An Analysis of a Recursive and an Iterative Algorithm for Generating Permutations Modified for 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 a comparative analysis between a recursive and an iterative algorithm when generating permutation. A number of studies discussing the problem and some methods dealing with its solution are analyzed. Recursion and iteration are approaches used in computer programs to implement different algorithms. An iterative approach is the repeated execution of the same source code until a certain end condition is met. On the other hand, a recursive approach uses a recursive function that repeatedly calls itself. This function contains a source code that must be executed repeatedly. Both algorithms presented in this paper can be used to generate permutations of an n element set. The algorithms are modified so that they can be used to solve the Travelling Salesman Problem (TSP) with a small number of vertices. Several publications that discuss the TSP and some approaches to its solution are also presented. The methodology and the conditions for conducting the experiments are described in details. The obtained results have been analyzed; they show that for the same conditions the iterative algorithm works from of 2^3 to 2^4 times faster than the recursive algorithm in all the tested input data. Several approaches to optimize the two algorithms in terms of the number of permutations tested when searching a minimal Hamiltonian cycle are presented.**

***Keywords*— iterative algorithm; recursive algorithm; travelling salesman problem; computer programming.**

I. INTRODUCTION

The most important components of a computer program are the algorithms used. This defines the amount of memory used (necessary for storing data structures) as well as the programme's performance (i.e., its effectiveness). Therefore, when developing computer algorithms, it is important to use such methods whereby the memory used is as small as possible and the time to perform computational processes is as short as possible. Research into the development of computer algorithms began many years ago. Exhaustive and systematized research in this field are presented in [1] and [2]. When creating an algorithm to solve a practical problem, first, it is necessary to select a suitable data structure. These data will be stored and processed during the computation process. The approach that will be chosen for modeling and solving the problem will determine how the computation process runs, e.g., recursive or iterative.

A study reveals the preferences of beginner programmers when developing computer algorithms in terms of whether to be recursive or iterative [3]. The results show that the preferences in recent years have been in favor of recursive algorithms. These algorithms are more frequently preferred than the iterative ones. Another study presented in [4] shows that when selecting a relevant approach, the specificity of the problem should be taken into account. In [5] a method based on incrementalization for transforming the recursion in iteration is proposed. The results show that in most experiments the iterative algorithms used more memory. On the other hand, these algorithms were performed faster. A method for recursion removal is proposed in [6]. Different methods for optimizing memory usage when using recursion are offered in [7]. In addition, similar situations in which nested cycles are used have been taken into consideration. An approach to transforming iterative cycles in recursive methods is presented in [8]. In earlier sources, similar approaches for creating a recursive algorithm based on an iterative process are also presented [9]. The idea of automated transforming of recursive methods in iterative loops may also be a subject of a study.

Iterative and recursive implementations of various algorithms are widely used in many fields of science and practice, such as those presented in [10]–[12]. An intensively investigated field of study is the graph theory. This is a part of discrete mathematics, which has undergone great development over the last few decades and has a huge practical application. In many cases, the description, analysis, and research of real systems is accomplished successfully and comparatively simply with graphs. A graph contains two sets of elements - vertices and edges. Each edge connects a pair of vertices. The ordered pair (V, E) is called an

undirected graph (G), where: $V = \{v_1, v_2, ..., v_n\}$ is a set of vertices and $E = \{e_1, e_2, ..., e_m\}$ is a set of edges. The V and E sets are finite. Each element $k \in E$, (k = 1, 2, .., m) is an ordered pair (v_i, v_j), $v_i, v_j \in V$, $1 \leq i, j \leq n$. When the pair of vertices (v_i, v_j) is ordered, the graph is called directed, and the edges are called arcs. If a function f(i, j) is given, comparing an integer value to each edge $(i, j) \in E$, f(i, j) = f(j, i), the graph is called undirected weighted. If a sequence of vertices $v_1, v_2, ..., v_k$ is given, such that for each i = 1, 2, ..., k-1 is satisfied $(v_i, v_{i+1}) \in E$, then this sequence is called a path in a directed graph. In the case of an undirected graph, the ordinance of elements in the pair (v_i, v_{i+1}) does not matter. The path where the start and final vertex coincide is called a cycle. A cycle that passes through each edge exactly once is called Eulerian, while a cycle that passes through each vertex exactly once is called Hamiltonian. A graph is called complete when for each pair of vertices there is an edge that connects them, i.e., an edge that is incidentally with them. In [13] more information related to the graphs theory is given: the main ways of presenting these structures in computer programs and a wide variety of algorithms to solve particular problems.

