International Journal on Advanced Science Engineering Information Technology

Vol.8 (2018) No. 3
ISSN: 2088-5334

Different Applications of the Genetic Mutation Operator for Symetric Travelling Salesman Problem

Velin Kralev

Department of Informatics, South-West University "Neofit Rilski", 66 Ivan Michailov Str., Blagoevgrad, 2700, Bulgaria
E-mail: velin_kralev@swu.bg

***Abstract*— This paper presents the results of an analysis of three algorithms for the Travelling Salesman Problem (TSP). The basic steps of genetic algorithms (GAs) and their benefits in solving combinatorial optimization problems are also presented. Moreover, several studies related to TPS and some approaches to its solution are discussed. An optimized version of the standard recursive algorithm for solving TSP using the backtracking method is presented. This algorithm is used to generate optimal solutions concerning the studied graphs. In addition, a standard genetic algorithm for solving TSP and its modification are also presented. The modified algorithm uses the genetic operator mutation in a different way. The results show that the recursive algorithm can be used successfully to solve the TSP for graphs with a small number of vertices, for instance, 25-30. The results of the two GAs were different. The modified GA found the optimal solutions for all tested graphs, while the standard GA found the optimal solutions in only 40% of the cases. These results were obtained for a reasonable time (in seconds), with appropriate values of the control parameters - population size and reproduction number. It appeared that the use of the genetic mutation operator yields better results when applied to identical solutions. If pairs of identical solutions are found in a population, then every second must mutate. The methodology and the conditions for conducting the experiments are described in details.**

***Keywords*— travelling salesman problem; genetic algorithm; crossover; mutation**

I. INTRODUCTION

In the recent years, the interest in GAs [1]–[3] and their modifications [4]–[6] has increased significantly. These algorithms generate optimal (or near optimal) solutions for a reasonable time [7]. What is typical of most heuristic approaches is that they work well for certain problems, but they hardly adapt from one problem to another [8]–[10]. Therefore, the development of more adaptive approaches that work effectively for different problems will be further explored.

For an important class of NP-hard problems, such as scheduling problems and graph problems, good approximate solutions were found using GAs [11]–[13]. This class of problems are known for being similar but not identical to the problems for which there are efficient algorithms. A small change in the problem definition can result in a great change in the performance of the best known algorithm [14].

When an algorithm uses the backtracking method, it checks a large number of possible solutions. Exploring all solutions is often unnecessary. Therefore, a number of methods have been proposed to reduce the number of solutions considered. Applying such approaches increases the performance of the algorithms used. This will also be shown in this paper as well, with presenting the results of the experiments.

Unlike the exact algorithms, the genetic ones are approximated and are based on ideas borrowed from nature. These algorithms process a collection (population) of possible solutions (individuals). These solutions are combined and modified to generate better ones. Each solution is presented in a digital form and evaluated based on a predefined criterion of optimality.

After generating all solutions in the initialization population, each one can be left unchanged in the next population, can be re-combined with another solution to get a new one or can be removed. Some of the solutions can be changed (i.e., mutated), with a predetermined probability [15].

Each iteration of GA generates a new generation (with individuals, i.e., solutions), thus performing a reproduction process. Defining whether a solution is better or worse is done by a fitness function. This function "evaluates" each solution (according to the criterion of optimality). The solutions are classified (as better or worse) based on these estimates [16]–[18].

The steps that one GA performs are schematically presented in Fig. 1 ÷ Fig. 10.

Step 1. At first, a number of acceptable solutions is created. They form the initialization population $P_0 = \{S_1, S_2, ..., S_k\}$. These solutions are the basis for the formation of the next generations and are most often generated randomly (Fig. 1).

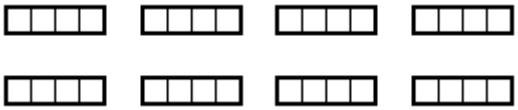

Fig. 1 Step 1 of GA

Step 2. The fitness function evaluates each solution according to the criterion of optimality. In this way, each solution is compared with a quantitative measure of its quality (Fig. 2).

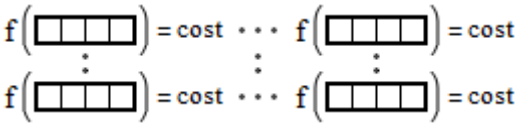

Fig. 2 Step 2 of GA

Step 3. All solutions are ordered descending according to their costs (calculated in step 2). In this way a selection of certain solutions can be made (Fig. 3).

Fig. 3 Step 3 of GA

Step 4. A number of solutions (such as k/2 or other even number) that have the best costs are selected. Since all solutions are ordered descending (in step 3), the first several solutions will be selected from the population. Other solutions will be removed. The number of selected solutions may vary from 10% to 50% (Fig. 4).

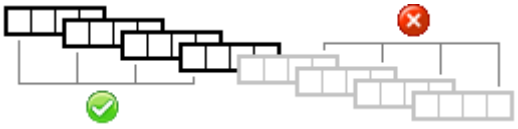

Fig. 4 Step 4 of GA

Step 5. The selected solutions (in step 4) are combined in pairs, for instance k/4 in number. The ways of doing so may be different, for example, by random means or by the two best successive solutions (Fig. 5).

Fig. 5 Step 5 of GA

Step 6. The combined pairs of solutions (total k/4) play the role of parents. By applying the crossover, each parent pair generates one, two or more new solutions. The total number of these solutions must match the number of solutions removed in step 4: k/2 (Fig. 6).

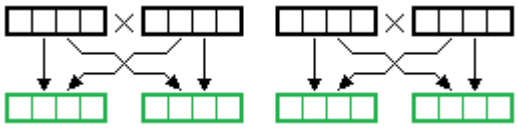

Fig. 6 Step 6 of GA

Step 7. Some of the new solutions are modified by applying the genetic mutation operator. This operator modifies a solution by changing one or more of its genes. ~~Generally,~~ The number of modified genes is small (e.g., up to 10% of the total number of genes in the solution). This genetic operator can also be applied in other cases, for example, when the population has identical solutions (Fig. 7).

