the current population characteristics, flow of the computation and other parameters. In (Zhang et al., 2004, 2007), a fuzzy logic is considered for adjusting the probabilities. The simplest mutation operator extracts a single gene (an element of the chromosome) and places it back to the chromosome by randomly choosing a new location (Potvin and Bengio, 1996). Initially the evolutionary algorithms had only selection and mutation, while the genetic algorithms also utilize the crossover operator (Reid, 2000). Both operators play an important role in genetic algorithms due to the success of recombination of the existing solutions into a new one.

Diversity maintenance and selective pressure. Two important factors of the genetic algorithm are a population diversity and a selective pressure. These two factors are related: if the selective pressure is increasing, the population diversity decreases and vice versa. The selective pressure is a task of the selection operator. Too-weak selective pressure can lead to ineffective search. The selection operator as well as other operators influences overall diversity of the population. A good performance is achieved, while maintaining the diversity of the population as long as possible. The mutation is important in the variation of individuals, when the population becomes homogeneous (Srinivas and Patnaik, 1994). The population diversity can also be maintained by increasing the size of the population or by having greater mutation rates, however, the performance factor should be taken into account. Other techniques are also used. A common approach is to avoid duplicates in the population. It means that the generated offspring is not allowed in the population, if it is the clone of the existing individual.

Termination. The genetic algorithms are stochastic methods that could run forever, if a termination criterion is not applied. Simple stopping criteria are the maximum computation time, the maximum iteration number or iterations that are counted from the last successful improvement of the best individual (Reid, 2000; Hong, 2002; Jung and Moon, 2002; Berger and Barkaoui, 2004; Yeniay, 2005). The probability to improve the best individual decreases proportionally to the computation time. So, the number of iterations without improvement is directly proportional to the probability of improvement. A large value would increase the computation time and possibly a better solution will be found, while a low value will involve an early stop with a poor solution found.

Many derived GA approaches can be found in the literature, some of which include multiple populations, dynamically chosen genetic operators or any hybrids with other known heuristic approaches (Yeun et al., 2008). However, the main principles of the genetic algorithm remains the same. In the rest of the thesis, we will investigate a genetic algorithm implementation for

VRP. The main research presented here is based on recombination operators used for solving VRP and their influence on the whole genetic algorithm.

1.4. Genetic algorithms and VRP

As already mentioned, VRP is a generalization of the TSP problem. VRP includes additional components, i.e. fleet of vehicles, and additional constraints. An additional component of the problem can affect computation and even require to design the problem specific genetic operators. Genetic algorithm approaches to solve the VRP can be categorized according to the following features:

- *Representation*. Solution in GA can be encoded as a chromosome (expressed as a literal string), or unencoded, where encoding of the solution within chromosome is not addressed.
- *Feasibility handling*. Genetic algorithm operators can be designed to preserve the feasibility of individuals within a population or allow the generation of infeasible individuals.

An example of VRP solution, where 3 routes are used to service customers expressed as a chromosome is as follows (Berger et al., 1998), where $\forall n_e$ belongs to one route, $\forall n_f$ belongs to the second route and $\forall n_g$ belongs to the third route:

$$|\, n_{e1}\, n_{e2} \ldots |\, n_{f1}\, n_{f2} \ldots |\, n_{g1}\, n_{g2} \ldots |$$

The standard genetic operators can be applied to such a chromosome, however, such a representation does not hold any problem specific information and, depending on the encoding approach, the selected genetic algorithm can be ineffective. Different approaches for encoding the VRP solution can be found in the literature, i.e. in (Thangiah et al., 1991), a chromosome representation based on the angles of vectors starting from a depot node is proposed, where the VRP is treated as a planar graph problem (Thangiah et al., 1991; Jung and Moon, 2002). Researches can be found that compare crossover operators designed to work with the chromosome representation (Jih et al., 1996; Misevičius and Kilda, 2005; Kumar et al., 2012).

When dealing with constraints, a stochastic approach to find optimal solutions can compute very long, until an acceptable solution has been found (Reid, 2000). For a constrained problem, there exist feasible and infeasible search spaces S_F ($\forall x \in S_F$ does not violate any of the defined constraints) and S_U ($\forall x \in S_U$ does violate at least one defined constraint). Let us define the whole search space S, then $S_F \subseteq S$, $S_U \subseteq S$, $S_U \cup S_F = S$, $S_U \cap S_F = \varnothing$. The solution x belongs to the feasible search space S_F, if $F_c(x) = 0$. Highly constrained problems are those, where the feasible search space is very small. Thus the probability to generate solutions in such a space for crossover and mutation operators can be adequately small (Reid, 2000). Approaches, where a solution is represented as a chromosome or where solutions are allowed to be generated in the infeasible search space S_U, require additional approaches for constraint handling. The following approaches are used to deal with the infeasibility in genetic algorithms:

- Applying *penalty* function.
- Treating problem as *multi-objective*.
- *Repairing* solution.
- *Preserving feasibility* in the genetic operators.

Penalty. The penalty function $p(x)$ transforms a constrained problem into an unconstrained one (Reid, 2000; Yeniay, 2005; Lukasiewycz et al., 2008; Lukasiewycz et al., 2008a). A penalty method is widely used in genetic algorithms for constrained problems. The main target is to add a significant value to the fitness value for the generated offsprings that violate constraints. In (Michalewicz, 1995) the author discusses the advantages and disadvantages of having feasible and infeasible solutions in genetic algorithms and how they influence the results. The discussion is carried out on the issue how the feasible and infeasible solutions can be compared. In general, two evaluation functions $f_f(x)$, where $x \in S_F$, and $f_u(x)$, where $x \in S_U$, are considered. Different evaluation functions $f_f(x)$ and $f_u(x)$ are defined because of the ability to compare the solutions in two distinct search spaces. However, the relation between these two functions can be designed via the extended function $q(\mathrm{x})$, where $q(x)$ can be either the penalty function or the cost for repairing the solution (Michalewicz, 1995; Yeniay, 2005). There are two main ways of penalty function application (Yeniay, 2005):

- additive: $f_u(x) = f_f(x) + q(x)$, where $q(x) = 0$, if none of the constraints is violated, and $q(x)>0$, otherwise.
- multiplicative: $f_u(x) = f_f(x)q(x)$, where $q(x) = 1$, if none of the constraints is violated, and $q(x)>1$, otherwise.

The penalty method is directly applied to the fitness value, where the highest benefit of the penalty function is to adjust the ranking mechanism in the population and increase the selective pressure on the feasible individuals. Good results are reported, when the penalty function is designed so that feasible results are always treated better than infeasible results (Michalewicz, 1995). Various penalty functions are considered on the basis of their application characteristics (Yeniay, 2005). Some of them can dramatically change the fitness value or completely remove from a population list. The death penalty has the penalty function $q(x) = +\infty$ for each $x \in S_U$. Although the death penalty will help to avoid having infeasible solutions, it is expected to work well when the feasible search space is a reasonable part of the whole search space (Michalewicz, 1995). However, for highly constrained problems the algorithm can suffer a degradation when trying to search for feasible solutions and if the feasible solution is found, the search may prevent to find a better one (Yeniay, 2005). Adaptive penalties update the parameters for each generation according to information gathered from the population. Although penalty functions help to identify infeasible solutions and keep individuals with the best characteristics in the population, they affect the generation of feasible solutions only indirectly and still allow the generation of infeasible solutions. It is the waste of computation time when infeasible solutions are generated and later eliminated (Reid, 2000).

Multi-objective. In a multi-objective approach, the constrained problem is transformed into a multi-objective problem. In (Berger and Barkaoui, 2004; Ombuki et al., 2006; Tan et al., 2006; Garcia-Najera and Bullinaria, 2011), the Pareto ranking method is used to solve the VRPTW expressed as multi-objective, where Pareto ranking, similarly to the penalty approach, is used to adjust the ranking mechanism of the genetic algorithm and assign the relative strength of individuals in the population. The ranking mechanism assigns the smallest rank to non-dominated individuals and the dominated individuals are ranked according to the individuals in the population and the defined criteria. Pareto ranking attempts to assign a single fitness score to the solution of a multi-objective problem. In literature there can be found Pareto ranking in the genetic algorithm treated as equivalent to the penalty approach (Michalewicz, 1995a).

Repair. The second approach for feasibility handling is a repair method. The repair method defines the transition function $y = r(x)$, where y is the repaired version of x, such as $y \in S_F$, and $x \in S_U$. The repair can be designed in two different ways:

- An individual is repaired for evaluation only, where $f_u(x) = f_f(y)$, and y is a repaired (i.e. feasible) version of x. It is the so-called *Lamarckian* approach (Michalewicz, 1995; Zhu, 2003; El-Mihoub et al., 2006). The weakness of such an approach is that it depends on the problem and a specific repair algorithm has to be designed (El-Mihoub et al., 2006).

- An individual is repaired and the previous individual is replaced by its repaired version. It is called a *Baldwinian* approach (Michalewicz, 1995; Zhu, 2003; El-Mihoub et al., 2006). This method has the same limitation as the previous one. The question of replacement is also widely considered. In some researches the fixed percent of the repaired individuals replace the previous one or this can be dependent on the problem or even on the evolution process.

In (Jung and Moon, 2002) the authors have proposed to use 2D chromosomes for VRP encoding to handle additionally the position of nodes in the 2D Euclidean space. The described crossover operator uses a 2D partitioning to interchange routes between two chromosomes, where each route represents the traveling path of a single vehicle. However, the repair algorithm is considered to connect separate fragments of the route by taking into account additional decision variables. Repair algorithms are very helpful for solving a single-constrained problems. However, identification of the parts for solution improvement can be quite complex because of constraints. A problem can arise when the improvement of one objective can lead to a degradation of others.

Preserving feasibility. The author in (Reid, 2000) discusses the possibility of having feasible solutions generated in crossover and mutation operations, where feasibility handling in a two-point crossover where a set of crossovers with different boundary indices is considered. Probability function is defined to find a feasible crossover for a linearly constrained optimization problem. However, for a highly constrained problem where a feasible space is very small as compared to the full search space, only a half-feasible crossover with a single boundary point is discussed. In order to handle feasibility in the mutation process, the proposed mutation operator is based on the crossover operator, where the selected individual is crossed with a randomly generated individual (Reid, 2000).

In (Tan et al., 2006) the authors use individuals that are composed of a set of routes, where each route contains a list of customers. A crossover is defined to exchange the routes between individuals. If the newly added route contains the customer that has already been visited in another route, the customer is removed from the previous one and left in a newly added route (Tan et al., 2006). If individuals selected for crossover are feasible, the offspring, generated from parent individuals, will remain feasible. However, a set of transitions is proposed for feasibility handling in the mutation operator, where constraint violation is evaluated after each transition. If mutation transitions generate an infeasible solution, the original routes are restored (Tan et al., 2006). Such approach does not help to generate feasible solutions, but it helps to avoid infeasibility.

In (Alvarenga et al., 2005) the authors have proposed a crossover where feasible routes from the parent individuals are inserted in the offspring. At first the routes with the maximum number of customers are inserted. After all feasible routes have been inserted in the offspring, the insertion of the remaining customers is tested in the existing routes. If some customers are still not included to any route, a new route is created and a stochastic push-forward insertion heuristic is used to insert customers (Alvarenga et al., 2005).

Other approaches. In literature we can find approaches of using genetic algorithms in a two-phase approach, where in the first phase genetic algorithms are used to solve a single objective and in the second phase different algorithms are used to continue the optimization process (Berger and Barkaoui, 2004; Alvarenga et al., 2005; Ombuki et al., 2006). The fluctuating population size is also considered to keep infeasible solutions in the solution set. It is proposed because some parts of infeasible solutions can sill remain significant for crossover and mutation operators (Reid, 2000). In (Alvarenga et al., 2005) the authors have proposed to use 10 hierarchical criteria to rank individuals in the population.

The authors in (Berger and Barkaoui, 2004) have proposed parallel two-population co-evolution genetic algorithms, Pop1 and Pop2, for VRPTW. The first population, Pop1, has the objective to minimize the travel distance to the fixed number of vehicles. On the other hand, Pop2 works to minimize the violated time window in order to find at least one feasible individual. In Pop2 the vehicle number is limited to the number obtained by Pop1 minus one. Each time a feasible individual is found, the population Pop1 is substituted by Pop2 and the fixed number of vehicles considered in both populations is decreased by one.

Different mutation and crossover operators can produce different offsprings and thus affect the performance of the genetic algorithm. Dynamic genetic algorithms are considered in (Hong et al., 2002). Since the efficiency of different genetic operators can depend on different problems and also on different stages of the genetic algorithm, the proposed dynamic genetic algorithm is designed to choose different operators as well as to dynamically adjust their application probabilities.

Local route improvement algorithms are considered for a chromosome improvement as an additional step of the genetic algorithm. Multiple improvement algorithms are also considered in computation to better exploit their characteristics. The local route improvement is used to add additional intensification to the genetic algorithm with a view to increase the convergence speed (Potvin and Bengio, 1996; Jung and Moon, 2002; Berger and Barkaoui, 2004; Nagata and Bräysy, 2009). In (Jung and Moon, 2002), usage of Or-opt, crossover and relocation methods together are investigated for the improvement of routes. Another known improvement algorithms, commonly used in VRP implementations are 2-opt, and also its generalization 3-opt and λ-opt.

Most of the feasibility handling approaches deal with the population control to preserve feasible individuals. The common approaches like penalty methods or repair algorithms can help to rank individuals for the next generation by identifying the infeasible ones. However, the crossover and mutation operators are still organized to generate solutions in the whole search space. It is still time consuming to get an acceptable solution. In literature we can find approaches to define the feasibility preserving operators. Limitations still exist where the constraint violation is evaluated after each step and the original solution is restored in an unsuccessful case. Repair algorithms usually take into account a specific problem or specific constraints.

1.5. Insertion heuristics in genetic algorithm operators

Genetic algorithm approaches that deal with infeasible individuals require additional approaches to intensify a search to a feasible search space. Usually these approaches require a specific improvement or repair methods to avoid situations where repair of a single constraint can have a negative impact on other constraints. Another approach is to avoid infeasibility in the created solutions.

Depending on the problem definition and constraints, a feasible solution, where all the nodes are visited without violating constrains, could not be possible to be created. Solutions are possible, where either some of the constraints are not satisfied or not all nodes are included in the solution.

In (Reid, 2000), probability functions are defined to find a feasibility-preserving two-point crossover for a linear constraint problem. However, for a highly constrained problem, where a feasible search space is reasonably small compared to an infeasible search space, only a half-feasible crossover with a single boundary point is discussed.

Usually, in order to create feasible solutions, various approaches of construction heuristics are taken into consideration. Construction heuristics can include the minimization function and work as the stand-alone algorithms. Insertion heuristics are one group of construction heuristics, where the routes are constructed by inserting all the nodes one by one into the routes.

Insertion heuristics are popular because they are easy to implement and they show good characteristics in creating feasible solutions (Campbell and Savelsbergh, 2004). However, they still depend on the methods of selecting the nodes and the place in the route for insertion. In this thesis the usage of the insertion heuristic together with the genetic algorithm approach, seeking for better efficiency, is considered. As already mentioned, the insertion heuristic is usually used in the initialization of solutions in the genetic algorithm. In (Potvin and Bengio, 1996; Jung and Moon, 2002) the authors have proposed the usage of Solomon insertion heuristic to create the initial population that is used in the genetic algorithm. Because of existence of adjustable weights in criteria functions of Solomon insertion heuristic I1 the initial set of different solutions can be generated. A similarity of insertion heuristics can be found in or-opt and relocation algorithms used in the mutation operation, proposed in the paper (Jung and Moon, 2002), where the constraint violation is evaluated for the nodes before inserting them in different parts of the solution. In the literature a random node insertion is also considered for creating the initial population for genetic algorithm (Tan et al., 2006).

In (Potvin and Dubé, 1994) the approach of the genetic algorithm is defined to find the best values of coefficients (α_1, α_2, μ) for Solomon insertion heuristic I1. The coefficient values in the range [0,1] are mapped to values [0, 127] and encoded in 7 symbol substrings as a binary expression and a single point crossover operator is used. The authors argue that the results of insertion heuristic can be greatly improved by a careful search for coefficients.

In (Alvarenga et al., 2005; Ombuki et al., 2006) a push-forward insertion heuristic (PFIH) is used to create an initial solution and also as part of the crossover operator. PFIH originally was defined for the VRPTW by Solomon (1987). PFIH starts by selecting the first node and forming the initial route from a depot. The algorithm inserts all the other nodes into the constructed route by minimizing the insertion cost function for each node. The concept "push-forward" originally means checking pushed-forward values of all the subsequent node in the route (Tan et al., 2001). In PFIH, the first node of the new route is identified deterministically, where the node to be inserted is the one that is distant from the depot, not too far from the last inserted node in the previous route, and that has an early time window. Other nodes are inserted by minimizing the insertion cost by evaluating insertion of all the free nodes in all the existing insertion positions in the route. An important characteristic of PFIH is that insertion of the node is possible, only if no constraint is violated.

From the overview of insertion heuristic usage in genetic algorithms for the VRP we can see that usually the insertion heuristic is used in the initialization step of GA to create the initial set of solutions. There are some approaches to use insertion heuristics in genetic algorithm operators, but the insertion heuristics used are still treated as the methods to support the main algorithm. The authors in (Campbell and Savelsbergh, 2004) describe the benefits of insertion heuristics in handling constraints and in generating feasible solutions. In contrast to insertion heuristics, genetic algorithms are designed to intensify the search towards an optimal solution. However, genetic algorithms require additional approaches to handle the constraints discussed in Section 1.4.

When the insertion heuristic is used as a part of the crossover operator, it plays an important role in the general genetic algorithm approach: a random insertion can increase the diversity of the population, whereas the usage of the minimization function in the insertion can give better results initially, but reduce the diversity. For further crossover operators, we define the following questions:

- what information is taken from parents to create partial (or full) offspring?
- which insertion approach is used to insert unassigned nodes back?

Best cost route crossover (BCRC), proposed in (Ombuki et al., 2006), creates two offsprings from two parents. For a better explanation, let us denote the parent solutions as x_{p1} and x_{p2}, denote the offspring solutions as x_{o1} and x_{o2} and intermediate offspring solutions as x'_{o1} and x'_{o2}. The defined crossover creates an offspring solution in the following steps:

1) N_{temp} = select a random route $r_r \in x_{p2}$;
2) create a partial solution $x'_{o1} = x_{p1} \backslash \{n \in N_{temp}\}$;
3) create x_{o1} by inserting the node $n \in N_{temp}$ into x'_{o1}, by randomly selecting a node from N_{temp} and inserting it with the minimal insertion cost: nodes are inserted into the existing routes; if it is not possible to do insertion due to constraint violation, a new route is created;
4) create x_{o2} by swapping the parent solution and repeating steps 1-3.

The defined crossover operator takes a single parent, forms an offspring from it, partly destroys it and reconstructs it back (*reconstruction* is not the same as *repair*, where *repair* is used to create a feasible version of an infeasible solution). For the stage of destruction, Ombuki et al. (2006) have proposed to use the second parent as a reference, where a single randomly chosen route provides information which node should be removed from the offspring solution. A couple of cases can be noticed in such an approach: a) if a solution has a lot of small routes, a single route could include a small set of nodes, where removal of a small number of nodes from the solution could not give the expected intensification result; b) if the problem is defined only for a single vehicle (i.e. TSP), the resulting solution will have only one route and, in the destruction stage, the whole route will be destroyed. The first case can be solved by increasing the number of routes selected as references. However, the question, what useful information is shared between the parents, and why this approach is better than random node remove, is not explained in (Ombuki et al., 2006). The design of BCRC leads to a minimization of routes, because the nodes to be removed can form the route in the second parent solution, and there exist a probability that the whole route will be removed in the offspring solution.

SBX. In (Potvin and Bengio, 1996) two crossover operators are proposed that repair the generated offspring by removing correlating nodes from it and reinserts them by minimizing the additional detour. The first crossover, called a *sequence-based crossover* (SBX), selects two routes from the parent solutions and merges them by selecting a split place (break-point) in each route:

1) $x'_{o1} = x_{p1}$;
2) select a random route r_{r1} from x_{p1} and a random route r_{r2} from x_{p2};
3) create a new route r_{new} by adding nodes from r_{r1} starting from the beginning till a randomly selected place;
4) append nodes to r_{new} from r_{r2} starting from a randomly selected place till the end;
5) remove duplicates from r_{new} if such exist;
6) $x'_{o1} = x'_{o1} \setminus \{n \in r_{new}\}$ – remove the nodes from x'_{o1} that belong to the new route r_{new};
7) remove r_{r1} from x'_{o1}, add $\forall n \in r_{r1}$ to N_{temp};
8) add r_{new} to x'_{o1};

9) create x_{o1} by inserting the nodes $n \in N_{temp}$ to x'_{o1} by evaluating the insertion cost function;
10) create x_{o2} by swapping the parent solution and repeating steps 1-9.

RBX. The second crossover proposed in (Potvin and Bengio, 1996) is called a *route-based crossover* (RBX). In this crossover, a route from one parent replaces one route from the second parent:

1) $x'_{o1} = x_{p1}$;
2) select a random route r_{r2} from x_{p2};
3) $x'_{o1} = x'_{o1} \setminus \{n \in r_{r2}\}$ – remove the nodes from x'_{o1} that belong to the route r_{r2};
4) remove a random route r_{r1} from x'_{o1}, add $\forall n \in r_r$ to N_{temp};
5) add the route r_{r2} to x'_{o1};
6) create x_{o1} by inserting the nodes $n \in N_{temp}$ to x'_{o1} by evaluating the insertion cost function;
7) create x_{o2} by swapping the parent solution and repeating steps 1-6.

Both crossovers, SBX and RBX, add some parts from both parents to the final solution. The first crossover merges two routes from the opposite parents, so it can be applied in the cases where parent solutions have only one route. If solutions have more than one route, the probability to select parent routes for a crossover, such that the created offspring were competitive in the population, decreases, when the number of routes increases. The operation of removing duplicates in the route might be insufficient. A merge of two routes at random positions can involve a violation of constraints in the offspring solution. For example, let us have a VRPTW, where time window constraints are defined for all nodes. Let us have a break-point selected in the first route r_{r1} after the node $n_{r1,i}$, and a break-point selected in the second route r_{r2} before the node $n_{r2,j}$. The new constructed route will connect two routes to the following route ($n_{r1,1}$, …, $n_{r1,i}$, $n_{r2,j}$, …). The node $n_{r2,j}$ could have an early time window constraint, then, since it was at the beginning of the route r_{r2}, the time window constraint will probably be violated when the node $n_{r2,j}$ is added to the "late" position in the new route. An additional constraint check should be applied to avoid a constraint violation in the offspring.

The RBX preserves a feasibility in the offspring. If the parent routes are feasible, then the routes in the offspring remain feasible. Randomly selected routes in both individuals may have no common node, so the removal of duplicate nodes and removal of a randomly selected route can reduce the number of routes in the intermediate solution. So, this crossover has a possibility to minimize the number of routes. However, after the reconstruction the number of routes can still be increased. If parent solutions have a larger number of routes, then the approach can be adjusted to take a larger number of routes from the second parent. However, there exist a limitation, if there is only one route in the parent solution.

LRX. The crossover used in (Alvarenga et al., 2005) is similar to that of RBX described above, because it combines the routes from the parent individual by evaluating the number of nodes in the routes (let us call it *largest route crossover* (LRX)). Originally, the genetic algorithm approach was defined to handle infeasibility as well. In the original crossover, infeasible routes are skipped in the offspring and added to the list of unassigned nodes. This crossover can also be applied to feasible solutions:

1) $L_r = \varnothing$ is a list of routes;
2) add $\forall\ r \in x_{p1}$ to L_r;
3) add $\forall\ r \in x_{p2}$ to L_r;
4) $x_{o1} = \varnothing$ is the initial empty solution;
5) r_s = select a route from L_r with the largest number of nodes;

6) for $\forall\ r \in L_r$, $r = r \setminus \{n \in r_s\}$ - remove the nodes belonging to r_s from all the other routes;
7) add r_s to x_{o1};
8) repeat the steps 5-7, while L_r has routes with at least one node.

The LRX produces only one offspring. Stochastic PFIH was used as a reconstruction method to insert unassigned nodes in the LRX. If in the deterministic PFIH, the first node (the initial route) is chosen deterministically, in the stochastic PFIH, each new route is started by choosing an unassigned node randomly. By inserting routes with a larger number of tasks, the described crossover intensifies the first objective of the VRPTW problem. So, this crossover is designed for a special problem (or a special objective) and is not effective in the cases, where parent solutions have only one route.

Reconstruction. All the described crossovers use an insertion heuristic for reconstruction of solutions. However, insertion approaches slightly differ in each crossover. In all of them, at first, nodes are inserted into the existing routes, if the constraints are not violated, and a new route created, otherwise. Such a method intensifies the route minimization objective of the VRPTW. Usually, in GA, intensification is a task of the selection operator and depends on a selective pressure. The usage of intensification in the crossover operator needs to be adequate to the intensification in the selection operator, otherwise, the crossover will, most probably, generate an offspring that will not survive in the population.

1.6. Shortest path search

At first sight, the shortest path problem seems to be very simple and global positioning system (GPS) devices and many other systems find the shortest path between two locations rather quickly. When the solution is given within a few seconds, it does not seem very slow and the result is acceptable.

