Online Parsing of Visual Languages
Using Adjacency Grammars*

Joaquim A. P. Jorge
Instituto de Engenharia de Sistemas e Computadores (INESC)
Rua Alves Redol 9, 1000 Lisboa, Portugal
Joaquim.Jorge@inesc.pt

Ephraim P. Glinert
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180, USA
glinert@cs.rpi.edu

Abstract
Visual computing environments continue to grow in importance, yet fast, general parsing algorithms for visual languages remain elusive. In this paper we present an incremental parsing algorithm for a broad class of visual languages which do not contain overlapping elements. Our algorithm is based on the concept of adjacency grammars, where adjacencies are defined so as to encompass both spatial and logical constraints. Our approach combines bottom up and top down methods to support incremental parsing of visual input, allowing for measurably efficient online parsing of diagram-like visual languages, with observed linear run times for large visual sentences.

1 Introduction
In recent years there has been rapidly growing interest among systems designers (and consumers) in new kinds of user interfaces where visual elements, which may contain both text and graphical components, and the spatial relationships among them, take precedence over concatenation and temporal sequence, which are the main organizing principles in string languages [1].

Grammars can be used to define such interfaces, since symbols and productions naturally express the state and flow of a dialogue. Productions can be associated with dialogue entities (application state), and terminal symbols with tokens received from the lower level components of the interface. Furthermore, the ability to associate constraints with productions in a grammar enhances the power of declarative semantics with the general control mechanism provided through parsing.

Grammars are superior to event handling models such as those used for some toolkits, because they are less complicated and cumbersome since there is no need to handle explicit state information. We also can effectively model multithreaded dialogues by using multiple grammars $G_i$ independently and simultaneously. This is equivalent to using a grammar $G$ with productions:

$$\hat{S} \rightarrow S_1 | S_2 | \ldots | S_n$$

where the $S_i$ are start symbols for the grammars $G_i$, respectively. Each of these can be parsed in parallel and associated with a different dialogue thread, to yield separate parse trees and corresponding side effects.

Some authors have argued that the serial arrangement of symbols in productions is an obstacle to order free dialogues. They contend that imposing an arbitrary sequence on the components of a dialogue may be perceived by the user as unnatural [2]. As we shall see, the order free nature of adjacency grammars makes them an ideal formal mechanism for describing user interface behavior. Another useful feature of these grammars is that using visual tokens to represent both user generated input and system output (i.e., feedback) enables interface developers to treat input and output as symmetrical entities. This greatly simplifies system design through enhanced modularity.

In what follows, we introduce extended adjacency grammars as a foundation for interactive parsing of visual languages. We then describe an incremental parsing algorithm for this class of grammars, discuss how to handle partial input, provide timely feedback and support interactive operations such as delete, undo/redo. Finally, we discuss the complexity of parsing local-
context distance bounded adjacency languages and show how to use spatial enumeration data structures to support efficient parsing of adjacency languages.

Our approach is closely related to the work by Wittenburg et al. [3, 4] on incremental visual language parsing for user interfaces. However, their approach restricts visual and spatial relations to visual terminal symbols, and the resource requirements are unclear. Golin [5] has shown that parsing arbitrary so-called “attributed multiset grammars” is NP-complete. Using an approach based on dynamic programming, he has developed an $O(N^3)$ off-line parsing algorithm for a subset of these grammars which he calls “picture layout grammars.”

Helm and Marriott [6] have taken a logic programming approach to parsing visual languages; a drawback is that when defining grammars one must go to considerable effort to ensure that no backtracking will occur when parsing. Marriott’s Constraint Multiset Grammars [7] combine logic constraints with attributed multiset grammars to produce constraint logic programs which are then used to parse visual sentences, yielding a formal framework and theoretical results. Although this work addresses the role of spatial queries, suggesting that polynomial parsing algorithms are possible for constraint multiset grammars, the problem is stated in a general formulation, and precise complexity bounds are not provided. All of these approaches stress order free parsing of visual sentences and focus on spatial relations as a main component in syntax analysis of visual languages.

