# Redundancy-free Island Parsing of Word Graphs

**natural language, speech processing, dialog processing**

## Abstract

Island parsing is a bidirectional parsing strategy mostly used in speech analysis, as well as in applications where robustness is highly relevant and/or processing resources are limited. Although there exists an efficient redundancy free island parsing algorithm for string input, it has not yet been applied to word graph input, an application which is central for speech analysis systems. This paper describes how the established algorithm can be generalized from string input to word graphs, increasing its flexibility by integrating the selection of island seeds into the search process inherent to parsing.

## 1 Introduction

Island parsing is a parsing strategy for context free grammars, mostly used in speech applications ([Ageno, 2003], [Gallwitz *et al.*, 1998], [Thanopoulos *et al.*, 1997], [Mecklenburg *et al.*, 1995], [Brietzmann, 1992]). It is a *bidirectional* strategy, in that incomplete parse items, which encode partially filled right hand sides of a context free rule, may extend in both directions. Furthermore, parsing starts at some highly ranked input items called *seeds* and tries to explore the "islands of certainty" first.

Since island parsing starts building all possible derivations from every seed in both directions, provisions must be taken so as to avoid multiple computation of the same subderivation, which would otherwise lead to spurious ambiguities and reduced efficiency. To my knowledge, there is only one description of an efficient fully redundancy free algorithm for island parsing, namely [Satta and Stock, 1994], which describes and compares different bidirectional parsing approaches for context-free grammars. There are two new aspects which are addressed in this paper that make it more feasible for speech applications.

Firstly, the original algorithm deals only with string input. Because many speech applications require the direct analysis of word graphs, it is desirable to extend the method to word graph input. Word graphs are acyclic directed graphs of input items with exactly one source node and one sink node (a node with in-degree resp. out-degree zero). They encode ambiguous input using possibly overlapping sub-paths, which lead to more complicated input configurations.

A modified algorithm must be able to deal with these configurations without losing efficiency, since speech applications are typically time critical and often have limited space resources. Therefore, care has been taken to preserve the efficiency and the redundancy avoidance of the original algorithm.

Secondly, the modified version integrates the selection of seed items into the search inherent to the parsing process. Since a lower number of seeds may result in faster parsing, it is advantageous to be able to base selection not only on information available before parsing starts, but also on information that is created in the parsing process, namely, which items have already been constructed and what their (combined) quality is.

In the modified algorithm, the selection of a seed becomes just one of the actions the parser can take, like the combination or creation of other parse items. This provides full flexibility in the design of the search strategy, which in resource-limited applications can have a big impact on the quality of the, possibly partial, results. Picking all seeds in advance is then just one of the possible options.

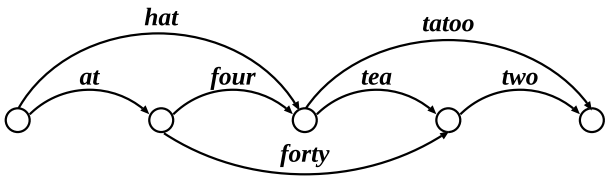

Figure 1: Example word graph

Furthermore, dynamic selection of seeds is performed in such a way as to guarantee that all sub-paths of the word graph will be be properly explored to arrive at a complete solution, which might not be the case if seeds were picked disadvantageously. If, for example, the seeds in figure 1 were the items labeled *hat* and *tea*, it might happen that the sub-path containing *forty* would not be considered.

## 2 The Original Algorithm

Because [Satta and Stock, 1994] aim at describing bidirectional context-free parsing in more generality, the formulation of the island parsing algorithm itself is somewhat complicated and its implementation is not obvious at first glance.

