Aplikace matematiky

Karel Čulík

Combinatorial problems in the theory of complexity of algorithmic nets without cycles for simple computers

Aplikace matematiky, Vol. 16 (1971), No. 3, 188–202

Persistent URL: http://dml.cz/dmlcz/103345

COMBINATORIAL PROBLEMS IN THE THEORY OF COMPLEXITY OF ALGORITHMIC NETS WITHOUT CYCLES FOR SIMPLE COMPUTERS[1])

KAREL ČULÍK

(Received June 10, 1970)

Algorithmic nets without cycles are a generalization of logical nets without cycles, i.e. they are finite, oriented and acyclic graphs or multigraphs with labelled vertices. Certain total orderings of their vertices are called their *courses*. By each course a *graph of certain intervals* is determined and one of its *chromatic decompositions* is chosen. The following *measures of complexity* of courses with decomposition are introduced: 1) *the length of the course*, i.e. the number of vertices of the considered net, 2) *the width of the course*, i.e. the maximal degree of a complete subgraph in the graph of intervals, 3) *the capacity of storage of the course*, i.e. the number of elements of the chosen decomposition, 4) *the non-efficiency of scopes of the course*, i.e. the sum of differences of the lengths of intervals and the output degrees of the corresponding vertices and finally, 5) *the non-efficiency of addresses*, i.e. the difference of the product of the capacity with the length and the sum of lengths of all intervals. The given problems are extremal and concern either the determination of the minimum of any mentioned measure of complexity for all courses of the given net or a decision about the compatibility and the dependencies of the mentioned measures. At the beginning, the motivations of measuring the complexity of *simple programs* for *simple computers* are given.

1. MOTIVATION OF PROBLEMS IN THE THEORY OF PROGRAMMING LANGUAGES

A *simple computer* (see [1]) ***Mach*** is determined by a set of *objects* ***Obj*** the computer is dealing with, and by a set of *basic functions* ***Fct*** the domain and range of which is ***Obj***, both representing its operational unit, and further by a set of *addresses*

[1]) This paper was presented at the Summer school on Number Theory and on Graph Theory in Modra-Piesok which was organized by Association of Slovak Mathematicians in May, 25—29, 1970.

Adr at which the objects are stored (or by which the objects are denoted) and by a set of *operational symbols* ***Opr***, which denote in a one-to-one correspondence the particular functions of ***Fct***. E.g. ***Obj*** is a set of rational numbers, ***Fct*** are four arithmetic operations defined for rational numbers and denoted, as usually, by the corresponding operational symbols of ***Opr*** $= \{+, ., -, /\}$ and ***Adr*** is the set of all small Roman letters.

In this example of a simple computer ***Mach***, the *commands* are just simple assignement statements, i.e. the strings arisen from the following scheme $X * Y = : Z$ by the substitution of "X", "Y" and "Z" by particular addresses from ***Adr*** and by the substitution of "$*$" by an operational symbol from ***Opr***. A *simple program* for ***Mach*** is a finite sequence of commands satisfying the following condition: if an address occurs on the right-hand side of two commands of the program, then it occurs at least once on the left-hand side of a command being between both considered commands (the addresses x and y occur on the left-hand side of the command $x + y =: z$ and the address z on its right-hand side).

A function expressed by the arithmetical expression $(a - b)/((a + b) . c + ab)$ can be computed or evaluated in our computer ***Mach*** by many different programs which differ in the addresses used to store the mediate results and in the order of the commands, i.e. in the order of evaluation of the partial expressions, e.g. first the numerator and then the denominator of the given expression can be evaluated or, on the contrary, first the denominator and then the numerator can be evaluated. In the following Figures 1 and 2 these two programs computing the given expression are shown.

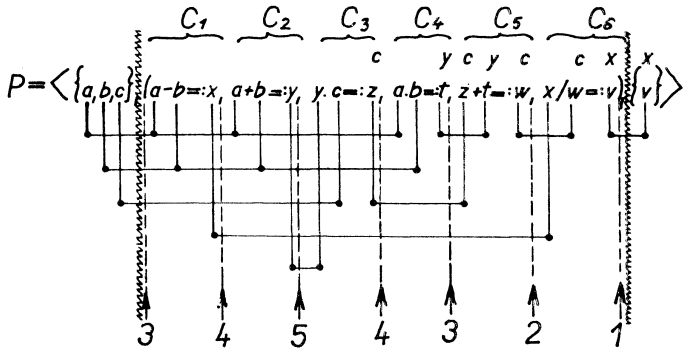

Fig. 1

The program P consists of 6 commands, because there exist 6 different occurrences of the operational symbols in the given expression, and therefore the length of P is 6. The addresses the first occurrence of which (from the left to the right) in P is on the left-hand side of a command are called *input addresses*. Thus a, b, c are the input addresses of P and they are given separately before the program. There are 3 input

addresses because the computed function is a function of 3 variables. Similarly, the address the last occurrence of which in P is on the right-hand side of a command is called the *output address* and it is given after the program. Thus v is the unique output address of P, because just one function is computed.

