# Dijkstra Graphs

Lucila M. S. Bento[a], Davidson R. Boccardo[b], Raphael C. S. Machado[c], Flavio K. Miyazawa[e], Vinícius G. Pereira de Sá[a], Jayme L. Szwarcfiter[a,d,*]

[a] *Federal University of Rio de Janeiro, Rio de Janeiro, Brazil*
[b] *Clavis Information Security, Rio de Janeiro, Brazil*
[c] *Nacional Institute of Metrology, Quality and Technology, Duque de Caxias, Brazil*
[d] *State University of Rio de Janeiro, Rio de Janeiro, Brazil*
[e] *State University of Campinas, Campinas, Brazil*

**Abstract**

We revisit a concept that has been central in some early stages of computer science, that of *structured programming*: a set of rules that an algorithm must follow in order to acquire a certain desirable structure. While much has been written about structured programming, an important issue has been left unanswered: given an arbitrary program, describe an algorithm to decide whether or not it is *structured*, that is, whether it conforms to the stated principles of structured programming. We refer to a classical concept of structured programming, as described by Dijkstra. By employing graph theoretic techniques, we formulate an efficient algorithm for answering this question. First, we introduce the class of graphs which correspond to structured programs, which we call *Dijkstra Graphs*. Then we present a greedy $O(n)$-time algorithm for recognizing such graphs. Furthermore, we describe an isomorphism algorithm for Dijkstra graphs, whose complexity is also linear in the number of vertices of the graph.

*Keywords:* graph algorithms, graph isomorphism, reducibility, structured programming

[☆] Author 3 has been supported by CNPq and FAPERJ, Brazil. Authors 4 and 6 have been supported by CNPq, Brazil.

[*] Corresponding author
*Email address:* jayme@nce.ufrj.br (Jayme L. Szwarcfiter)

## 1. Introduction

Structured programming was one of the main topics in computer science in the years around 1970. It can be viewed as a method for the development and description of algorithms and programs. Basically, it consists of a top-down formulation of the algorithm, breaking it into blocks or modules. The blocks are stepwise refined, possibly generating new, smaller blocks, until refinements no longer exist. The technique constrains the description of the modules to contain only three basic control structures: *sequence*, *selection* and *iteration*. The first of them corresponds to sequential statements of the algorithm; the second refers to comparisons leading to different outcomes; the last one corresponds to sets of actions performed repeatedly in the algorithm.

One of the early papers about structured programming was the article by Dijkstra "Go-to statement considered harmful" [5], which brought the idea that the unrestricted use of go-to statements is incompatible with well structured algorithms. That paper was soon followed by a discussion in the literature about go-to's, as in the papers by Knuth [13], Knuth and Floyd [14] and Wulf [27]. Other classical papers are those by Dahl and Hoare [4] and Hoare [12], among others. The basic ideas of structured programming appear in detail in an article by Dijkstra [6]. The concept has been also handled by Wirth [26], among others. Kosaroju [16] describes the idea of reducibility among flowcharts. Moreover, [16] has introduced and characterized the class of *D-charts*, which in fact are graphs properly containing all those which originate from structured programming. Williams [24] also describes variations of different forms of structuredness, including those by Dijkstra, as well as D-charts. The different forms of unstructuredness were described in papers by Williams [23] and McCabe [18]. The conversion of a unstructured flow diagram into a structured one has been considered by Williams and Ossher [25], and Oulsnam [19]. Formal aspects of structured programming include the papers by Böhm and Jacopini [2], Harel [8], and Kozen and Tseng [17]. A mathematical theory for modeling structuredness, designed for flow graphs, in general, has been described by Fenton, Whitty and

Kaposi [7]. The actual influence of the concept of structured programming in the development of algorithms for solving various problems in different areas occurred right from the start, either explicitly, as in the papers by Henderson and Snow [11], and Knuth and Szwarcfiter [15], or implicitly as in the various graph algorithms by Tarjan, e.g. [20, 22].

A natural question regarding structured programming is to recognize whether a given program is structured. To our knowledge, such a question has not been solved neither in the early stages of structured programming, nor later. This is the main purpose of the present paper. We formulate an algorithm for recognizing whether a given program is structured, according to Dijkstra's model [6]. Note that the input comprises the binary code, not the source code. A well-known representation that comes in handy is that of the *(control) flow graph* of a program. A maximal straight line in the program's instructions corresponds to a basic block, and is represented by a vertex in that graph. A directed edge $AB$ (from the exit of block $A$ to the start of block $B$) represents the program flowing from $A$ to $B$ at runtime. Considering as input the control flow graph of the program, the problem becomes graph-theoretic: given a flow graph, decide whether it has been produced by a structured program. We employ a reducibility method, whose reduction operations iteratively obtain smaller graphs. Reducibility methods of this kind have been applied in papers, as [16, 21].

In this paper, we first define the class of graphs which correspond to structured programs. Such class has then been named as *Dijkstra graphs*. We describe a characterization that leads to a greedy $O(n)$ time recognition algorithm for a Dijkstra graph with $n$ vertices. Among the potential applications of the proposed algorithm, we can mention software watermarking through graphs [1, 3]. sec:isomorphism Additionally, we formulate an isomorphism algorithm for the class of Dijkstra graphs. The method consists of defining a convenient code for a graph, which consists of a string of integers. Such a code uniquely identifies the graph, and it is shown that two Dijkstra graphs are isomorphic if and only if their codes coincide. The code itself has size $O(n)$ and the time complexity of the isomorphism algorithm is also $O(n)$.

Some basic definitions and terminology are given in the next section. Section 3 defines the class of Dijkstra graphs, whose recognition is described in the Section 4. A method for verifying isomorphism of Dijkstra graphs is given in Section 5. Some additional remarks, as well as a generalization of the class, form the last section. Some of the proofs that have been omitted through the text appear in an appendix.

## 2. Preliminaries

In this paper, all graphs are finite and directed. For a graph $G$, we denote its vertex and edge sets by $V(G)$ and $E(G)$, respectively, with $|V(G)| = n$, $|E(G)| = m$. For $v, w \in V(G)$, an edge from $v$ to $w$ is written as $vw$. We say $vw$ is an *out-edge* of $v$ and an *in-edge* of $w$, with $w$ an *out-neighbor* of $v$, and $v$ an *in-neighbor* of $w$. We denote by $N^+_G(v)$ and $N^-_G(v)$ the sets of out-neighbors and in-neighbors of $v$, respectively. We may drop the subscript when the graph is clear from the context. For $S \subseteq V$, define $N^+(S) = \cup_{v \in S} N^+(v)$. Also, we write $N^{2+}(v)$ meaning $N^+(N^+(v))$. For $v, w \in V(G)$, $v$ *reaches* $w$ when there is a path in $G$ from $v$ to $w$. A *source* of $G$ is a vertex that reaches all other vertices in $G$, while a *sink* is one which reaches no vertex, except itself. Denote by $s(G)$ and $t(G)$, respectively, a source and a sink of $G$. A *(control) flow* graph $G$ is one which contains a distinguished source $s(G)$. A *source-sink* graph contains both a distinguished source $s(G)$ and a distinguished sink $t(G)$. A *trivial* graph contains a single vertex.