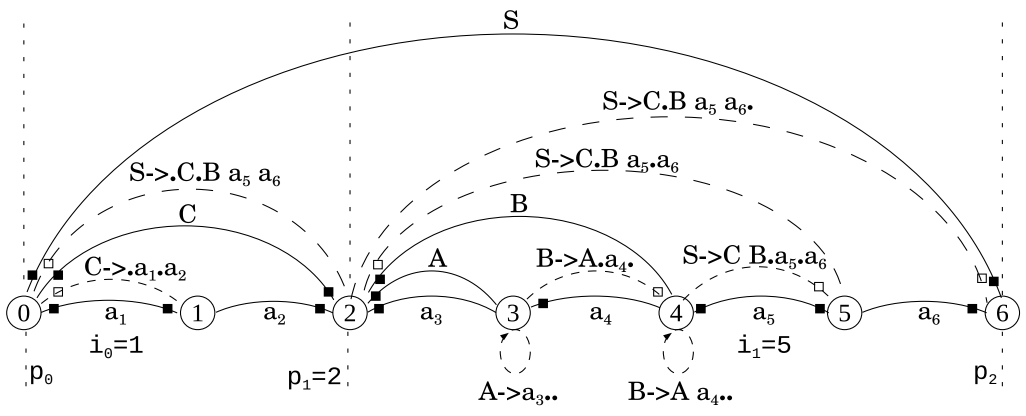

Figure 2: A decorated parse chart generated by the original algorithm

To facilitate the description of the modifications, the original algorithm is presented first, albeit in an alternative, but equivalent form. Similar notation is used as far as possible to stress the connection between the two formulations. In many respects, it is an ordinary chart parsing algorithm, that is only complicated by the bookkeeping necessary to avoid redundant computations. After introducing the notation, the algorithm is presented as a set of pseudo-code procedures.

The algorithm uses a context free grammar $G = (\Sigma, N, P, S)$, where $\Sigma$ and $N$ are finite sets of *terminal* and *nonterminal* symbols, respectively, $P$ is the set of *rules* $r := D_r \rightarrow Z_{r,1} \cdots Z_{r,\pi_r}$, where $D_r \in N$ and each $Z_{r,i} \in \Sigma \cup N$. $\pi_r$ is the number of symbols on the right hand side of rule $r$. $S \in N$ is the start symbol of the grammar. The grammar must not contain empty rules, i.e, rules of the form $A \rightarrow \epsilon$. The input is a string of $L$ terminal symbols $a_1, \ldots, a_L$. The algorithm uses a chart of size $L+1$: a two-dimensional array $t_{i,j}$, $i, j \in \{0, \ldots, L\}$, which contains two kinds of items: complete and incomplete items.

A *complete* chart item is a triple $(NT, i, j)$ with $NT \in \Sigma \cup N$ being the terminal or nonterminal category, $i$ and $j$ the index of the start and end node of the item, respectively.

For *incomplete* items, we introduce symbols $I_r^{k,l}$ that represent dotted items (partial derivations) of a rule $r \in P$:

$$I_r^{k,l} \equiv D_r \rightarrow Z_{r,1} \cdots Z_{r,k} \bullet Z_{r,k+1} \cdots Z_{r,l} \bullet Z_{r,l+1} \cdots Z_{r,\pi_r}$$

with $k, l \in \{0, \ldots, \pi_r\}$ and $k \leq l$. Let $I_G$ be the set of all symbols $I_r^{k,l}$ for the dotted items of grammar $G$. Analogous to complete items, incomplete items are triples $(I_r^{k,l}, i, j)$, where $k > 0$ or $l < \pi_r$, or both.

The island seeds are represented by a set of indices $\mathcal{S} = \{i_1, \ldots, i_q\}$ of the corresponding input symbols. For the island parsing algorithm, the chart is divided into regions, such that every region contains exactly one seed. The indices of the region borders are named $p_k$, with $p_0 = 0$, $p_q = L$ and $i_k \leq p_k < i_{k+1}, k \in \{1, \ldots, q-1\}$. The region between a seed and its left border is called *right substring*, because items in this region are built in a right-to-left top-down fashion. Analogously, there is a *left substring* to the right of a seed, where items are built from left-to-right, respectively.

The algorithm is started by adding all tuples $(a_i, i-1, i)$ to $t_{i-1,i}$ and calling the procedure *add_complete* for all of these tuples afterwards. The program terminates whenever a derivation from the start symbol to the input string was found and the **exit** statement in *add_new* was reached, or else, if there are no more items to add, in which case the string is rejected.

Although the island strategy is bidirectional, i.e., items can potentially combine with other items at both sides, their direction of expansion is restricted during parsing to avoid redundant computation of sub-derivations. These restrictions are implemented using two additional two-dimensional $L+1 \times L+1$ arrays *block_left* and *block_right*, that contain symbols of $N \cup \Sigma \cup I_G$. If, for example, $A \in$ *block_right*$(i,j)$, the item $(A, i, j) \in t_{i,j}$ cannot combine with any item adjacent to its right.