The *scope* S_p *of an input address* p is the shortest interval in P containing the first command and as the last one that command on the left-hand side of which the input address occurs for the last time. Thus $S_a = (C_1, C_2, C_3, C_4)$, $S_b = (C_1, C_2, C_3, C_4)$ and $S_c = (C_1, C_2, C_3)$ are the input scopes of P.

The *scope* S_i *of the command* C_i in P is the interval $(C_{i+1}, \ldots, C_j)$ such that the address occurring on the right-hand side of C_i occurs on the left-hand side of C_j and if it occurs in C_k where $i + 1 \leqq k \leqq j$ then it occurs there on its left-hand side and finally j is as large as possible as far as the considered address on the right-hand side of C_i is not an output address; and in the opposite case, C_j is the last command of the whole program, i.e. $j = n$. Thus $S_1 = (C_2, C_3, C_4, C_5, C_6)$, $S_2 = (C_3)$, $S_3 = (C_4, C_5)$, $S_4 = (C_5)$, $S_5 = (C_6)$ and $S_6 = \emptyset$ are the command scopes of P.

In Fig. 1 the scopes as intervals in P are represented by lines with dots which correspond to the particular occurrences of addresses in P. The first (from the left to the right) dot in an input scope corresponds to the distinguished occurrence of an input address not belonging to any command and in a command scope S_i this dot corresponds to the address occurring on the right-hand side of C_i, which is called the *address of this scope*. The maximal number of parallel lines = scopes in Fig. 1 is the *width of* P. In Fig. 1 the width is 5.

The number of the used addresses in P is 9 and simultaneously this is also the *capacity of the storage* of P.

If r_p or r_i is the number of dots on the line corresponding to the input scope S_p or to the command scope S_i respectively, diminished by 1, then $S(P) - R(P) = 8$ is the *unefficiency of scopes in* P where $S(P) = |S_a| + |S_b| + |S_c| + |S_1| + \ldots + |S_6| = 21$ and $R(P) = r_a + r_b + r_c + r_1 + \ldots + r_6 = 13$, and

(*the length of* P) . (*the capacity of storage of* P) $- R(P) = 6 \cdot 9 - 13 = 41$ is the *unefficiency of addresses* in P.

The addresses corresponding to the scopes satisfy the condition that two intersecting scopes do not have the same corresponding address. With respect to this condition some addresses can be changed as shown in Fig. 1 where over some of the original addresses the new ones are written. Then the new program P' has the capacity of storage 5 which is equal to its width which is the same as in P. Obviously also the length of P' and P is the same.

The unefficiency of scopes in P' is again the same as in P but the unefficiency of addresses is smaller than in P because it equals to $6 \cdot 5 - 13 = 17$.

In Fig. 2 we present the second program Q computing the above given arithmetic expression which is treated in the same way as P in Fig. 1.

The length and the capacity of storage of Q are the same as those of P but its

width is 4, i.e. less than the width of P. The unefficiency of scopes in Q is $S(Q) - R(Q) = 19 - 13 = 6$ and the unefficiency of addresses in Q is $6 \cdot 9 - 13 = 41$.

Again the program Q may be changed by a certain *readdressing* as shown directly in Fig. 2 where the new addresses are given over some of the original ones. This new modified program Q' has its capacity of storage equal to 4, thus again it is the same number as its width, and its unefficiency of addresses is $4 \cdot 6 - 13 = 11$, which is essentially less than in Q.

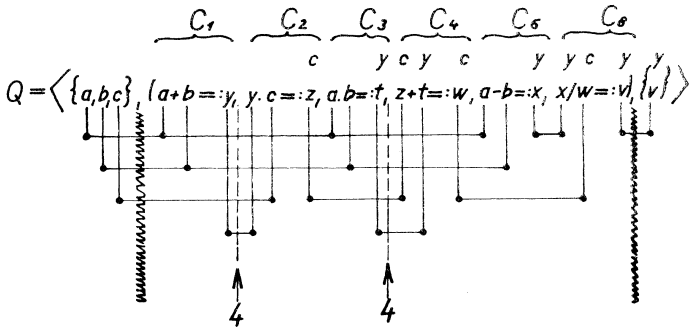

Fig. 2

All these examples of programs which compute the same arithmetic expression show that there are nontrivial problems to find among all the programs which compute the prescribed function those which have the least width or the least unefficiency of scopes etc. and to investigate the compatibility and dependency of all these measures of complexity of programs for one given computer or for all possible computers etc.

To any simple program for ***Mach*** the following construction is applicable: the commands are considered as *rewriting* (or *substitution*) *rules*, i.e. what is on the right-hand side of a command should be replaced by what is on its left-hand side and this should always be closed in brackets. These rules should be applied consecutively from the left to the right as follows: the i-th rule must be applied to all r_i occurrences of the corresponding address on the left-hand side of commands belonging to the i-th command scope S_i and after all these applications the i-th rule is left out from the program. During this construction some commands are changed, because on their left-hand side a more or less complicated arithmetical expressions occur. These new commands are called macro-commands. After finishing the construction the number of macro-commands left is equal to the number of output addresses in the original program.

By this construction a set of arithmetical expressions (with full bracketing) computed by the program is uniquely determined, i.e. they are the expressions on the left-hand sides of the remained macro-commands. Thus one can define that two

simple programs are structurally equivalent if they compute the same set of arithmetical expressions.