A graph with no directed cycles is called *acyclic*. In an acyclic graph if there is a path from vertex $v$ to vertex $w$, then $v$ is an ancestor of $w$, and the latter a descendant of $v$. Let $G$ be a flow graph with source $s(G)$, and $C$ a cycle of $G$. The cycle $C$ is called *single-entry* if it contains a vertex $v \in C$ that separates $s(G)$ from the vertices of $C \setminus \{v\}$. A flow graph in which each of its cycles is single-entry is called *reducible*. Reducible graphs were characterized by Hecht and Ullman [9, 10], while efficient recognition has been described by Tarjan [21].

In a *depth-first search (DFS)* of a directed graph, in each step a vertex is

inserted in a stack, or removed from it. Every vertex is inserted and removed from the stack exactly once. An edge $vw \in E(G)$, such that $v$ is inserted in the stack after $w$, and before the removal of $w$, is called a *cycle edge*. Let $E_C$ be the set of cycle edges of a graph, relative to some DFS. Clearly, $G - E_C$ is acyclic. The following characterization is relevant for our purposes.

**Theorem 2.1.** *[10, 21] A flow graph $G$ is reducible if and only if, for any depth-first search of $G$ starting at $s(G)$, the set of cycle edges is invariant.*

In a flow graph $G$, we may write DFS of $G$, as to mean a DFS of $G$ staring at $s(G)$. In addition, if $G$ is reducible, we may use the terms *ancestor* or descendant of $G$, as to mean *ancestor* or descendant of $G - E_C$. A *topological sort* of a graph $G$ is a sequence $v_1, \ldots, v_n$ of its vertices, such that $v_i v_j \in E(G)$ implies $i < j$. Finally, write $G_1 \cong G_2$, to denote that graphs $G_1, G_2$ are *isomorphic*.

## 3. The Graphs of Structured Programming

In this section, we describe the graphs of structured programming. First, we introduce a family of graphs, following Dijkstra's description [6].

A *statement graph* is defined as being one of the following: (a) *trivial* graph; (b) *sequence* graph; (c) *if* graph; (d) *if-then-else* graph; (e) *p-case* graph, $p \geq 3$; (f) *while* graph; (g) *repeat* graph.

For our purposes, it is convenient to assign labels to the vertices of statement graphs as follows. Each vertex is either an *expansible vertex*, labeled $X$, or a *regular vertex*, labeled $R$. See Figures 1 and 2, where the statement graphs are depicted with the corresponding vertex labels. All statement graphs are source-sink. If there is more than one candidate to be the source of a statement graph, we choose always the one which appears as the topmost vertex in the corresponding figure. Then vertex $v$ denotes the source of the graph in each case of Figures 1 and 2.

Let $G$ be an unlabeled reducible graph, and $H$ a subgraph of $G$, having source $s(H)$ and sink $t(H)$. We say $H$ is *closed* when

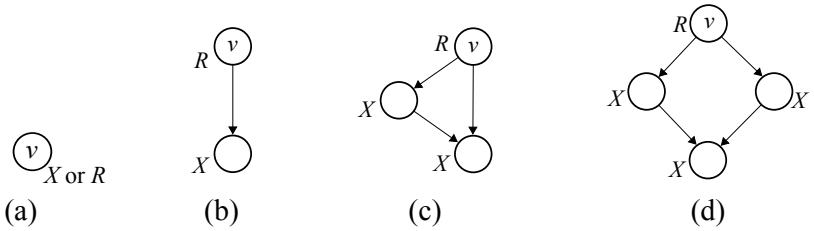


Figure 1: Statement graphs (a)-(d)

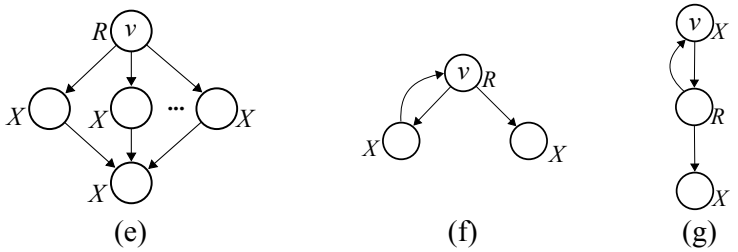


Figure 2: Statement graphs (e)-(g)

- $v \in V(H) \setminus s(H) \Rightarrow N^-(v) \subseteq V(H)$;
- $v \in V(H) \setminus t(H) \Rightarrow N^+(v) \subseteq V(H)$; and
- $vs(H)$ is a cycle edge $\Rightarrow v \in N^+(s(H))$.

Note that the while and repeat graphs, respectively, (f) and (g) of Figure 2, are isomorphic when considered in an isolated framework. However, this is not so in the context of flow reducible graphs, as observed in the lemma below.

**Lemma 3.1.** *Let $G$ be a flow reducible graph, containing both a while graph $A$ and a repeat graph $B$, as induced subgraphs. Then $A$ and $B$ are distinguishable subgraphs, even when there are no labels.*

*Proof.* The cycle edge, which is an invariant for flow reducible graphs, can distinguish between the while and repeat graph. In the while graph the source has an out-edge to the sink, while this is not so in the repeat graph. □

Let $H$ be an induced subgraph of $G$. Say $H$ is *prime* when it is closed and isomorphic to some non-trivial statement graph.

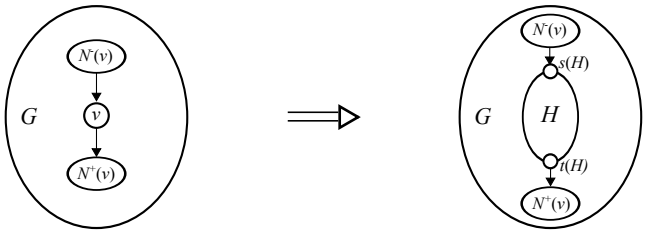


Figure 3: Expansion operation

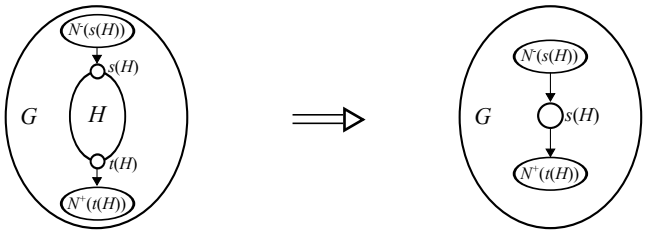


Figure 4: Contraction operation

Next, let $G, H$ be two graphs, $V(G) \cap V(H) = \emptyset$, $H$ source-sink, $v \in V(G)$. The *expansion* of $v$ into a source-sink graph $H$ (Figure 3) consists of replacing $v$ by $H$, in $G$, such that $N_G^-(s(H)) := N_G^-(v)$, $N_G^+(t(H)) := N_G^+(v)$, and preserving the remaining adjacencies.

