On–Line Construction of Compact Directed Acyclic Word Graphs [*]

Shunsuke Inenaga[1], Hiromasa Hoshino[1], Ayumi Shinohara[1], Masayuki Takeda[1], Setsuo Arikawa[1], Giancarlo Mauri[2], and Giulio Pavesi[2]

[1] Dept. of Informatics, Kyushu University, Japan
{s-ine,hoshino,ayumi,takeda,arikawa}@i.kyushu-u.ac.jp
[2] Dept. of Computer Science, Systems and Communication, University of Milan–Bicocca, Italy
{mauri,pavesi}@disco.unimib.it

Abstract. A Compact Directed Acyclic Word Graph (CDAWG) is a space–efficient text indexing structure, that can be used in several different string algorithms, especially in the analysis of biological sequences. In this paper, we present a new on–line algorithm for its construction, as well as the construction of a CDAWG for a set of strings.

1 Introduction

Several different string problems, like those deriving from the analysis of biological sequences, can be solved efficiently with a suitable text–indexing structure. Perhaps, the most widely used and known structure of this kind is the *suffix tree*, that can be built in linear time and permits to efficiently find and locate all the substrings of a given string. The main drawback of suffix trees is the additional space required to implement the structure. In many applications, like sequence analysis and pattern discovery in biological sequences, keeping as many data as possible in main memory might provide significant advantages. This fact has led to the introduction of more space–efficient structures, like *suffix arrays* [1], *suffix cacti* [2], and others.

In this work, we focus our attention on the *Compact Directed Acyclic Word Graph* (CDAWG), first described in [3]. The CDAWG for a string can be seen either as a compaction of the *Directed Acyclic Word Graph* (DAWG) [4], or a minimization of the suffix tree, from which it can be derived as shown in [3, 5] for DAWGs and [6] for suffix trees. In the latter case, the basic idea is to merge redundant parts of the suffix tree (see Fig. 1). Experimental results [3, 5] have shown how CDAWGs provide significant reductions of the memory space required by suffix trees and DAWGs when applied to genomic sequences. A linear time algorithm for the direct construction of the CDAWG of a string is presented in [5], so to avoid the additional space required by the preliminary construction of

[*] The results described in this work were reached independently by the Kyushu and Milan groups, submitted simultaneously to the conference, and merged into a joint contribution.

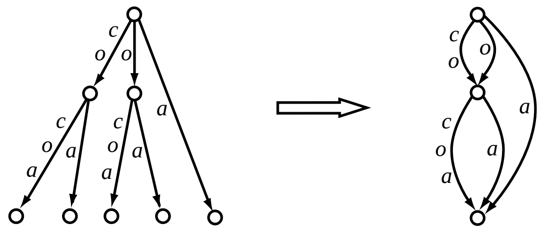

Fig. 1. Suffix tree and CDAWG for string *cocoa*. Substrings *co* and *o* occur as prefix of the same suffixes: the corresponding nodes are merged as well as the subtrees rooted at the nodes. Leaves are merged into a single final node.

the DAWG or the suffix tree. The algorithm is similar to McCreight's algorithm for suffix trees [7]. In this paper, we present a new algorithm for the construction of CDAWGs, based on Ukkonen's algorithm for suffix trees [8]. The algorithm is *on–line*, that is, it processes the characters of the string from left to right one by one, with no need to know the whole string beforehand. Furthermore, we show how the algorithm can be used to build a CDAWG for a set of strings, a structure first described in [3], where was derived by compacting a DAWG for a set of strings. The main drawback of this approach was the fact that, when a new string was added to the set, the DAWG had to be built again from scratch. Instead, the algorithm we present allows to add a new string directly to the compact structure.