Two functions expressed by the following set of two arithmetic expressions $\{(a+b) / / (a - b), (a + b) . (a - b)\}$ can be computed by only one program more efficiently than by two separated programs. The following program computs both of these functions $P = (a + b =: x, a - b =: y, x/y =: z, a + b =: t, a - b =: w, t . w = =: v)$. It has the length 6, two input addresses a, b and two output addresses z, v. Another program $Q = (a + b =: x, a - b =: y, x/y =: z, x . y =: v)$ is equivalent to P but its length is 4. Thus the problem to find the shortest program among all equivalent ones is not trivial.

Finally it is well known that each arithmetical expression (with full bracketing) can be represented by certain oriented graphs without cycles but with labelled vertices, which is called logical net in the Switching Theory [2]. Very simple modifications of these logical nets are called here algorithmic nets because they express classes of programs, i.e. classes of algorithms. The algorithmic nets will be studied in the next section, where all above mentioned and some further problems are formulated as combinatorial and graph theoretical problems.

2. GRAPH THEORETICAL FORMULATION OF PROBLEMS

A *net* is finite, nonvoid, oriented multi-graph without cycles (and therefore also without slings). If a net is a graph, i.e. no parallel edges occur in it, then it is a set with an acyclic (and therefore also asymmetric) binary relation. A vertex in which no edge terminates is called an *input vertex* of the net and a vertex in which no edge starts is called an *output vertex*.

2.1 Characteristics of nets

In each net there exists at least one input vertex and at least one output vertex. In a connected net a vertex is simultaneously an input vertex and an output one too if and only if the net contains just one vertex and no edge. Each vertex of a net belongs to a path which starts in an input vertex and terminates in an output one. By omitting of a net an input vertex which is not an output one and all edges starting in it, we obtain again a net.

Proofs are obvious.

A net containing just one vertex (and no edge) is called *unproper*, and a net whose connected components are not unproper is called a *proper net*.

A proper net the input vertices of which are labelled by the addresses from ***Adr*** in such a way that

(2.1) two different input vertices are always labelled by two different addresses

and all the other vertices of which are labelled by operational symbols from ***Opr*** in such a way that

(2.2) the number of edges terminating at a vertex is equal to the number of variables of that function from ***Fct*** which is denoted by the operational symbol by which the vertex is labelled,

is called an *algorithmic net for* ***Mach*** if all operational symbols used in it denote symmetrical functions. If an operational symbol denotes a nonsymmetrical function, then all the edges terminating at the vertex labelled by that symbol must be distinguished from each other e.g.

(2.3) if there are $k \geqq 1$ different edges terminating at a vertex, then they are labelled by integers $1, 2, \ldots, k$.

The reason is to recognize which variable or which place of the denoted function corresponds to a given edge.

According to [2] where logical nets were introduced, the output vertex satisfies the following stronger condition: just one edge terminates at it and it is labelled by an output address (see Fig. 6 where this vertex is denoted by a dotted line). This last fact makes the net more symmetrical with respect to the input and output vertices (all other vertices are called *inner ones*), but from the algebraical point of view it is an unessential but superfluous complication of the structure.

2.2 Algorithm for determination of the algorithmic net of a simple program

First of all one chooses as many vertices as there occur different input addresses in the program (it is assumed that in a simple program at least one input and one output address occurs) and labels them by all the particular input addresses. Now one considers the first command which has the following form $f(X_1, X_2, \ldots, X_k) =: Y$ where $X_i, Y \in \boldsymbol{Adr}$ for $i = 1, 2, \ldots, k$ and $f \in \boldsymbol{Opr}$, chooses a new vertex, labels it by "f" and then chooses an edge starting at the input vertex labelled by X_i and terminating in the considered vertex for each $i = 1, 2, \ldots, k$, provided the function from ***Fct*** denoted by f is a symmetric one. If f is not symmetric then the chosen edge is labelled by i for $i = 1, 2, \ldots, k$. If all the commands from the program, the 1-st, 2-nd, ... $(n-1)$-st have been considered and the corresponding vertices and edges have been chosen, then the n-th command, which has again the form $f(X_1, X_2, \ldots, X_k) =: Y$ as above, is considered as follows. One chooses a new vertex labelled by "f" and for each $i = 1, 2, \ldots, k$ one chooses an edge starting either at the vertex corresponding to the m-th command where $1 \leqq m < n$ and m is as great as possible and such that on the right-hand side of the m-th command the address X_i occurs (in other words that the n-th command belongs to the scope $\boldsymbol{S}_m$) or, if there does not exist such a command, one chooses an edge starting at that input vertex

which is labelled by X_i, provided the function from ***Fct*** denoted by f is a symmetric one. If f is not symmetric then the chosen edge is labelled by "i" for $i = 1, 2, \ldots, k$.

2.3 The oriented multi-graph constructed by the algorithm 2.2 from a simple program for ***Mach*** is an algorithmic net for ***Mach*** having as many input vertices as the number of the input addresses in the program is and as many output vertices as the number of the output addresses in the program is.

Proof is obvious.

