Efficient Incremental Parsing for Context-Free Languages

Manuel Vilares Ferro

Computer Sciences Department
University of La Coruña
A Coruña, Campus de Elviña 15071, Spain

Bernard Andre Dion

Simulog
Sophia Antipolis, HB2 Dolines 06901, France

Abstract

An incremental parsing algorithm based on dynamic programming techniques is described[1]. *The analyzer takes the automaton generated from a general class of context-free grammars as driver, and any finite string as input. Given an input string that has been modified, the algorithm cuts out the parts of the old analysis that had been generated by the parts of the input that has changed. What remains are the parts of the analysis that were generated by the stable part of the input. The new system has been baptized ICE, after* ***I****ncremental* ***C****ontext-Free* ***E****nvironment. In an empirical comparison, ICE appears to be superior to the other context-free analyzers and comparable to standard deterministic parsers, when the input string is not ambiguous.*

Key Words and Phrases: *Context-Free Parsing, Dynamic Programming, Earley Parsing, Incremental Parsing, Parse Forest, Push-Down Automata.*

1 Introduction

Reasons to achieve incremental parsing are well known. For example, in the context of incremental program development, several consecutive corrections of the input are usually made. Therefore incremental parsing can be used to make the overall parsing process efficient. In this paper, we try to reconcile, in practice, general context-free parsing and incremental treatment.

1.1 Previous work

In relation to context-free parsing, we chose to work in the context of the application of deterministic techniques to generate very efficient non-deterministic parsers, as proposed by Lang in [11, 12]. Essentially, Lang considers a simple variation of Earley's dynamic programming construction [6]. In order to solve the problems derived from grammatical constraints, the author extends Earley's classic algorithm to push-down transducers (PDTs), separating the execution strategy from the implementation of the push-down automaton (PDA) interpreter. So, Lang obtains a family of general context-free parallel parsers for each family of deterministic context-free push-down ones, simulating all possible computations of any PDT, at worst in cubic time. However, the method is linear in a large class of grammars.

In relation to incremental parsing, the classic literature [1, 4, 7, 9] does not consider the case of context-free grammars. So, we have studied the state of the art in the deterministic case and compared the adaptability of the different methods to dynamic programming techniques. Finally, the algorithm we developed is closer to the bottom-up algorithm of Ghezzi and Mandrioli in [7] than to the top-down one of Jalili and Gallier in [9]. The reason for that is the natural predisposition of Earley-like algorithms to incremental bottom-up approaches. Effectively, most of the additional structures required for this class of incrementality, are provided at no expense by the item concept in Earley's method, essentially a structure representing the state of the parsing process for a branch of the parse forest, at each point in the scan.

1.2 A simple road map

In section 2 of this paper, we discuss how we have implemented an efficient general context-free parser based on the algorithm reported by Lang in [11]. In section 3, we describe the incremental algorithm considered in the ICE system, justifying tactical decisions from a practical point of view and differentiating two cases: *total* and *partial recovery*, with respect to the nature of the modified shared

[1] this work was partially supported by the **E**ureka **S**oftware **F**actory project.

parse forest. In section 4, we give an extensive range of comparative tests between ICE and the best deterministic and non-deterministic parsers. Section 5 is a conclusion about the work presented. Finally, appendices A and B review from a practical point of view all the theoretical results previously explained in relation with standard and incremental parsing.

2 Context-Free Parsing

The following is an informal overview of parsing by dynamic programming interpretation of PDAs, such as it is implemented in ICE. Our aim is to parse sentences in the language $\mathcal{L}(\mathcal{G})$ generated by a context-free grammar $\mathcal{G} = (N, \Sigma, P, S)$, where N is the set of non-terminals, Σ the set of terminal symbols, P the rules and S the start symbol. The empty string will be represented by ε.

2.1 The recognizer

We assume that, using a standard technique, we produce a PDA from the grammar $\mathcal{G}$. In practice, we chose a LALR(1) recognizer, possibly non-deterministic, for the language $\mathcal{L}(\mathcal{G})$. This choice will be justified later in the paper.

Formally, we shall assume for PDAs a formal definition taken from Lang in [11], that can fit most usual construction techniques. So, a PDA is represented by a 7-tuple $\mathcal{A} = (\mathcal{Q}, \Sigma, \Delta, \delta, q_0, Z_0, \mathcal{Q}_f)$ where: $\mathcal{Q}$ is the set of states, Σ the set of input symbols, Δ the set of stack symbols, q_0 the initial state, Z_0 the initial stack symbol, $\mathcal{Q}_f$ the set of final states, and δ a finite set of transitions of the form $p\ X\ a \mapsto q\ Y$ with $p, q \in \mathcal{Q}$, $a \in \Sigma \cup \{\varepsilon\}$ and $X, Y \in \Delta \cup \{\varepsilon\}$.

Succinctly, we describe the behavior of our recognizer. Let the PDA be in a configuration $(p, X\alpha, ax)$, where p is the current state, $X\alpha$ is the stack contents with X on the top, ax is the remaining input where the symbol a is the next to be shifted, $x \in \Sigma^*$. The application of a transition $p\ X\ a \mapsto q\ Y$ results in a new configuration $(q, Y\alpha, x)$ where the terminal symbol a has been scanned, X has been popped, and Y has been pushed. If the terminal symbol a is ε in the transition, no input symbol is scanned. If X is ε then no stack symbol is popped from the stack. In a similar manner, if Y is ε then no stack symbol is pushed on the stack.

Of course, in the case of ambiguous recognizing, several such transitions can be applied. As a consequence, the recognizer has to manage a set of parallel stacks in an efficient manner. In this context, the algorithm proceeds by building a collection of *items*, essentially compact representations of the recognizer stacks in order to guarantee a good level of sharing of the computational process.

New items are produced by applying transitions to existing ones, until no new application is possible. The algorithm associates a set of items $S^w_{w_i}$, usually called *itemset*, for each input symbol w_i at the position i in the input string of length n, $w_{1..n}$. We shall also use the notation S^w_i, when the context is clear.

An item is of the form $[p, X, S^w_j, S^w_i]$, where p is a PDA state, X is a stack symbol, S^w_j is the *back pointer* to the itemset associated to the input symbol w_i at which we began to look for that configuration of the automaton, and S^w_i is the current itemset. To simplify notations, we shall call these attributes *I.state, I.symbol, I.back* and *I.current* respectively, for an item I.

In relation to fairness and completeness, an equitable selection order must be established in the treatment of items. We use a technique of *merit ordering*. In essence, we process the items in an itemset in order, performing none or some transitions on each one depending on the form of the item. These operations may add more items to the current itemset and may also put items in the itemset corresponding to the following token to be analyzed from the input string. To ignore redundant items we use a simple subsumption relation based on the equality.

