Olympiads in Informatics, 2008, Vol. 2, 90–104

Tasks on Graphs

Krassimir MANEV
Department of Mathematics and Informatics, Sofia University and
Institute of Mathematics and Informatics, Bulgarian Academy of Sciences
5, J. Bourchier blvd., 1164 Sofia, Bulgaria
e-mail: manev@fmi.uni-sofia.bg

Abstract. Despite missing of the topic in standard school curricula, tasks on graphs are among the most used in programming contest for secondary school students. The paper starts with fixing the used terminology. Then one possible classification of tasks on graphs is proposed. Instead of the inner logic of Graphs Theory, which is typical for some profiled textbooks, the classification approach of the general textbooks on algorithms was used. The tasks on graphs from last ten editions of the Bulgarian National Olympiad in Informatics were considered and classified in order to check the usability and productivity of the proposed classification.

Key words: graph structures, directed and undirected graphs and multi-graphs, classification of tasks on graphs, algorithmic schemes.

1. "In the beginning ..."[1]

It is a fact that the task proposed in the first International Olympiad in Informatics (IOI), held in Bulgaria in 1989, was a task on a graph. The statement of the task, as given to the students, was published in (Kenderov and Maneva, 1989) and a more simple and contemporary form of the statement of the task was included in (Manev *et al.*, 2007, p. 114).

Two years before the first IOI, an open competition on programming for school students was organized just before and in connection with the Second International Conference and Exhibition "Children in the Information Age" of IFIP, which took place in Sofia, Bulgaria, from May 19 till May 23, 1987. The contestants were divided in three age groups (junior – under 14 years, intermediate – under 16 years, and senior – older than 16 years). It is interesting that the task proposed to students of the senior group was also a task on a graph. The statement of the task, as given to the students, was also published in (Kenderov and Maneva, 1989). We would like to give here a contemporary form of this task too.

***Task 1* (*Programming contest "Children in the Information Age"*, 1987).** Let the bus stops in a city be labeled with the numbers $1, 2, \ldots, N$. Let also all bus routes of the city be given: $M_1 = (i_{1,1}, i_{1,2}, \ldots, i_{1,m1})$, $M_2 = (i_{2,1}, i_{2,2}, \ldots, i_{2,m2}), \ldots$, $M_r =$

[1] The Bible, Genesis 1:1.

$(i_{r,1}, i_{r,2}, \ldots, i_{r,mr})$, $1 \leqslant i_{j,k} \leqslant N$, $i_{j,k} \neq i_{j,l}$ when $k \neq l$. Each bus starts from one end of its route, visits all stops of the route in the given order and, reaching the other end of the route, goes back visiting all stops in reverse order. Write a program that (i) checks whether one can get from any stop to any other stop by bus; (ii) for given stops i and j prints all possible ways of getting from stop i to stop j by bus; (iii) for given stops i and j finds fastest possible way of getting from stop i to stop j by bus, if times for travel from stop to stop are equal and 3 times less than the time to change busses.

During the years after the first IOI tasks on graphs became a traditional topic in the programming contests for school students – international, regional and national. Something more – the number of the tasks on graphs given in these contests is significant.

Why this subject, which was never included in school curricula, is so attractive for programming contests? When and how do we have to start introducing of graph concepts and algorithms on the training of young programmers? Which tasks and algorithms on graphs are appropriate for the students of different age groups, and which concepts have to be introduced in order that students are able to understand and solve corresponding tasks? How should we present the abstract data type graph (and related to it abstract type tree) in data structures? These are only few of the questions that arise in the process of teaching the algorithmics of graphs. In this paper we will try to give answers of a part of them, based on more than 25 years experience of teaching algorithms on graphs, as well as preparing tasks, solutions and test cases.

In Section 2 we will introduce some basic notions, not because the reader is not familiar with them but just to escape misunderstanding, because the different authors use different terminology. In Section 3 one possible classification of tasks on graphs is presented, based on the character of used algorithms. The effectiveness and productivity of proposed classification was checked on a set of tasks on graphs from the Bulgarian olympiads in informatics and the results are presented in Section 4.

2. "... What's in a name? ..."[2]

Speaking about the terminology, we are strongly influenced by the remarkable book (Harary, 1969). As it was mentioned in Chapter 2 of this book, which is dedicated to the terminology, most of the scientists that work in the domain are using their own terminology. Something more, the author supposed that Graph Theory will never have uniform terminology. Back then, we totally accepted the terminology of Frank Harary and believed that uniformity was possible. About 40 years later we have to confess that he was right. Uniform terminology in Graph Theory still does not exist and those used by us, even if strongly influenced by that of Harary, is different from the original.

2.1. *Graph Structures*

Each *graph structure* $G(V, E)$ is defined over a finite set V of *vertices* as a collection of *links* E (see Fig. 1) such that each link is connecting a couple of vertices. The link

[2] W. Shekspeare, *Romeo and Julet*, Act II, Scene 2.