However, modern systems deal with much broader route planning tasks – vehicle routing problems. The objective of the VRP is to find the optimal path, when a number of customers are serviced. The real-world VRP depends on the road network and a task to reach separate nodes in the road graph from the starting node which is the shortest path problem. In order to solve the logistic task made up of $k+1$ nodes, the $(k+1)\times k$ shortest paths need to be calculated. In the simplest case, the distance between these nodes can be calculated according to the coordinates of nodes. However, the difference between the real shortest path and the straight line can be significant for small distances. For example, if the river exists between two nodes, then the shortest path will increase depending on the nearest bridge, or the shortest path is searched in the city with many one-way roads, and then the search of the shortest path may increase several times. For VRP the $(k+1)\times k$ shortest paths need to be calculated and the standard Dijkstra's algorithm can be used to find the shortest path between all pick-up and delivery places. This task becomes more complicated, if we take into account the additional detailed information, for example, a permissible maximum weight of a bridge, and the parameters of the vehicle, or even dynamic information, such as traffic jams that could influence the value of the shortest path. While searching for the VRP solution by the genetic algorithm, different solutions are created that include different travel paths in road network. Each initially calculated shortest path needs to be verified by taking into account the additional information at runtime. If the additional data involves changes in the path (i.e. because of traffic jams the travel time significantly increases) the path needs to be recalculated. If all the paths are calculated by taking into account the additional data initially, it would require a lot of time and all the shortest paths would not be necessary used in VRP optimization. However the search of shortest path at runtime of VRP optimization will affect a calculation time. The usage of the different shortest path search speed-up techniques could reduce this time. So, the shortest path search is of a great importance in vehicle routing problems that involve real-world road network.

When searching for the shortest path in the road network, a graph with non-negative weights is commonly used. An edge in the graph can be described by any numerical value: distance, time, speed, etc. And a commonly used approach for finding the shortest path in the graph is Dijkstra's algorithm.

1.6.1. Shortest-path computation speed-up techniques

Route planning and shortest path problems have gained more and more attention in recent researches. There are some attempts to develop new algorithms and accelerate the already known ones, by adding a new ingredient or processing additional information. Some of them pay attention to dynamic information, such as road congestion or weather conditions. The others refer to the fact that the path between two points is static, i.e., a graph does not change during the calculation of the shortest path between two nodes, and no additional calculations, based on the traffic condition changes, are made. Such algorithms are simply called static route planning algorithms.

As mentioned before, one of the most popular static algorithms is Dijkstra's algorithm (Dijkstra, 1959). It is known as the most efficient algorithm for the shortest path problem in a directed weighted graph. Here we focus on the published results on speed-up techniques of Dijkstra's algorithm. Dijkstra's algorithm is a weighted breadth-first search algorithm. Although this algorithm was designed to calculate the “shortest distance” from one node to other nodes in the graph, it can be easily used for calculating the distance from one node to the destination node. One of the speed-up modification algorithms is a bidirectional Dijkstra's algorithm (Goldberg et al., 2006; Koehler et al., 2006; Berrettini et al., 2009). This method calculates a path starting a search operation from both sides at the same time. The calculation “meets” and stops somewhere in the middle of the road and gives the answer in quest. We will review the mentioned modification more in detail in Section 1.6.3.

Another attempt to speed-up Dijkstra's algorithm is index usage for the priority queue of labeled nodes. A Fibonacci heap (von Lossow, 2007) or a binary heap (Madduri et al., 2007) is proposed for indexing a set of nodes. Usage of such an index in a queue of nodes speeds up the algorithm just by one step, extracting nodes with the minimum distance from all the available labeled nodes. Using this technique, the calculation accelerates only in a very large graph, where the count of labeled nodes significantly increases during calculation. However, one-step acceleration does not yield a significant result.

Also, attempts are made to implement Dijkstra's algorithm in reconfigurable hardware. The paper (Tommiska and Skytta, 2001) provides an overview of applications of reconfigurable computing in network routing, where a FPGA-based (FPGA – field-programmable gate array) version of Dijkstra's shortest path algorithm is also presented and differences of the performance between the FPGA-based and microprocessor-based versions of the same algorithm are compared. Another interesting hardware used for Dijkstra's algorithm implementation is DAPDNA-2 – dynamically reconfigurable processor developed by IPFlex (Ishikawa et al., 2007). The modified algorithm finds the shortest paths in parallel, using the *processing elements* (PE) matrix. Although the use of the array of 376 PE compared to the microprocessor gives better results, the implemented schema is designed only for DAPDNA-2 and is not suitable for the other hardware.

Since Dijkstra's algorithm is static and calculations are made with a static graph, various preprocessing techniques are used for speeding up the process. One of the easiest methods is to count the shortest paths between all k nodes (Romeijn and Smith, 1999). The obtained $k \times k$ matrix then can be easily used in the next level route planning system. However, the use of such an approach together with road data of the real world would be very inefficient. Great speed-up factors can be achieved using highway hierarchies (Koehler et al., 2006; Knopp et al., 2007). The highway hierarchy method is based on the idea that only a highway network needs to be searched outside a fixed size neighborhood around the source and target. A highway approach is faster in preprocessing as compared to the Arc-flags approach, but calculates the shortest path more slowly (Koehler et al., 2006). The main "arc-flag" method idea is a graph partition into regions. Then, all the edges are reviewed by marking with property flags, which indicate whether the edge is on the shortest route to the regions or not. During the route search, only those edges are selected the properties of which are appropriate, and the rest are rejected.

In (Koehler et al., 2006), the graph partitioning techniques are reviewed: rectangular partition (grid), quadtree, kd-tree, multi-way arc separator. The splitting technique can be used to form a second-level graph, i.e., the graph is split into parts, which together make a new graph of macronodes and the macronodes are comprised of smaller graphs (Romeijn and Smith, 1999). Thus, Dijkstra's algorithm would be first used in the macronode graph, and then, according to the obtained results, it would be used in other smaller graphs. All of these preprocessing technologies give good results, but they also have disadvantages: each preprocessing technique requires additional data storage for the edge or node, such as in the arc-flag method, where each node keeps property flags about all regions. Thus, for a very huge graph (for example, OpenStreetMap data is made up of ~600M nodes), we should have to deal with memory problems, or will reduce the algorithm efficiency using a hard disk and reading it constantly. Another drawback is a difficult implementation using dynamic data, such as roads closed for repair, or congestion, etc. Then all the partitions will have to be preprocessed once more, which may last very long for a very large graph. Important aspect is that the preprocessing of the data is performed through the calculation of distances: the division is made by taking into account geographical features of the road. Since Dijkstra's algorithm may calculate the route using the edge cost, which may include not only the distance, but also the time or other values, preprocessing methods lose their value.

Another useful technology to speed up Dijkstra's algorithm is parallel computing. The approach has already been mentioned in adapting Dijkstra's algorithm to reconfigurable hardware. To speed up the preprocessing part, the $k \times k$ road calculation is proposed using the parallel computing. To calculate all roads, the usage of total $\sqrt{k}$ processors is proposed (Romeijn and Smith, 1999; Lu and Chen, 2006). However, this method is still very limited, even with a modern technology. It is suitable to use more in local computer network, which usually covers a smaller physical area, like home, office, or a small group of buildings, than in street routes. Dijkstra's algorithm is iterative and, in each iteration, it uses the data obtained in the previous iteration. Due to these properties, the algorithm cannot be easily adapted to parallel computing, but it is still widely considered. Usually, due to the large amount of data, it is difficult to adapt Dijkstra's algorithm to the distributed memory parallel computing. However, today's multi-core technology allows us to easily implement parallel calculations based on a shared memory technique, called *transactional memory* (TM), thus avoiding synchronization of large amounts of data. TM is a technology in multi-core platforms that allows several different processes to access the same memory location. The developer has a possibility to mark certain parts of the code, indicating that during the program execution at this point, some memory allocations can be accessed by several different processes. TM monitors process the transactions, and, if several processes are trying to access the conflicting memory, TM decides how to handle (Anastopoulos et al., 2009; 2009a). In general, all the processes are blocked, only one process can access the memory allocation and, when the operation is completed, the blocked process is again continued. If the transactions are not conflicting, processes are carried out without any interruption. Because of their properties TM is very useful for parallel computations, based on the shared memory. There are some attempts to implement Dijkstra's algorithm in parallel computing using the transactional memory together with helper threads (Lu and Chen, 2006; Anastopoulos et al., 2009, 2009a). Such a parallel computing technology is proposed to use in the inner loop of Dijkstra's algorithm, where nodes are processed for labeling. The helper thread reads the tentative distance of the vertex in the queue and attempts to relax its outgoing edges based on this value (Anastopoulos et al., 2009). When the processes are finished, the main process continues its work up to the next inner loop. It is also proposed not to wait to the end of the helper processes, and the main process continues to work without paying attention to the adjacent processes (Anastopoulos et al., 2009, 2009a). Such an approach cannot work properly without TM technology. However, such helper thread computing can last shorter than its starting and termination. The periodic creation and destruction of such processes could adversely affect the operation of the algorithm.

The paper (Edmonds et al., 2006) gives a similar parallel computing method, but introduces an additional heuristic, for example, choosing which edges need to be dismissed. Also, attempts are made to adapt parallel computing to algorithms that are based on preprocessing. In a modified arc-flag algorithm, a linear preprocessing method was left, but only the shortest path searches are performed in parallel (Berrettini et al., 2009). The parallel computing is possible without TM technology, and then each new process will have to keep in memory a separate copy of the road data (Lu and Chen, 2006). However, bearing in mind the size of the modern road networks, problems can arise due to technological limitations, while having multiple copies of the same graph in memory. Although this method can be easily implemented in parallel with a separate memory, this method requires an additional synchronization in order to update data everywhere.

Acceleration and parallelization of Dijkstra's algorithm still remains a complex problem and completely unsolved. In order to speed-up this algorithm, several methods together are often used, for example, Fibonacci heap and highway hierarchies. However, in order to speed up Dijkstra's algorithm, the main idea of this algorithm is often distorted: if it is the “shortest” path search algorithm. And many of speed-up methods lead to nearly the shortest path calculation, such as heuristic introduction or highway hierarchies.

1.6.2. Dijkstra's algorithm

Dijkstra's algorithm finds the shortest distance from one node to all the others in the graph with non-negative weights. Let us consider a road graph $G^r = (N^r, E^r)$, which consists of the nodes of $n^r \in N^r$ and edges $e^r \in E^r$. Let us denote by $l_s(n^r)$ the distance from the node n^r to the starting node $n_s^r \in N^r$, and by $l(n_v^r, n_u^r)$ the distance from node n_v^r to node n_u^r. All the labeled nodes are organized by the algorithm in the priority queue Q and all the visited nodes are stored in array N_S^r. During each iteration the algorithm extracts node n_k^r from queue Q with the lowest value of $l_s(n_k^r)$. Then all the outgoing edges of node n_k^r are relaxed, which could reduce the keys of the corresponding neighbors. Relaxing an edge (n_k^r, n_v^r) means testing whether we can improve the shortest path to n_v^r found so far by going through node n_k^r. If $l_s(n_k^r) + l(n_k^r, n_v^r)$ is less than $l_s(n_v^r)$ found so far, $l_s(n_v^r)$ is replaced by a new value. If the adjacent nodes have not yet been labeled, they are inserted in queue Q. This operation is performed in the decreaseKey operation (Goldberg et al., 2006; Ishikawa et al., 2007; Anastopoulos et al., 2009a). The following pseudo-code illustrates

Dijkstra's algorithm. The algorithm terminates when the destination node is found.

```
procedure Dijkstra($G^r = (N^r, E^r), n_s^r, n_d^r$)
        $l_s(n_s^r) = 0$
        $N_S^r = \varnothing$ // visited nodes
        $Q = \varnothing$ // labeled nodes
        $Q.insert(n_s^r, 0)$
        while $Q$ is not empty // outer loop
                $n_u^r = Q.extractMin()$
                $S.addNode(n_u^r)$
                if $n_u^r = n_d^r$ // stopping criterion
                        break // route found
                end if
                for each $n_v^r$ adjacent to $n_u^r$ // inner loop
                        $sum = l_s(n_u^r) + l(n_u^r, n_v^r)$
                        if $l_s(n_v^r) > sum$
                                $Q.decreaseKey(n_v^r, sum)$
                                $l_s(n_v^r) = sum$
                                $pr(n_v^r) = n_u^r$ // set predecessor
                        end if
                end for
        end while
end
```

Dijkstra's algorithm is a labeling algorithm. When the distance from the current node to the start node is known, then adjacent nodes are labeled. So by starting labeling of the nodes adjacent to the start node, the algorithm iterates until the set of labeled nodes is empty. Dijkstra's algorithm is a greedy algorithm because at each step, the best alternative is chosen. The algorithm produces a correct shortest-paths tree whose top is the start node, and to every other node in the graph G^r there is only one possible path. So, while executing the search from the start node n_s^r to the target node n_d^r, the algorithm must visit all the nodes $n_v^r \in N^r$, with the distance $l_s(n_v^r) < l_s(n_d^r)$.

1.6.3. Bidirectional algorithm

In search of the path between two specific nodes of the graph, a modified Dijkstra's algorithm – bidirectional method – can be used. This method performs the searches starting from the start and end nodes (Goldberg et al., 2006; Berrettini et al., 2009). The algorithm is simply run by executing one step on each side in a single period. At first, the processing step with the extracted node is performed from the start node, and then the same calculation is made from the end node. During such a step a single node is extracted from the priority queue, marked as visible and all the corresponding edges are relaxed (inner loop). Such process will not necessarily be symmetrical. It will depend on the number of edges of all the visited nodes.

To execute such algorithm, separate data containers must be used, so each search must have its own sets for labeled and visited nodes. In the following pseudo-code, a forward search is using the priority queue Q_S and the set of visited nodes N_S^r and a backward search is using the priority queue Q_D and the set of visited nodes N_D^r.

```
procedure bidirectionalDijkstra($G^r = (N^r, E^r), n_s^r, n_d^r$)
    $Q_S = \varnothing$ // labeled nodes in search from start
    $Q_D = \varnothing$ // labeled nodes in search from end
    $N_S^r = \varnothing$ // visited nodes in search from start
    $N_D^r = \varnothing$ // visited nodes in search from end
    ...
    while $Q_S$ is not empty and $Q_D$ is not empty // outer loop
        // calc from start
        $n_u^r = Q_S$.extractMin()
        $N_S^r$.addNode($n_u^r$)
        ...
        if stoppingCriterion() is true // stopping criterion
            break
        end if
        for each $n_v^r$ adjacent to $n_u^r$ // inner loop
            ...
        end for
        // calc from end
        $n_u^r = Q_D$.extractMin()
        $N_D^r$.addNode($n_u^r$)
        ...
        if stoppingCriterion() is true // stopping criterion
            break
        end if
```

```
                for each $n_v^r$ adjacent to $n_u^r$ // inner loop
                ...
                end for
        end while
end
```

The bidirectional algorithm stops when the stopping criterion (stoppingCriterion()) is met. This occurs somewhere in the middle between the start and end nodes. Since the bidirectional Dijkstra's algorithm is two-sided and the search is carried out from the start node and from the end node at the same time, it is necessary to establish a clear stopping criteria. An algorithm without such criteria will be run as long as the search from the start node will find the end node or the search from end node will find the start node. In this case, there will be a lot of nodes that will be visited twice by contrariwise computations. So, there will be such a set $N_H^r = N_S^r \cap N_D^r$, and the set N_H^r will consist of all the twice visited nodes. To increase the efficiency of the bidirectional Dijkstra's algorithm, the set N_H^r needs to be minimized. The papers (Goldberg et al., 2006; Berrettini et al., 2009) present two possible stopping criteria: when the current labeled node has already been labeled by the other search and when the current visited node has already been visited by the other search. With the first stopping criterion the algorithm stops when such node n_w^r is found, where $n_w^r \in Q_S$ and $n_w^r \in Q_D$. This stopping criterion can be defined as the intersection of two sets: $Q_S \cap Q_D = \{n_w^r\}$. If we denote the shortest path from the node n_v^r to the start node by $P(n_v^r)$, $n_v^r \in N^r$, and the shortest path from the node n_u^r to the end node by $P^|(n_u^r)$, $n_u^r \in N^r$, we get the shortest path $P(n_w^r) \cup P^|(n_w^r)$ with the first stopping criterion. Application of this stopping criterion to the search in the example graph is illustrated in Figure 4. The shortest path is ACB.

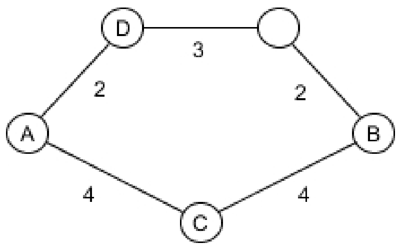

Fig. 4. The bidirectional algorithm chooses the shortest path ACB by applying the first stopping criterion

The algorithm stops by applying the second stopping criterion, when the node n_z^r is found, where $n_z^r \in N_S^r$ and $n_z^r \in N_D^r$. This stopping criterion can be defined as the intersection of two sets: $N_S^r \cap N_D^r = \{n_z^r\}$. Then the shortest path from the node n_s^r to the node n_d^r is $P(n_z^r) \cup P^|(n_z^r)$. Figure 5 illustrates the example graph, in which we get the wrong result by applying the second stopping criterion. It happens because $N_S^r \cap N_S^r = \{E\}$, and the nodes B and C remain just labeled by different searches.

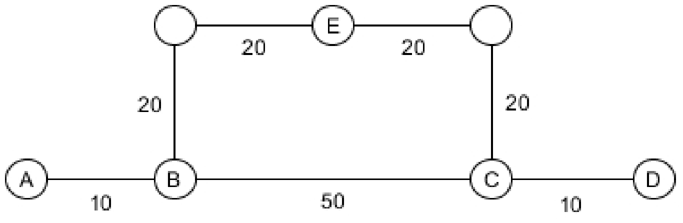

Fig. 5. The bidirectional algorithm chooses the path ABECD by applying the second stopping criterion

By applying the second stopping criterion to the first example, we also get the "wrong shortest" path, however, in the second example, the difference between the correct and wrong shortest paths is more obvious. By applying the second stopping criterion to the second example, we get the shortest path ABECD instead of ABCD. It happens because there exists such node E, where $l(B, E) < l(B, C)$ and $l(C, E) < l(B, C)$. In the worst case, the path BEC can be almost twice longer than the path BC. Such situation can lead to a really significant inaccuracy.

1.7. Summary

Various vehicle routing problems are analyzed in the literature. Different variants of the VRP includes specific constraints that describe some specific situation of the real-world vehicle routing problem. A set of algorithms, their modified variants and various hybrids are analyzed for solving the mentioned problems. Only a small number of researches focus on creating algorithm that could be applicable to a larger group of VRP.

The genetic algorithm is one of the metaheuristic approaches that is also used for solving VRP. The genetic algorithm is a stochastic approach that is based on ideas of evolution theory. The search in the genetic algorithm depends on two factors: selective pressure and population diversity. These factors play an important role in the genetic algorithm, where the selective pressure describes the intensification of search for the optimal solution by choosing better individuals for reproduction in each next generation, and the diversity maintenance is responsible for having a non-homogeneous population. New solutions are obtained by applying recombination operators on selected individuals from the previous generation. When dealing with a constrained problem, genetic operators can generate solution in the whole search space, thus requiring additional approaches to find feasible solutions. Penalty methods are common approaches to deal with constraints in GA. However, a penalty approach does not prevent generation of solution in infeasible search space. Repair and improve methods are other approaches for dealing with constraints in genetic algorithm, however usually they are hardly adjustable to new constraints. Existing approaches for solving VRP are usually designed for special problem and are hardly applicable to the different problem with different constraints or objective, so there still is a need for the algorithm that could be extended with additional parameters and applicable for a larger group of VRPs.

The shortest path calculation is an important part for solving a real-world VRP. The different shortest paths can exist depending on the additional search data (i.e. permissible maximum weight of a bridge and the parameters of the vehicle, or even dynamic information, such as traffic jams). This chapter reviews Dijkstra's shortest path algorithm speed-up techniques for calculation of the shortest path while searching for real world vehicle routing problem solution that involves real road data. One of the modification of Dijkstra's algorithm, a bidirectional Dijkstra's algorithm lacks of stopping criterion definition and in some cases this leads to the significant error of the calculation.

The review presented in the chapter has been also published in (Vaira and Kurasova, 2010; Vaira and Kurasova, 2011; Vaira and Kurasova, 2013; Vaira and Kurasova, 2013a; Vaira and Kurasova, 2014).

Chapter 2
A new algorithm for vehicle routing problem

This chapter presents the main theoretical results of the doctoral research. **Section 2.1** proposes the genetic algorithm for vehicle routing problem. Proposed algorithm involves usage of insertion heuristic in genetic operators, proposed expression of the individuals in population, second population usage in mutation operator, and genetic algorithm operators based on repeated reinsertion approach. **Section 2.2** describes a general vehicle routing problem definition and genetic algorithm operators designed for intensifying search. **Section 2.3** describes a parallel version of Dijkstra's algorithm for the shortest path search. **Section 2.4** summarizes this chapter.

2.1. Genetic algorithm for vehicle routing problem

A genetic algorithm based on insertion heuristics without considering any additional local search methods for the improvement is proposed in this section. The definition “genetic algorithm” can describe either a general approach or a set of the specific genetic operators. In this thesis the proposed version of genetic algorithm for VRP with constraints will be called “new genetic algorithm” further on to distinguish it from other approaches.

Genetic algorithms and insertion heuristics combine together their best characteristics to search for the optimal solution. It is generally accepted that any genetic algorithm for solving a problem should have basic components, such as a genetic representation of solutions, the way to create the initial solution, the evaluation function for ranking solutions, genetic operators, values of the parameters (i.e. population size, probabilities for applying genetic operators, etc.). In Sections 2.1.1, 2.1.2, 2.1.3, 2.1.4 there are described all components of the new genetic algorithm in more detail.

2.1.1. Incorporating insertion heuristics

As already mentioned in Section 1.5, in some variants of the genetic algorithm for VRP the insertion heuristic is used in the initialization step, where Solomon I1 method has been used with random coefficients for the criterion function. Although such a random insertion is possible, randomization is limited by the defined criterion function.

Let us assume that we have a set of nodes $N=\{n_0, ..., n_k\}$, where $N\backslash\{n_0\}$ are the nodes that should be visited by a single vehicle and n_0 is the depot. The constructed partial solution is $x_0=(\{n_0\}, r_0=\varnothing, N_{r0}=N\backslash\{n_0\})$, where r_0 is the empty set of arcs, N_{r0} is a set of unvisited nodes. So the solution contains only the depot n_0.

In the first iteration the randomly selected node n_{r1} from N_{r0} is inserted into a partial solution x_0. The new constructed partial solution is $x_1=(\{n_0, n_{r1}\}, r_1=\{(n_0, n_{r1}), (n_{r1}, n_0)\}, N_{r1}=N_{r0}\backslash\{n_{r1}\}=N\backslash\{n_0, n_{r1}\})$. Two new arcs (n_0, n_{r1}), (n_{r1}, n_0) have been created in the solution. Assume that the route is feasible and it can be agreed that it would be the shortest route for a single customer problem $\{n_0, n_{r1}\}$.

In the second iteration a random node n_{r2} is selected from N_{r1}. For the newly selected node there exist two possible places for insertion in the solution x_1: either in the arc (n_0, n_{r1}) or in the arc (n_{r1}, n_0). Assume that both insertions are feasible and the arc (n_{r1}, n_0) has a lower insertion cost than the arc (n_0, n_{r1}). So the newly constructed partial solution is $x_2=(\{n_0, n_{r1}, n_{r2}\}, r_2=\{(n_0, n_{r1}), (n_{r1}, n_{r2}), (n_{r2}, n_0)\}, N_{r2}=N\backslash\{n_0, n_{r1}, n_{r2}\})$. The newly constructed partial solution is feasible and optimal.

In the third iteration another random node n_{r3} is selected from N_{r2} and a new optimal solution x_3 is created.

In each next iteration k a random node n_{rk} is selected from $N_{r(k-1)}$. If there exists such an arc in r_{k-1}, where the inserted node does not violate any constraints and produces a new feasible partial solution x_k, the added node n_{rk} removes the existing arc (n_i, n_j) and adds two new arcs (n_i, n_{rk}) and (n_{rk}, n_j). If we find the optimal partial solution in the iteration $k-1$, the solution created in iteration k is not necessarily optimal, because two new arcs (n_i, n_{rk}) and (n_{rk}, n_j) are created and there can exist a shorter path to some nodes in the route r_k.

The random insertion heuristic with only one minimization objective, i.e. traveling salesman problem (TSP) with the total traveling path minimization, has a complexity $O(k^2)$ to construct a single solution, where the complexity of search for the best arc to insert a single node is $O(k)$. When adding additional constraints, the computation time is affected. Solving VRPTW by the insertion heuristic has the complexity $O(k^3)$. The handling time window constraint involves additional check for any violations occurring in a partial route after inserting a new node. So, for each node to be inserted the best arc search has the complexity $O(k^2)$, where the insertion complexity for all nodes is $O(k^3)$.

As already mentioned, when solving the problem with constraints by the genetic algorithm, the constraint violation is checked per solution, usually in the form of penalty or repair cost. In the proposed algorithm the constraint violation is evaluated in the insertion. For each randomly selected node a feasible insertion needs to be determined, where feasible insertion means finding such an arc of a partial solution, where the inserted node as well as all the previously added nodes do not violate any constraints. The partial solution with a new inserted node should remain feasible. Let us define the function $h_c(n, a)$ that evaluates the violation for the certain constraint c with the newly inserted node n in arc a. The function h_c is similar to the function f_c, where f_c evaluates the whole solution for constraint violation, but h_c is applied to a single node insertion only. The function h_c is defined here as follows: $h_c(n, a) = 0$, if the constraint $c \in C$ is satisfied, and $h_c(n, a) > 0$, otherwise. Insertion of the node n into the arc a is feasible, if $H_c(n, \mathrm{a}) = 0$, where

$$H_c(n,a) = \sum_{c \in C} h_c(n,a)$$

Subject to the insertion order and the constraint set, a partial route can be constructed in such a way that no additional nodes can be inserted without violating constraints. So, usage of random insertion heuristics does not always guarantee the creation of a feasible solution, but the feasibility can be preserved in a partial solution. An infeasible insertion would require an additional definition of constraint hierarchy or any decision variables for ranking constraints or different penalty approaches for the evaluation of the constraint violation.