Similarly, in the *contraction* of a source-sink graph $H$ into a single vertex (Figure 4), we identify (coalesce) the vertices of $H$ into the source $s(H)$ of $H$, and remove all possible parallel edges and loops.

A *Dijkstra graph (DG)* is one whose vertices are labeled $X$ or $R$, recursively defined as:

1. A trivial graph is a DG.

2. Any graph obtained from a DG by expanding some $X$-vertex into a statement graph is also a DG (See Figure 5).

The above definition leads directly to a method for constructing Dijkstra graphs, as follows. Find a sequence of graphs $G_0, \ldots, G_k$, such that $G_0$ is the trivial graph, and $G_i$ is obtained from $G_{i-1}$, $i \geq 1$, by expanding some X-vertex

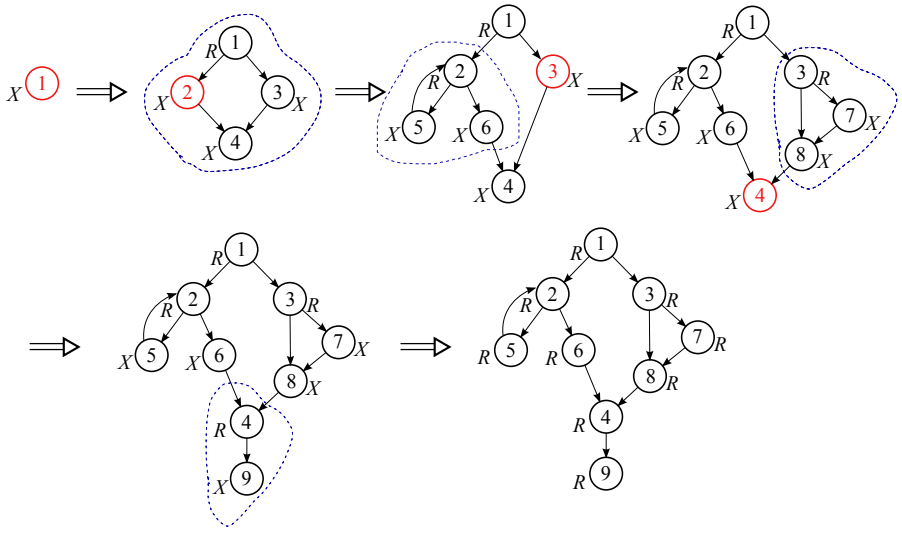

Figure 5: Obtaining a Dijkstra graph via vertex expansions

$v$ of it into a statement graph $H$.

It is relevant to note that the labels are used merely for constructing the graphs. For the actual recognition process, there are no labels. We are interested in the problem of deciding whether a given *unlabeled* flow graph is actually a Dijkstra graph.

## 4. Recognition of Dijkstra Graphs

By hypothesis, we are given an arbitrary unlabeled flow graph $G$, and the aim is to decide whether or not $G$ is a DG.

### *4.1. Basic Lemmas*

We describe some lemmas which are implicitly employed in the recognition process.

**Lemma 4.1.** *If $G$ is a Dijkstra graph, then*

*(i) $G$ contains some prime subgraph;*

*(ii) $G$ is a source-sink graph; and*

*(iii) $G$ is reducible.*

*Proof.* By definition, there is a sequence of graphs $G_0, \ldots, G_k$, where $G_0$ is trivial, $G_k = G$ and $G_i$ is obtained from $G_{i-1}$ by expanding some $X$-vertex $v_{i-1} \in V(G_{i-1})$ into a statement graph $H_i \subseteq G_i$. Then no vertex $v_i \in V(H_i)$, except $s(H_i)$ has in-neighbors outside $H_i$, and also no vertex $v_i \in V(H_i)$, except $t(H_i)$, has out-neighbors outside $H_i$. Furthermore, if $H_i$ contains any cycle then $H_i$ is necessarily a while graph or a repeat graph. The latter implies that such a cycle is $s(H)v$, where $v \in N^+(s(H))$. Therefore $H_i$ is prime in $G_i$ meaning that $(i)$ holds. To show $(ii)$ and $(iii)$, first observe that any statement graph is single-source and reducible. Next, apply induction. For $G_0$, there is nothing to prove. Assume it holds for $G_i$, $i > 1$. Let $v_{i-1} \in V(G_{i-1})$ be the vertex that expanded into the subgraph $H_i \subseteq G_i$. Then the external neighborhoods of $H_i$ coincide with the neighborhoods of $v_{i-1}$, respectively. Consequently, $G_i$ is single-source. Now, let $C_i$ be any cycle of $G_i$, if existing. If $C_i \cap H_i = \emptyset$ then $C_i$ is single-entry, since $G_{i-1}$ is reducible. Otherwise, if $C_i \subset V(H_i)$ the same is valid, since any statement graph is reducible. Finally, if $C_i \not\subset V(H_i)$, then $v_{i-1}$ is contained in a single-entry cycle $C_{i-1}$ of $G_{i-1}$. Then $C_i$ has been formed by $C_{i-1}$, replacing $v_{i-1}$ by a path contained in $H_i$. Since $C_{i-1}$ is single-entry, it follows that $C_i$ must be so. □

Denote by $\mathcal{H}(G)$ the set of non-trivial prime graphs of $G$. Let $H, H' \in \mathcal{H}(G)$. Call $H, H'$ *independent* when

- $V(H) \cap V(H') = \emptyset$, or
- $V(H) \cap V(H') = \{v\}$, where $v = s(H) = t(H')$ or $v = t(H) = s(H')$.

**Lemma 4.2.** (Prime Independency)*: If $H, H' \in \mathcal{H}(G)$ then $H, H'$ are independent.*

*Proof.* If $V(H) \cap V(H') = \emptyset$ the lemma holds. Otherwise, let $v \in V(H) \cap V(H')$. The alternatives $v = s(H_1) = s(H_2)$, $v = t(H_1) = t(H_2)$, $v \neq s(H_1), t(H_1)$ or $v \neq s(H_2), t(H_2)$ do not occur because they imply $H_1$ or $H_2$ not to be