2 Definitions

Let Σ be a nonempty finite alphabet, and Σ^* the set of strings over Σ. If $s = \alpha\beta\gamma$, with $\alpha, \beta, \gamma \in \Sigma^*$, then α is a prefix of s, γ is a suffix of s, and α, β, and γ are substrings (factors) of s. If $s = s_1 \ldots s_n$ is a string in Σ^*, $|s|$ denotes its length, and $s[i..j]$ its substring $s_i \ldots s_j$. With $Suf(s)$ we will denote the set of all suffixes of s. Let X be a subset of Σ^*. For any string $u \in \Sigma^*$, $u^{-1}X = \{x \mid ux \in X\}$. Given a string s, we define the syntactic congruence on Σ^* associated with $Suf(s)$ and denoted by $\equiv_{Suf(s)}$ as:

$$u \equiv_{Suf(s)} v \iff u^{-1}Suf(s) = v^{-1}Suf(s) \quad \text{(for any } u, v \in \Sigma^*\text{)}$$

That is, u and v occur as prefixes of the same suffixes of s. In other words, the occurrences of u and v must end at the same positions in the string. Hence, if u and v occur in the string, one must be a suffix of the other. As in [3, 5], we will call *classes of factors* the congruence classes of the relation $\equiv_{Suf(s)}$. The class of all strings that are not substrings of s is called the *degenerate* class. The longest string in a non–degenerate class of factors is the *representative* of the

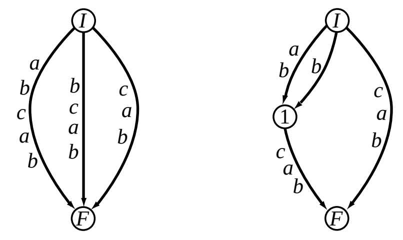

Fig. 2. Implicit CDAWG and CDAWG for string *abcab*.

class. Given a non–degenerate class of factors C of $\equiv_{Suf(s)}$, and its representative u, if there are at least two characters $a, b \in \Sigma$ such that ua and ub are substrings of s, then C is a *strict class of factors* of $\equiv_{Suf(s)}$. From now on, we will say that two substrings are *strictly congruent* if they belong to the same strict class of factors. We are now ready to give a formal definition of a CDAWG.

Definition 1. *The* compact directed acyclic word graph *(CDAWG) of a string s is a directed acyclic graph, where:*

1. *two distinct nodes are marked as initial and final;*
2. *edges are labeled with non empty substrings of s;*
3. *labels of two edges leaving the same node cannot begin with the same character;*
4. *every suffix of s corresponds to a path on the graph starting from the initial node and ending at a node, such that the concatenation of the edge labels on the path exactly spells the suffix. From now on, we will call a node corresponding to a suffix of s* terminal node*;*
5. *substrings spelled by paths starting from the initial node and ending at the same non–terminal node of the graph belong to the same strict class of factors.*

The CDAWG of a string s has at most $|s| + 1$ nodes and $2|s| - 2$ edges [3, 5]. According to the definition of a strict class of factors, non–terminal nodes must have at least two outgoing edges. We will denote with (p, α, q) the edge $p \rightarrow q$ of the graph labeled with substring α. The following definitions will be useful throughout the paper:

Definition 2. *The* implicit *CDAWG of a string s is a CDAWG where nodes with outdegree one are removed, and each edge entering a node with outdegree one is merged with the edge leaving it.*

In the implicit CDAWG of a string s, the suffixes of s are spelled out by paths in the graph starting at the initial node, but not necessarily ending at a node. An example is shown in Fig. 2. For every node p, let $length_s(p)$ be the length of the longest substring spelled by a path from the initial node to p. Edges belonging to the spanning tree of the longest paths from the initial node are called *solid* *edges*. In other words, an edge (p, α, q) is solid iff $\mathit{length}_s(q) = \mathit{length}_s(p) + |\alpha|$. Finally, we assume that the label of each edge is implemented with a pair of integers denoting the starting and ending points in the string of the substring corresponding to the label, and every node is annotated with the length of the longest path from the initial node.

