Online-Journals.org

iJEP International Journal of Engineering Pedagogy

iJEP | eISSN: 2192-4880 | Vol. 13 No. 1 (2023) | OPEN ACCESS

https://doi.org/10.3991/ijep.v13i1.35661

PAPER

An Interactive Application for Learning and Analyzing Different Graph Vertex Cover Algorithms

Velin Kralev(✉),
Radoslava Kraleva,
Viktor Ankov

South-West University, Blagoevgrad, Bulgaria

velin_kralev@swu.bg

ABSTRACT

This paper deals with an analysis of three algorithms for the graph vertex cover problem. Certain methods for solving this problem are analyzed. In addition, different studies on the problem and some approaches to its solution are discussed as well. An exact algorithm (based on the backtracking approach) is presented. Calculating the average time for execution of this algorithm is consistent with the multitasking way of work of the operating system. For this purpose, four different starts of the algorithm are made and then the average time of all of them is calculated. The exact algorithm found the optimal solutions for all analyzed graphs. Besides this algorithm, two other heuristic algorithms for solving the problem are discussed. For this study, an interactive application is developed to visualize the performance of the three algorithms and display the obtained results. The results show that for small graphs with no more than 25 vertices, the exact algorithm can be used to solve optimally the graph vertex cover problem. For the largest graphs, none of the two heuristic algorithms found the optimal solutions, but these algorithms generated solutions that are very close to the optimal ones. In summary, when the size of the graph increases linearly, the execution time of the heuristic algorithms increases linearly, while the execution time of the exact algorithm increases exponentially.

KEYWORDS

graph theory, vertex cover problem, heuristic algorithms

1 INTRODUCTION

Graph theory is an important part of discrete mathematics [1]. This theory has developed significantly over the past several decades [2]. Graphs are non-linear data structures used in discrete mathematics and computer science. These are abstract structures that provide opportunities to visually and very effectively formalize and solve sufficiently complex applied problems. Different types of problems from different fields of practice can be formulated and modeled by graphs so that they can be solved by appropriate algorithms. This presupposes finding

Kralev, V., Kraleva, R., Ankov, V. (2023). An Interactive Application for Learning and Analyzing Different Graph Vertex Cover Algorithms. *International Journal of Engineering Pedagogy (iJEP)*, 13(1), pp. 4–19. https://doi.org/10.3991/ijep.v13i1.35661

Article submitted 2022-09-27. Resubmitted 2022-11-27. Final acceptance 2022-12-01. Final version published as submitted by the authors.

© 2023 by the authors of this article. Published under CC-BY.

precise structural and numerical characteristics of real objects that are represented by graph structures [3–8].

Many transportation problems, site allocation problems, optimal route selection problems or service center placement problems, and problems related to creating optimal schedules and timetables are also described by graphs and solved by the corresponding algorithms [9–13]. Graphs are also used for modeling in the field of learning [14–16] and assessment [17]. In this way, various physical, chemical, economic and managerial systems and processes can be successfully represented and studied with graphs. Solving these problems without a computer and appropriate software is impossible except in trivial cases where the size of the input data is very small. But even with the use of a computer and appropriate software, solving some problems accurately in an acceptable amount of time could also be impossible. These include all problems from the complexity classes NP-complete and NP-hard, for which no solution that has a complexity other than exponential, for example polynomial, has been found yet [18, 19].

Some of these problems are linear, i.e., both the objective function and each of the constraints are linear, meaning they are solvable through polynomial algorithms [20]. However, real problems have a large input size, so it is necessary to search for efficient algorithms that generate approximate solutions [21, 22]. Graph theory provides good opportunities in this direction [23].

Therefore, in the study of graph theory, it is of particular importance to develop and use interactive applications (software products) for working with graphs and for analyzing algorithms in graphs. These applications can visualize the modeled problems and also test the corresponding algorithms related to these problems. In addition, the obtained results can be analyzed through graphical representation. The subject of the present research is the development of an application for working with graphs and implementing different algorithms for graph vertex covering.

2 LITERATURE REVIEW

In the field of graph theory, a vertex cover of a graph is a set of vertices such that every edge of the given graph is incident to at least one of the vertices of this set. The problem of finding a minimum vertex cover of a graph is an optimization problem that falls into the category of classical NP-complete problems [24].

Each graph can be described by two sets – V and E. The set V contains the nodes (vertices) of the graph and the set E contains its edges. The set of edges contains elements that connect exactly certain pairs of vertices. The sets V and E are finite. The vertices of each edge can be represented as an ordered or an unordered pair. If the vertices of each edge are an ordered pair, then the graph is called directed. Otherwise, the graph is called undirected. If a numerical value is assigned to each edge, then the graph is called weighted [25, 26]. Once these notations have been introduced, the graph vertex cover problem can also be defined.

A vertex cover VC of an undirected graph G = (V, E) is a subset of the vertices V such that at least one of the two vertices of any edge (v1, v2) $\in$ E of the graph G belongs to the set VC. In other words, either v1 $\in$ VC, or v2 $\in$ VC, or both v1 and v2 $\in$ VC. Then a vertex cover of a graph can also be defined as a set of vertices VC such that each edge of the graph has at least one of its two vertices (or both of its vertices) as an element belonging to the set VC (of both vertices, incident to this edge). Such a set of vertices is said to “cover” all the edges of the graph. A minimum vertex cover is one that has the smallest possible power. In other words, a vertex cover is a minimum vertex cover if it contains the smallest possible number of

vertices such that all edges in the graph are incident to at least one of the vertices in that set [27]. Since the problem of finding a minimal vertex cover in a graph is an NP-complete problem, this means that no efficient (polynomial) algorithm is yet known to solve this problem exactly.

For some special types of graphs, for example, tree graphs, there is a polynomial algorithm that finds a minimum vertex cover in polynomial time. This algorithm searches for a vertex in the graph that is of type leaf, then adds its parent to the current vertex cover (i.e., the cover that has been built so far), then deletes the found leaf, the parent, and all associated edges. The algorithm continues according to the same scheme until the vertices in the graph are exhausted [28].