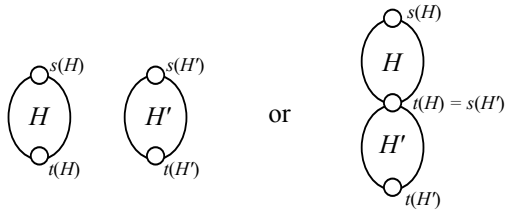


Figure 6: Independent primes

closed. Next, let $v_1, v_2 \in V(H_1) \cap V(H_2)$, $v_1 \neq v_2$. In this situation, examine the alternative where $v_1 = s(H_1) = t(H_2)$ and $v_2 = s(H_2) = t(H_1)$. The latter implies that exactly one of $H_1$ or $H_2$, say $H_2$, is a while graph or a repeat graph. Then there is a cycle edge $ws(H_1)$, satisfying $w \in N^-(s(H_1))$ and $w \in V(H_2) \setminus \{t(H_2)\}$. Consequently, $w \notin N^+(s(H_1))$, contradicting $H_1$ to be closed. The only remaining alternative is $V(H_1) \cap V(H_2) = \{v\}$, with $v = s(H_1) = t(H_2)$ or $v = s(H_2) = t(H_1)$. Then $H_1, H_2$ are indeed independent (see Figure 6). □

For a graph $G$, denote by $G \downarrow H$ the graph obtained from $G$ by contracting $H$. For $v \in V(G)$, the *image* of $v$ in $G \downarrow H$, denoted $I_{G\downarrow H}(v)$, is

$$I_{G\downarrow H}(v) = \begin{cases} v, & \text{if } v \notin V(H) \\ s(H), & \text{otherwise.} \end{cases}$$

For $V' \subseteq V(G)$, define the (*subset*) *image* of $V'$ in $G \downarrow H$, as $I_{G\downarrow H}(V') = \cup_{v \in V'} I_{G\downarrow H}(v)$. Similarly, for $H' \subseteq G$, the (*subgraph*) *image* of $H'$ in $G \downarrow H$, denoted by $I_{G\downarrow H}(H')$, is the subgraph induced in $G \downarrow H$ by the subset of vertices $I_{G\downarrow H}(V(H'))$.

Let $G$ be an arbitrary flow graph, $H, H' \in \mathcal{H}(G)$, $H \neq H'$.

**Lemma 4.3.** (Prime preservation)*: If $H, H' \in \mathcal{H}(G)$, $H \neq H'$, then $I_{G\downarrow H}(H') \in \mathcal{H}(G \downarrow H)$.*

*Proof.* By Lemma 3, $H, H' \in \mathcal{H}(G)$, $H \neq H'$. By Lemma 4.2, $H, H'$ are independent. If $H, H'$ are disjoint the contraction of $H$ does not affect $H'$, and

the lemma holds. Otherwise, by the independence condition, it follows that $V(H) \cap V(H') = \{v\}$, where $v = s(H) = t(H')$ or $v = s(H') = t(H)$. Examine the first of these alternatives. By contracting $H$, all neighborhoods of the vertices of $I_{G\downarrow H}(H')$ remain unchanged, except that of $I_{G\downarrow H}(s(H'))$, since its in-neighborhood becomes equal to $N_G^-(s(H))$. On the other hand, the contraction of $H$ into $v$ cannot introduce new cycles in $H'$. Consequently, $H'$ preserves in $G \downarrow H$ its property of being a non-trivial and closed statement graph, moreover, prime. Finally, suppose $v = s(H) = t(H')$. Again, the neighborhoods of the vertices of $I_{G\downarrow H}(H')$ are preserved, except possibly the out-neighborhoods of the vertices of $I_{G\downarrow H}(t(H'))$, which become $N_G^+(t(H))$, after possibly removing self-loops. Consequently, $I_{G\downarrow H}(H') \in \mathcal{H}(G \downarrow H)$. □

**Lemma 4.4.** (Commutative law)*: If subgraphs $H, H' \in \mathcal{H}(G)$, it follows that* $(G \downarrow H) \downarrow (I_{G\downarrow H}(H')) \cong (G \downarrow H') \downarrow (I_{G\downarrow H'}(H))$.

*Proof.* Let $A \cong (G \downarrow H) \downarrow (I_{G\downarrow H}(H'))$ and $B \cong (G \downarrow H') \downarrow (I_{G\downarrow H'}(H))$. By Lemma 4.2, $H, H'$ are independent. First, suppose $H, H'$ are disjoint. Then $I_{G\downarrow H}(H') = H'$ and $I_{G\downarrow H'}(H) = H$. It follows that, in both graphs $A$ and $B$, the subgraphs $H$ and $H'$ are respectively replaced by a pair of non-adjacent vertices, whose in-neighborhoods are $N_G^-(s(H))$ and $N_G^-(s(H'))$, and out-neighborhoods $N_G^+(t(H))$ and $N_G^+(t(H'))$, respectively. Then $A = B$. In the second alternatives, suppose $H, H'$ are not disjoint. Then $V(H) \cap V(H') = \{v\}$, where $v = s(H) = t(H')$, or $v = t(H) = s(H')$. In both cases, and in both graphs $A$ and $B$, the subgraphs $H$ and $H'$ are contracted into a common vertex $w$. When $v = s(H) = t(H')$, it follows $N_A^-(w) = N_G^-(s(H')) = N_B^-(w)$ and $N_A^+(w) = N_G^+(t(H)) = N_B^+(w)$. Finally, when $v = t(H) = s(H')$, we obtain a similar result. Consequently, $A = B$ in any situation. □

### *4.2. Contractile Sequences*

A sequence of graphs $G_0, \ldots, G_k$ is a *contractile sequence* for a graph $G$, when

- $G \cong G_0$, and
- $G_{i+1} \cong (G_i \downarrow H_i)$, for some $H_i \in \mathcal{H}(G_i)$, $i < k$. Call $H_i$ the *contracting prime* of $G_i$.

We say $G_0, \ldots, G_k$ is *maximal* when $\mathcal{H}(G_k) = \emptyset$. In particular, if $G_k$ is the trivial graph then $G_0, \ldots, G_k$ is maximal.

Let $G_0, \ldots, G_k$, be a contractile sequence of $G$, and $H_j$ the contracting prime of $G_j$. That is, $G_{j+1} \cong (G_j \downarrow H_j)$, $0 \leq j < k$. For $H'_j \subseteq G_j$ and $q \geq j$, the *iterated image* of $H'_j$ in $G_q$ is the subgraph $I_{G_q}(H'_j)$ of $G_q$, obtained by iteratively finding the image $I_{G_{j+1}}(H'_j)$ of $H'_j$ in $G_{j+1} = G_j \downarrow H_j$, and then the image $I_{G_{j+2}}(H'_j)$ of $I_{G_{j+1}}(H'_j)$ in $G_{j+2} = G_{j+1} \downarrow H_{j+1}$, and so on until reaching the image $I_{G_q}(H'_j)$ of $I_{G_{q-1}}(H'_j)$ in $G_q = G_{q-1} \downarrow H_q$. That is $I_{G_q}(H'_j)$ can be defined recursively as

$$I_{G_q}(H'_j) = \begin{cases} H'_j, & \text{if } q = j \\ I_{G_{q-1} \downarrow H_{q-1}}(I_{G_{q-1}}(H'_j)), & \text{otherwise.} \end{cases}$$

Finally, we describe the required characterization.

