K-GRAPH MACHINES:
Generalizing Turing's Machines and Arguments

by

Wilfried Sieg and John Byrnes

June 1994
revised July 1995

Report CMU-PHIL-55

Philosophy
Methodology
Logic

Pittsburgh, Pennsylvania 15213-3890

K-GRAPH MACHINES:
generalizing Turing's machines and arguments

Wilfried Sieg and John Byrnes
Department of Philosophy
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract: The notion of *mechanical process* has played a crucial role in mathematical logic since the early thirties; it has become central in computer science, artificial intelligence, and cognitive psychology. But the discussion of Church's Thesis, which identifies the informal concept with a mathematically precise one, has hardly progressed beyond the pioneering work of Church, Gödel, Post, and Turing. Turing addressed directly the question: *What are the possible mechanical processes a human computor can carry out in calculating values of a number-theoretic function?* He claimed that all such processes can be simulated by machines, in modern terms, by deterministic Turing machines. Turing's considerations for this claim involved, first, a formulation of boundedness and locality conditions (for linear symbolic configurations and mechanical operations); second, a proof that computational processes (satisfying these conditions) can be carried out by Turing machines; third, the central thesis that all mechanical processes carried out by human computors must satisfy the conditions. In Turing's presentation these three aspects are intertwined and important steps in the proof are only hinted at. We introduce *K-graph machines* and use them to give a detailed mathematical explication of the first two aspects of Turing's considerations for more general configurations, i.e. K-graphs. This generalization of machines and theorems provides, in our view, a significant strengthening of Turing's argument for his central thesis.

INTRODUCTION. Turing's analysis of effective calculability is a paradigm of a foundational study that (i) led from an informally understood concept to a mathematically precise notion, (ii) offered a detailed investigation of that mathematical notion, and (iii) settled an important open question, namely the *Entscheidungsproblem*. The special character of Turing's analysis was recognized immediately by Church in his review of Turing's 1936 paper. The review was published in the first issue of the 1937 volume of the Journal of Symbolic Logic, and Church contrasted in it Turing's mathematical notion for effective calculability (via idealized machines) with his own (via λ-definability) and Gödel's general recursiveness and asserted: "Of these, the first has the advantage of making the identification with effectiveness in the ordinary (not explicitly defined) sense evident immediately ... ."

Gödel had noticed in his (1936) an "absoluteness" of the concept of computability, but found only Turing's analysis convincing; he claimed that Turing's work provides "a precise and unquestionably adequate definition of the general concept of formal system" (1964, p. 369). As a formal system is simply defined to be a mechanical procedure for producing theorems, the adequacy of the definition rests on Turing's analysis of mechanical procedures. And with respect to the latter Gödel remarked (pp. 369-70): "Turing's work gives an analysis of the concept of 'mechanical procedure' (alias 'algorithm' or 'computation procedure' or 'finite combinatorial

procedure'). This concept is *shown* [our emphasis] to be equivalent with that of a 'Turing machine'." Nowhere in Gödel's writings is there an indication of the nature of Turing's conceptual analysis or of a proof for the claim that the (analyzed) concept is equivalent with that of a Turing machine.

Gödel's schematic description of Turing's way of proceeding is indeed correct: in section 9 of (Turing 1936) there is an analysis of effective calculability, and the analysis is intertwined with a sketch of an argument showing that mechanical procedures on linear configurations can be performed by very restricted machines, i.e., by deterministic Turing machines over a two-letter alphabet. Turing intended to give an analysis of mechanical processes on planar configurations; but such processes are not described, let alone proved to be reducible to computations on linear objects. This gap in Turing's considerations is the starting-point of our work. We formulate broad boundedness and locality conditions that emerge from Turing's conceptual analysis, give a precise mathematical description of planar and even more general computations, and present a detailed reductive argument. For the descriptive part we introduce *K-graph machines*; they are a far-reaching generalization of Post production systems and thus, via Post's description of Turing machines, also of Turing machines.

1. TURING's ANALYSIS[1]. In 1936, the very year in which Turing's paper appeared, Post published a computation model strikingly similar to Turing's. Our discussion of Post's model is not to emphasize this well-known similarity, but rather to bring out the strikingly dissimilar methodological attitudes underlying Post's and Turing's work. Post has a worker operate in a *symbol space* consisting of "a two way infinite sequence of spaces or boxes ...".[2] The boxes admit only two conditions: they can be empty or unmarked, or they can be marked by a single sign, say a vertical stroke. The worker can be in and operate in just one box at a time, and he can perform a number of *primitive acts*; namely, make a vertical stroke [V], erase a vertical stroke [E], move to the box immediately to the right [M_r] or to the left [M_l] (of the box he is in), and determine whether the box he is in is marked or not [D]. In carrying out a

---

[1] This section is based on (Sieg 1994) which was completed in June of 1991; for details of the reconstruction of Turing's analysis and also for the broader systematic and historical context of our investigations we refer the reader to that paper.

[2] Post remarks that the infinite sequence of boxes is ordinally similar to the series of integers and can be replaced by a potentially infinite one, expanding the finite sequence as necessary.

particular "combinatory process" the worker begins in a special box and then follows directions from a finite, numbered sequence of instructions. The i-th direction, i between 1 and n, is in one of the following forms: (i) carry out act V, E, M_r, or M_l and then follow direction j_i, (ii) carry out act D and then, depending on whether the answer is positive or negative, follow direction j_i' or j_i''. (Post has a special stop instruction, but that can be replaced by the convention to halt, when the number of the next direction is greater than n.)

Are there intrinsic reasons for choosing Post's *Formulation 1* as an explication of effective calculability, except for its simplicity and Post's expectation that it will turn out to be equivalent to recursiveness? An answer to this question is not clear from Post's paper, at the end of which he wrote:

> The writer expects the present formulation to turn out to be equivalent to recursiveness in the sense of the Gödel-Church development. Its purpose, however, is not only to present a system of a certain logical potency but also, in its restricted field, of psychological fidelity. In the latter sense wider and wider formulations are contemplated. On the other hand, our aim will be to show that all such are logically reducible to formulation 1. We offer this conclusion at the present moment as a *working hypothesis*. And to our mind such is Church's identification of effective calculability with recursiveness.

Investigating wider and wider formulations and *reducing* them to *Formulation 1* would change for Post this "hypothesis not so much to a definition or to an axiom but to a *natural law*". It is methodologically remarkable that Turing proceeded in *exactly* the opposite way when trying to justify that all computable numbers are machine computable or, in our way of speaking, that all effectively calculable functions are Turing computable: He did not try to extend a narrow notion reducibly and obtain in this way quasi-empirical support, but rather he analyzed the intended broad concept and reduced it to a narrow one -- once and for all. The intended concept was effective and mechanical calculability by a human being, and in the reductive argument Turing brought to bear limitations of the computing agent.

Turing's *On computable numbers* opens with a description of what is ostensibly its subject, namely, "real numbers whose expressions as a decimal are calculable by finite means". Turing was quick to point out that the problem of explicating "calculable by finite means" is the same when considering, e.g., computable functions of an integral variable. Thus it sufficed to address the question: "What does it mean for a real number to be calculable by finite means?" But Turing developed first the theory of his

machines.[3] A Turing machine consists of a finite, but potentially infinite tape; the tape is divided into squares, and each square may carry a symbol from a finite alphabet, say, the two-letter alphabet consisting just of 0 and 1. The machine is able to scan one square at a time and perform, depending on the content of the observed square and its own internal state, one of four operations: print 0, print 1, or shift attention to one of the two immediately adjacent squares. The operation of the machine is given by a finite list of commands in the form of quadruples $q_i s_k c_l q_m$ that express: if the machine is in internal state q_i and finds symbol s_k on the square it is scanning, then it is to carry out operation c_l and change its state to q_m. The deterministic character of the machine operation is guaranteed by the requirement that a program must not contain two different quadruples with the same first two components.

In section 9 Turing argues that the operations of his machines "include all those which are used in the computation of a number". But he does not try to establish the claim directly; he rather attempts to answer what he views as "the real question at issue": "What are the possible processes which can be carried out [by a computor[4]] in computing a number?" Turing imagines a computor writing symbols on paper that is divided into squares "like a child's arithmetic book". As the two-dimensional character of this computing space is taken -- *without any argument* -- not to be essential, Turing considers the one-dimensional tape divided into squares as the basic computing space and formulates one important restriction. That restriction is motivated by limits of our sensory apparatus to distinguish *at one glance* between symbolic configurations of sufficient complexity. It states that only finitely many distinct symbols can be written on a square. Turing suggests as a reason that "If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent", and we would not be able to distinguish at one glance between them. A second and clearly related way of arguing this point uses a finite number of symbols and strings of such symbols. E.g., Arabic numerals like 9979 or 9989 are seen by us at one glance to be different; however, it is not possible for us to determine immediately that 9889995496789998769 is different from 98899954967899998769. This second

[3] Note that the presentation of Turing machines we give is not Turing's, but rather the one that evolved from Post's formulation in (1947).

[4] Following Gandy, we distinguish between a *computor* (a human carrying out a mechanical computation) and a *computer* (a mechanical device employed for computational purposes); cf. (Gandy 1988), p. 81, in particular fn. 24.

avenue suggests that a computor can operate directly only on a finite number of (linear) configurations.