2.2 The shared forest constructor

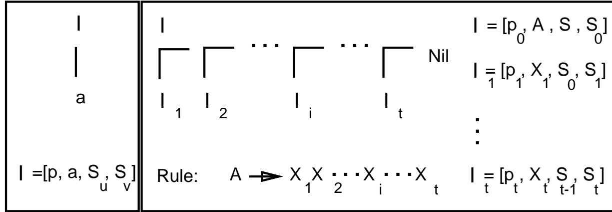
An apparently major difference with most other parsers is that we represent a parse as the chain of the context-free grammar rules used in a leftmost reduction of the parsed sentence, rather than as a parse tree, following the work of Lang in [11, 12]. When the sentence has several distinct parses, the set of all possible parse chains is represented in finite shared form by a context-free grammar that generates that possibly infinite set. However, this difference is only appearance, as proved by Billot and Lang in [2].

In effect, context-free grammars can be represented by AND-OR graphs that in our case are precisely the shared-forest graph. In this graph, AND-nodes correspond to the usual parse-tree nodes, while OR-nodes correspond to ambiguities. Sharing of structures is represented by nodes accessed by more than one other node and it may correspond to sharing of a complete subtree, but also sharing of a part of the descendants of a given node, in fact, a consequence of the binary nature of the transition

protocol precedingly described. This feature allows us to obtain a cubic space complexity for the shared forest in the worst case, whichever it is the form of the grammar.

More formally, the output of this parser will be a sequence of rules of a context-free grammar to be used in a left-to-right reduction of the input sentence, which is equivalent to producing a parse forest. To generate it, items are used as non-terminals of an output grammar $\mathcal{G}_o = (N_o, \Sigma_o, P_o, S_o)$, where N_o is the set of all items, Σ_o the set of input symbols of the original grammar $\mathcal{G}$, and the rules in P_o are constructed together with their left-hand-side item I by the parsing algorithm. We generate a rule for the output grammar each time a reduce or a shift action from the grammar defining the language is applied on the stack as shown in figure 1. In both cases, the left-hand-side of this rule is the new item describing the resulting configuration. In relation to the right-hand-side, it is composed by the token recognized in the case of a shift and by the items popped from the stack in that action, in the case of a reduction. The start symbol S_o is the last item produced by a successful computation.

Figure 1: Node generation in ICE



We shall build a push-down transducer (PDT) from our PDA $\mathcal{A} = (\mathcal{Q}, \Sigma, \Delta, \delta, q_0, Z_0, \mathcal{Q}_f)$ augmenting it with a component Π representing the set of output symbols[2], and considering transitions in δ of the form $p\ X\ a \longmapsto q\ Y\ u$ with $p, q \in \mathcal{Q}$; $a \in \Sigma \cup \{\varepsilon\}$; $X, Y \in \Delta \cup \{\varepsilon\}$, and $u \in \Pi^*$. We shall denote a PDT as a 8-tuple $\mathcal{T}_\mathcal{G} = (\mathcal{Q}, \Sigma, \Delta, \Pi, \delta, q_0, Z_0, \mathcal{Q}_f)$. Given a transition $\tau = \delta(p, X, a) \ni (q, Y, u)$, we translate it to items of the following form:

1. $\tilde{\delta}([p, X, S_j^w, S_i^w], a) \ \ni\ ([q, \varepsilon, S_i^w, S_i^w], \varepsilon)$
2. $\tilde{\delta}([p, X, S_j^w, S_i^w], a) \ \ni\ ([p, Y, S_i^w, S_{i+1}^w], a)$
3. $\tilde{\delta}([p, X, S_j^w, S_i^w], a) \ \ni\ (I_1, I_1 \rightarrow I_2)$
4. $\tilde{\delta}([p, \varepsilon, S_j^w, S_i^w], a) \ \ni\ \tilde{\tau}_d$

[2] in our case, rules of a context-free grammar.

if an only if

1. $Y = X$
2. $Y = a$
3. $Y \in N$
4. $Y = \varepsilon,\ \forall q \in \mathcal{Q}$ such that $\exists\ \delta(q, X, \varepsilon) \ni (p, X, \varepsilon)$

respectively. Where we have considered:

$$\tilde{\tau}_d = \tilde{\delta}_d([q, \varepsilon, S_l^w, S_i^w], a) \ni ([q, \varepsilon, S_l^w, S_j^w], I_3 \rightarrow I_4 I_5)$$

$$\begin{array}{ll} I_1 = [p, Y, S_i^w, S_i^w], & I_2 = [p, X, S_j^w, S_i^w] \\ I_3 = [q, \varepsilon, S_l^w, S_i^w], & I_4 = [q, X, S_l^w, S_j^w] \\ I_5 = [p, \varepsilon, S_j^w, S_i^w] & \end{array}$$

and

$$\left\{ \begin{array}{l} \tilde{\delta} : \mathrm{It} \times \Sigma \cup \{\varepsilon\} \longrightarrow \{\mathrm{It} \cup \tilde{\delta}_d\} \times \Pi^* \\ \tilde{\delta}_d : \mathrm{It} \times \Sigma \cup \{\varepsilon\} \longrightarrow \mathrm{It} \end{array} \right.$$

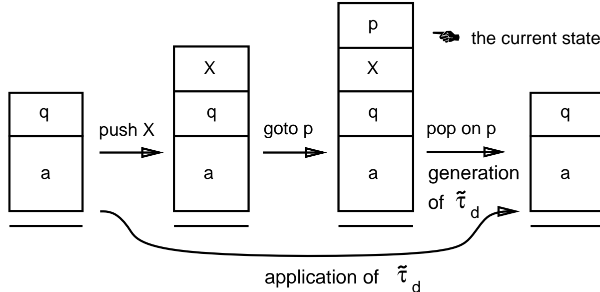
with *It* the set of all items developed in the parsing process and $\tilde{\delta}_d$ is called the set of *dynamic transitions*. Succinctly, we can describe the preceding cases as follows:

1. Corresponds to a goto action from the state p to state q under transition X in the LALR(1) automaton.
2. Corresponds to a push of terminal a from state p. The new item belongs to the next itemset S_{i+1}^w.
3. Corresponds to a push of non-terminal Y from state p.
4. Corresponds to a pop action from state p, where q is an ancestor of state p under transition X in the LALR(1) automaton. In this case, we do not generate a new item, but a *dynamic transition* $\tilde{\tau}_d$ to treat the absence of information about the rest of the stack. This transition is applicable not only to the configuration resulting of the first one, but also on those to be generated and sharing the same syntactic structure, as shown in figure 2.