In order to avoid any additional complexities, we define the solution x of the genetic algorithm as follows:

$$x = (R = \{r_1, \ldots, r_t\}, U = \{n_1, \ldots, n_u\})$$
$$r_i = (N_i, A_i)$$
$$N_i = \left\{n_0, n_{i_1}, \ldots, n_{i_p}\right\}$$
$$A_i = \left\{a_{i_1} = (n_0, n_{i_1}), a_{i_2} = \left(n_{i_1,} n_{i_2}\right), \ldots, a_{i_p} = \left(n_{i_{p-1}}, n_{i_p}\right), a_{i_{p+1}} = \left(n_{i_p}, n_0\right)\right\}$$
$$N_v = \{N_1 \cup \ldots \cup N_t\} \backslash n_0$$

where each route $r_i \in R$ represents a vehicle traveling path and U is a set of unassigned nodes left due to constraint violation. The single route r_i is represented as a graph, where each arc a_i is the shortest path between two nodes. Set U is part of the solution x, where $N_v \cup U = N$ and $N_v \cap U = \varnothing$. If set U is empty, then the solution x is feasible or infeasible, otherwise.

In the proposed algorithm the insertion is carried out as follows. An initial solution has an empty route list ($R = \varnothing$) and an empty unassigned node list ($U = \varnothing$). The node n_r is randomly selected from the set N. All arcs are checked for feasible insertion of the selected node. All the arcs that pass the constraint violation check are then evaluated by the insertion cost function $c(n_s, n_r, n_t)$, where the path length is the cost. The arc with the smallest value $c(n_s, n_r, n_r)$ is chosen for the insertion of the node n_r. If no arcs pass the violation check, a new route is started. If the maximum number of routes is reached, the node n_r is added to an unassigned node list U. The following pseudo-code describes the insertion process:

```
$N_u = N$
$R = \emptyset$
$U = \emptyset$
while $N_u \neq \emptyset$
        $n_r$ = select a random node from $N_u$
        $A$ = get all arcs from $R$
        for each constraint $c \in C$
                remove arcs from $A$ where insertion of $n_r$ violates constraint $c$
        end for
        if $A = \emptyset$ and $|R| < c_v$
                $r_i$ = new route($\{n_0, n_r\}$)
                add $r_i$ to $R$
        else if ($A \neq \emptyset$)
                find $a \in A$, $a = (n_s, n_t)$ by minimizing the function $c(n_s, n_r, n_t)$
                insert $n_r$ to $a$
        else
                add $n_r$ to $U$
        end if
end while
```

Figure 6 shows the insertion steps where a filled circle represents the depot node n_0, the arrows and empty circles represent the route r_i, a dotted circle represents the node n_r selected for insertion and the dotted arrows show possible insertion arcs. The maximum vehicle number constraint $c_v \in C$ check is integrated in the insertion process.

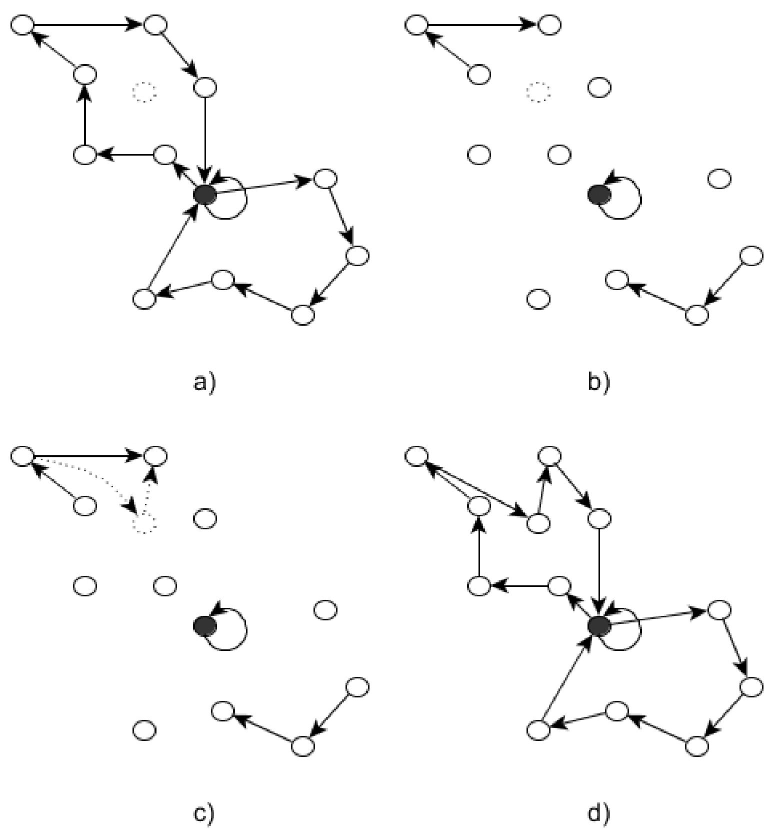

Fig. 6. Node insertion process: a) current solution constructed; b) arcs, where the feasible insertion of a node is possible; c) search for the minimal insertion cost; d) solution with the inserted node

Capacity constraints $C_c \subseteq C$ define the maximal allowed amount of goods that can be assigned to a vehicle. The load increases by assigning a new node to the route. Let us define the function $d_c(n_j)$ that evaluates the load of the single node n_j. The condition $c_i^c \geq \Sigma d_c(n_j)$, $\forall n_j \in r_i$ and $c_i^c \in C_c$, should not be violated in the VRPTW. The check of this constraint has the complexity O(1) for a single node insertion. In the VRPPD the load varies during traveling and the function $d_c(n_j)$ represents loading or unloading at a specified node n_j, where $d_c(n_j) > 0$, if goods are loaded, and $d_c(n_j) < 0$, if goods are unloaded. Node insertion to one place can involve capacity violation in other places. The function $g_c(a_i)$ calculates the current capacity load in the arc a_i. For node to be inserted into the arc a_i all subsequent arcs in a current partial solution need to be evaluated for possible constraint violation and this can take a long computation time. To avoid this, we initially compute an available load space in each arc and then determine all arcs, where insertion does not violate capacity. The function $g_{fc}(a_i)$ is used to determine the maximal capacity available for the node insertion in the arc a_i without involving the constraint violation in the subsequent parts of the route. The function $h_{capacitiy}(a_i, n_r)$ defines constraint check function that checks the violation of capacity constraints, while inserting the node n_r into the arc a_i. The check for capacity constraint has the complexity O(k) for a single node insertion. The insertion does not violate capacity constraints, if $h_{capacitiy}(a_i, n_r) = 0$:

$$h_{capacity}(a_{i_s}, n_r) = \begin{cases} 0 & \text{if } d_c(n_r) \le g_{fc}(a_{i_s}) \\ d_c(n_r) - g_{fc}(a_{i_s}) & \text{otherwise} \end{cases}$$

$$g_{fc}(a_{i_s}) = \begin{cases} \min\{g_{fc}(a_{i_{s+1}}), c_i^c - g_c(a_{i_s})\} & \text{if } a_{i+1} \text{exist} \\ g_c(a_{i_s}) & \text{otherwise} \end{cases}$$

$$g_c(a_{i_s}) = \sum_{j=0}^{i_s} d_c(n_j), n_j \in r_i$$

Time window constraints $C_{tw} \subseteq C$ have characteristics similar to capacity constraints in the VRPPD: adding a new node in one place can involve a constraint violation in other places of the current route. The defined function $t_a(n_j)$, $n_j \in r_i$, identifies the arrival time to node n_j in route r_i. For node to be inserted into $\forall a \in r_i$ all subsequent parts in current partial solution need to be evaluated for possible time violation: for each subsequent node $n_j \in r_i$ equation $t_a(n_j) \le c_j^{tw}$ should be satisfied after insertion. Evaluation of the equation can take long computation time. To speed-up insertion, we initially compute available time space in each arc and then determine all arcs, where the insertion does not violate time windows constraint. Figure 7 represents two possible situations of arrivals at customers. If a vehicle arrives at a customer in the defined time window, the available time is equal to the time left to the end of window (Figure 7 b). If a vehicle arrives too early, the waiting time is added to the available time (Figure 7 a).

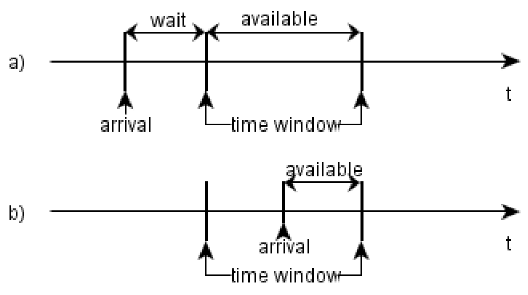

Fig. 7. Arrival at a customer with time window constraint: a) arrival too early; b) arrival in the time window

The function $g_{ft}(a_i)$ is used to determine the maximal amount of time available for the node insertion in the arc a_i without involving a constraint violation in subsequent parts of the route. Let us define the function $h_{tw1}(a_i, n_r)$ that evaluates time window constraint violation in a subsequent part of route and the function $h_{tw2}(a_i, n_r)$ that evaluates time window constraint violation for newly inserted node n_r. The insertion does not violate time window constraints if $h_{tw1}(a_i, n_r) + h_{tw2}(a_i, n_r) = 0$. The time window constraint evaluation does not increase the complexity of a single node insertion and it still remains O(k).

$$g_{ft}(a_{i_s}) = \begin{cases} \min\{g_{ft}(a_{i_{s+1}}), g_t(a_{i_s})\} & \text{if } a_{i_{s+1}} \text{ exist} \\ g_t(a_{i_s}) & \text{otherwise} \end{cases}$$

$$g_t(a_{i_s}) = c_{i_s}^{tw} - (t_a(n_{i_s}) + t_w(n_{i_s})), c_{i_s}^{tw} \in C_{tw}, n_{i_s} \in r_i$$

$$t_a(n_{i_s}) = \begin{cases} t_t(n_0, n_{i_s}) & \text{if } s = 1 \\ t_l(n_{i_{s-1}}) + t_t(n_{i_{s-1}}, n_{i_s}) & \text{otherwise} \end{cases}$$

$$t_l(n_{i_s}) = t_a(n_{i_s}) + t_w(n_{i_s}) + t_s(n_{i_s}) - \text{leaving time}$$

$$t_s(n_{i_s}) - \text{service time}$$

$$t_t(n_{i_{s-1}}, n_{i_s}) - \text{travel time}$$

$$t_w(n_{i_s}) - \text{waiting time}$$

$$h_{tw_1}(a_{i_s}, n_r) = \begin{cases} 0 & \text{if } t_i(a_{i_s}, n_r) \leq g_{ft}(a_{i_s}) \\ t_i(a_{i_s}, n_r) - g_{ft}(a_{i_s}) & \text{otherwise} \end{cases}$$

$$t_i(a_{i_s}, n_r) = \begin{cases} t_t(n_0, n_r) + t_w(n_r) + t_s(n_r) + t_t(n_r, n_{i_s}) & \text{if } s = 1 \\ t_t(n_{i_{s-1}}, n_r) + t_w(n_r) + t_s(n_r) + t_t(n_r, n_{i_s}) & \text{otherwise} \end{cases}$$

$$h_{tw_2}(a_{i_s}, n_r) = \begin{cases} 0 & \text{if } t_a(a_{i_s}, n_r) \leq c_r^{tw} \\ t_a(a_{i_s}, n_r) - c_r^{tw} & \text{otherwise} \end{cases}$$

$$t_a(a_{i_s}, n_r) = \begin{cases} t_t(n_0, n_r) & \text{if } s = 1 \\ t_l(n_{i_{s-1}}, n_r) + t_t(n_{i_{s-1}}) & \text{otherwise} \end{cases}$$

The pick-up and delivery constraints $C_{pd} \subseteq C$ connect pick-up and delivery nodes with a logical relation. In order to determine the arcs for a feasible insertion of node n_r, the opposite node n_{op} (pick-up or delivery node) has to be examined. If n_{op} is not yet assigned to the route, all arcs in the partial solution remain competitive for the insertion of the node n_r. If n_{op} has already been assigned to the route, the following rules are applied:

- if n_r is a delivery node, arcs, where the insertion of node n_r is possible, are in the same route as the node n_{op} and after the node n_{op};
- if n_r is a pick-up node, arcs, where the insertion of node n_r is possible, are in the same route as the node n_{op} and before the node n_{op}.

In the initialization of the genetic algorithm an initial population with the above described random insertion process is created, where the creation of single solution has complexity O(k^2). The infeasible solutions can still be generated by the defined insertion method, where infeasible solutions have not empty set of unassigned nodes U. In the proposed algorithm a feasible solution is always treated better than the infeasible solution (infeasible solution is the one, that has some nodes not assigned to routes). The following pseudo-code shows how a better solution x_{min} is identified from the two solutions x_i and x_j:

```
**if** $f_u(x_i) \neq f_u(x_j)$, where $f_u(x_i) = |U_i|$, $U_i \in x_i$
    **if** $f_u(x_i) < f_u(x_j)$,
        $x_{min} = x_i$
    **else**
        $x_{min} = x_j$
**else if** $f_v(x_i) \neq f_v(x_j)$
    **if** $f_v(x_i) < f_v(x_j)$
        $x_{min} = x_i$
    **else**
        $x_{min} = x_j$
**else**
    **if** $f_d(x_i) < f_d(x_j)$
        $x_{min} = x_i$
    **else**
        $x_{min} = x_j$
```

Described random insertion approach is suitable to increase diversification in genetic algorithm population by creating initial solutions. By removing some nodes and reinserting them back different solutions can be created, however, they can be either better ones or worse ones. Genetic operators that involve a described random insertion approach are proposed in Sections 2.1.3 and 2.1.4. Section 2.1.2 presents the proposed genetic algorithm.

2.1.2. New genetic algorithm

In the proposed genetic algorithm crossover and mutation operators are defined in the "remove and reinsert" approach. The approach is similar to a single point relocation method, where the node is extracted and inserted into a different place. However, reinsertion of a single node in a different place can be unsuccessful, because the constructed routes have reached constraint limits and cannot be extended by an additional node. If a single node has been chosen for reinsertion, there is a large probability that the node will be inserted in the same place from which it has been removed. In order to enable the node reinsertion, multiple nodes have to be extracted. The crossover and mutation operators that follow the idea of node reinsertion are defined in Sections 2.1.3 and 2.1.4.

In the proposed algorithm the mutation operator is applied with probability $MP = 0.1$ and the crossover operator is applied to all individuals selected for mating. In the crossover operation new offsprings are generated from two parent solutions that are selected from population by using the ranking method. The new offsprings are added to the population and the worst individuals are removed from the population to keep the same population size in each iteration. The defined mutation operators are based on a random insertion and can produce individuals that will not survive. In order to increase the probability of the mutation operator to generate individuals that will survive, a second population is created. The success of the mutation operator depends on the generated solution in comparison to the solutions in the population. If the fitness value of generated solution is better than the average fitness value in the population, such solution will have a higher probability to be selected for reproduction. If the fitness value of generated solutions is similar to the worst fitness value in the population, there is higher probability that the solution will be removed from the population in next generations. There is no benefit if the solution generated in the mutation operator does not participate further in the reproduction. Figure 8 presents the behavior of mutation operators that are applied in typical way as it is presented in Section 1.3 (applied mutation operators are presented in Section 2.1.4) and Figure 9 presents the behavior of mutation operator, where the new population is created and computed. The solid line in Figure 8 and Figure 9 presents the best fitness value in the population, the dashed line presents the average fitness value of the population and the dots present the average fitness value obtained in the mutation operator in each iteration. The fitness value obtained in the mutation operator is not presented as a line because the mutation is applied with certain probability and in some iterations it is not applied at all and in some iterations it can be applied couple times where the averaged value is presented in this case. When the second population is created in mutation operation (Fig. 9), the generated solutions have better than average fitness value, so these solutions have higher probability to "survive" and to be selected for crossover operator. This can increase the diversification in the genetic algorithm.

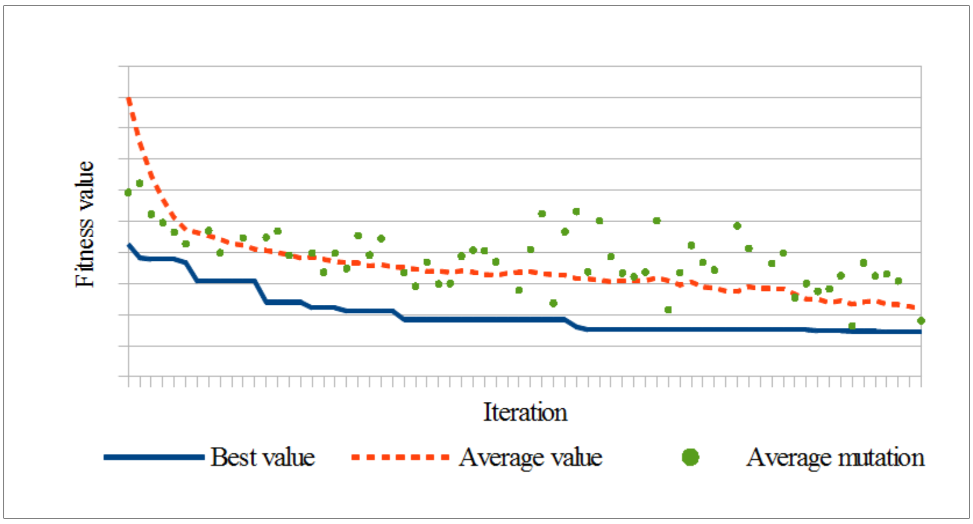

Fig. 8. The behavior of mutation operator applied in typical way

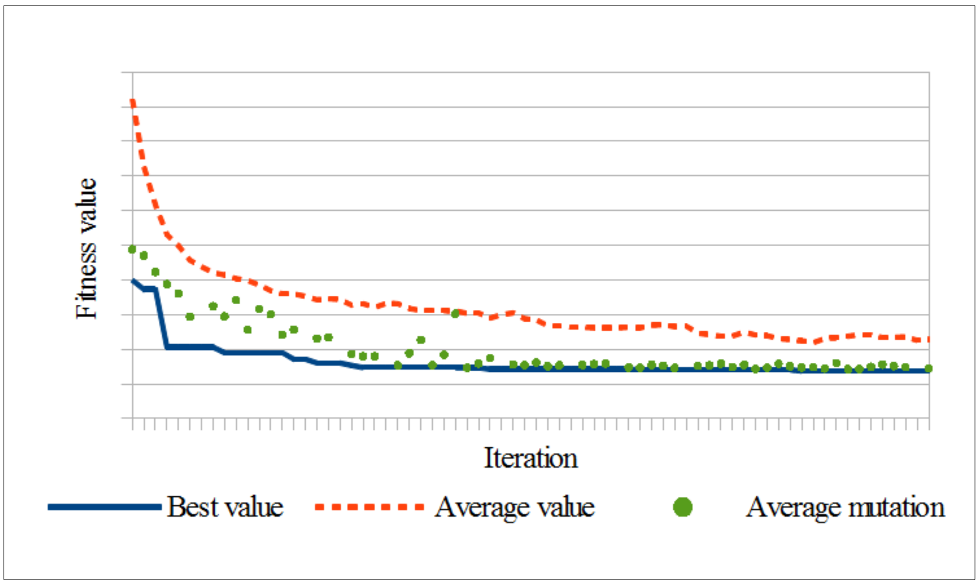

Fig. 9. The behavior of mutation operator when the second population in mutation operator is created and computed

The following pseudo-code represents the proposed genetic algorithm:

```
$Pop_1$ – initial population of size $PS_1$ //create a set of solutions using the feasible insertion method
while number of iterations without improvement < $IL_1$ and time < $TL_1$
    sort($Pop_1$) // sort individuals with a defined comparison function
    remove ($|Pop_1| - PS_1$) worst individuals from $Pop_1$
    for $i$=1...$PL_1$
        $x_{p11}$, $x_{p12}$ – select parents from $Pop_1$ by using the ranking method
        $x_{c11}$ = crossover($x_{p11}$, $x_{p12}$) // generate offspring
        $x_{c12}$ = crossover($x_{p12}$, $x_{p11}$) // generate offspring
        add $x_{c11}$ and $x_{c12}$ to $Pop_1$
        if random(0,1) < $MP$ // apply the mutation with probability $MP$
            ($x'_{m1}$, $N'_{m1}$) – create partial solution $x'_{m1}$ and node list $N'_{m1}$ by
                    mutating $x_{c11}$
            $Pop_2$ – create population of size $PS_2$ by inserting $N'_{m1}$ to $x'_{m1}$
            PROCESS $Pop_2$
            $x_{m1}$ = select the best individual from $Pop_2$
            add $x_{m1}$ to $Pop_1$
        end if
    end for
end while
best solution is $Pop_1$[1] // the best individual in the first population

PROCESS $Pop_2$
    while number of iterations without improvement < $IL_2$
        sort($Pop_2$) // sort individuals with a defined comparison function
        remove ($|Pop_2| - PS_2$) worst individuals from $Pop_2$
        for $j$=1...$PL_2$
            $x_{p21}$, $x_{p22}$ – select parent solutions from $Pop_2$
            $x_{c21}$ = crossover($x_{p21}$, $x_{p22}$)
            $x_{c22}$ = crossover($x_{p22}$, $x_{p21}$)
            add $x_{c21}$ and $x_{c22}$ to $Pop_2$
            if random(0,1) < $MP$
                $x_{m2}$ – generate offspring by mutating $x_{c21}$
                add $x_{m2}$ to $Pop_2$
            end if
        end for
    end while
```

Crossover and mutation operators are randomly selected from operators defined in the Sections 2.1.3 and 2.1.4. In the mutation operation the new population is created in two steps. Firstly, the mutation operator is applied to the solution x_{c11} to create a partial solution x'_{m1} with some routes left as well as the set of nodes N'_{m1} that need to be reinserted back. Then Pop_2 is created by copying the partial solution x'_{m1} and inserting N'_{m1} using a random insertion and computation of second population is invoked.

Computation of Pop_2 stops when the best solution is not improved in iterations IL_2. The stopping criterion in Pop_2 intentionally does not include the maximal time limit. The value IL_2 is chosen to be small to avoid redundant computation in Pop_2. The values used in the experimental evaluation (Section 3.3) are as follows: $PS_1 = 100$, $PL_1 = 10$, $IL_1 = 50$, $TL_1 = 5\text{min}$, $MP = 0.1$, $PS_2 = 20$, $IL_2 = 5$, $PL_2 = 2$.

2.1.3. Crossover operators

In the genetic algorithm new crossover operators that are based on insertion heuristics are proposed. However, apart from the insertion heuristics, the proposed crossover operators handle most of the negative aspects of the reviewed crossover operators in Section 1.5 and include another intensification approach.

The crossover operators, proposed here, are based on the idea of a large neighborhood search (LNS) heuristic presented in Section 1.2. The effectiveness of LNS depends on the degree of destruction, where, if only a small part is destroyed, LNS can have troubles in exploring the search space, or can be involved in the repeated re-optimization, if a very large space is destroyed (Pisinger and Ropke, 2009). It should be such destruction method which would explore the search space, where the global optimum is expected to be found.

The BCRC crossover, mentioned in Section 1.5, involves the destruction of the parent individuals to build an offspring, but the exploration depends on the route from the second parent individual. The nodes in the route from the second individual could be assigned depending on their time window constraint. Thus, it means that the removed nodes can have a low probability to change their positions in the solution. Other crossovers (SBX, RBX, LRX) convey the *union* of the solutions, where some parts from both individuals are combined with the intention to find a better solution. Such crossovers explore only a small neighborhood and only in the cases, where additional unassigned nodes are left during the recombination of parents.

Differently than the *union* crossover operators, the proposed crossovers are designed to preserve common parts of the two selected individuals. The common parts could be the nodes assigned to the same route, the nodes assigned to the route starting from the same depot, or the nodes that are related to the same type of cargo, etc. By removing the nodes that do not belong to the common parts of solutions, the common neighborhood of two solutions is identified. A size of the neighborhood is inversely proportional to the size of the common parts. If the initial individuals in a genetic algorithm are created in a stochastic way, by preserving the nodes that have common characteristic in both parents, the nodes will be preserved that more probably are optimally constructed than the other parts of the solution. Most probably, the nodes will be removed that prolong the overall path, where long paths can lead to a larger number of routes.

The target of the proposed crossover operators is to identify the common parts in the parent individuals, preserve it in the intermediate solution and reconstruct it in the offspring individual. Three different crossovers (*common nodes crossover*, *common arcs crossover* and *longest common sequence crossover*) are defined to increase the probability of convergence to the global optimum where each crossover produces an offspring by focusing on a different information obtained from parents. Crossover operators generate new solutions from the chosen parent solutions x_i and x_j. Each of them produces a single offspring partial solution x'_o and the set of extracted nodes N_{temp} from the parent solutions x_i and x_j. The offspring solution x_o is created by inserting all unassigned nodes N_{temp} to x'_o using the defined random insertion method.

The *common nodes crossover* (CNX) intersects the routes $R_i \in x_i$ and $R_j \in x_j$ according to the visited node sets N_i and N_j:

```
$N_{temp} = U_j$
$x'_o$ – offspring solution
for each route $r_s \in R_j$
    $R_{temp} = \varnothing$ // temporary set of routes
    for each node $n \in r_s$, $n \neq n_0$
        $r_t$= find route in $R_i$, where $n \in r_t$
        if $r_t$ not found
            add $n$ to $N_{temp}$
        else if $r_{t(tmp)} \in R_{temp}$
            assign node $n$ to $r_{t(tmp)}$
        else
            $r_{t(tmp)}$ = new route($\{n_0,n\}$)
            add $r_{t(tmp)}$ to $R_{temp}$
        end if
    $r_{best}$ = select a route with the maximal number of nodes visited from $R_{temp}$
    assign all nodes $n \in R_{temp} \setminus r_{best}$ to $N_{temp}$.
    add $r_{best}$ to $x'_o$
end for
create $x_o$ by inserting $\forall n \in N_{temp}$ into $x'_o$ using the defined random insertion
```

The first crossover examines all nodes in each route and groups them into partial routes according to the attendance in the routes from the opposite solution. The partial route with the maximum number of nodes is selected to preserve the path. All other partial routes are discarded by adding nodes to the unassigned node list N_{temp}. In the worst case the crossover operators have a complexity of O(k^2). This crossover still has one negative aspect: it can not be applied to the case when each of the parent solutions x_i and x_j have only one route. In this case a created offspring will always be equal to the parent solution.