3 Construction of the CDAWG for a Single String

Given an alphabet Σ, let $s = s_1 \dots s_n$ be a string on Σ. Our algorithm is divided in n phases, building at each phase i the implicit CDAWG $\mathcal{G}_i$ for each prefix $s[1..i]$ of s. More in detail, the implicit CDAWG $\mathcal{G}_{i+1}$ for $s[1..i+1]$ is constructed starting from graph $\mathcal{G}_i$ for $s[1..i]$. Each phase $i+1$ is divided in $i+1$ extensions, one for each of the $i+1$ suffixes of $s[1..i+1]$. In extension j of phase $i+1$, the algorithm finds the end of the path from the initial node labeled with substring $s[j..i]$, and extends it by adding character s_{i+1} to the path, unless it is already there. Therefore, in phase $i+1$, substring $s[1..i+1]$ is first put on the graph, followed by $s[2..i+1]$, $s[3..i+1]$, and so on. Extension $i+1$ of phase $i+1$ adds the single character s_{i+1} after the initial node. The initial graph $\mathcal{G}_1$ has one initial node I and one final node F, connected by an edge labeled by character s_1. The algorithm can be sketched as follows:

```
1. Construct graph $\mathcal{G}_1$
2. For $i$ from 1 to $n-1$ do
3.     For $j$ from 1 to $i+1$ do
4.         Find the end of the path from $I$ labeled $s[j..i]$
5.         Add character $s_{i+1}$ if needed
6.     End for
7. End for
```

At extension j of phase $i+1$, once the end of the path spelling $s[j..i]$ has been located, the CDAWG can be updated according to three different rules:

1. In the current graph, the path spelling $s[j..i]$ ends in F. To update the graph, character s_{i+1} is appended to the label of the edge entering F.
2. The path corresponding to $s[j..i]$ does not continue with s_{i+1}, but continues with at least one character c. If the path ends at a node p, we create a new edge (p, s_{i+1}, F). Otherwise, we create a new node q at the end of the path, splitting the edge in two at the point where the path ends. Then, we create a new edge (q, s_{i+1}, F).
3. Some path at the end of $s[j..i]$ continues with s_{i+1}. In this case, substring $s[j..i+1]$ is already in the current graph: we do nothing (hence the implicit graph).

These rules, however, do not guarantee that at the end of the phase we correctly constructed a CDAWG. In fact, the algorithm must also check whether a substring strictly congruent to another one has been encountered, or, conversely,

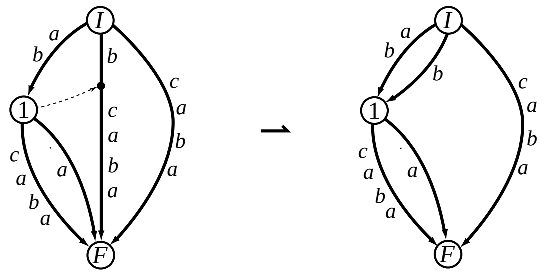

Fig. 3. Implicit CDAWG for string *abcaba* before (left) and after redirection of an edge, at phase 6, extension 5. Node 1, labeled *ab*, was created at the previous extension, after the insertion of *a* at the end of the path labeled *ab*. Now, path corresponding to *b* is found ending in the middle of non–solid edge $(I, bcaba, F)$, that is redirected to node 1 and becomes $(I, b, 1)$.

whether a substring has to be removed from a strict class of factors, so that at the end of phase $i+1$ paths ending at the same node correspond to strict classes of factors of $s[1..i+1]$, and vice versa. Here we sketch how the algorithm has to be modified. A more detailed description of the algorithm and its implementation can be found in [9].