Our work with adjacency grammars presented in this paper has produced a visual compiler-compiler which generates C++ code that can be embedded in interactive applications.

2 Example: Directed Graphs

In this section we present an example grammar for the language of directed graphs. This is highly appropriate, since many useful visual languages are based on some kind of graph formalism. Therefore, it is desirable that a general purpose visual language parsing algorithm should be able to analyze such languages efficiently. Graph-like languages embody abstract relationships (a graph is a set of nodes and arcs connecting those nodes), as well as spatial constraints (containment, pointing to, etc) that must be met by the constituents of a visual sentence, as illustrated by Figure 1.

Figure 2 shows a visual sentence in the language described by our example grammar. There are several points worth noting in this example. First, by using contextual symbols we are able to parse graph structures without separate explicit constraints on nodes as arc endpoints, because contextual symbols can participate in several different derivations. This leads to non-tree branches, as exemplified in Figure 7, as happens with remote symbols in Picture Layout Grammars, and existential constraints in Constraint Multiset Grammars [7].

Note that several productions in our grammar are unconstrained. Relieving the grammar writer from having to specify constraints for each and every production is especially desirable when describing structures like graphs. After all, entities such as lists of nodes and arcs don’t have any topological information associated with them, and so could only be artificially constrained. Note that this situation occur only at the “top” of the grammar, i.e., all such unbound productions must be directly reachable from the start symbol. In the example shown, $p_1$ is an unbound production, $p_2$ and $p_3$ are recursive productions, while $p_4$ and $p_5$ are bound productions, since their right-hand-side (RHS)

\begin{align*}
p_1 & \rightarrow \{ \text{nodes, arcs} \} \\
p_2 & \rightarrow \{ \text{nodes, node} \} | \{ \text{node} \} \\
p_3 & \rightarrow \{ \text{arcs, arc} \} | \{ \text{arc} \} \\
p_4 & \rightarrow \{ \text{CIRCLE, TEXT} \} \\
p_5 & \rightarrow \{ \text{context}(\text{node}_1), \text{ARROW}, \text{context}(\text{node}_2) \} \\
& \hspace{1em} \text{if} \text{Contains}(\text{CIRCLE, TEXT}) \\
& \hspace{1em} \text{if} \text{PointsFrom}(\text{node}_1, \text{PointsTo}(\text{node}_2, \text{ARROW}))
\end{align*}

Figure 1: An Extended Adjacency Grammar for Directed Graphs

Figure 2: An Example Visual Sentence
symbols are connected by adjacency relations.

Adjacency constraints play a key role in our grammars to enable efficient parsing of input. Adjacency relations group and strongly connect elements in a production; these are first class concepts in our formalism, enabling especially efficient adjacency driven parsing algorithms. Informally, we can think of an adjacency as a constraint with an associated query function, returning the neighbor set of a given visual symbol. We will discuss these in more detail in a later section.

In contrast with string grammars, the result of parsing a visual sentence using adjacency grammars is a rooted directed acyclic graph (RDAG), in which the root is labeled by the grammar’s start symbol and the leaves are labeled by each input token. The next section provides a more formal definition of adjacency grammars.

3 Adjacency Grammars

Adjacency grammars extend picture layout grammars with adjacency operators and unbound productions. Formally, an adjacency grammar is defined by a 5-tuple:

\[(V_T, V_N, S, P, A)\]

where \(V_T\) is the set of terminal symbols, \(V_N\) is the set of nonterminal symbols, \(S \in V_T\) is the start symbol, and \(P\) is a set of grammar productions of the form:

\[\alpha \vdash \beta_1, \ldots, \beta_n \text{ if } \Gamma(\beta_1, \ldots, \beta_n) \text{ after } \Delta\]

where \(\alpha \in V_N\), and the \(\beta_j \in V_T \cup V_N\) constitute a possibly empty multiset (i.e., an unordered set in which elements may appear more than once [8]) of terminal and nonterminal symbols. \(\Gamma\) is an adjacency constraint defined over the attributes of the \(\beta_j\). \(\Delta\) is a set of expressions over attribute values of the \(\beta_j\), which synthesize attributes of \(\alpha\). \(A\) is a family of adjacency constraints, such that \(\Gamma\) includes only members of \(A\).