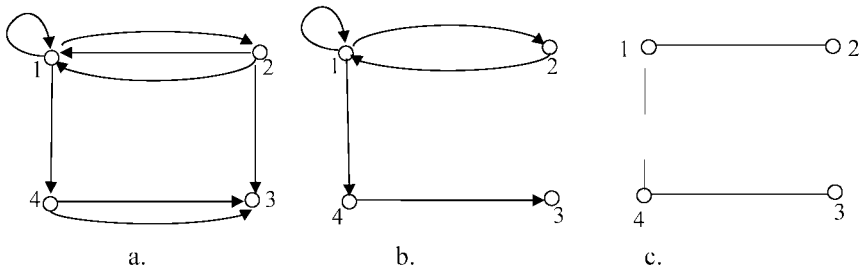


Fig. 1. Directed multi-graph (a), directed graph (b) and undirected graph (c).

is *directed* when the couple is ordered and *undirected* when the couple is unordered (2-elements subset of V). We will call undirected links *edges* and directed links – *arcs*. Unfortunately, most of the authors of books on graph structures denote the edge that links vertices v and w by the ordered couple (v, w) instead of more precise set notation $\{v, w\}$. Such notation sometimes leads to misunderstandings. We will be happy to see the tradition changed, but for the moment it seems impossible.

If E is a set of links (edges or arcs) we are speaking of *graphs*. If a multi-set E of links is considered, i.e., a repetition of elements of E is permitted, than we are speaking of *multi-graphs*. Applying ordering and repetition principles of Combinatorics, four kinds of graph structures are obtained – *directed graphs* (or *digraphs*), *undirected graphs* (simply *graphs*), *directed multi-graphs* (simply *multi-digraphs*) and *undirected multi-graphs* (simply *multi-graphs*). Distinguishing the four kinds of graph structures in the training process is important because sometimes algorithms solving the same task in different structures are different. A trivial example is the procedure of presenting graph structure in a *matrix of incidence* `g[][]`. When the structure is given in the input file as a *list of links* it is enough to assign `g[v][w]=1` for the arc (v, w), but for the edge (v, w) both assignments `g[v][w]=1` and `g[w][v]=1` will be necessary.

Special kind of links (v, v) are called *loops*. Our experience suggests that it makes sense to allow loops in directed graph structures and do not allow them in undirected, which will be our assumption through the paper. But this does not mean that loops should not be included in undirected graph structures at all. It is quite possible that an interesting task idea could presume existing of loops in a graph or multi-graph.

2.2. *Traversals in Graph Structures*

The idea of "moving" in a graph structure, passing from a vertex to another vertex that is linked with the first, is one of the most fundamental in the domain. We will call such moving in a graph a *traversal*. It is worth separating directed traversals from undirected. We will call the sequence $v_0, v_1, \ldots, v_l$ of vertices of a directed graph structure a *course of length l from v_0* to v_l, if there is an arc (v_i, v_{i+1}) in E for $i = 0, 1, \ldots, l-1$. When $v_0 = v_l$ then the course is called a *circuit*. We will call the sequence $v_0, v_1, \ldots, v_l$ of vertices of an undirected graph structure a *path of length l from v_0* to v_l, if there is an arc (v_i, v_{i+1}), for $i = 0, 1, \ldots, l-1$ and $v_{i-1} \neq v_{i+1}$ for $i = 1, 2, \ldots, l-1$. When $v_0 = v_l$

then the path is called a *cycle*. The constrain $v_{i-1} \neq v_{i+1}$ in the undirected case is crucial. Without this constrain it will happen that the undirected graph from Fig. 1(c.) contains the cycle 1,2,1 which is unacceptable. In the digraph from Fig. 1(b.), for example, the same sequence 1,2,1 is a circuit and even the sequence 1,1,1,2,1 is a circuit.

A graph structure in which there is a course, respectively path, from each vertex to each other vertex is called *connected.* Directed graph structures in which for each two vertices there is a course in at least one of the two possible directions are called *weakly connected*. When a graph is not connected, then it is composed of several connected sub-graphs called *connected components*.

Some authors prefer to call courses and paths that not repeat links and/or vertices with specific names (sometime different for both cases – lack of repeated links and lack of repeated vertices only). Others prefer to call them *simple courses* and *simple paths*, respectively (for the circuits and cycles $v_0 = v_l$ is not considered a repetition of vertex). We will use the short names courses and paths for the most frequent case when the repetition of vertices is not allowed (which imply no repetition of links too). For rare cases of repetitions other names, longer and even self-explaining could be used.

Traversals of graph structures that pass trough each link once are called *Euler traversals*. Traversals of undirected graph structures that pass trough each vertex once are called *Hamilton traversals.*

2.3. *Graph Structures with Cost Functions*

On each graph structure it is possible to define *cost function* on vertices c_V: $V \rightarrow C$, *cost function* on links c_E: $E \rightarrow C$, or both, where C is usually some numerical set of possible values (natural numbers, rational numbers, real numbers, etc. or subset of those sets). Values of the cost functions, beside *cost*, are called also *length, weight, priority, etc.* depending on the situation. If a graph structure has no cost function defined then we will presume that the cost of each vertex and link is 1. Cost functions are usually extended in some natural way on sub-graphs and other sub-structures defined in the graph structure. For example the cost of a path in a graph is usually defined as a sum of costs of its vertices, of its edges or both of vertices and edges (if applicable).

The notion *path* (or *course*) *of minimal cost*, called also *shortest path* (or *course*) is fundamental for algorithmics on graphs. Both tasks mentioned in Section 1, the task from IOI'89 and the task from the contest organized in parallel with the International Conference and Exhibition "Children in the Information Age", included searching of shortest path. Let us now reformulate them in terms introduced here.

***Task 2* (*IOI'1989*):** Let V be the set of all strings of length $2N$ composed of $N-1$ letters 'A', $N-1$ letters 'B', and 2 consecutive letters 'O'. Two strings (vertices of V) are linked by an edge if one of the strings could be obtained from the other by swapping letters 'O' and two other consecutive letters, conserving their order. The strings in which all letters 'A' are leftmost of all letters 'B' (does not mater where the letters 'O' are) are called *final*. Write a program that for a given string S finds and prints one path of minimal length (trivial cost of each edge is 1) from S to some final string. If there is no path between S and a final string, the program has to print the corresponding message.