To illustrate the components and the behaviour of the algorithm, figure 2 shows an example chart. Complete and incomplete items are represented by solid respectively dashed arcs, which bear the symbols from $N$, $\Sigma$, and $I_G$ as labels. The input items $a_1$ and $a_5$ are the seeds, the border between the two is at $p_1 = 2$. The little blocks at the end of the arcs depict the values of *block_left* and *block_right*, respectively.

The blocking of a complete item is based on its relation to the seeds. Items dominating a seed, i.e., items whose yield contains at least one seed, are blocked on both sides and will only be extended by the projection step in the procedure *add_complete* below. This is the case for all seed items, but also for the item labeled C, which projects to the incomplete item $S \rightarrow \bullet C \bullet B\, a_5\, a_6$. Complete items in a right substring, like the items labeled B, A, or $a_3$, are blocked at the left side and can therefore only combine with active items to their right. Thus, items in a right substring will be built right-to-left starting at the seed. Complete items in a left substring are treated analogously.

An unusual feature of this algorithm is that two incomplete items can be combined (see the second and fourth **for** loop in procedure *add_incomplete*), while other chart parsing algorithms only allow the combination of an incomplete with a complete item. At the borders, these are the only possible combinations, since all complete items have been blocked.

When incomplete items are created, they can at first extend in both directions, except for those where one of the dots is at its outermost position. Incomplete items are blocked when combined with another item for the first time. If they combine to the right, the left side will be blocked and they can only combine to the right from that time on, and vice versa. This mechanism synchronizes incomplete items at the borders between two seeds, which guarantees that items whose yield contains more than one seed are built in exactly one manner.

```
proc add_complete($NT, i, j$) $\equiv$
  if $i < i_h \leq j$ for some $h$ then
    /* project step: add $D_r \rightarrow \ldots Z_{r,k} \bullet NT \bullet Z_{r,k+2} \ldots$ */
    for $I_r^{k,k+1} \in I_G$ with $Z_{r,k+1} = NT$ do
        /* seed dominating: block both sides */
        $block\_right(i,j) := block\_right(i,j) \cup \{NT\}$
        $block\_left(i,j) := block\_left(i,j) \cup \{NT\}$
        $add\_new(I_r^{k,k+1}, i, j)$
    od
  else
    if $p_{h-1} < i < j < i_h$ for some $h$ then
      /* right substring : block complete item left */
      $block\_left(i,j) := block\_left(i,j) \cup \{NT\}$
      /* combine with $D_r \rightarrow \ldots NT \bullet Z_{r,k+1} \ldots \bullet \ldots$ */
      for $(I_r^{k,l}, j, m) \in t_{j,m}$
            with $Z_{r,k} = NT \wedge I_r^{k,l} \notin block\_left(j,m)$ do
          $block\_right(j,m) := block\_right(j,m) \cup \{I_r^{k,l}\}$
          $add\_new(I_r^{k-1,l}, i, m)$
      od
    else /* $i_h \leq i < j \leq p_{h+1}$ for some $h$ */
      /* left substring : block complete item right */
      $block\_right(i,j) = block\_right(i,j) \cup \{NT\}$
      /* combine with $D_r \rightarrow \ldots \bullet \ldots Z_{r,l} \bullet NT \ldots$ */
      for $(I_r^{k,l}, m, i) \in t_{m,i}$
            with $Z_{r,l+1} = NT \wedge I_r^{k,l} \notin block\_right(m,i)$ do
          $block\_left(m,i) := block\_left(m,i) \cup \{I_r^{k,l}\}$
          $add\_new(I_r^{k,l+1}, m, j)$
      od
    fi
  fi
end
```

```
proc add_incomplete($I_r^{k,l}, i, j$) $\equiv$
  /* $I_r^{k,l}$ : $D_r \rightarrow \cdots Z_{r,k} \bullet Z_{r,k+1} \cdots Z_{r,l} \bullet Z_{r,l+1} \cdots$ */
  if $p_{h-1} \leq i < i_h$ for some $h$ then l-predict($I_r^{k,l}, i$) fi