   The concept of *dynamic transition* is due to Villemonte de la Clergerie [16] who applied it in the context of the generation of efficient and complete definite clause programs compilers.

At this point, items are not only elements of the computation process, but also non-terminals of the output grammar. That allows us to identify items with nodes in the resulting parse forest. So, given an item I we shall denote by $I.forest$ the value of a pointer to the set of rules in the output context-free grammar whose left-hand-side is I.

Figure 2: Dynamic transitions in ICE



2.3 The choice of the deterministic kernel

The choice of the deterministic parsing scheme to be the kernel of ICE depends *in fine* on: the grammar used, the corpus of sentences to be analyzed, and a balance between computational and sharing efficiency, and parser size. As a consequence, the problem is a practical one and the best basis to decide, experimental. In this sense, the results achieved by Billot and Lang in [2], and earlier by Bouckaert, Pirotte and Snelling in [3] show, on an experimental basis, that techniques close to straightforward bottom-up methods, such as LALR(1), are the most appropriate.

From an intuitive point of view, that conclusion was expected. Effectively, those algorithms combine certain characteristics making them good candidates for a non-deterministic approach:

- A very large deterministic domain. In this direction, it should be good to remember that user often write grammars sufficiently close to deterministic ones. This phenomenon, called *local determinism*, has already been observed by Lang in [11]. Therefore, the use of the above parallel parsing techniques allows an important economy in time and space.
- An efficient treatment of the sharing problem of computations and structures:
  - The consideration of some determinization techniques such as the classic LR(k) ones, in principle the best, has as a consequence the distortion of the initial grammar due to the application of a predictive technique in the generation of the parser in relation to the lookahead facility. This leads to the well known *state splitting phenomenon*. Sharing of syntactic structures can then be affected.
  - In the case of LALR(1) parsers, not only the necessary tests to implement lookahead facility are easier to perform, but the state splitting phenomenon remains reasonable. That allows us to assure a good sharing of computation and parsing structures.

3 Incremental Parsing

The goal of the incremental context-free parser is to recover stable parts of a shared forest between consecutive parsing steps, at no cost in space and time. In particular, we have to guarantee the same level of sharing in the resulting parse forest, as in standard parsing.

In the context of the recovery process, we shall define *stable* items between the initial parsing of $w_{1..n}$ and the parsing of the modified input string $x_{1..n+k}$, $k \in [-n, \infty)$, as those items that represent a stable configuration of the PDT that would be reconstructed if we had redone an entire parse of the modified input string up to that point. More formally, an item $I_i^w = [p, X, S_j^w, S_i^w]$ is stable if and only if there exists an item $I_{w_i}^x = [p, X, S_j^w, S_{w_i}^x]$. We shall denote it as $I_i^w \equiv I_{w_i}^x$. In this case, items I_i^w and $I_{w_i}^x$ would represent equivalent trees of their shared forest. Henceforth, we shall also denote $\sqsubset$ the relation of inclusion induced by $\equiv$ in the sets of items.

From a practical point of view, we shall only consider the case of recovery on the basis of complete itemsets[3]. This is equivalent to the recovery of all the OR-descendants of a node in a shared forest. Eventually, there will be no recovery of trivial nodes in an itemset. Even if this does not guarantee that all superfluous computations will be avoided, it allows to notably reduce the comparison between stack configurations corresponding to the original and the modified input string. At this point, our experience has shown that the incremental treatment is not interesting from a practical point of view when only a part of an itemset is stable, as it is usually the case when the input grammar has a lot of ambiguities generating crossed forests.

3.1 An informal description of the problem

To begin with, we consider the case of a single modification. So, we assume $x_{1..n+k}$, $k \in [-n, \infty)$

[3] that is, shift actions are checkpoints in the parsing process.

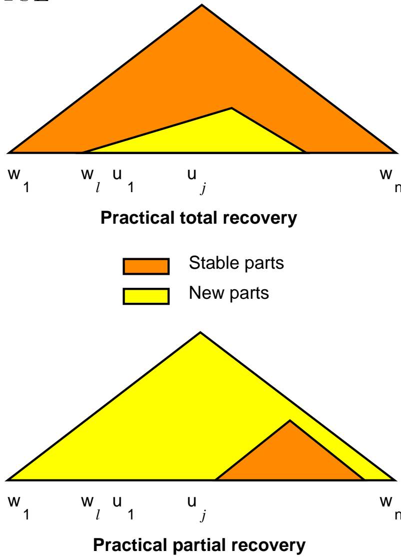
is a modified input string from $w_{1..n}$, where x is of the form: $x_{1..n+k} = w_1 \cdots w_\ell \; u_1 \cdots u_j \; w_{\ell+\hbar+1} \cdots w_n$ with

$$\left\{ \begin{array}{l} u_{1..j} \in \Sigma^* \\ \hbar = \mid u \mid -k \end{array} \right. \quad \text{and} \quad \mid u \mid = \left\{ \begin{array}{l} k, \text{ if } k \geq 0 \\ 0, \text{ otherwise} \end{array} \right.$$

where we call S^w_ℓ, a *point of modification relative to w and x*. That is, we are assuming that our modification is the substitution[4] of $w_{\ell+1..\ell+\hbar}$ by $u_{1..j}$.

By dynamic programming construction, items in $S^w_{1..\ell-1}$ correspond to the stable part of the parsing process[5], while items in $S^w_\ell \, S^u_1 \cdots S^u_j$ correspond to the part of the parsing process which is new. Finally, items in $S^w_{\ell+\hbar+1..n}$ correspond to the part of the parsing process that will eventually be recovered.

Figure 3: Practical incremental recovery of shared forests in ICE



3.2 Practical incremental recovery

As shown in figure 3, we shall now consider two different cases depending on the nature of the recovery of complete itemsets.

- *Total recovery.* The idea is to detect when the parsing process becomes independent of the modification.
- *Partial recovery.* In this case, recovery applies to all trees of the shared forest corresponding to a part of the remaining input.

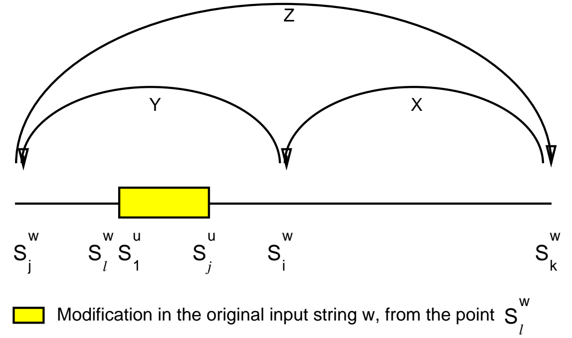
[4] for deletion we take $u = \varepsilon$ and $k < 0$, for insertion we assume $\mid u \mid = k$ with $k > 0$ and for substitution we consider $k = 0$.