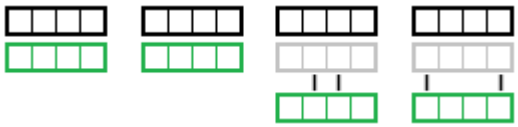

Fig. 7 Step 7 of GA

Step 8. The new solutions are obtained as a result of the implementation of the genetic operators crossing and mutation. These solutions are also evaluated by the fitness function (Fig. 8).

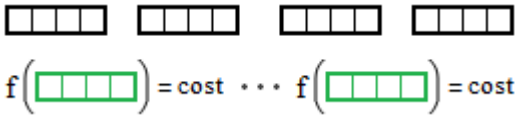

Fig. 8 Step 8 of GA

Step 9. At this step of GA, parents and descendants are united. In this way the new P_{t+1} population is formed (Fig. 9).

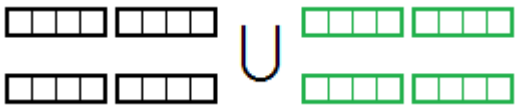

Fig. 9 Step 9 of GA

Step 10. Creating new generations continues until the algorithm end criterion is met. Such a criterion, for example, is the creation of a certain number of generations, or the number of generations after which there is no improvement to the last best solution found (Fig. 10).

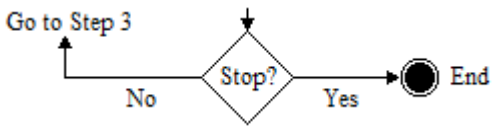

Fig. 10 Step 10 of GA

GAs are widely used to solve a large number of optimization problems from various fields of science, such as the graph theory. This is a part of computer science, which has an excessive practical application. In many cases, the analysis and description of different systems is done successfully with graph structures [19]. One major class of graph theory problems – NP-hard, can be solved well by some approximate algorithms such as GAs [20]–[24]. Finding an exact solution to these problems (with a large input size) can take a long time [25]. The approaches based

on the backtracking method yield the best results but only for problems with small input size. This method, though possible, is practically inapplicable. For instance, in a complete graph with 26 vertices and 26*(26-1)/2=325 edges, the number of all possible Hamiltonian cycles is considerable, respectively: (26-1)!/2=7 755 605 021 665 490 000 000 000. Therefore, the modifying and the improvement of existing heuristic algorithms is understandable [26]–[29].

The problem to find a minimal Hamiltonian cycle in a graph is an optimization problem. It is also known as the Travelling Salesman Problem, and its variants and detailed description are presented in [30] and [31]. There are several approaches and a large number of algorithms for TSP solving [32]–[34]. However, only two of them are basic - the exact and approximate methods. The exact algorithms always find the best solution, but a lot of computational time is required. It is applicable when the number of the vertices in a graph is small [35] and [36]. On the other hand, approximated algorithms find a solution that is close to the optimal one, and the time for this is acceptable. Such algorithms have been discussed in [37]–[39].

II. Material And Algorithm

This section presents the description and the analysis of three different algorithms which can be used to solve TSP. The first algorithm is based on the backtracking method, and it always finds the exact solution. The other two algorithms are GA modifications which can be used to solve approximate TSP. The main results that will be analyzed are the length of the found Hamiltonian cycle (it should be as small as possible) and the algorithms execution time.

```
01 var
02 | VertexCount: Integer;
03 | CycleLength: Integer;
04 | MinimalCycleLength: Integer;
05 | HamiltonianCycle: array of Integer;
06 | Individuals: Integer;
07 | Generations: Integer;
08 | MarkedArray: array of Boolean;
09 | ScoreArray: array of Integer;
10 | Population: array of array of Integer;
11 | AdjacencyMatrix: array of array of Integer;
12 initialization
13 | SetVertexCount(30);
14 | SetIndividuals(640);
15 | SetGenerations(1000);
16 | SetLength(MarkedArray, VertexCount+1);
17 | SetLength(ScoreArray, VertexCount+1);
18 | SetLength(Population,
19 |  Individuals+1, VertexCount+1);
```

Fig. 11 Code of the global declarations

For the implementation of algorithms and the parameter analysis, it is necessary to pre-declare and initialize (with the appropriate methods) some global data structures (dynamic arrays and variables) as shown in Fig. 11 (in Delphi).

~~An~~ Optimization of a recursive algorithm [26] to find a minimal Hamiltonian cycle in a complete undirected graph is shown in Fig. 12. This algorithm based on the Depth-first search (DFS) approach. The FindMinimalHamiltonianCycle procedure is recursive and calls itself on lines 26 and 27. When the last vertex from the graph - VertexCount is added to the constructed path (line 8), the length of the resulting Hamiltonian cycle is stored as minimal. The verification of whether the generated cycle is minimal or not is performed at the previous recursive procedure call of line 24. After every recursive procedure call to the current length of the constructed path, the length of the edge that connects the Iteration and J vertices is added (lines 22 and 23). If the length of the currently constructed cycle (albeit incomplete) is greater than the length of the last one, it is a step back and the last added edge is removed from the constructed path (lines 28 and 30). In this way, the search process is optimized, as all other possible extensions are not made. This optimization does not reduce the complexity of the algorithm, which remains exponential.

```
01 procedure FindMinimalHamiltonianCycle
02 | (Iteration, Position, VertexCount: Integer);
03 var
04 | J: Integer;
05 begin
06 | if ((Iteration = 1) and (Position > 1)) then
07 | begin
08 | | if (Position = (VertexCount + 1)) then
09 | | begin
10 | | | MinimalCycleLength := CycleLength;
11 | | end;
12 | | Exit;
13 | end;
14 | if (MarkedArray[Iteration] = True) then Exit;
15 | MarkedArray[Iteration] := True;
16 | for J := 1 to VertexCount do
17 | begin
18 | | if ((AdjacencyMatrix[Iteration,J] > 0) and
19 | |   (J <> Iteration)) then
20 | | begin
21 | | | HamiltonianCycle[Position] := J;
22 | | | CycleLength := CycleLength +
23 | | | AdjacencyMatrix[Iteration,J];
24 | | | if (CycleLength < MinimalCycleLength) then
25 | | | begin
26 | | | | FindMinimalHamiltonianCycle
27 | | | | (J, Position + 1, VertexCount);
28 | | | end;
29 | | | CycleLength := CycleLength -
30 | | | AdjacencyMatrix[Iteration,J];
31 | | end;
32 | end;
33 | MarkedArray[Iteration] := False;
34 end;
```

Fig. 12 Code of the recursion based algorithm

Two other algorithms for finding a minimal Hamiltonian cycle in a complete undirected graph will be presented. The first is a standard genetic algorithm (SGA) that performs the basic steps presented in Fig. 1÷10. The second algorithm (MGA) is a modification of the first. The difference being in the mode of application of the genetic mutation operator. The main difference is the application of the genetic mutation operator. The necessary methods (procedures and functions) that are common to both algorithms have been implemented in advance.