Detecting strictly congruent factors. Two substrings α and β belong to the same class C iff they are prefixes of the same suffixes, and there are at least two characters $a, b \in \Sigma$ such that αa, αb, βa, and βb occur in s. Moreover, α must be a suffix of β, or vice versa. We suppose w.l.o.g. that $\alpha = c\beta$, with $c \in \Sigma$. We also assume that α and β have occurred just once, that substrings αa and βa have been put in the graph in some previous phase (in two consecutive extensions), and in the current extension we have to insert αb. The path spelling α ends in the middle of an edge, and the next character on the edge is a. A new node p is created at the end of the path, as well as a new edge (p, b, F). At the following extension, we have to locate β in the graph. If β has occurred only once (together with α), it now belongs to the same strict class of factors, and we end in the middle of a *non–solid* edge that continues with a. In this case, we *redirect* the edge to p, labeling it with the part of the label that was contained in the path of β (see Fig. 3). Since there can be more than two consecutive substrings to be assigned to the same class, it is possible that we again end along non–solid edges in the following extensions. In this case, we redirect the non–solid edges to p as well, until we reach an extension where we end at a node or along a solid edge. Otherwise, if β had previously occurred also by itself, either the path corresponding to β ends at a node (β has been followed by characters different from a), or the edge we end on is solid (β had been followed only by a). In the former case, if there is not an edge labeled b leaving the node we create a new edge labeled b to the final node. In the latter case, we create a new node and

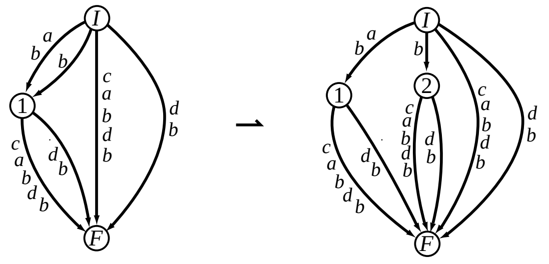

Fig. 4. CDAWG for string *abcabdb* at phase 7, extension 7. Character *b* is found at the end of the non–solid edge $(I, b, 1)$. At extension 6, the path spelling *db* ended at the final node. Thus, *b* has to be removed from the class associated with node 1, that is cloned into node 2. Edge $(I, b, 1)$ becomes $(I, b, 2)$.

connect it to the final node with an edge labeled b. Then, there may be again non–solid edges that have to be redirected into the newly created node.

Splitting a strict class of factors. Conversely, a substring that has been assigned to a strict class of factors has to be removed from the class if it does not occur as a suffix of the representative when a new character s_{i+1} is added to the string. Let α and β, $\alpha = c\beta$, be the two substrings assigned to the same class in the previous example. Now, suppose that in phase $i+1$ we have to insert β in the graph. In this case, s_{i+1} is the last character of β, and we find it at the end of the edge entering node p, that is non–solid, since β is not the representative of the class. Now we have two cases: s_{i+1} was found at the end of an edge that entered node p also at the previous extension, or we ended up somewhere else. In the former case, we had also inserted α at the previous extension of the same phase, therefore β still belongs to the same class. In the latter, we have detected an occurrence of β not preceded by α, that is, not as a suffix of α, and we have to remove it from the class. To reflect this in the graph, we *clone* the node p into a new node q, and redirect the non–solid edge to q keeping the same label. The redirected edge becomes solid. An example is shown in Fig. 4. If also some suffixes of β had been previously assigned to the same class as β, in the following extensions we will again find s_{i+1} at the end of a non–solid edge entering p. These edges are redirected to q. It can be proved that it suffices to check only the last edge on each path to ensure that a class has to be split. No cloning takes place if a character is found at the end of an edge entering the final node.

The two observations outlined above can be implemented in the algorithm by modifying Rules 2 and 3 accordingly. It is worth mentioning that both redirection of edges to a newly created node and node cloning can take place during the same phase. An example is shown in Fig. 5.