The graphs provide opportunities to formulate and solve complex practical problems effectively in a natural way and in an accessible language. Numerous problems in different fields, both in science and in practice (for example, transport problems, resource allocation problems, search for optimal routes and the location of service centers, problems related to optimal schedules and timetables and many others), can be modeled with graphs and solved with appropriate algorithms [13]. In many cases these problems are linear, which means that they are solved with linear optimization methods. For some problems with large input data, it is necessary to look for efficient algorithms (usually approximated) so that they can be solved for a reasonable time. Solving these problems without a computer and a suitable application would be very difficult except for some cases in which the input data are very small. But even with a computer and appropriate application software, solving some problems would be impossible, or theoretically, it would take a long time, for example years, to find an exact solution [14]. Such are all NP-hard problems in which a solution cannot be found in polynomial time. The methods based on the "backtracking" approach yield good ressults but only for problems with small input data. For instance, in a complete undirected graph with 25 vertices and 25 * (25-1) / 2 = 300 edges, the number of all Hamiltonian cycles (if necessary to be checked) is very large, respectively: (25-1)! / 2 = 310 224 200 866 620 000 000 000. With the capabilities of modern computers, this approach, though possible, is practically inapplicable. Therefore, the interest in developing new or modifying the existing heuristic or approximate algorithms (not only for graphs) is explicable. This helps to find a solution to some "hard" problems for a reasonable time.

The number of practical problems that can be modeled by graphs is big. A problem to find a minimal Hamiltonian cycle in a complete undirected weighted graph will be selected for the present study. This is a combinatorial optimization problem that has been researched very actively in recent years. A detailed description of the Travelling Salesman Problem (TSP) is presented in [15]. Moreover, in [16] different variants of it are presented. Due to the great practical application of this problem, many algorithms have been created for it. There are two main approaches on the basis of which all algorithms for solving the TSP are developed. The first approach is based on the creation of exact algorithms. These algorithms always find the exact solution, but a lot of computing time is required for their execution. It is recommended that these algorithms be used in small input data (in this case, a small number of vertices in a graph, for instance, up to 20). Examples of such algorithms are presented in [17]–[19]. The second approach is based on the creation of approximate or heuristic algorithms. These algorithms find solutions that are close to optimal or optimal, but this cannot be proven. Most of the developed algorithms for solving TSP are in this category. Examples of such algorithms are presented in [20]–[24].

II. Material And Method

This paper presents the results from an experimental study of two algorithms for generating permutations, respectively recursive and iterative [2]. Both algorithms have been modified to be used to search for an exact TSP solution. The analysis of the results aims at determining the way the algorithmic implementation influences on the computation process. For this purpose, both algorithms need to generate the same solutions. The main metrics that will be analyzed are the quality of the solutions found (minimum length of the Hamiltonian cycle) and the execution time (the time required to check all possible Hamiltonian cycles).

Let a set $A = \{a_1, a_2, ..., a_n\}$ with n elements be given. Each sequence (or order) of these elements (without repetition) is called permutation. The set of all possible permutations is denoted with P_n and its power is marked with $|P_n|$, which is equal to n!. The following recursive relationship can be used to generate permutations: after placing a particular element in position k, the sequential generation of all possible permutations of the remaining n-k elements follows. After generating each permutation, the length of a Hamiltonian cycle formed by the sequence of the vertices defined in the current permutation will be calculated.

When executing the algorithms, some global arrays and variables will be used. They need to be pre-declared, as shown in Fig. 1 (in Delphi language).

```
01 | AdjacencyMatrix: array of array of Integer;
02 | MarkedVertices: array of Boolean;
03 | HamiltonianCycle: array of Integer;
04 | MinimumHamiltonianCycle: array of Integer;
05 | CycleLength: Integer;
06 | MinimumCycleLength: Integer;
07 | Counter: Int64;
08 | CounterCycles: Int64;
09 | CounterBetterCycles: Int64;
10 | BestCycleIndex: Int64;
11 | StartTickCount: Cardinal;
12 | FinishTickCount: Cardinal;
```

Fig. 1 Source code of the global declarations

A recursive algorithm used to search for a minimal Hamiltonian cycle is presented in Fig. 2. This algorithm is based on the generation of permutations and uses the "backtracking" method.

```
01 procedure HamiltonianCycleByRecursion
02   (Iteration, Position, VertexCount: Integer);
03 var
04   J: Integer;
05 begin
06   Inc(Counter);
07   if ((Iteration = 1) and (Position > 1)) then
08   begin
09     if (Position = VertexCount+1) then
10     begin
11       Inc(CounterCycles);
12       CycleLength := 0;
13       for J := 0 to VertexCount-1 do
14         CycleLength := CycleLength +
15         AdjacencyMatrix[HamiltonianCycle[J]]
16                        [HamiltonianCycle[J+1]];
17       if (MinimumCycleLength > CycleLength) then
18       begin
19         BestCycleIndex := CounterCycles;
20         Inc(CounterBetterCycles);
21         MinimumCycleLength := CycleLength;
22         for J := 1 to VertexCount do
23           MinimumHamiltonianCycle[J] :=
24             HamiltonianCycle[J];
25       end;
26     end;
27     Exit;
28   end;
29   if (MarkedVertices[Iteration]=True) then Exit;
30   MarkedVertices[Iteration] := True;
31   for J := 1 to VertexCount do
32   begin
33     if ((AdjacencyMatrix[Iteration][J] > 0) and
34       (J <> Iteration)) then
35     begin
36       HamiltonianCycle[Position] := J;
37       HamiltonianCycleByRecursion
38         (J, Position+1, VertexCount);
39     end;
40   end;
41   MarkedVertices[Iteration] := False;
42 end;
```