There can be zero RHS symbols in a production (i.e., \(\epsilon\) productions are allowed). “Rewriting productions” of the form \(A \rightarrow B\), are also legal.

The symbols \(\beta_j\) can appear in any order, in contrast with context free string grammars, where the order of symbols in a production dictates the concatenation of derived strings in the language. For adjacency grammars, if:

\[PointsFrom(A, B)\]

then we consider all 6 possible permutations of \(\mu, \beta\) and \(\lambda\) as equally valid substitutions for \(\alpha\).

Last, the adjacency constraint on productions which rewrite the start symbol can be omitted, to yield unbound productions, if there are two or more symbols in the RHS (rewriting productions of the form \(A \rightarrow B\) are not bound by adjacency constraints). The LHS of an unbound production is an unbound symbol. Adjacency grammars do not allow unbound symbols to appear in the RHS of bound productions, with the further restriction that bound symbols, cannot appear as the LHS of unbound productions. Bound productions are those which have all their RHS symbols bound by adjacency constraints. For example, in the grammar presented in Figure 1, there is one unbound production:

\[\text{graph} \rightarrow \{ \text{nodes, arcs} \} \]

Unbound productions are useful to establish logical aggregation and set relations between geometrically unrelated items. These useful mechanisms are not present as such in other visual grammar formalisms, which only allow grouping of spatially related items.

4 Adjacencies

Adjacency constraints are used to derive meaning from the relative positions and spatial relations of input tokens. Examples of these are the binary predicates:

\[\text{Contains}(t_1, t_2)\]
\[\text{PointsTo}(t_1, t_2)\]
\[\text{PointsFrom}(t_1, t_2)\]
\[\text{Intersects}(t_1, t_2)\]

some of which are illustrated in Figure 3. Spatial constraints can be viewed as predicates over the geometric
attributes of visual symbols. For example, suppose the extent or bounding rectangle of a token \(a\) is denoted by \(a.b.b\). Then \(\text{Contains}(a, b)\) becomes “\(a.b.b\) contains \(b.b.b\)”.

The concept of adjacency is a very powerful and intuitive one. We have identified four main types of adjacency, all of which are useful in parsing adjacency grammars. We are especially interested in three main kinds of adjacency:

**Algebraic adjacency** relations generalize the intuitive idea that with respect to some relation \(R\) two elements are adjacent “if there is nothing in the middle.” For example, in Figure 4, which illustrates the relation defined by the colloquial expression “\(x\) is immediately to the right of \(y\)”, where we have \(aRb\) and \(bRc\), but not \(aRc\), since \(b\) “stands in the middle”.

To generalize the concept we say that two items \(x\) and \(y\) are adjacent or adjacency related, with respect to some relation \(R\), iff \(\exists z\) such that \(xRz \land zRy\). \(R\) is called an adjacency relation or simply **Adjacency**, if all its members are adjacent (that is, \(R\) is antitransitive). This concept of adjacency is very important to interactive parsing, because it requires us to keep track of which adjacencies were involved in the reduction of a parse item, in case a later item “comes in between” thereby invalidating a derivation and all items built on it.

**Spatial Adjacency** relations are tied to the geometric distance which separates candidate items. The idea of a geometric neighborhood is related to the “spatial clustering” mechanism of human perception—we tend to relate objects that are “close enough.” As an example in Figure 5, label \(T\) is a neighbor of, or associated to circle \(a\), while it is too far apart from circle \(b\) to be associated to it. The concept of spatial adjacency is a very powerful one, because it allows the use of query functions rooted on computational geometry methods to expedite the search of items that may be related to an item under examination, while at the same time pruning large amounts of irrelevant data without looking at them. This distinguishing feature we call **adjacency driven** parsing, as it enables us to analyze efficiently large classes of useful visual languages. In particular, large classes of visual languages allow us to establish a “cut off” distance \(d_a\), beyond which items can no longer be associated. We can then use a data structure such as a uniform grid, whose cell size is related to \(d_a\) to expedite neighbor searches.