Task 3 (***Programming contest, 1987***): A graph $G(V = \{1, 2, \ldots, n\}, E)$ is given. The set E of edges is defined by r of its paths of length $m_1, m_2, \ldots, m_r$, respectively in such a way that each edge of G is included in at least one of the given paths. The cost of each vertex is 3 and the cost of each edge is 1. Write a program (i) to check whether the graph is connected; (ii) for given two vertices v and w, to generate all paths between v and w; (iii) for given two vertices v and w, to find the path between v and w with minimal cost.

Let $G(V, E)$ be a graph with cost function c_E: $E \to C$, where C is a numeric set with non negative values. Then the function d: $V \times V \to C$, where $d(v, w)$ is the cost of the shortest path from v to w is a *distance* in classic mathematical sense of the word because (i) $\forall v, w \in V$, $d(v, w) \geqslant 0$ and $d(v, w) = 0$ iff $v = w$; (ii) $\forall v, w \in V$, $d(v, w) = d(w, v)$; (iii) $\forall v, w, u \in V$, $d(v, w) \leqslant d(v, u) + d(u, w)$.

Introducing of distance function gives us the possibility to consider the graph $G(V, E)$ as a geometric object and to define the corresponding notions. For example, a *center* of the graph G is each vertex v, which minimize $D(v) = \max\{d(v, w)|w \in V\}$ and the *diameter* of the graph G is $D(G) = \max\{d(v, w)|v, w \in V\}$. The analogy between the "geometry" of a graph and the geometry of well known Euclidean space is an origin of interesting tasks on graphs.

2.4. *Graph Structures and Relations*

Let A and B be arbitrary sets. Each subset R of the Cartesian product $A \times B$ is called a *relation*. School and university curricula in mathematics provides a large amount of useful relations: among numbers ("x is less then y", "x is less then or equal to y", "x is equal to y", etc.), among geometric objects ("the point p lies on the line l", "the line l passes trough the point p", "lines l and m are parallel" etc.), among subsets of some universal set ("A is a subset of B", "A and B intersect", etc.).

A lot of relations we could find outside mathematics, in the world around us. For example, relations among people – "x is a son of y", "x likes y", "x and y are in the same class", etc; or the very popular relation among villages "the village x is linked with the village y by a road" (similar relations could be established among city crossroads linked by streets, railway stations linked by railway roads, electricity sources, distribution centers and customers linked by electricity lines, etc.). That is why many different tasks can arise, in a natural way, in connection with a specific finite relation – abstract (mathematical) or from the real world.

Unfortunately, school curricula (and even some university curricula) use many relations without to consider the notion itself and its properties – especially the properties of relations over Cartesian squares $A \times A$ – *reflexivity, symmetry, anti-symmetry, transitivity*. Some specific relations over Cartesian squares – equivalences (reflexive, symmetric and transitive), partial orders (reflexive, anti-symmetric and transitive) and total orders (reflexive, strongly anti-symmetric and transitive) are significant both for mathematics and algorithmics.

The notion *finite relation* coincides with the notion digraph. Indeed, each digraph $G(V, E)$ could be considered as a relation $E \subseteq V \times V$ and vice versa. A finite relation

$E \subseteq V \times V$ that is symmetric (and optionally reflexive) is really a graph. That is why, each task connected with some relation could be considered as a task on a digraph or graph. Let us consider some examples. It will be helpful to fix the set V to be $\{1, 2, \ldots, n\}$.

***Task 4*:** Let $E \subseteq V \times V$ be equivalence. Find the number of classes of equivalence of E. Is this number equal to 1? If the number of classes is great than 1, find the classes of equivalence of E.

This task (really set of very similar tasks) is classic for relations of equivalence. Because equivalence is reflexive and symmetric relation, $G(V, E)$ is a graph. From the graph point of view this task could be formulated as: "How many connected components has the graph $G(V, E)$? Is the graph connected? If not, then find the vertices of each connected component of G".

***Task 5*:** Let $E \subseteq V \times V$ be total order (we will denote $(x, y) \in E$ with $x \leqslant y$) and $V' \subseteq V$, $|V'| = M$. Find a sequence $a_1, a_2, \ldots, a_M$ of all elements of V' such that $a_1 \leqslant a_2 \leqslant \ldots \leqslant a_M$.

Of course, this is the task for *sorting* a subset of elements of a given total order. It is so popular that a specific branch of the Theory of Algorithms is dedicated to it. Anyway, the task could be formulated as a task on digraph. It is well known that relations of ordering, considered as digraphs, have no circuits. So the task for sorting a given subset of a totally ordered set will look like: "Given a digraph $G(V', E')$ without circuits. Find a course with a maximal length in G". Relation E' in this formulation is, obviously, the *restriction* of E on V'. Digraphs without circuits are very popular and have specific name – *dag* (abbreviation of **d**irected **a**cyclic **g**raphs, because some authors use the notion cycle for digraphs too).

***Task 6*:** Let $E \subseteq V \times V$ be a partial order which is not total (we will denote $(x, y) \in E$ with $x \leqslant y$ again). Find a sequence $a_1, a_2, \ldots, a_M$ of elements of V with maximal length such that $a_1 \leqslant a_2 \leqslant \ldots \leqslant a_M$.

Formulation of this task as a task in digraph will be: "Given a dag $G(V, E)$. Find a course with a maximal length in G".