A vertex cover in a graph can be generated such that it has an approximation factor no worse than twice the optimal one. This can easily be done by repeatedly selecting an edge from the graph that has not been used yet, then adding the two incident vertices with that edge to the currently constructed vertex cover, then removing them from the graph [29]. Whenever the two vertices incident to the selected edge are removed from the graph, all incidents with those edge vertices are also removed. The set containing the vertices selected in the described manner is actually a vertex cover of the given graph. Also, if the edge $e = \{v1, v2\} \in E$, then every cover of the graph will contain either a vertex v1 or a vertex v2 or both vertices; otherwise, the edge connecting vertices v1 and v2 will not be covered. Thus, the optimal cover contains at least one of the two vertices of any edge, i.e., any one covering set of a given graph will be at most twice as large (in power) as the optimal covering set for that graph [30]. There are also algorithmic techniques where the coefficient of approximation to the optimal solution is smaller [31, 32].

One more feature of the problem of finding a minimum vertex cover is important to mention: finding a vertex cover of minimum size is actually equivalent to finding the maximum set of independent vertices (in the same graph). This problem is known as Maximum-Size Independent Set [33–37].

The problem of finding a minimal vertex cover is used to model many real and theoretical problems. For example, in a commercial establishment, it is necessary to install as few video surveillance cameras as possible. The corridors in the site are a closed circuit, i.e., it is necessary to "cover" all corridors (edges) that connect all rooms (vertices). This problem can easily be modeled as a minimum vertex cover problem. The ability to use the minimum vertex cover problem to model different processes and activities makes this problem widely applicable in other areas as well. Another application of this approach is in modeling some processes in the field of biology [38].

Since finding a minimum vertex cover is an NP-complete problem, in addition to one exact algorithm, two heuristic algorithms will be presented to solve it. The idea behind a heuristic algorithm is that it targets one of all subcases of the problem and solves only it. The selection of this subcase is based on a local optimality criterion (such as the number of edges incident to a given vertex). In this way, the heuristic algorithm is always directed to the best choice, but at a local level. However, this choice may prove inappropriate at the global level [39–42]. Heuristic algorithms are not difficult to create, respectively, and their implementation is not complicated. A major drawback of these algorithms is that the solutions they generate are not always optimal. However, this does not reduce their usefulness, since these algorithms quickly find a solution to the problem which can be optimal or close to it. For many practical problems, it is impossible to examine all possible cases when searching for a solution. In this case, quickly finding an approximate solution (close to the optimal one) can be much more useful, instead of the long search for the real optimal solution [43, 44].

In this paper, three algorithms for the graph vertex cover problem will be presented – one exact and two heuristic. The exact algorithm uses a backtracking approach and finds the exact solutions for all graphs [22, 43]. The other two algorithms are heuristic (approximate) and therefore are characterized by the fact that it is not always possible to find the optimal (exact) solution. It is also characteristic of these algorithms that if they do not find the optimal solution, the solution found will be close to the optimal one [45, 46].

3 IMPLEMENTATION OF THE ALGORITHMS

As discussed in the previous section, three algorithms will be implemented to find a minimum vertex cover in a graph. The main idea is to implement the algorithms in such a way that they can be easily integrated into an application that uses them. If the implementation of the algorithms and the presentation of their results is implemented in an interactive way, the students will be able to visually see the step-by-step execution of the algorithms and analyze their results [47–49]. When the application runs, the algorithms will access some global data structures (variables and dynamic arrays). The algorithms will use these data structures when they are executed. These data structures will also be accessible by all functions at the runtime of the application. Dynamic array and variable declarations are predefined as shown in Figure 1 (in C++ language).

```
01  unsigned long tS, tE;
02  int CoverVertexCount, CoverEdgeCount;
03  DynamicArray < int > permutation;
04  DynamicArray < int > best_permutation;
05  DynamicArray < TVertex > v; DynamicArray < TEdge > e;
```

Fig. 1. Global declarations source code

The variables CoverVertexCount and CoverEdgeCount (of integer type int) will be used as counters. They will store the number of covered vertices and edges in the graph. The tS integer variable of the type long will store a timestamp (in milliseconds) from the start of the operating system boot to the moment a particular algorithm has been run. Accordingly, the variable tE will store the completion time of the execution of the given algorithm. The other global declarations are of the type dynamic arrays. The arrays [v] and [e] are of the structure types TVertex and TEdge, respectively. The arrays [permutation] and [best_permutation] are of integer type int. Dynamic arrays are characterized by the fact that when the application starts, no RAM is allocated for them, but only a pointer to the address of the structure in RAM. The size of this structure dynamically changes as the application runs, depending on the number of items that need to be stored. The main idea of this type of dynamic structures is to allocate the necessary memory only when it is needed and then during the execution of the application. The dynamic array [v] of type TVertex will store the data for the vertices of the graph; respectively, the dynamic array [e] of type TEdge will store the data for the edges of the same graph. The dynamic array [permutation] of integer type int will store a randomly generated permutation of the vertices of the graph. The dynamic array [best permutation], also of integer type int, will store the permutation of the graph vertices in which the generated vertex cover is respectively optimal.

An implementation of the first algorithm (BTR) using the backtracking method to find a minimum vertex cover of a graph is presented in Figure 2. This algorithm generates all combinations of n elements of the kth class, for values of k from 1 to n. For each combination (i.e., for each subset of k number of vertices of the graph), the algorithm checks whether all edges of the graph can be covered by only these k vertices. When the algorithm finds a solution, it terminates its execution. The solution found will be the first solution for a given k (i.e., k will be the minimum number of vertices needed to cover all edges of the graph), which is actually the condition of the minimum vertex cover problem.

```
01 void BTRStart() {
02   FoundSolution = false;
03   Terminated = false;
04   n = g.v; c = 0;
05   tS = GetTickCount();
06   for (k = 1; k <= n; k++) {
07     c = CalcCombs(n, k);
08     combination.set_length(k);
09     generate(1);
10     if (FoundSolution) { break; }
11     if (Terminated) { break; } }
12   tE = GetTickCount();
13   if ((Terminated == true) && (FoundSolution == false)) {
14     ShowMessage("The process is terminated!"); }
15   if ((Terminated == false) && (FoundSolution == true)) {
16     DrawGraph();
17     sb->Items[3]->Text = " CV : " + IntToStr(CoverVertexCount);
18     sb->Items[4]->Text = " CE : " + IntToStr(CoverEdgeCount);
19     sb->Items[5]->Text = " time : " + IntToStr(tE-tS) + " ms";
20     ShowMessage("Found solution."); }
21 }
```

Fig. 2. Source code of the BTR algorithm based on the backtracking method

The two global variables – tS and tE – are used to calculate the execution time of the algorithm. They are of the type long and store the two moments in time (relative to the start of the operating system), respectively, for the beginning and for the end of the execution of the algorithm (lines 05 and 12).