**Logical Adjacencies** are a broad class of relations that do not operate on spatial attributes of their arguments. However, logical adjacencies play an important part in visual languages such as complex circuit diagrams and flowcharts: the use of labeled connection points is often used to signify adjacent parts in an electrical network or to join disconnected parts of a flowchart. Here, the concept of adjacency is enforced by the **same-label**(\(x, y\)) relation which compares textual labels associated with different elements of a diagram. Another example of a logical relation is **List**(\(a, a\)) which enables to gather sets or lists of otherwise unrelated elements. These relations can be directly inferred from recursive productions, as illustrated in our example grammar, in which the production

\[
\text{nodes} \rightarrow \{ \text{nodes}, \text{node} \} \mid \{ \text{node} \}
\]

is equivalent to:

\[
\text{nodes} \rightarrow \{ \text{List(nodes, node)} \} \mid \{ \text{node} \}
\]

It is possible to compute **List** adjacencies in constant time, through the use of set attributes associated with each parse item.

Adjacencies make up such a fundamental part of adjacency grammars, that they can specify visual attributes of the left-hand-side item once a production is reduced through **postconditions**. These ensure that geometry attributes of items are properly updated and the resulting visual symbols “look right” after creation.

## 5 A Parsing Algorithm

We will now introduce a simple parsing algorithm inspired by the dynamic programming similar to that of Golin [5], but with several crucial differences and extensions.

Our parsing algorithm stores **parse items** for each production it matches against the input. Each parse
item is labeled by the left-hand-side symbol of the production it reduces. In contrast with Golin’s approach, our invariant is based on the cover set introduced by Wittenburg et al. [3]. The cover set of a parse item is the set of terminal symbols derived by that item. A key point in our algorithm is that no two parse items can have the same production and cover set. This use of cover as a standard attribute allows us to avoid infinite loops in grammars with recursive overlapping productions. Picture layout grammars, on the other hand, rely on the uniqueness of geometric attributes, and thus require extra passes over the parse structure to identify and validate a DAG spanning all input primitives.

In an interactive environment, incremental parsing methods are required in order to provide timely feedback to the user. Our algorithm works online. By this we mean that as each token is scanned, it is immediately possible to tell whether the visual sentence seen so far is accepted or not, just by looking at the parser state. This contrasts with other approaches, which require additional processing after the last token has been read to determine whether or not the given visual sentence is valid.

Our recognition algorithm is presented in Figure 6. Note that extracting a parse RDAG from the parse item list is trivial, if we store back pointers with each parse item. The set $\mathcal{R}$ contains all parse items of the form $[X, \text{cover}(X)]$ reduced by the algorithm. At each step, the work set $\mathcal{W}$ contains the parse items created but not yet matched against others.

Informally, our algorithm builds one or more parse RDAGs using an alternating bottom up / top down approach, from the individual tokens to the start symbol. This is shown in Figure 7, which depicts both a visual sentence and its analysis DAG. There exists a one to one correspondence between the nodes in a parse RDAG and a subset of the parse items, where the cover of a given node is the set of leaf elements (tokens) reachable from that node. A new item only gets reduced if there is no other instance with the same production and cover already in $\mathcal{R} \cup \mathcal{W}$. This ensures algorithm termination, since only a finite number of different items can be added to $\mathcal{R}$. The algorithm uses two different sets to ensure that no pair of parse items $\{X, Y\}$ is ever checked twice.