It is clear that never an unproper net may arise by 2.2 from a simple program. Thus it could seem to be reasonable to exclude these unproper nets of our considerations at all. Although the unproper nets do not have any direct correspondence with programs they play an important auxiliary role in the inductive reasoning as it will be shown further. It should be mentioned by this occasion that in the study of phrase-markers of sentences which are certain trees (see [3]), the isolated vertices played also an important auxiliary role.

2.4 Algorithm of unification of algorithmic nets N_i, $i = 1, 2, \ldots, n$

First of all the input vertices of all nets N_i where $i = 1, 2, \ldots, n$ are identified if they are labelled by the same address. Then the obtained labelled net is an algorithmic net again (because obviously (2.1) and (2.2) are satisfied). Further, one repeats the following step as long as possible: all the non-input vertices of an algorithmic net are identified which 1) are labelled by the same operational symbol, 2) all the edges terminating at any two of these vertices are starting in the same set of vertices and, if necessary, 3) all the edges terminating at any two of these vertices and starting at one and the same vertex are labelled by the same label, and further all edges terminating at all but one identified vertices are omitted (and of course all other edges starting at any of the identified vertices are preserved but all of them start at the unified vertex).

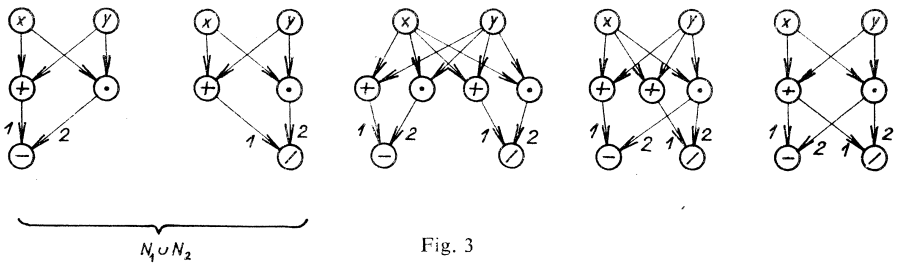

Fig. 3

In Fig. 3 there is an example of the process of unification where all nets are graphs (not multi-graphs), but in Fig. 4 it is shown that the process of unification can lead from a graph to a multi-graph.

According to [4] a notion of homomorphism and of simplicity of algorithmic nets can be introduced as follows: if $N = \langle V, E, I, \lambda_V, \lambda_E \rangle$ is an algorithmic net where V is its set of vertices, E its set of edges $(V \cap E = \emptyset)$, I its *incidence*, i.e. the function assigning to each edge from E an ordered pair of vertices from $V \times V$, λ_V is its *labelling of vertices* by elements from ***Adr*** $\cup$ ***Opr*** which satisfies (2.1) and (2.2) and λ_E is

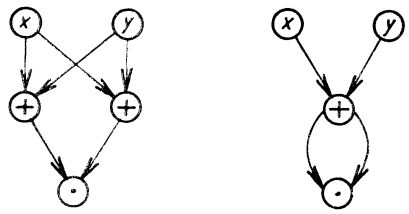

Fig. 4

its *labelling of edges* by integers 1, 2, 3, ... satisfying (2.3) and $N' = \langle V', E', I', \lambda_{V'}, \lambda_{E'} \rangle$ is another algorithmic net, then a mapping f of $V \cup E$ onto $V' \cup E'$ such that

$$f(V) = V', \quad f(E) = E', \tag{2.4}$$

$$I(e) = [x, y] \Rightarrow I'(f(e)) = [f(x), f(y)] \text{ for each } e \in E, \tag{2.5}$$

$$id(x) = id(f(x)),\ \lambda_V(x) = \lambda_{V'}(f(x)),\ \lambda_E(e) = \lambda_{E'}(f(e)) \text{ for each } x \in V \text{ and } e \in E \tag{2.6}$$

is called *homomorphism* where $id(x)$ is the *input degree*, i.e. the number of edges which terminate in the vertex x. If f is a one-to-one mapping, then it is called *isomorphism*. An algorithmic net is called *simple* (see [4]) if each homomorphic image of it is isomorphic with it.

It is easy to see that each algorithmic net in Fig. 3 or 4 is a homomorphic image of any algorithmic net (or of a labelled net $N_1 \cup N_2$) located on the left-hand side of it. The far right nets in Fig. 3 and 4 are simple nets.

It follows by (2.6) that a homomorphism f of an algorithmic net N onto another one N' satisfies also the following condition

(2.7) the homomorphism f preserves the parallelism of edges, i.e. if $I(e) = I(e') = [x, y]$ where $e, e' \in E$ and $e \neq e'$ then $f(e) \neq f(e')$,

2.5 A simple algorithmic net contains the smallest number of vertices among all algorithmic nets which can be mapped by an homomorphism onto it.

The proof is obvious.

2.6 Characteristics of unificated algorithmic nets

The oriented and labelled multigraph constructed by the algorithm 2.4 is a simple algorithmic net.