The *for* loop (lines 06–11) is used to iterate through every possible combination of k (k is a global variable that indicates how large the subset of vertices of the graph is through which an attempt is made to cover all edges of the graph). This process runs until the smallest k is found for which a vertex cover is generated, or until the process is terminated by the user. The *if* statement on line 13 checks if the process has been interrupted by the user while no solution has been found yet. If the value of the boolean expression of the conditional operator is true, then information about the partially generated solution is not displayed. If a solution is found and the process is not interrupted by the user, the source code in the *if* statement block (lines 15–20) redraws the graph and displays information about the values of the variables involved in the process, for example, the number of edges covered (CoverEdgeCount), the number of vertices used (CoverVertexCount), and the elapsed time determined by the difference between the values of the variables tE and tS.

Another algorithm (RND) used to find a vertex cover in a graph is presented in Figure 3. This algorithm is heuristic and is based on the generation of random orders of vertices (permutations).

```
01 void RNDStart(int Count) {
02   if ((Count < 1) || (Count > 10000))
03     { ShowMessage("Incorrect parameter [Count]"); return; }
04   permutation.set_length(g.v);
05   best_permutation.set_length(g.v);
06   int MinCoverVertexCount = MaxInt, MinCounter = 0;
07   tS = GetTickCount();
08   for (int counter = 1; counter <= Count; counter++) {
09     for (int I = 1; I <= g.v; i++) { permutation[i] = I; }
10     random_shuffle(permutation.begin() + 1, permutation.end());
11     for (int I = 1; I <= g.v; i++) {
12       int v_id = permutation[i];
13       v[v_id].c = clLime; CoverVertexCount ++;
14       for (int j = 1; j <= g.e; j++) {
15         if ((e[j].v1 == v_id) || (e[j].v2 == v_id)) {
16           if(e[j].c != clBlue) {
17             e[j].c = clBlue; CoverEdgeCount ++; } } }
18       if (CoverEdgeCount == g.e) { break; } }
19     if (MinCoverVertexCount > CoverVertexCount) {
20       MinCoverVertexCount = CoverVertexCount;
21       MinCounter = counter;
22       for (int I = 1; I <= g.v; i++) {
23         best_permutation[i] = permutation[i]; } } }
24     tE = GetTickCount();
25   for (int I = 1; I <= g.v; i++) {
26    permutation[i] = best_permutation[i]; }
27   DrawGraph();
28   sb->Items[3]->Text = " CV : " + IntToStr(CoverVertexCount);
29   sb->Items[4]->Text = " CE : " + IntToStr(CoverEdgeCount);
30   sb->Items[5]->Text = " time : " + IntToStr(tE-tS) + " ms";
31 }
```

Fig. 3. Source code of the RND algorithm based on the generation of random orders of vertices

One input parameter of this algorithm is the number of iterations (specified by the user) which determines how many iterations the algorithm will make to generate random orders (permutations) of the vertices. For each order of vertices, the algorithm will "cover" the edges of the graph by traversing the vertices in the current order. If the number of iterations is not set correctly by the user, the RNDStart function exits.

Similarly to the BTR algorithm, the RND algorithm uses the variables tE and tS to calculate the execution time of the algorithm. The *for* loop (lines 08–24) first sorts the elements in the permutation array from 1 to g.v, where g.v is the number of vertices in the graph (line 09). The elements in the dynamic array permutation are then shuffled randomly (line 10). In the next step, the vertices are traversed in the specified (random) order, and the edges incident to the corresponding vertices are colored, i.e., covered (lines 14–17). The conditional *if* statement (line 18) checks whether a solution has already been found for this arrangement of vertices. This solution must be better than the last best found. The value of the last best solution (i.e., the minimum number of vertices needed to cover the edges of the graph under the previous order) is stored in the MinCoverVertexCount variable. If the condition check on line 19 is true, then the current permutation of the vertices is stored in the best_permutation array, and the MinCoverVertexCount variable stores the value of the CoverVertexCount variable. After Count orders of vertices have been generated, the sequence of vertices where the best solution is found is transferred to the permutation array. The last best permutation is still stored in the best_permutation array, so its contents are transferred to the permutation array (lines 25–26). The graph is then redrawn and the values of the parameters are visualized (lines 27–303).

The implementation of the third algorithm (SRT) for finding a vertex cover of a graph is presented in Figure 4. This algorithm is also heuristic and approximate.

```
01 void SRTStart() {
02   permutation.set_length(0); permutation.set_length(g.v + 1);
03   for (int i = 0; i <= g.v; i++) { permutation[i] = 0; }
04   LoadTemporaryGraph();
05   tS = GetTickCount(); int vindex = 1;
06   while (tg.e > 0) {
07     int max_tvindex = GetTemporaryVertexIndexWithMaxDegree();
08     permutation[vindex] = tv[max_tvindex].c;
09     DeleteTemporaryVertex(max_tvindex); vindex ++; }
10   for (int i = 1; i <= g.v; i++) {
11     if (permutation[i] == 0) { break; }
12     int v_id = permutation[i];
13     v[v_id].c = clLime; CoverVertexCount ++;
14     for (int j = 1; j <= e.High; j++) {
15       if ((e[j].v1 == v_id) || (e[j].v2 == v_id)) {
16         if(e[j].c != clBlue) {
17           e[j].c = clBlue; CoverEdgeCount ++; } } } }
18   tE = GetTickCount();
19   DrawGraph();
20   sb->Items[3]->Text = " CV : " + IntToStr(CoverVertexCount);
21   sb->Items[4]->Text = " CE : " + IntToStr(CoverEdgeCount);
22   sb->Items[5]->Text = " time : " + IntToStr(tE-tS) + " ms";
23 }
```

Fig. 4. The code of the SRT algorithm based on ordering the vertices according to their degree

When executing the SRT algorithm, the graph is first copied into a temporary graph using the LoadTemporaryGraph function (line 4). A timestamp is then stored against which the execution time of the algorithm will be calculated. The method for calculating the execution time of the algorithm is the same as for the other two algorithms. Through the loop that is initialized on line 6, the execution of the first step of the algorithm begins, where the index of the vertex with the greatest degree is found. The index of this vertex is then stored in the dynamic array permutation (line 8). In the next step, all edges that are incident to the current vertex are removed (line 9), then the number of vertices forming the current vertex cover is increased by one. This cover may still be partial. This process is repeated until the condition of the while loop is true. When all the edges of the graph are covered, i.e., the condition of the while loop assumes a value of false, the execution of the loop is terminated and the execution of the *for* loop initialized on line 10 (through line 17) is started. Through this loop, all the vertices are traversed and only those of them that are stored in the dynamic permutation array are colored, since they are part of the vertex cover. The edges incident to these vertices are also colored. The graph is then redrawn to visualize the result.