  if $i_h \leq j \leq p_h$ for some $h$ then r-predict($I_r^{k,l}, j$) fi
  if $k > 0 \wedge I_r^{k,l} \notin block\_left(i,j)$ then
    /* combine to the left with complete items */
    for $(Z_{r,k}, m, i) \in t_{m,i}$ with $Z_{r,k} \notin block\_right(m,i)$ do
        $block\_right(i,j) := block\_right(i,j) \cup \{I_r^{k,l}\}$
        $add\_new(I_r^{k-1,l}, m, j)$
    od
    /* combine to the left with incomplete items */
    for $(I_r^{n,k}, m, i) \in t_{m,i}$ with $I_r^{n,k} \notin block\_right(m,i)$ do
        /* block both incomplete items appropriately */
        $block\_left(m,i) := block\_left(m,i) \cup \{I_r^{n,k}\}$
        $block\_right(i,j) := block\_right(i,j) \cup \{I_r^{k,l}\}$
        $add\_new(I_r^{n,l}, m, j)$
    od
  fi
  if $l < \pi_r \wedge I_r^{k,l} \notin block\_right(i,j)$ then
    /* combine to the right with complete items */
    for $(Z_{r,l+1}, j, m) \in t_{j,m}$ with $Z_{r,l+1} \notin block\_left(j,m)$ do
        $block\_left(i,j) := block\_left(i,j) \cup \{I_r^{k,l}\}$
        $add\_new(I_r^{k,l+1}, i, m)$
    od
    /* combine to the right with incomplete items */
    for $(I_r^{l,n}, j, m) \in t_{j,m}$ with $I_r^{l,n} \notin block\_left(j,m)$ do
        $block\_left(i,j) := block\_left(i,j) \cup \{I_r^{k,l}\}$
        $block\_right(j,m) := block\_right(j,m) \cup \{I_r^{l,n}\}$
        $add\_new(I_r^{k,n}, i, m)$
    od
  fi
end
```

```
proc add_new($X, i, j$) $\equiv$
  if $X = I_r^{0,\pi_r}$ then
    if $D_r = S \wedge i = 0 \wedge j = L$ then exit(accept) fi
    if $(D_r, i, j) \notin t_{i,j}$ then
      $t_{i,j} := t_{i,j} \cup \{(D_r, i, j)\}$
      $add\_complete(D_r, i, j)$
    fi
  else
    if $(X, i, j) \notin t_{i,j}$ then
      $t_{i,j} := t_{i,j} \cup \{(X, i, j)\}$
      $add\_incomplete(X, i, j)$
    fi
  fi
end
```

```
proc l-predict($I_r^{k,l}, i$) $\equiv$
  if $Z_{r,k} \in N \wedge Z_{r,k} \notin predict\_left(i)$ then
    $predict\_left(i) := predict\_left(i) \cup \{Z_{r,k}\}$
    for $I_u^{\pi_u,\pi_u}$ with $D_u = Z_{r,k}$ do
        $add\_incomplete(I_u^{\pi_u,\pi_u}, i, i)$
        l-predict($I_u^{\pi_u,\pi_u}, i$)
    od
  fi
end
```

```
proc r-predict($I_r^{k,l}, i$) $\equiv$
  if $Z_{r,l+1} \in N \wedge Z_{r,l+1} \notin predict\_right(i)$ then
    $predict\_right(i) := predict\_right(i) \cup \{Z_{r,l+1}\}$
    for $I_u^{0,0}$ with $D_u = Z_{r,l+1}$ do
        $add\_incomplete(I_u^{0,0}, i, i)$
        r-predict($I_u^{0,0}, i$)
    od
  fi
end
```

The procedures *l-predict* and *r-predict* recursively generate top down predictions for an incomplete item, to both sides, if the item is dominating a seed, to the left, if it is in a right substring, and to the right otherwise. They keep track of the predictions generated so far using two arrays of length $L+1$, storing the nonterminals for which left or right predictions have been introduced at a specific chart node.

The loops at chart node 3 and 4 in figure 2 have been generated by *l-predict*. Items that lie completely in right or left substrings stem from these top down predictions, like the item labeled with $\mathsf{B} \rightarrow \mathsf{A} \bullet \mathsf{a}_4 \bullet$ or the complete item with label B.

For a more formal description of the algorithm, including an invariant describing its behaviour, see [Satta and Stock, 1994].

## 3 Modified Algorithm

In the modified algorithm, instead of fixing seed and border indices in advance, every chart item is assigned a *state*, which is one of right substring, left substring or seed dominating (*right*, *left* and *seed* in the pseudo-code, respectively). Additionally, complete items with a terminal category, i.e., input items, can have neutral state, in fact, they are given this state during initialization.