Now we turn to the question: What determines the steps of the computor, and what kind of elementary operations can he carry out? The behavior is *uniquely* determined at any moment by two factors: (i) the symbolic configuration he observes and (ii) his "internal state". This uniqueness requirement may be called the **determinacy condition (D)**; it guarantees that computations are deterministic. Internal states, or as Turing also says "states of mind", are introduced to have the computor's behavior depend possibly on earlier observations and, thus, to reflect his experience. Since Turing wanted to isolate operations of the computor that are "so elementary that it is not easy to imagine them further divided", it is crucial that symbolic configurations relevant for fixing the circumstances for the actions of a computor are immediately recognizable. So we are led to postulate that a computor has to satisfy two **boundedness conditions:**

(B.1) *there is a fixed bound for the number of symbolic configurations a computor can immediately recognize;*
(B.2)[5] *there is a fixed bound for the number of internal states that need be taken into account.*

For a given computor there are consequently only boundedly many different combinations of symbolic configurations and internal states. Since his behavior is, according to **(D)**, uniquely determined by such combinations and associated operations, the computor can carry out at most finitely many different operations. These operations are restricted by the following **locality conditions:**

(L.1) *only elements of observed symbolic configurations can be changed;*
(L.2) *the distribution of observed squares can be changed, but each of the new observed squares must be within a bounded distance of an immediately previously observed square.*

Turing emphasized that "the new observed squares must be immediately recognisable by the [computor]"; that means the observed configurations arising from changes according to **(L.2)** must be among the

[5] Gödel objected in (1972) to this condition for a notion of human calculability that might properly extend mechanical calculability; for a computor it is quite unobjectionable.

finitely many ones of **(B.1)**. Clearly, the same must hold for the symbolic configurations resulting from changes according to **(L.1)**. Since some steps may involve a change of internal state, Turing concluded that the most general single operation is a change *either* of symbolic configuration and, possibly, internal state *or* of observed square and, possibly, internal state. With this restrictive analysis of the steps a computor can take, the proposition that his computations can be carried out by a Turing machine is established rather easily.[6] Thus we have:

Turing's Theorem (for calculable functions) *Any number theoretic function F that can be calculated by a computor satisfying the determinacy condition* **(D)** *and the conditions* **(B)** *and* **(L)** *can be computed by a Turing machine.*

As the Turing computable functions are recursive, F is recursive. This argument for F's recursiveness does not appeal to any form of Church's Thesis; rather, such an appeal is replaced by the assumption that the calculation of F is done by a computor satisfying the conditions **(D)**, **(B)**, and **(L)**. If that assumption is to be discharged a substantive thesis is needed. We call this thesis -- that a mechanical computor must satisfy the conditions **(D)** and **(B)**, and that the elementary operations he can carry out must be restricted as conditions **(L)** require -- **Turing's Central Thesis.**

In the historical and systematic context Turing found himself, he asked exactly the right question: What are the possible processes a computor can carry out in calculating a number? The general problematic *required* an analysis of the idealized capabilities of a mechanical computor, and exactly this feature makes the analysis epistemologically significant. The separation of conceptual analysis (leading to the axiomatic conditions) and rigorous proof (establishing Turing's Theorem) is essential for clarifying on what the correctness of his general thesis rests; namely, on recognizing that the axiomatic conditions are true for computors who proceed mechanically. We have to remember that clearly when engaging in methodological discussions concerning artificial intelligence and cognitive science. Even Gödel got it wrong, when he claimed that Turing's argument in the 1936 paper was

---

[6] Turing constructed machines that mimic the work of computors on linear configurations directly and observed: "The machines just described do not differ very essentially from computing machines as defined in § 2, and corresponding to any machine of this type a computing machine can be constructed to compute the same sequence, that is to say the sequence computed by the computer [in our terminology: computor]." Cf. section 2 below for this reductive claim.

intended to show that "mental processes cannot go beyond mechanical procedures".

2. POST PRODUCTIONS & PUZZLES. Gödel's misunderstanding of the intended scope of Turing's analysis may be in part due to Turing's provocative, but only figurative, attribution of "states of mind" to machines; it is surprising nevertheless, as Turing argues at length for the eliminability of states of mind in section 9 (III) of his paper. He describes a modified computor and avoids the introduction of "state of mind", considering instead "a more physical and definite counterpart of it". The computor is now allowed to work in a desultory manner, possibly doing only one step of the computation at a sitting: "It is always possible for the [computor] to break off from his work, to go away and forget all about it, and later to come back and go on with it." But on breaking off the computor must leave a "note of instruction" that informs him on how to proceed when returning to his job; such notes are the "counterparts" of states of mind. Turing incorporates notes into "state formulas" (in the language of first order logic) that describe states of a machine mimicking the computor and formulates appropriate rules that transform a given state into the next one.

Post used in (1947) a most elegant way of describing Turing machines purely symbolically via his production systems (on the way to solving, negatively, the word-problem for semi-groups).[7] The configurations of a Turing machine are given by *instantaneous descriptions* of the form $\alpha q_l s_k \beta$, where α and β are possibly empty strings of symbols in the machine's alphabet; more precisely, an *id* contains exactly one state symbol, and to its right there must be at least one symbol. Such *id*'s express that the current tape content is $\alpha s_k \beta$, the machine is in state q_l, and it scans (a square with symbol) s_k. Quadruples $q_i s_k c_l q_m$ of the program are represented by rules; for example, if the operation c_l is *print* 0, the corresponding rule is:

$$\alpha q_l s_k \beta \Rightarrow \alpha q_m 0 \beta.$$

That can be done, obviously, for all the different operations; one just has to append 0 or s_0 to α (β) in case c_l is the operation *move to the left* (*right*) and α (β) is the empty string -- reflecting the expansion of the only potentially infinite tape by a blank square. This formulation can be generalized so that

[7] Post's way of looking at Turing machines underlies also the presentation in (Davis 1958); for a more detailed discussion the reader is referred to that classical text.

machines operate directly on finite strings of symbols; such operations can be indicated as follows:

$$\alpha\gamma q_l\delta\beta \Rightarrow \alpha\gamma^* q_m\delta^*\beta.$$

If in internal state q_l a *string machine* recognizes the string $\gamma\delta$ (i.e., takes in the sequence at one glance), it replaces that string by $\gamma^*\delta^*$ and changes its internal state to q_m. Calling ordinary Turing machines *letter machines*, Turing's claim reported in note 6 can be formulated as a **Reduction Lemma:** Any computation of a string machine can be carried out by a letter machine.

The rule systems describing string machines are semi-Thue systems and, as the latter, not deterministic, if their programs are just sequences of production rules. The usual non-determinism certainly can be excluded by requiring that, if the antecedents of two rules coincide, so must the consequents. But that requirement does not remove every possibility of two rules being applicable simultaneously: consider a machine whose program includes in addition to the above rule also the rule

$$\alpha\gamma^{\#} q_l\delta^{\#}\beta \Rightarrow \alpha\gamma^{\perp} q_n\delta^{\perp}\beta,$$

where $\delta^{\#}$ is an initial segment of δ, and $\gamma^{\#}$ is an end segment of γ; then both rules would be applicable to $\gamma q_l\delta$. This kind of non-determinism can be excluded in a variety of ways, for example, by ordering the rules and applying always the first applicable rule.

However, as we emphasized a number of times, Turing had intended to analyze genuine planar computations (not just string machines or letter machines operating in the plane)[8]. To formulate and prove a version of the above Reduction Lemma for planar computations, one has to specify the finite symbolic configurations that can be operated on and also the mechanical operations that can be performed. Turing recognized the significance of Post's presentation for achieving mathematical results, but also for the conceptual analysis of calculability: as to the former, Turing extended in his (1950) Post's and Markov's result concerning the unsolvability of the word-problem for semi-groups to semi-groups with cancellation; as to the latter, we will look briefly at Turing's semi-popular and most informative presentation of *Solvable and Unsolvable Problems* (1953).

---

[8] Such machines are also discussed in Kleene's *Introduction to Metamathematics*, pp. 376-381, in an informed and insightful defense of Turing's Thesis. However, in Kleene's way of extending configurations and operations, much stronger normalizing conditions are in place; e.g., when considering machines corresponding to our string machines the strings must be of the same length.

Turing starts out with a description of puzzles: square piece puzzles, puzzles involving the separation of rigid bodies or the transformation of knots; i.e., puzzles in two and three dimensions. "Linear" puzzles are described as Post systems and called *substitution puzzles*. They are viewed by Turing as a "normal" or "standard" form of describing puzzles; indeed, a form of the Church-Turing thesis is formulated as follows:

> Given any puzzle we can find a corresponding *substitution puzzle* which is equivalent to it in the sense that given a solution of the one we can easily find a solution of the other. If the original puzzle is concerned with rows of pieces of a finite number of different kinds, then the substitutions may be applied as an alternative set of rules to the pieces of the original puzzle. A transformation can be carried out by the rules of the original puzzle if and only if it can be carried out by the substitutions ... (1953, p.15)

Turing admits, with some understatement, that this formulation is "somewhat lacking in definiteness" and claims that it will remain so. In his further discussion, Turing characterizes its status as lying between a theorem and a definition: "In so far as we know *a priori* what is a puzzle and what is not, the statement is a theorem. In so far as we do not know what puzzles are, the statement is a definition which tells us something about what they are." Of course, Turing continues, one could define puzzle by a phrase beginning with 'a set of definite rules', or one could reduce its definition to that of 'computable function' or 'systematic procedure'. A definition of any of these notions would provide one for puzzles.

Even before we had seen Turing's marvelous 1953 paper, our attempts of describing mechanical procedures on general symbolic configurations had made use of the puzzle-metaphor. The informal idea had three distinct components: a computor was to operate on finite *connected* configurations; such configurations were to contain a unique *distinguished element* (corresponding to the scanned square); the operations were to *replace neighborhoods* (of a bounded number of different forms) of the distinguished element by appropriate other neighborhoods resulting in a new configuration, and such replacements were to be given by *generalized production rules*. Naturally, the question was how to transform this into appropriate mathematical concepts; referring to Turing's statement above, we were unwittingly trying to remove (as far as possible) the *lack of definiteness* in the description of general puzzles. But in contrast to Turing, we wanted to analyze deterministic procedures and follow more closely his own analysis

given in 1936. For this purpose we introduced *K-graph machines*. These machines were inspired, in part, by the analysis of algorithms given in 1958 by Kolmogorov and Uspensky.