Proof. The union of the given algorithmic nets N_i, $1 \leqq i \leqq n$ in the algorithm 2.4 need not be an algorithmic net only because the condition (2.1) need not be satisfied, but (2.1) is satisfied after the first step of the algorithm when the input vertices labelled in the same way are identified. Further it is clear that after each step of identification again an algorithmic net is constructed from an algorithmic net. If an algorithmic net N not allowing any further identification of vertices were not simple, there would exist a homomorphism f of it onto another net N' which is not isomorphism, i.e. there would exist two vertices $v_1, v_2 \in V$, $v_1 \neq v_2$ such that $f(v_1) = f(v_2)$. Then by (2.5) and (2.6) it must hold $\lambda_V(v_1) = \lambda_V(v_2)$ and to each $e_1 \in E$ which terminates at v, i.e. $I_E(e_1) = (v, v_1)$ for a $v \in V$ there exists $e_2 \in E$ such that $I_E(e_2) = (v, v_2)$ and moreover $\lambda_E(e_1) = \lambda_E(e_2)$. This means that v_1 and v_2 may be identified which is a contradiction.

With respect to 2.6 one can call the result of the algorithm of unification 2.4 the *unificated net.*

Two algorithmic nets are called *equivalent* if their unificated nets are *almost isomorphic*, i.e. if the condition (2.6) of isomorphism is weakened, i.e. it is replaced by the following one:

$$id(x) = id(f(x)) \text{ for each } x \in V,$$

(2.6*) $\lambda_V(x) = \lambda_{V'}(f(x))$ for each $x \in V$ which is not an input vertex and

$$\lambda_E(e) = \lambda_{E'}(f(e)) \text{ for each } e \in E,$$

which expresses full independence on the input addresses.

With respect to the algorithm 2.2 there arises a natural and important question how to determine all the simple programs for ***Mach*** such that their algorithmic nets constructed by 2.2 are equivalent.

Two simple programs having equivalent algorithmic nets are called *structurally equivalent.*

First of all a more special question will be answered, i.e. what are all the simple programs by which the same algorithmic net (using the algorithm 2.2) is determined.

If $N = \langle V, E, I \rangle$ is a net, $|V| = n$, then the total ordering $P = (v_1, v_2, \ldots, v_n)$ of the set of its vertices V is called a *course of the net* N if the following condition is satisfied

(2.8) v_i is an input vertex of the net $N_i = \langle V_i, E_i, I_i \rangle$ for each $i = 1, 2, \ldots, n$, when $N_1 = N$ and $V_{i+1} = V - \{v_1, v_2, \ldots, v_i\}$, $E_{i+1} = E - \{$the set of all edges starting or terminating in any of the vertices $v_1, v_2, \ldots, v_i\}$ and $I_{i+1} = I|_{E_{i+1}}$ for each $i = 1, 2, 3, \ldots, n-1$.

The number of the input vertices or output vertices of a net N is called its *input width* or *output width* respectively and denoted by inwi (N) or ouwi (N). Further inwi (N_i) or ouwi (N_i) where N_i is from (2.8) is called *input width* or *output width of* P *in* v_i and $\max_{1 \leqq i \leqq n}$ inwi (N_i) or $\max_{1 \leqq i \leqq n}$ ouwi (N_i) is called the *input width* or *output width of* P and denoted inwi (P) or ouwi (P) respectively.

By each net $N = \langle V, E, I \rangle$ which is a multi-graph, another net $\overline{N} = \langle V, \varrho \rangle$ which is a graph is determinated by the requirement $\varrho = \{I(e);\ e \in E\}$, because obviously ϱ is again an acyclic relation.

It is easy to see that a total ordering $P = (v_1, v_2, \ldots, v_n)$ of V satisfies (2.8) for N if an only if P satisfies (2.8) for $\overline{N}$, i.e. P is always simultaneously a course of both N and $\overline{N}$.

2.7 Characteristics of nets and their courses

Let N be a net and $\overline{N} = \langle V, \varrho \rangle$ the corresponding net without parallel edges. Then $\langle V, T\varrho \rangle$ where $T\varrho$ is the transitive closure of the relation ϱ is a partially ordered set, the maximal or the minimal elements of which are the input vertices or output vertices respectively (when $x \varrho y$ means $x \geqq y$). Further let $P = (v_1, v_2, \ldots, v_n)$ be a total ordering of V and let $\sigma = \{(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)\}$. Then P is a course of N if and only if $T\varrho \subset T\sigma$, i.e. if $T\sigma$ is a total extension of the partial ordering $T\varrho$ (see [5]).

Proof. The first part is obvious because the transitive closure of an acyclic relation must be again acyclic and therefore asymmetric, i.e. it must be a partiall ordering.

In the second part, first of all let $(v_1, v_2, \ldots, v_n) = P$ be a course of N, and therefore of $\overline{N}$ too, and let $(v_j, v_h) \in T\varrho$, i.e. there exists a path $(u_0, u_1, \ldots, u_k)$ in $\overline{N}$ such that $k \geqq 1$ and $u_0 = v_j$ and $u_k = v_h$. We want to prove that $(v_j, v_h) \in T\sigma$, which is true if and only if $j < h$. It is clear that $(u_{p-1}, u_p) \in \varrho$ for each $p = 1, 2, \ldots, k$ and by (2.8) it follows that the vertex u_{p-1} must precede the vertex u_p in each course of $\overline{N}$ and therefore also in the course P, i.e. if $u_{p-1} = v_r$ and $u_p = v_s$, then $r < s$ for each $p = 1, 2, \ldots, n$. Thus obviously $u_0 (= v_j)$ must precede $u_k (= v_h)$ in P too, which means $j < h$.