The core of the algorithm is the $\text{bottomUp()}$ function, which matches items from the work set ($\mathcal{W}$) with their neighbors in the reduced ($\mathcal{R}$) set at each parse step.

$$\text{bottomUp}(\mathcal{W}, \mathcal{R}) = \begin{array}{l}
\text{repeat} \\
\forall (X \in \mathcal{W}) \\
\forall (p: \{A \to \{X, Y\} \mid \Gamma \} \in P) \\
\forall (Y \in \mathcal{R} \cap \text{neighbors}(X, \Gamma)) \\
\quad \text{if } \text{cover}(X) \cap \text{cover}(Y) = \emptyset \\
\quad \chi = \text{cover}(X, p) \cup \text{cover}(Y, p) \\
\quad \text{if } \text{unique}([p, \chi]) \\
\quad \mathcal{W} = \mathcal{W} \cup \{[p, \chi]\} \\
\quad \text{follow}(\mathcal{W}, A); \\
\mathcal{R} = \mathcal{R} \cup \{X\}; \mathcal{W} = \mathcal{W} \setminus \{X\}
\end{array} \text{ until } \mathcal{W} = \emptyset;$$

This bottom up step is a departure from previous methods, in that only neighbor items are matched against each candidate item extracted from the work set $\mathcal{W}$. Without this restriction the algorithm’s performance would be limited by a quadratic lower bound dependency on the number of tokens, regardless of the nature of the language being parsed. Since each adjacency relation has an associated query function, we use that query function to return the neighbor set of a candidate item, for each production in a grammar. This is simply the set of items $Y$ that satisfy the adjacency criterion $\Gamma(X, Y)$ for the parse item $X$. We will see ahead how to compute neighbor sets efficiently using spatial enumeration data structures.

$\text{bottomUp}$ matches a pair of parse items against all productions in the grammar $G$, and creates a new parse item if all constraints are met. Both pairs $\{X, Y\}$ and $\{Y, X\}$ are tested to ensure that a production will fire regardless of the order in which its constituents appear. In other words, a valid parse RDAG will be constructed by the algorithm, if it exists, given any permutation in the order of leaf nodes (tokens). This latter property exactly characterizes the order free nature of this parsing method.

$$\text{unique}(X) \equiv [X, p, X.\text{cover}] \not\in (\mathcal{R} \cup \mathcal{W})$$
The function unique() ensures that all parse items reduced are unique up to their production and cover set. Note that we could still factor all derivations with the same production and cover if we wanted to produce all possible analysis RDAGs for ambiguous visual sentences and grammars.

Function cover ensures that a context item does not contribute to the cover set of any items which reduce it.

\[\text{cover}(X, P) \equiv \begin{cases} \emptyset & \text{if } (X, \text{sym is a contextual symbol in production } P) \text{ then } \emptyset \text{ else } X.\text{cover}; \\ \end{cases}\]

follow() recursively tracks sequences of productions of the form \(a \rightarrow \beta\). It is called after each new parse item is created by bottomUp. The resulting parse items are added to the work set \(W\), for further matching. Note that since all items are tested for uniqueness this function cannot enter an endless loop, even if the grammar has a (silly) production of the form \(A \rightarrow A\):

\[\text{follow}(W, X) \equiv \begin{cases} \forall (p : [A \rightarrow \{X\}] \in P) \\ \text{if } (X, \text{unique}(\{p, \text{cover}(X, p)\})) \\ \text{then } W \leftarrow W \cup \{[p, \text{cover}(X, p)]\}; \\ \text{follow}(W, A); \\ \end{cases}\]

Informally, we say that an input visual sentence \(\omega \in L(G)\) for an adjacency grammar \(G\), if and only if there exists a RDAG whose root node is labeled by the start symbol \(S\) and whose leaf nodes correspond to the elements of \(\omega\). From the brief description above, it should be easy to see that a one to one correspondence can be established between a subset of \(\mathcal{R}\) and the nodes in a parse RDAG, and that the algorithm will trace such a parse RDAG from the leaves up.

\[\text{topDown}(W, \mathcal{R}) \equiv \begin{cases} \forall (\text{unbound productions } p : S \rightarrow \beta_1, \ldots, \beta_n) \\ \text{find maximal cover sequence } \beta_{M1}, \ldots, \beta_{Mn}; \\ \text{s.t. } \forall i \neq j \text{ cover}(\beta_{Mi}) \cap \text{cover}(\beta_{Mj}) = \emptyset \\ \chi \leftarrow \bigcup_{i=1}^n \text{cover}(\beta_{Mi}) \\ \text{if } (\text{unique}(\{p, \chi\})) \\ \mathcal{R} \leftarrow \mathcal{R} \cup \{[p, \chi]\} \\ \end{cases}\]