K-graph machines operate on finite connected graphs whose vertices are labeled by symbols and that contain a uniquely labeled *central* vertex. The graphs have the crucial property of satisfying the *principle of unique location*: every path of labels (starting with the label of the central vertex) determines a unique vertex. We call these graphs *K-graphs*. K-graph machines replace distinguished K-subgraphs by other K-graphs; their programs are finite lists of generalized rules specifying such replacements. As these replacements are local, we also say that the machines satisfy the *principle of local action*. (The subtle difficulties surrounding this principle, even for the case of Post productions or string machines, are discussed in Remark 1 of the next section.)[9]

Turing machines, when presented by production systems as above, are easily seen to be K-graph machines. Conversely, the theorem in section 5 shows that computations of K-graph machines can be carried out by Turing machines. Given this mathematical analysis, Turing's central thesis is turned into the thesis that K-graph machines, clearly satisfying the boundedness and locality conditions, subsume directly the work of computors. Our main theorem thus reduces mechanical processes carried out by computors to Turing machine computations. -- We want to emphasize very forcefully that our generalization of Turing's analysis is a direct extension of the latter, both technically and conceptually. This is in striking contrast to other such generalizations, e.g., those of Friedman and Shepherdson; see (Shepherdson 1988); Gandy's penetrating analysis of *machine computability* is discussed briefly in section 4, Remark 2, and in the Concluding Remarks.

3. K-GRAPH MACHINES. To state and prove the main theorem we have to review some concepts from graph theory. A *graph* G is an ordered pair $\langle V,E\rangle$; V is the set of *vertices* of G, and E the set of *edges* of G, i.e., pairs $\{u,v\}$ of distinct vertices; for a given graph G, we denote the set of vertices also by V_G

[9] Thus, our K-graph machines are deterministic graph rewriting systems; there is a considerable literature in computer science that discusses such systems, for example, the survey article (Courcelle 1990). The category theoretic way of presenting rewrite systems is for our purposes, however, not suitable: the replacement operations have to be graphically concrete and direct, not indirectly obtainable through pushout diagrams. Tim Herron (1995) used the category theoretic framework to characterize K-graph machines.

and the set of edges by E_G. As we consider arbitrarily large finite graphs, we need a potentially infinite set of vertices; to be concrete, we choose natural numbers. Edges between u and v are denoted by uv, and the vertices u and v are called *adjacent*. For a vertex $v \in V$ and an edge $uv \in E$, we write also $v \in G$ and $uv \in G$. Given $G=\langle V,E\rangle$, $V' \subseteq V$, and $E' \subseteq \{uv \in E \mid u,v \text{ both in } V'\}$ we call $G'=\langle V', E'\rangle$ a *subgraph* of G and write $G' \subseteq G$; we define $G \cup G' = \langle V \cup V', E \cup E'\rangle$. We sometimes denote a function $f:V \to V'$ by $f:G \to G'$ and may refer to f as a function on graphs. f is *edge preserving* if $\forall u,v \in V[uv \in E \Leftrightarrow f(u)f(v) \in E']$. A *path in* G from u_1 to u_n is a sequence $u_1u_2...u_n$ of distinct vertices of G such that for every pair of consecutive vertices u_i and u_{i+1} the edge u_iu_{i+1} is in G. A vertex v *belongs to the path* if v is an element of the sequence; an edge uv belongs to the path when u and v are consecutive vertices in the sequence. The *length* of a path is defined as the number of edges belonging to the path; $len(u,v)$ is the length of a shortest path from u to v, if any path from u to v exists. A *component* of a graph G is a maximal subgraph G' of G such that for any two vertices u and v in G', there is a path in G' from u to v. G is called a *connected* graph if $G=G'$.

In the same way in which symbols (or rather tokens of symbols) are written on the tape of a Turing machine, we will "write" symbols from some alphabet $\mathbf{L}$ on the vertices of graphs. An $\mathbf{L}$-*labeled graph* K is an ordered pair $\langle G,Lb\rangle$, where G is a graph and Lb is a function from V_G to $\mathbf{L}$; Lb is a *labeling* of G. For a distinguished element $*$ of $\mathbf{L}$, an $\mathbf{L}_*$-labeled graph G is an $\mathbf{L}$-labeled graph that contains exactly one vertex with label $*$; this vertex is called the *central vertex of* G and is referred to simply as $*$ when the context is clear. For an $\mathbf{L}_*$-labeled graph K, K^* is the (unique) component of K containing $*$. For $\mathbf{L}$-labeled graphs $K=\langle G,Lb\rangle$ and $K'=\langle G',Lb'\rangle$, $f:K \to K'$ is *label preserving*, if $\forall v \in G[Lb'(f(v))=Lb(v)]$. A label and edge preserving bijection $f:K \to K'$ is an isomorphism; if there is an isomorphism between K and K' we write $K \approx K'$. $K \lesssim K'$ abbreviates that K is isomorphic to a subgraph of K'. For ease of presentation, we may use the term *graph* to apply to graphs or to $\mathbf{L}$-labeled graphs: we can redefine for $\mathbf{L}$-labeled graphs all of the above notions for graphs, extending and restricting the labeling functions in obvious ways. Finally, we come to a notion that is special for our purposes: a sequence α of labels associated with a path from the central vertex $*$ to some vertex v is called a *label-sequence* for v; the set of such sequences is denoted by $Lbs(v)$. If labeled graphs have the property that, for any vertex v, a label sequence from

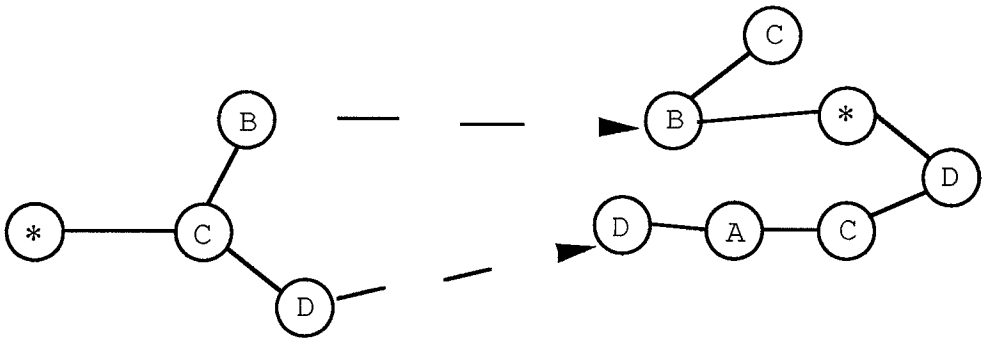
$Lbs(v)$ labels a path to v and not to any other vertex, then the labeling provides a *coordinate system.* Notice, however, that each vertex may have a number of different "coordinates". This leads to the following definition:

Definition. A connected $\mathfrak{L}_*$-labeled graph K is a *Kolmogorov-graph,* or *K-graph,* over $\mathfrak{L}$ if and only if $(\forall\alpha)\ (\forall u,v\in K)[\alpha\in Lbs(u)\cap Lbs(v) \Rightarrow u=v]$.

We refer to the above property of graphs as the *principle of unique location;* this principle and its relation to condition (α) in Kolmogorov and Uspensky's work is discussed in remark 2 below, as well as the fact that isomorphisms between K-graphs are uniquely determined. -- K-graphs constitute the class of finite symbolic configurations on which our machines operate, and we describe now what elementary operations are allowed on such configurations. The operations take the form of generalized production rules and are directly motivated as puzzle-replacement operations.

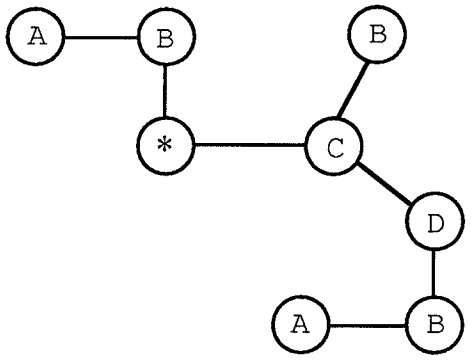
Definition. A *rule* R is an ordered triple $\langle A,C,\phi\rangle$, where (R's *antecedent*) A and (R's *consequent*) C are K-graphs and ϕ is an injective, label preserving partial function from A to C. For a given rule R we let A_R be A, C_R be C, and ϕ_R be ϕ.

Below is an example of a rule, where the dotted arrows indicate ϕ. We use capital letters to indicate the elements of $\mathfrak{L}$ associated with each vertex.

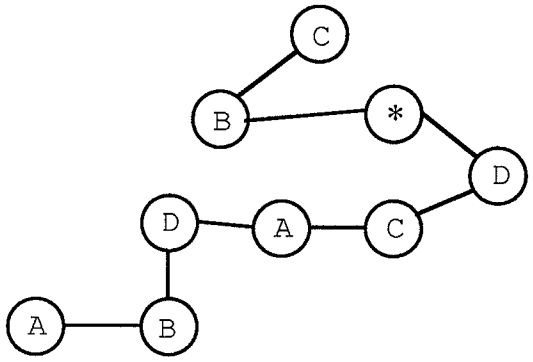


A C

The rules are production rules that allow us to replace a (subgraph of a) K-graph that is isomorphic to the antecedent A of a rule by that rule's consequent C. For example, if we apply the above rule to the following K-graph K:



we get the K-graph:



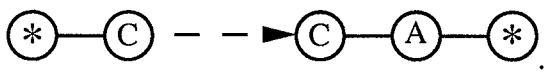
The rule application removes, first of all, the part of K that is isomorphic to A. Then C is connected to the resulting graph by drawing, for each vertex v in the image of ϕ, an edge between v and the vertex adjacent in K to $\phi^{-1}(v)$. (The vertices of K that remain unconnected to $*$ may be removed.) Of course, once A is removed from K, we assume that we are taking a disjoint union of the remaining vertices with the vertices of C. As some of the vertices of C may have the same names as vertices in K, we may have to rename the vertices of C and obtain an isomorphic graph C'. Extending this idea, we can simplify the removal of the subgraph isomorphic to A, by renaming A to some isomorphic A' that actually is a subgraph of K. Finally, we can choose the renaming of C to C' in such a way that ϕ' (the function resulting by renaming the vertices in ϕ according to the renamings from A to A' and C to C') is the identity on its domain. We can make the renaming "canonical" by choosing always the least natural number not yet used to rename the least remaining vertex of C. This discussion of the steps involved in computations of a K-graph machine motivates, we hope, the following definitions.