Fig. 2 Source code of the recursion based algorithm

This algorithm uses a modification of the Depth-first search (DFS) method. The HamiltonianCycleByRecursion procedure is called recursively (lines 37 and 38) until all Hamiltonian paths starting from a first vertex and passing through all other vertices are generated. Finally, the edge that connects the end with the first vertex is added to this path to form a Hamiltonian cycle. This is done when the condition of line 9 is true. The length of the new Hamiltonian cycle is calculated on lines 13–16. Then, whether this cycle is shorter than the shortest one that has been found so far (line 17) is checked. If this proves true, the current cycle is stored as the shortest one (lines 19–24). The complexity of the algorithm is exponential because it generates all (VertexCount-1)! Hamiltonian cycles (including those in the opposite direction).

Another algorithm used to search for a minimal Hamiltonian cycle is presented in Fig. 3. This algorithm, however, is based on the generation of permutations using an iterative approach.

This algorithm generates iteratively all permutations of (VertexCount-1) elements – formed by the vertex indexes, respectively, 2, 3, ..., VertexCount. After generating the next Hamiltonian path (composed of vertices 1, 2, ..., VertexCount), the edge (VertexCount, 1) is added to it. Since the explored graphs are complete, such an edge certainly exists.

```
01 procedure HamiltonianCycleByIteration
02   (VertexCount: Integer);
03 var
04   LeftPos, RightPos, Temp: Integer;
05   Terminated, NewPermutation: Boolean;
06 begin
07   for Temp := 0 to VertexCount-1 do
08     HamiltonianCycle[Temp] := Temp + 1;
09   HamiltonianCycle[VertexCount] := 1;
10   CounterCycles := 1;
11   MinimumCycleLength := 0;
12   for Temp := 0 to VertexCount-1 do
13     MinimumCycleLength := MinimumCycleLength +
14     AdjacencyMatrix[HamiltonianCycle[Temp]]
15                    [HamiltonianCycle[Temp+1]];
16   BestCycleIndex := 1;
17   CounterBetterCycles := 1;
18   for Temp := 0 to VertexCount do
19     MinimumHamiltonianCycle[Temp] :=
20       HamiltonianCycle[Temp];
21   Terminated := False;
22   repeat
23     Inc(Counter);
24     NewPermutation := False;
25     LeftPos := VertexCount;
26     while (LeftPos > 0) do
27     begin
28       Inc(Counter); Dec(LeftPos);
29       RightPos := VertexCount;
30       while (RightPos > LeftPos) do
31       begin
32         Inc(Counter); Dec(RightPos);
33         if (HamiltonianCycle[LeftPos] <
34           HamiltonianCycle[RightPos]) then
35         begin
36           Temp := HamiltonianCycle[LeftPos];
37           HamiltonianCycle[LeftPos] :=
38             HamiltonianCycle[RightPos];
39           HamiltonianCycle[RightPos] := Temp;
40           Inc(LeftPos);
41           RightPos := VertexCount - 1;
42           while (LeftPos < RightPos) do
43           begin
44             Inc(Counter);
45             Temp := HamiltonianCycle[LeftPos];
46             HamiltonianCycle[LeftPos] :=
47               HamiltonianCycle[RightPos];
48             HamiltonianCycle[RightPos] := Temp;
49             Inc(LeftPos);
50             Dec(RightPos);
51           end;
52           Inc(CounterCycles);
53           CycleLength := 0;
54           for Temp := 0 to VertexCount-1 do
55             CycleLength := CycleLength +
56               AdjacencyMatrix
57                 [HamiltonianCycle[Temp]]
58                 [HamiltonianCycle[Temp+1]];
59           if(MinimumCycleLength>CycleLength)then
60           begin
61             BestCycleIndex := CounterCycles;
62             Inc(CounterBetterCycles);
63             MinimumCycleLength := CycleLength;
64             for Temp := 1 to VertexCount do
65               MinimumHamiltonianCycle[Temp] :=
66                 HamiltonianCycle[Temp];
67           end;
68           NewPermutation := True; Break;
69         end;
70       end;
71       if ((LeftPos=1) and (RightPos=1)) then
72       begin Terminated := True; Break; end;
73       if NewPermutation then Break;
74     end;
75   until (Terminated);
76 end;
```

Fig. 3 Source code of the iterative based algorithm

In lines 7–9, an initialization Hamiltonian cycle is generated. This cycle contains the following sequence of vertices: 1, 2, ..., VertexCount, 1. In lines 12–15, the length of the generated Hamiltonian cycle is calculated using the adjacency matrix. The lengths of all edges are stored in this matrix. At this point only this cycle is generated, this is why it is stored as a minimum (lines 18–20).