[5] S^w_ℓ can be recovered when the lookahead has not changed.

3.2.1 Total recovery

We now have to establish a condition under which total recovery is possible from a given input position i. To do that, it is sufficient to find a condition capable to assure that all future pop transitions do not depend on the modification, as shown in figure 4 below, with a pop transition $XY \mapsto Z$ that does not depend on the modification $S^u_1 \cdots S^u_j$ at the point S^w_ℓ. This is because pop transitions are the only ones depending on the past of the parsing process.

Figure 4: A pop transition $XY \mapsto Z$ totally recovering a modification



Therefore a sufficient condition for S^w_i to allow total recovery from that itemset is that, for each item $I \in S^w_{i-1}$ such that:

1. I is the argument in a pop transition from a valid suffix of $w_{1..i}$.
2. All pop transition taking I as argument, implies a return to an itemset S^w_t, $t < i - 1$.

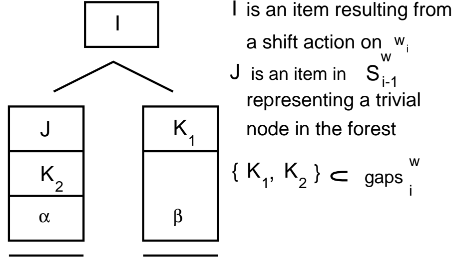
I is stable, and $t < \ell$ the point of modification relative to w and x.

More formally, we call items verifying conditions 1 and 2 $gaps^w_{w_i}$ or $gaps^w_i$ when the context is clear. This concept is similar to the *essential parse sequences* proposed by Celentano in [4], which favored these non trivial stack configurations having a shift as possible action to perform, for LR parsers.

Observe that the concept of $gaps^w_i$ does not include those items in the same itemset S^w_i representing trivial nodes from which the only possible actions to perform are empty reductions followed by a shift. Even if this does not guarantee that all superfluous computations will be avoided, it allows to notably reduce the comparison tests between stack configurations corresponding to the original and the modified input string.

In practice, the sets $gaps_i^w$ are computed by considering the basis of S_i^w, that is, those items that were introduced into S_i^w by a push transition corresponding to a shift action of the PDT. From each one of those items, we put into $gaps_i^w$ the first descendant in the stack forest in S_{i-1}^w representing a non-trivial node, as is shown in figure 5.

Figure 5: How ICE computes gaps from the stacks



Once we have computed the gaps, we can give a sufficient condition for total recovery: Given $x_{1..n+k}$, $k \in [-n, \infty)$ a modified input string from $w_{1..n}$ and S_ℓ^w a point of modification relative to w and x, verifying

$\exists\, l \in [\ell + \hbar + 1, n)$, such that

1. $gaps_l^w \sqsubset gaps_{w_l}^x$ (*resp.* $gaps_l^w \equiv gaps_{w_l}^x$)

2. $\forall I \in gaps_{w_l}^x$, $I.back \leq \ell$

then, from the point of view of the recognizer $S_t^w \sqsubseteq S_{w_t}^x$, $\forall t \in [l, n)$ (resp. $S_t^w \equiv S_{w_t}^x$, $\forall t \in [l, n)$).

Effectively, if the set of possible pop transitions is the same in two parsing process and the not yet parsed substring is also the same, then the future is identical.

This result can be extended to the case of several simultaneous modifications of the input string, in a single recovery process. To do that, we define the notion of a *totally recovered point of modification relative to w and x*, as a point in the recovery process such that the corresponding modification has been totally recovered. Then the condition becomes: Given $x_{1..n+k}$, $k \in [-n, \infty)$ a modified input string from $w_{1..n}$ and $S_{\ell_{1..m}}^w$ m contiguous points of modification relative to w and x, verifying

$\exists\, i \in [1, m]$, $l \in [\ell_i + \hbar_i + 1, \ell_{i+1})$, such that

1. $gaps_l^w \sqsubset gaps_{w_l}^x$ (*resp.* $gaps_l^w \equiv gaps_{w_l}^x$)

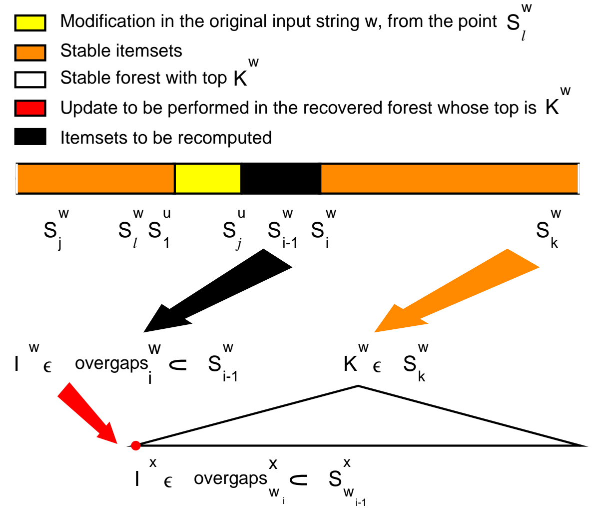
2. $\forall I \in gaps_{w_l}^x$, $I.back \leq \ell_i$

3. $\forall j \in [1, i-1]$, $S_{\ell_j}^w$ has been totally recovered

then, from the point of view of the recognizer $S_t^w \sqsubseteq S_{w_t}^x$, $\forall t \in [l, \ell_{i+1})$ (resp. $S_t^w \equiv S_{w_t}^x$, $\forall t \in [l, \ell_{i+1})$).

In the case of the parser, we must find in addition the scope of the modifications in the original forest, a simple task given that the nodes affected by changes in its structure are those common with the new parse forest having at least a changed descendant in relation to the new development, that is, those common items capable to be accessed in the continuation of the parsing process in the case no incremental treatment was applied. To update one of these nodes it will be sufficient to find its stable descendants in the parse forest which have been effectively recomputed and to replace them by the original corresponding structure in the recovered parse forest.

Figure 6: How ICE extends total recovery to the parser



Taking into account that we only recover complete itemsets, this phenomenon is limited to those items representing stable trivial nodes for which their ancestors in the parse forest are not computed in the same itemset S_l^w, where they are included. We call these items $overgaps_{w_l}^w$ or more simply $overgaps_l^w$ when the context is clear[6].