**Theorem 4.5.** *Let $G$ be an arbitrary flow graph, with $G_0, \ldots, G_k$ and $G'_0, \ldots, G'_{k'}$ two contractile sequences of $G$. Then $G_k \cong G'_{k'}$. Furthermore, $k = k'$.*

*Proof.* Let $G_0, \ldots, G_k$ and $G'_0, \ldots, G'_{k'}$ be two maximal contractile sequences, denoted respectively by $S$ and $S'$ of a graph $G$. Let $H_j$ and $H'_j$ be the contracting primes of $G_j$ and $G'_j$, respectively. That is, $G_{j+1} \cong (G_j \downarrow H_j)$ and $G'_{j+1} \cong (G'_j \downarrow H'_j)$, $j < k$ and $j < k'$. Without loss of generality, assume $k \leq k'$. Let $i$ be the least index, such that $G_j \cong G'_j$, $j \leq i$. Such an index exists since $G \cong G_0 \cong G'_0$. If $i = k$ then $G_k \cong G'_{k'}$, implying $k = k'$ and the theorem holds. Otherwise, $i < k$, $G_i \cong G'_i$ and $G_{i+1} \ncong G'_{i+1}$. Since $G_i \cong G'_i$, it follows $H_i \in \mathcal{H}(G'_i)$. By Lemma 4.3, the iterated image $H_{i_q}$, of $H_i$ in $G'_q$ is preserved as a prime subgraph for all $G'_q$, as long as it does not become the contracting prime of $G'_{q-1}$. Since $G'_{k'}$ has no prime subgraph, it follows there exists some index $p$, $i < p < k'$, such that $G'_{p+1} \cong (G_p \downarrow H_{i_p})$, where $H_{i_p}$ represents the iterated

image of $H_i$ in $G'_p$. Let $H_{i_{p-1}}$ be the iterated image of $H_i$ in $G'_{p-1}$. Clearly, $H'_{p-1}, H_{i_{p-1}} \in \mathcal{H}(G'_{p-1})$, and by Lemma 4.2, $H'_{p-1}$ and $H_{i_{p-1}}$ are independent in $G'_{p-1}$. Since $((G'_{p-1} \downarrow H'_{p-1}) \downarrow H_{i_p}) \cong G'_{p+1}$, by Lemma 4.3, it follows that $((G'_{p-1} \downarrow H_{i_{p-1}}) \downarrow H''_{p-1}) \cong G'_{p+1}$, where $H''_{p-1}$ represents the image of $H'_{p-1}$ in $G'_{p-1} \downarrow H_{i_{p-1}}$. Consequently, we have exchanged the positions in $S'$ of two contracting primes, respectively at indices $p-1$ and $p$, while preserving graphs $G'_q$, for $q < p-1$ and $q > p$. In particular, also preserving $G'_{p+1}$ and the graphs lying after $G'_{p+1}$ in $S'$, together with their contracting primes.

Finally, apply the above operation iteratively, until eventually the iterated image of $H_i$ becomes the contracting prime of $G'_i$. In the latter situation, the two sequences coincide up to index $i+1$, while preserving the original graphs $G_k$ and $G'_{k'}$. Again, applying iteratively such an argument, we eventually obtain that the two sequences turned coincident, preserving the original graphs $G_k$ and $G'_{k'}$. Consequently, $G_k \cong G'_{k'}$ and $k = k'$. □

*4.3. The Recognition Algorithm*

We start with a bound for the number $m$ of edges of Dijkstra graphs.

**Lemma 4.6.** *Let $G$ be a DG graph. Then $m \leq 2n - 2$.*

*Proof.* If $G$ is a DG graph there is a sequence of graphs $G_0, \ldots G_k$, where $G_0$ is the trivial graph, $G_k \cong G$ and $G_i$ is obtained from $G_{i-1}$ by expanding an $X$-vertex of $G_{i-1}$ into a statement graph. Apply induction on the number of expansions employed in the construction of $G$. If $k = 0$ then $G$ is a trivial graph, which satisfies the lemma. For $k \geq 0$, Suppose the lemma true for any graph $G' \cong G_i$, $i < k$. In particular, let $G_i \cong G_{k-1}$. Let $n'$ and $m'$ be the number of vertices and edges of $G'$, respectively. Then $m' \leq 2n' - 2$. We know that $G_k$ has been obtained by expanding a vertex of $G_{k-1}$ into a statement graph $H$. Consider the alternatives for $H$. If $H$ is the trivial graph then $n = n'$ and $m = m'$. If $H$ is a sequence graph then $n = n' + 1$ and $m = m' + 1$. If $H$ is an if graph, a while graph or repeat graph then $n = n' + 2$ and $m = m' + 3$. If $H$ is an if then else graph or a $p$-case graph then $n = n' + p + 1$ and $m = m' + 2p$,

where $p$ is the outdegree of the source of $H$. In any of these alternatives, a simple calculation implies $m \leq 2n - 2$. □

We describe an algorithm for recognizing Dijkstra graphs based on Theorem 4.5. Let $G$ be a flow reducible graph. Construct a contractile sequence $G_0, \ldots, G_k$ of $G$. That is, find iteratively a non-trivial prime subgraph $H_i$ of the $G_i$ and contract it, until either the graph becomes trivial or otherwise no such subgraph exists. In the first case the graph is a DG, while in the second it is not. Recall from Lemma 4.3 that whenever $G_i$ contains another prime $H_j \neq H_i$ then the iterated image of $H_j$ is preserved, as long as it does not become the contracting prime. On the other hand, the contraction $G_i \downarrow H_i$ may generate a new prime $H_i'$, as shown in Figure 7. However, the generation of new primes obeys a rule, described by the lemma below.

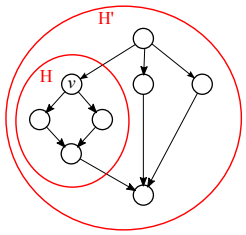


Figure 7: Contracting H generates prime H'

**Lemma 4.7.** *Let $G$ be a reducible graph, $H \in \mathcal{H}(G)$, $H' \in \mathcal{H}(G \downarrow H) \setminus \mathcal{H}(G)$. Then $s(H)$ is a proper descendant of $s(H')$ in $G \downarrow H$.*

Let $G$ be a reducible graph, $G_0, \ldots, G_k$ a contractile sequence $\mathcal{C}$ of $G$, and $H_i$ the contracting prime of $G_i$, $0 \leq i < k$. Say that $\mathcal{C}$ is a *bottom-up sequence* of $G$ when $s(H_i)$ is not a descendant of $s(H)$, for any prime $H \neq H_i$ of $G_i$.

The recognition algorithm then becomes as follows. Let $G$ be a reducible graph. Traverse $G$ in a bottom-up order. Iteratively, find a lowest vertex $v$ of $G$, s.t. $v$ is the source of a prime subgraph $H$ of $G$. Then contract $H$. Stop when no primes exist any longer.