If on the contrary $T\varrho \subset T\sigma$, then we want to prove that $P = (v_1, v_2, \ldots, v_n)$ is a course of $\overline{N}$, i.e. that P satisfies (2.8) for each v_i where $i = 1, 2, \ldots, n$. If this were not true for an index i, $1 \leqq i < n$, then v_i would not be an input vertex of $\overline{N}_i$, which means that there exists $v_j \in V_i$ such that $(v_j, v_i) \in \varrho$ and $i < j$. Therefore by $(v_j, v_i) \in \varrho$ it follows $(v_j, v_i) \in T\sigma$ and by $i < j$ (and by the definition of σ) it follows $(v_i, v_j) \in T\sigma$, which means that $T\sigma$ is not an acyclic relation; thus the required contradiction is found.

Further let us consider a course $P = (v_1, v_2, \ldots, v_n)$ of a net $N = \langle V, E, I \rangle$ or, which is the same, of the corresponding net $\overline{N} = \langle V, \varrho \rangle$. The *scope of the vertex* v_i *in* P is an interval $\mathbf{Sc}_P(v_i)$ for $i = 1, 2, \ldots, n$, defined as follows:

if v_i is an output vertex of $\bar{N}$, then $\mathbf{Sc}_P(v_i) = (v_{i+1}, v_{i+2}, \ldots, v_n)$;

if v_i is not an output vertex of $\bar{N}$, then there exists v_j such that $(v_i, v_j) \in \varrho$ and j is as great as possible and either v_i is an input vertex and then $\mathbf{Sc}_P(v_i) = (v_1, v_2, \ldots, v_j)$ or v_i is not an input vertex and then $\mathbf{Sc}_P(v_i) = (v_{i+1}, v_{i+2}, \ldots, v_j)$.

If v_i is and input vertex $\mathbf{Sc}_P(v_i)$ is called an *input scope*, otherwise a *non-input scope*.

It should be mentioned that $\mathbf{Sc}_P(v_i) = \emptyset$ if an only if $i = n$ and that $\mathbf{Sc}_P(v_i) = \mathbf{Sc}_P(v_j)$ can happen for $i \neq j$ only if both v_i and v_j are the input vertices.

The number of scopes of the course P, which contain a vertex v_i, is called the *scope width of P in v_i* and is denoted $\mathrm{scwi}(v_i)$ and the number $\mathrm{scwi}(P) = \max_{1 \leqq i \leqq n} \mathrm{scwi}(v_i)$ is called the *scope width of P*.

The *scope graph* $G_P = \langle V, H_P \rangle$ of the course P is an unoriented graph without slings where V is the set of vertices of the considered net N and $H_P = \{\{v_i, v_j\}; i \neq j$ and $v_i, v_j \in V$ and $\mathbf{Sc}_P(v_i) \cap \mathbf{Sc}_P(v_j) \neq \emptyset\}$.

2.8 The chromatic number of G_P is equal to the scope width of P, i.e. $\chi(G_P) = \mathrm{scwi}(P)$.

Proof. It is clear that in G_P there exists a complete subgraph with $\mathrm{scwi}(P)$ vertices and therefore $\mathrm{scwi}(P) \leqq \chi(G_P)$. Further let us have $\mathrm{scwi}(P)$ different colours and let us colour the vertices of G_P with them as follows from the left to the right according to $P = (v_1, v_2, \ldots, v_n)$: v_1 is coloured arbitrarily and if $v_1, v_2, \ldots, v_{k-1}$, where $1 < k < n$, are coloured in such a way that no two of them connected by an edge are of the same colour, then v_k is connected by an edge with at most $\mathrm{scwi}(P) - 1$ preceding vertices and with no vertex v_j for $j > i$ and therefore v_k can be coloured by one of the remaining colours which is different from all those used for the vertices connected by an edge with v_k. This proves $\mathrm{scwi}(P) \leqq \chi(G_P)$.

Finally let D be a chromatic decomposition of G_P and let λ_D be a one-to-one mapping of D into ***Adr***.

Now it is easy to see that the above mentioned question can be answered as follows:

2.9 Characteristics and construction of all simple programs for a *Mach* which lead by the algorithm 2.2 to the same algorithmic net for this *Mach*

Each simple program for a ***Mach*** which leads by the algorithm 2.2 to the prescribed (unique with respect to an isomorphism) algorithmic net $N = \langle V, E, I, \lambda_V, \lambda_E \rangle$ can be obtained by the following construction, which depends on the choice of a course $P = (v_1, v_2, \ldots, v_n)$ of N, of a chromatic decomposition D of G_P and of a one-to-one labelling λ_D of D into ***Adr*** such that each input-scope is labelled by that address by which the input vertex belonging to this scope is labelled: using λ_D, an auxiliary labelling of edges from E is introduced in such a way that all edges starting at the vertex v_i from a class $D_j \in D$ are labelled by the address $\lambda_D(D_j)$ for each $i = 1, 2, \ldots, n$ and then to each non-input vertex the corresponding command is chosen according

to Fig. 5. Finally the ordering of these commands in the constructed program is the same as the ordering of the corresponding non-input vertices in P.