Definition. (i) A rule R *renames to* R' ($R \approx > R'$) if there exist isomorphisms $\psi_1: A_R \to A_{R'}$ and $\psi_2: C_R \to C_{R'}$ such that $\phi_{R'} \bullet \psi_1 = \psi_2 \bullet \phi_R$.
(ii) A rule R is *applicable* to a K-graph K iff $A_R \lesssim K$.
(iii) Let $R=\langle A, C, \phi\rangle$ be applicable to K-graph K, let $R'=\langle A', C', \phi'\rangle$ be given such that $R \approx > R'$, ϕ' is the identity on its domain, $A' \subseteq K$, and $(K-A') \cap C' = \varnothing$. Then the *result of applying* R to K, $R(K)$, is given by

$$V_{R(K)} = (V_K - V_{A'}) \cup V_C \quad \text{and} \quad E_{R(K)} = (E_K \cup E_{C'}) \restriction V_{R(K)},$$

where the latter operation indicates the "restriction" of the set of edges to those with vertices in $V_{R(K)}$.

Allowing non-deterministic computations means allowing different rules to have the same antecedent. But even if we require that all rules in a given set have different antecedents, it may still be (as in the case of the string machines in section 2) that a number of different rules are applicable to a given state K. For example, if the above rule is applicable to some K, then the following rule is also applicable to K:



To avoid this kind of non-determinism, we order the rules linearly and always use the first (in that ordering) applicable rule. The ordinary kind of non-determinism is excluded by requiring that for Q, R in a rule sequence $\mathfrak{R}$: if $A_Q \approx A_R$, then $Q=R$. A sequence of rules satisfying this condition is called a *program*. Finally, having defined the structures on which our machines operate and the steps they can take, we define the machines themselves.

Definition. Let $\mathfrak{L}$ be a finite alphabet with a distinguished element $*$ and let $\mathfrak{M}=\langle \mathfrak{S}, \mathfrak{F}\rangle$, where $\mathfrak{S}$ is the set of all K-graphs over $\mathfrak{L}$ and $\mathfrak{F}$ is a partial function from $\mathfrak{S}$ to $\mathfrak{S}$. $\mathfrak{M}$ is a *K-graph Machine over* $\mathfrak{L}$ if and only if there is a program $\mathfrak{R}=\langle R_0, \ldots, R_n\rangle$ such that:

For every $S \in \mathfrak{S}$, if there is an $R \in \mathfrak{R}$ that is applicable to S, then $\mathfrak{F}(S) = (R_i(S))^*$, where $i = \min\{j \mid R_j \in \mathfrak{R} \text{ is applicable to } S\}$; otherwise $\mathfrak{F}$ is undefined for S.

The elements of $\mathfrak{S}$ are the *states* of $\mathfrak{M}$, and $\mathfrak{F}$ is the machine's *transition function*. We say that $\mathfrak{F}$ satisfies the *principle of local action*. -- We can give an obviously equivalent definition of K-graph machines that brings out the special principles more directly. Let $\mathfrak{M}=\langle\mathfrak{S},\mathfrak{F}\rangle$, where $\mathfrak{S}$ is a set of $\mathbf{L}_*$-labeled graphs and $\mathfrak{F}$ is a partial function from $\mathfrak{S}$ to $\mathfrak{S}$; $\mathfrak{M}$ is a *K-graph Machine over* $\mathbf{L}$ if and only if (i) $\mathfrak{S}$ is the largest set of $\mathbf{L}_*$-labeled graphs that satify the principle of unique location, and (ii) $\mathfrak{F}$ satisfies the principle of local action. (For a less restrictive formulation see Remark 1 in section 4.)

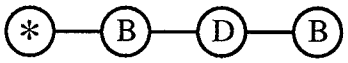
Remarks. 1. *Turing's conditions.* K-graph machines clearly satisfy the determinacy condition (cf. 2 below), but also the boundedness and locality conditions -- when those are suitably interpreted: the number of "immediately recognizable" symbolic configurations is given by the number of distinct antecedents and consequents of the machine's program; operations are quite properly viewed as modifying observed configurations, and observed labeled vertices lie always within a fixed "radius" around the central vertex. (The radius can be read off from the program, e.g., it can be taken to be the maximal length of paths in any K-graph of the program.) We make some additional remarks about the principle of local action, as it might be thought that -- even in the case of string machines -- locality is violated! The reason being, that in an "implementation" of those machines, e.g., on a standard Turing machine, the total tape content is affected when using a rule that replaces a string by either a longer or a shorter one. This seems to be pertinent only if the tape has a rigid extrinsic coordinate system as given, for example, by the set of integers $\mathbf{Z}$. When a different presentation of Turing machines is chosen, as suggested for example in (Gandy 1980), or when the underlying structure is flexible to insertions, as in our set-up, the concern disappears.[10] It is precisely the use of an *intrinsic coordinate system*, guaranteed through the principle of unique location, that makes for the locality of the replacement operations.

2. *The principle of unique location.* In the brief discussion preceding the definition of K-graph machines, we mentioned two sources of non-determinism present already for letter and string machines (and the standard way of circumventing them). However, there is one additional source, when

[10] These two ways of dealing with the issue are two sides of the same coin.

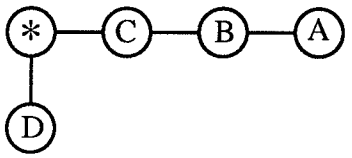
graphs, even labeled ones with a central vertex, are considered as the structures on which production rules operate: if the antecedent of a rule can be embedded into a given graph, it usually can be done in a variety of ways; the result of the rule application will in general depend on the chosen embedding. It is precisely this kind of indeterminacy that is excluded by the principle of unique location, as it guarantees that there is a unique embedding, if a K-graph can be embedded into another K-graph at all. The principle is also related to a second conceptual issue, namely, immediate recognizability. For any K-graph machine $\mathfrak{M}$ and any K-graph whatsoever, we can decide in constant time, whether any rule of $\mathfrak{M}$'s program is applicable. Mathematically, the principle is exploited for the reduction in section 5. It guarantees that, for any $\mathfrak{L}_*$-labeled graph, paths starting at the central vertex are uniquely characterized by the sequence of symbols labeling their vertices. Thus, using the lexicographical ordering on strings of labels from $\mathfrak{L}$, we can choose for each vertex a unique address which picks out that vertex in terms only of labels.

Kolmogorov and Uspensky used their condition (α) for similar purposes. That condition is formulated in our setting as follows: For every graph $S \in \mathfrak{S}$, if u and v are vertices of S both adjacent to some vertex w of S, then $Lb(u) \neq Lb(v)$. We call a connected $\mathfrak{L}_*$-labeled graph satisfying condition (α) a *Kolmogorov complex over* $\mathfrak{L}_*$ or, briefly, a K-complex. Condition (α) implies our principle of unique location, but is not implied by it. The first claim is easily established; for the second claim one sees directly that the following graph K' is an example of a K-graph that is not a K-complex.

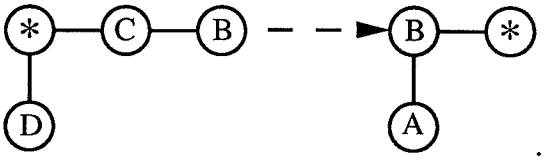


Thus, the principle of unique location is strictly weaker than condition (α).

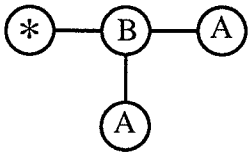
3. *Preservation under rule application.* We require that $\mathfrak{F}$ is a partial function from $\mathfrak{S}$ to $\mathfrak{S}$. As matters stand, programs and K-graph machines cannot be "identified"; the reason is this: not every rule, when applied to a K-graph, yields a K-graph. For example, consider the K-graph K



and the rule R



The application of R to K yields the graph:



$R(K)$ is clearly is not a K-graph; just note that there are two distinct vertices both having label sequence *BA.

It is straightforward to modify any given machine program $\mathfrak{R}$ to a program $\mathfrak{R}'$ such that $\mathfrak{R}$ and $\mathfrak{R}'$ yield the same result when $\mathfrak{R}$ transforms a K-graph into a K-graph, but $\mathfrak{R}'$ makes the machine diverge on K-graph inputs that $\mathfrak{R}$ transforms into graphs not satisfying the principle of unique location. Given this modification, $\mathfrak{R}'$ defines a unique partial function from $\mathfrak{S}$ to $\mathfrak{S}$. -- Kolmogorov and Uspensky required a particular structure on rules preserving condition (α). As one has to be careful only about the symbols adjacent to vertices in the image of ϕ, they imposed in effect the following condition (β) for a given rule $\langle A, C, \phi\rangle$:

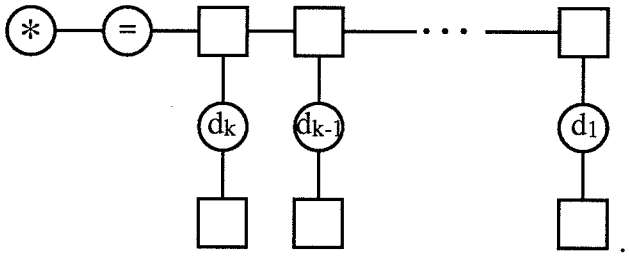
(β) $(\forall y \in C)\ \{(y=\phi(x)\ \&\ vy \in C) \Rightarrow [Lb(v)=*$ or $(\exists w)(wx \in A$ and $Lb(w)=Lb(v))]\}$.