The examples given above concerned relations over the Cartesian square. But relations over the Cartesian product of two different domains could also be considered in graph formulation.

***Task 7*:** Let $R_1 \subseteq A \times B$ and $R_2 \subseteq B \times A$ are such that $(a, b) \in R_1$ if and only if $(b, a) \in R_2$. Find a subset $\{(a_1, b_1), (a_2, b_2), \ldots, (a_M, b_M)\}$ of R_1 (or $\{(b_1, a_1), (b_2, a_2), \ldots, (b_M, a_M)\}$ of R_2, which is the same) with maximal numbers of elements such that $a_i \neq a_j$ and $b_i \neq b_j$, $1 \leqslant i < j \leqslant M$.

Relations R_1 and R_2 with mentioned above property are called *mutually reversed*. Examples of mutually reversed relations are the above mentioned couple "the point p lies on the line l" and "the line l passes through the point p". An example from real life could be the couple "the person p could do the work w" and "the work w could be done by the person p". For each couple of mutually reverse relations we can build a graph $G(V = A \cup B, R_1)$ (or $G(V = A \cup B, R_2)$, which is the same) considering elements of R_1 (R_2, respectively) as not ordered. Such graph is called *bipartite*. Searched subset M of edges such that each vertex is an end of at most one edge in M is called *matching*.

In graph formulation the task will be: "Given a bipartite graph $G(V = A \cup B, R_1)$. Find one maximal matching of G".

2.5. *Trees and Rooted Trees*

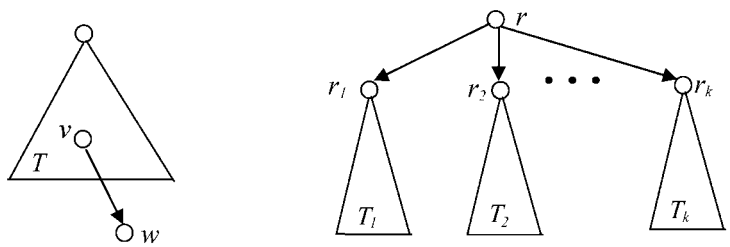
Discussion of tasks on graph structures is impossible without introducing the notion *tree*. By the classic definition, graph $T(V, E)$ is a tree if it is connected and has no cycles. For the purposes of algorithmics the notion *rooted tree* is more helpful. Two equivalent inductive definitions of rooted tree are given below.

DEFINITION 1. (i) The graph $T(\{r\}, \varnothing)$ is a rooted tree. r is a *root* and a *leaf* of T; (ii) Let $T(V, E)$ be a rooted tree with root r and leaves $L = \{v_1, v_2, \ldots, v_k\}$. Let $v \in V$ and $w \notin V$; (iii) Then $T'(V' = V \cup \{w\}, E' = E \cup \{(v, w)\})$ is also a rooted tree. r is a *root* of T' and leaves of T' are $(L - \{v\}) \cup \{w\}$. This definition (Fig. 2(a.)) is more appropriate for building of rooted trees.

DEFINITION 2. (i) The graph $T(\{r\}, \varnothing)$ is a rooted tree. r is a *root* and a *leaf* of T; (ii) Let $T_1(V_1, E_1), T_2(V_2, E_2), \ldots, T_k(V_k, E_k)$, be rooted trees with roots $r_1, r_2, \ldots, r_k$, and leaves $L_1, L_2, \ldots, L_k$, respectively. Let $r \notin V_1 \cup V_2 \cup \ldots \cup V_k$; (iii) Then $T'(V' = V_1 \cup V_2 \cup \ldots \cup V_k \cup \{r\}, E' = E_1 \cup E_2 \cup \ldots \cup E_k \cup \{(r, r_1), (r, r_2), \ldots, (r, r_k)\})$ is also a rooted tree. r is a *root* of T' and leaves of T' are $L_1 \cup L_2 \cup \ldots \cup L_k$. Rooted trees $T_1, T_2, \ldots, T_k$ are called *subtrees* of T'. This definition (Fig. 2(b.)) is more appropriate for analyzing rooted trees. Introducing the notion sub-tree it is leading to natural recursive procedures.

By definition rooted trees are undirected graphs. Anyway, Definition 1 is introducing an *implicit direction* on the edges of the rooted tree. We could say that v is a *parent* of w and that w is a *child* of v. Obviously each rooted tree is a tree and each tree could be rebuild as rooted when we choose one of the vertices for its root.

If $G(V, E)$ is a graph and $T(V, E')$ is a (rooted) tree such that $E' \subseteq E$ than T is called a *spanning* (*rooted*) *tree* of G. Graph G is connected if and only if it has a spanning tree. So, the most natural way to check whether the graph G is connected is to try to build a



a. b.

Fig. 2. Two equivalent definitions of rooted tree.

spanning tree of G. If c: $E \to C$ is a cost function on edges of $G(V, E)$ we could extend it to spanning trees of G, defining $c(T(V, E')) = \sum_{e \notin E'} c(e)$. Each spanning tree T of G with minimal (maximal) $c(T)$ is called *minimal* (*maximal*) *spanning tree* of G.

2.6. *Presentation of Graph Structures and Trees*

As in each other domain, presenting abstract data types *graph*, *digraph*, *multi-graph*, *multi-digraph* and *tree* in data structures is crucial for creating of efficient algorithms. Traditionally, the graph structures are given in input files, as it was mentioned above, in form of a *list of links* preceded by the number n of its vertices and the number m of its links and an instruction that links have to be interpreted as undirected (edges) or directed (arcs).