The iterative generation of permutations is performed by three nested loops (*while — do*). These loops begin at lines 26, 30 and 42, respectively. From the right to the left, each subsequent number of the current series is permutated sequentially. When this is done (line 51), a new permutation is already generated. Based on this permutation, the next Hamiltonian cycle is formed and its length is calculated (lines 54–58). If the length of the last generated Hamiltonian cycle is shorter than the smallest length found so far, the length is stored as the smallest one (line 63) and the corresponding cycle is stored as the shortest one (lines 64–66). The generation process is repeated until the left and right positions of the indexes of the elements become equal to 1. Since the next permutations, starting with the vertex numbers 2, 3, ..., VertexCount, respectively, are already checked for previous generations, they are not considered. These are identical cycles where the starting vertex is different from 1.

This iterative algorithm initially generates a Hamiltonian cycle, respectively: 1, 2, ..., VertexCount, 1, and calculates its length. The first order of the vertices is used as the initialization permutation from which all the next ones are generated. In the recursive algorithm, this initial permutation occurs when the recursive procedure is called (VertexCount+1) times to form this first cycle. Therefore, chronometers that report the execution time of the two algorithms are started before the initialization process. A variant of the procedure for starting the two algorithms is shown in Fig. 4.

```
01 procedure Run;
02 begin
03   SetLength(MarkedVertices, VCount+1);
04   SetLength(MinimumHamiltonianCycle, VCount+1);
05   SetLength(HamiltonianCycle, VCount+1);
06   MinimumCycleLength := MaxInt;
07   CycleLength := 0;
08   HamiltonianCycle[0] := 1;
09   MinimumHamiltonianCycle[0] := 1;
10   CounterCycles := 0;
11   CounterBetterCycles := 0;
12   BestCycleIndex := 0;
13   Counter := 0;
14   StartTickCount := GetTickCount();
15   HamiltonianCycleByRecursion(1,1,VCount);
16   // HamiltonianCycleByIteration(VCount);
17   FinishTickCount := GetTickCount();
18   ShowHamiltonianCycleInformation;
19 end;
```

Fig. 4 Source code of the run method

Both algorithms can be run by calling the Run procedure. Memory allocation for the dynamic arrays where the required information will be stored is performed on lines 3–5. The initialization values of the global variables are set on lines 6–12. Since a similar initialization is performed before the iteration algorithm is executed, the source code of these lines may not be executed. Before the iterative algorithm can be started, lines 6–12 and 15 may be commenting out and line 16 to be uncommenting. The runtime of both algorithms is counted using the GetTickCount function. The result after performing this function is the elapsed time (in milliseconds) from the start of the operating system (at the current work session). The first call to the function is before the start of both algorithms, and the second call is after the completion of their execution. To count the elapsed time, the first value is subtracted from the second value. Typically, the operating systems use a multi-tasking mode. This means that more work is simulated at the same time. During the operation of the operating system, different processes are executed. In order to accurately measure the runtime of both algorithms, they are run 10 times. Then an average runtime is calculated.

The presentation of algorithms with pseudo-codes or block diagrams does not guarantee that their implementation will be correct. In order to verify the results obtained, it is necessary to implement the presented algorithms correctly. Therefore, the complete source codes of both algorithms are presented in this paper. In this way, it is easier (and safer) for both algorithms to be implemented in another programming language. Also, by compiling into a suitable development environment, these algorithms can be executed immediately.

III. Results and Discussion

The aim of the experiments is to determine the behavior of both algorithms with the same input data. For this reason it is necessary to make a comparative analysis between these algorithms in order to determine for what graphs (with how many vertices and edges) they will be able to generate optimal solutions (in terms of the length of the Hamiltonian cycles) but for a reasonable time.

A. Methodology of the experiment

Six complete and weighted graphs were created for the experiments, respectively with 10÷15 vertices. Each graph (except K_{10}) was created by adding a new vertex (n) and n-1 edges. These edges connect the new vertex with all other vertices. K_{15} graph is presented in Fig. 5.

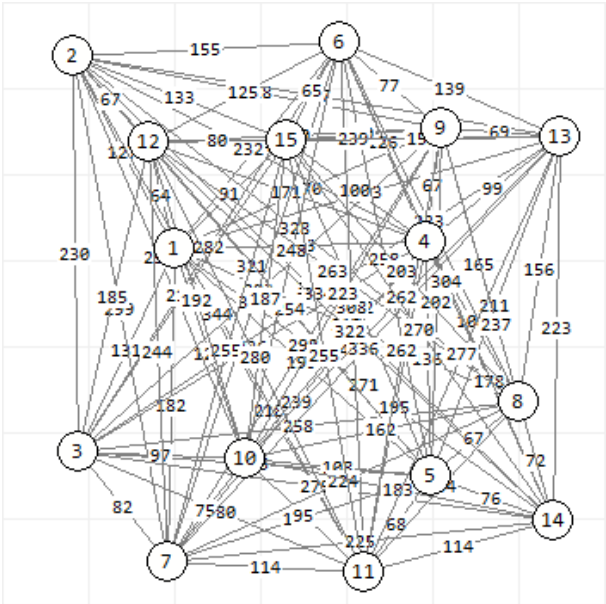

Fig. 5 K_{15} graph (with 15 vertices and 105 edges)

The coordinates of the vertices are shown in Table I. These are the screen coordinates of the centers of the vertices. The abbreviations of the columns in Table I are as follows: V – vertex number; X – the x coordinate of the vertex, and Y – the y coordinate of the vertex.