Because the search strategy of the parser shall be adaptable, a priority is assigned to every item, which is used in connection with a priority queue (an *agenda*) to expand the best items first. The assignment of priority values is omitted here for the sake of clarity.

During initialization, all input items are added to the chart, their state is set to neutral and they are added to the priority queue. Parsing then continues by taking the highest ranked item from the priority queue and expanding it. A seed is selected when a neutral terminal item is retrieved from the queue. Its state is updated to *seed dominating*, i.e., the item itself becomes a seed. This puts seed selection on a level with the expansion of items, simplifying the implementation of a search strategy, owing to uniformity.

If terminal items are neutral when they are combined with another item in the first or third **for** loop of the modified *add_incomplete* procedure, they change state accordingly, either to *left* or *right*, depending on whether the incomplete item grew to the left, in which case the item is now member of a right substring, or vice versa. When such a terminal item is retrieved from the priority queue later during parsing, its state is already set and it does not become a seed.

Any other complete or incomplete combined items inherit their state from their daughters: if at least one of the daughters is seed dominating, the new item becomes seed dominating too, otherwise all daughters are members of the same substring, and the new item gets assigned the same state.

All conditionals that use the seed and border indices in the original algorithm are replaced by conditionals checking the state of the items, as a consequence, the seed and border indices are no longer needed.

Instead of a string with $L$ elements, the parser gets a word graph as input. A word graph is an acyclic directed graph $W$ of terminal items $(a, i, j)$ with exactly one source and one sink node (nodes with in-degree resp. out-degree zero). The start and end node indices of the input items are typically in topological order, so that the source node gets index zero and the sink node gets the maximal end node index of all input items.

Parsing stops when either a complete derivation was found or the priority queue becomes empty, which means that the word graph must be rejected. Since all input items were added to the priority queue in the beginning, it is also guaranteed that every sub-path of the word graph has been processed properly if parsing should stop with a failure. Every input item will then have a non-neutral state, which means that it at least took part in some of the derivations.