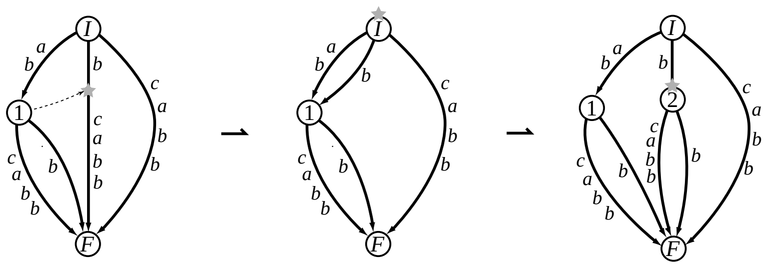

Fig. 5. From left to right, CDAWG for string *abcabb* at phase 6, extensions 5, 6, and 7. Character *b* is put in the graph after substring *ab*, and the path spelling *b* is found in the middle on non–solid edge $(I, bcabb, F)$ (left) that is redirected to node 1 (center). Then, at extension 7 (that adds *b* after the empty string) *b* is found at the end of a non–solid edge. Node 1 is thus cloned into node 2 (right).

3.1 Using Suffix Links

Naively, locating the end of $s[j..i]$ in extension j of phase $i+1$ would take $O(i-j)$ time by walking from the initial node and matching the characters of $s[j..i]$ along the edges of the graph. This would lead to an overall $O(n^3)$ time complexity for the construction of the whole graph. We will now reduce it, as in [8], to $O(n)$ by introducing *suffix links* and with some remarks.

Definition 3. *Let p be a node of the graph, different from the initial or final node. Let β be the representative of the class associated with p. The* suffix link *of p, denoted by $L(p)$, is the node q whose representative γ is the longest suffix of β whose path does not end at p.*

The suffix link of a node p can be implemented with a pointer from p to $L(p)$. If γ is empty, then $L(p)$ is the initial node. Suffix links are not defined for the initial and the final node. Although the definition does not guarantee that every node in the graph has a suffix link, we can prove the following:

Lemma 1. *Any node created during phase $i+1$ will have a suffix link from it by the end of the phase.*

Proof. In extension j of phase $i+1$ a new node p can be created at the end of the path spelling substring $s[j..i]$ by application of Rule 2 or by cloning. In the former case, $L(p)$ will be the first node to be created or encountered at the end of the path corresponding to a suffix of $s[j..i]$ (possibly after edge redirections). Such a node always exists, since the last extension locates the empty suffix at the initial node. In the latter case, let us suppose that a node q is cloned into node p with path spelling $s[j..i+1]$. Substring $s[j..i+1]$ is the longest suffix of the representative of q that does not belong to the same class. Thus, $L(q)$ is set to p. Suffix link $L(p)$ is left undefined until one of the suffixes of $s[j..i+1]$ ends at a node other than p (that again could be I). □

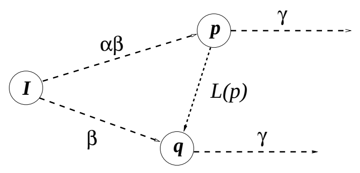

Fig. 6. A suffix link. Node p corresponds to class $\alpha\beta$, node q corresponds to β. Paths labeled with suffixes of $\alpha\beta$ longer than β end at p. If at some extension j character s_{i+1} is added after $\alpha\beta\gamma$, then extensions from $j+1$ to $j+|\alpha|$ are implicitly performed as well.

During any phase, the only node of the graph other than the initial and the final without a suffix link from it is the last created one. Let us suppose that the algorithm has completed extension j of phase $i+1$. Suffix links are used to speed up the search for the remaining suffixes of $s[j..i]$. Starting from the end of $s[j..i]$ in the graph, we walk backwards along the path corresponding to $s[j..i]$ up to either the initial node or a node p that has a suffix link. This requires traversing at most one edge. Let γ be the concatenation of the edge labels of the path from p to $s[j..i]$. If p is not the initial node, we move to node $L(p)$ and follow from it the path spelling γ. Otherwise, we search for $s[j+1..i]$ starting from I. Finally we add s_{i+1} according to one of the extension rules, redirecting an edge or cloning a node if needed. Notice that, if node p is the end of $l \geq 2$ different paths, the position reached after searching from γ from $L(p)$ will be the end of path $s[j+l..i]$, that is, extensions from $j+1$ to $j+l-1$ have been implicitly performed at extension j.