The method of evaluating the quality of a solution is CalculateScore (Fig. 13). This function receives as an input parameter a solution index, and as a result, it returns its score. The computational complexity of this method is linear~~,~~ since only one loop of the vertices in the graph is performed (line 10). This process starts from the edge that connects the latter with the first vertex (line 6-9). All weights of edges that connect every two consecutive remaining vertices are also added (lines 12-14).

```
01 function
02 | CalculateScore(Individual: Integer): Integer;
03 var
04 | Index: Integer;
05 begin
06 | Result :=
07 |   AdjacencyMatrix[
08 |     Population[Individual,VertexCount],
09 |     Population[Individual,1]];
10 | for Index := 1 to VertexCount-1 do
11 | begin
12 | | Inc(Result, AdjacencyMatrix[
13 | |   Population[Individual,Index],
14 | |   Population[Individual,Index+1]]);
15 | end;
16 end;
```

Fig. 13 Code of the CalculateScore fucntion

The method for generating the initial population is GenerateRandomIndividuals (Fig. 14). This procedure generates random Hamiltonian cycles to fill the initial population (lines 7-19). The computational complexity of this method is $\Theta(n.m)$, where n is the number of individuals in the population, and m is vertices in the graph. All generated Hamilton cycles are evaluated with the fitness function – CalculateScore (lines 20-21).

```
01 procedure GenerateRandomIndividuals;
02 var
03 | Individual, Gene, Value, T: Integer;
04 begin
05 | for Individual := 1 to Individuals do
06 | begin
07 | | for Gene := 1 to VertexCount do
08 | | begin
09 | | | MarkedArray[Gene] := False;
10 | | | Value := Random(VertexCount-Gene);
11 | | | T := 0;
12 | | | repeat
13 | | | | while MarkedArray[T] do Inc(T);
14 | | | | Dec(Value); Inc(T);
15 | | | until (Value = 0);
16 | | | Dec(T);
17 | | | Population[Individual,Gene] := T;
18 | | | MarkedArray[T] := True;
19 | | end;
20 | | ScoreArray[Individual] :=
21 | | CalculateScore(Individual);
22 | end;
23 end;
```

Fig. 14 Code of the GenerateRandomIndividuals procedure

The OrderIndividualsByScore method performs ordering of individuals in the population depending on their scores, i.e., depending on the lengths of the formed Hamiltonian cycles (Fig. 15). The computational complexity of this method is $\Theta(m.n^2)$, where n is the number of individuals in the population, and m is vertices in the graph. When changing the order of the individuals in the population (lines 5 and 7), all elements of the solution are also copied (lines 11-13). After that, the scores of the solutions also exchanged (line 14).

The Swap procedure is pre-declared. It gets two parameters that are passed by address and then exchanging their values. Since the computational complexity of this method is determined by the population size (rather than by the number of the vertices in the graph) when the number of individuals is large, it is better to replace selection sort method by another one, such as a quick sort or merge sort.

```
01 procedure OrderIndividualsByScore;
02 var
03 | Index, T, V: Integer;
04 begin
05 | for Index := 1 to (Individuals-1) do
06 | begin
07 | | for T := (Index+1) to Individuals do
08 | | begin
09 | | | if (ScoreArray[T]< ScoreArray[Index]) then
10 | | | begin
11 | | | | for V := 1 to VertexCount do
12 | | | | | Swap(Population[Index,V],
13 | | | | |      Population[T,V]);
14 | | | | Swap(ScoreArray[Index], ScoreArray[T]);
15 | | | end;
16 | | end;
17 | end;
18 end;
```

Fig. 15 Code of the OrderIndividualsByScore procedure

One of the essential methods of GA is the one that performs the genetic crossover operator. This method has been implemented in the DoCrossover procedure (Fig. 16).

```
01 procedure DoCrossover(P1, P2, F1, F2: Integer);
02 var
03 | Index, Left, Right: Integer;
04 begin
05 | Left := Random(VertexCount);
06 | repeat
07 | | Right := Random(VertexCount);
08 | until (Right <> Left);
09 | if Left > Right then Swap(Left, Right);
10 | for Index := 1 to Left-1 do
11 | begin
12 | | Population[F1,Index]:= Population[P1,Index];
13 | | Population[F2,Index]:= Population[P2,Index];
14 | end;
15 | for Index := Left to Right-1 do
16 | begin
17 | | Population[F1,Index]:= Population[P2,Index];
18 | | Population[F2,Index]:= Population[P1,Index];
19 | end;
20 | for Index := Right to VertexCount do
21 | begin
22 | | Population[F1,Index]:= Population[P1,Index];
23 | | Population[F2,Index]:= Population[P2,Index];
24 | end;
25 | ScoreArray[F1] := CalculateScore(F1);
26 | ScoreArray[F2] := CalculateScore(F2);
27 end;
```

Fig. 16 Code of the DoCrossover procedure

The parameters P1 and P2 are the indices of solutions from the current population that are selected for parents. The other two parameters, F1 and F2, are indexes of solutions from the second half of the current population that will be replaced by the generated descendants. The individuals in the population are ordered by their score, which means that the worst solutions are in the second half of the population. The crossing points are determined by the values of the Left and Right variables (set in lines 5-9). These values are generated in such a way that they are random, different, and Left is less than Right. The new solutions are obtained by copying different sections of genes from both parents, then combining them. In this way of crossing it is possible in some of the new solutions to appear repetitive and respectively missing vertices. This problem is solved by repeating vertices being replaced by missing ones. After generating new solutions, their scores are calculated using the CalculateScore function (lines 25-26). The complexity of

this method is $\Theta(m^2)$, where m is the number of vertices in the graph.