TABLE I
THE COORDINATES OF THE VERTICES OF THE K_{15} GRAPH

V	X	Y	V	X	Y	V	X	Y
1	100	143	**6**	196	23	**11**	210	331
2	41	31	**7**	96	325	**12**	85	81
3	44	261	**8**	300	232	**13**	324	78
4	246	139	**9**	255	73	**14**	320	301
5	249	275	**10**	141	265	**15**	165	80

Both algorithms use an adjacency matrix – A[VertexCount] [VertexCount]. When there is an edge (u, v) between two vertices, for example u and v, then A[u][v] > 0, or, otherwise, A[u][v] = 0. Each element A[u][v] > 0 is equal to the length of the edge (u, v). The adjacency matrix of the K_{15} graph is shown in Table II.

TABLE II
THE ADJACENCY MATRIX OF K_{15} GRAPH

V\V	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	127	131	146	199	154	182	219	170	129	218	64	233	271	91
2	127	0	230	232	321	155	299	328	218	254	344	67	287	388	133
3	131	230	0	236	205	282	82	258	283	97	180	185	334	279	218
4	146	232	236	0	136	126	239	108	67	164	195	171	99	178	100
5	199	321	205	136	0	258	161	67	202	108	68	254	211	76	212
6	154	155	282	126	258	0	318	233	77	248	308	125	139	304	65
7	182	299	82	239	161	318	0	224	298	75	114	244	336	225	255
8	219	328	258	108	67	233	224	0	165	162	134	263	156	72	203
9	170	218	283	67	202	77	298	165	0	223	262	170	69	237	90
10	129	254	97	164	108	248	75	162	223	0	95	192	262	183	187
11	218	344	180	195	68	308	114	134	262	95	0	280	277	114	255
12	64	67	185	171	254	125	244	263	170	192	280	0	239	322	80
13	233	287	334	99	211	139	336	156	69	262	277	239	0	223	159
14	271	388	279	178	76	304	225	72	237	183	114	322	223	0	270
15	91	133	218	100	212	65	255	203	90	187	255	80	159	270	0

All elements in the matrix (except those in the main diagonal) have a value other than 0 because the graphs under consideration are complete. Also, the adjacency matrix is symmetrical (relative to its main diagonal) in these graphs. The element values are calculated from the coordinates of each pair of vertices and are equal to the Euclidean distance between these vertices.

B. Experimental Conditions

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

C. Experimental results

In Table III, the main properties of the studied graphs are shown. These properties are as follows: the graph abbreviation – G, the number of the vertices – |V|, the number of the edges – |E|, the number of the Hamiltonian cycles – (|V|-1)!/2, and the number of the cycles that are to be verified – (|V|-1)!.

TABLE III
THE MAIN PROPERTIES OF THE GRAPHS

G	|V|	|E|	(|V|-1)!/2	(|V|-1)!
K_{10}	10	45	181 440	362 880
K_{11}	11	55	1 814 400	3 628 800
K_{12}	12	66	19 958 400	39 916 800
K_{13}	13	78	239 500 800	479 001 600
K_{14}	14	91	3 113 510 400	6 227 020 800
K_{15}	15	105	43 589 145 600	87 178 291 200

The minimum Hamiltonian cycles that were generated by both algorithms for all studied graphs are shown in Fig. 6 ÷ Fig. 11.