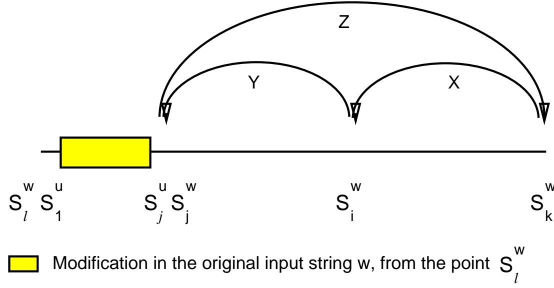
At this point, to extent total recovery to the parser, we must perform for all $I^w \in overgaps_l^w$ the assignments $I^w.forest := I^x.forest$, where I^w and I^x represent in each case the same node in the parse forest, that is $I^w \equiv I^x$. This process is illustrated in figure 6.

[6] the reason for which we have chosen the name *overgaps* is that these items are built from the gaps of the corresponding itemset.

3.2.2 Partial recovery

We start by considering the case of a single modification, and we now have to establish a condition under which partial recovery is possible from itemset S^w_i to itemset S^w_j. To do that, it is sufficient to find a condition under which pop transitions would not depend on the modification for itemset $S^x_{w_i}$ to itemset $S^x_{w_j}$, as it is shown in figure 7. We assume no additional modification in the substring $w_{i..j}$.

Figure 7: A pop transition $XY \mapsto Z$ independent of the modification



Therefore a sufficient condition for S^w_i to allow partial recovery from that itemset to some itemset S^w_j is that for each item $I \in S^w_{i-1}$ such that:

1. I is the argument in a pop transition from a valid suffix of $w_{1..i}$.
2. All pop transition taking I as argument, imply a return to an itemset S^w_t, $t < i-1$.

there does not exist a pop transition in $S^w_{i..j}$ taking I as argument[7].

It is important to remark that the above condition does not imply the stability of the item I. This is because it does not take into account the past of the parsing process represented by the back pointer, as was the case in total recovery. More formally, an item $I^w_i = [p, X, S^w_j, S^w_i]$ is *weakly stable* if and only if there exists an item $I^x_{w_i} = [p, X, S^x_{w_j}, S^x_{w_i}]$. We shall denote it as $I^w_i \cong I^x_{w_i}$. In this case, items I^w_i and $I^x_{w_i}$ would not necessarily represent equivalent trees of their shared forest. We shall denote $\prec$ the relation of inclusion induced by $\cong$ in the sets of items.

Taking into account the definition of $gaps^w_i$, we conclude that there is no pop transition in $S^w_{i..k}$ taking $I \in gaps^w_i$ as argument.

[7]in practice, the realization of this test just necessitates to store the minimun value of j when applying pop transitions after S^w_i.

More formally, we can assure that given $x_{1..n+k}$, $k \in [-n, \infty)$ a modified input string from $w_{1..n}$ and $S^w_{\ell_{1..m}}$ m contiguous points of modification relative to w and x, verifying that

$$\exists\, i \in [1, m],\ l, j \in [\ell_i + \hbar_i + 1, \ell_{i+1}),\ \text{such that}$$

1. $gaps^w_l \preceq gaps^x_{w_l}$ (*resp.* $gaps^w_l \cong gaps^x_{w_l}$)
2. S^w_j is the first itemset applying a pop on $gaps^w_l$

then, from the point of view of the recognizer, $S^w_t \sqsubseteq S^x_{w_t}$, $\forall t \in [l, j-1]$ (resp. $S^w_t \equiv S^x_{w_t}$, $\forall t \in [l, j-1]$). To extend this result to the parse forest, it is sufficient to perform for all $I^w \in overgaps^w_l$ the assignments $I^w.forest := I^x.forest$, where $I^w \cong I^x$.

3.3 Complexity bounds

We assume $x_{1..n+k}$, $k \in [-n, \infty)$ is a modified input string from $w_{1..n}$, and $S^w_{\ell_{1..m}}$ m contiguous points of modification relative to w and x. In this context, the application of our incremental test takes a time $\mathcal{O}(n^3)$ and a space $\mathcal{O}(n^2)$, in the worst case. The reasons for this are:

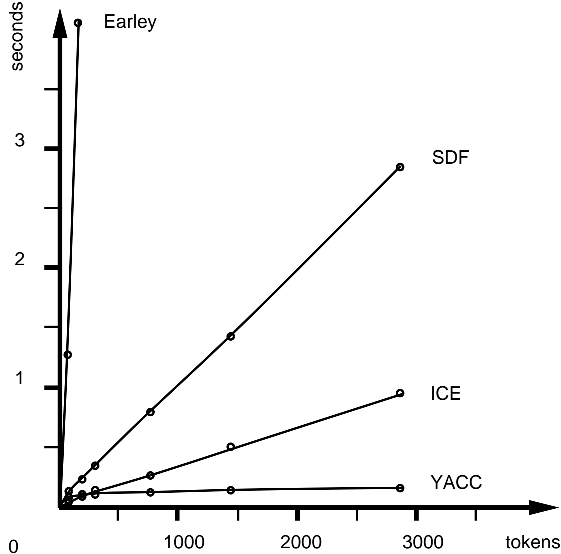
1. The number of items in S^w_l is $\mathcal{O}(l)$, as it is proved by Lang in [11]. Therefore, the number of items in $gaps^w_l$ is also $\mathcal{O}(l)$, in the worst case. The result is the same for the number of items in $overgaps^w_l$.
2. As a consequence, time complexity for the test $gaps^w_l \sqsubset gaps^x_{w_l}$ is $\mathcal{O}(l^2)$. On the other hand, we need a space $\mathcal{O}(l)$ to store $gaps^w_l$.
3. From that, we have that in the worst case, the consideration of the incremental mode takes a time $\mathcal{O}(\Sigma_{l=\ell_i+\hbar_i+1}^{\ell_i-1} l^2) \leq \mathcal{O}(\Sigma_{l=0}^{n} l^2) = \mathcal{O}(n^3)$ and a space $\mathcal{O}(\Sigma_{l=\ell_i+\hbar_i+1}^{\ell_i-1} l) \leq \mathcal{O}(\Sigma_{l=0}^{n} l) = \mathcal{O}(n^2)$

We can characterize the class of grammars which the algorithm do in time $\mathcal{O}(n)$. For some grammars, called *bounded item grammars*, the number of items in a given itemset cannot grow indefinitely. In this case, the number of items is $\mathcal{O}(k)$ with k constant, whichever it is the considered itemset. As a consequence, the test for incrementality takes a time $\mathcal{O}(\Sigma_{l=\ell_i+\hbar_i+1}^{\ell_i-1} k^2) \leq \mathcal{O}(\Sigma_{l=0}^{n} k^2) = \mathcal{O}(n)$, and a space $\mathcal{O}(\Sigma_{l=\ell_i+\hbar_i+1}^{\ell_i-1} k) \leq \mathcal{O}(\Sigma_{l=0}^{n} k) = \mathcal{O}(n)$. At this point, we must remark that the complexity bounds for the standard parser [11] are exactly the same.