Another important method of GA is the one that performs the genetic mutation operator. This method is implemented in the DoMutate procedure (Fig. 17). This procedure obtains the index of a particular solution as an input parameter. The exchange positions (genes) are determined by the values of two variables - LeftGene and RightGene (lines 5-8). These values are generated in a similar way as in the DoCrossover procedure, i.e., they are random and different, but the value of LeftGene is not obligatorily smaller than the value of RightGene. The new solution is obtained after the values of these two positions are exchanged through the Swap procedure (lines 9-11). After generating the new solution, its score is calculated using the CalculateScore function (lines 12-13). The computational complexity of this method is a constant because only one individual of the population and two of its genes are changed (i.e., only two vertices are counted in the graph, not all of them).

```
01 procedure DoMutate(Individual: Integer);
02 var
03 | LeftGene, RightGene: Integer;
04 begin
05 | LeftGene := Random(VertexCount);
06 | repeat
07 | | RightGene := Random(VertexCount);
08 | until (RightGene <> LeftGene);
09 | Swap(
10 |   Population[Individual,LeftGene],
11 |   Population[Individual,RightGene]);
12 | ScoreArray[Individual] :=
13 | CalculateScore(Individual);
```

Fig. 17 Code of the DoMutate procedure

In the present study, two variants of the reproduction generation procedure will be used and analyzed. In the first variant, the standard steps of the GA, which were presented in the previous section, are implemented. The code of the StandardReproduce procedure is presented in Fig. 18.

```
01 procedure StandardReproduce;
02 var
03 | Generation, Individual, I: Integer;
04 | IsEqual: Boolean;
05 begin
06 | GenerateRandomIndividuals;
07 | for Generation := 1 to Generations do
08 | begin
09 | | OrderIndividualsByScore;
10 | | Individual := 1;
11 | | while (Individual < (Individuals div 2)) do
12 | | begin
13 | | | DoCrossover(Individual, Individual+1,
14 | | |   Individuals-Individual,
15 | | |   Individuals-(Individual-1));
16 | | | Inc(Individual,2);
17 | | end;
18 | | for I := 1 to Round(Individuals*0.1) do
19 | | begin
20 | | | DoMutate(Individuals –
21 | | |   Random((Individuals div 2)));
22 | | end;
23 | end;
24 end;
```

Fig. 18 Code of the StandardReproduce procedure

Initially, steps 1 and 2 (combined in the GenerateRandom Individuals method, line 6) are performed. This method generates the initial population, while also invoking the CalculateScore function to evaluate each generated solution. The reproduction generation process begins at line 7. The number of these reproductions is determined by the value of the Generations variable, which is initialized at the beginning of the program. Creating a new population starts with the implementation of step 3 on line 9. Then (combined) steps 4, 5, 6, 8 and 9 (lines 10-17) are performed. After the execution of the crossover operator, 20% of the new solutions mutate. This is equivalent to 10% (0.1) of the population size. This is done on lines 18-22 and corresponds to a combined execution of steps 7, 8 and 9. Checking for the end of the process of generating new reproductions is done at line 22 (corresponds to step 10 of the GA). If the specified number of reproductions (the value of the Generations variable) is reached, the transition to step 3 (line 9) is not done. Otherwise the process of generating the next reproduction will continue. The computational complexity of this procedure is quadratic, depending on the number of vertices in the graph and the number of individuals in the population (the values of the VertexCount and Individuals variables). In the analysis of the computational complexity, the value of the Generations variable can also be included as the overall complexity of the program depends linearly on the number of generated reproductions.

A modified reproduction generation procedure that is different from the standard one will be presented (Fig. 19).

```
01 procedure ModifiedReproduce;
02 var
03 | Generation, Individual, I, Gene: Integer;
04 | IsEqual: Boolean;
05 begin
06 | GenerateRandomIndividuals;
07 | for Generation := 1 to Generations do
08 | begin
09 | | OrderIndividualsByScore;
10 | | Individual := 1;
11 | | while (Individual < (Individuals div 2)) do
12 | | begin
13 | | | DoCrossover(Individual, Individual+1,
14 | | |   Individuals-Individual,
15 | | |   Individuals-(Individual-1));
16 | | | Inc(Individual,2);
17 | | end;
18 | | for Individual := 1 to Individuals - 1 do
19 | | begin
20 | | | for I := Individual + 1 to Individuals do
21 | | | begin
22 | | | | IsEqual := True;
23 | | | | for Gene := 1 to VertexCount do
24 | | | | | if(Population[Individual, Gene] <>
25 | | | | | |  Population[I, Gene]) then
26 | | | | | |  begin IsEqual:= False; Break; end;
27 | | | | if IsEqual then DoMutate(I);
28 | | | end;
29 | | end;
30 | end;
31 end;
```

Fig. 19 Code of the ModifiedReproduce procedure

The difference is when using the genetic mutation operator. After the process of generating new solutions (lines 11-17) is completed, it is checked whether there are same solutions among them. When two identical solutions are found, the second one mutates (lines 18-29). This leads to an increase in the computational complexity of the whole procedure, which is already cubic and depends both on the

number of vertices in the graph and the number of individuals in the population, along with the number of reproductions generated.

The experiments showed that with a large number of individuals in the population (many times higher than the number of vertices in the graph), a large percentage (in some cases more than 50%) of the generated solutions are identical.

III. Results and Discussion

Two experiments will be conducted in this study. First, it will be checked experimentally for which graphs and with how many vertices (respectively edges), the recursive algorithm (using the backtracking and branch-and-bound methods), can be used to find a minimal Hamiltonian cycle for a reasonable time. Second, a comparative analysis between the two GA variants will be made, analyzing the quality of the found solutions, the time to find them, and the effect of the use of the genetic mutation operator.

A. The methodology of the experiments

Twelve complete and weighted graphs were created for the experiments, respectively with 15÷26 vertices. Each graph (except K_{15}) was created by adding a new vertex (n) and n-1 edges. These edges connect the new vertex with all other vertices. The coordinates of the vertices are shown in Table I. These are the screen coordinates of the centers of the vertices.