Bottom up parsing can only be efficiently applied to pure adjacency grammars (i.e., those without unbound productions). If we were to apply bottom up parsing methods to unbound productions, the algorithm would take exponential time to enumerate all possible covers reducing a given production. Unbound productions are handled using a top down match implemented by topDown, which attempts to rewrite the start symbol as a sequence of bound symbols by finding a maximal disjoint sequence of bound items that reduce the start symbol.
6 Uniform Grids for Queries

As we have implied in previous sections, adjacency driven parsing takes advantage of query functions associated with adjacencies, to expedite the search for items possibly adjacent to the one being considered at a given time. In this section we discuss how to use computational geometry methods to accomplish this. Our approach to adjacency driven parsing was motivated by previous work and results on lexical composition systems [9] which sped up the search for lexical components in visual language systems through plane sweep methods [10]. The only disadvantage that plane sweep methods possess besides implementation complexity, lies in that they do not lend themselves to online processing, which is contrary to our main motivation.

We decided to implement spatial proximity queries using a uniform grid. Uniform grids, besides being simple to implement, exhibit locality of reference which is a desirable property in virtual memory implementations, and require little computational resources to implement queries, which translates to fast operations. A uniform grid is an array of cells which tile the visual space, such that each cell has the same size. We have found it simple to implement adjacency constraints which modify and query this simple data structure, mapping the visual primitives into three distinct categories, based on their geometry attributes (points, closed shapes and lines). Insertion and adjacency query methods for general primitives specialize one of the six combinations of these types of primitives.

The space requirements for the grid data structures are \( O\left(\frac{|\omega|}{\text{cell size}}\right) \), where \(|\omega|\) is the number of primitives in a visual sentence, \( L \) is the “typical” visual element dimension. We have found in practice that the grid data structure is tolerant to disparities of one order of magnitude between visual element and cell sizes. Within these constraints we can say that the simplest distance bound adjacency queries such as those shown in Figure 3, take up \( O(1) \) time per pair of elements tested.

7 Parsing Complexity

Strong performance results are hard to provide for general families of visual languages, given the number of degrees of freedom most formalisms have. Because of this, visual language families are difficult to characterize and visual language formalisms are difficult to compare. We are interested in characterizing a family of visual languages that can be efficiently analyzable using our parsing algorithm. We call such languages context limited deterministic adjacency languages or CLD\((k)\). The index \( k \) refers to an incremental property of parse items: for any given parse item and a grammar production \( p \), there are at most \( k \) other items in its neighbor set with regard to \( p \). When applied to the symbols in a visual sentence, this translates to a measure of visual ambiguity. In a CLD\((1)\) language, each visual symbol in a visual sentence is related to at most one other symbol. This requires not only that the grammar be unambiguous, but also precludes ambiguous arrangements of visual symbols!

When parsing CLD\((1)\) grammars using our algorithm, the total number of parse items created is \( O(|G| |\omega|) \), so the number of parse items reduced will be \( O(|G|) \) for each new token read. This yields \( O(|G|) \) production lookups per iteration in the bottom up step. List adjacencies use cover sets to achieve \( O(1) \) set intersection query, so that CLD\((1)\) grammars can include recursive productions. It is also possible to find a maximally disjoint pair reducing the topmost unbound production \( p_1 \) with \( O(1) \) queries, so that the grammar shown in Figure 1 describes a CLD\((1)\) language.

Parsing CLD\((1)\) languages will require \( O(|\omega|) \) set operations and adjacency queries. This is consistent with our experimental findings. These establish the superiority of adjacency driven parsing methods over “pure” syntax driven methods which suffer from a quadratic lower bound complexity on sentence size and are not suited to interactive parsing.