Consequently, if a program $\mathfrak{R}$ satisfies (β), then $\mathfrak{M}=\langle\mathfrak{S},\mathfrak{F}\rangle$ is a K-graph machine, where $\mathfrak{S}$ is the set of all K-graphs on $\mathfrak{L}$ and $\mathfrak{F}$ is the unique function on $\mathfrak{S}$ defined by $\mathfrak{R}$.

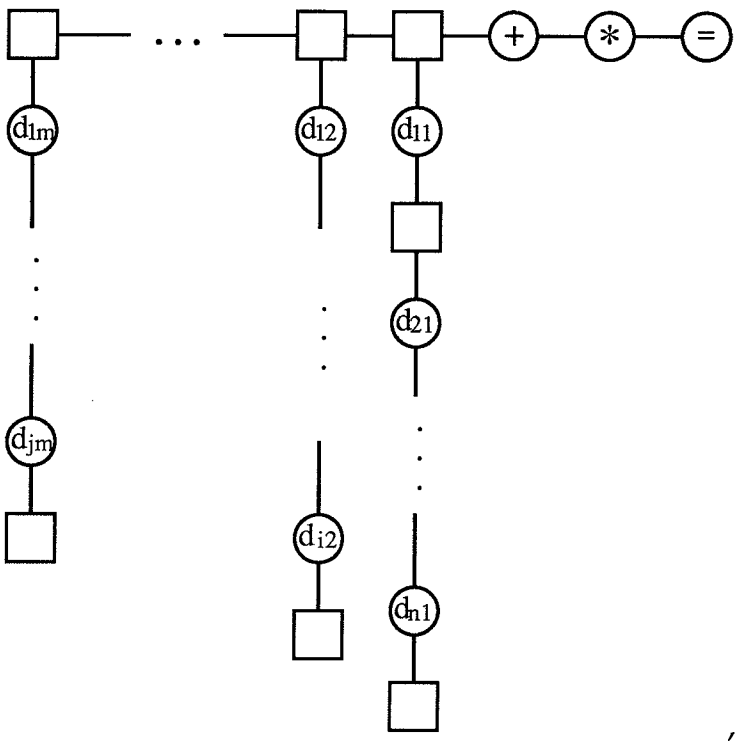
4. **SUBSUMPTIONS.** In this section we indicate first that standard planar algorithms (with minor normalization) are K-graph machine computations; we

do this by considering an example that might have appeared in Turing's "arithmetic book for children". Then, we show that a variety of computation models fall directly under, or are subsumed by, our K-graph machine concept.

A *planar addition problem* consists of a set of n integers, each of at most m digits (when expressed in decimal form) and at least one of which has exactly m digits. We present a K-graph machine which carries out an algorithm for adding these integers similar to the familiar algorithm that a child is taught in elementary school. We assume that the integers are expressed in decimal form and arranged in order of nonincreasing length. We use the symbols '+' and '=' to differentiate between the remaining part of the problem and the current partial sum. A box vertex is used as a buffer between digits so that we can manipulate the structure independently of the value of those digits that are not currently being scanned. Let d_{ij} be the jth digit of the ith integer (counting digits from the right). Their sum, $d_k d_{k-1} \ldots d_1$, will be given in the form:

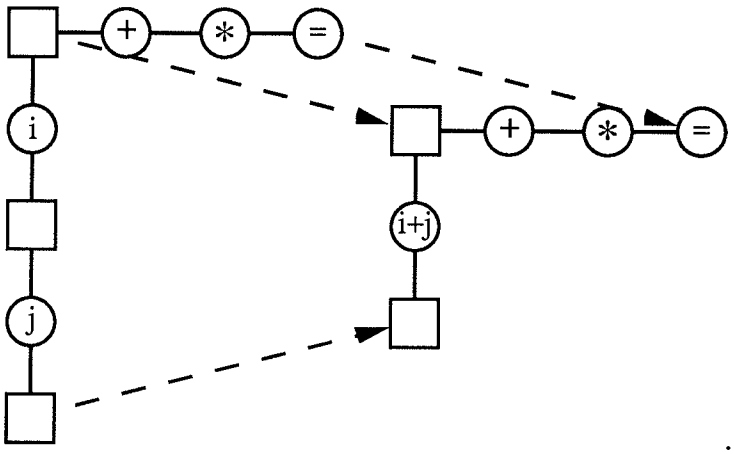


The problem itself is presented by:

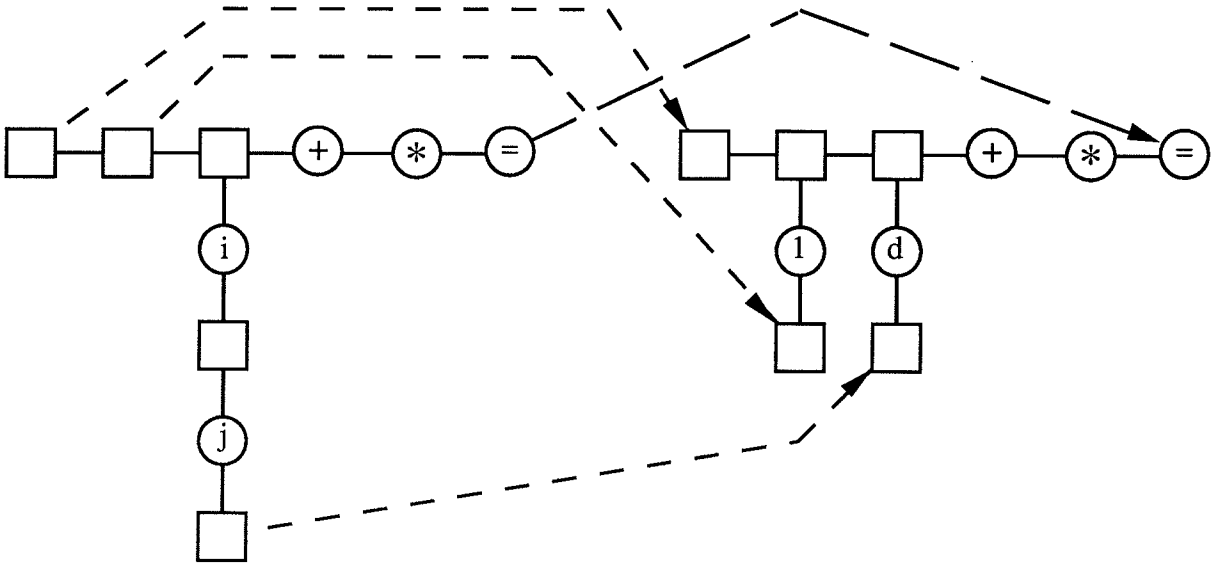


where only the first j-many numerals have m-digits, only the first i-many numerals have at least 2 digits, and so on.

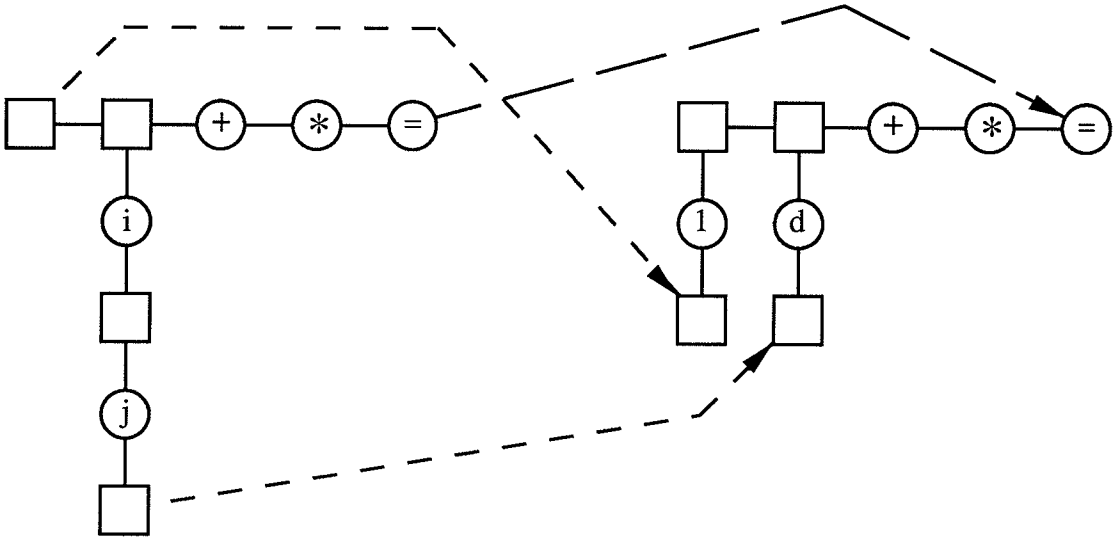
The algorithm collapses each column, two digits at a time, into the single digit which should appear below that column in the sum. The rules must carry digits into the next column, move from the current column to the next, and store the partial sum when we shift columns. We present the rules in the order in which they appear in the program. For each $i,j \in \{0,...,9\}$, $i+j<10$, there is a rule which replaces i and j with the sum $i+j$:



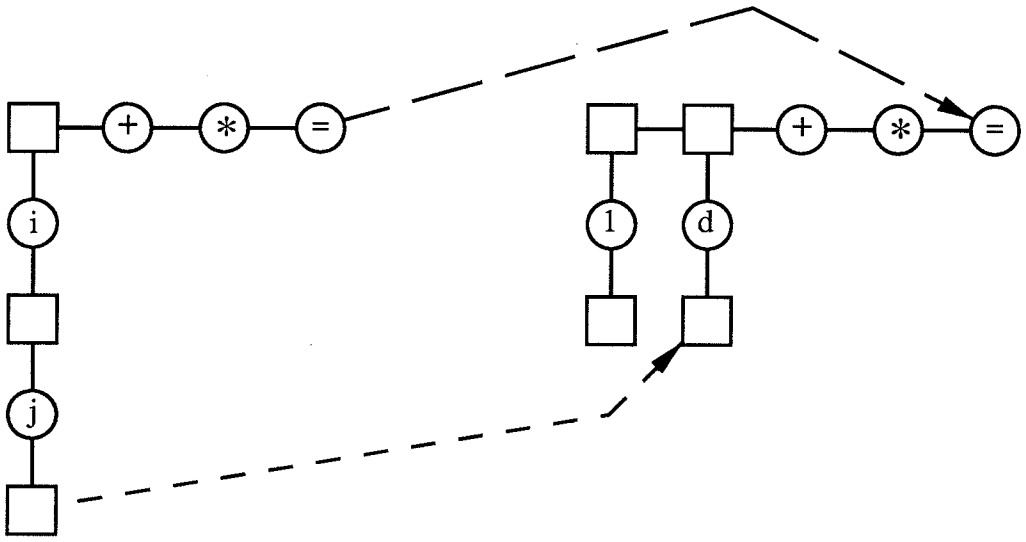
For each $i+j \geq 10$, there are 3 rules. These rules must carry a '1' to the next column to the left and replace i and j with $d=i+j-10$. The first rule is for the case when at least two columns occur to the left of the current column--here we carry the '1' to the top of the first column to the left and maintain the connection to the second column.