4 Experimental Results

We use the syntax of complete Pascal to show the efficiency of ICE, comparing it with YACC [10], the standard deterministic parser generator of UNIX and SDF [8], one of the most efficient non-deterministic parser generator which is based on Tomita's algorithm [14]. We have also compared ICE with an implementation of the classic Earley's algorithm proposed by Vilares in [15]. Results are

Figure 8: Performances on non-ambiguous Pascal



shown in figures 8, 9 for the standard parser, and figures 10, 11 for the incremental parser, where:

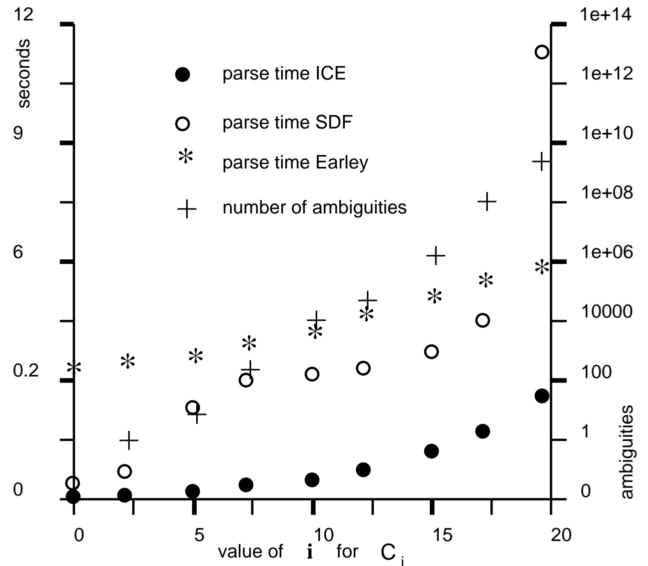
1. All tests have been performed using the same input programs for each one of the parsers and the time needed to "print" parse trees was not measured.

2. We don't include garbage collection time, and in order to avoid errors of UNIX standard clocks we have considered the middle time of twenty different runs for each example.

3. We include the time corresponding to lexical analysis, since it is not possible in SDF to differentiate that from parsing. In all other cases, LEX [13] has been used to generate the lexical analyzer.

4. All the measurements have been performed on a *Sun SPARCstation 2*, weakly loaded. The ambiguous version of Pascal includes the *dangling else* and the ambiguity for the arithmetic expressions. To illustrate non-determinism, we measure the time needed to parse programs of the form:

Figure 9: Performances on ambiguous Pascal



program P (**input**, **output**);
  var a, b : **integer**;
begin
  $a := b\{+b\}^i$
end.

where i is the number of +'s. As the grammar contains a rule "Expression ::= Expression + Expression", these programs have a number of ambiguous parses which grows exponentially with i. This number is:

$$C_i = \begin{cases} 1 & \textbf{if} \quad i = 0, 1 \\ \begin{pmatrix} 2i \\ i \end{pmatrix} \dfrac{1}{\mathrm{i+1}} & \textbf{if} \quad i > 1 \end{cases}$$

5. SDF works on an extended LR(0) machine, and ICE uses an extended LALR(1) one. So, the ambiguous Pascal grammar used for SDF has 357 shift/reduce conflict, while that used for ICE contains only 305.

6. Given that in both, SDF and ICE, mapping between concrete and abstract syntax is fixed, we have generated in the case of YACC, a simple recognizer.

Figure 10: Incrementality on ambiguous Pascal

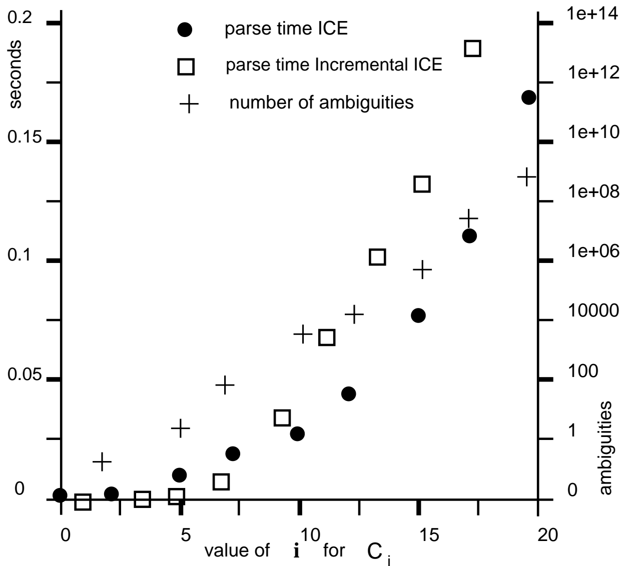
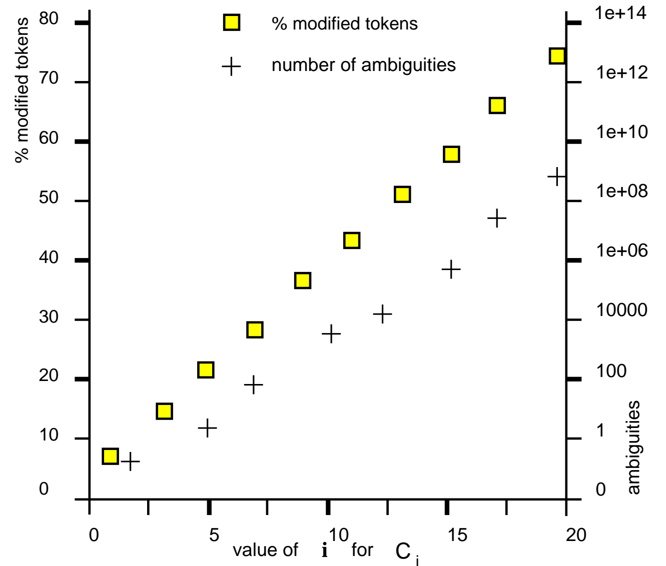


Figure 11: Incrementality on ambiguous Pascal



7. To illustrate incrementality in the general context-free case, we analize programs of the form:

   program P (**input**, **output**);
   var a, b : **integer**;
   begin
   $a := (b + b) + b\{+(b + b) + b\}^l$
   end.

   to obtain

   program P (**input**, **output**);
   var a, b : **integer**;
   begin
   $a := b + b\{+b + b\}^l$
   end.

   by substitution of expressions $(b + b)$ by b, where $l \geq 0$. Results are given in figure 11, in relation to the number of tokens modified in the original program, and in figure 10 in relation to the time needed to reparse them.