VRP solution is the sequence of the nodes that need to be visited. Solutions varies depending on the order of nodes, i.e. by having a different sequence of nodes there are different travel path lengths in the solutions. So, the sequence of nodes is important characteristic that needs to be preserved in the offspring solutions. The next two crossovers (c*ommon arcs crossover* and *longest common sequence crossover*) are defined to handle the same sequence of the nodes in both parents.

The *common arcs crossover* (CAX) preserves arcs in the first parent solution, if the corresponding arcs exist in the second parent solution, where the corresponding arc has the same start and the same end. The common arcs crossover preserves the sequence between two nodes in the graph. The CAX operator intersects two sets of arcs $A_i \in x_i$ and $A_j \in x_j$. The complexity of this crossover is O(k), where k is the total number of nodes in the problem. The algorithm to find the common arcs in two parent solutions for new problem definition is as follows:

```
$N_{temp} = U_i$
$x'_o$ – offspring solution
for each arc $a_i \in A_i$
    if $a_i$ exist in $A_j$
        add $a_i$ to $x'_o$
    else
        add node $n_s \in a_i$ to $N_{temp}$, where $n_s$ is the starting node of arc $a_i$
    end if
end for
create $x_o$ by inserting $\forall n \in N_{temp}$ into $x'_o$ using the defined random insertion
```

Figure 10 represents the behavior of CNX and CAX crossover operators, where a) and b) are parent solutions, c) is the intermediate solution obtained by CNX, where the common nodes from both parents are displayed as gray circles, d) is the intermediate solution obtained by CAX, where the common arcs from both parents are displayed with solid arrows. Dotted circles show unassigned nodes N_{temp} that will be inserted back. Dotted lines in c) and d) are new arcs that connect the nodes according to their position in the first parent solution to form the route.

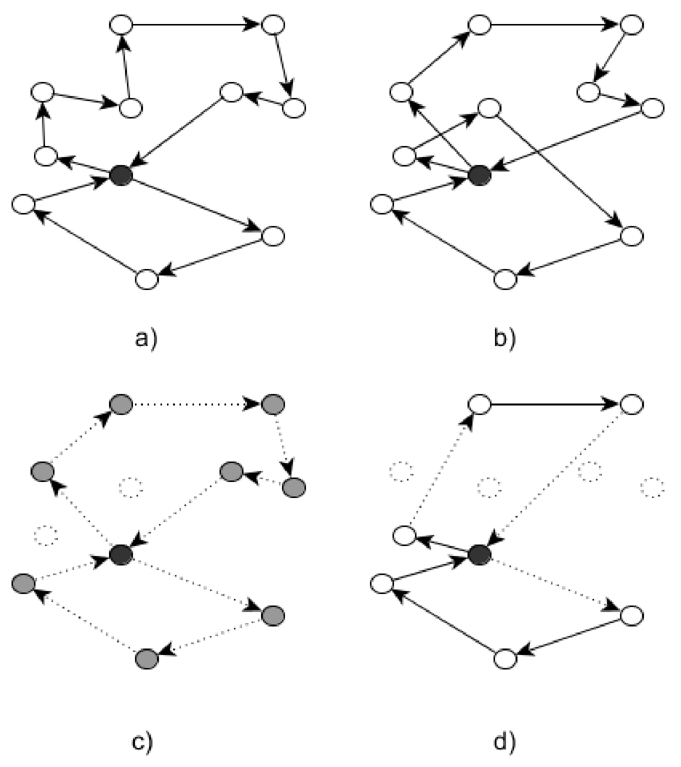

Fig. 10. Behavior of crossover operators: a) b) two parents; c) partial offspring obtained by the CNX; d) partial offspring obtained by the CAX

The *Longest common sequence crossover* (LCSX) is the third crossover operator, proposed in this thesis. It examines the two parent solutions by searching for the longest common sequences in all the routes. An example of the longest common sequence between two routes is displayed in Figure 11, where a) displays the first route with the indexed nodes in the route (literal string displays the indexed sequence); b) for all the nodes in the second route indexes are assigned according to the route in a); c) displays the longest common sequence solution example, where the solution is found by solving the *longest common increasing subsequence* (LCIS) (Schensted, 1961; Yang et al., 2005; Chan et al., 2007; Kutz et al., 2011) for the index line, identified in b).

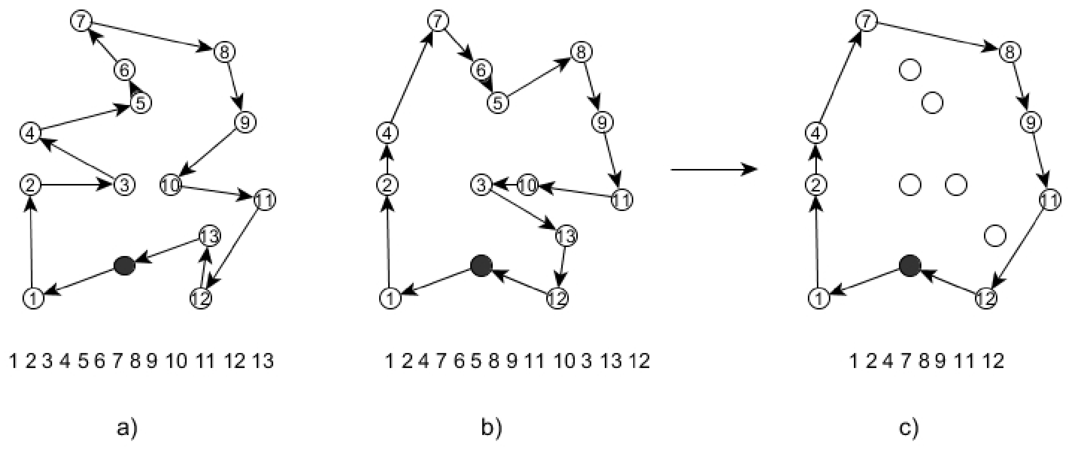

Fig. 11. The longest common increasing sequence between two routes

The longest common increasing sequence presented in c) is not the only one, there exist different longest common sequences that have the same length as in c). All the possible longest common sequences are as follows:

1 2 4 5 8 9 10 12
1 2 4 5 8 9 10 13
1 2 4 5 8 9 11 12
1 2 4 5 8 9 11 13
1 2 4 6 8 9 10 12
1 2 4 6 8 9 10 13
1 2 4 6 8 9 11 12
1 2 4 6 8 9 11 13
1 2 4 7 8 9 10 12
1 2 4 7 8 9 10 13
1 2 4 7 8 9 11 12
1 2 4 7 8 9 11 13

For LCSX, all the longest common sequences are identified and a single sequence is chosen randomly as the longest common sequence for the offspring. In LNS the parts of solution are destroyed by evaluating a single solution. In the LCSX crossover, some parts of solution are destroyed by evaluating the selected solution and another solution taken from the population.

In Figure 12, an example of finding common sequences among more than one route in a solution is presented. For each route r_i in the first individual, the routes in the second individual are identified that contain at least one node belonging to r_i (Figure 12, d)). Then the route with the largest number of common nodes is selected (Figure 12, e)) and the longest common sequence between routes in c) and e) is found. In e), the removed routes could also be evaluated for the longest common sequence, however, search for the longest common sequence in all of the routes increases the complexity of the crossover. To avoid extra complexity, the longest increasing sequence is identified for the routes with the largest number of common nodes. The same method is applied to all the routes in the first individual to get the intermediate solution x'_o (Figure 13):

```
$N_{temp} = U_i$
$x'_o$ – offspring solution
for each route $r_i \in R_i$
        $r_{int}$ = find the intersecting route $r_{int} \in x_j$ that has the largest number of common nodes
        SEQ = find the longest common sequence between the routes $r_i$ and $r_{int}$
        $r_{new} = \varnothing$
        for each node $n \in r_i$
                if n exist in the sequence SEQ
                        add the node n into $r_{new}$
                else
                        add n to $N_{temp}$
                end if
        end for
        if $|r_{new}| > 1$
                add $r_{new}$ to $x'_o$
        else
                $\forall n \in r_{new}$ add to $N_{temp}$
        end if
end for
create $x_o$ by inserting $\forall n \in N_{temp}$ into $x'_o$ using the defined random insertion
```

The CAX crossover preserves the sequence only for each the two subsequent nodes in the route, where the LCSX crossover preserves the longest common sequence between two solutions. In all three crossover operators, if the route has one node, it is removed and the node is added to the list of unassigned nodes. The complexity of LCSX is O(k^2) in the worst case including computation of the longest common sequence.

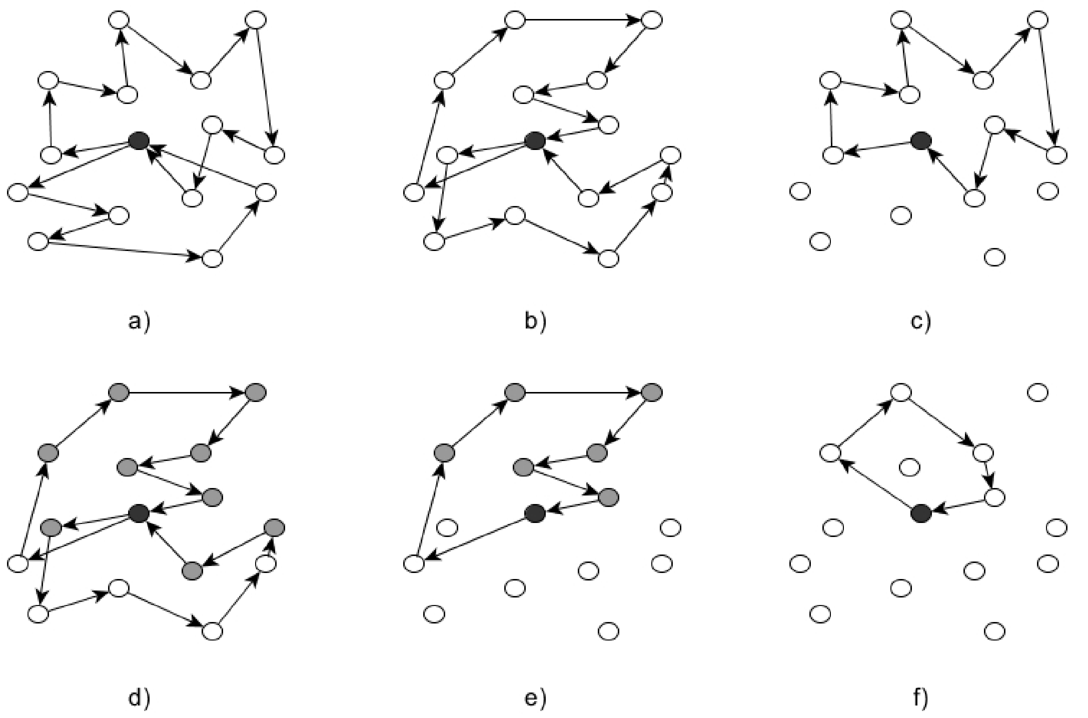

Fig. 12. Identification of the longest common sequence in all routes: a) and b) two parent solutions; c) the selected route from the first individual for evaluation; d) the routes in the second individual are identified that include the same nodes as in the c) selected route; e) the route with the largest number of common nodes is selected from the routes in d); f) the longest common sequence between the routes from c) and e) is identified

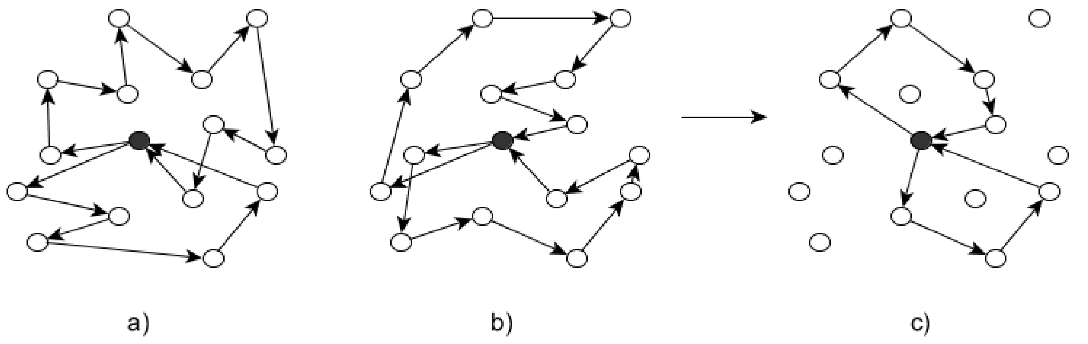

Fig. 13. The longest common sequence crossover: a) and b) two parent solutions; c) the intermediate solution found by the crossover

The proposed crossover operators preserve common parts in all routes. In most cases, the crossovers will create an intermediate offspring that has the same number of routes as in the parent solution. To minimize the number of routes, as it is defined in the objective, the number of routes should be reduced in the intermediate offspring by removing a randomly selected route. The last step of each proposed crossover is processed as follows:

1) while $f_v(x'_o) > f_v(x_i) - \delta$, remove the randomly selected route r_r from x'_o and insert $\forall n \in r_r$ to N_{temp};
2) create x_o by inserting $\forall n \in N_{temp}$ into x'_o using the defined random insertion.

In step 1), the value δ can have values from 0 to $f_v(x_i)–1$, where x_i is a parent solution participating in the crossover operation. If $\delta = 0$, the crossover will not try to minimize the number of routes. For the proposed crossover operators δ is a random value from the set $\{0,1\}$, if $f_v(x_i) > 1$, and $\delta = 0$ otherwise.

In Section 3.2, the new crossover operators are compared to other crossover operators, described in Section 1.5.

2.1.4. Mutation operators

The behavior of the proposed mutation operators are similar to the defined crossover operators, however, mutation operators deal only with a single solution x_i. The designed mutation operators extract a subset of nodes from the solution in the defined ways and reinsert them back by applying a random insertion. By extracting a set of nodes we aim to preserve one part of the solution and reorganize the other one. A set of mutation operators that are applied by selecting one of them randomly is defined.

The first mutation operator selects a node set for extraction randomly with the limit of $0.5z|N|$, where z is a random number in the range (0, 1). The complexity of random extraction is $O(k)$.

The second mutation operator picks up a random node n_r from x_i and extracts the set of nodes closest to n_r by minimizing the distance function $l(n_r, n_i)$, where $\forall n_i \in R_i$:

```
$x'_o = x_i$
$N_{temp} = U'_o$
$U'_o = \varnothing$
$n_r$ = select a random node from $x'_o$
extract $n_r$ from $x'_o$
add $n_r$ to $N_{temp}$
while limit reached is not reached
    $n_{ri}$ = find the nearest node to $n_r$ in $x'_o$
    extract $n_{ri}$ from $x'_o$
    add $n_{ri}$ to $N_{temp}$
end while
create $x_o$ by inserting $\forall n \in N_{temp}$ into $x'_o$ using the defined random insertion
```

The number of extracted nodes is limited to $0.5z|N|$, where z is a random number in the range (0, 1). The complexity of the second mutation operator is $O(k^2)$.

The third mutation operator extracts random routes with the limit of $0.5z|R_i|$ routes, where $R_i \in x_i$. The complexity of the described method is $O(k)$.

The fourth mutation operator extracts nodes with the longest detour. The search selects nodes with maximal values of the function $l_r(n_r) = l(n_{r-1}, n_r) + l(n_r, n_{r+1}) - l(n_{r-1}, n_{r+1})$. The number of extracted nodes is limited to $0.5z|N|$, where z is a random number in the range (0, 1). In the worst case the complexity of the fourth mutation operator is $O(k^2)$. The fourth mutation operator is combined with other mutation operators, where initially the first mutation operator is applied and then the fourth mutation operator is applied with probability 0.1.

The fifth mutation operator searches for nodes visited around the same time. This mutation operator is similar to the second mutation operator. At first, a random node n_r is selected, afterwards other nodes are selected by minimizing the function $t_r(n_i)= |t_a(n_r) - t_a(n_i)|$, where $\forall n_i \in R_i, R_i \in x_i$. The fifth mutation operator is applied when time constraints are defined in the problem. The complexity of the fifth mutation operator is $O(k^2)$.

In Figure 14, the behavior of mutation operators is presented, where a filled circle shows a depot, empty circles show visited nodes and dotted circles show the extracted nodes.

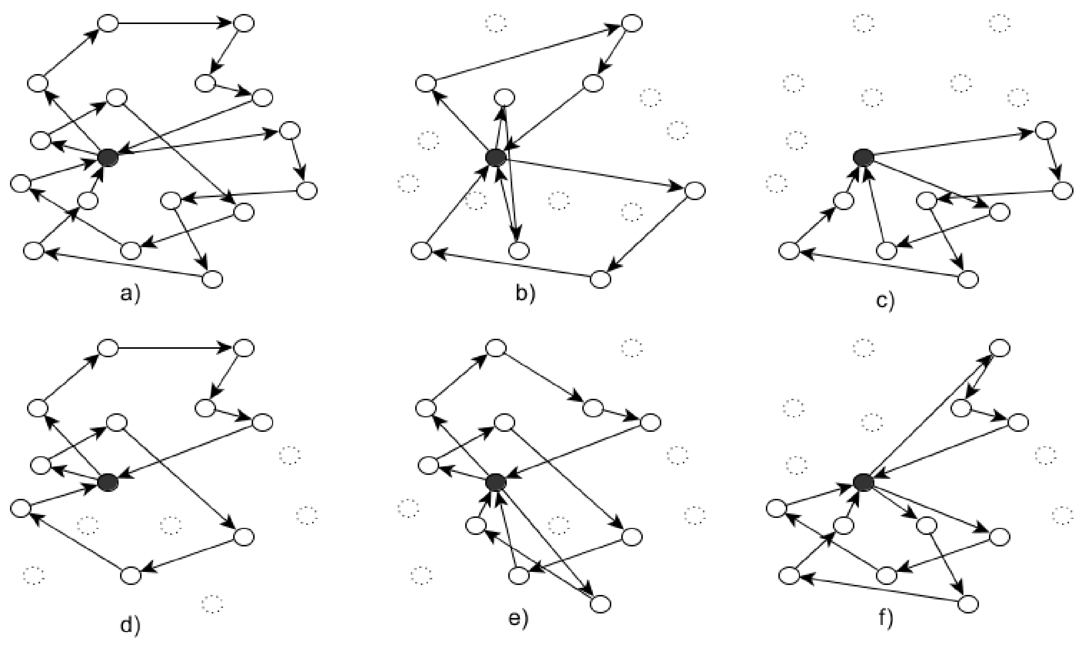

Fig. 14. Mutation operators: a) initial solution; other cases show the nodes extracted in b) the first, c) the second, d) the third, e) the fourth, and f) the fifth mutation operators

2.2. Genetic operators for rich vehicle routing problem

2.2.1. Rich vehicle routing problem

The typical VRP can be extended by adding additional constraints and other parameters to the problem. The MDVRP includes additional depot nodes and CVRP includes load capacity limitation for a vehicle. VRP with time windows (VRPTW) is an extension, where time window constraints are added.

The time window constraint defines a time frame in which a customer can be serviced, i.e. loading or unloading of a vehicle. A vehicle may arrive earlier, but it must wait until the start of the service is possible. The VRP can be extended with some additional constraints, like driver working hours, time, required for a driver to take a rest, etc. Similarly, depending on additional parameters, other variants of VRP are defined. In (Yeun et al., 2008) particular mathematical formulations can be found for each VRP, VRPTW, VRPPD, CVRP problem, where each formulation is based on a customer set, represented as nodes in a graph. Jih and Hsu (2004) have proposed the problem definition for PDPTW, based on transportation requests as tasks to be completed.

Although in academic literature specific problems are investigated, there are attempts to generalize vehicle routing problem. The aim of this research is to create the algorithm for the general VRP: rich vehicle routing problem. The first attempt to define rich vehicle routing problem can be found in (Toth and Vigo, 2002). The paper (Hasle and Kloster, 2007) refers to this problem as industrial vehicle problem. In (Goel and Gruhn, 2008) it is called the general vehicle routing problem. Usually rich vehicle routing problem is a description of different information and constraints reflecting real world situation. A summary of real-world constraints is described in (Drexl, 2012) where the fundamental activity to be planned is treated as request. To generalize a VRP, it can be divided into the following components:

- *data*, used in the problem;
- *tasks*, defined to be accomplished;
- *constraints* that should be satisfied;
- *objective* of the problem.

Data definition includes the graph $G = (N, E)$, which consists of the nodes N and edges E. The data definition also includes a set of vehicles

$V = \{v_1, \ldots, v_t\}$. The set of nodes can be divided into subsets of a) N_d – depots, b) N_c – customers, c) N_o – other nodes that can be divided into rest areas, gasoline stations, etc. For data definition, a start position is assigned to each vehicle v_i, where the initial node can be marked as n_i^{init}. Additional data, like drivers and their properties, vehicle parameters or types of the goods, can be defined within the problem.

Tasks, similarly as in (Jih and Hsu, 2004; Hasle and Kloster, 2007), define a set of targets to be achieved. Let us define a set $M = \{m_1, \ldots, m_q\}$ as a set of q requests and $T = \{t_1, \ldots, t_k\}$ as a set of k tasks to complete requests. Each request m_i can be expressed via a set of tasks $m_i = \{t_{i1}, t_{i2},...\}$, where $\forall t_{ij} \in T$, $|m_i| > 0$, $\forall m_i \in M$, $m_1 \cap \ldots \cap m_q = \varnothing$ and $m_1 \cup \ldots \cup m_q = T$. The main difference between the request and task is that the task can be processed one at a time by a single vehicle and the requests may be processed in parallel. The task can have other smaller subtasks, in such a way granularity increases, however, for VRP, the task does not require to be split to smaller tasks, if it means "to be processed one at a time by the vehicle". In the VRP, each task t_i is defined as $t_i = (n_i^{start}, n_i^{end})$, where the node $n_i^{start} \in N$ is a start node of the task t_i and $n_i^{end} \in N$ is the end node. To complete the task t_i, at first, a vehicle needs to arrive to the node n_i^{start} to start service, and then to complete service at the node n_i^{end}.

For VRP that deal with a delivery of cargo, the request can be defined as $m_i = \{t_i^+, t_i^-\}$, where m_i is a request to deliver cargo from one place to another and to complete it tasks t_i^+ (to load cargo at a specific place) and t_i^- (to unload cargo at a specific place) have to be performed. The properties of cargo are defined for each request. Let us define a function $w(m_i)$ that evaluates the cargo capacity value $w_i = w(m_i)$. Tasks in the delivery problem can be defined as $t_i = (n_i^{start}, n_i^{end}, w_i)$ for loading/unloading of w_i at the node $n_i^{start} = n_i^{end}$. Usually VRP defines the return to the depot tasks $T^{end} = \{t_1^{end}, \ldots, t_t^{end}\} \subseteq M$, $\forall t_{vi} \in T$.

An example of the task that starts and ends at different places could be found in the taxi problem, where a service of each customer starts at pick-up place and ends at the destination place.

The VRP target is to complete tasks by using vehicles. Let us define a single solution of the VRP as $x = \{s_1, \ldots, s_t\}$, where $\forall s_j = (t_{j1}, t_{j2}, \ldots)$, $\forall t_{ji} \in T$, $s_1 \cap \ldots \cap s_t = \varnothing$ and $s_1 \cup \ldots \cup s_t = T$. $\forall s_j$ defines a sequence of tasks assigned to the vehicle $v_j \in V$ and $|x| \leq |V|$. Let us define a function $F_m(x)$ that evaluates the solution x for incompleteness of requests and $f_m(x)$ that evaluates a single request for task incompleteness.

$$F_m(x) = \sum_{m \in M} f_m(m, x)$$

$$f_m(m, x) = \sum_{t \in m} f_t(x, t)$$

$$f_t(x, t) = \begin{cases} 0 & \text{if the task } t \text{ is completed in the solution } x \\ 1 & \text{otherwise} \end{cases}$$

All the requests are completed, if $F_m(x) = 0$.

Constraints define restrictions to the problem that usually reflect real life situations. Let us define a set of constraints C, where $c \in C$ defines a single constraint. The constraints can be defined for a task (i.e. time window), for a vehicle (i.e. not exceed capacity), for a cargo, etc. One of the constraint of the delivery problem (VRPPD) is that t_i^+ needs to be completed before t_i^- or return to the depot task should be completed after all the other tasks. So, the constraint can define the order of tasks. There can be also constraints defined for a driver, i.e. time required for a driver to take a rest, or constraint not to empty the fuel tank. Let us define a function $F_c(x)$ that evaluates violation of constraints in the solution x, and $f_c(x)$ that evaluates violation of the single constraint $c \in C$:

$$F_c(x) = \sum_{c \in C} f_c(x)$$

$$f_c(x) = \begin{cases} 0 & \text{if the constraint } c \text{ is satisfied} \\ z, \text{ where } z \in \mathbb{R}, z > 0 & \text{otherwise} \end{cases}$$

The solution x does not violate any constraint, if $F_c(x) = 0$.