Proof. First of all let us show that the constructed sequence of commands is a simple program for ***Mach***. By the construction it is clear that all the chosen commands are commands for ***Mach*** because N was an algorithmic net for ***Mach***. The sequence of these commands is a simple program for ***Mach***, because with one exception (of the last command) all other commands have non void scopes.

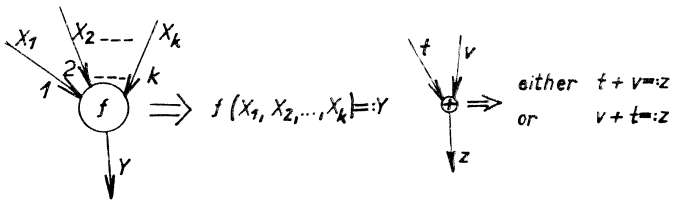

Fig. 5

Further one proves by induction with respect to the number of vertices of N that by 2.2 each constructed program leads to the original net N and finally that each program with this property can be obtained by the described construction.

In Fig. 6 the unificated algorithmic net is given which corresponds to all the programs P, P' in Fig. 1 and Q, Q' in Fig. 2. The auxiliary labellings by addresses mentioned in 2.9 are not shown in Fig. 6, but they are different in different cases.

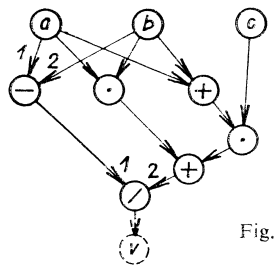

Fig. 6

Finally, the original question concerning the structurally equivalent programs is answered as follows:

2.10 Characteristics of structural equivalence of simple programs

If P is a (simple) program for a ***Mach***, then all programs for this ***Mach*** which are structurally equivalent to P can be obtained as follows: a) one constructs the algorithmic net A by the algorithm 2.2 for P, b) then one constructs the unificated net

A' to A by the algorithm 2.4, c) further one constructs an arbitrary unificated net B' which is almost isomorphic with A' by an arbitrary' change of addresses of input vertices of A' (according to the condition (2.1)), d) now an arbitrary homomorphic pattern B'' of B' is constructed in such a way than one takes an arbitrary course $(v_1, v_2, \ldots, v_n)$ of B' and splits each non-input vertex v_i and each edge which terminates in it into h_i new vertices and h_i new edges, respectively, consecutively (but backwards) for $i = n, n-1, \ldots, 2, 1$ where $h_i \geqq 1$ is a quite arbitrary integer for an output vertex v_i but h_i satisfies $1 \leqq h_i \leqq od(v_i)$ for a non-output vertex v_i (where the output degree $od(v_i)$ is the number of edges which start in v_i with respect to the mediate net constructed in the previous step v_{i+1}; for $i = n$ one starts with B' itself); the vertex v_i is splitted into (or replaced by) h_i new vertices labelled by the same symbol as v_i in such a way that at each of the new vertices at least one edge starts, which originally started in v_i and each splitted edge starts in the same vertex as the original edge, is labelled (if necessary) by the same symbol as the original edge, and all h_i new splitted edges terminate exactly in all h_i new vertices, and finally e) one constructs all courses and then also all simple programs Q to each B'' (by the construction 2.9).

Now different extremal problems can be formulated either in the class of all equivalent nets or in the class of all courses (or programs) of a given net or in the class of all structurally equivalent courses.

Using the measures of complexity of courses defined above, several questions may be formulated.

Problem 1. For a given net N determine $\min_{P \in P(N)} \text{inwi}(P)$, $\min_{P \in P(N)} \text{ouwi}(P)$, $\max_{P \in P(N)} \text{inwi}(P)$ and $\max_{P \in P(N)} \text{ouwi}(P)$ where $P(N)$ is the set of all courses P of the net N. Further find an efficient algorithm for the construction of a course P (or of all courses P) of the net N such that its inwi (P) or ouwi (P) is extremal.

Of considerable interest is the following

Problem 2. For a given net N determine $\min_{P \in P(N)} \text{scwi}(P)$ and $\max_{P \in P(N)} \text{scwi}(P)$ where $P(N)$ is the set of all courses P of the net N. Further find and efficient algorithm for the construction of a course P (or of all courses P) of the net N such that inwi (P) or ouwi (P) is extremal.

It is easy to see a strong connection between the algorithmic nets and the transportation nets (see e.g. [6]), where the notion of a cut is introduced. E.g. $\min_{P \in P(N)} \text{scwi}(P)$ is a measure of complexity of the net N itself, because it means the minimal number of edges starting at different vertices which must be cut or contained in a cut of the net N.

A further important and well known measure of complexity of a course is its length which is equal to the number of vertices of its net. On the other hand, it should be

mentioned here that if a program and its algorithmic net are considered, then the length of the program plus the number of its input addresses is equal to the number of vertices of its net.