8. ICE and SDF have been written in Le_Lisp [5], while YACC is written in C.

5 Summary and Conclusions

In conclusion, we have developed a practical and efficient implementation of the class of context-free parsers introduced by Lang in [11]. The efficiency of these parsers is due to a compromise between the sharing of computations and the sharing of resulting forests. The best results has been obtained by using an LALR(1) automaton as the basis for the parser. These non-deterministic parsers were then extended to allow an incremental treatment, and the validity of this approach has been proved on examples where the number of ambiguities stays reasonably small, as is the case in practice.

Furthermore, the parser generator we have developed is compatible with YACC [10], which permits to freely use all the input that were developed for this generator.

Acknowledgements

The authors gratefully acknowledge Bernard Lang for helpful comments on the work described in this paper.

References

[1] Agrawal, R. and Detro, K. D.
An Efficient Incremental LR Parser for Grammars with Epsilon Productions.
Acta Informatica, vol. 19, pp. 369-376, 1983.

[2] Billot, S. and Lang, B.
The Structure of Shared Forest in Ambiguous Parsing.

Research Report n°1038, INRIA Rocquencourt, France, 1989.

[3] Bouckaert, M.; Pirotte, A. and Snelling, M.
Efficient Parsing Algorithms for General Context-Free Grammars.
Information Sciences, vol. 8, pp. 1-26, 1975.

[4] Celentano, A.
Incremental Parsers.
Acta Informatica, vol. 10, pp. 307-321, 1978.

[5] Chailloux, J.
Le_Lisp. Version 15.23. Reference Manual.
INRIA. Rocquencourt, France, 1990.

[6] Earley, J.
An Efficient Context-Free Parsing Algorithm.
Communications of the ACM, vol. 13, n°2, pp. 94-102, 1970.

[7] Ghezzi, C. and Mandrioli, D.
Augmenting Parsers to Support Incrementality.
Journal of the ACM, vol. 27, n°3, pp. 564-579, 1980.

[8] Heering, J. and Klint, P.
A Syntax Definition Formalism.
ESPRIT '86:Results and Achievements, North-Holland Publishing Company, New York, U.S.A, pp. 619-630, 1987.

[9] Jalili, F. and Gallier, J. H.
Building Friendly Parsers.
ACM Ninth Symposium on the Principles of Programming Languages, pp. 196-206, 1982.

[10] Johnson, S. C.
YACC. Yet Another Compiler Compiler.
Computer Sciences Technical Report n°32, AT&T Bell Laboratories, Murray Hill, New Jersey, U.S.A., 1975.

[11] Lang, B.
Deterministic Techniques for Efficient Non-deterministic Parsers.
Research Report n°72, INRIA Rocquencourt, France, 1974.

[12] Lang, B.
Towards a Uniform Formal Framework for Parsing.
Current Issues in Parsing Technology, M. Tomita ed., Kluwer Academic Publishers, pp. 153-171, 1991.

[13] Lesk, M. E. and Schmidt, E.
Lex: a Lexical Analyzer Generator.
UNIX Programmer's Manual 2. Murray Hill, New Jersey, U.S.A., AT&T Bell Laboratories, 1975.

[14] Tomita, M.
An Efficient Augmented-Context-Free Parsing Algorithm.
Computational Linguistics, vol. 13, n°1, 2, pp. 31-36, 1987.

[15] Vilares Ferro, M.
Efficient Incremental Parsing for Context-Free Languages.
Doctoral thesis, University of Nice, France, 1992.

[16] Villemonte de la Clergerie, E.
DyALog. Une Implementation des Clauses de Horn en Programmation Dynamique.
Actes du 9th Séminaire de Programmation en Logique, CNET, Lannion, France, pp. 207-228, 1990.

A An example for the standard parser

The first example, without incremental treatment, is intended to familiarize the reader with our constructions. To illustrate the discussion, we shall assume the grammar IE given by the productions:

(0)	$\Phi \rightarrow S \dashv$	(1)	$S \rightarrow S\ a\ S$	(2)	$S \rightarrow b$
(3)	$S \rightarrow c\ d\ E$	(4)	$S \rightarrow a$	(5)	$S \rightarrow \varepsilon$
(6)	$E \rightarrow e$				

whose LR(0) finite state machine is represented in figure 12.

We shall consider as input strings $w = baacde \dashv$ and $x = bacde \dashv$, for which itemsets corresponding to their parsing process are shown in table 1. We only include in those tables items corresponding to the recognition of a syntactic category in the original grammar IE.

Parse forests corresponding to both input strings w and x can be respectively seen in figure 13 and figure 14. In order to facilitate understanding, we have included in each node corresponding to the reduction of a non-terminal in the original input grammar IE, the number of the considered rule. Patterns will be used later to illustrate incremental treatment.