4 EXPERIMENTAL RESULTS

For this study, an interactive application (named VertexCoverFinder) was developed to visualize the performance of the algorithms and conduct the experiments as well as to display the results as shown in Figure 5.

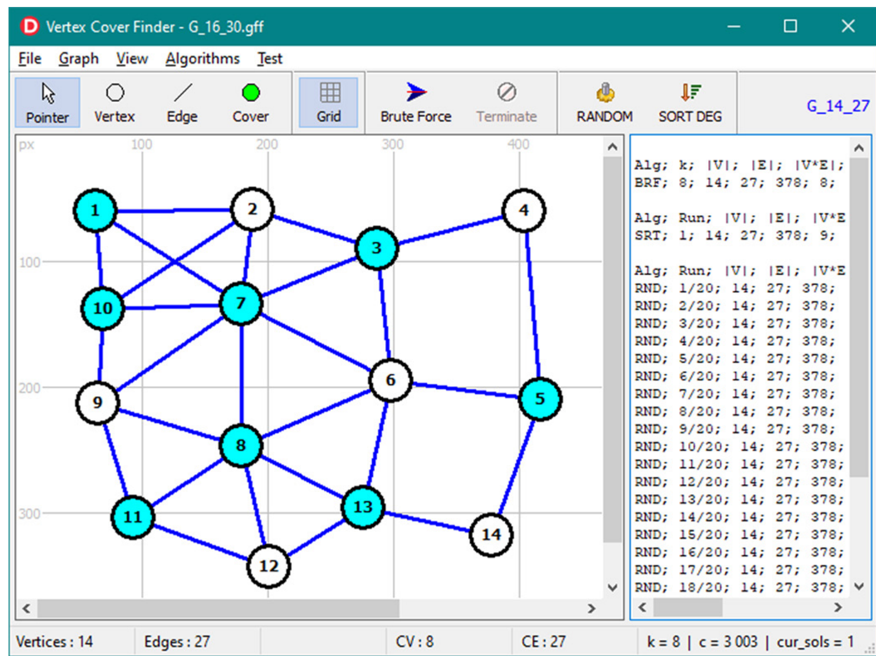

Fig. 5. A working session with the VertexCoverFinder applicaton

The VertexCoverFinder application offers a rich set of possibilities that the user can use in the work process. This application provides the ability to interactively create and edit graph structures. In addition, it allows running the three algorithms for finding the minimum vertex cover in a graph, which were presented in the previous section. The application also provides the ability to save in an external file the graphs created. The saved graphs can later be loaded from the external file and edited into the designer. The activity diagram of the application is shown in Figure 6.

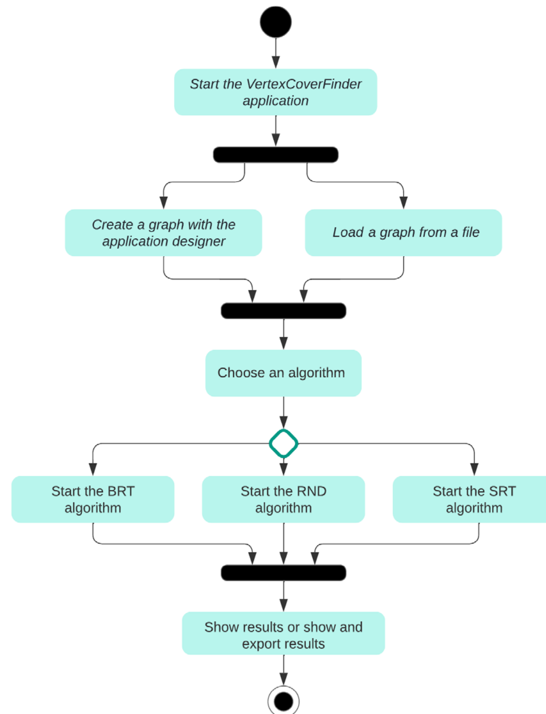

Fig. 6. Activity diagram for the VertexCoverFinder application

The aim of the experiment, using an application developed to check graphs for the number of vertices (and, respectively, edges), was to identify the exact algorithm (based on the backtracking method) that can be used to find an optimal solution (i.e., to find the minimum number of vertices which are incident to all edges of the graph). It was necessary to analyze the execution times of the algorithms and the quality of the generated solutions. For this purpose, it was necessary to make a comparative analysis between the algorithms. The comparison between the algorithms was performed with the same input data.

The VertexCoverFinder application was run on a standard computer configuration with Windows 10 Home 64-bit operating system and the following hardware configuration: CPU: Intel(R) Core(TM) i7-7700MQ CPU @ 2.80GHz 2.81GHz; RAM memory: 16.00 GB. The experiment results may be stored in a database and accessed by a web service, as presented in [50] and [51].

For the purposes of the experiments, 18 graphs were created and stored using the CoverVertexFinder application. Table 1 presents summary data for the graphs. In addition, Table 1 also shows the results of the exact algorithm (BRT). The data presented are for the minimum number of vertices that need to be selected (colored) to "cover" all the edges in the graph. Also, the time is shown (in two different formats) to execute the algorithm for each of the analyzed graphs.