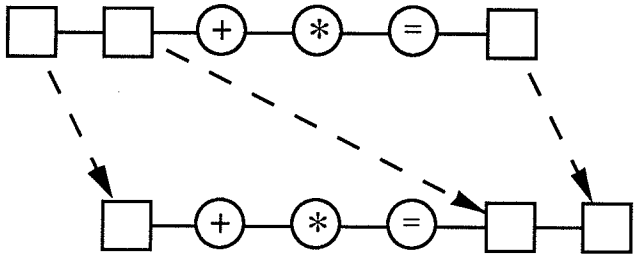
The second rule treats the case when only one column occurs to the left (and hence no connection has to be maintained).



A third rule is needed for the case that the current column is the only one; in this case a new column must be created.



Finally, we need rules for moving to the next column and storing the result from the current column. These rules are ordered last, and hence will fire just in case there is a single digit in the current column. Each of these rules will take the current column and move it to the immediate right of the '=' sign. If there is a column to the left of the current one, then it becomes the current column; if there is a column to the right of the '=' sign, then we must connect the current column to it. There are four rules, as either of these conditions may or may not apply. For example, when both conditions apply, we have the following rule.



Not only are particular informal algorithms, like this addition example, directly seen to be K-graph computations, but classes of algorithms presented as computations of particular computation models are subsumed under K-graph computations. We start this discussion by showing that *Turing's machines are K-graph machines.* To do that we make a slight modification in the definition of K-graph machines; this is done only for convenience (and to indicate the flexibility of the notion).

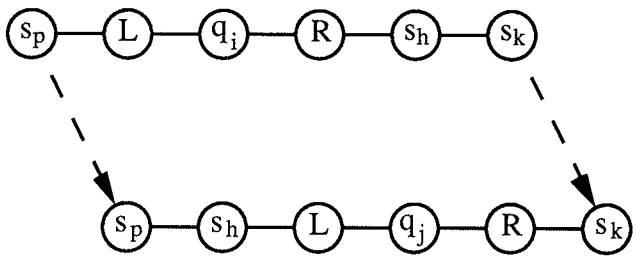
Remark 1. We supplement the finite alphabet $\mathfrak{L}$ now by a set Q of "states"; $\mathfrak{L}_Q$-labeled graphs are labeled with elements from $\mathfrak{L}$, and exactly one vertex is labeled by an element from Q. The $\mathfrak{L}_*$-labeled graphs defined above are the $\mathfrak{L}_{\{*\}}$-labeled ones according to the present definition. If in addition the Q-label is surrounded by "direction indicators" L and R in the one-dimensional case, by pairs $\alpha\beta$ (where α and β are either L or R) in the two-dimensional case, etc., we call the resulting graph a ${}^+\mathfrak{L}_Q$-labeled one.

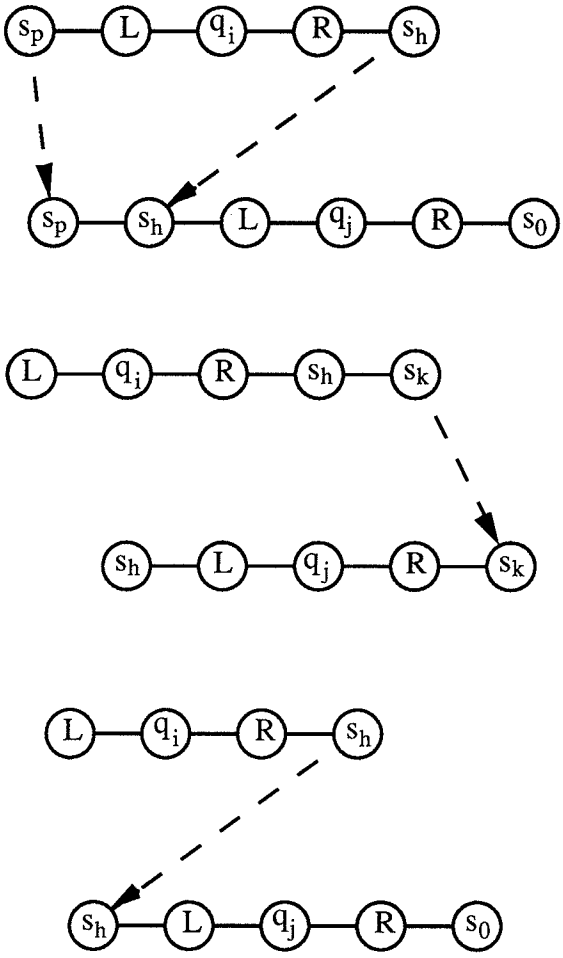
For an arbitrary Turing machine **M** we define a K-graph machine $\mathfrak{M}$ which simulates **M**. Let $q_0,\ldots,q_n$ be the states of **M**, and $s_0,\ldots,s_m$ be the symbols which can appear on the tape. Let $\mathfrak{S}$ be the set of all ${}^+\mathfrak{L}_Q$-labeled K-graphs. We represent the instantaneous description

$$s_{i_0}s_{i_1}\ldots s_{i_j}q_k s_{i_{j+1}}\ldots s_{i_l}$$

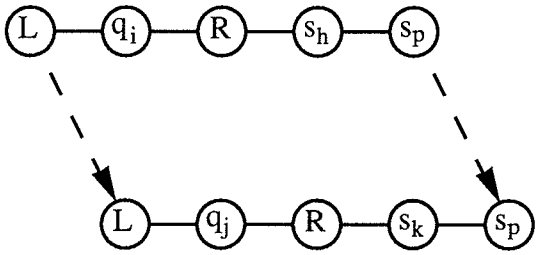
of **M** by the following K-graph (thus without the square-buffers that were used in the addition example):

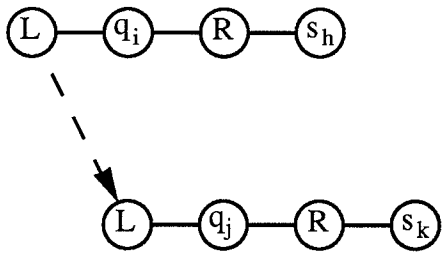
L and R indicate left and right with respect to the central Q-vertex. Note that without L and R the tape has no orientation; with L and R the principle of unique location is automatically satisfied. -- For each command $q_i s_h R q_j$ in **M** (and each s_p and s_k in $\mathfrak{L}$) we have rules in $\mathfrak{M}$, reflecting the different possible contexts, as follows:





The L-commands are treated similarly. -- For each command $q_i s_h s_k q_j$ in **M** (and every s_p in $\mathfrak{L}$) we have these rules in $\mathfrak{M}$:





$\mathfrak{M}$ "simulates" **M** -- one computation step of $\mathfrak{M}$ corresponding to one of **M**. Consequently, the notion of K-graph machine directly generalizes the notion of Turing machine.

THEOREM. *Turing machines are subsumed under K-graph machines.*

These considerations can be carried out in a similarly direct way for string machines and their generalizations to higher dimensions, thus in particular for the generalized Turing machines described in (Kleene 1952). Using the observations in Remark 2, section 3, *Kolmogorov machines* as defined in (Uspensky and Semenov) are also K-graph machines. For that we have only to observe that such machines operate on Kolmogorov complexes of a particular radius (the "active zone" of the complex) and that the immediate transformations specifying the program of a machine satisfy (β). -- For a host of other models of computations one can show that they are also subsumed under K-graph machines, for example, the register machines introduced by Shepherdson and Sturgis. Joining these observations with the main result of the next section, we have an *absolutely uniform way of reducing computations of a particular model to Turing machine computations*: We have only to verify that the computation model is subsumed under K-graph machines.

Remark 2. For Turing the ultimate justification for his restrictive conditions lies in the *necessary limitation* of human memory, and that can be directly linked to physical limitations also for machines (cf. Mundici and Sieg, section 3). Church in his review of Turing's paper seems to have mistaken Turing's analysis as an analysis of machine computations. Church's apparent misunderstanding is common: see, for example, Mendelson (1990). So it is worthwhile to point out that machine computability was analyzed only much later in (Gandy 1980). Turing's three-step-procedure of analysis, axiomatic

formulation of general principles, and proof of a reduction theorem is followed there, but for "discrete deterministic mechanical devices". Gandy showed that everything computable by a device satisfying his principles, a *Gandy machine*, can already be computed by a Turing machine. To see clearly the difference between Turing's analysis and Gandy's, note that Gandy machines incorporate parallelism: they compute directly Conway's game of life and operate, in parallel, on bounded parts of symbolic configurations of possibly unbounded size. The boundedness conditions for Gandy machines and, in particular, the *principle of local causation* are motivated by physical considerations.