When the essential part of the algorithm that will be implemented on a graph structure $G(V, E)$ is an iterative instruction of form

`for e` $\in$ `G do` $\{\ldots\}$

then the list of links is a perfect presentation and the implementation will have time complexity $\mathrm{O}(m)$. If the same algorithm is implemented over a presentation of G with an *adjacency matrix* (i.e., 2-dimentional array `g` such that `g[v][w]` is the number of the links between v and w) than the time complexity will be $\mathrm{O}(n^2)$ and the implementation will be slower for graph structures with a relatively small number of links.

If the essential part of the algorithm is an iterative instruction of the form

`for` $v \in V' \subseteq V$ `do{for` $w \in V'' \subseteq V$ `do{...if` $(v, w) \in E\{\ldots\}\}\}$

then the implementation with list of links will be of complexity $\mathrm{O}(|V'||V''|.m)$ and with an adjacency matrix – of complexity $\mathrm{O}(|V'||V''|)$, which is much better.

If the essential part of the algorithm is an iterative instruction of the form

`for` $v \in V$ `do{for` w such that $(v, w) \in E$ `do`$\{\ldots\}\}$

then the implementation with adjacency matrix will be of complexity $\mathrm{O}(n^2)$. In such case it will be more appropriate to use another presentation of the graph structures – *lists of neighbors* (in undirected case) or *lists of children* (in directed case) which will give us an implementation of complexity $\mathrm{O}(m)$.

Especially for rooted trees, we would like to mention the presentation *list of parents* – an array `g[]` such that `g[i]` is the parent of `i` for each vertex that is not the root `r` and `g[r]=0`. This presentation of rooted trees is very convenient when it is necessary to build a rooted tree (spanning, minimal spanning, etc.) or to maintain it.

It is worth also mentioning that different specific tree structures (heaps, index trees, suffix trees, etc.) are an inevitable part of efficient implementation of many algorithms but discussing of such specific tree structures is far beyond the scope of this paper.

3. Classification of Tasks on Graphs

Classification of tasks on graphs is important due to different reasons. Good classification could be very helpful for preparing curricula and organizing the training process – for deciding which classes of tasks are to be taught and in which order, just to have a smooth

passing from more easy to more difficult tasks. Classification could be very helpful for preparing contests too – for avoiding tasks of a same class in one contest or similar tasks in two consecutive contests. In this chapter we will consider first some classifications of tasks on graphs and then will discuss the place of tasks on graphs among other classes of tasks used in programming contests.

3.1. *Classifications of the Profiled Textbooks*

Profiled textbooks – see, for example, (Christophides, 1975) and (Gondran and Minoux, 1984) – prefer to classify tasks on graphs following the inner, graph-theoretical, logic of the book. Our preferable way is another but it is worth discussing briefly these classifications.

In (Christophides, 1975) the classification is based totally on the theory. The following main classes of tasks are considered:

Connectivity and accessibility; Independent and dominating sets of vertices; Colorings; Centers (radii, diameters); Medians; Spanning trees; Shortest paths; Euler traversals; Hamilton traversals and traveling salesman problem; Flows; Matchings.

The approach of (Gondran and Minoux, 1984) is a bit different. The textbook first separates the following classes of tasks, for which good (polynomial) algorithms exist:

Connectivity; Shortest paths; Spanning trees; Flows; Matchings; Euler traversals.

Then the authors consider a group of tasks, for which polynomial algorithms still do not exist. Some algorithmic concepts which are useful for approaching algorithmically hard tasks are also considered – greedy, backtracking (branch and bound), dynamic programming, etc.

Classifications on the base of the inner, graph-theoretical, logic have their reasons. But they are not convenient for the education and training of young programmers. Such an approach can sometimes hide important common features of significantly different (from graph-theoretical point of view) tasks. For example, both sources referred to above consider as different topics *connectivity, spanning trees* and *shortest paths.* But the simplest way of checking the connectivity of an undirected graph structure is to try to build a spanning tree of this graph. From the other side, a spanning tree of a graph with root r, built *in breadth*, is a *tree of shortest paths* from r to each other vertex of the graph.

Without underestimating theoretical classifications of the tasks in graphs, we prefer the "algorithmic" classifications – such that collects in one class tasks, solvable by similar algorithms or, more general, by same *algorithmic scheme*. As we will see it is possible that some class of tasks could be a result of both classification approaches. This will happen when tasks of some, "theoretically" identified, class are solvable with a specific algorithmic scheme, not applicable at all or inefficient for another kind of tasks – *Euler traversals*, for example.

3.2. *Classifications of the Textbooks in Algorithms*

Classification of tasks on the basis of the used algorithms is an approach that is typical for the general textbooks on algorithms. These books are not aimed at considering the

tasks of a specific mathematical domain but to introduce the general principles and approaches of design (and analysis, of course) of algorithms. That is why these textbooks are trying, usually, to identify as much as possible tasks that are solvable by the algorithm (or algorithmic scheme) in consideration.

As a base of our attempt for classification we used the leader among the textbooks in algorithms (Cormen *et al.*, 1990) and compared it with some other popular textbooks – (Reingold *et al.*, 1977; Aho *et al.*, 1978; Sedgewick, 1990; Brassard and Bratley, 1996), as well as the most popular in Bulgaria (Наков и Добриков, 2003). It is obvious that the chapter on graphs of (Reingold *et al.*, 1977) is organized in a similar way as the profiled books on graph algorithms, so we will not consider it.