TABLE I
THE COORDINATES OF THE VERTICES OF THE K_{26} GRAPH

V	X	Y	V	X	Y	V	X	Y
1	178	499	**10**	261	509	**19**	159	156
2	99	515	**11**	307	419	**20**	83	139
3	258	227	**12**	320	171	**21**	48	251
4	34	184	**13**	117	433	**22**	248	28
5	110	245	**14**	49	437	**23**	22	326
6	127	300	**15**	232	435	**24**	85	367
7	126	84	**16**	301	275	**25**	60	35
8	234	317	**17**	235	137	**26**	35	93
9	286	78	**18**	207	75			

All three algorithms use an adjacency matrix – A[|V|,|V|]. Each item A[i,j] is greater than 0, for $\forall\ i \neq j$, and is equal to the length of the edge (i,j). These values are equal to the Euclidean distance between each pair of vertices. Since the tested graphs are undirected and have no loops, the elements in the matrix below and above the main diagonal are equal (i.e., the matrix A is symmetrical), and those on the main diagonal - A [i,i] have values equal to 0.

B. Experimental Conditions

The experimental conditions are the following: PC with 64–bit Operating System Windows 10, x64–based processor and hardware configuration: Processor: Intel (R) Core (TM) i7–4712MQ CPU at 2.30 GHz; RAM: 8GB DDR3.

C. Experimental results

The results of the recursion based algorithm for $K_{15} \div K_{26}$ graphs are shown in Table II.

TABLE II
THE RESULTS OF THE RECURSION BASED ALGORITHM

G	Length	Imp	Recursive Calls	Time (in ms)	Rcpms
K_{15}	1 450	21	407 554	47	8 671
K_{16}	1 569	24	1 593 906	188	8 478
K_{17}	1 612	27	7 980 375	953	8 374
K_{18}	1 696	35	27 655 276	3 390	8 158
K_{19}	1 746	37	125 799 721	16 257	7 738
K_{20}	1 755	43	680 150 761	90 625	7 505
K_{21}	1 777	47	2 317 052 334	327 547	7 074
K_{22}	1 780	37	7 051 026 768	1 087 625	6 483
K_{23}	1 853	40	51 144 113 151	10 154 156	5 037
K_{24}	1 871	43	180 806 431 928	42 860 625	4 218
K_{25}	1 898	41	814 421 034 545	258 414 656	3 152
K_{26}	2 002	58	6 583 890 702 285	2 312 113 594	2 848

The minimum Hamiltonian cycles that were generated by the recursive based algorithm for the K_{26} graph is shown in Fig. 20.

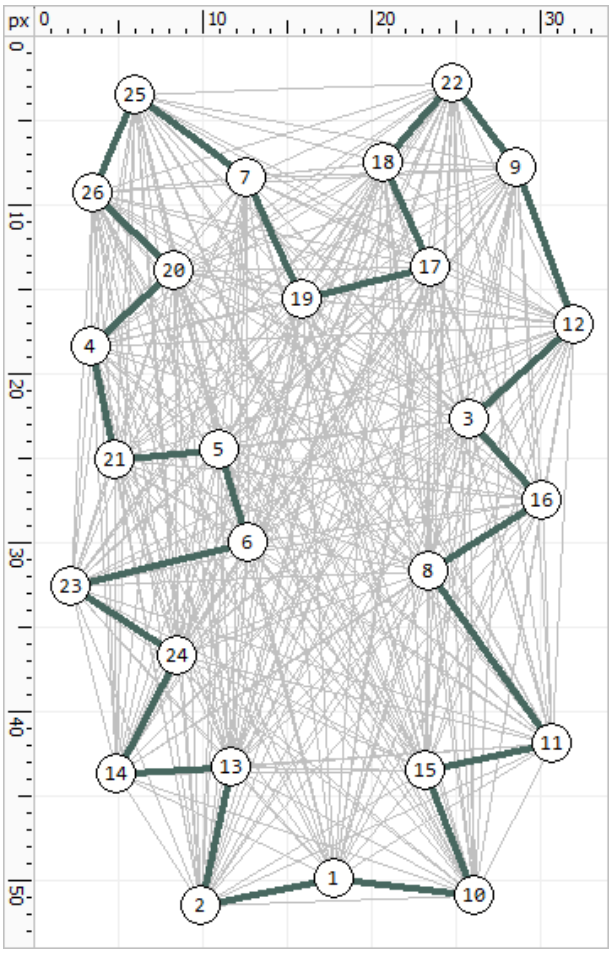

Fig. 20 K_{26} minimal Hamiltonian cycle

Table II shows that with the addition of a new vertex n (and n-1 edges), the time to find the minimal Hamiltonian cycle increases exponentially. In order to accurately measure this time, ten program runs for the $K_{15} \div K_{23}$ graphs, five runs for the K_{24} and K_{25} graphs, and one run for the K_{26} graph were made. For all graphs, the values in the Time (in ms)

column, except for the K_{26}, are calculated in arithmetic mean from all runs. The results show that when using the recursive algorithm to find the minimal Hamiltonian cycle for the K_{26} graph, the program ran for 26 days, 18 hours and 15 minutes.

Two other factors also need to be analyzed when interpreting the time to execute the recursive algorithm. These are the number of recursive calls made for 1 ms and the number of improvements found in the search process. These dependencies and their trends are shown in Fig. 21.

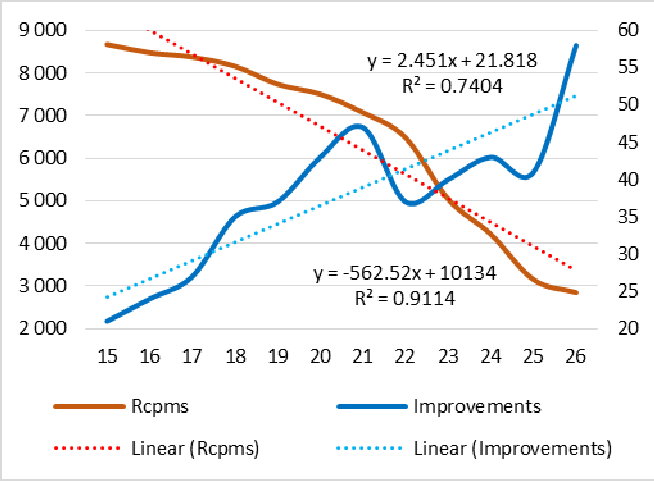

Fig. 21 Influence of the number of vertices (the x-axis) on the recursive calls per millisecond (the left y-axis) and on the improvements (the right y-axis) for recursive algorithms (for all tested graphs)

The number of recursive calls per millisecond (Rcpms) is calculated by dividing the values in the Recursive Calls column by those in the Time (in ms) column (Table II). The recursive algorithm accumulates the number of improvements during the minimal Hamiltonian cycle search process.