The corresponding extremal problems concerning the length of a course P are solved by 2.5 and by homomorphic characteristics (see [4]) of any homomorphic pattern of the simple net. A very important role is played by the measures of complexity of programs which express different efficiencies concerning the storage. If the duration of all instructions is the same then the time unit of storage is one address during one instruction. In the graph-theoretical version the following two numbers are assigned to a course $P = (v_1, v_2, \ldots, v_n)$ of a net N. First of all it is the *non-efficiency of scopes of the course P*

$$\mathrm{nefsc}\,(P) = \sum_{i=1}^{n} (s_i - r_i)$$

where $s_i = |Sc(v_i)|$ and r_i is the number of different vertices such that there is an edge terminating at one of them and starting at v_i (i.e. r_i is the number of edges starting at v_i in the corresponding net $\bar{N}$ without parallel edges); obviously $r_i \leqq s_i$ (and in the language interpretation s_i denotes how long something must be stored and r_i how many times or in how many instructions it is used).

Secondly it is the *non-efficiency of adresses of the course P*

$$\mathrm{nefadr}\,(P) = \chi(G_P) \,.\, \mathrm{leng}\, P - \sum_{i=1}^{n} s_i$$

where leng P is the length of the course P and all other terms are defined above. In the language interpretation $\sum_{i=1}^{n} s_i$ is the actual time measure of storage and $\chi(G_P) \,.\, \mathrm{leng}\, P$ is its possible maximum.

Problem 3. For a given net N determine $\min_{P \in P(N)} \mathrm{nefsg}\,(P)$, $\max_{P \in P(N)} \mathrm{nefsc}\,(P)$, $\min_{P \in P(N)} \mathrm{nefadr}\,(P)$ and $\max_{P \in P(N)} \mathrm{nefadr}\,(P)$. Further find an efficient algorithm for the construction of a course P (or of all courses P) such that its nefsc (P) and nefadr (P) are extremal.

Now very natural questions arise concerning the compatibility and mutual dependence of all the defined measures.

Problem 4. For a given net N does there exist its course P such that $\mathrm{scwi}\,(P) = \min_{Q \in P(N)} \mathrm{scwi}\,(Q)$ and simultaneously $\mathrm{nefadr}\,(P) = \min_{Q \in P(N)} \mathrm{nefadr}\,(Q)$? Similarly for other pairs of measures the same question arises.

References

[1] *K. Čulík:* On semanitics of programming languages. *J. Dörr, G. Hotz:* Automatentheorie und formale Sprachen, Bibliographisches Institut AG, Mannheim 1970, 291—302.

[2] *K. Čulík, V. Doležal, M. Fiedler:* Combinatorial analysis in praxis (Czech), SNTL, Prague 1967.

[3] *K. Čulík:* On some transformations in context-free grammars and languages, Czech. Math. Jour. 17 (1967), Academia, 278—311.

[4] *K. Čulík:* Zur Theorie der Graphen, Čas. pro pěst. mat. 83 (1958), 133—155.

[5] *G. Birkhoff:* Lattice theory, New York 1948.

[6] *C. Berge:* Théorie des graphes et ses applications, Dunod, Paris 1958.

[7] *R. Sethi, J. D. Ullman:* The Generation of Optimal Code for Arithmetic Expressions. Journal of ACM, Vol. 17, No. 4, October 1970, pp. 715—728.

[8] *A. Blikle:* Addressless units for carrying out loop-free computations, Polish Academy of Sciences, Institute of Mathematics, July 1970, Warsaw.

Remark. In [7] and [8] a special type of algorithmic net is involved, where 1) $id(x) = 2$ for each non-input vertex x, 2) there is only one output vertex and 3) no parallel paths occur, i. e. a binary rooted tree. In [8] a push down store is assumed.

Souhrn

KOMBINATORICKÉ PROBLÉMY V TEORII SLOŽITOSTI ALGORITMICKÝCH SÍTÍ BEZ CYKLŮ PRO JEDNODUCHÉ POČÍTAČE

KAREL ČULÍK

Algoritmické sítě bez cyklů jsou zobecněním logických sítí bez cyklů, tj. jsou to konečné, orientované a acyklické grafy nebo multigrafy s ohodnocenými uzly. Jistá úplná uspořádání jejich uzlů se nazývají jejich *průběhy*. Každým průběhem je určen jistý *intervalový graf* a je zvolen jeden z jeho *chromatických rozkladů*. Jsou uvedeny následující *míry složitosti* průběhů se zvolenými rozklady: 1) *délka průběhu*, tj. počet uzlů uvažované sítě, 2) *šířka průběhu*, tj. maximální řád úplného podgrafu v příslušném intervalovém grafu, 3) *kapacita paměti průběhu*, tj. počet prvků zvoleného rozkladu, 4) *neefektivnost rozsahů uvažovaného průběhu*, tj. součet rozdílů délek intervalů a výstupních stupňů odpovídajících uzlů a konečně 5) *neefektivnost adres*, tj. rozdíl součinu kapacity s délkou a součtu délek všech intervalů. Předložené problémy jsou extremální a týkají se buď určení minima některé z uvedených měr složitosti pro všechny průběhy dané sítě a nebo se týkají slučitelnosti a závislosti jednotlivých měr. Na začátku jsou uvedeny motivace zavedených měr složitosti pocházející z měření složitosti jednoduchých programů pro jednoduché počítače.

Author's address: Prof. Dr. *Karel Čulík*, DrSc, Matematický ústav ČSAV v Praze, Žitná 25, Praha 1.