Objective. The objective of the typical VRP is to minimize the number of the used vehicles and then to minimize the length of the total travel path. So, the objective is, at first, to minimize the function $f_v(x)$, then $f_d(x)$, in addition, the equalities $F_c(x) = 0$ and $F_m(x) = 0$ need to be satisfied:

$$f_v(x) = \sum_{j=1}^{|x|} f_a(s_j), \forall s_j \in x$$

$$f_d(x) = \sum_{j=1}^{|x|} d_l(s_j), \forall s_j \in x$$

$$f_a(s_j) = \begin{cases} 1 & \text{if } d_l(s_j) > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$d_l(s_j) = \sum_{i=1}^{|s_j|} l\left(n_{j_i}^0, t_{j_i}\right), \text{where } \forall t_{j_i} \in s_j, n_{j_i}^0 = n_{j_{i-1}}^{end}, n_{j_0}^{end} = n_j^{init}$$

$$l(n_i, t_j) = l\left(n_i, n_j^{start}\right) + l\left(n_j^{start}, n_j^{end}\right)$$

$$l(n_i, n_j) - \text{distance between the nodes } n_i \text{ and } n_j$$

Different objectives could include different minimization/maximization functions, i.e. a real life problem could be defined, where the feasible solution that completes all the tasks is not possible. The objective of such a problem could be to find a solution, where the maximum number of requests is completed.

2.2.2. Crossover and mutation operators for RVRP

In Section 2.2.1 we have defined the solution of the problem as a set of sequences constructed from the tasks. Such a definition differs from a widely used definition for VRP, where the solution is defined as a sequence of nodes visited by separate vehicles, where each sequence is called a route. Tasks are accomplished “traveling by the vehicle in the graph”. The sequence of nodes can be expressed via a sequence of tasks, where the solution does not have duplicated tasks. The sequence of tasks can be expressed via the sequence of nodes, however, the same node can be visited a couple of times per solution, if a couple of different tasks include it. Further on the methods and algorithms will be expressed by a sequence of tasks instead of the original expression – route of nodes. For a better explanation, sequence of tasks will be called a route of tasks.

Solutions within a proposed genetic algorithm are treated to be feasible, if all the constraints are satisfied. However, the solution can be incomplete – not all the tasks are completed in the constructed solution. It requires to extend the objective to handle this approach: to find the solution that accomplishes a higher number of tasks. The solution x for the RVRP is defined as follows:

$$x = (R = \{r_1, \dots, r_t\}, T_u = \{t_1, \dots, t_u\})$$

Let us express the route of tasks $r_r \in x$ as a graph $G_r^T = (T_r, A_r)$, where the set $T_r = \{t_r^{init}, t_{r1}, \dots, t_r^{end}\}$ defines the set of tasks assigned to the route, and t_r^{init} represents the start of the route at the node n_r^{init}. Set T_u defines the set of unhandled tasks in the solution x, where $\{T_1, \dots, T_t\} \cup T_u = T$ and $\{T_1, \dots, T_t\} \cap T_u = \varnothing$. Expression of route via tasks eliminates the possibility of duplicate entries, where the expression of a route via nodes could give the same node, visited a couple of times, if the same node is a part of a couple of different tasks. In Figure 15 the solution of nodes and related solution of tasks are presented. The arcs from the set A_r connect the tasks: $\forall a_{ri} \in A_r$, $a_{ri} = (t_{ri}, t_{r(i+1)})$, where t_{ri} is the start of the arc and $t_{r(i+1)}$ is the end. Insertion of the new task t_m into the arc a_{ri}, means to:

1) remove the arc a_{ri} from the set A_r;
2) add the task t_m to T_r;

3) add two new arcs (t_{ri}, t_m) and (t_m, $t_{r(i+1)}$) to the set A_r.

An empty route (vehicle without tasks assigned) is still a graph with $T_r = \{t_r^{init}, t_r^{end}\}$ and $A_r = \{(t_r^{init}, t_r^{end})\}$ (Figure 15). The insertion process can be split into the following parts and intensification can be applied in both of them:

- select a task for insertion;
- select a place for insertion.

The following approaches can be used for implementing the insertion heuristic:

- Choose a random task and then search for the best arc to insert in. A random task selection is a stochastic approach that can be used to choose a task for insertion. This method does not affect overall intensification and corresponds to the general idea of the genetic algorithm being a stochastic approach. The usage of the minimization function then can be applied to arc selection.
- Choose a random arc and then search for the unassigned task for insertion. The usage of the minimization function can be applied to node selection.
- Search for a task and arc at the same time by evaluating the minimization function. The minimization function could also include evaluation of the vehicle and additional information that could help to identify the best task and the best place for the next insertion.

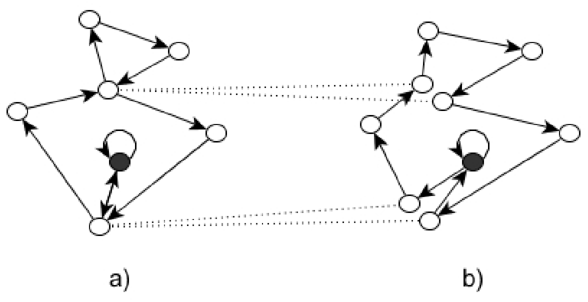

Fig. 15. Solution of different expressions: a) solution of nodes; b) solution of tasks

To select a place for the task insertion means to select an arc from the set of A_{all} (all the arcs in all the routes in the partial solution x'). For a feasible insertion, we have to evaluate the violation of all the constraints. Let us have a function $F_c(a_i, t_m)$ that checks the violation of all constraints, while inserting the task t_m into the arc $a_i \in x'$, $x^{i,m}$ is the solution x' with the task t_m inserted into the arc a_i:

$$F_c(a_i, t_m) = \Delta F_c^{i,m}(x') = F_c\left(x'^{i,m}\right) - F_c(x')$$
$$F_c(a_i, t_m) = \sum_{c \in C} f_c(a_i, t_m)$$

The function $f_c(a_i, t_m)$ checks the violation of single constraint, while inserting the task t_m into the arc a_i. The insertion of a task does not increase the constraint violation, if $F_c(a_i, t_m) = 0$. To follow the objective of the problem, the arc needs to be selected by minimizing the functions $f_v(a_i, t_m)$ and $f_d(a_i, t_m)$ that evaluate the difference of objective functions, while inserting the task t_m into the arc a_i. For the VRP, there exist two functions for task insertion. At first, the difference of the route number is evaluated and, in the second function, the difference of the route length is evaluated:

$$f_v(a_i, t_m) = \Delta f_v^{i,m}(x') = f_v\left(x'^{i,m}\right) - f_v(x')$$
$$f_d(a_i, t_m) = \Delta f_d^{i,m}(x') = f_d\left(x'^{i,m}\right) - f_d(x')$$

The overall insertion process with the feasibility check is similar to node insertion process defined in Section 2.1.1 and presented in Figure 6.

Crossovers defined in Section 2.1.3 can be applied to the problem defined via tasks. Overall idea of the crossover is not changed. The defined crossovers search for common parts in two parent solutions. Instead of searching for the nodes, the crossovers will search for the common parts in the solution expressed via tasks: CNX will search for the common set of tasks handled in the routes of parent solutions; CAX will search for common arcs that connect the same tasks; LCSX will search for the longest common sequence of tasks in parent solutions. Identified parts are preserved in the intermediate offspring solution and all unhandled tasks are inserted in to the solution by using the defined construction heuristic in a reconstruction phase. A random insertion heuristic is chosen for a reconstruction to preserve the stochastic approach of the genetic algorithm. A task is chosen randomly from the list of unassigned tasks and inserted into the route by evaluating the feasibility and minimizing the insertion cost functions $f_v(a_i, t_m)$ and $f_d(a_i, t_m)$. If the crossover operators search for the common parts in the solutions to intensify the search, the random task insertion involves a diversification in a population.

Mutation operators proposed in Section 2.1.4 can also be applied for problem expressed via tasks. By applying defined mutation operators tasks will be removed from solution and reinserted back to create a new one. If crossover operators involve intensification by preserving common parts between two solutions, the mutation operator creates the new solution by recombining single one and may destroy the part that is probably correct one. The second population usage in mutation operators proposed in Section 2.1.2 increases probability that in mutation operator generated solutions will be competitive in the population.

The defined crossover and mutation operators and insertion method can be applied to any problem that can be described in the form as defined in Section 2.2.1. Depending on the defined constraints, random insertion method needs to be adjusted to preserve feasible solutions, i.e. for periodic vehicle routing problem multiple tasks will be added for each visit to the customer, constraints that the tasks will be completed per defined period need to be preserved. For problems, where refueling is important, additional travel time check for refueling needs to be taken into account. Each refueling will take time to travel detour and it is important to refuel not too often. Refueling will add additional weight to arc and it needs to be recalculated after each insertion, however this can be optimized with precomputing described in Section 2.1.1. Problems, where the target is the visit of the arc (i.e., road cleaning), can also be defined in the new form. The start and the end of the task in such problem will coincide with the start and end of the target arc. The problem expressed in the new form can be further solved by the genetic algorithm with proposed new genetic operators.

2.3. Parallel bi-directional shortest path algorithm

As we have already mentioned, Dijkstra's algorithm was developed to calculate the shortest path. As we have seen, the bidirectional algorithm and many other Dijkstra's algorithm modifications reviewed in Section 1.6 produces nearly the shortest path. However, sometimes the error of calculation can be significant. To achieve accurate results by means of the bidirectional algorithm, we combine both stopping criteria. The algorithm is extended with a set Z of possible answers and with a finish phase. If the visited node of contrariwise computation is found for the first time (found node n_z^r, where $n_z^r \in N_S^r$ and $n_z^r \in N_D^r$), then the algorithm enters the finish phase. In this phase, the process continues without appending the priority queue with new labeled nodes, i.e., without the inner loop. After completing these additional steps, we get alternative nodes $n_z^r \in N_Z^r$, where $N_z^r = (N_S^r \cup Q_S) \cap (N_S^r \cup Q_D)$. During such process other alternative routes are found and added to the set Z. Thus, after finishing such process, the set $Z = \{P(n_z^r) \cup P^{|}(n_z^r), \forall n_z^r \in N_Z^r\}$. The calculation ends, when the priority queues Q_S and Q_D are empty. Then the shortest path can be extracted from the set Z of possible answers. The following pseudo-code implements the proposed algorithm.

```
***procedure*** *biDijkstra*($G^r = (N^r, E^r), n_s^r, n_d^r$)
        ***boolean*** *finish* = ***false***
        $Z = \varnothing$ // possible answers list
        …
        ***while*** $Q_S$ *is not empty* ***and*** $Q_D$ *is not empty* // outer loop
                // PERFORM FORWARD SEARCH
                $n_u^r = Q_S$.*extractMin*()
                $N_S^r$.*addNode*($n_u^r$)
                ***if*** $n_u^r \in N_S^r$ ***and*** $n_u^r \in N_D^r$ // stopping criterion
                        *finish* = **true** // set finish phase
                        *Z.addAnswer*()
                ***end if***
                ***if*** *finish* = ***false***
                        ***for*** *each* $n_v^r$ *adjacent to* $n_u^r$ // inner loop
                        …
                        ***end for***
                ***end if***
                // PERFORM BACKWARD SEARCH
                …
        ***end while***
        *Z.findShortest*() // get shortest path from list
***end***
```

We get $(N_S^r \cup Q_S) \cap (N_D^r \cup Q_D)$ = {E, B, C} according to this termination condition, where the example, presented in Figure 5, is analyzed. We get the set Z = {P(E) ∪ $P^|$(E), P(B) ∪ $P^|$(B), P(C) ∪ $P^|$(C)} and the obtained results |P(E) ∪ $P^|$(E)| = 100, |P(B) ∪ $P^|$(B)| = 70 and | P(C) ∪ $P^|$(C) | = 70. By selecting the smallest of these values we get the shortest path ABCD. In the first example, when solving the shortest path problem in the same way, we get the shortest path ADB. The modified bidirectional Dijkstra's algorithm finds the same path as the standard Dijkstra's algorithm. The bidirectional Dijkstra's algorithm is 2 times faster than the standard one (Goldberg et al., 2006; Berrettini et al., 2009). The modified bidirectional algorithm lasts slightly longer because of the finish phase. However, the algorithm is still faster than the standard Dijkstra's algorithm.

Parallel algorithm. The scheme of the parallel algorithm, proposed in this research, is essentially based on the modern multi-core technology. The modified bidirectional Dijkstra's algorithm is transformed into a parallel form, based on a shared memory technology. For this method, an ordinary two-core processor is required. Thus, the two processes may look for the shortest path from the start and end nodes at the same time. Like in the modified bidirectional Dijkstra's algorithm, each process must access data of the contrariwise computation to check for stopping criteria. If the stopping criterion is reached, then the process proceeds to the finish phase and “tells” the opposite process to do the same. Both processes fill out the set *Z* of possible answers with the shortest paths found. The processes are stopped, when the priority queues Q_S and Q_D become empty. Then the main process selects the shortest path from set *Z*.

```
procedure parallelDijkstra($G^r = (N^r, E^r), n_s^r, n_d^r$)
    $Z = \varnothing$ // possible answers list
    $N_S^r = \varnothing, N_D^r = \varnothing$ – empty sets of visited nodes
    …
    // call backward process
    start second thread → calcBackwardsDijkstra($G^r, n_d^r, N_S^r, N_D^r, Z$)
    boolean finish = false
    $Q_S = \varnothing$ // labeled nodes
    $Q_S$.insert($n_s^r$,0)
    while $Q_S$ is not empty // outer loop
        $n_u^r = Q_S$.extractMin()
        $N_S^r$.addNode($n_u^r$)
        if $n_u^r \in N_S^r$ and $n_u^r \in N_D^r$ // stopping criterion
            finish = true // set finish phase
            Z.addAnswer()
        end
        if finish = false
            for each $n_v^r$ adjacent to $n_u^r$ // inner loop
                …
            end
        end
    end
    waitSecondThread() //wait for second thread to finish
    Z.findShortest()
end
```

```
procedure $calcBackwardsDijkstra(G^r, n_d^r, N_S^r, N_D^r, Z)$
    boolean finish = false
    $Q_D = \varnothing$ // labeled nodes
    $Q_D.insert(n_d^r, 0)$
    while $Q_D$ is not empty // outer loop
        $n_u^r = Q_D.extractMin()$
        $N_D^r.addNode(n_u^r)$
        if $n_u^r \in N_S^r$ and $n_u^r \in N_D^r$ // stopping criterion
            finish = true // set finish phase
            $Z.addAnswer()$
        end
        if finish = false
            for each $n_v^r$ adjacent to $n_u^r$ // inner loop
                …
            end
        end
    end
end
```

Since this algorithm is based on parallel shared memory computing, it allows avoiding the additional data transfer, which is necessary in the distributed memory parallel technology. However, the problem of sharing data in memory needs to be solved, because the access to shared data cannot be achieved in an uncontrolled fashion. In this thesis, we have already mentioned about the *transactional memory* technology by which this problem can be solved. However, this problem can be solved also by means of other technologies, for example, *mutual exclusion* (*mutex*) directives. Such directives in the program can ensure that only one process is executing an operation protected by the *mutex* object. In this research we do not explicitly deal with the efficiency of the mentioned technologies or other similar technologies. However, no matter which technology is chosen, we need to identify certain parts of the algorithm, which cannot be carried out simultaneously by two separate processes. By identifying those areas of the algorithm we can guarantee that the two processes at the same time will not operate the data at the same memory location. In the proposed algorithm, such areas are the sets

N_S^r, N_D^r and Z. The parts of the algorithm in which these two sets used are either read or modified are as follows:

```
$N_S^r.addNode(n_u^r)$.
…
$N_D^r.addNode(n_u^r)$
…
if $n_u^r \in N_S^r$ and $n_u^r \in N_D^r$ // stopping criterion
…
$Z.addAnswer()$
```

Those identified parts of the algorithm need to be synchronized so that only one process at a time could perform the operation, and one process at a time would be blocked until another process ends the transaction. However, the other part of the code will run in parallel.

The main disadvantage of this algorithm is that it cannot be adapted to a larger number of processes, when a single path is calculated. However, an additional process will be started and destroyed only once and will take the most time during the calculation. This allows us to avoid a number of additional delays that occur in the start-up and destruction of an additional process. The mentioned approach can be adapted to calculate the shortest-path between more than two places in the graph at the same time or even forward and backward at the same time.

In Section 3.4, parallel bidirectional Dijkstra's algorithm is experimentally evaluated.

2.4. Summary

In order to keep solutions in the feasible search space, we propose a genetic algorithm that is based on a random insertion heuristics. The random insertion heuristic is considered to preserve a stochastic characteristic of the genetic algorithm, and to generate solutions in the feasible space by checking compliance to the defined constraints in the insertion process. Precomputation scheme is proposed for speed-up evaluation of constraints in insertion heuristic. Infeasibility is still allowed in the proposed algorithm because the random insertion approach can create infeasible initial solutions in a highly constrained problem. The defined GA individual includes feasible partial routes and a set of customers that were not serviced due to constraint violation. The novelty of the proposed approach is the usage of random insertion heuristics in combination with the proposed crossover and mutation operators. Differently from other genetic algorithms, the proposed crossover operators do not construct the offspring directly, but by evaluating information from previous generation, identify those parts of solutions that should be preserved for the next generation and weak parts that should be reconstructed. The crossover and mutation operators are defined to identify those weak parts of the solution. The second population is used in the mutation process, where the second population increases the probability that the solution, obtained in the mutation process, will survive in the first population, and increase the probability to find the global optimum. In contrast to other approaches, the proposed algorithm does not involve additional local search methods to improve the solution; therefore it does not depend on the local search limitations and can be easily extended with additional constraints.

The rich vehicle routing problem definition consisting of the tasks to be completed. Node and arc routing problems can be transformed to the proposed general formulation of task problem. The proposed operators are applicable to the defined RVRP, where the crossover operators intensify the search by identifying the common sequence of tasks in parent solutions.

A parallel version of Dijkstra's algorithm is proposed for searching for the shortest paths in a road graph. The algorithm can be used for recalculation of the shortest paths between two places in road network while executing the genetic algorithm to optimize routes. The recalculation may be necessary if the additional data involves changes in the path (i.e. because of traffic jams the travel time significantly increases). The stopping criterion is defined for bidirectional Dijkstra's algorithm that prevents of inaccuracies in the shortest path search. The parallel version of bidirectional Dijkstra's algorithm is proposed for speed-up the search. The proposed algorithm is based on a transactional memory to avoid large data synchronization issues.

The results of this chapter have been published in (Vaira and Kurasova, 2010; Vaira and Kurasova, 2011; Vaira and Kurasova, 2013; Vaira and Kurasova, 2013a; Vaira and Kurasova, 2014).

Chapter 3
Experimental researches

This chapter presents the results of experimental evaluation. **Section 3.1** describes VRP data sets used for evaluation of the genetic algorithm. **Section 3.2** provides experimental evaluation of the proposed crossover operators. Two different cases were evaluated: with mutation operator applied and without it. **Section 3.3** provides results of experimental evaluation of the genetic algorithm scheme. In **Section 3.4** parallel bidirectional Dijkstra's algorithm for the shortest path calculation is evaluated. **Section 3.5** summarizes experimental researches.

3.1. Data sets used for experimental evaluation

3.1.1. Solomon problem instances

The first data set includes the well-known Solomon instances of the VRPTW, where all instances have 100 customers, distributed over the geographical area (Solomon, 1987). The Solomon problem consists of 6 different problem sets R1, R2, C1, C2, RC1, RC2, where all 56 VRPTW instances are categorized as:

- set C (C1, C2) – nodes (customers) are located in geographical clusters;
- set R (R1, R2) – nodes (customers) are randomly distributed over the geographical area;
- set RC (RC1, RC2) – some customers are randomly distributed and some customers are located in clusters.

Problem instances are also split into Class 1 (R1, C1, RC1) and Class 2 (R2, C2, RC2), where problem instances in Class 1 have a small vehicle capacity and narrow time windows and problem instances in Class 2 have a large vehicle capacity and large time windows. Each Solomon problem instance defines the central depot, the maximum vehicle number, limits of vehicle capacity, demands for each node and also the maximum travel time for a single vehicle (Figure 16).

```
VEHICLE
NUMBER     CAPACITY
25         200

CUSTOMER
CUST NO.   XCOORD.   YCOORD.   DEMAND   READY TIME   DUE DATE   SERVICE TIME
0          40        50        0        0            1236       0
1          45        68        10       912          967        90
…..
100        55        85        20       647          726        90
```

Fig. 16. Structure of data file of Solomon problem instance

For the problem instances the Euclidean distance between two nodes is treated as the shortest path value. For computation the shortest path distance value is also treated as travel time value.

3.1.2. Li and Lim problem instances

The second data set is defined for the VRPPD (Li and Lim, 2003). VRPPD instances LC1, LC2, LR1, LR2, LRC1, LRC2 are generated from Solomon problem sets C1, C2, R1, R2, RC1, RC2 respectively, described in Section 3.1.1. Problem instances have 100 customers. VRPPD instances include the central depot, time window constraints, pick-up and delivery nodes and the maximal travel time for a single vehicle (Figure 17).

Similarly as for VRPTW instances, for the VRPPD problem the Euclidean distance between two nodes is treated as the shortest path value and the same value is also treated as travel time value.

NUMBER OF VEHICLES				CAPACITY				
25				200				
CUST NO.	X	Y	DEMAND	EARLIEST PICKUP/ DELIVERY TIME	LATEST PICKUP/ DELIVERY TIME	SERVICE TIME	PICKUP (index to sibling)	DELIVERY (index to sibling)
0	35	35	0	0	230	0		
1	41	49	-20	133	198	10	65	0
2	35	17	7	22	87	10	0	55
….								

Fig. 17. Structure of data file of VRPPD problem instance

3.2. Evaluation of crossover operators

The crossover operators proposed in Section 2.1.3 are implemented with the Java programming language in order to compare it with other crossover operators. The other crossover operators (BCRC, RBX, LRX) described in Section 1.5, are also implemented for comparison. For these crossover operators that use insertion heuristics, the construction of solutions is the same as in the algorithm definition. For comparison of crossovers, other parts of the genetic algorithm are common:

- The initialization of the population is performed by randomly selecting nodes for insertion and inserting them by evaluating the feasibility and minimizing the cost. The same population is used for a single experiment with all the crossover operators. The population size is equal to 100.
- For evolution strategy, the *k-tournament selection* of the size $k = 2$ is chosen. In each iteration, 10 new offsprings are created.
- Mutation. To identify the features of a crossover better two groups of experiments are carried out. The first group of experiments does not involve the mutation operator. In the second group of experiments, the mutation operator is applied. The mutation operator used extracts $0.5z$

nodes from the solution and reconstructs the solution by the same reconstruction method that is used in the crossover, z is the random value within the range (0,1). Mutation is applied with the probability 0.15.

- Computation is stopped, when the best solutions are not improved for 300 iterations or when the maximum computation time (5 minutes) is reached.