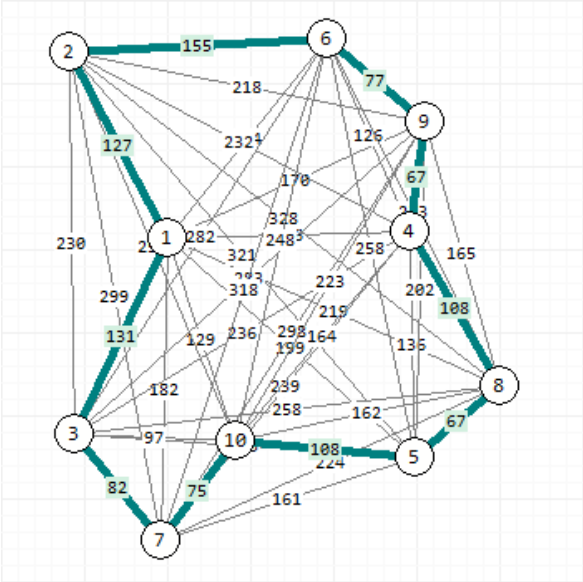

Fig. 6 K_{10} minimal Hamiltonian cycle

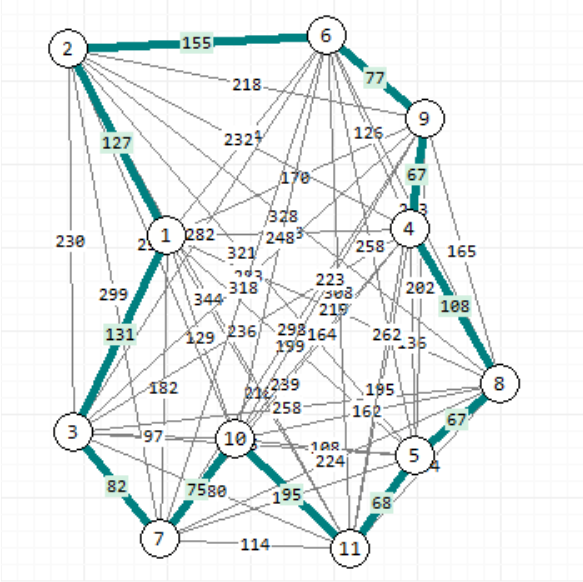

Fig. 7 K_{11} minimal Hamiltonian cycle

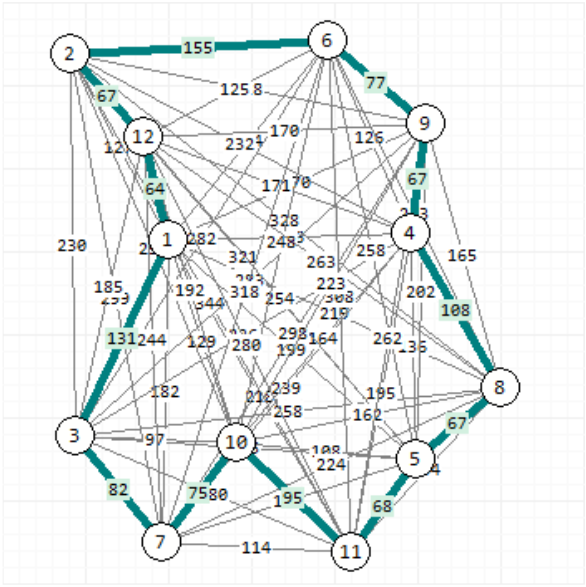

Fig. 8 K_{12} minimal Hamiltonian cycle

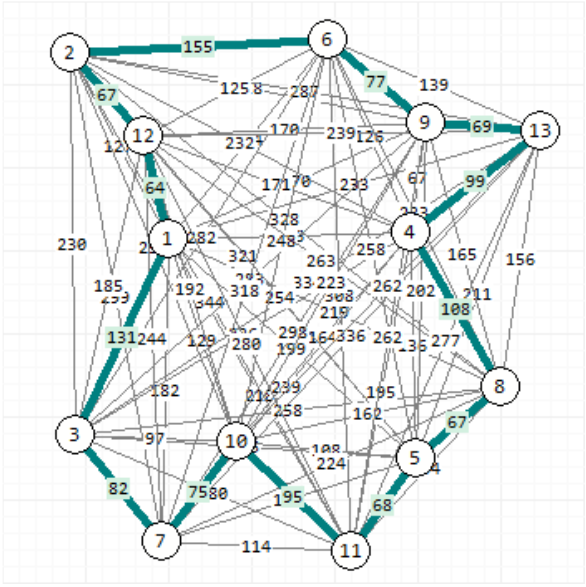

Fig. 9 K_{13} minimal Hamiltonian cycle

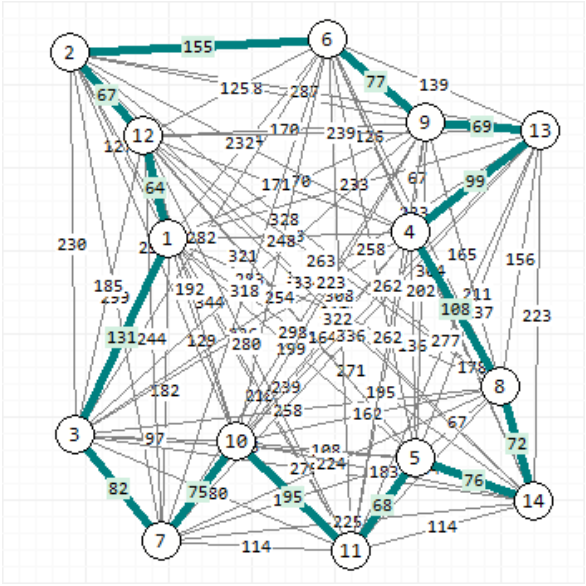

Fig. 10 K_{14} minimal Hamiltonian cycle

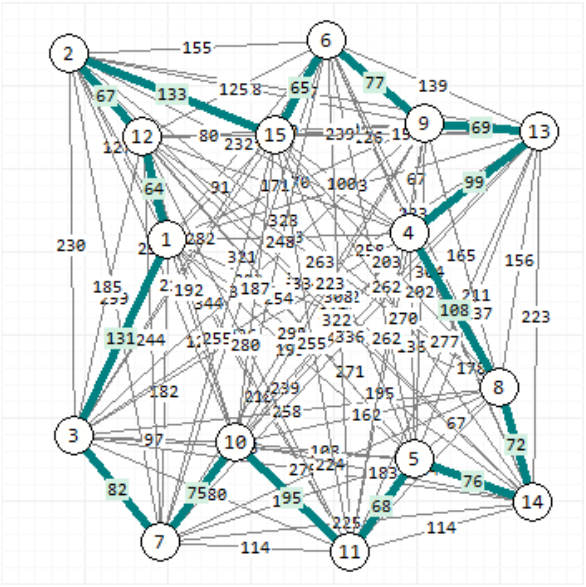

Fig. 11 K_{15} minimal Hamiltonian cycle

The results of the recursion based algorithm for input data $K_{10} – K_{15}$ are shown in Table IV.