Fig. 21 shows that when increasing the execution time, the number of recursive calls per millisecond decreases. This is because the operating system dynamically changes the priority of the programs (and processes) that run for a longer time. In addition, when increasing the number of vertices in the graph, the improvements (when looking for better Hamiltonian cycles) are also increased.

In the second experiment, the two variants of GA will be analyzed by comparing the quality of the solutions found, the time for their generation, and the startup number where the best solution is found. The number of reproductions for all tested graphs will be set to 1 000. The population size will be calculated by multiplying the number of vertices in the graph by 10. If the resulting number is not exactly divisible to 4 (necessary to form the parent pairs), it will be equal to the closest integer that fulfills this requirement. For example, for graph K_{15}, this value will be 152.

The SGA results for all graphs K_{15}-K_{26} are presented in Table III. The abbreviations of the columns in Table III are as follows: "G" – the abbreviation of the graph; "P Size" – the population size; "Cycle Length" – the length of the minimal Hamiltonian cycle (in pixels); "B/S" – the best result of total runs; "Generated" – the number of all generated individuals (including mutated); "Mutated" – the number of mutated individuals only; "Time (ms)" – the execution time (in milliseconds) of the algorithm.

TABLE III
THE RESULTS OF THE STANDARD GENETIC ALGORITHM

G	P Size	Cycle Length	B/S	Individuals		Time (in ms)
				Generated	Mutated	
K_{15}	152	1 476	6/10	91 152	15 000	328
K_{16}	160	1 595	3/10	96 160	16 000	344
K_{17}	172	**1 612**	8/8	103 172	17 000	390
K_{18}	180	**1 696**	4/4	108 180	18 000	406
K_{19}	192	1 772	5/10	115 192	19 000	422
K_{20}	200	**1 755**	3/3	120 200	20 000	438
K_{21}	212	1 814	7/10	127 212	21 000	469
K_{22}	220	**1 780**	6/6	132 220	22 000	484
K_{23}	232	1 872	4/10	139 232	23 000	516
K_{24}	240	1 997	2/10	144 240	24 000	563
K_{25}	252	**1 898**	8/10	151 252	25 000	578
K_{26}	260	2 052	5/10	156 260	26 000	625

The number of all generated solutions (the "Generated" column) can be easily calculated. For example, for K_{26}, with a population size of 260 individuals, the new solutions that will be generated at each reproduction are exactly half of this value, i.e., 260 / 2 = 130. For 1 000 reproductions, the total number of generated solutions will be 1 000 x 130 = 130 000. When performing the mutation operator, 20% of the new solutions mutate, i.e., 130 x 0.2 = 26. In this way, the total number of mutated solutions for all reproductions is 26 x 1 000 = 26 000. These solutions are summed up with the other solutions, which are generated by the crossover operator, i.e., the total number of all solutions is 130 000 + 26 000 = 156 000. A further 260 solutions from the initial population must be added to them, with the final number of all generated solutions being 152 000 + 260 = 152 260.

The MGA results for all graphs K_{15}-K_{26} are presented in Table IV. The column names in Table IV are the same as those in Table III.

TABLE IV
THE RESULTS OF THE MODIFIED GENETIC ALGORITHM

G	P Size	Cycle Length	B/S	Individuals		Time (in ms)
				Generated	Mutated	
K_{15}	152	**1 450**	2/2	132 485	56 333	688
K_{16}	160	**1 569**	1/1	133 964	53 804	766
K_{17}	172	**1 612**	2/2	146 534	60 362	875
K_{18}	180	**1 696**	1/1	154 116	63 936	968
K_{19}	192	**1 746**	1/1	157 427	61 235	1 046
K_{20}	200	**1 755**	1/1	169 285	69 085	1 250
K_{21}	212	**1 777**	1/1	164 820	58 608	1 156
K_{22}	220	**1 780**	1/1	158 503	48 283	1 266
K_{23}	232	**1 853**	2/2	191 205	74 973	1 547
K_{24}	240	**1 871**	4/4	194 526	74 286	1 734
K_{25}	252	**1 898**	5/5	185 679	59 427	1 578
K_{26}	260	**2 002**	2/2	200 880	70 620	1 969

The number of all generated solutions (the "Generated" column) can be calculated in a similar way to SGA. For example, for graph K_{26} with a population size of 260

individuals, the new solutions that are generated by the genetic crossover operator will also be 130 000. However, when performing the genetic mutation operator, the number of mutated solutions depends on the number of the identical ones among them. For graph K_{26}, the total number of mutated solutions is 70 620. These solutions are summed up with the other solutions that are generated by the crossover operator, so the total number of all solutions is 130 000 + 70 620 = 200 620. A further 260 solutions from the initial population must be added to them, with the final number of all generated solutions being 200 620 + 260 = 200 880.

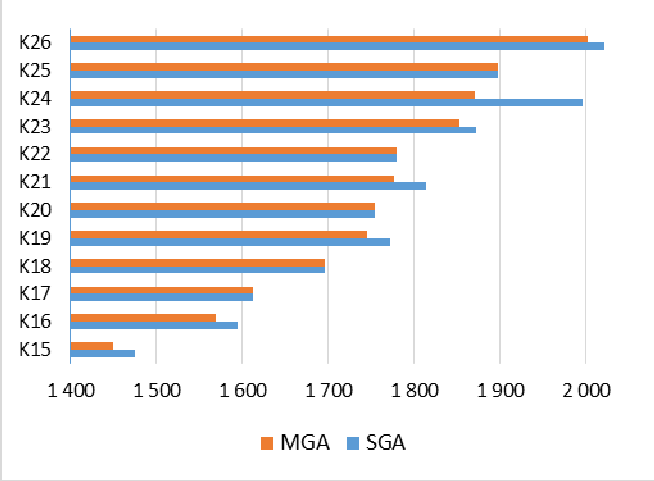

Fig. 22 Results of the SGA and MGA

Tables III and IV, and Fig. 22 indicate that the MGA has found optimal solutions for all tested columns. In 6 out of 10 cases, this happened with the first run. Unlike the MGA, SGA has found optimal solutions only in 5 out of 12 cases. The solutions in the other 7 cases are not optimal, but they are close to them. Typical of SGAs is that more runs were needed to find the best solutions (including the optimal ones). This means that a larger total number of solutions has been generated from all runs compared to the MGA. Another important result is that in the MGA, the average percentage of mutated solutions (in all 12 cases) is 62%, while at the SGA this percentage was fixed at 20%. However, the MGA execution time is 2.61 times higher than the SGA. This is understandable because the computational complexity of the MGA is greater than that of the SGA. This influences the time to generate each solution and accordingly the time to generate all solutions. Despite this time difference (of about 2 seconds), the MGA gives better results than the SGA in all cases.