The experiments are carried out using the Solomon problem instances described in Section 3.1.1. For the evaluation of crossovers, we choose two problem instances from each set of problems. For each instance, the genetic algorithms with different crossovers run 10 times, and each time, a new initial population is created**.** Computations are performed on a personal computer with Intel Core 2 Duo 2.2 GHz CPU and 4GB RAM. Tables 1 and 2 summarize the results, where computations are performed without applying the mutation operator, and Tables 3 and 4 summarize the results, where computations involve the mutation operator. In Tables 1 and 3, the best results identified are presented, and, in Tables 2 and 4, the averaged results for each crossover operator are presented. The results in the tables show the difference from the best known solutions, reported in the papers (Solomon, 1987; Potvin and Bengio, 1996; Tan et al., 2001; Jung and Moon, 2002; Ombuki et al., 2002; Berger and Barkaoui, 2004; Alvarenga et al., 2005; Ombuki et al., 2006; Tan et al., 2006; Garcia-Najera and Bullinaria, 2011). The results are displayed in the form "*difference of the total path length / difference of the vehicle number*". Problem instances used in the experiments and the best known solutions are as follows:

- C104 – vehicles 10, total path length 824.78;
- C106 – vehicles 10, total path length 828.94;
- C204 – vehicles 3, total path length 590.6;

- C207 – vehicles 3, total path length 588.29;
- R101 – vehicles 19, total path length 1645.79;
- R105 – vehicles 14, total path length 1377.11;
- R205 – vehicles 3, total path length 994.42;
- R209 – vehicles 3, total path length 909.16;
- RC101 – vehicles 14, total path length 1696.94;
- RC107 – vehicles 11, total path length 1230.48;
- RC201 – vehicles 4, total path length 1406.91;
- RC208 – vehicles 3, total path length 828.14;

Table 1. The difference between the best results, found by using crossover operators without applying mutation, and the best known solutions

	BCRC	RBX	LRX	CNX (proposed)	CAX (proposed)	LCSX (proposed)
C104	22.27/0	22.98/0	907.52/0	**0/0**	**0/0**	**0/0**
C106	**0/0**	**0/0**	886.69/2	**0/0**	**0/0**	**0/0**
C204	**0/0**	10.61/0	708.94/1	**0/0**	**0/0**	**0/0**
C207	**0/0**	8.49/0	696.19/1	**0/0**	**0/0**	**0/0**
R101	46.0/0	46.3/0	186.5/0	*9.0/0*	17.32/0	**7.74/0**
R105	30.35/1	58.77/1	286.72/1	26.7/0	*10.38/0*	**0/0**
R205	46.04/0	82.61/0	698.16/1	*68.0/0*	73.010/0	**27.32/0**
R209	34.35/0	61.04/3	485.63/1	*35.57/0*	41.92/0	**19.43/0**
RC101	-8.5/1	30.55/1	125.31/2	-45.09/1	*-52.68/1*	**-53.07/1**
RC107	62.43/0	54.62/1	438.83/2	*8.05/0*	268.73/1	**3.81/0**
RC201	22.61/0	34.94/0	724.57/1	28.1/0	*10.86/0*	**6.61/0**
RC208	39.38/0	70.4/0	922.42/1	19.7/0	*18.1/0*	**4.22/0**

Table 2. The difference between the averaged results, found by using crossover operators without applying mutation, and the best known solutions

	BCRC	RBX	LRX	CNX (proposed)	CAX (proposed)	LCSX (proposed)
C104	61.88/0	61.03/0	1292.83/0	*8.98/0*	20.16/0	**2.86/0**
C106	47.53/0	91.65/0.1	1043.65/2.2	**0/0**	**0/0**	**0/0**

	BCRC	RBX	LRX	CNX (proposed)	CAX (proposed)	LCSX (proposed)
C204	1.69/0	27.96/0	860.58/3	3.24/0	**0/0**	**0/0**
C207	**0/0**	23.48/0	1078.35/1	**0/0**	**0/0**	**0/0**
R101	83.64/0.3	83.56/0.5	194.31/0.5	*23.32/0*	37.58/0.3	**15.93/0**
R105	73.68/1	134.69/1.1	322.18/1.9	120.5/0.5	*77.18/0.3*	**16.32/0.2**
R205	80.76/0	122.82/0	722.36/1	*99.11/0*	119.46/0	**65.15/0**
R209	58.99/0	110.87/0	646.42/1	*66.40/0*	75.48/0	**44.18/0**
RC101	33.07/1.7	98.23/1.5	186.03/2.1	*-28.1/1.2*	**4.4/1.1**	-38.89/1.3
RC107	71.15/0.8	1253.52/1	488.63/2.9	*85.04/0.5*	303.1/1.1	**25.9/0.5**
RC201	76.01/0	128.65/0	894.62/1.1	*74.55/0*	93.9/0	**30.99/0**
RC208	59.51/0	98.21/0	982.84/1	81.85/0	*63.43/0*	**17.69/0**
Averaged CPU time	32.82	94.84	7.37	89.131	82.99	35.75

Table 3. The difference between the best results, found by using crossover operators and mutation also being applied, and the best known solutions

	BCRC	RBX	LRX	CNX (proposed)	CAX (proposed)	LCSX (proposed)
C104	**0/0**	15.17/10	94.75/0	**0/0**	**0/0**	**0/0**
C106	**0/0**	**0/0**	**0/0**	**0/0**	**0/0**	**0/0**
C204	**0/0**	**0/0**	0.57/0	**0/0**	**0/0**	**0/0**
C207	**0/0**	**0/0**	**0/0**	**0/0**	**0/0**	**0/0**
R101	8.51/0	8.97/0	36.58/0	*6.38/0*	6.78/0	**5.22/0**
R105	0.66/1	106.17/0	56.61/0	*0.61/0*	10.38/0	**0/0**
R205	45.69/0	38.91/0	99.77/0	39.63/0	*28.97/0*	**38.83/0**
R209	25.19/0	44.51/0	62.35/0	37.72/0	*13.01/0*	**6.0/0**
RC101	–40.5/1	–24.87/1	-14.91/1	*-50.11/1*	-40.62/1	**-53.95/1**
RC107	28.92/0	32.35/1	109.31/0	5.59/0	*5.53/0*	**3.53/0**
RC201	22.83/0	56.88/0	99.72/0	17.46/0	**6.61/0**	*8.09/0*
RC208	32.49/0	45.69/0	128.38/0	34.72/0	*17.47/0*	**0.87.01/0**

Table 4. The difference between the averaged results, found by using crossover operators and mutation also being applied, and the best known solution

	BCRC	RBX	LRX	CNX (proposed)	CAX (proposed)	LCSX (proposed)
C104	31.95/0	61.83/0	313.5/0	1.76/0	*0.19/0*	**0/0**
C106	29.71/0	34.78/0	3.81/0	**0/0**	**0/0**	**0/0**
C204	0.74/0	5.69/0	38.79/0.3	**0/0**	**0/0**	**0/0**
C207	**0/0**	**0/0**	56.23/0.3	**0/0**	**0/0**	**0/0**
R101	33.29/0.1	33.06/0.3	76.54/0.3	15.8/0	*14.47/0*	**14.34/0**
R105	41.33/1	62.68/0.9	83.43/0.5	96.57/0.4	*23.59/0.1*	**5.18/0**
R205	82.19/0	78.21/0	189.15/0.6	69.14/0	**56.66/0**	*66.98/0*
R209	52.97/0	79.53/0	192.97/0.7	79.56/0	*49.47/0*	**24.33/0**
RC101	-2.48/1.4	21.69/1.4	26.81/1.3	-35.46/1	*-39.06/1*	**-41.44/1**
RC107	67.39/0.7	96.1/1.1	122.5/0.8	68.07/0.4	*78.58/0.3*	**36.39/0.3**
RC201	69.73/0	82.74/0	147.17/0.7	71.18/0	*33.53/0*	**18.95/0**
RC208	50.89/0	70.6/0	167.23/0.3	75.17/0	*44.25/0*	**23.48/0**
Averaged CPU time, s	39.78	150.79	110.44	96.62	92.74	43.29

The results are compared according to the defined objective: at first, the vehicle number differences are compared, and afterwards differences of the total path length is compared. The best values are bold in Tables 1-4 and the second best values are displayed in italics.

In Figure 18 a comparison of the average difference of path lengths found by the described crossover operators are presented. In Figure 19 a comparison of the average difference of vehicle number found by the described crossover operators are presented. We can see that the results found by LCSX crossover are smaller than the results of other crossover operators when comparing results obtained with mutation applied and without mutation applied. We can also see that LCSX crossover found better results without mutation applied in comparison with results of other crossover operators when mutation was applied.

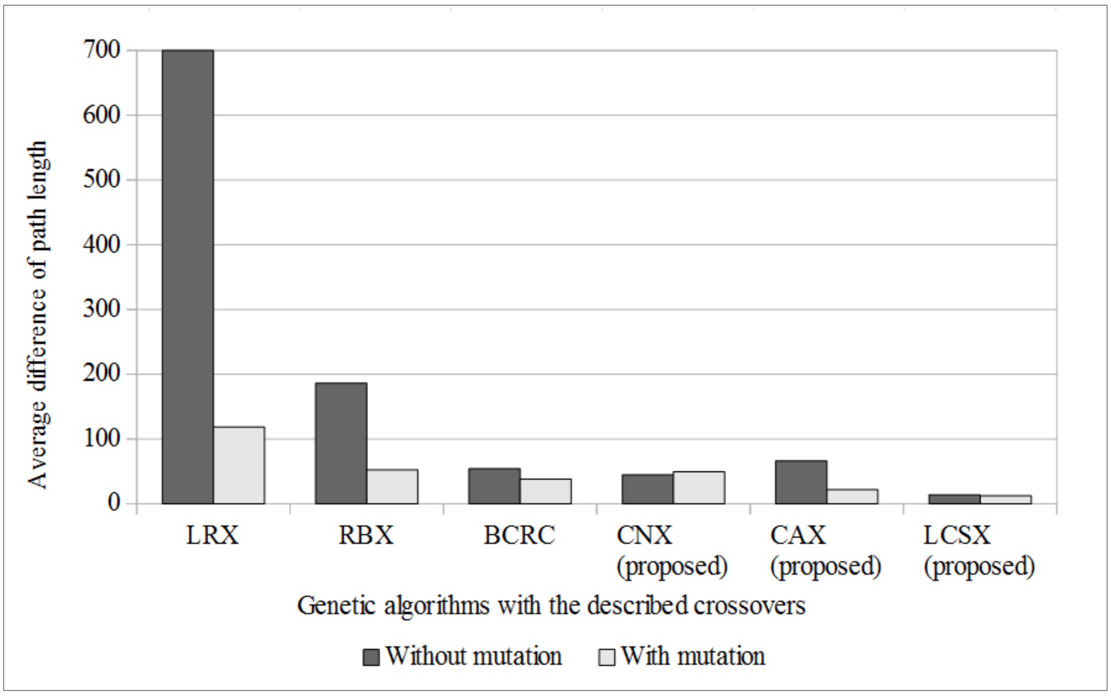

Fig. 18. Average difference of total path length found by the described crossover operators

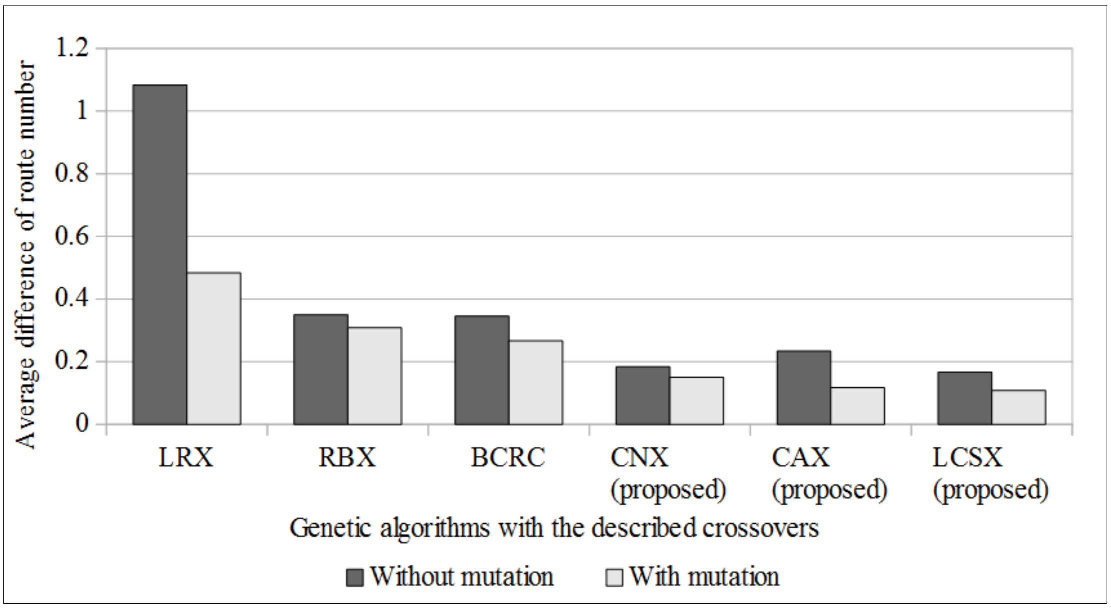

Fig. 19. The average difference of route number found by the described crossover operators

The results show that the LCSX crossover, proposed in this research, has found better average solutions than the other crossover operators in 22 cases out of 24 (in ~92% cases). In 2 cases out of 24, the better solutions has been found by CAX. Comparing the best results, LCSX has found better solutions in 23 cases out of 24 (in 96% cases), and in the experiments, where mutation was not applied, LCSX has found better solutions in all the cases. For the RC101 problem instance, the identified path length difference is negative, however, the difference of vehicle number is positive. It means that a better path length is identified for the RC101 problem, but the number of routes was not minimized to the best known number. The best CPU time in the cases, where the mutation was not applied, belongs to crossover LRX, however, the solutions found by this crossover are worst. The computation with the LRX crossover stopped early without finding better solutions, so it leads to a short CPU time and not so good solutions found. The LRX crossover showed better characteristics, when the mutation was applied, but the results are still worst comparing to other crossovers.

The best computation time in the cases, where mutation is applied, belongs to BCRC. The computation time of LCSX is the second best one and is quite similar to the computation time of BCRC. However, the solutions found by LCSX are more accurate than that found by BCRC. The genetic algorithm with CAX has found the second best solutions in most cases, where mutation was applied, and in some cases, where mutation was not applied. The CPU time of CAX is longer than LCSX, because preservation of the common arcs at the beginning produces more unrouted nodes and requires more insertion trials while searching for a solution that could be competitive in the population. The CPU time of CNX is similar to the time of CAX, because more iterations requires to improve the best solution comparing to LCSX: the CNX preserves common nodes and produces small amount of unrouted nodes, so CNX requires to cross more solutions to find better one. However, in all cases the proposed genetic operators found the best and the second best results.

Although the CPU time of BCRC is shorter than LCSX and CAX, BCRC cannot be applied to the problems, where the solutions is one route.

It is worth mentioning, that crossovers, defined by other authors (BCRC, RBX, LRX), can be dependent on other parts in the genetic algorithm, i.e. on the created initial population or the selection operator, etc. LCSX has showed better results when computing without mutation or with mutation that randomly removes and reinserts nodes.

3.3. Results of the proposed genetic algorithm

The algorithm proposed in Section 2.1 is tested using two problem sets described in Sections 3.1.1 and 3.1.2. The proposed genetic algorithm is implemented using the Java programming language. All computations are performed by a personal computer (Intel Core 2 Duo 2.2 GHz CPU, 4GB RAM). In the experimental evaluation, parameter values in the genetic algorithm are defined as follows:

- $PS_1 = 100$ – size of the first population;
- $PL_1 = 10$ – recombination operations in a single iteration;
- $IL_1 = 50$ – maximum iteration number without improvement in the first population;
- $TL_1 = 5$min – maximum execution time of the algorithm;
- $MP = 0.1$ – mutation rate;
- $PS_2 = 20$ – size of the second population;
- $IL_2 = 5$ – maximum iteration number without improvement in the second population;
- $PL_2 = 2$ – recombination operations in a single iteration performed in the second population.

All the obtained results are compared with the best results obtained by other algorithms in the following papers:

[1] (Solomon, 1987);

[2] (Berger et al., 1998);

[3] (Ho et al., 2001);

[4] (Tan et al., 2001);

[5] (Tan et al., 2001a);

[6] (Jung and Moon, 2002);

[7] (Bent and Hentenryck, 2003);

[8] (Li and Lim, 2003);

[9] (Zhu, 2003);

[10] (Berger and Barkaoui, 2004);

[11] (Alvarenga et al., 2005);

[12] (Ombuki et al., 2006);

[13] (Tan et al., 2006);

[14] (Hasle and Kloster, 2007);

[15] (Garcia-Najera and Bullinaria, 2011).

The above mentioned papers describe different algorithms for solving VRPTW and VRPPD. The different genetic algorithms designed to solve VRPTW include different crossover operators reviewed in Sections 1.3 and 1.5. Different algorithms also include different approaches for a diversification where various mutations are applied that involve the generation of feasible and infeasible solutions, a simultaneous evolution of two populations is also involved to increase the diversification. To intensify the search different repair and improvement methods based on the local search heuristics, as well as hybrid approaches that involve the ant colony optimization, the tabu search, the column generation heuristic, the parallelization and multi-objective optimization, are used. The dynamic adaptation of the crossover probability and the mutation rate that depends on the changing population dynamics is also used. The algorithms used for comparison of VRPPD results involve hybrid approaches of large neighborhood search, tabu search, simulated annealing where different algorithms are applied in different stages of the computation.

In Tables 5-10, the results of VRPTW instances are summarized. The first column defines the problem instance name; three next columns present the best known solutions, the best known solutions obtained by other genetic algorithms and the best solution obtained by the proposed algorithm. In the last three columns, there are presented the published best average results obtained by other genetic algorithms, average results obtained by the proposed algorithm and the average CPU time used in calculation.

The average results are obtained by executing the proposed algorithm 10 times for each problem instance when each time a new initial population is created. The results in Tables are displayed in the form "*total path length /vehicle number*".

Table 5. Results of problem set R1

Problem	Best distance/vehicles			Average distance/ vehicle		Average CPU time of the proposed algorithm
	Best solution	Best GA solution	Best solution of the proposed algorithm	Best GA solution	Solution of the proposed algorithm	
R101	1645.79 /19	**1650.8 /19** [15]	**1650.8 /19**	1693.23 /19.6 [12]	**1650.8 /19**	42.5
R102*	1486.12 /17	1487.31 /17 [15]	**1486.12 /17**	1525.46 /18.2 [12]	**1487.04 /17**	27.27
R103	1292.68 /13	1299.18 /13 [15]	**1296.29 /13**	1281.32 /13.8 [12]	**1234.48 /13.8**	31.48
R104	1007.24 /9	999.82 /10 [15]	**982.02 /10**	1035.10 /10 [12]	**989.96 /10**	48.09
R105*	1377.11 /14	**1377.11 /14** [15]	**1377.11 /14**	1430.86 /14.9 [12]	**1385.56 /14**	52.85
R106	1251.98 /12	1263.21 /12 [15]	**1252.03 /12**	1298.27 /12.8 [12]	**1259.28 /12**	59.64
R107	1104.66 /10	1164.14 /11 [6]	**1117 /10**	1115.87 /11 [12]	**1127.04 /10**	70.20

Problem	Best distance/vehicles			Average distance/ vehicle		Average CPU time of the proposed algorithm
	Best solution	Best GA solution	Best solution of the proposed algorithm	Best GA solution	Solution of the proposed algorithm	
R108	960.99 /9	**960.99 /9** [10]	968.97 /9	990.39 /10 [12]	**970.18 /9**	51.8
R109	1194.73 /11	1156.05 /12 [15]	**1245.32 /11**	1244.87 /12.5 [12]	**1175.5 /11.8**	21.12
R110	1118.59 /10	1119 /10 [10]	**1119 /10**	1146.11 /11.9 [12]	**1091.95 /10.9**	43.66
R111	1096.72 /10	1084.76 /11 [12]	**1096.74 /10**	1132.51 /11 [12]	**1107.13 /10**	75.04
R112	982.14 /9	**953.63 /10** [6]	962.03 /10	1022.51 /10.3 [12]	**977.05 /10**	52.71

Table 6. Results of problem set R2

Problem	Best distance/vehicles			Average distance/ vehicle		Average CPU time of the proposed algorithm
	Best solution	Best GA solution	Best solution of the proposed algorithm	Best GA solution	Solution of the proposed algorithm	
R201	1252.37 /4	1253.32 /4 [15]	**1253.23 /4**	1313.23 /4 [12]	**1262.83 /4**	17.32
R202	1191.7 /3	1081.6 /4 [15]	**1195.3 /3**	1114.77 /4 [12]	**1196.60 /3**	27.76
R203	939.50 /3	959.75 /3 [15]	**947.09 /3**	974.51 /3 [12]	**966.71 /3**	14.92
R204	825.52 /2	760.82 /3 [12]	**846.42 /2**	777.37 /3 [12]	**849.17 /2**	60.68
R205	994.42 /3	1030.92 /3 [15]	**1029.1 /3**	1070.66 /3 [12]	**1052.89 /3**	31.00
R206	906.14 /3	919.73 /3 [12]	**918.75 /3**	949.25 /3 [12]	**932.26 /3**	26.07
R207*	890.61 /2	821.32 /3 [12]	**890.61 /2**	848.30 /3 [12]	**911.02 /2**	88.19
R208*	726.82 /2	736.47 /2 [15]	**726.82 /2**	747.98 /3 [12]	**734.53 /2**	37.43

Problem	Best distance/vehicles			Average distance/ vehicle		Average CPU time of the proposed algorithm
	Best solution	Best GA solution	Best solution of the proposed algorithm	Best GA solution	Solution of the proposed algorithm	
R209	909.16 /3	921.37 /3 [15]	**913.14 /3**	955.46 /4 [12]	**931.54 /3**	40.72
R210	939.34 /3	**954.12 /3** [10]	**954.12 /3**	999.02 /3 [12]	**969.81 /3**	29.89
R211	885.71 /2	906.19 /2 [10]	**900.88 /2**	823.34 /3 [12]	**929.60 /2**	80.99

Table 7. Results of problem set C1

. Problem	Best distance/vehicles			Average distance/ vehicle		Average CPU time of the proposed algorithm
	Best solution	Best GA solution	Best solution of the proposed algorithm	Best GA solution	Solution of the proposed algorithm	
C101*	828.94 /10	**828.94 /10** [6]	**828.94 /10**	**828.94 /10** [6]	**828.94 /10**	12.2
C102*	828.94 /10	**828.94 /10** [6]	**828.94 /10**	**828.94 /10** [6]	**828.94 /10**	13.1
C103*	828.06 /10	**828.06 /10** [6]	**828.06 /10**	**828.06 /10** [6]	**828.06 /10**	15.25
C104*	824.78 /10	**824.78 /10** [6]	**824.78 /10**	**824.78 /10** [6]	**824.78 /10**	16.4
C105*	828.94 /10	**828.94 /10** [6]	**828.94 /10**	**828.94 /10** [6]	**828.94 /10**	12.55
C106*	828.94 /10	**828.94 /10** [6]	**828.94 /10**	**828.94 /10** [6]	**828.94 /10**	12.86
C107*	828.94 /10	**828.94 /10** [6]	**828.94 /10**	**828.94 /10** [6]	**828.94 /10**	12.85
C108*	828.94 /10	**828.94 /10** [6]	**828.94 /10**	**828.94 /10** [6]	**828.94 /10**	13.24
C109*	828.94 /10	**828.94 /10** [6]	**828.94 /10**	**828.94 /10** [6]	**828.94 /10**	14.67

Table 8. Results of problem set C2

Problem	Best distance/vehicles			Average distance/ vehicle		Average CPU time of the proposed algorithm
	Best solution	Best GA solution	Best solution of the proposed algorithm	Best GA solution	Solution of the proposed algorithm	
C201*	591.56 /3	**591.56 /3** [6]	**591.56 /3**	**591.56 /3** [6]	**591.56 /3**	12.34
C202*	591.56 /3	**591.56 /3** [6]	**591.56 /3**	**591.56 /3** [6]	**591.56 /3**	12.57
C203*	591.17 /3	**591.17 /3** [6]	**591.17 /3**	**591.17 /3** [6]	**591.17 /3**	13.29
C204*	590.6 /3	**590.6 /3** [6]	**590.6 /3**	**590.6 /3** [6]	**590.6 /3**	15.03
C205*	588.88 /3	**588.88 /3** [6]	**588.88 /3**	**588.88 /3** [6]	**588.88 /3**	12.68
C206*	588.49 /3	**588.49 /3** [6]	**588.49 /3**	**588.49 /3** [6]	**588.49 /3**	12.85
C207*	588.29 /3	**588.29 /3** [6]	**588.29 /3**	**588.29 /3** [6]	**588.29 /3**	12.86
C208*	588.32 /3	**588.32 /3** [6]	**588.32 /3**	**588.32 /3** [6]	**588.32 /3**	12.88

Table 9. Results of problem set RC1

Problem	Best distance/vehicles			Average distance/ vehicle		Average CPU time of the proposed algorithm
	Best solution	Best GA solution	Best solution of the proposed algorithm	Best GA solution	Solution of the proposed algorithm	
RC101	1696.94 /14	1636.92 /15 [12]	**1697.43 /14**	1668.52 /15.4 [12]	**1649.63 /14.8**	76.94
RC102*	1554.75 /12	1470.26 /13 [13]	**1554.75 /12**	1536.04 /13.8 [12]	**1547.69 /12.4**	63.34
RC103	1261.67 /11	**1267.86 /11** [13]	1273.81 /11	1350.15 /12 [12]	**1280.27 /11**	84.99
RC104	1135.48 /10	1136.81 /10 [6]	**1135.83 /10**	1184.29 /10.4 [12]	**1141.37 /10**	46.27

Problem	Best distance/vehicles			Average distance/ vehicle		Average CPU time of the proposed algorithm
	Best solution	Best GA solution	Best solution of the proposed algorithm	Best GA solution	Solution of the proposed algorithm	
RC105	1629.44 /13	**1629.44 /13** [10]	1540.18 /14	1618.63 /15 [12]	**1556.01 /14**	88.91
RC106	1424.73 /11	**1424.73 /11** [10]	1376.26 /12	1450.3 /12.8 [12]	**1390.15 /12**	38.35
RC107	1230.48 /11	1235.37 /11 [15]	**1230.95 /11**	1227.81 /12.03 [6]	**1232.78 /11**	58.62
RC108*	1139.82 /10	1141.34 /10 [12]	**1139.82 /10**	1135.81 /11 [6]	**1151.75 /10**	88.54

Table 10. Results of problem set RC2

Problem	Best distance/vehicles			Average distance/ vehicle		Average CPU time of the proposed algorithm
	Best solution	Best GA solution	Best solution of the proposed algorithm	Best GA solution	Solution of the proposed algorithm	
RC201	1406.91 /4	1423.73 /4 [12]	**1417.45 /4**	1492.67 /4 [12]	**1435.06 /4**	30.06
RC202	1365.65 /3	1162.54 /4 [15]	**1367.09 /3**	1212.49 /4 [12]	**1415.48 /3**	105.47
RC203	1049.62 /3	**1058.33 /3** [15]	**1058.33 /3**	1152.64 /3 [12]	**1088.31 /3**	44.42
RC204*	798.46 /3	801.90 /3 [15]	**798.46 /3**	826.19 /3 [12]	**812.77 /3**	28.03
RC205*	1302.42 /4	1304.93 /4 [15]	**1302.42 /4**	1378.44 /4 [12]	**1330.06 /4**	20.93
RC206*	1146.32 /3	1203.7 /3 [12]	**1146.32 /3**	1164.33 /3.3 [12]	**1159.36 /3**	43.09
RC207	1061.14 /3	1093.25 /3 [12]	**1070.85 /3**	1052.13 /3.7 [12]	**1080.66 /3**	77.18
RC208*	828.14 /3	834.88 /3 [15]	**828.14 /3**	938.24 /3 [12]	**851.43 /3**	28.13

The asterisks near the names of the problems in Tables 5-10 show which instance solution found by the proposed algorithm is equal to the best known solution. The results are compared in the same way as they have been defined in the objective: firstly, the found vehicle numbers are compared and afterwards the found traveling distances are compared. The best solutions obtained by other genetic algorithms are compared to the best solutions found by the proposed algorithm and the best average results are compared with the average results found by the proposed algorithm. Better values are in bold in Tables 5-10.

We see that the proposed algorithm shows very good results for the problem set C, where the average results are equal to the best known values and the computation time is very small. The results show that for other problem sets R and RC the best values are found only in some cases. However, the results obtained by the proposed algorithm, in comparison with other genetic algorithm approaches, show that the same or better results are obtained for 51 out of 56 problem instances (in ~91% cases) of for the best solutions, and the same or better results for 56 out of 56 problem instances (in ~100% cases) comparing with the best published average results.

Table 11 shows the best results that are averaged over categories (C, R, RC). The columns show the results from different papers as well as the average results obtained by the proposed algorithm.

The bold values in Table 11 show the minimal value compared at first according to the found vehicle number and then according to the found shortest distance. The results show that for problem sets C1 and C2 the proposed algorithm finds solutions that are equal to the best results. The proposed algorithm finds solutions that are better than other ones for problem sets R2 and RC2, where problems have large time windows. Better results were obtained in [10] for problems with narrow time windows, R1 and RC1.