TABLE IV
THE RESULTS OF THE RECURSION BASED ALGORITHM

G	Better Cycles	Recursive Calls	Time (ms)	Cycle Length
K_{10}	23	8 877 691	219	997
K_{11}	34	98 641 011	2 281	1 052
K_{12}	37	1 193 556 233	27 938	1 056
K_{13}	42	15 624 736 141	356 672	1 157
K_{14}	47	220 048 367 319	5 215 391	1 238
K_{15}	52	3 312 775 065 600	79 940 943	1 281

Table V shows the results from the iterative based algorithm for input data $K_{10} – K_{15}$.

TABLE V
THE RESULTS OF THE ITERATIVE BASED ALGORITHM

G	Better Cycles	Iterations by Loops	Time (ms)	Cycle Length
K_{10}	23	3 025 967	31	997
K_{11}	34	30 259 771	250	1 052
K_{12}	37	332 857 608	2 797	1 056
K_{13}	42	3 994 291 441	35 031	1 157
K_{14}	47	51 925 788 909	467 781	1 238
K_{15}	52	726 961 044 923	6 762 694	1 281

Table IV and Table V show the results of both algorithms. The columns are as follows: "G" – the abbreviation of the graph; "Better Cycles" – the number of the better cycles found in the search process; "Recursive Calls" – the number of recursive calls; "Iterations by Loops" – the number of iterations made by the iterative algorithm; "Time (ms)" – the execution time (in milliseconds) of the corresponding algorithm; "Cycle Length" – the length of the minimal Hamiltonian cycle (in pixels).

The results show that the found minimal Hamiltonian cycles are the same for both algorithms. This provides

grounds to analyze the values in the column "Time (ms)" that shows the execution time of both algorithms.

The influence of the number of vertices on the execution time for both algorithms is shown in Fig. 12.

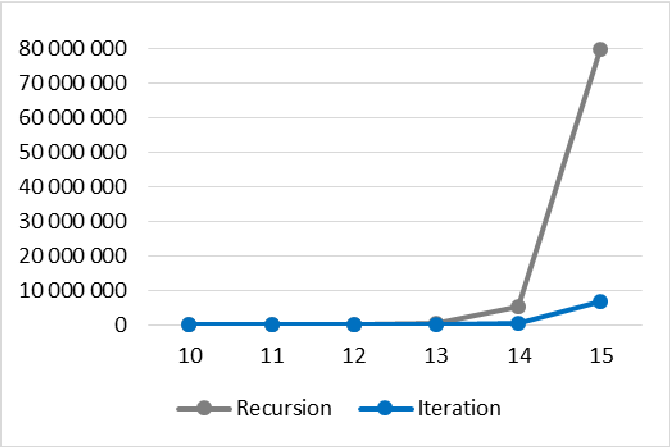

Fig. 12 Influence of the number of vertices (the x-axis) on the execution time (the y-axis in milliseconds) for both algorithms (for all input data)

Fig. 12 shows that the execution time of both algorithms increases exponentially with the increase of the number of the vertices. Additionally, the execution time of the recursive algorithm is significantly longer than the iterative algorithm (for the same input data).

Fig. 13 shows a chart of the data versus the execution time for both algorithms (the y-axis is transformed into a logarithmic one with base 2).

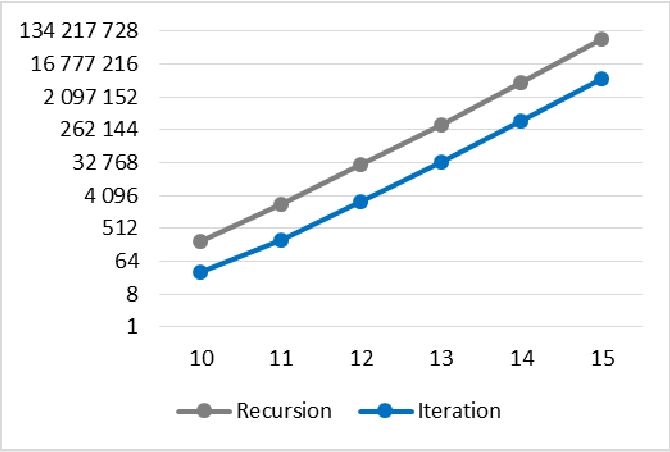

Fig. 13 Influence of the number of vertices (the x-axis) on the execution time (the y-axis in a logarithmic scale with base 2) for both algorithms

After transforming the y scale into a logarithmic one (with base 2), it can be seen that the delay of the recursive algorithm compared to that of the iterative one is within the range of 2^3 to 2^4 times (in milliseconds). This ratio is kept almost constant for all input data sets. The results show that under the same conditions (identical input data and operations), the iterative variant of this process compared to the recursive one is executed several times faster. This gives some idea of the nature of the computation process when it is modeled recursively or iteratively. The extra time for executing the recursive algorithm is due to the fact that additional computer instructions are executed at each recursive call. These instructions allocate and release additional memory for the new copies of all local variables (which are used by the recursive function). In the iterative algorithm, this process is executed by a counter and a transition instruction (to organize a cyclic process). This peculiarity in the realization of the two processes leads to the great difference in their execution time.