5. SIMULATION THEOREM. We reduce computations of K-graph machines to computations of Turing machines; more precisely, for an arbitrary K-graph machine $\mathfrak{M}=\langle \mathfrak{S},\mathfrak{F}\rangle$ over the language $\mathbf{L}$ we construct a Turing machine[11] **M** over the alphabet {0, 1} that simulates $\mathfrak{M}$. The simulation requires (i) that we give linear representations of K-graphs, and (ii) that we show for every $S\in\mathfrak{S}$, if $\sigma=S,S_1,\ldots,S_n$ is a computation of $\mathfrak{M}$, then $\tau=T_0,T_1,\ldots,T_m$ is a computation of **M**. Here T_0 represents S, and T_m represents a K-graph isomorphic to S_n; furthermore, there exists a subsequence $T_{i_1},T_{i_2},\ldots,T_{i_{n-1}}$ of τ, such that for $1\leq j<n\text{-}1, T_{i_j}$ represents a K-graph isomorphic to S_j. (The conditions for infinite computations are similar.) The subsequent considerations establish:

THEOREM. *Any K-graph machine $\mathfrak{M}$ can be simulated by some Turing machine* **M**.

Let $<$ be a linear ordering on $\mathbf{L}$ and $<'$ the lexicographical ordering on finite sequences of symbols from $\mathbf{L}$ induced by $<$. The ordering $<$ on finite sequences of symbols is defined by $\alpha<\beta$ iff α is shorter than β or α and β are of the same length and $\alpha<'\beta$. For a given vertex v in V the *address* $Ad(v)$ *of* v is the $<$-minimal element of $Lbs(v)$. By connectedness such an address exists, and by the principle of unique location Ad is injective.

Definition. For an arbitrary edge $uv\in E$ we define the *location description* $LD(uv)$ by:

---

[11] In contrast to our earlier discussion, we are going to use a Turing machine whose tape is extendable only to the right.

$$LD(u\ v) = \begin{cases} \langle u, Lb(u), v, Lb(v) \rangle & \text{if} \quad Ad(u) < Ad(v) \\ \langle v, Lb(v), u, Lb(u) \rangle & \text{if} \quad Ad(v) < Ad(u) \end{cases}$$

When we refer to an edge uv we assume from now on that $Ad(u)<Ad(v)$. Let G be the graph whose LDs appear on the tape; we say that the tape *represents* the K-graph G^*. In order to encode the graph on the tape initially, we have to choose a particular way of proceeding, for example, by exploiting the ordering of the location descriptions. We obtain a canonical *graph representation* $GR(K)$ for a given K-graph K by ordering $LD(E)=\{LD(uv) \mid uv \in E\}$ as follows: $LD(u_1v_1)<LD(u_2v_2)$ iff $Ad(u_1)<Ad(u_2)$ or $u_1=u_2$ and $Ad(v_1)<Ad(v_2)$. $GR(K)$ is thus the <-ordered sequence of location descriptions for K.

To obtain a tape representation of this sequence in the binary alphabet {0,1}, we assign a natural number to each of the symbols in $\mathbf{L}$, using 0 for the <-least element of $\mathbf{L}$, 1 for the next element (under <), and so on. Of course, every vertex $v \in V$ is already a natural number, and we assume without loss of generality that a state S with n vertices consists of {1,...,n}; $Lb(v) \in \mathbf{L}$ is a natural number as well. We represent natural numbers in a modified binary form obtained from the standard one by replacing every 1 with 11 and every 0 with 10. We assume that the initial encoding of a sequence of numbers separates them by exactly two 0's; a sequence of sequences of numbers is obtained by placing three 0's between each coded sequence.

We describe the program for a Turing machine **M** that is to simulate $\mathfrak{M}$ and assume that **M** has states $q_0,\ldots,q_n$. The program transforms any state S into $\mathfrak{F}(S)$, then it returns to its initial Turing state, and, if possible, further transforms the resulting state $\mathfrak{F}(S)$; the machine halts, when none of the rules defining $\mathfrak{F}$ can be applied. (It should be obvious that this yields the kind of simulation indicated above.) In the following, we will always use S to refer to the graph currently coded on the tape, even though there are stages when some edges are removed and others are added.

The rules $R_0,\ldots,R_{r-1}$ in the program $\mathfrak{R}$ computing $\mathfrak{F}$ are encoded into the program of **M**.[12] For a given rule R the antecedent A_R and the consequent C_R are encoded by $GR(A_R)$ and $GR(C_R)$. In order to simplify renaming of vertices, we assume that for every rule R, ϕ_R is already the identity function. We are only concerned with the isomorphism class represented by each rule, so we are free to encode R_i by any R that is isomorphic to R_i $(0\leq i<r)$; we modify rules as follows: let N be the maximum number of vertices occurring in any rule and replace each rule R_i by an isomorphic R_i' such that for every vertex v of R_i', $N\cdot i<v\leq N\cdot(i+1)$. $\mathfrak{R}$ will indicate now the modified program $R_0',\ldots,R_{r-1}'$. We try to apply each rule of $\mathfrak{R}$ in the given order to the tape and execute the first applicable one; the tape contains then all the edges of $\mathfrak{F}(S)$. We repeat this until none of the rules is applicable to the state represented on the tape.

Initialize the tape. S is put on the tape; each vertex v in S is replaced by $v+M$ [where $M=N\cdot r$] so that all (vertex-) numbers of S are greater than M. This has the following advantage: by looking at a vertex we can determine immediately, whether it occurred in the original state or whether it was written by a rule; in the latter case we can decide which rule wrote it. Furthermore, $C_R\cap(S-A_R)=\varnothing$ for any R.

Find the appropriate rule. We examine each rule in the given order and define a subroutine CHECK-R that determines, whether R can be applied to S. If R is applicable, then CHECK-R transforms the tape to contain $S'\approx S$ with $A_R\subseteq S'$; i.e., the algorithm tries to modify S to an isomorphic S', such that A_R is a subgraph of S'; we say A_R is *matched with* a subgraph of S. If R is found not to be applicable, CHECK-R may nevertheless have changed some vertices on the tape, but the modified graph is isomorphic to S.

CHECK-R starts with the head leftmost on the tape and proceeds by moving to the right, searching for appropriate edges one at time. Let *min* be the least and *max* the greatest vertex of R. In the ith step, we are looking for an edge in S which matches the ith edge uv of A_R. u is the second vertex of some edge (already matched to an edge in S) and occurs consequently in S. We search for w in S, such that the edge uw is in S, and such that

[12] An alternative to this simulation is the following: We can also represent a rule on the tape simply by writing $GR(A)$ followed by a separator (say, '0000') and then by $GR(C)$. To represent the program we represent each rule in order, separating them by, say, '00000'. Then, we can describe a Turing machine **U** that simulates any K-graph machine conceptually in exactly the same way in which a universal Turing machine simulates any other Turing machine **M**. In the latter case the input to the machine is a Gödel number of **M** and an input to **M**. Here, **U** will take as input the coded program for the K-graph machine followed by the initial state. Then **U** will carry out systematically the coded program.

$Lb(w)=Lb(v)$. If such a vertex is found, we distinguish two cases: (i) if $min \leq w \leq max$ and $w \neq v$, then w was written by R and is the image of some vertex in A_R other than v, and the search for the correct vertex has to be continued; (ii) if $w<min$ or $w>max$ or $w=v$, then w is the correct vertex, and we substitute v for w everywhere on the tape and proceed to search for the next matching edge in S.

In sum, if we reach the end of the tape before finding such a vertex, we fail for this R; if CHECK-R fails for every rule R, the machine halts; if CHECK-R finds a matching edge for every edge in A_R, R is applicable. (Note that the principle of unique location allows us to avoid backtracking in case the algorithm fails for a particular vertex.)

Apply the appropriate rule. For each rule R there is a subroutine APPLY-R which applies R to the current state: all edges which contain any vertex from $V_{A_R}-V_{C_R}$ are erased and all those from C_R are inserted -- leftmost onto the tape (in the order of their appearance in the canonical encoding $GR(C_R)$). The tape contains now a representation of the next state $\mathfrak{F}(S)$. Finally, the head is returned to the left end of the tape, and the state is set to q_0. -- This concludes the description of the Turing machine simulation of K-graph machine computations.

We want to determine now, in a rough way, the number of steps **M** needs to transform a K-graph S with n vertices into $\mathfrak{F}(S)$. Assume that the language for $\mathfrak{M}$ contains l symbols, and that $\mathfrak{M}$'s program has r rules of size at most N (i.e., at most N vertices occur in $A \cup C$). For a given state S with n vertices, the maximal degree of each vertex is $l+1$; otherwise, the principle of unique location would be violated. Thus at most $\frac{n(l+1)}{2}$ edges have to be represented. The largest (vertex-) number to be represented is n, which has length $2\log n$ in our modified binary notation. The largest LD has length of order $\log(2n+2l)$. Thus the representation of S is of length $O(n\log n)$.

The renumbering step must traverse the entire tape. Since we wish to increase each vertex by at least M, we take M' to be the least power of two greater than or equal to M and add M' to each vertex. This operation requires shifting all of the cells right of the vertex being updated up to $\log M'$ cells to the right. This requires rewriting up to $O(n\log n)$ cells for each vertex. Since this operation is done to all occurrences (of which there may be up to $l+1$

many) of each of the n vertices, it is an $O(n^2\log n)$ operation. (The rewriting itself can be done in a single pass over the number and requires $\log n$ steps.)

The further rewriting operations required for finding the applicable rule all involve replacing numbers greater than M' by numbers smaller than M'; thus, no shifting is involved in these operations, since we allow extra 0's to occur between integers. Attempting to match a given edge in a rule to one on the tape might require looking at the entire tape and is an $O(n\log n)$ operation; that may have to be done for every edge in every antecedent.