Table 1. Summary results after running the exact algorithm

Graph (G)	|V|	|E|	|V×E|	VC	Time (ms)	Time (h, min, s)
G_15_24	15	24	360	9	156	< 1 s
G_16_30	16	30	480	10	422	< 1 s
G_17_35	17	35	595	11	1,062	1 s
G_18_39	18	39	702	12	3,109	3 s
G_19_45	19	45	855	13	6,171	6 s
G_20_49	20	49	980	13	13,140	13 s
G_21_53	21	53	1,113	14	30,859	31 s
G_22_57	22	57	1,254	15	69,843	1 min, 10 s
G_23_65	23	65	1,495	16	173,250	2 min, 53 s
G_24_73	24	73	1,752	17	484,734	8 min, 6 s
G_25_83	25	83	2,075	18	1,135,625	18 min, 54 s
G_26_95	26	95	2,470	19	3,954,000	1 h, 6 min
G_27_103	27	103	2,781	20	5,987,609	1 h, 40 min
G_28_114	28	114	3,192	21	14,234,515	3 h, 57 min
G_29_120	29	120	3,480	22	41,219,172	11 h, 27 min
G_30_133	30	133	3,990	23	95,067,359	26 h, 14 min
G_31_144	31	144	4,464	24	212,546,766	59 h, 2 min
G_32_157	32	157	5,024	25	480,086,110	133 h, 21 min

In Table 1, the columns |V| and |E| contain the number of vertices and edges in each graph. The "VC" column contains the number of vertices that form the vertex cover for each of the analyzed graphs.

The purpose of this experiment was to experimentally determine how increasing the size of the input data (i.e., the number of vertices and edges in the corresponding graph) affects the execution time of the exact algorithm.

Figure 6 shows the influence of increasing the number of vertices (and, respectively, edges) in the graph (x-axis) on the execution time of the exact algorithm (y-axis – in hours).

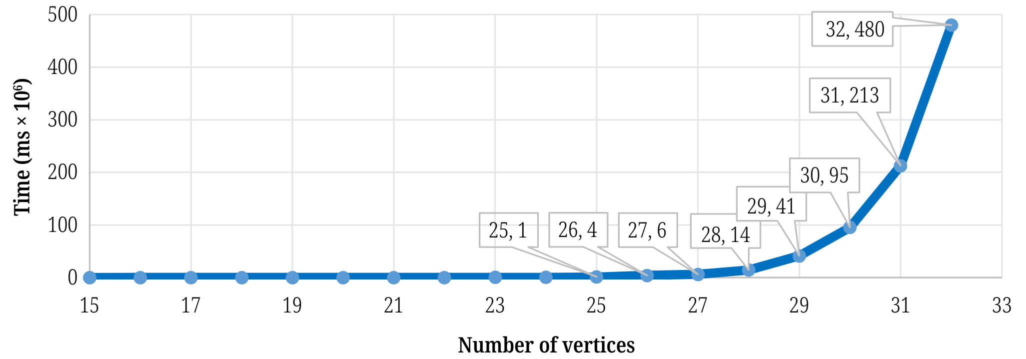

Fig. 7. Effect of increasing the number of vertices in the graph on the execution time of the exact algorithm

Table 1 and Figure 7 show that for a graph with 32 vertices and 157 edges, the execution time of the exact algorithm was unacceptably long (480,086,110 milliseconds, or 133 hours and 21 minutes, respectively). If a conditional bound on the acceptable execution time of the exact algorithm is set, for example 10 minutes, then the results show that for graphs with more than 24 vertices, the exact algorithm will run for an unacceptably long time. Figure 7 also shows that with a linear increase in the number of vertices in the graph (and edges, respectively), the execution time of the exact algorithm increases exponentially.

Table 2 shows the summary results after running the three algorithms for the 8 graphs with the most vertices and edges, respectively from G_25_83 to G_32_157.

Table 2. Summary results after running the three algorithms (BTR, SRT and RND)

Graph	BTR		SRT		RND	
	VC	Time (ms)	VC	Time	VC	Time (ms)
G_25_83	18	1,135,625	19	< 0.01 s	19	42
G_26_95	19	3,954,000	20	< 0.01 s	20	47
G_27_103	20	5,987,609	21	< 0.01 s	22	56
G_28_114	21	14,234,515	22	< 0.01 s	23	62
G_29_120	22	41,219,172	23	< 0.01 s	23	78
G_30_133	23	95,067,359	24	< 0.01 s	26	94
G_31_144	24	212,546,766	25	< 0.01 s	25	103
G_32_157	25	480,086,110	26	< 0.01 s	27	116

The results obtained for the eight graphs with the most vertices and edges were compared, both in terms of the number of vertices forming a vertex cover (the VC

columns in Table 2) and also in terms of the execution time of the algorithms (the Time columns in Table 2).

Figure 7 shows a comparison between the values generated for the number of vertices forming a vertex cover (for the analyzed graphs) by the three algorithms.

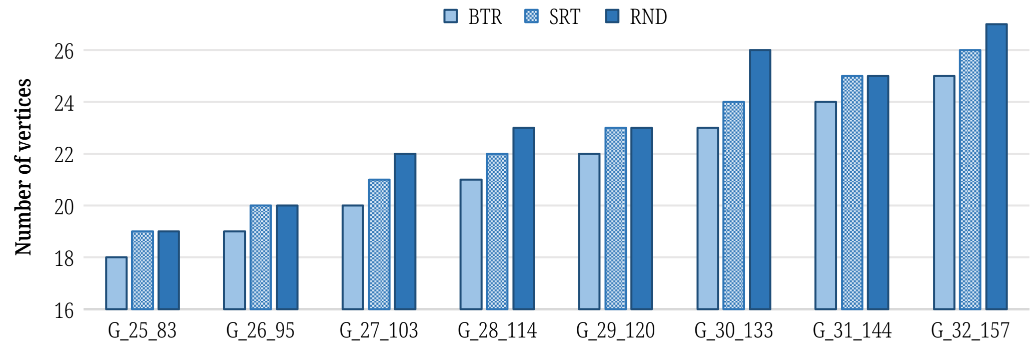

Fig. 8. Comparison between the solution values generated by the three algorithms (for graphs G_25_83–G_32_157)

Table 2 and Figure 8 show that for the largest graphs, respectively G_25_83–G_32_157, none of the two heuristic algorithms (STR and RND) found the optimal solutions. We will note that the optimal solutions for these graphs are known (generated by the exact algorithm – BRT) and are presented in Table 1. Upon a more detailed analysis of the data in Table 2, it can be noticed that the SRT algorithm generated solutions that are very close to optimal (the difference is exactly one vertex). Unlike the SRT algorithm, in half of the cases the RND algorithm generated worse solutions than the SRT algorithm. However, compared to the exact algorithm (respectively, compared to the optimal solutions), the RND algorithm generated significantly worse ones, with differences ranging between 1 and 3 vertices (relative to the optimal). Figure 9 shows a comparison between the results of the approximate algorithms.