A path spelling γ starting from $L(p)$ always exists, since all the suffixes of $s[j..i]$ are already in the graph. Thus, to find the path spelling γ the algorithm just matches the first characters on the edges encountered. To obtain a linear time algorithm, we need just two more "tricks".

Remark 1. When during any extension Rule 3 is applied, that is, a given substring $s[j..i+1]$ is already on the graph, then the same rule will apply to all further extensions, since all the suffixes of $s[j..i+1]$ are already in the graph as well. Therefore, once Rule 3 is applied (and no node has to be cloned or edges redirected), we can stop and move on to the next phase, since all the strings to be inserted are already in the graph and no adjustment is needed for the classes.

Remark 2. If a new edge is created entering the final node during extension j of any phase i, then Rule 1 will always apply at extension j in any successive phase. That is, new characters will always be appended at the end of the last edge in the path associated with $s[j..i]$, that will enter the final node. Thus, when a new edge is created entering the final node with label $s[j..i+1]$, we label it with integers h and e ($j \leq h \leq i+1$), where e denotes the current phase, that is, the current end position in the string. If we implement e with a global variable, and set it to $i+1$ at the beginning of each phase $i+1$, we perform implicitly all the extensions that would end up at the final node.

Every phase i starts with a series of applications of Rules 1 and 2, that put s_i at the end of an edge entering the final node; when Rule 3 is applied for the first time, it will be also applied to all further extensions. Now, let j_i be the first extension where Rule 3 is applied with cloning in phase i, and j_i^* the first extension where it is applied without edge redirection to the cloned node. Extensions j_i+1 to j_i^*-1 will redirect edges to the last node created. Extensions from j_i^*+1 to i need not to be performed, since in each of them we would not do anything. In phase $i+1$, all extensions from 1 to j_i-1 will apply Rule 1, therefore they are implicitly performed by setting the counter e to $i+1$. Thus, we can start phase $i+1$ directly from extension j_i^*-1, until we find an extension where Rule 3 is applied without cloning or edge redirection. This can be done by starting phase $i+1$ from the position in the graph of the last suffix of $s[1..i]$ that had to be redirected to the cloned node. This took place at extension j_i^*-1. The first extension in phase $i+1$ will have to look for s_{i+1} exactly at the endpoint of the last extension of phase i. This will also implicitly perform all extensions from j_i to j_i^*-1. Of course, if in phase i Rule 3 is first applied without cloning we can move on to phase $i+1$ as well.

The algorithm does not need to know which extension is currently performing. That is, it starts phase $i+1$ from the endpoint of phase i, adding s_{i+1}. Then it starts moving in the graph by using suffix links, and adding s_{i+1} at the end of each path. If the backward walk ends at I, and $\gamma = \gamma_1 \dots \gamma_k$ is the label of the path traversed, then it looks for the path labeled $\gamma_2 \dots \gamma_k$. Phase $i+1$ ends when the algorithm applies for the first time Rule 3 without node cloning or edge redirection. Moreover, whenever we find s_{i+1} at the end of a non–solid edge, we no longer have to check what happened at the previous extension, and just clone the node. In fact, if the representative of the class had been met during one of the previous extensions, we would have stopped the phase at that point, without reaching the current extension.

At the end of phase n, we have constructed the implicit CDAWG for string s. In order to obtain the actual CDAWG, we perform an additional extension phase $n+1$, extending the string to a dummy symbol \$ that does not belong to the string alphabet. Anyway, we do not increment the phase counter e to $n+1$, so to avoid appending \$ to edges entering the final node. Moreover, whenever a new node p has to be created, we do not add the edge $(p, \$, F)$ to the graph. Nodes created in this phase will thus have outdegree one, and will correspond to terminal nodes of the CDAWG. Notice that, whenever a path $s[j..n]$ ends along an edge, we always create a new node and mark it as terminal, while cloning of nodes and redirection of edges work as in the previous phases. When a path $s[j..n]$ ends at a node, we mark the node as terminal. At the end of the additional phase, the implicit CDAWG has been transformed into the actual CDAWG for string s. An example of the on–line construction of a CDAWG is shown in Fig. 7. With arguments analogous to Ukkonen's algorithm for suffix trees, we can prove the following:

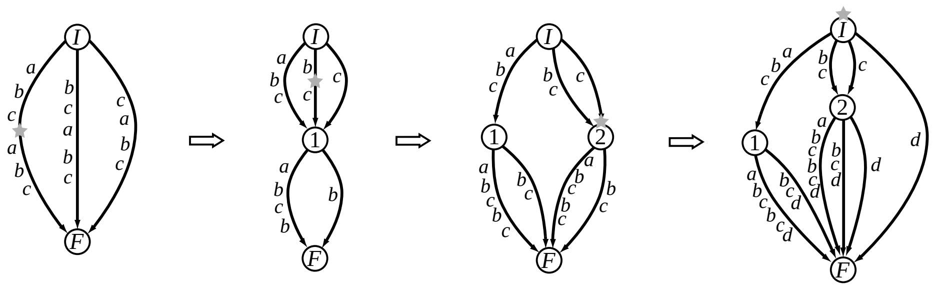

Fig. 7. From left to right, construction of the CDAWG for string *abcabcbcd*: at the end of phase 6 (implicit CDAWG for string *abcabc*); at the end of phase 7 (*abcabcb*, where *abc*, *bc*, and *c* belong to the same strict class of factors); at the end of phase 8 (*abcabcbc*, where *bc* and *c* have been removed from the class with representative *abc*); the final structure. Stars indicate the position in the graph reached at the end of the last explicit extension of each phase.

Theorem 1. *Given a string $s = s_1 \dots s_n$ over a finite alphabet Σ, the algorithm implemented with suffix links and implicit extensions builds the CDAWG for s in $O(n)$ time and $O(n|\Sigma|)$ space if the graph is implemented with a transition matrix, or in $O(n|\Sigma|)$ time and $O(n)$ space with adjacency lists.*

Proof (sketch). The operations performed in any explicit extension (creation or cloning of nodes, edge redirections), that is, extensions that are not performed implicitly by incrementing the e counter, take constant time. Let j_i^* the last explicit extension performed at phase i, and j_{i+1} the first explicit extension performed at phase $i+1$. In the worst case, we have $j_{i+1} = j_i^* - 1$. Moreover, for each i, $j_i \leq j_{i+1}$. Thus, at most $3n$ explicit extensions are performed by the algorithm. At any extension j of phase i, to locate the endpoint of $s[j..i]$ the algorithm walks back at most one edge from the endpoint of $s[j-1..i]$, follows a suffix link, and then traverses some edges checking the first symbol on each edge. If the graph is implemented with a transition matrix, traversing an edge takes constant time. Else, it takes $O(|\Sigma|)$ time. The only thing unaccounted for is the overall number of edges traversed. For every node p of the graph, let the *node depth* of p be the number of nodes on the path from the root to p labeled with the representative of the class associated with p. As in [8], the sum of the node depths counted during all the explicit extensions is reduced at most by $O(n)$, and since the maximum node–depth is n, the maximum number of edges traversed is bounded by $O(n)$. □

4 The CDAWG for a Set of Strings

The basic idea of the CDAWG for a set of strings $S = \{s^1, \dots, s^k\}$ is the same of the single string structure. Now, the nodes of the structure correspond to patterns that occur as prefix of the same suffixes in every string of the set. In

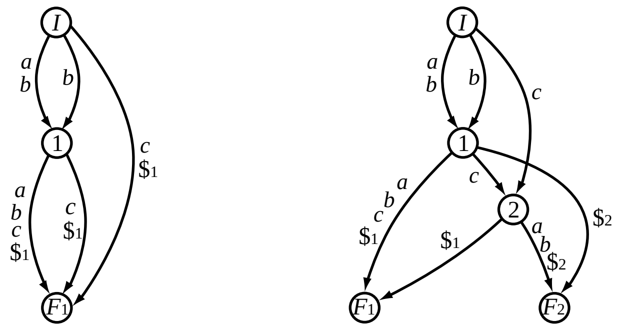

Fig. 8. CDAWG for strings $ababc\$_1$ and $abcab\$_2$, after the insertion of $ababc\$_1$ (left) and $abcab\$_2$ (right). Characters $\$_1$ and $\$_2$ are used as terminations. Edges $(I, \$_1, F_1)$ and $(I, \$_2, F_2)$ have been omitted.