If we succeed in finding an applicable rule, we apply it; i.e., we transform the tape by erasing all edges from A_R and inserting all edges from C_R at the beginning of the tape. This may require shifting all $O(n\log n)$ symbols by at most the length of the largest $GR(A_R)$. Hence only $O(n\log n)$ many steps are required for rule selection and application. But once this has been accomplished, the entire transformation is complete, so the complexity of the simulation is $O(n^2\log n)$.

Now let us consider simulating a full computation of $\mathfrak{M}$. If we let $k=\max\{|C_R|-|A_R| \mid R\in\mathfrak{R}\}$, then for any $S\in\mathfrak{S}$, $|\mathfrak{F}(S)|\leq|S|+k$; here $|K|$ is the cardinality of the set of vertices of K. Let h be a natural number, such that $hn^2\log n$ is the complexity of the "step-simulation" for **M** of $\mathfrak{M}$ we just discussed. Assume, in a first example, that $\mathfrak{M}$ runs in constant time, say, in m steps. Then the length of the computation of **M** for input of size n is bounded by

$$s=hn^2\log n+h(n+k)^2\log(n+k)+\ldots+h(n+km)^2\log(n+km).$$

Clearly,

$$n^2\log n\leq s\leq mh(n+km)^2\log(n+km)=O(n^2\log n),$$

so $s=O(n^2\log n)$.

If $\mathfrak{M}$ runs in higher order time, however, the step-complexity of **M** is not preserved. Assume, for example, that $\mathfrak{M}$ runs in mn^c-many steps, for some m and c. Then the complexity of **M** for input of size n is bounded by

$$s=hn^2\log n+h(n+k)^2\log(n+k)+\ldots+h(n+kmn^c)^2\log(n+kmn^c)$$

$$=\sum_{j=0}^{mn^c}h(n+kj)^2\log(n+kj).$$

Clearly, $\sum_{j=0}^{mn^c}hj^2\log n\leq\sum_{j=0}^{mn^c}h(n+kj)^2\log(n+kj)\leq\sum_{j=0}^{mn^c}h(n+kj)^2\log(n+kmn^c)$.

On the left, we have:

$$\sum_{j=0}^{mn^c} hj^2 \log n = h \log n \sum_{j=0}^{mn^c} j^2$$
$$= h \log n \left[\tfrac{1}{6}\left(2(mn^c)^3 + 3(mn^c)^2 + mn^c\right)\right]^{13}$$
$$= O(n^{3c} \log n).$$

On the right:

$$\sum_{j=0}^{mn^c} h(n+kj)^2 \log(n+kmn^c) = h \log(n+kmn^c) \sum_{j=0}^{mn^c} (n+kj)^2$$
$$= h \log(n+kmn^c) \sum_{j=0}^{mn^c} (n^2 + 2nkj + k^2 j^2)$$
$$= h \log(n+kmn^c) \left[n^2(mn^c+1) + 2nk \frac{mn^c(mn^c+1)}{2} + k^2 \frac{(2mn^c)^3 + 3(mn^c)^2 + mn^c}{6} \right]$$
$$= O(n^{3c} \log n).$$

Thus s is $O(n^{3c} \log n)$.

This analysis illustrates the quite obvious point that the complexity of computing a given function depends on the machine used to carry out the computation. An interesting example is multiplication: it can be computed in linear time by RAM's and SMM's (Schönhage), but it is also known that Turing machines cannot multiply in linear time (Cook and Aanderaa; Paterson e.a.); we do not know, whether K-graph machines can. This question and related ones raise many interesting issues about complexity, particularly whether one model allows a more fundamental analysis of the complexity of algorithms than another. (These matters, as well as connections of K-graph machines to other general models, e.g., Friedman and Shepherdson's computations on arbitrary structures, the evolving algebras of Gurevich, will be dealt with in a future paper.)

CONCLUDING REMARKS. We have been concerned with an explication and generalization of Turing's arguments for his thesis, i.e., the claim that all

---

[13] One can easily verify by induction on n that $\sum_{j=0}^{n} j^2 = \frac{1}{6}\left(2n^3 + 3n^2 + n\right)$.

mechanical processes can be simulated by (Turing) machines. We are coming back to the starting-point of our considerations through three remarks.

First, Turing analyzed *mechanical* processes of a human computor. The reduction of string machines or of K-graph machines to letter machines over a two-element alphabet does not show that *mental* processes cannot go beyond mechanical ones; it only shows that Turing machines can serve as a "normal form" for machines, because of the simplicity of their description.[14] The question, whether (different kinds of) machines are adequate mathematical models for mental processes, is left completely open. That is an empirical issue!

Second, the formulation of the boundedness and locality conditions for mechanical processes and the design of general machine models allow us to give uniform reductions. A natural generalization of K-graph machines, not giving up the above broad conditions, would allow the description of parallel computations -- carried out by "discrete deterministic mechanical devices", not computors; cf. Remark 2 in section 4. Gandy developed in his (1980) a most important way of formulating such a generalization.

Third, support for Turing's thesis is best given in two distinct steps: (i) mechanical processes satisfying boundedness and locality conditions can be recognized -- without coding or other effective transformations -- as computations of a general model; (ii) computations of the general model can be simulated by Turing machines. The plausibility of Turing's thesis rests exclusively on the plausibility of the *modified central thesis* (i); after all, (ii) is a mathematical fact. Our modification of Turing's *central thesis* states that mechanical processes are (easily seen to be) computations of K-graph machines; in our view, this is a most plausible claim.

---

[14] For this reason Turing machines are most suited for theoretical investigations. This state of affairs is analogous to that involving logical calculi: natural deducton calculi reflect quite directly the structure of ordinary arguments, but have a somewhat involved metamathematical description; in contrast, axiomatic logical systems are not suited as frameworks for direct formalizations, but -- due to their simple description - are most suitable for metamathematical investigations.

REFERENCES.

A. Church, Review of (Turing 1936); J. Symbolic Logic 1(1), 1937, 42-3.

S.A. Cook and S.O. Aanderaa, On the minimum computation time of functions; Trans. Amer. Math. Soc. 142, 1969, 291-314.

B. Courcelle, Graph Rewriting: An Algebraic and Logical Approach; in: J. van Leeuwen (ed.), Handbook of Theoretical Computer Science, 1990, 195-242.

M. Davis, *Computability and Undecidability*, McGraw-Hill, 1958.

R. Gandy, Church's thesis and principles for mechanisms; in: Barwise, Keisler, and Kunen (eds.), *The Kleene Symposium*, Amsterdam, 1980, 123-48.

R. Gandy, The confluence of ideas in 1936; in: R. Herken (ed.), *The Universal Turing Machine - A Half Century Review*, 1988, 55-111.

K. Gödel, On undecidable propositions; Lecture Notes, Princeton, 1934 - with a Postscriptum from 1964, reprinted in: Gödel's *Collected Works I*, Oxford, 1986, 346-71.

K. Gödel, Über die Länge von Beweisen; Ergebnisse eines math. Kolloquiums 77, 1936, 23-4, reprinted in: Gödel's *Collected Works I*, Oxford, 1986, 394-99.

K. Gödel, Some remarks on the undecidability results; written in 1972, reprinted in: Gödel's *Collected Works II*, Oxford, 1990, 305-6.

Y. Gurevich, Evolving algebras. A tutorial introduction; Bulletin of the European Association for Theoretical Computer Science, 43, February 1991.

T. Herron, An Alternative Definition of Pushout Diagrams and Their Use in Characterizing K-Graph Machines; Carnegie Mellon University, May 1995.

S.C. Kleene, *Introduction to Metamathematics*, Groningen, 1954.

A.N. Kolmogorov and V.A. Uspensky, On the definition of an algorithm; Uspekhi Mat. Nauk 13 (Russian), 1958; English translation in: *AMS Translations*, 2, 21 (1963), 217-245.

E. Mendelson, Second thoughts about Church's Thesis and mathematical proofs; The Journal of Philosophy, 87(5), 1990, 225-33.

D. Mundici and W. Sieg, Paper machines; Philosophia Mathematica 3, 1995, 5-30.

M.S. Paterson, M.J. Fischer, and A.R. Meyer, An improved overlap argument for on-line multiplication; SIAM-AMS Proceedings, 7, 1974, 97-111.

E. Post, Finite combinatory processes. Formulation I; J. Symbolic Logic 1, 1936, 103-5.

E. Post, Recursive unsolvability of a problem of Thue; J. Symbolic Logic 12, 1947, 1-11.

A. Schönhage, Storage modification machines; SIAM Journal on Computing 9, 1980, 490-508.

J.C. Shepherdson and H.E. Sturgis, Computability of recursive functions; J. Assoc. Computing Machinery 10, 1963, 217-55.

J.C. Shepherdson, Mechanisms for computing over arbitrary structures; in: R. Herken (ed.), *The Universal Turing Machine - A Half Century Review*, 1988, 581-601.

W. Sieg, Mechanical procedures and mathematical experience; in: *Mathematics & Mind*, A. George (ed.), Oxford University Press, 1994, 71-117.

A. Turing, On computable numbers, with an application to the Entscheidungsproblem; *Proc. London Mathematical Society*, ser. 2, vol. 42 (1936-7), 230-265.

A. Turing, The word problem in semi-groups with cancellation; Ann. of Math., 52, 1950, 491-505.

A. Turing, Solvable and unsolvable problems; Science News 31, 1953, 7-23.

V.A. Uspensky, Kolmogorov and mathematical logic; J. Symbolic Logic 57, 1992, 385-412.

V.A. Uspensky and A.L. Semenov, What are the gains of the theory of algorithms: Basic developments connected with the concept of algorithm and with its application in mathematics; in: *Algorithms in Modern Mathematics and Computer Science*, A.P. Ershov and D.E. Knuth (eds.), Lecture Notes in Computer Science 122, 1981, 100-235.