A complete description of the algorithm is then detailed. The algorithm answers YES or NO, according to respectively $G$ is a Dijkstra graph or not.

**Algorithm 1:** Dijkstra graphs recognition algorithm

```
1  G, arbitrary flow graph (no labels)
2  Count the number m of edges of G, stopping counting if m reached
     2n − 1.
3  If m > 2n − 2 then return NO
4  E_C, set of cycle edges of a DFS of G, starting at s(G)
5  v_1, ..., v_n, topological sorting of G − E_C
6  i := n
7  while i ≥ 1 do
8      if G is the trivial graph
9          then return YES
10     if v_i is the source of a prime subgraph H of G
11         then G := G ↓ H
12     i := i − 1
13 return NO
```

The correctness of Algorithm 1 follows basically from Theorem 4.5 and Lemma 4.7. However, the latter relies on the fact that $G$ is a reducible graph, whereas the input to Algorithm 1 is an arbitrary graph, and no explicit step for recognizing whether $G$ is a reducible graph is performed. The purpose was to avoid such a previous recognition, whose complexity is not linear. The lemma below justifies it.

**Lemma 4.8.** *Let $G$ be an arbitrary flow graph input to Algorithm 1. If $G$ is not a reducible graph then the algorithm would correctly answer NO.*

*Proof.* If $G$ is not a reducible graph let $E_C$ be the set of cycle edges, relative to some DFS starting at $s(G)$. Then $G$ contains some cycle $C$, such that $w$ does not separate $s(G)$ from $v$, where $vw \in E_C$ is the cycle edge of $C$. Without loss

of generality, consider the innermost of these cycles. The only way in which the edge $vw$, or any of its possible images, can be contracted is in the context of a while or repeat prime subgraph $H$, in which the cycle would be contracted into vertex $w$, or a possible iterated image of it. However there is no possibility for $H$ to be identified as such, because the edge entering the cycle from outside prevents the subgraph to be closed. Consequently, the algorithm necessarily would answer NO. □

As for the complexity, first observe that to decide whether the graph contains a non-trivial prime subgraph whose source is a given vertex $v \in V(G)$, we need $O|(N^+(v)|$ steps. Therefore, when considering all vertices of $G$ we require $O(m)$ time. There can be $O(n)$ prime subgraphs altogether, and each time some prime $H$ is identified, it is contracted. The size of the graph decreases by $|E(H)|$. The number of steps required to contract a prime $H$ is $O|E(H)|$. If an edge is contracted it is not considered again in the process. Hence each edge is examined at most a constant number of times during the entire process. Finding a topological sorting of a graph can be done in $O(m)$. Thus, the time complexity is $O(m)$, that is, $O(n)$, by Lemma 4.6.

## 5. Isomorphism of Dijkstra Graphs

In this section, we describe a linear time algorithm for the isomorphism of Dijkstra graphs.

Given a Dijkstra graph $G$, define a code $C(G)$ for $G$, s.t. for any two Dijkstra graphs $G_1, G_2$, $G_1 \cong G_2$ if and only if $C(G_1) = C(G_2)$.

As in the recognition algorithm, the codes are obtained by constructing a bottom-up contractile sequence of each graph. The codes consist of (linear) strings and refer explicitly to the statement graphs having source $v$ as depicted in Figures 1 and 2. The string $C(G)$ that will be coding $G$ is constructed over an alphabet whose symbols belong to the set $\{1, \ldots, \Delta^+(G)+4\}$, where $\Delta^+(G)$ is the maximum out-neighborhood size of $G$. Let, $A, B$ be a pair of strings. Denote by $A||B$ the string formed by $A$, immediately followed by $B$.

We assign an integer, named $type(H)$, for each statement graph $H$, a code $C(v)$ for each vertex $v \in V(G)$, and a code $C(H)$ for each prime subgraph $H$ of a bottom-up contractile sequence of $G$. The code $C(G)$ of graph $G$ is defined as equal to $C(s(G))$. For a subset $V' \subseteq V(G)$, the code $C(V')$ of $V'$ is the set of strings $C(V') = \{C(v_i)|v_i \in V'\}$. Write $lex(C(V')) = C(v_1)||...||C(v_r)$ whenever $V' = \{v_1, \ldots, v_r\}$ and $C(v_i)$ is lexicographically not greater than $C(v_{i+1})$.

Table 1: Statement graph types and codes $C(H)$ of prime subgraphs $H$

| statement graphs $H$ | $type(H)$ | $C(H), v = s(H)$ |
|---|---|---|
| trivial | 1 | |
| sequence | 2 | $2\|\|C(N^+(v))$ |
| if-then | 3 | $3\|\|C(N^+(v)) \setminus N^{+2}(v))\|\|C(N^{+2}(v))$ |
| while | 4 | $4\|\|C(N^+(v) \cap N^-(v))\|\|C(N^+(v) \setminus N^-(v))$ |
| repeat | 5 | $5\|\|C(N^+(v))\|\|C(N^{+2}(v) \setminus \{v\})$ |
| if-then-else | 6 | $6\|\|lex(C(N^+(v)))\|\|C(N^{+2}(v))$ |
| $p$-case | $p+4$ | $p+4\|\|lex(C(N^+(v)))\|\|C(N^{+2}(v))$ |

The types of the statement graphs are shown in the second column of Table 1. For a vertex $v \in V(G)$, the code $C(v)$ is initially set to 1. Subsequently, if $v$ becomes the source of a prime graph $H$, the string $C(v)$ is updated by assigning $C(v) := C(v)||C(H)$, where $C(H)$ is given by the third column of the table. In fact it corresponds to the expansion of $v$. A possible expansion of some vertex $w \neq v$ contained in $H$ would imply in a new update of $C(v)$, and so iteratively.

### *5.1. The Isomorphism Algorithm*

Let $G$ be a DG. Algorithm 2 constructs the code $C(G)$ for $G$.

As an example, we determine the code of the DG of Figure 8. The codes of all vertices are initially set to 1. Using a bottom-up sequence, the following vertices $v_i$ would be iteratively chosen as sources of primes, leading to codes

**Algorithm 2:** Dijkstra graphs isomorphism algorithm

$G$, DG; $E_C$, set of cycle edges of $G$

Find a topological sorting $v_1, \ldots, v_n$ of $G - E_C$

**for** $i = n, n-1, \ldots, 1$ **do**

  $C(v_i) := 1$

  **if** $v_i$ is the source of a prime subgraph $H$ **then**

$$C(v_i) := C(v_i)|| \begin{cases} 2||C(N^+(v_i)), & \text{if } H \text{ is a sequence graph;} \\ 3||C(N^+(v_i) \setminus N^{+2}(v_i))||C(N^{+2}(v_i)), & \\ & \text{if } H \text{ is an if-then graph;} \\ 4||C(N^+(v_i) \cap N^-(v_i))||C(N^+(v_i) \setminus N^-(v_i)), & \\ & \text{if } H \text{ is a while graph,} \\ 5||C(N^+(v_i))||C(N^{+2}(v_i) \setminus \{v_i\}), & \\ & \text{if } H \text{ is a repeat graph;} \\ 6||lex(C(N^+(v_i)))||C(N^{+2}(v_i)), & \\ & \text{if } H \text{ is an if-then-else graph.} \\ p+4||lex(C(N^+(v_i)))||C(N^{+2}(v_i)), & \\ & \text{if } H \text{ is a p-case graph.} \end{cases}$$