other words, given $Suf(S)$ (the set of the suffixes of the k strings), the nodes of the CDAWG correspond to strict classes of factors for $\equiv_{Suf(S)}$. The only difference is that now we have k final nodes $F_1 \dots F_k$, one for each string, and we want all the suffixes of s^i to end at the corresponding final node F_i. This result can be obtained by appending a different termination symbol, not belonging to the string alphabet, to each string of the set. More formally:

Definition 4. *The CDAWG for a set of strings $s^1 \dots s^k$ is a directed acyclic graph, with a node marked as initial and k distinct nodes $F_1 \dots F_k$ marked as final. Edges are labeled with non empty substrings of at least one of the strings. Labels of two edges leaving the same node cannot begin with the same character. For every string s^i in the set, all suffixes of s^i are spelled by patterns starting at the initial node and ending at node F_i. Paths ending at non final nodes correspond to strict classes of factors of the congruence relation $\equiv_{Suf(S)}$.*

The CDAWG for a set of strings can be constructed with the algorithm presented in the previous section. First, we build the CDAWG for string s^1 (with the termination symbol) and final node F_1. Notice that, since the termination symbol does not occur anywhere else in s^1, the resulting structure is a CDAWG, with no need to perform the additional phase. Then, string s^2 is added to the graph, but in this case with final node F_2. The same will apply to every other string in the set. Node cloning and edge redirection rules ensure the correctness of the resulting structure. It can be proved that the algorithm takes $O(N)$ time to construct the structure, implemented with a transition matrix, where $N = \sum_{i=1}^{k} |s^i|$. This structure (with marginal differences) was first described in [3], where it was built by reducing a DAWG. Therefore, adding a new string to the set required the construction of a new DAWG from scratch. The algorithm presented here, instead, permits to add strings directly to the compact structure (see Fig. 8). As in [3] we can give an upper bound on the size of the structure.

Theorem 2 (Blumer et al., [3]). *The CDAWG for a set of strings $s^1 \dots s^k$, has at most $N + k$ nodes, where $N = \sum_{i=1}^{k} |s^i|$.*

5 Conclusions

A CDAWG is a space–efficient text–indexing structure that represents all the substrings of a string. We presented a new on–line algorithm for its construction, as well as the construction of a CDAWG for a set of strings. The same structures can be computed by reduction starting from the corresponding DAWGs or suffix trees; however, the approach presented in this paper permits to save time and space simultaneously, since the CDAWGs can be built directly. Moreover, once the structure has been built for a set of strings, new strings can be added directly to the compact structure.

Acknowledgements

The Milan–Bicocca group has been supported by the Italian Ministry of University, under the project "Bioinformatics and Genomic Research".

References

1. U. Manber and G. Myers. Suffix arrays: a new method for on–line string searches. *SIAM J. Computing*, 22(5):935–948,1993.
2. J. Kärkkäinen. Suffix cactus: a cross between suffix tree and suffix array. *Combinatorial Pattern Matching*, 937:191–204, July 1995.
3. A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, 1987.
4. A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. Chen, and J. Seiferas. The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55, 1985.
5. M. Crochemore and R. Verin, On compact directed acyclic word graphs, Springer Verlag LNCS 1261, pp.192–211, 1997.
6. D. Gusfield, Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology, Cambridge University Press, New York, 1997.
7. E. McCreight. A space–economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.
8. E. Ukkonen. On–line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.
9. S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, and S. Arikawa, On–line construction of compact directed acyclic word graphs. DOI Technical Report 183, Kyushu University, January 2001.