The part dedicated to graphs of (Cormen *et al.*, 1990) starts with the chapter "Elementary Graph Algorithms", which discusses the *traversals* of the vertices of graph structures called *Breadth-First search* and *Depth-First search* (BFS and DFS). Both approaches are applicable for solving different tasks related to connectivity of graph structures and the accessibility of a vertex from another vertex. But these two algorithmic schemes are used for solving some specific tasks also. BFS, for example, is building a *tree of shortest paths* in graphs without cost function on the edges and DFS is a basic step for efficient *topological sorting*, finding of *strongly connected components*, *articulation points* and *edges*, etc. All other mentioned above books consider both BFS and DFS. That is why BFS and DFS will be different classes in our classification.

The second chapter of (Cormen *et al.*, 1990) is dedicated to algorithms for finding minimum spanning tree of graphs (MSP). This topic is included in all of considered textbooks. So, *Minimum spanning tree* will be a class in our classification too. It is worth to mention that (Brassard and Bratley, 1996) discuss algorithms for MST in the chapter on greedy algorithms in a special section dedicated to applying greedy scheme on graphs (we will discus this fact later).

The next two chapters of (Cormen *et al.*, 1990) are "Single-Source shortest paths" and "All-Pairs Shortest Paths". We will suppose that the splitting of the topic *Shortest paths* in two is made by the authors just to limit the size of the chapters. No other among the considered textbooks makes the distinction. And let us append two remarks. First, the word "shortest" has to be considered in a larger sense – tasks for finding the *largest path, more reliable path, etc.*, are solvable with the same approach – *the relaxation approach.* And second, the tasks for finding the center(s), the median(s), the radius (or diameter), etc., of graphs (considered in depth only in (Наков и Добриков, 2003) are also solvable by the relaxation approach.

The last chapter of (Cormen *et al.*, 1990) is "Maximum Flow". Beside some basic algorithms for finding *Maximum flow in a network*, the chapter also considers the strongly related but specific task for finding *Maximum matching in bipartite graphs*. (Brassard and Bratley, 1996) does not consider these two topics at all and the other textbooks consider them separately.

Something that is missing in (Cormen *et al.*, 1990) but is included in all other textbooks is the *Exhaustive search* – the general way for solving a huge amount on *NP*-complete problems in Graph Theory. The tasks for finding *Hamilton traversal* of graph

and closely related *Traveling salesman* problem are the most used examples for this class of tasks. Speaking of exhaustive search in graphs we usually have in mind *backtracking* traversals. But in this class could be included any task, for the solving of which it will be necessary to generate all permutation of the vertices, all subsets of vertices (or edges), etc.

Only (Наков и Добриков, 2003) includes a chapter dedicated to such specific topic as *Euler traversal* of multi-graphs and related problems. None of the textbooks consider the topic *Games in graphs* (of type Nim and similar). From graph-theoretical point of view these tasks could be classified in the topic *Independent and Dominating Subsets*. There is a specific approach for solving these tasks (functions of Sprague-Grundy and splitting a game in sum of more simple games). We will include such specific topics in our classification too.

3.3. *Tasks on Graphs in the General Classification of Tasks*

As it was mentioned above, general textbooks on algorithms have always a chapter (or few chapters) dedicated to tasks on graphs (and corresponding algorithms). Anyway, some authors are inclined to put some graph tasks in other chapters of their books. One example, which was mentioned above, was classifying MST task in (Brassard and Bratley, 1996). The book is considering the *algorithm of Prim* and the *algorithm of Kruskal* for finding MST as *greedy*.

Such classification has a very serious reason. In the *cyclic matroid* [3] of a graph the spanning trees, and only they, are maximal independent sets. As it is well known from the theory, greedy algorithms that search a maximal independent set of a matroid with some optimal property always find the optimum. Following such logic, the algorithm of Dijkstra for the task Single-source shortest path is also greedy. Its goal is also a maximal independent set of the cyclic matroid of the graph (i.e., rooted tree) with additional optimum property – to be a tree of the shortest paths from the source.

Let us mention some other examples. In (Келеведжиев, 2001), which is a short introduction to Dynamic Programming (DP), the *algorithm of Dijkstra* for finding the *shortest path* was considered as an example for applying the DP approach and there is a reason for this too. Dijkstra's algorithm is maintaining a table of vertices for which the shortest path from the origin is found (i.e., of sub-tasks solved to the moment). Solving the task for the remaining vertices is reduced (by relaxation steps) to already solved subtasks.

As another example let us consider the exceptional book (Кирюхин и Окулов, 2007), which collects statements and solutions of tasks from the first eighteen IOI. The above mentioned task from the first IOI, that by our classification is a typical BFS task, is classified in the book as a task on a sequence. With the same success authors could classify it as *Exhaustive search*. In the way, as exhaustive search authors classified, for example, both the task for first day of IOI'1991 (*Hamilton path*) and the task "Camelot" from second day of IOI'1998, which we prefer to classify as *Breadth-first search*. Different classifications of tasks on graphs give us different points of view to the ways the task could be solved.

[3] For short introduction to Theory of Matroids see, for example, (Welsh, 1976).

3.4. *Graphs in the Proposed IOI Syllabus*

In (Verhoeff *et al.*, 2006) a proposal for a Syllabus of IOI was published. It is interesting to see the place of the specified above topics in the Syllabus. Briefly, the authors explicitly suggest excluding from the topics of IOI *matching in bipartite graphs* and *flows in networks*. And more, the Syllabus does not mention at all (and so exclude implicitly) *games of type Nim* (and related) in graphs. All other topics are covered in one or another form.