Table 11. Travel distance and the number of vehicles, averaged over categories

	[4]	[5]	[3]	[9]	[12]	[10]	[11]	[15]	Solution of the proposed algorithm
C1	861 /10.1	860.62 /10.1	833.32 /10	828.9 /10	828.48 /10	**828.38 /10**	**828.38 /10**	**828.38 /10**	**828.38 /10**
C2	619 /3.3	624.47 /3.3	593 /3	**589.86 /3**	590.6 /3	589.93 /3	590.9 /3	591.74 /3	**589.86 /3**
R1	1227 /13.2	1314.79 /14.4	1203.32 /12.6	1242.7 /12.8	1220.92 /12.5	**1221.1 /11.92**	1224 /11.92	1187.32 /13.08	1213.66 /12.08
R2	980 /5	1093.37 /5.6	951.17 /3.2	1016.4 /3	938.75 /3.1	975.43 /2.73	1012 /2.73	897.95 /4	**961.44 /2.73**
RC1	1427 /13.5	1512.94 /14.6	1382.06 /12.8	1412 /13	1386.35 /12.12	**1389.89 /11.5**	1417 /11.5	1348.22 /12.63	1370.01 /11.75
RC2	1123 /5	1282.47 /7	1132.79 /3.8	1201.2 /3.7	1132.12 /3.38	1159.37 /3.25	1195 /3.25	1036.65 /5.63	**1126.75 /3.25**

It is worth mentioning that the results, obtained by the proposed algorithm, were identified on average in 38.97 seconds for VRPTW instances by Intel Core 2 Duo 2.2 GHz (1.09 Gflops/s for single core operations). The results obtained in [11] were found in 15 minutes by Pentium IV 2.4.GHz (0.9 Gflops/s), in [9] they were found in 592 seconds by Pentium IV 2.4 GHz (0.9 Gflops/s) and in [10] the presented results were found in 30 minutes by Pentium 400 MHz (54 Mflops/s). In [15] the results were found in 117 seconds by a computer cluster with dual-processor dual-core AMD Opteron 2.6 GHz (1.23 Gflop/s for single core operations). In Figure 20 the average floating point operations used to solve VRPTW instances are displayed, where the proposed genetic algorithm performs ~2 times less floating point operations to find the results comparing to the best value of other algorithms.

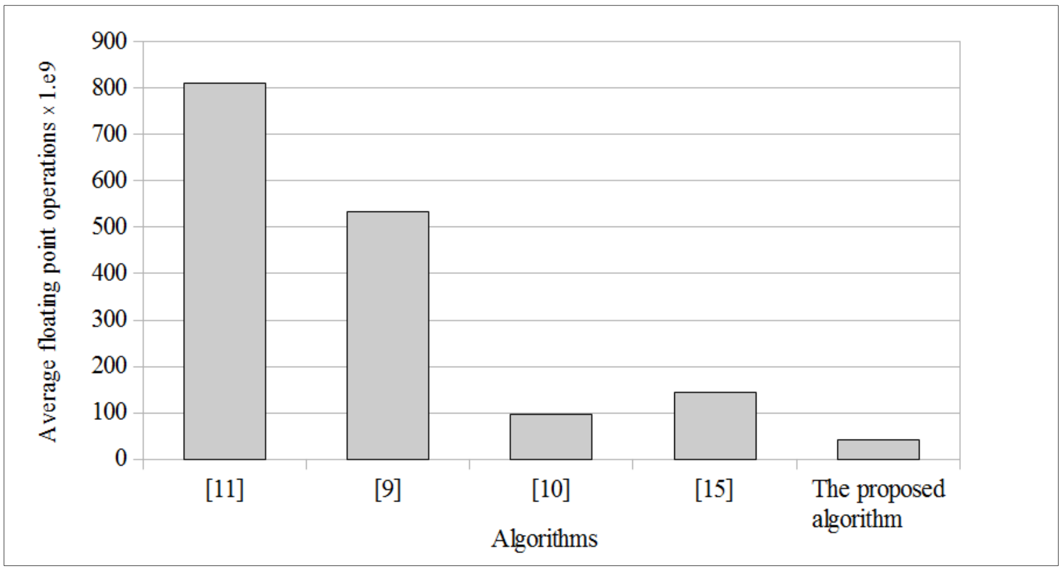

Fig. 20. Average floating point operations ×1.e9

The results in Figure 20 are presented by taking into account computer performance values obtained by public benchmarks. The theoretical performance of computer used in the experiments is 8.8 Gflops/s (4 flops per cycle) for single core. The theoretical performance of the computer used in [10] is 400 Mflops/s (1 flop per cycle). In the experiments 8.8 Gflops/s × 38.97 seconds = 342.94×10^9 average floating point operations are used to solve VRPTW. To obtain results in [10] 400 Mflops/s × 1800 seconds = 720×10^9 average floating point operations are used. Taking into account a theoretical CPU performance, the proposed genetic algorithm still performs ~2 times less floating point operations to find the results comparing to other algorithms.

In Figure 21 the average difference of routes found by the proposed algorithm and other genetic algorithms comparing to the route numbers of the best known results is presented. Route numbers found in [10] and [11] are equal to route numbers of the best known results.

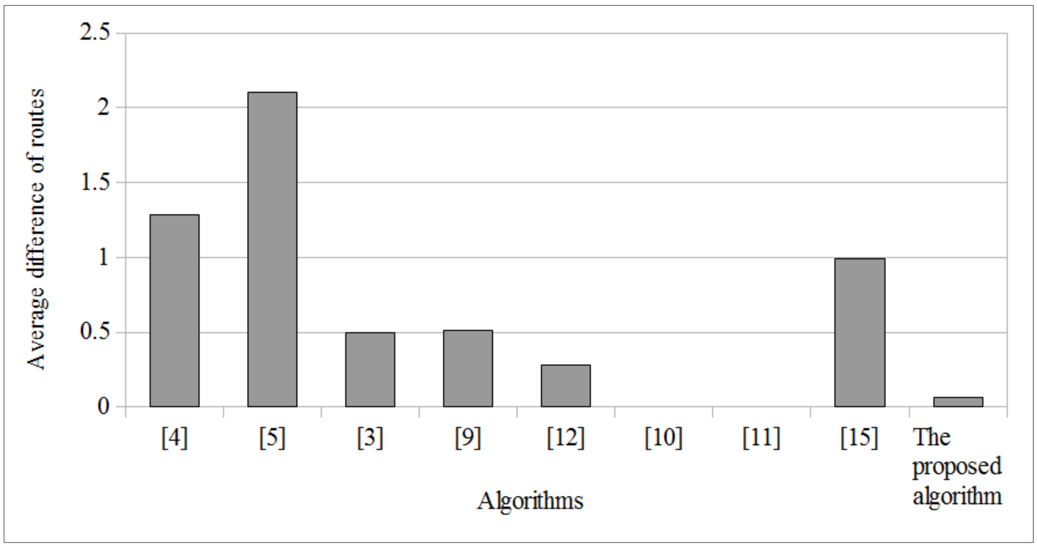

Fig. 21. Average difference of routes numbers comparing to the route number of the best know results

In Tables 12-17, the results of VRPPD instances are presented. The first column shows instance names, in the second column the best known results are presented; the third column presents the best results, obtained by the proposed algorithm, and the last two columns show the average results and average CPU time obtained by the proposed algorithm.

Table 12. Results of problem set LR1

Problem	Best distance/vehicles		Average results	
	Best solution	Solution of the proposed algorithm	Solution of the proposed algorithm	CPU time of the proposed algorithm
LR101	1650.8/19 [8]	**1650.8/19**	**1650.8/19**	15.315
LR102	1487.57/17 [8]	**1487.57/17**	**1487.57/17**	17.115
LR103	1292.68/13 [8]	**1292.68/13**	**1292.68/13**	17.661
LR104	1013.39/9 [8]	**1013.39/9**	**1013.39/9**	44.836
LR105	1377.11/14 [8]	**1377.11/14**	**1377.11/14**	15.959
LR106	1252.62/12 [8]	**1252.62/12**	**1252.62/12**	14.673
LR107	1111.31/10 [8]	**1111.31/10**	**1111.31/10**	20.001
LR108	968.97/9 [8]	**968.97/9**	**968.97/9**	16.577
LR109	1208.96/11 [14]	**1208.96/11**	**1208.96/11**	46.617
LR110	1159.35/10 [8]	**1159.35/10**	1167.55/10.7	50.68

Problem	Best distance/vehicles		Average results	
	Best solution	Solution of the proposed algorithm	Solution of the proposed algorithm	CPU time of the proposed algorithm
LR111	1108.9/10 [8]	**1108.9/10**	**1108.9/10**	35.007
LR112	1003.77/9 [8]	**1003.77/9**	**1003.77/9**	41.956

Table 13. Results of problem set LR2

Problem	Best distance/vehicles		Average results	
	Best solution	Solution of the proposed algorithm	Solution of the proposed algorithm	CPU time of the proposed algorithm
LR201	1253.23/10 [14]	**1253.23/10**	**1253.23/10**	13.59
LR202	1197.67/3 [8]	**1197.67/3**	1213.39/3.3	26.806
LR203	949.40/3 [8]	**949.40/3**	**949.40/3**	15.59
LR204	849.05/2 [8]	**849.05/2**	**849.05/2**	20.18
LR205	1054.02/3 [8]	**1054.02/3**	**1054.02/3**	16.68
LR206	931.63/3 [8]	**931.63/3**	**931.63/3**	14.40
LR207	903.06/2 [8]	**903.06/2**	921.41/2.3	29.76
LR208	734.85/2 [8]	**734.85/2**	**734.85/2**	17.75
LR209	930.59/3 [14]	**930.59/3**	939.92/3.1	16.6
LR210	964.22/3[8]	**964.22/3**	999.74/3	22.09
LR211	911.52/2 [14]	**911.52/2**	**911.52/2**	31.3

Table 14. Results of problem set LC1

Problem	Best distance/vehicles		Average results	
	Best solution	Solution of the proposed algorithm	Solution of the proposed algorithm	CPU time of the proposed algorithm
LC101	828.94/10 [8]	**828.94/10**	**828.94/10**	12.245
LC102	828.94/10 [8]	**828.94/10**	**828.94/10**	12.458
LC103	1035.35/9 [7]	**1035.35/9**	1057.70/9	32.96
LC104	860.01/9 [14]	**860.01/9**	839.31/9.5	27.353
LC105	828.94/10 [8]	**828.94/10**	**828.94/10**	12.38
LC106	828.94/10 [8]	**828.94/10**	**828.94/10**	12.477
LC107	828.94/10 [8]	**828.94/10**	**828.94/10**	12.454

Problem	Best distance/vehicles		Average results	
	Best solution	Solution of the proposed algorithm	Solution of the proposed algorithm	CPU time of the proposed algorithm
LC108	826.44/10 [8]	**826.44/10**	**826.44/10**	12.609
LC109	1000.6/9[7]	1036.41/9	896.72/9.7	26.813

Table 15. Results of problem set LC2

Problem	Best distance/vehicles		Average results	
	Best solution	Solution of the proposed algorithm	Solution of the proposed algorithm	CPU time of the proposed algorithm
LC201	591.56/3 [8]	**591.56/3**	**591.56/3**	12.265
LC202	591.56/3 [8]	**591.56/3**	**591.56/3**	12.307
LC203	585.56/3 [8]	591.17/3	591.17/3	12.479
LC204	590.60/3 [14]	**590.60/3**	**590.60/3**	13.166
LC205	588.88/3 [8]	**588.88/3**	**588.88/3**	12.432
LC206	588.49/3 [8]	**588.49/3**	**588.49/3**	12.546
LC207	588.29/3 [8]	**588.29/3**	**588.29/3**	12.516
LC208	588.32/3 [8]	**588.32/3**	**588.32/3**	12.475

Table 16. Results of problem set LRC1

Problem	Best distance/vehicles		Average results	
	Best solution	Solution of the proposed algorithm	Solution of the proposed algorithm	CPU time of the proposed algorithm
LRC101	1708.80/14 [8]	**1708.80/14**	**1708.80/14**	18.2
LRC102	1558.07/12 [14]	**1558.07/12**	**1558.07/12**	20.383
LRC103	1258.74/11 [8]	**1258.74/11**	**1258.74/11**	19.47
LRC104	1128.40/10 [8]	**1128.40/10**	**1128.40/10**	16.787
LRC105	1637.62/13 [8]	**1637.62/13**	**1637.62/13**	23.077
LRC106	1424.73/11 [14]	**1424.73/11**	**1424.73/11**	36.105
LRC107	1230.15/11 [8]	**1230.14/11**	**1230.14/11**	19.488
LRC108	1147.43/10 [14]	**1147.43/10**	1168.4/10.7	25.531

Table 17. Results of problem set LRC2

Problem	Best distance/vehicles		Average results	
	Best solution	Solution of the proposed algorithm	Solution of the proposed algorithm	CPU time of the proposed algorithm
LRC201	1406.94/4 [14]	**1406.94/4**	**1406.94/4**	42.016
LRC202	1374.27/3 [8]	**1374.27/3**	1392.59/3.6	37.966
LRC203	1089.07/03 [8]	**1089.07/03**	**1089.07/03**	18.329
LRC204	818.66/3 [14]	**818.66/3**	**818.66/3**	15.961
LRC205	1302.20/4 [8]	**1302.2/4**	**1302.20/4**	29.366
LRC206	1159.03/3 [14]	**1159.03/3**	**1159.03/3**	21.527
LRC207	1062.05/3 [14]	**1062.05/3**	**1062.05/3**	22.121
LRC208	852.76/3 [8]	**852.76/3**	**852.76/3**	18.563

The bold numbers in Tables 8-13 for the VRPPD problem show where the best solutions, obtained by the proposed algorithm, are equal to the best known solutions. The results for VRPPD instances show that the solutions, found by the proposed algorithm, are equal to the best known solutions for 54 out of 56 problem instances (in ~96% cases) and the average results found are equal to the best known solutions for 45 out of 56 problem instances (in ~80% cases). For VRPTW and VRPPD instances the minimal computation time is for problem set C, where customers are located in clusters.

3.4. Results of the parallel bi-directional shortest path algorithm

This parallel bidirectional Dijkstra's algorithm was implemented with *pthread* – a POSIX standard for threads. Each process of the proposed parallel approach is implemented as a separate thread. In order to solve conflicted access to the same memory area, we used "mutexes" - pthread algorithms that are used in concurrent programming to avoid the simultaneous use of the common resource by pieces of the computer code called critical sections. The algorithm was developed in the C language and tested in the latest 64-bit Fedora Linux Operating System. The experiments were carried out using personal computer with the Intel Core 2 Duo 2.2GHz processor and 4GB RAM. The algorithm was tested on a real road network using the OpenStreetMap data (Haklay and Weber, 2008). Figure 22 provides examples that were visualized with the UMN-Mapserver open source software. The description of UMN-Mapserver can be found in (Vatsavai et al., 2006).

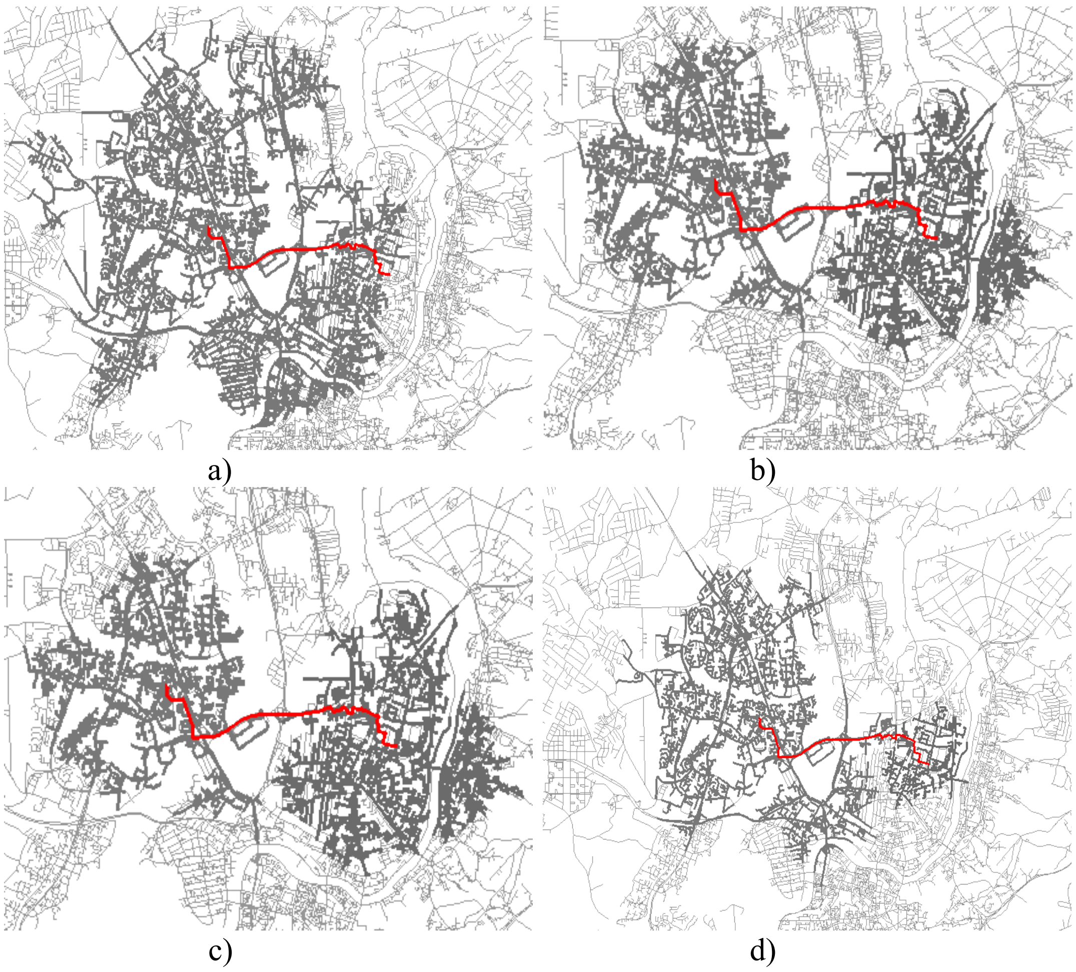

Fig. 22. The shortest path calculation: a) using the standard Dijkstra's algorithm; b) using the modified bidirectional Dijkstra's algorithm; c), d) using the proposed parallel scheme

The steady reading of data from the disk can cause delays, which distorts the results. In order to avoid that, a Lithuanian road network has been selected and placed in the computer memory before testing. Five random nodes A, B, C, D and E were selected in the graph in the same city and all the shortest paths between them were calculated. The tests were calculated by the standard Dijkstra's algorithm (Figure 22 a), the modified bidirectional Dijkstra's algorithm (Figure 22 b), and by the parallel algorithm proposed (Figure 22 c). Figure 22 d) shows the shortest path calculation, where the start of the second process is delayed because of the operating system loads. The test results are presented in Figures 23 and 24. These results indicate that the modified bidirectional Dijkstra's algorithm is still 2 times faster than the standard one. However, the parallel Dijkstra's algorithm is almost 2.9 times faster than the standard one and 1.4 times faster than the bidirectional Dijkstra's algorithm. So the efficiency $E(p)$ of the proposed parallel algorithm is 0.7 (p = 2), where T_1 is the execution time of the sequential algorithm, and T_p is the execution time of the parallel algorithm with p processors:

$$E(p) = \frac{S_p}{p}$$

$$S_p = \frac{T_1}{T_p}$$

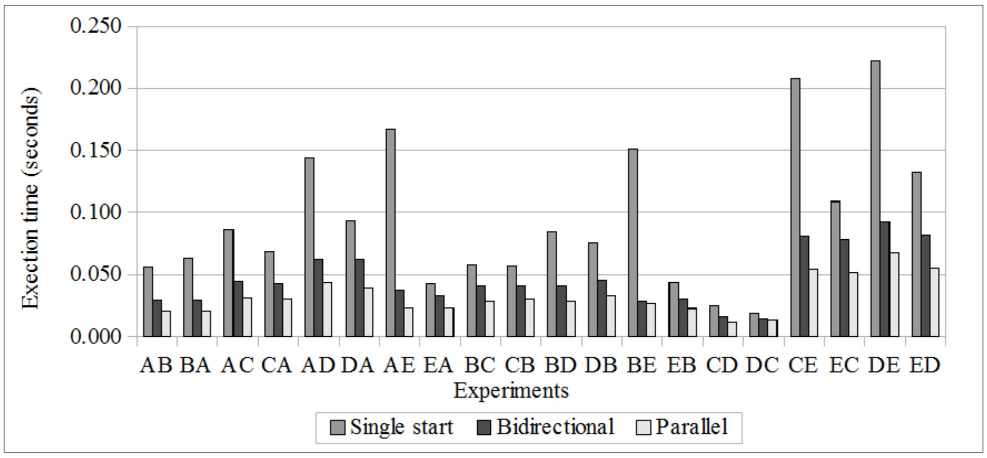

Fig. 23. The execution time of the calculation of the shortest path between nodes A B C D E

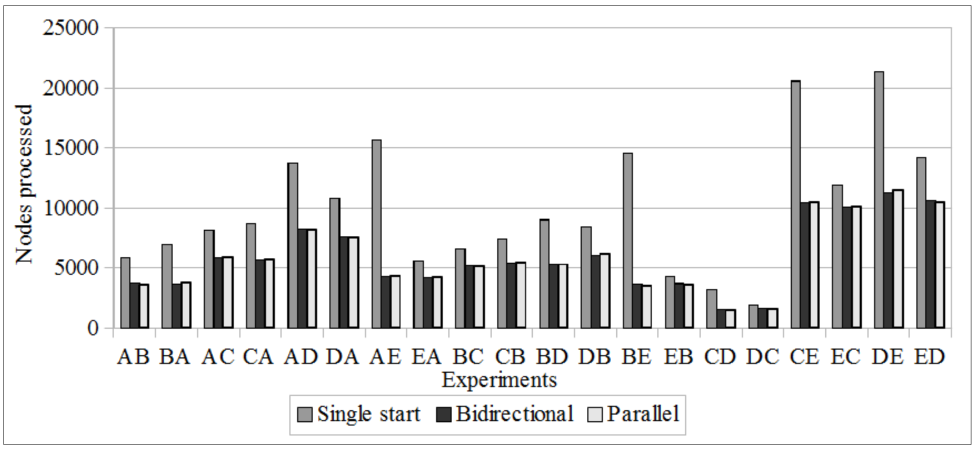

Fig. 24. The number of processed nodes in the calculation of the shortest path between nodes A B C D E

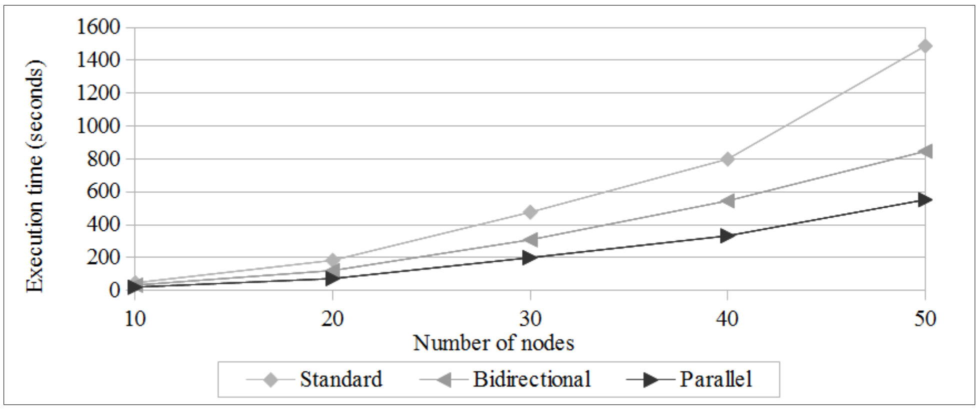

Fig. 25. The execution time of the calculation of all the shortest paths between random *k* nodes

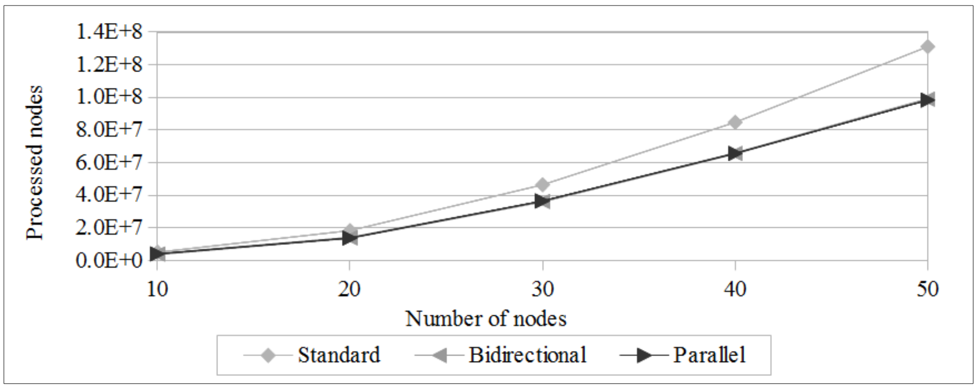

Fig. 26. The number of processed nodes in the calculation of all the shortest paths between random *k* nodes

Table 18. The execution time and the number of processed nodes in the calculation of all the shortest paths between random *k* nodes

	Execution time (seconds)			Processed nodes		
k	Standard	Bidirectional	Parallel	Standard	Bidirectional	Parallel
10	47.56	33.68	21.02	5271562	4163721	4244089
20	184.19	122.50	73.34	18511173	14073612	13961447
30	477.07	309.28	200.67	46566414	36361419	36652408
40	800.10	546.23	333.48	84806593	65769961	65779506
50	1488.55	847.68	553.10	131125250	98978796	98404846

Table 19. Average execution time and average numbers of processed nodes

	Execution time	Processed nodes
Standard	0.536	53738.97
Bidirectional	0.350	41530.77
Parallel	0.219	41671.60

In the second experiment, the total time of the shortest path calculations between all *k* randomly selected nodes was measured. The experiments were done by using the whole road network of Lithuania. Test results are presented in Table 18, Figure 25 and Figure 26. The results show that the calculation of all the shortest paths between all randomly selected 50 nodes lasted ~24 minutes on the same hardware. Meanwhile, the parallel Dijkstra's algorithm calculates the same shortest paths in ~9 minutes. The average execution time and the average numbers of processed nodes are presented in Table 19.