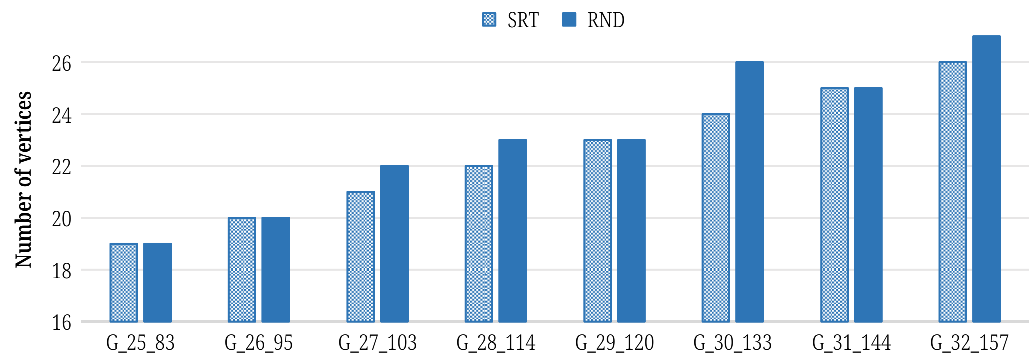

Fig. 9. Comparison between the solution values generated by the two approximate algorithms (for graphs G_25_83–G_32_157)

Figure 10 shows how increasing the number of vertices and edges in the graph affects the execution time of the RND algorithm.

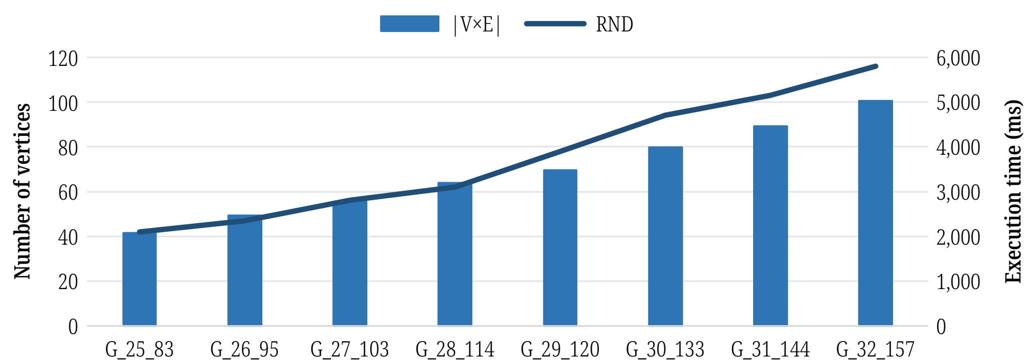

Fig. 10. Influence of the number of vertices and edges in the graph on the execution time of the algorithm with randomly generated vertex orders – RND

Table 2 and Figure 10 show that as the number of vertices in the graph increases linearly, the execution time of the RND heuristic algorithm also increases linearly. Since the analyzed graphs are a small number of vertices and edges, this time is short (values on the order of milliseconds). Unlike the RND algorithm, the SRT algorithm runs significantly faster. The execution time of the SRT algorithm is negligible.

5 SUMMARY AND CONCLUSIONS

In this paper, an analysis of three algorithms for the graph vertex cover problem was discussed. Some methods for solving this problem were analyzed. In addition, different studies of the problem and some approaches to its solution were discussed. An exact algorithm (based on the backtracking approach) was presented. Calculating the average time for execution of this algorithm was consistent with the multitasking way of work of the operating system. For this purpose, 4 different starts of the algorithm were made, and then the average time of all of them was calculated. The exact algorithm found the optimal solutions for all analyzed graphs. Besides this algorithm, two other heuristic algorithms for solving the problem were discussed as well. For this study, an interactive application was developed to visualize the performance of the algorithms and display their results. The results show that for small graphs with not more than 25 vertices, the exact algorithm can be used to solve optimally the graph vertex cover problem. For the largest graphs, neither of the two heuristic algorithms found the optimal solutions, but these algorithms generated solutions that are very close to optimal ones. In summary, when the size of the graph (the number of the vertices) increases linearly, the execution time of the exact algorithm increases exponentially, but the execution time of the heuristic algorithms increases only linearly.

The obtained experimental results show that in all considered cases, the SRT algorithm generated solutions that differ from the optimal ones by only one vertex. These values show that this algorithm keeps a relatively good (constant) approximation of the obtained results (compared to the optimal ones). Moreover, the execution time of this algorithm is very short (for all analyzed graphs on the order of less than 100 ms).

The interactive GraphVertexFinder application enables the creation and editing of various graph structures in a convenient and easy way. The possibility of step-by-step implementation of the algorithms integrated in the application, as well

as the option of analyzing the results obtained from them, makes possible easier teaching by teachers, as well as easier studying by students, of the educational material related to topics in the field of graph algorithms. The study can be extended by implementing and integrating additional graph vertex cover algorithms.

6 REFERENCES

[1] Alekseev, V.E., Boliac, R., Korobitsyn, D.V., Lozin, V.V. (2007). NP-hard graph problems and boundary classes of graphs. Theoretical Computer Science, 389(1–2): 219–236. https://doi.org/10.1016/j.tcs.2007.09.013

[2] Kralev, V., Kraleva, R. (2020). Methods for software visualization of large graph data structures. International Journal on Advanced Science, Engineering and Information Technology, 10(1): 1–8. https://doi.org/10.18517/ijaseit.10.1.10739

[3] Sabeen, S., Arunadevi, R., Kanisha, B., Kesavan, R. (2019). Mining of sequential patterns using directed graphs. International Journal of Innovative Technology and Exploring Engineering, 8(11): 4002–4007. https://doi.org/10.35940/ijitee.K2242.0981119

[4] Chen, J. (2010). An UpDown directed acyclic graph approach for sequential pattern mining. IEEE Transactions on Knowledge and Data Engineering, 22(7): 913–928. https://doi.org/10.1109/TKDE.2009.135