IV. Conclusion

In this paper, a comparative analysis between two algorithms for generating permutations but modified to find a minimal Hamiltonian cycle in complete undirected weighted graph was made. Both algorithms (respectively recursive and iterative) generate all permutations of n numbers, where n equals to the number of vertices in a given graph. For each generated permutation the Hamiltonian cycle length is calculated. This cycle is obtained after passing through each vertex of a graph in a sequence determined by the order of the numbers in the current permutation. The results obtained from both algorithms are identical both for the generated permutations and for the formed Hamiltonian cycles with minimum length. The difference between the two algorithms is the time for their execution. It was experimentally found out that the recursive algorithm is executed several times more slowly (in the order of 2^3 to 2^4 times) than the iterative algorithm.

In this study, TSP results are used to analyze the performance of both algorithms. Moreover, these results reveal that the solution to the TSP with exact methods (although it is possible) is not applicable for graphs with many vertices. There are approaches, such as branch-and-bound and others, that can reduce the number of the Hamiltonian cycles that are formed. However, for complete undirected weighted graphs with more than 50 vertices these approaches are not applicable. Therefore, for TSP with large complete graphs (e.g., with thousands of vertices) approximate methods should be used. Taking into consideration the results obtained in this study, one can suggest that if these methods can be implemented algorithmically by an iterative process, then, it must be chosen leaving aside the recursive one.

Acknowledgment

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 the paper.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed., Cambridge, Massachusetts, USA: The MIT Press, 2009.

[2] R. Sedgewick and K. Wayne, *Algorithms*, 4th ed., New Jersey, USA: Person Education, Inc., 2011.

[3] R. McCauley, B. Hanks, S. Fitzgerald, and L. Murphy, "Recursion vs. iteration: An empirical study of comprehension revisited," in *Proc. SIGCSE'15, 2015*, pp. 350-355.

[4] S. Bhaskar, "A Difference in Complexity Between Recursion and Tail Recursion," *Theory of Computing Systems*, vol. 60(2), pp. 299–313, Feb. 2017.

[5] Y. A. Liu and S. D. Stoller, "From recursion to iteration: What are the optimizations?," in *Proc. ACM SIGPLAN'00*, 2000, paper 57078, p. 73.

[6] P. G. Harrison and H. Khoshnevisan, “A new approach to recursion removal,” *Theoretical Computer Science*, vol. 93(1), pp. 91–113, Feb. 1992.
[7] K. Sundararajah, L. Sakka, and M. Kulkarni, “Locality transformations for nested recursive iteration spaces,” in *Proc. ASPLOS’17*, 2017, paper 127193, p. 281.
[8] D. Insa and J. Silva, “Automatic transformation of iterative loops into recursive methods,” *Information and Software Technology*, vol. 58, pp. 95–109, Feb. 2015.
[9] A. Filinski, “Recursion from iteration,” *LISP and Symbolic Computation*, vol. 7(1), pp. 11–37, Jan. 1994.
[10] J. Ding, “Data filtering based recursive and iterative least squares algorithms for parameter estimation of multi-input output systems,” *Algorithms*, vol. 9(3), 49, Sep. 2016.
[11] M. Kazemi and M. M.Arefi, “A fast iterative recursive least squares algorithm for Wiener model identification of highly nonlinear systems,” *ISA Transactions*, vol. 67, pp. 382–388, Mar. 2017.
[12] Z. He and D. Ding, “Efficient recursive-iterative solution for EM scattering problems,” *Electronics Letters*, vol. 51(4), pp. 306–308, Feb. 2015.
[13] R. J. Wilson, *Introduction to Graph Theory*, 5th ed., New Jersey, USA: Prentice Hall, 2010.
[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] 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.
[16] G. Gutin and A.P. Punnen, The Traveling Salesman Problem and Its Variations (Combinatorial Optimization), 2nd ed., New York City, USA: Springer, 2007.
[17] 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.
[18] G. Benoit and S. Boyd, “Finding the exact integrality gap for small traveling salesman problems,” *Mathematics of Operations Research*, vol. 33(4), pp. 921–931, Nov. 2008.
[19] 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.
[20] 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, Jul. 2017.
[21] 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.
[22] M. Mestria, “A hybrid heuristic algorithm for the clustered traveling salesman problem,” *Pesquisa Operacional*, vol. 36(1), pp. 113–132, Jan-Apr. 2016.
[23] 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.
[24] 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.