Figure 12: The LR(0) machine for the IE grammar

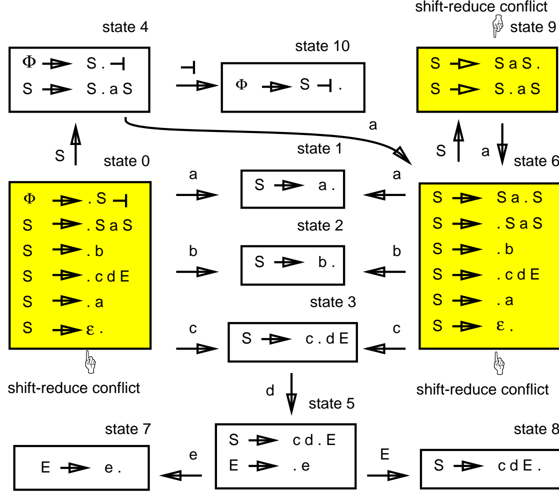


Table 1: Itemsets for $w = baacde \dashv$ and $x = bacde \dashv$

S_0^w	$\overset{\circ}{\$}$	S_0^x	$\overset{\circ}{\$}$
S_1^w $(w_1 = b)$	$I_0^1 \equiv [0, b, S_0^w, S_1^w]$ $I_1^1 \equiv [0, S, S_0^w, S_1^w]$	S_1^x (x_1)	$I_0^1 \equiv [0, S, S_0^x, S_1^x]$ $I_1^1 \equiv [0, S, S_0^x, S_1^x]$
S_2^w $(w_2 = a)$	$I_0^2 \equiv [4, a, S_1^w, S_2^w]$ $I_1^2 \equiv [6, S, S_2^w, S_2^w]$ $I_2^2 \equiv [0, S, S_0^w, S_2^w]$		
S_3^w $(w_3 = a)$	$I_0^3 \equiv [6, a, S_2^w, S_3^w]$ $I_1^3 \equiv [9, a, S_2^w, S_3^w]$ $I_2^3 \equiv [4, a, S_2^w, S_3^w]$ $I_3^3 \equiv [6, S, S_2^w, S_3^w]$ $I_4^3 \equiv [0, S, S_0^w, S_3^w]$	S_2^x (x_2)	$I_0^2 \equiv [4, a, S_1^x, S_2^x]$
S_4^w $(w_4 = c)$	$I_0^4 \equiv [6, c, S_3^w, S_4^w]$	S_3^x (x_3)	$I_0^3 \equiv [6, c, S_2^x, S_3^x]$
S_5^w $(w_5 = d)$	$I_0^5 \equiv [3, d, S_4^w, S_5^w]$	S_4^x (x_4)	$I_0^4 \equiv [3, d, S_3^x, S_4^x]$
S_6^w $(w_6 = e)$	$I_0^6 \equiv [5, e, S_5^w, S_6^w]$ $I_1^6 \equiv [5, E, S_5^w, S_6^w]$ $I_2^6 \equiv [6, S, S_3^w, S_6^w]$ $I_3^6 \equiv [0, S, S_0^w, S_6^w]$ $I_4^6 \equiv [6, S, S_2^w, S_6^w]$	S_5^x (x_5)	$I_0^5 \equiv [5, e, S_4^x, S_5^x]$ $I_1^5 \equiv [5, E, S_4^x, S_5^x]$ $I_2^5 \equiv [6, S, S_2^x, S_5^x]$ $I_3^5 \equiv [0, S, S_0^x, S_5^x]$
S_7^w $(w_7 = \dashv)$	$I_0^7 \equiv [4, \dashv, S_6^w, S_7^w]$	S_6^x (x_6)	$I_0^6 \equiv [4, \dashv, S_5^x, S_6^x]$

B An example for the incremental parser

Gaps for both input strings $w = baacde \dashv$ and $x = bacde \dashv$ in relation to the grammar IE are shown in table 2.

Table 2: Gaps for $w = baacde \dashv$ and $x = bacde \dashv$

$gaps_1^w = \emptyset$	$gaps_1^x = \emptyset$
$gaps_2^w = \{I_1^1\}$	
$gaps_3^w = \{I_0^2,\ I_2^2\}$	$gaps_2^x = \{I_1^1\}$
$gaps_4^w = \{I_1^3,\ I_2^3\}$	$gaps_3^x = \{I_0^2\}$
$gaps_5^w = \{I_0^4\}$	$gaps_4^x = \{I_0^3\}$
$gaps_6^w = \{I_0^5\}$	$gaps_5^x = \{I_0^4\}$
$gaps_7^w = \{I_3^6\}$	$gaps_6^x = \{I_3^5\}$

B.1 Total recovery

To illustrate total recovery, we shall first consider the original input string $x = bacde \dashv$ and the modified one $w = baacde \dashv$ taking IE as input grammar. That is, we shall here assume that:

- The only point of modification relative to x and w is S_1^x.
- The modification consists in the insertion of the string $u_{1..j} = u_{1..1} = a$, immediately after the first b character.
- $\hbar = \mid u \mid - k = 0$.

Given that the lookahead of the point of modification has not changed, we have that:

$$\left\{ \begin{array}{l} S_0^x \equiv S_0^w \\ S_1^x \equiv S_1^w \end{array} \right.$$

which corresponds to node I_1^1 in figure 13 and figure 14.

From $gaps_2^x$ and $gaps_{x_2}^w = gaps_3^w$, we obtain that $gaps_2^x \sqsubset gaps_3^w$. We conclude that $S_t^x \sqsubseteq S_{x_t}^w$, $\forall t \in [2, 6]$. From the viewpoint of the parse, we recover all the AND-OR graph represented in figure 14. To facilitate the task, the recovered graph has been painted in figure 13, using the same pattern.

B.2 Partial recovery

We now consider:

- $x = bacde \dashv$ is a modified input string from $w = baacde \dashv$, whose only point of modification is S_1^x.

- The modification consists in the deletion of the first a character.

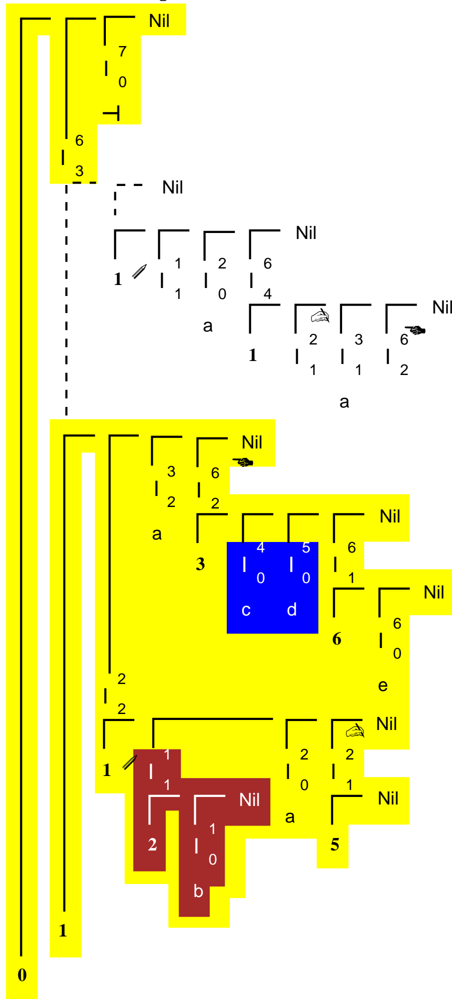
- $\hbar = \mid u \mid - k = 1$.

As in the total recovery case, the lookahead of the point of modification has not changed, and we have that:

$$\begin{cases} S_0^x \equiv S_0^w \\ S_1^x \equiv S_1^w \end{cases}$$

which corresponds to node I_1^1 in figure 13 and figure 14.

Figure 13: AND-OR graph for the input $w = baacde \dashv$



Given that $\ell + \hbar + 1 = 3$, we shall firstly compare the gaps from S_3^w and $S_{w_3}^x = S_2^x$, to obtain that $gaps_3^w \not\simeq gaps_2^x$, reason for which we shall continue to successively compare $gaps_4^w$ with $gaps_3^x$ and $gaps_5^w$ with $gaps_4^x$. In this case, we shall have $gaps_5^w \cong gaps_4^x$, and therefore we can assure that $S_5^w \equiv S_4^x$, given that the first itemset applying a pop transition on $gaps_5^w$ is S_6^w.

Intuitively, we have recovered all the development corresponding to nodes labeled by I_0^4 and I_0^5 in figure 13. To facilitate its location, it has been painted using the same pattern in figure 14.

Figure 14: AND-OR graph for the input $x = bacde \dashv$