$C(G) := C(v_1)$

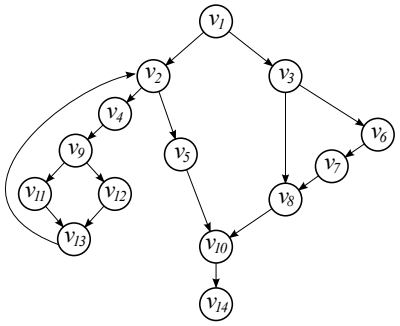


Figure 8: Example for isomorphism algorithm

$C(v_i)$:

source: $v_{10} \Rightarrow C(v_{10}) := 12||C(v_{14}) = 121$

source: $v_9 \Rightarrow C(v_9) := 16||lex(C(v_{11}), C(v_{12}))||C(v_{13}) = 16111$

source: $v_4 \Rightarrow C(v_4) := 12||C(v_9) = 1216111$

source: $v_6 \Rightarrow C(v_6) := 12||C(v_7) = 121$

source: $v_3 \Rightarrow C(v_3) := 13||C(v_6)||C(v_8) = 131211$

source: $v_2 \Rightarrow C(v_2) := 14||C(v_4)||C(v_5) = 1412161111$

source: $v_1 \Rightarrow C(v_1) := 16||lex(C(v_2), C(v_3))||C(v_{10}) = 161312111412161111121$

Finally, $C(G) := C(v_1) = 161312111412161111121$

*5.2. Correctness and Complexity*

**Theorem 5.1.** *Let $G, G'$ be Dijkstra graphs, and $C(G), C(G')$ their codes, respectively. Then $G, G'$ are isomorphic if and only if $C(G) = C(G')$.*

*Proof.* First, consider that $G, G'$ are isomorphic. We show that it implies $C(G) = C(G')$. Following the isomorphism algorithm, observe that the number of 1's in the strings $C(G), C(G')$ represents the number of vertices of $G, G'$, respectively, whereas each integer $> 1$ in the strings, represents the contraction of a prime subgraph. Furthermore, each prime subgraph $H$, which is initially contained in the input graph $G$, corresponds in $C(G)$, to a substring formed by the integer $type(H)$ followed by one 1, if $type(H) = 2$; or two 1's, if $type(H) = 3$; or three 1's, if $4 \leq type(H) \leq 6$; or $type(H)+1$ 1's, if $type(H) > 6$; respectively. Clearly, the same holds for the graph $G'$ and its code $C(G')$. The proof is by induction on the number $k$ of contractions needed to reduce both $G$ and $G'$ to a trivial vertex. By Theorem 4.5, $k$ is invariant and applies for both graphs $G$ and $G'$. If $k = 0$ then both $G$ and $G'$ are trivial graphs, and the theorem holds, since $C(G) = C(G') = 1$. When $k > 0$, assume that if $G_-$ and $G'_-$ are isomorphic DG graphs which require less than $k$ contractions for reduction then $C(G_-) = C(G'_-)$. Furthermore, assume also by the induction hypothesis, that if $v, v'$ are vertices of $G_-, G'_-$, corresponding to 1's at the same relative positions in $C(G)$ and $C(G_-)$, respectively, then $v' = f(v)$, where $f$ is the isomorphism function between $G_-$ and $G'_-$. Now, consider the graphs $G$ and $G'$. Choose a

prime subgraph $H$ of $G$, and let $v = s(H)$. Let $v' = f(v)$ be a vertex of $G'$ corresponding to $v$ by the isomorphism. Since $G \cong G'$, it follows that $v'$ is the source of a prime subgraph $H'$ of $G'$. Moreover $H \cong H'$. Consider the contractions $G \downarrow H$ and $G' \downarrow H'$, leading to graphs $G_-$ and $G'_-$, respectively. Let $C_-(G)$ and $C_-(G')$ be the strings obtained from $C(G)$ and $C(G')$, respectively by contracting the substrings corresponding to $H$ and $H'$, as above. That is, all the 1's of $C(H)$ and $C(H')$ are compressed into the positions of $v = s(H)$ and $v' = s(H')$, respectively, while the integers $type(H)$ and $type(H')$ become 1, maitaining their original positions. It follows that $C(G_-) = C_-(G)$ and $C(G'_-) = C_-(G')$. By the induction hypothesis $C(G_-) = C(G'_-)$ and the 1's corresponding to $v$ and $v'$ lie in the same relative positions in the strings. Consequently, by replacing the latter 1's for the substrings which originally represented $H$ and $H'$, we conclude that indeed $C(G) = C(G')$, and moreover the induction hypothesis is still verified. The converse is similar. □

**Corollary 5.2.** *Let $G$ be a DG. The following affirmatives hold.*

1. *There is a $1-1$ correspondence between the 1's of $C(G)$ and vertices of $G$.*

2. *The code $C(G)$ of $G$ is unique and is a representation of $G$.*

Finally, consider the complexity of the isomorphism algorithm.

**Lemma 5.3.** *Let $G$ be a Dijkstra graph, and $C(G)$ its code. Then $|C(G)| = n + k \leq 2n - 1$, where $n$ is the number of vertices of $G$ and $k$ the number of contractions needed to reduce it to a trivial vertex.*

*Proof.* The encoding $C(G)$ consists of exactly $n$ 1's, together with elements of a multiset $U \subseteq \{2, 3, \ldots, \Delta^+(G)+4\}$. We know that $C(G)$ starts and ends with an 1, and it contains no two consecutive elements of $U$. Therefore $C(G) \leq 2n - 1$. When $G$ consists of the induced path $P_n$, it follows $|C(P_n)| = 2n - 1$, attaining the bound. □

**Theorem 5.4.** *The isomorphism algorithm terminates within $O(n)$ time.*

*Proof.* Recall that $m = O(n)$, by Lemma 4.6. The construction of a bottom-up contractile sequence requires $O(n)$ steps. For each $v \in V(G)$, following the isomorphism algorithm, $C(v)$ can be constructed in time $|C(v)|$. We remark that lexicographic ordering takes linear time on the total length of the strings to be sorted. It follows that the algorithm requires no more than $O(n)$ time to construct the code $C(G)$ of $G$. □

## 6. Conclusions

The analysis of control flow graphs and different forms of structuring have been considered in various papers. To our knowledge, no full characterization and no recognition algorithm for control flow graphs of structured programs have been described before. There are some related classes for which characterizations and efficient recognition algorithms do exist, e.g. the classes of reducible graphs and D-charts. However, both contain and are much larger than Dijkstra graphs.