[5] Kurapov, S.V., Davidovsky, M.V., Tolok, A.V. (2018). A modified algorithm for planarity testing and constructing the topological drawing of a graph. The thread method. Scientific Visualization, 10(4): 53–74. https://doi.org/10.26583/sv.10.4.05

[6] Xu, J., Qiang, X., Zhang, K., Zhang, C., Yang, J. (2018). A DNA computing model for the graph vertex coloring problem based on a probe graph. Engineering, 4(1): 61–77. https://doi.org/10.1016/j.eng.2018.02.011

[7] Tognon, C.H., Kharabsheh, R.A. (2022). Some properties of the formal local cohomology module and application in the theory of graphs. Applied Mathematics and Information Sciences, 16(1): 45–49. https://doi.org/10.18576/amis/160105

[8] Sarkar, D., Konwar, P., De, A., Goswami, S. (2020). A graph theory application for fast and efficient search of optimal radialized distribution network topology. Journal of King Saud University – Engineering Sciences, 32(4): 255–264. https://doi.org/10.1016/j.jksues.2019.02.003

[9] Wong, E.Y.C., Tai, A.H., So, S. (2020). Container drayage modelling with graph theory-based road connectivity assessment for sustainable freight transportation in new development area. Computers and Industrial Engineering, 149, 106810. https://doi.org/10.1016/j.cie.2020.106810

[10] Li, S., Xu, J., Cele, S. (2019). Application of graph theory in transportation linkage in logistics management and its computer aided model design. Journal of Intelligent and Fuzzy Systems, 37(3): 3319–3326. https://doi.org/10.3233/JIFS-179134

[11] Balaji, S., Obaidat, M.S., Suthir, S., Rajesh, M., Suresh, K.C. (2021). Selection of intermediate routes for secure data communication systems using graph theory application and grey wolf optimization algorithm in MANETs. IET Networks, 10(5): 246–252.

[12] Alam, T., Qamar, S., Dixit, A., Benaida, M. (2021). Genetic algorithm: Reviews, implementations and applications. International Journal of Engineering Pedagogy, 10(6): 57–77. https://doi.org/10.3991/ijep.v10i6.14567

[13] Devi, R.K., Murugaboopathi, G. (2019). An efficient clustering and load balancing of distributed cloud data centers using graph theory. International Journal of Communication Systems, 32(5), e3896. https://doi.org/10.1002/dac.3896

[14] Yue, H., Lin, H., Jin, Y., Zhang, H., Cai, K. (2022). Opening knowledge graph model building of artificial intelligence curriculum. International Journal of Emerging Technologies in Learning, 17(14): 64–77. https://doi.org/10.3991/ijet.v17i14.32613

[15] Wu, Z., Jia, F. (2022). Construction and application of a major-specific knowledge graph based on big data in education. International Journal of Emerging Technologies in Learning, 17(7): 64–79. https://doi.org/10.3991/ijet.v17i07.30405

[16] Huang, Y., Zhu, J. (2021). A personalized English learning material recommendation system based on knowledge graph. International Journal of Emerging Technologies in Learning, 16(11): 160–173. https://doi.org/10.3991/ijet.v16i11.23317

[17] Capuano, N., Caballé, S., Miguel, J. (2016). Improving peer grading reliability with graph mining techniques. International Journal of Emerging Technologies in Learning, 11(7): 24–33. https://doi.org/10.3991/ijet.v11i07.5878

[18] De Figueiredo, C.M.H. (2012). The P versus NP-complete dichotomy of some challenging problems in graph theory. Discrete Applied Mathematics, 160(18): 2681–2693. https://doi.org/10.1016/j.dam.2010.12.014

[19] Guturu, P., Dantu, R. (2008). An impatient evolutionary algorithm with probabilistic tabu search for unified solution of some NP-hard problems in graph and set theory via clique finding. IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics, 38(3): 645–666. https://doi.org/10.1109/TSMCB.2008.915645

[20] Komusiewicz, C., Nichterlein, A., Niedermeier, R., Picker, M. (2019). Exact algorithms for finding well-connected 2-clubs in sparse real-world graphs: Theory and experiments. European Journal of Operational Research, 275(3): 846–864. https://doi.org/10.1016/j.ejor.2018.12.006

[21] Gao, Z., Chen, D., Wu, H.-C. (2020). Graph coloring inspired approximate algorithm for wireless energy redistribution in WSNs. IEEE Transactions on Green Communications and Networking, 4(1): 124–138. https://doi.org/10.1109/TGCN.2019.2947172

[22] Kralev, V., Kraleva, R., Ankov, V., Chakalov, D. (2022). An analysis between exact and approximate algorithms for the k-center problem in graphs. International Journal of Electrical and Computer Engineering, 12(2): 2058–2065. https://doi.org/10.11591/ijece.v12i2.pp2058-2065

[23] Wan, X., Wang, H., Li, J. (2019). LKAQ: Large-scale knowledge graph approximate query algorithm. Information Sciences, 505: 306–324. https://doi.org/10.1016/j.ins.2019.07.087

[24] Karp, R.M. (2010). Reducibility among combinatorial problems. 50 Years of Integer Programming 1958–2008: From the Early Years to the State-of-the-Art, 219–241. https://doi.org/10.1007/978-3-540-68279-0_8

[25] Debnath, L. (2010). A brief historical introduction to Euler's formula for polyhedra, topology, graph theory and networks. International Journal of Mathematical Education in Science and Technology, 41(6): 769–785. https://doi.org/10.1080/00207391003675166

[26] Chartrand, G., Eroh, L., Schultz, M., Zhang, P. (2001). An introduction to analytic graph theory. Utilitas Mathematica, 59: 31–55.