We would not like to guess the reasons of authors to exclude (explicitly or implicitly) these specific topics – general reasons for excluding topics from the Syllabus are given in the mentioned paper. We would like only to stress that tasks from excluded classes are proposed in national olympiads and could appear in task sets of future IOIs too, because the Syllabus of IOI has to be instructive, rather then restrictive. Discussion of this topic and especially attempts to exclude some topics from the Syllabus of IOI is going beyond the scope of this paper but put in front of the community very serious question: what kind of mathematical knowledge we have to give to the new generation of mathematicians – the computer programmers?

4. Tasks on Graphs in the Bulgarian Programming Contests

To conclude this paper we would like to consider the place of tasks on graphs in Bulgarian programming contests. For this purpose we checked large amount of tasks from all Bulgarian programming contests from the last 10 years published in (Infoman, 2008). A set of 85 tasks was identified and each task was classified in one of the classes that we specified in previous section. Results are given in Table 1. This will help us to realize which classes of tasks are most used in programming contests.

The tasks of each class were additionally classified by the age group for which they were proposed. This will help us to realize when the tasks of a specific class appear for the first time in competitions and which the preferred tasks for each age group are. The definitions of age groups in Bulgarian programming contest changed over the years (see (Manev *et al.*, 2007)) so we are using the following average definition: group C – 14–15 years; group B – 16–17 years; group A – 18–19 years. When a task was given during a contest for selection of Bulgarian national team, it was classified in group A.

From Table 1 it is obvious that the most preferable class of tasks in Bulgarian programming contests is *Shortest path* for graphs with cost function on edges or on edges and vertices. From the two cases – single-source and all-sources – the first dominates (19 versus 6 tasks). Tasks with classical formulation (solvable with algorithm of Dijkstra or algorithm of Floyd-Warshal) are very few – the two tasks for group C and two of the tasks for group B.

The usual way to escape classical formulation is to define the graph implicitly or to put some additional obstacles (or optimization criteria) on vertices and/or links. Sometime the shortest path task is combined with some task from different domain. As an example of such combination let us mention the following task.

Table 1

Tasks on graph from the Bulgarian programming contests for last 10 years

Class of tasks	Number of tasks	Age group		
		C	B	A
Breadth first search and related (including connectivity checking, identifying connected components, shortest path in graphs without cost function, etc.)	15	6	3	6
Depth first search and related (including topological sorting + some optimization, strongly connected components, etc.). **Remark.** Tasks solvable both by BFS and DFS are classified in the previous group.	15	2	4	9
Euler traversals	3	1		2
Minimum spanning tree	2			2
Shortest path (both *single-source* and *all-sources)* and related	25	2	7	16
Matching and flows in networks	6		2	4
Games in graph *(of type Nim)*	2		1	1
Exhaustive search	15	1	2	12
Difficult to classify	2			2

Task 8 (***Winter tournament 2000, group A***) (Infoman, 2008). Vertices V of a graph are points of the Euclidean plane and edges are line segments linking some of the vertices. Let $C \subseteq V$ are the vertices of the graph from the convex hull of V. For each vertex v of V find the closest to v vertex of C.

It is not unexpected that the second place is shared by the BFS and DFS. Because tasks solvable both by BFS and DFS (i.e., connectivity, accessibility, etc.) are classified only in BFS, it is possible to say that, in Bulgarian national contests, the couple *BFS&DFS* is even more popular than *Shortest path*. Something more, exploring a graph in depth is, or may be, the first task on graphs that young programmers have to solve. This is easy to explain – with a recursive implementation of DFS we could escape introducing the abstract data type *stack*. Objectively the BFS had to be easier to understand in age 14–15 but it will need introduction of the abstract data type *queue*.

Tasks for BFS and DFS in which the graphs are explicitly given are very rare. Usually the graph is extracted from *mazes of squares*, *spaces of situations* with an operation for transforming one situation in another (like the task from the first IOI), some *relation* (for example, the interval $[a, b]$ is included in the interval $[c, d]$), etc. The most frequently used task that is solvable by DFS is *longest course* in a dag.

The fourth most popular category in Bulgarian national programming contests is *Exhaustive search.* It is obvious that in the process of creating of tasks it is impossible to escape the numerous NP-complete tasks of Graph Theory. Especially because there are many situations from the real life, which are modeled as NP-complete tasks (traveling

salesmen, splitting group of people in cliques, etc.). It seems normal that the topics recommended for excluding in the Syllabus of IOI (matching, flows and games of type Nim) are rare. But it is strange that the tasks of the categories *Euler traversals* and *Minimal spanning tree* are very rare. We have not reasonable explanation of this fact.

We did not succeed to classify only 2 of considered tasks. One of them – *Lowest common ancestor* in a rooted tree is a popular tasks, but a specific approach for solving it is necessary. We would like to present here the second of unclassified tasks.

Task 9 (Autumn tournament, 2005) (Infoman, 2008)**.** A set V of vertices and the positive integers $d(v, w)$ for each $v,\ w \in V,\ v \neq w$ are given. Find a graph $G(V, E)$ with minimal number of edges and a positive integer length of each edge in such way that the shortest path for each couple of vertices $v, w \in V$ is equal to the given $d(v, w)$.

5. Conclusions

Graph structures are **an important origin of tasks** for olympiads in informatics. They are modeling real life situations and so the tasks become more natural and easy for understanding. Graphs are not included in classical mathematical school curriculum but most of the notions and concepts are understandable by relatively young students. As it was mentioned, tasks on graphs appear in Bulgarian national contests for student aged 14–15 years. So teaching of graph concepts and algorithms really starts at the age of 12–13 years.