IV. Conclusion

This paper has shown an analysis of three algorithms for the TSP. The main steps of genetic algorithms and their benefits in solving combinatorial optimization problems were presented. Furthermore, a number of studies analyzing the TSP and some approaches for its solution were discussed as well. An optimized version of the standard recursive algorithm for TSP which uses the backtracking method was introduced. This algorithm does not generate all solutions, but only those that are closest to the optimal. Also, a standard genetic algorithm for TSP and one of its modification were presented. Modification itself is the use of the genetic mutation operator. The results showed that the recursive algorithm can be used to solve the TSP for graphs with a small number of peaks 20-25. The results of GAs have shown that the MGA finds optimal solutions in all cases, and SGA in only 40% of them. Both algorithms are executed for a reasonable time (up to 2 seconds). Also, it was found that GAs finds optimal solutions only if appropriate values of their parameters – population size and number of reproductions are set. In addition, the genetic mutation operator performs better if it is used to change the identical solutions in the population instead of changing a predefined number of solutions. However, the more the number of vertices in the graph, the more difficult it becomes for the MGA to find the best solution, i.e., more runs (and therefore, more generated solutions) are needed to find a better solution.

Future guidelines for research include conducting additional experiments to study the impact of population size on the quality of GA solutions. This parameter must be set to match the optimal relationship between the number of vertices in the graph and the population size. Besides this parameter, there is another one that needs to be analyzed - this is the number of reproductions. This parameter must be set so that the population can reach a convergence state. This is the state where the population contains identical solutions, and further improvement cannot be achieved.

Acknowledgments

I would like to express my sincere gratitude to Dr. Radoslava Kraleva and Dr. Dafina Kostadinova from the South-West University in Bulgaria, for their suggestions and constructive criticism regarding this paper.

References

[1] Y. Wang, “A genetic algorithm with the mixed heuristics for traveling salesman problem,” *International Journal of Computational Intelligence and Applications*, vol. 14(1), pp. 33–46, Mar. 2015.

[2] W. R. Alkhayri, S. S. Owais, and M. Shkoukani, "A New Selection Operator - CSM in Genetic Algorithms for Solving the TSP," *International Journal of Advanced Computer Science and Applications*, vol. 7(10), pp. 62-66, Oct. 2016.

[3] M. Yamada, "1/f Noise in the Simple Genetic Algorithm Applied to a Traveling Salesman Problem," *Fluctuation and Noise Letters*, vol. 16(3), 1750026, Sep. 2017.

[4] S. Meneses, R. Cueva, M. Tupia, Manuel, and M. Guanira, "A genetic algorithm to solve 3D traveling salesman problem with initial population based on a GRASP algorithm," *Journal of Computational Methods in Sciences and Engineering*, vol. 17(S1), pp. S1-S10, Jan. 2017.

[5] J. Wang, O. K. Ersoy, M. He, and F. Wang, "Multi-offspring genetic algorithm and its application to the traveling salesman problem," *Applied Soft Computing*, vol. 43, pp. 415-423, Jun. 2016.

[6] S. Maity, A. Roy, and M. Maiti, "A Modified Genetic Algorithm for solving uncertain Constrained Solid Travelling Salesman Problems," *Computers & Industrial Engineering*, vol. 83, pp. 273-296, May 2015.

[7] P. V. Paul, N. Moganarangan, S. Sampath Kumar, R. Raju, T. Vengattaraman, and P. Dhavachelvan, "Performance analyses over population seeding techniques of the permutation-coded genetic algorithm: An empirical study based on traveling salesman problems," *Applied Soft Computing*, vol. 32, pp. 383-402, Jul. 2015.

[8] V. Kralev and R. Kraleva, "A Local Search Algorithm Based on Chromatic Classes for University Course Timetabling Problem,"*International Journal of Advanced Research in Computer Science*, vol. 7(28), pp. 1-7, Jan. 2017.

[9] M. Traykov, S. Angelov, and N. Yanev, "A New Heuristic Algorithm for Protein Folding in the HP Model," *Journal of Computational Biology*, vol. 23(8), pp. 662-668, Aug. 2016.

[10] V. Kralev, R. Kraleva, and B. Yurukov, "An event grouping based algorithm for university course timetabling problem," *International Journal of Computer Science and Information Security*, vol. 14(6), pp. 222-229, Jun. 2016.

[11] V. Kralev, "A genetic and memetic algorithm for solving the university course timetabling problem," *International Journal "Information Theories & Applications*, vol. 16(3), pp. 291-299, 2009.

[12] V. Vladimirov, F. Sapundzhi, R. Kraleva, and V. Kralev, "Modified Genetic Algorithm to Traveling Salesman Problem for Large Input Datasets," *Biomath Communications*, vol. 3(1), p. 71, Jun. 2016.

[13] A. Chowdhury, A. Ghosh, S. Sinha, S. Das, and Av. Ghosh, "A novel genetic algorithm to solve travelling salesman problem and blocking flow shop scheduling problem," *International Journal of Bio-Inspired Computation*, vol. 5(5), pp. 303-314, Oct. 2013.

[14] V. E. Alekseev, R. Boliac, D. V. Korobitsyn, and V. V. Lozin, "NP-hard graph problems and boundary classes of graphs," *Theoretical Computer Science*, vol. 389(1), pp. 219–236, Dec. 2007.