[27] Pemmaraju, S., Skiena, S. (2003). Minimum Vertex Cover. In Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica. Cambridge University Press, 317. https://doi.org/10.1017/CBO9781139164849

[28] Fernau, H., Niedermeier, R. (2001). An Efficient Exact Algorithm for Constraint Bipartite Vertex Cover. Journal of Algorithms, 38(2): 374–410. https://doi.org/10.1006/jagm.2000.1141

[29] Fujito, T., Doi, T. (2004). A 2-approximation NC algorithm for connected vertex cover and tree cover. Information Processing Letters, 90(2): 59–63. https://doi.org/10.1016/j.ipl.2004.01.011

[30] Astrand, M., Floréen, P., Polishchuk, V., Rybicki, J., Suomela, J., Uitto, J. (2009). A local 2-approximation algorithm for the vertex cover problem. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 5805 LNCS, 191–205. https://doi.org/10.1007/978-3-642-04355-0_21

[31] Karakostas, G. (2009). A better approximation ratio for the vertex cover problem. ACM Transactions on Algorithms, 5(4), 41: 1–8. https://doi.org/10.1145/1597036.1597045

[32] Sun, C., Qiu, H., Sun, W., Chen, Q., Su, L., Wang, X., Zhou, Q. (2022). Better approximation for distributed weighted vertex cover via game-theoretic learning. IEEE Transactions on Systems, Man, and Cybernetics: Systems, 52(8): 5308–5319. https://doi.org/10.1109/TSMC.2021.3121695

[33] Razgon, I. (2009). Faster computation of maximum independent set and parameterized vertex cover for graphs with maximum degree 3. Journal of Discrete Algorithms, 7(2): 191–212. https://doi.org/10.1016/j.jda.2008.09.004

[34] Pullan, W. (2009). Optimisation of unweighted/weighted maximum independent sets and minimum vertex covers. Discrete Optimization, 6(2): 214–219. https://doi.org/10.1016/j.disopt.2008.12.001

[35] Klobučar, A., Manger, R. (2020). Independent sets and vertex covers considered within the context of robust optimization. Mathematical Communications, 25(1): 67–86.

[36] Casel, K., Fernau, H., Ghadikoalei, M.K., Monnot, J., Sikora, F. (2019). Extension of vertex cover and independent set in some classes of graphs. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 11485 LNCS, 124–136. https://doi.org/10.1007/978-3-030-17402-6_11

[37] Bourgeois, N., Escoffier, B., Paschos, V.T. (2011). Approximation of max independent set, min vertex cover and related problems by moderately exponential algorithms. Discrete Applied Mathematics, 159(17): 1954–1970. https://doi.org/10.1016/j.dam.2011.07.009

[38] Hossain, A., Lopez, E., Halper, S.M., Cetnar, D.P., Reis, A.C., Strickland, D., Klavins, E., Salis, H.M. (2020). Automated design of thousands of nonrepetitive parts for engineering stable genetic systems. Nature Biotechnology, 38(12): 1466–1475. https://doi.org/10.1038/s41587-020-0584-2

[39] Zhang, W., Tu, J., Wu, L. (2019). A multi-start iterated greedy algorithm for the minimum weight vertex cover P3 problem. Applied Mathematics and Computation, 349: 359–366. https://doi.org/10.1016/j.amc.2018.12.067

[40] Bouamama, S., Blum, C., Boukerram, A. (2012). A population-based iterated greedy algorithm for the minimum weight vertex cover problem. Applied Soft Computing Journal, 12(6): 1632–1639. https://doi.org/10.1016/j.asoc.2012.02.013

[41] Wang, Y., Lü, Z., Punnen, A.P. (2021). A fast and robust heuristic algorithm for the minimum weight vertex cover problem. IEEE Access, 9: 31932–31945. https://doi.org/10.1109/ACCESS.2021.3051741

[42] Zhang, Y., Wu, J., Zhang, L., Zhao, P., Zhou, J., Yin, M. (2018). An efficient heuristic algorithm for solving connected vertex cover problem. Mathematical Problems in Engineering, 2018, 3935804. https://doi.org/10.1155/2018/3935804

[43] Wang, L., Hu, S., Li, M., Zhou, J. (2019). An exact algorithm for minimum vertex cover problem. Mathematics, 7(7): 603. https://doi.org/10.3390/math7070603

[44] Xiao, M., Kou, S. (2017). Exact algorithms for the maximum dissociation set and minimum 3-path vertex cover problems. Theoretical Computer Science, 657: 86–97. https://doi.org/10.1016/j.tcs.2016.04.043

[45] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. (2001). Introduction to Algorithms. The MIT Press. ISBN: 0262032937

[46] Khot, S., Regev, O. (2008). Vertex cover might be hard to approximate to within 2 – ε. Journal of Computer and System Sciences, 74(3): 335–349. https://doi.org/10.1016/j.jcss.2007.06.019

[47] Alenezi, M., Akour, M. (2022). Methodical software testing course in higher education. International Journal of Engineering Pedagogy, 12(1): 51–62. https://doi.org/10.3991/ijep.v12i1.26111

[48] Karnalim, O., Kurniawati, G., Sujadi, S.F., Nathasya, R.A. (2020). Comparing the impact of programming assessment type: In-class vs take-home. International Journal of Engineering Pedagogy, 10(4): 125–132. https://doi.org/10.3991/ijep.v10i4.13509

[49] Atoum, I. (2019). A spiral software engineering model to inspire innovation and creativity of university students. International Journal of Engineering Pedagogy, 9(5): 7–23. https://doi.org/10.3991/ijep.v9i5.10993

[50] Kralev, V., Kraleva, R., Sinyagina, N., Koprinkova-Hristova, P., Bocheva, N. (2018). An analysis of a web service based approach for experimental data sharing. International Journal of Online Engineering, 14(9): 19–34. https://doi.org/10.3991/ijoe.v14i09.8740

[51] Halim, M., Adadi, N., Berrada, M., Tahiri, A., Chenouni, D. (2022). Proposition of web services discovery and composition approach: Application in E-learning platform. International Journal of Emerging Technology and Advanced Engineering, 12(9): 49–62. https://doi.org/10.46338/ijetae0922_06

7 AUTHORS

Velin Kralev is Associate Professor of Computer Science at the South-West University, Blagoevgrad, Bulgaria. He defended his Ph.D. thesis in 2010. His research interests include optimization problems of the graph theory and component-oriented software engineering.

Radoslava Kraleva is Associate Professor of Computer Science at the South-West University, Blagoevgrad, Bulgaria. She defended her Ph.D. thesis in 2014. Her research interests include speech recognition, mobile app development, and computer graphic. She is a reviewer of *International Journal on Advanced Science, Engineering and Information Technology* (e-mail: rady_kraleva@swu.bg).

Viktor Ankov received his M.Sc. degree in Computer Science from the South-West University, Blagoevgrad, Bulgaria in 2021. He is a Ph.D. student of Computer Science at the South-West University, Bulgaria.