The classification of tasks on graphs, proposed in this paper, is **one of many possible**. It could be discussed and ameliorated. It is possible a classification of tasks on graphs to be based on another principles. But some **classification of tasks on graphs is necessary** for each team of teachers that is coaching contestants in programming. On the base of the proposed classification we could conclude that in the "Bulgarian model" of teaching graphs concepts and algorithms we are starting with BFS and DFS on age 14–15 years. At the age of 16–17 years shortest path tasks are included in the training process. At the age of 18–19, beside algorithmically hard tasks, solvable by different kind of exhaustive search, some specific topics, like Matching in bipartite graphs, Flows in networks and Games of type Nim, also appear in the sets of contests' tasks.

Classification of tasks together with analysis of results during the contests could help us to better organize both the training process and the contests (national and local) – to identify kinds of tasks that are not appropriate for some age group, to identify kinds of tasks that are included in the tasks sets more or less frequently than usual, etc.

We would like to thanks numerous authors of task on graphs for Bulgarian programming contests as well as all Bulgarian contestants for their work, which made this research possible.

References

Aho, A., Hopcroft, J. and Ulman J. (1978). *Data Strucures and Algorithms*. Addison-Wesley.
Brassard, G. and Bratley, P. (1996). *Fundamentals of Algorithmics*. Prentice Hall.

Christophides, N. (1975). *Graph Theory. An Algorithmic Approach.* Academic Pres.
Cormen, T.H., Leiserson, Ch.E. and Rivest R. L. (1990). *Introduction to Algorithms*, Second Edition. The MIT Press.
Gondran, M. and Minoux, M. (1984) *Graphs and Algorithms*. John Wiley & Sons.
Harary, F. (1969). *Graph Theory*. Addison-Wesley Publishing Company.
Infoman (2008). Bulgarian portal for competitive programming. `http://infoman.musala.com` (visited in February2008)
Kenderov, P. and Maneva, N. (Eds.) (1989). *Inernatonal Olympiad in Informatics*. Sofia.
Manev, K., Kelevedjiev, E. and Kapralov, S. (2007). Programming contests for school students in Bulgaria. *Olympiads in Informatics*, **1**, 112–123.
Келеведжиев, Е. (2001). *Динамично програмиране.* Анубис, София.
Кирюхин, В.М. и Окулов, С.М. (2007). *Методика решения задач по информатике.* Международные олимпиады. БИНОМ, Москва.
Наков, Пр. и Добриков, П. (2003). *Програмиране = + + Алгоритми.* Top Team Co, София.
Reingold, E.M., Nivergelt, J. and Deo, N. (1977). *Combinatorial Algorithms. Theory and Practice*. Prentice Hall.
Sedgewick, R. (1990). *Algorithms in C*. Addison-Wesley.
Verhoeff, T., Horváth, G., Diks, K. and Cormack, G. (2006). A Proposal for an IOI Syllabus. *Teaching Mathematics and Computer Science*, **VI**(I), 193–216.
Welsh, D.J.A. (1976). *Matroid Theory*. Academic Press.

K. Manev is an associate professor in discrete mathematics and algorithms at Sofa University, Bulgaria. He is a member of the Bulgarian National Commission for Olympiads in Informatics since 1982 and was a president of the commission (1998–2002). He was a member of the organizing team of first (1989) and second (1990) IOI, president of the SC of Balkan OI'1995 and 2004, leader of the Bulgarian national team for IOI'1998, 1999, 2000 and 2005 and BOI'1994, 1996, 1997, 1999 and 2000. In 2007 he was a leader of Bulgarian team for First Junior Balkan OI. From 2000 to 2003 he was an elected member of IC of IOI. In 2005 he was included again in IC of IOI as a representative person of the host country of IOI'2009. He is author of more than 30 scientific papers, 1 university textbook and 9 textbooks for secondary schools.

Surprisingly, machine learning tasks are defined much differently on graphs and we can categorize it into 4 types: node classification, link prediction, learning over the whole graph, and community detection. In this article, we look closer at how theyâ€™re defined and understand why theyâ€™re so much different from standard machine learning tasks. Node Classification. Letâ€™s imagine that we have a network of friendships between animals . 3 Basic Pretext Tasks on Graphs. 4 Preliminary Analysis. 5 Advanced Pretext Tasks on Graphs. 6 Experiments. 7 Related Work. 8 Conclusion. Self-supervised Learning on Graphs: Deep Insights and New Directions. arXiv:2006.10141v1 [cs.LG] 17 Jun 2020. Wei Jin Michigan State University. Tasks on Graphs 95. E âŠ† V Ã—V that is symmetric (and optionally reï¬‚exive) is really a graph. That is why,each. task connected with some relation could be considered as a task on a digraph or graph. Let us consider some examples.Â tasks on graphs and then will discuss the place of tasks on graphs among other classes of. tasks used in programming contests. 3.1. Classiï¬cations of the Proï¬led Textbooks. Proï¬led textbooks â€“ see, for example, (Christophides, 1975) and (Gondran and Minoux Completing tasks on a graph. Ask Question. Asked today.Â The task at vertex v takes $q_v > 0$ minutes for $v \neq 0$, and we must complete all tasks. We start at Vertex 0, and we have $q_0 = 0$. It will obviously take us exactly $\sum_{i = 1}^{n} q_i$ total time for us to complete all of the tasks on our own (it doesn't take any time to traverse the edges). However, suppose we have a single helper.