8 Performance Data

Performance data for the algorithm described in this paper, when applied to the grammar of Figure 1, is shown in Table 1. Memory requirements and elapsed (wall clock) time were measured on an i486 PC running Linux. Visual sentences were manually generated with xfig and run through the incremental parser.

The algorithm’s memory requirements are roughly proportional to the input visual sentence size (number of tokens). The observed run time complexity is sub-quadratic with the number of tokens. Interestingly, even for the largest case (1634 tokens) the time requirements average just 3 milliseconds of elapsed time per token, which is still quite usable for interactive applications.

Although the experimental results support the conclusion that a CLD\((1)\) language can be parsed with a
linear number of set operations, the set representation we have used (bit vectors) takes $O(n)$ operations for the set intersection, union and equality testing. While this behavior is negligible for “small” sets due to word level parallelism exploited by this representation, overhead for larger sets (812 members) becomes significant, and our algorithm exhibits quadratic behavior. While these considerations are irrelevant for small visual sentence size, typical of many interactive applications, they may be relevant for visual sentences with tens of thousands of primitives, e.g., in complex diagram analysis and map understanding applications.

<table>
<thead>
<tr>
<th>Tokens read</th>
<th>11</th>
<th>57</th>
<th>81</th>
<th>102</th>
<th>203</th>
<th>406</th>
<th>812</th>
<th>1034</th>
<th>1937</th>
<th>4068</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parse Items created</td>
<td>34</td>
<td>184</td>
<td>279</td>
<td>349</td>
<td>694</td>
<td>1389</td>
<td>2779</td>
<td>3545</td>
<td>6644</td>
<td>13938</td>
</tr>
<tr>
<td>Constraints checked</td>
<td>17</td>
<td>108</td>
<td>165</td>
<td>230</td>
<td>460</td>
<td>930</td>
<td>1835</td>
<td>2330</td>
<td>4327</td>
<td>9103</td>
</tr>
<tr>
<td>Grammar lookups</td>
<td>67</td>
<td>367</td>
<td>557</td>
<td>897</td>
<td>1387</td>
<td>2777</td>
<td>5547</td>
<td>7089</td>
<td>13287</td>
<td>27877</td>
</tr>
<tr>
<td>Memory (Kbytes)</td>
<td>213</td>
<td>248</td>
<td>270</td>
<td>285</td>
<td>374</td>
<td>561</td>
<td>1011</td>
<td>1340</td>
<td>2254</td>
<td>6028</td>
</tr>
<tr>
<td>Parse Time (s)</td>
<td>0.03</td>
<td>0.09</td>
<td>0.13</td>
<td>0.17</td>
<td>0.34</td>
<td>0.75</td>
<td>1.93</td>
<td>2.8</td>
<td>7.01</td>
<td>26.99</td>
</tr>
<tr>
<td>Parse Time/token(ms)</td>
<td>3</td>
<td>2</td>
<td>1.6</td>
<td>0.8</td>
<td>1.67</td>
<td>1.8</td>
<td>2.4</td>
<td>2.7</td>
<td>3.6</td>
<td>6.6</td>
</tr>
</tbody>
</table>

Table 1: Experimental Results Using Adjacencies

9 Discussion

We have described an online parsing algorithm for a general class of visual languages, which extends and enhances previous approaches to allow parsing graph-like visual languages. The chief advantages of the approach described are (a) the algorithm’s simplicity, and (b) the ease with which visual grammars can be specified and efficiently parsed. Extended adjacency grammars support fast parsing of some visual languages, through relational adjacencies which enable efficient analysis of visual expressions.

We have achieved subquadratic order-free parsing of deterministic context languages using adjacency queries supported by a uniform grid data structure.

We believe there can be hierarchies of formal visual languages drawn on the expressive power of spatial and logic constraints, which constitute the fundamental distinctive feature separating visual languages from string languages. Further, we believe that further gains in efficiency can be obtained through a better combination of bottom up and top down strategies to yield better and faster parsing algorithms, or to widen the families of languages that can be efficiently parsed.

References