3.5. Summary

The proposed new genetic algorithm is applied to two different problems (VRPTW, VRPPD). As the results show, the proposed genetic algorithm finds solutions that in most cases are better or equal than the ones found by other genetic algorithms. The results are compared according to the objectives defined for the test problem instances. Although the found solutions are not equal to the best known solutions in all cases, they are found in a reasonably short time. However, no additional improvement/repair algorithms or local search algorithms are used here. That makes the proposed algorithm competitive with other known algorithms.

The new crossover operators that search for common parts between parent solutions are compared to other crossover operators that also deal with insertion heuristics for constructing feasible solutions. The proposed crossover operators are applied to VPRTW instances for comparison. The experimental evaluation shows that the new crossover operators, in most cases, find better solutions than other crossover operators. The computation time of the new crossover operator LCSX is similar to that of other crossovers, however, the found solutions are more accurate as compared to that found by the other crossovers. The solutions are found in the experimental evaluation by applying the mutation operator that randomly removes parts of the solutions and reconstructs the solution. However, such mutation operator was chosen just for the comparison of crossover operators. Different mutation operators could be used to find better solutions. Also, it is worth mentioning that some solutions are equal to the best known solutions even in the cases, where the mutation was not applied, however, no additional improvement approaches are used.

The proposed algorithm can be applied to any problem that can be expressed as a graph. Mutation and crossover operators of the proposed genetic algorithm are based on a random insertion heuristic. The operators are not designed to a certain specific problem and can be applied to different problems. The proposed algorithm can be applied in general cases.

The proposed parallel version of the Dijkstra's algorithm is almost 2.9 times faster than the standard one and 1.4 times faster than the bidirectional Dijkstra's algorithm. Although the evaluation was performed only between two places in the graph, the mentioned approach can be adapted to calculate the shortest-path between more than two places in the graph at the same time or even forward and backward at the same time.

The results of this chapter have been published in (Vaira and Kurasova, 2010; Vaira and Kurasova, 2011; Vaira and Kurasova, 2013; Vaira and Kurasova, 2013a; Vaira and Kurasova, 2014).

Conclusions

The research completed in this thesis has led to the following conclusions:

1. In contrast to crossover operators, where solutions are constructed from parts of the parent solutions, the proposed crossover operators, that search and preserve parts of the solution that are common to both parents, find the results that in most of the cases are more accurate than the ones found by other crossover operators. Some solutions are equal to the best known solutions even in the cases, where mutation was not applied.
2. As results of VRPTW instances show, the proposed algorithm, based on feasible reinsertion approach in genetic algorithm operators, on crossovers preserving common parts, and on the second population in mutation operator, finds better solutions for 4 out of 6 problem instance groups in comparison with other genetic algorithm approaches.
3. By repeatedly applying random insertion heuristic, the diversification is enabled in the population and, by dealing only with feasible solutions, infeasible search space is not examined, thus avoiding unnecessary computation and increasing overall computation speed. The solutions are found on average in 38.97 seconds. The proposed genetic algorithm performs ~2 times less floating point operations to find the results comparing to the best value of other algorithms.
4. The best solutions for VRPPD instances, obtained by the proposed algorithm, are equal to the best known solutions in ~96% cases. The found average solutions are equal to the best known solutions in ~80% cases.

5. The results of the shortest path search experiments indicate that the modified bidirectional Dijkstra's algorithm is 2 times faster than the standard one and the parallel Dijkstra's algorithm is almost 2.9 times faster than the standard one and 1.4 times faster than the bidirectional Dijkstra's algorithm.
6. Mutation and crossover operators in the proposed genetic algorithm are based on a random insertion heuristic. The operators are not designed to a certain specific problem and can be applied to different problems. The proposed algorithm can be applied for the rich vehicle routing problem. No additional repair or improvement methods are used that could be a problem for extending scheme with a new constraint handling. Proposed genetic operators do not break main genetic algorithm principles, so different objective functions can be applied to rank solutions in the population, including multi-objective approaches.

References

Alvarenga, G. B., de A. Silva, R. M., Sampaio, R. M. (2005). A Hybrid Algorithm for the Vehicle Routing Problem with Time Window, INFOCOMP Journal of Computer Science, 4(2), 9–16.

Anastopoulos, N., Nikas, K., Goumas, G., Koziris, N. (2009). Early Experiences on Accelerating Dijkstra's Algorithm Using Transactional Memory. Parallel & Distributed Processing, IPDPS 2009, IEEE, 1–8.

Anastopoulos, N., Nikas, K., Goumas, G., Koziris, N. (2009a). Employing Transactional Memory and Helper Threads to Speedup Dijkstra's Algorithm. In Proc. of International Conference on Parallel Processing (ICPP), Vienna, Austria, IEEE Computer Society, 388–395

Archetti, C., Speranza, M. G., Hertz, A. (2006). A Tabu Search Algorithm for the Split Delivery Vehicle Routing Problem. Transportation Science, 40 (1), 64–73.

Archetti, C., Speranza, M.G. (2012). Vehicle routing problems with split deliveries. International Transactions on Operations Research, 19, 3–22.

Baptista, S., Oliveira, R., Zuquete, E. (2002). A period vehicle routing case study. European Journal of Operational Research, 139 (2), 220–229.

Bard, J.F., Huang, L., Dror, M., Jaillet, P. (1998). A branch and cut algorithm for the VRP with satellite facilities. IIE Transactions, 30, 831–834.

Bektas, T., Erdogan, G., Ropke, S. (2011). Formulations and Branch-and-Cut Algorithms for the Generalized Vehicle Routing Problem. Transportation Science, 45 (3), 299–316

Bent, R., Hentenryck, P. V. (2003). A Two-Stage Hybrid Algorithm for Pickup and Delivery Vehicle Routing Problems with Time Windows. In Proceedings of the 9th International Conference on the Principles and Practice of Constraint Programming (CP 2003), volume 2833 of Lecture Notes in Computer Science, Springer, 123–137.

Berger, J., Barkaoui, M. (2004). A parallel hybrid genetic algorithm for the vehicle routing problem with time windows. Computers & Operations Research, 31, 2037–2053.

Berger, J., Salois, M., Begin, R. (1998). A Hybrid Genetic Algorithm for the Vehicle Routing Problem with Time Windows. In Proceedings of the 12th Biannual Conference of the Canadian Society for Computational Studies of Intelligence, volume 1418 of Lecture Notes in Computer Science, Springer, 114–127.

Berrettini, E., D'Angelo, G., Delling, D. (2009). Arc-Flags in Dynamic Graphs. In 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS 2009), Volume 09002 of Dagstuhl Seminar Proceedings, Germany.

Blanton, J. L., Wainwright, R. L. (1993). Multiple Vehicle Routing with Time and Capacity Constraints Using Genetic Algorithms. In proceedings of the 5th International Conference on Genetic Algorithms (ICGA), Morgan Kaufmann, 452–459.

Brandão J, (2004). A tabu search heuristic algorithm for open vehicle routing problem. European Journal of Operational Research, 157(3), 552–564.

Campbell, A. M., Savelsbergh, M. W. P. (2004). Efficient Insertion Heuristics for Vehicle Routing and Scheduling Problems. Transportation Science, 38(3), 369–378.

Černý, V. (1985). A thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm. Algorithm Journal of Optimization Theory and Applications, 45(1), 41–15.

Chan, W.., Zhang, Y., Fung, S., Ye, D., Zhu, H. (2007). Efficient algorithms for finding a longest common increasing subsequence. J. Comb. Optim., 13(3), 277–288.

Clarke, G., Wright, J. (1964). Scheduling of vehicles from a central depot to a number of delivery points. Operations Research, 12, 568–581.

Cordeau, J. F., Gendreau, M., Hertz, A., Laporte, G., Sormany, J. S. (2005). New Heuristics for the Vehicle Routing Problem. Logistics Systems: Design and Optimization (A. Langevin and D. Riopel, Eds.), Springer, 270–297.

Cordeau, J.-F., Laporte, G. (2003). A tabu search heuristic for the static multi-vehicle dial-a-ride problem (2003). Transportation Research Part B: Methodological, 37(6), 579–594.

Cordeau, J.F., Laporte, G., Mercier, A. (2001). A unified tabu search heuristic for vehicle routing problems with time windows. Journal of the Operational Research Society, 52 (8), 928–936.

Crainic, T.G., Mancini, S., Perboli, G., Tadei, R. (2010). Two-echelon vehicle routing problem: a satellite location analysis. PROCEDIA Social and Behavioral Sciences, 2, 5944–5955.

Dantzig, G. B., Ramser J. H. (1959). The truck dispatching problem. Management Science 6 (1959), 80–91.

Deep, K., Adane, H. M. (2011), New Variations of Order Crossover for Travelling Salesman Problem. IJCOPI 2(1), 2–13.

Dijkstra, E. (1959). A note on two problems in connexion with graphs. Numerische Mathematik, 1, 269–271.

Drexl, M. (2012). Rich vehicle routing in theory and practice. Logistics Research, 5(1-2), 47–63

Dzemyda, G., Sakalauskas, L. (2011). Large-Scale Data Analysis Using Heuristic Methods. Informatica, 22(1), 1–10.

Edmonds, N., Breuer, A., Gregor, D., Lumsdaine, A. (2006). Single-Source Shortest Paths with the Parallel Boost Graph Library. In 9th DIMACS Implementation Challenge – Shortest Paths.

El-Mihoub, T. A., Hopgood, A. A., Nolle, L., Battersby, A. (2006). Hybrid Genetic Algorithms: A Review. Engineering Letters, 13 (2), 124–137.

Erdogan, S., Miller-Hooks, E. (2012). A green vehicle routing problem. Transportation Research Part E: Logistics and Transportation Review, 48(1), 100–114.

Felinskas, G. (2007). Investigation of heuristic methods and application to optimization of resource constrained project schedules. Doctoral dissertation, Vytautas Magnus University.

Garcia-Najera, A., Bullinaria, J. A. (2011). An improved multi-objective evolutionary algorithm for the vehicle routing problem with time windows. Computers & OR, 38, 287–300.

Gendreau, M., Laporte, G., Musaraganyi, C., Taillard, É. D. (1999). A tabu search heuristic for the heterogeneous fleet vehicle routing problem. Computers & Operations Research, 26 (12), 1153–1173.

Gillett, B., Miller, L. (1974). A heuristic algorithm for the vehicle dispatch problem. Operations Research, 22, 340–349.

Goel, A., Gruhn, V. (2008). A General Vehicle Routing Problem. European Journal of Operational Research, 191(3), 650–660

Golberg, D. E., Deb, K. (1991). A comparative analysis of selection schemes used in genetic algorithms. Foundations of Genetic Algorithms, San Mateo, CA, Morgan Kaufmann, 69–93.

Goldberg, A.V., Kaplan, H., Werneck, R.F. (2006). Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In Workshop on Algorithm Engineering and Experiments (ALNEX), 129–143.

Haklay, M., Weber, P. (2008). OpenStreetMap: User-Generated Street Maps. IEEE Pervasive Computing, 7(4), 12–18.

Hartl, R., Hasle, G., Janssens, G. (2006). Special Issue on Rich Vehicle Routing Problems. Central European Journal of Operations Research, 14(1), 103 – 104.

Hasle, G., Kloster, O. (2007). Industrial Vehicle Routing Problems. Chapter in Hasle G., K-A Lie, E. Quak (eds): Geometric Modelling, Numerical Simulation, and Optimization – Applied Mathematics at SINTEF, Springer, 397–432.

Helsgaun, K. (2000). An effective implementation of the Lin-Kernighan traveling salesman heuristic. European Journal of Operational Research, 126 (1), 106–130.

Ho, W.-K., Ang, J. C., Lim, A. (2001). A Hybrid Search Algorithm for the Vehicle Routing Problem with Time Windows. International Journal on Artificial Intelligence Tools, 10 (3), 431–449.

Holland, J. (1975). Adaptation in Natural and Artificial Systems: An Introductory Analysis with applications to biology, Control and Artificial Intelligence. The University of Michigan Press.

Hong, T.-P., Wang, H.-S., Lin, W.-Y., Lee, W.-Y. (2002). Evolution of Appropriate Crossover and Mutation Operators in a Genetic Process. Applied Intelligence, 16 (1), 7–17.

Ichoua, S., Gendreau, M., Potvin, J.-Y. (2003). Vehicle dispatching with time-dependent travel times. European Journal of Operational Research, 144 (2), 379–396.

Iori, M., González, J. J. S., Vigo, D. (2007). An Exact Approach for the Vehicle Routing Problem with Two-Dimensional Loading Constraints. Transportation Science, 41 (2), 253–264.

Ishikawa, H., Shimizu, S., Arakawa, Y., Yamanaka, N., Shiba, K. (2007). New Parallel Shortest Path Searching Algorithm based on Dynamically Reconfigurable Processor DAPDNA-2. In Proceedings of IEEE

International Conference on Communications, ICC 2007, Glasgow, Scotland, IEEE, 1997–2002.

Jančauskas, V., Kaukas, G., Žilinskas, A., Žilinskas, J. (2012). On Multi-Objective Optimization Aided Visualization of Graphs Related to Business Process Diagrams. In: A. Čaplinskas, G. Dzemyda, A. Lupeikienė, O. Vasilecas (Eds.), Local Proceedings and Materials of Doctoral Consortium of the Tenth International Baltic Conference on Databases and Information Systems, 71-80.

Jih, W., Chen, Y., Hsu, Y. (1996). A Comparative Study of Genetic Algorithms for Vehicle Routing with Time Constraints. Proceedings of the 1996 International Computer Symposium, 17–24.

Jih, W., Hsu, Y. (2004). A family competition genetic algorithm for the pickup and delivery problems with time window. Bull. Coll. Eng. N.T.U. 90, 89–98.

Jung, S., Moon, B. R. (2002). A Hybrid Genetic Algorithm For The Vehicle Routing Problem With Time Windows. In Proceedings of Genetic and Evolutionary Computation Conference (GECCO 2002), Morgan Kaufmann, 1309–1316.

Knopp, S., Sanders, P., Schultes, D., Schulz, F., Wagner, D. (2007). Computing Many-to-Many Shortest Paths Using Highway Hierarchies. In Proceedings of the Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, 36–45.

Koehler, E., Moehring, R. H., Schilling, H. (2006). Fast Point-to-Point Shortest Path Computations with Arc-Flags. In Proc. of 9th DIMACS Implementation Challenge – Shortest Paths.

Kumar, N., Karambir, Kumar, R. (2012). A Comparative Analysis of PMX, CX and OX Crossover operators for solving Travelling Salesman Problem. International Journal of Latest Research in Science and Technology, 1(2), 98–101.

Kutz, M., Brodal, G. S., Kaligosi, K., Katriel, I. (2011). Faster algorithms for computing longest common increasing subsequences. J. Discrete Algorithms, 9(4), 314–325

Lančinskas, A. (2013). Parallelization of random search global optimization algorithms. Doctoral dissertation, Vilnius University.

Lančinskas, A., Ortigosa, P.M., Žilinskas, J. (2013). Multi-objective single agent stochastic search in non-dominated sorting genetic algorithm. Nonlinear Analysis: Modelling and Control, 18(3), 293-313.

Laporte, G., Gendreau, M., Potvin, J.-Y., Semet, F. (1999). Classical and Modern Heuristics for the Vehicle Routing Problem. Les Cahiers du GERAD, G98-54, Group for Research in Decision Analysis, Montreal, Canada.

Li, H., Lim, A. (2003). A Metaheuristic for the Pickup and Delivery Problem with Time Windows. International Journal on Artificial Intelligence Tools, 12(2), 173–186.

Lin, S., Kernighan, B. W. (1973). An Effective Heuristic Algorithm for the Traveling-Salesman Problem. Operations Research, 21, 498–516.

Lu, C.-L., Chen, Y. (2006). Using Multi-Thread Technology Realize Most Short-Path Parallel Algorithm. Enformatika, 15 (11), 11–13.

Lukasiewycz, M., Glass, M., Haubelt, C., Teich, J. (2008). A feasibility-preserving local search operator for constrained discrete optimization problems. In Proceedings of IEEE Congress on Evolutionary Computation, 1968–1975.

Lukasiewycz, M., Glass, M., Teich, J. (2008a). A Feasibility-Preserving Crossover and Mutation Operator for Constrained Combinatorial Problems. Proceedings of the 10th International Conference on Parallel Problem Solving from Nature (PPSN), Volume 5199 of Lecture Notes in Computer Science, Springer, 919–928.

Lysgaard, J., Letchford, A.N., Eglese R.W. (2004). A new branch-and-cut algorithm for the capacitated vehicle routing problem. Mathematical Programming, 100, 423–445.

Mačiūnas, D. (2013). Multi-objective global optimization of grillages using genetic algorithms. Doctoral dissertation, Vilnius Gediminas Technical University.

Madduri, K., Bader, D. A., Berry, J. W., Crobak, J. R. (2007). An Experimental Study of A Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances. In Proceedings of the Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA.

Michalewicz, Z. (1995). Do Not Kill Unfeasible Individuals. In Proceedings of the 4th Intelligent Information Systems Workshop (IIS'95), 110–123

Michalewicz, Z. (1995a). A Survey of Constraint Handling Techniques in Evolutionary Computation Methods. Evolutionary Programming, 135–155.

Misevičius, A. (2003). A Modified Simulated Annealing Algorithm for the Quadratic Assignment Problem. Informatica, Lith. Acad. Sci., 14 (4), 497–514.

Misevičius, A. (2009). Testing of Hybrid Genetic Algorithms for Structured Quadratic Assignment Problems. Informatica, Lith. Acad. Sci., 20 (2), 255-272.

Misevičius, A., Kilda, B. (2005). Comparison of crossover operators for the quadratic assignment problem. Information Technology and Control, 34(2), 109–119.

Nagata, Y., Bräysy, O. (2009). Edge assembly-based memetic algorithm for the capacitated vehicle routing problem. Networks, 54(4), 205–215.

Ombuki, B. M., Nakamura, M., Maeda, O. (2002). A hybrid search based on genetic algorithms and tabu search for vehicle routing. In 6th IASTED Intl. Conf. On Artificial Intelligence and Soft Computing (ASC 2002), edited by A.B. Banff, H Leung, ACTA Press, 176–181.

Ombuki, B. M., Ross, B., Hanshar, F. (2006). Multi-Objective Genetic Algorithms for Vehicle Routing Problem with Time Windows. Applied Intelligence, 24(1), 17–30.

Pisinger, D., Ropke, S. (2009). Large neighborhood search. Handbook of Metaheuristics, 2nd edition, M. Gendreau and J.-Y. Potvin(eds).

Potvin, J.-Y., Bengio, S. (1996). The Vehicle Routing Problem with Time Windows Part II: Genetic Search. INFORMS Journal on Computing, 8(2), 165–172.

Potvin, J.-Y., Dubé, D. (1994). Improving a Vehicle Routing Heuristic Through Genetic Search. Proceedings of the 1st IEEE Conference on Evolutionary Computation, 194–199.

Potvin, J.-Y., Rousseau, J.M. (1993). A parallel route building algorithm for the vehicle routing and scheduling problem with time windows. European Journal of Operational Research, 66, 331–240.

Redondo, J. L., Ortigosa, P. M., Žilinskas, J. (2012). Multimodal Evolutionary Algorithm for Multidimensional Scaling with City-Block Distances. Informatica, 23 (4), 601-620.

Reid, D. J. (2000). Feasibility and Genetic Algorithms: the Behaviour of Crossover and Mutation. DSTO Electronics and Surveillance Research Laboratory.

Rizzoli, A. E., Montemanni, R., Lucibello, E., Gambardella, L. M. (2007). Ant colony optimization for real-world vehicle routing problems. Swarm Intelligence, 1(2), 135–151.

Romeijn, H. E., Smith, R. L. (1999). Parallel Algorithms for Solving Aggregated Shortest Path Problems. Computers & Operations Research, (26), 941–953.

Ropke, S., Pisinger, D. (2006). A unified heuristic for a large class of Vehicle Routing Problems with Backhauls. European Journal of Operational Research, 171 (3), 750–775.

Rosenkrantz, D. J., Stearns, R. E., Lewis II, P. M. (1977). An Analysis of Several Heuristics for the Traveling Salesman Problem. SIAM J. Comput., 6(3), 563–581.

Schensted, C. (1961), Longest increasing and decreasing subsequences. Canad. J. Math. 13, 179–191.

Schneider, M., Stenger, A., Goeke, D. (2012). The Electric Vehicle Routing Problem with Time Windows and Recharging Stations. In Tech. Report 02/2012, BISOR, TU Kaiserslautern.

Šešok, D. (2008). Topology optimization of truss structures using genetic algorithms. Doctoral dissertation, Vilnius Gediminas Technical University.

Solomon, M.M. (1987). Algorithms for the Vehicle Routing and Scheduling Problems with Time Window Constraints. Operations Research, 35(2), 254–265.

Srinivas, M., Patnaik, L. M. (1994). Adaptive probabilities of crossover and mutation in genetic algorithms. IEEE Transactions on Systems, Man, and Cybernetics, 24(4), 656–667.

Tan, K. C., Chew, Y. H., Lee, L. H. (2006). A Hybrid Multiobjective Evolutionary Algorithm for Solving Vehicle Routing Problem with Time Windows. Computational Optimization and Applications, 34 (1), 115–151.

Tan, K. C., Hay, L. L., Ke., O. (2001). A hybrid genetic algorithm for solving vehicle routing problems with time window constraints. Asia-Pacific Journal of Operational Research, 18(1), 121–130

Tan, K. C., Lee, L. H., Zhu, K. Q., Ou, K. (2001a). Heuristic methods for vehicle routing problem with time windows. Artificial Intelligence in Engineering, 15(3), 281–295.

Thangiah, S. R., Vinayagamoorty, R., Gubbi, A. V. (1993). Vehicle Routing and Time Deadlines Using Genetic and Local Algorithms. In proceedings of the 5th International Conference on Genetic Algorithms (ICGA), Morgan Kaufmann, 506–515.

Thangiah, S., Nygard, K., Juell, P. (1991). GIDEON: A genetic algorithm system for vehicle routing with time windows. In 7th Conference on Artificial Intelligence Applications, 322–328.

Tommiska, M., Skytta, J. (2001), Dijkstra's Shortest Path Routing Algorithm in Reconfigurable Hardware., in G. J. Brebner., R. Woods, ed., Field-Programmable Logic and Applications, 11th International Conference, Springer, 2147, 653–657.

Toth, P., Vigo, D. (2001). Branch-and-bound algorithms for the capacitated VRP. Society for Industrial and Applied Mathematics, 29–51.

Toth, P., Vigo, D. (2002). The Vehicle Routing Problem, Society for Industrial and Applied Mathematics.

Vatsavai, R. R., Shekhar, S., Burk, T. E. and Lime, S. (2006), UMN-MapServer: A High-Performance, Interoperable, and Open Source Web Mapping and Geo-spatial Analysis System., in M. Raubal; H. J. Miller; A. U. Frank and M. F. Goodchild, ed., GIScience, Springer, 400–417.

Vidal, T., Crainic, T. G., Gendreau, M., Prins, C. (2013). Heuristics for multi-attribute vehicle routing problems: A survey and synthesis. European Journal of Operational Research, 231 (1), 1–21.

Von Lossow, M. (2007). A min-max version of Dijkstra's algorithm with application to perturbed optimal control problems. In Proceedings in Applied Mathematics and Mechanics ICIAM 2007/GAMM 2007, 7, Zurich, Switzerland.

Yang, I, Huang, C., Chao, K. (2005). A fast algorithm for computing a longest common increasing subsequence. Inf. Process. Lett., 93(5), 249–253.

Yeniay, O. (2005). Penalty function methods for constrained optimization with genetic algorithms. Mathematical and Computational Applications, 10(1), 45–56.

Yeun, L. C., Ismail, W. R., Omar, K., Zirour, M. (2008). Vehicle Routing Problem: Models and Solutions, Journal of Quality Measurement and Analysis (JQMA), 4(1), 205–218.

Zhang, J., Chung, H. S.-H., Hu, B.J. (2004). Adaptive probabilities of crossover and mutation in genetic algorithms based on clustering technique Evolutionary Computation. In Proceedings of IEEE Congress on Evolutionary Computation (CEC2004), 2, 2280–2287.

Zhang, J., Chung, H. S.-H., Lo, W.-L. (2007). Clustering-Based Adaptive Crossover and Mutation Probabilities for Genetic Algorithms. IEEE Transactions on Evolutionary Computation, 11(3), 326–335.

Zhong, J., Hu, X., Gu, M., Zhang, J. (2005). Comparison of Performance between Different Selection Strategies on Simple Genetic Algorithms.

Proceeding of the International Conference on Computational Intelligence for Modelling, Control and automation, and International Conference of Intelligent Agents, Web Technologies and Internet Commerce (CIMCA/IAWTIC), IEEE Computer Society, 1115–1121

Zhu, K. Q. (2003). A Diversity-Controlling Adaptive Genetic Algorithm for the Vehicle Routing Problem with Time Windows. Proceedings of the 15th IEEE International Conference on Tools for Artificial Intelligence (ICTAI 2003), 176–183.

Žilinskas, A., Žilinskas, J. (2007). Parallel genetic algorithm: assessment of performance in multidimensional scaling. In proceedings of the 9th annual conference on Genetic and evolutionary computation (GECCO 2007), Association for Computing Machinery (ACM), 1492-1499.

Žilinskas, J. (2008). On dimensionality of embedding space in multidimensional scaling. Informatica, 19(3), 447-460.