[15] N. A. M. Zin, S. N. H. S. Abdullah, N. F. A. Zainal, and E. Ismail, "A Comparison of Exhaustive, Heuristic and Genetic Algorithm for Travelling Salesman Problem in PROLOG," *International Journal on Advanced Science, Engineering and Information Technology*, vol. 2(6), pp. 49-53, Dec. 2012.

[16] S. Yuan, B. Skinner; S. Huang, and D. Liu, "A new crossover approach for solving the multiple travelling salesmen problem using genetic algorithms," *European Journal of Operational Research*, vol. 228(1), pp. 72-82, Jul. 2013.

[17] C. W. Tsai, S. P. Tseng, M. C. Chiang, C. S. Yang, and T. P. Hong, "A High-Performance Genetic Algorithm: Using Traveling Salesman Problem as a Case," *The Scientific World Journal*, vol. 2014, 178621, May 2014.

[18] Y. Nagata and D. Soler, "A new genetic algorithm for the asymmetric traveling salesman problem," *Expert Systems with Applications*, vol. 39(10), pp. 8947–8953, Aug. 2012.

[19] R. J. Wilson, *Introduction to Graph Theory*, 5th ed., New Jersey, USA: Prentice Hall, 2010.

[20] M. Alameen, M. Abdul-Niby, A. Salhieh, and A. Radhi, "Improved Genetic and Simulating Annealing Algorithms to Solve the Traveling Salesman Problem Using Constraint Programming," *Engineering Technology & Applied Science Research*, vol. 6(2), pp.927-930, Apr. 2016.

[21] A. F. El-Samak and W. Ashour, "Optimization of Traveling Salesman Problem Using Affinity Propagation Clustering and Genetic Algorithm," *Journal of Artificial Intelligence and Soft Computing Research*, vol. 5(4), pp. 239-245, Oct. 2015.

[22] C. Groba, A. Sartal, and X. H. Vazquez, "Solving the dynamic traveling salesman problem using a genetic algorithm with trajectory prediction: An application to fish aggregating devices," *Computers & Operations Research*, vol. 56, pp. 22-32, Apr. 2015.

[23] Y. Wang, "A Genetic Algorithm with the Mixed Heuristics for Traveling Salesman Problem," *International Journal of Computational Intelligence and Applications*, vol. 14(1), 1550003, Mar. 2015.

[24] M. K. Rafsanjani, S. Eskandari, and A. B. Saeid, "A similarity-based mechanism to control genetic algorithm and local search hybridization to solve traveling salesman problem," *Neural Computing & Applications*, vol. 26(1), pp. 213-222, Jan. 2015.

[25] V. Kralev,"An Analysis of a Recursive and an Iterative Algorithm for Generating Permutations Modified for Travelling Salesman Problem," *International Journal on Advanced Science, Engineering and Information Technology*, vol. 7(5), pp. 1685-1692, Oct. 2017.

[26] C. Changdar, G. S. Mahapatra, and R. K. Pal, "An efficient genetic algorithm for multi-objective solid travelling salesman problem under fuzziness," *Swarm and Evolutionary Computation*, vol. 15, pp. 27-37, Apr. 2014.

[27] Y. Nagata and S. Kobayashi, "A Powerful Genetic Algorithm Using Edge Assembly Crossover for the Traveling Salesman Problem," *Informs Journal on Computing*, vol. 25(2), pp. 346-363, Sep. 2013.

[28] A. B. A. Hassanat, E. Alkafaween, N. A. Al-Nawaiseh, M. A. Abbadi, M. Alkasassbeh, and M. B. Alhasanat, "Enhancing Genetic Algorithms using Multi Mutations: Experimental Results on the Travelling Salesman Problem," *International Journal of Computer Science and Information Security*, vol. 14(7), pp. 785-801, Jul. 2016.

[29] M. Albayrak and N. Allahverdi, "Development a new mutation operator to solve the Traveling Salesman Problem by aid of Genetic Algorithms," *Expert Systems with Applications*, vol. 38(3), pp. 1313-1320, Mar. 2011.

[30] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook, *The Traveling Salesman Problem: A Computational Study*, 2nd ed., Princeton, USA: Princeton University Press, 2007.

[31] G. Gutin and A.P. Punnen, The Traveling Salesman Problem and Its Variations (Combinatorial Optimization), 2nd ed., New York City, USA: Springer, 2007.

[32] A. Khanra, M. K Maiti, and M. Maiti, "A hybrid heuristic algorithm for single and multi-objective imprecise traveling salesman problems," *Journal of Intelligent and Fuzzy Systems*, vol. 30(4), pp. 1987–2001, Mar. 2016.

[33] M. Mestria, "A hybrid heuristic algorithm for the clustered traveling salesman problem," *Pesquisa Operacional*, vol. 36(1), pp. 113–132, Jan-Apr. 2016.

[34] Z. A. Othman, N. H. Al-Dhwai, A. Srour, and W. Diyi, "Water Flow-Like Algorithm with Simulated Annealing for Travelling Salesman Problems," *International Journal on Advanced Science, Engineering and Information Technology*, vol. 7(2), pp. 669-675, Apr. 2017.

[35] M. Battarra, A. A. Pessoa, A. Subramanian, and E. Uchoa, "Exact algorithms for the traveling salesman problem with draft limits," *European Journal of Operational Research*, vol. 235(1), pp. 115–128, May. 2014.

[36] J. Kinable, B. Smeulders, E. Delcour, F. C. R. Spieksma, "Exact algorithms for the Equitable Traveling Salesman Problem," *European Journal of Operational Research*, vol. 261(2), pp. 1339–1351, Sep. 2017.

[37] Y. Deng, Y. Liu, and D. Zhou, "An Improved Genetic Algorithm with Initial Population Strategy for Symmetric TSP," *Mathematical Problems in Engineering*, vol. 2015, 212794, Oct. 2015.

[38] A. Hussain, Y. S. Muhammad, M. N. Sajid, I. Hussain, A. M. Shoukry, and S. Gani, "Genetic Algorithm for Traveling Salesman Problem with Modified Cycle Crossover Operator," *Computational Intelligence and Neuroscience*, vol. 2017, 7430125, Oct. 2017.

[39] C. Contreras-Bolton and V. Parada, "Automatic Combination of Operators in a Genetic Algorithm to Solve the Traveling Salesman Problem," *PLoS ONE*, vol. 10(9), 0137724, Sep. 2015.