An important question solved in this paper is that of recognizing whether two control flow graphs (of structured programs) are syntactically equivalent, i.e., isomorphic. Such question fits in the area of *code similarity analysis*, with applications in clone detection, plagiarism and software forensics.

Since the establishment of structured programming, some new statements have been proposed to add to the original structures which forms the classical structured programming, enlarging the collection of allowed statements. Some of such relevant statements are depicted in Figure 9.

(a) *break-while*: Allows an early exit from a while statement;

(b) *continue-while*: Allows a while statement to proceed, after its original termination;

(c) *break-repeat*: Allows an early exit from a repeat statement;

(d) *continue-repeat*: Allows a repeat statement to proceed, after its original termination;

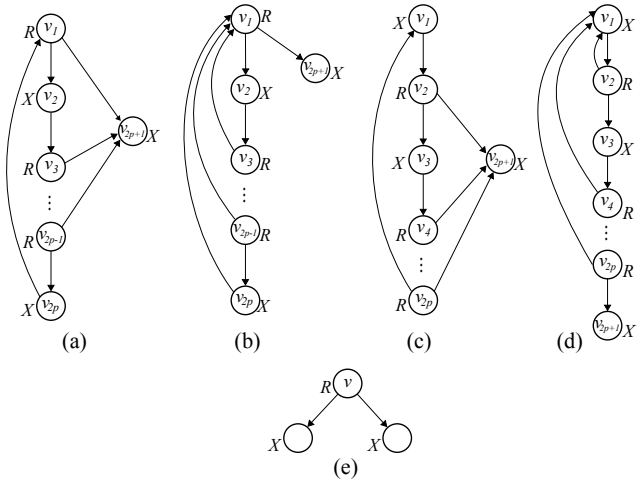


Figure 9: Generalized Dijkstra graphs

(e) *divergent-if-then-else*: A selection statement, similar to the standard *if-then-else*, except that the comparisons do not converge afterwords to a same point, but lead to disjoint structures. Note that the corresponding graph has no longer a (unique) sink.

In fact, the inclusion of some of the above additional control blocks into structured programming has been already predicted in some papers, as [13]. The basic ideas and techniques described in the present work can be generalized, so as to efficiently recognize graphs that incorporate the above statements, in addition to those of Dijkstra graphs. Similarly, for the isomorphism algorithm.

Acknowledgments

The authors are grateful to Victor Campos for the helpful discussions and comments during the French-Brazilian Workshop of Graphs and Optimizations, in Redonda, CE, Brazil, 2016. He pointed out the possibility of decreasing the complexity of a previous version of the recognition algorithm from $O(n^2)$ to $O(n)$.

## References


[1] L. M. S. Bento, D. Boccardo, R. C. S. Machado, V. G. Pereira de Sá, J. L. Szwarcfiter, Towards a Provably Resilient Scheme for Graph-Based Watermarking, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 50–63.

[2] C. Böhm, G. Jacopini, Flow diagrams, turing machines and languages with only two formation rules, Commun. ACM 9 (1966) 366–371.

[3] C. Collberg, S. Kobourov, E. Carter, C. Thomborson, Error-correcting graphs for software watermarking, Lecture Notes in Computer Science 2880 (2003) 156–167.

[4] O.-J. Dahl, C. A. R. Hoare, Structured programming, Academic Press Ltd., London, UK, UK, 1972, pp. 175–220.

[5] E. W. Dijkstra, Letters to the editor: Go to statement considered harmful, Commun. ACM 11 (1968) 147–148.

[6] E. W. Dijkstra, Structured programming, Academic Press Ltd., London, UK, UK, 1972, pp. 1–82.

[7] N. Fenton, R. Whitty, A. Kaposi, A generalised mathematical theory of structured programming, Theoretical Computer Science 36 (1985) 145–171.

[8] D. Harel, On folk theorems, Commun. ACM 23 (1980) 379–389.

[9] M. S. Hecht, J. D. Ullman, Flow graph reducibility, in: Proceedings of the Fourth Annual ACM Symposium on Theory of Computing, STOC '72, ACM, New York, NY, USA, 1972, pp. 238–250.

[10] M. S. Hecht, J. D. Ullman, Characterizations of reducible flow graphs, J. ACM 21 (1974) 367–375.

[11] P. Henderson, R. Snowdon, An experiment in structured programming, BIT Numerical Mathematics 12 (1972) 38–53.

[12] C. A. R. Hoare, Structured programming, Academic Press Ltd., London, UK, UK, 1972, pp. 83–174.

[13] D. E. Knuth, Structured programming with go to statements, ACM Comput. Surv. 6 (1974) 261–301.

[14] D. E. Knuth, R. W. Floyd, Notes on avoiding "go to" statements, Inf. Process. Lett. 1 (1971) 23–31.

[15] D. E. Knuth, J. L. Szwarcfiter, A structured program to generate all topological sorting arrangements, Information Processing Letters 2 (1974) 153–157.

[16] S. R. Kosaraju, Analysis of structured programs, Journal of Computer and System Sciences 9 (1974) 232–255.

[17] D. Kozen, W.-L. D. Tseng, The böhm–jacopini theorem is false, propositionally, in: P. Audebaud, C. Paulin-Mohring (Eds.), Mathematics of Program Construction: 9th International Conference, MPC 2008, Marseille, France, July 15-18, 2008. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 177–192.

[18] T. J. McCabe, A complexity measure, in: Proceedings of the 2Nd International Conference on Software Engineering, ICSE '76, IEEE Computer Society Press, Los Alamitos, CA, USA, 1976, p. 407.

[19] G. Oulsnam, Unravelling unstructured programs, The Computer Journal 25 (1982) 379–387.

[20] R. Tarjan, Depth first search and linear graph algorithms, SIAM Journal on Computing 1 (1972) 146–160.

[21] R. Tarjan, Testing flow graph reducibility, in: Proceedings of the Fifth Annual ACM Symposium on Theory of Computing, STOC '73, ACM, New York, NY, USA, 1973, pp. 96–107.

[22] R. Tarjan, Finding dominators in directed graphs, SIAM Journal on Computing 3 (1974) 62–89.

[23] M. H. Williams, Generating structured flow diagrams: the nature of unstructuredness, Computer Journal 20 (1977) 45–50.

[24] M. H. Williams, Flowchart schemata and the problem of nomenclature, Computer Journal 26 (1983) 270–276.

[25] M. H. Williams, H. L. Ossher, Conversion of unstructured flow diagrams to structured form., Computer Journal 21 (1978) 101–107.

[26] N. Wirth, Program Development by Stepwise Refinement, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 149–169.

[27] W. A. Wulf, A case against the goto, in: Proceedings of the ACM Annual Conference - Volume 2, ACM '72, ACM, New York, NY, USA, 1972, pp. 791–797.