The procedures *l-predict* and *r-predict* are the same as in the original algorithm, and are omitted here.

```
proc $add\_complete(NT, i, j)$ $\equiv$
  if $state(NT, i, j) = seed$
    /* project step: add $D_r \rightarrow \ldots Z_{r,k} \bullet NT \bullet Z_{r,k+2} \ldots$ */
    for $I_r^{k,k+1} \in I_G$ with $Z_{r,k+1} = NT$ do
      $add\_new(I_r^{k,k+1}, i, j, seed, seed)$
    od
  else
    if $state(NT, i, j) = right$
      /* combine with $D_r \rightarrow \ldots NT \bullet Z_{r,k+1} \ldots \bullet \ldots$ */
      for $(I_r^{k,l}, j, m) \in t_{j,m}$
          with $Z_{r,k} = NT \wedge I_r^{k,l} \notin block\_left(j, m)$ do
        $add\_new(I_r^{k-1,l}, i, m, right, state(I_r^{k,l}, j, m))$
        $block\_right(j, m) := block\_right(j, m) \cup \{I_r^{k,l}\}$
      od
    elsif $state(NT, i, j) = left$
        /* combine with $D_r \rightarrow \ldots \bullet \ldots Z_{r,l} \bullet NT \ldots$ */
        for $(I_r^{k,l}, m, i) \in t_{m,i}$
            with $Z_{r,l+1} = NT \wedge I_r^{k,l} \notin block\_right(m, i)$ do
          $add\_new(I_r^{k,l+1}, m, j, left, state(I_r^{k,l}, m, i))$
          $block\_left(m, i) := block\_left(m, i) \cup \{I_r^{k,l}\}$
        od
    fi
  fi
end

proc $add\_incomplete(I_r^{k,l}, i, j)$ $\equiv$
  /* $I_r^{k,l} : D_r \rightarrow \cdots Z_{r,k} \bullet Z_{r,k+1} \cdots Z_{r,l} \bullet Z_{r,l+1} \cdots$ */
  if $state(I_r^{k,l}, i, j) \in \{seed, right\}$ then $l\text{-}predict(I_r^{k,l}, i)$ fi
  if $state(I_r^{k,l}, i, j) \in \{seed, left\}$ then $r\text{-}predict(I_r^{k,l}, j)$ fi
  if $k > 0 \wedge I_r^{k,l} \notin block\_left(i, j)$ then
    /* combine to the left with complete items */
    for $(Z_{r,k}, m, i) \in t_{m,i}$
        with $state(Z_{r,k}, m, i) \in \{right, neutral\}$ do
      if $state(Z_{r,k}, m, i) = neutral$
        then $state(Z_{r,k}, m, i) := right$ fi
      $add\_new(I_r^{k-1,l}, m, j, state(Z_{r,k}, m, i), state(I_r^{k,l}, i, j))$
      $block\_right(i, j) := block\_right(i, j) \cup \{I_r^{k,l}\}$
    od
    /* combine to the left with incomplete items */
    for $(I_r^{n,k}, m, i) \in t_{m,i}$ with $I_r^{n,k} \notin block\_right(m, i)$ do
      $add\_new(I_r^{n,l}, m, j, state(I_r^{n,k}, m, i), state(I_r^{k,l}, i, j))$
      $block\_left(m, i) := block\_left(m, i) \cup \{I_r^{n,k}\}$
      $block\_right(i, j) := block\_right(i, j) \cup \{I_r^{k,l}\}$
    od
  fi
  if $l < \pi_r \wedge I_r^{k,l} \notin block\_right(i, j)$ then
    /* combine to the right with complete items */
    for $(Z_{r,l+1}, j, m) \in t_{j,m}$
        with $state(Z_{r,l+1}, j, m) \in \{left, neutral\}$ do
      if $state(Z_{r,l+1}, j, m) = neutral$
        then $state(Z_{r,l+1}, j, m) := left$ fi
      $add\_new(I_r^{k,l+1}, i, m, state(I_r^{k,l}, i, j), state(Z_{r,l+1}, j, m))$
      $block\_left(i, j) := block\_left(i, j) \cup \{I_r^{k,l}\}$
    od
    /* combine to the right with incomplete items */
    for $(I_r^{l,n}, j, m) \in t_{j,m} \wedge \neg block\_left(I_r^{l,n}, j, m)$ do
      $add\_new(I_r^{k,n}, i, m, state(I_r^{k,l}, i, j), state(I_r^{l,n}, j, m))$
      $block\_left(i, j) := block\_left(i, j) \cup \{I_r^{k,l}\}$
      $block\_right(j, m) := block\_right(j, m) \cup \{I_r^{l,n}\}$
    od
  fi
end
```

```
proc add_new(X, i, j, state1, state2) ≡
  if $state1 = seed \lor state2 = seed$
    $newstate := seed$
  elsif $state1 = right$ then $newstate := right$
  else $newstate := left$ fi
  if $X = I_r^{0,\pi_r}$ then
    $X := D_r$
    if $X = S \land i = 0 \land j = n$ then exit(accept) fi
  fi
  $state(X, i, j) := newstate$
  $t_{i,j} := t_{i,j} \cup \{(X, i, j)\}$
  push((X, i, j), p_queue)
end

proc main(W) ≡
  for $(a, i, j) \in W$ do
      $t_{i,j} := t_{i,j} \cup \{(a, i, j)\}$
      $state(a, i, j) := neutral$
      push((a, i, j), p_queue)
  od
  while ¬empty(p_queue) do
        $(X, i, j) := pop\_max(p\_queue)$
        if $X \in \Sigma$ then
          if $state(X, i, j) = neutral$ then $state(X, i, j) := seed$ fi
          add_complete(X, i, j)
        else
          if $X \in N$ then add_complete(X, i, j)
          else add_incomplete(X, i, j) fi
        fi
  od
  exit(reject)
end
```

## 4 Correctness of the modified algorithm

It is quite easy to see that if $W$ contains only string input and priorities are set appropriately to select the right seeds, the modified algorithm works like the original. It remains to be shown that in case of true word graph input, the algorithm will still be correct and redundancy-free. New situations arise from the fact that there are parallel sub-paths of neutral input items to previously treated regions of the chart, and new, possibly derived items can now interact with existing ones created from previous expansions.

Although there are six cases in total to be considered (three each for left and right substrings), the treatment of left and right substrings will be completely analogous, so we will content ourselves with the discussion of the former.

### 4.1 A new left substring ends in a right substring

In this situation, node $j$ behaves like a new border node between seed $s_0$ and $s_2$. Because of the completeness of the original algorithm, all possible derivations compatible with the old seed $s_0$ must be available at node $j$, although some of them may be blocked. Assume we lose a complete derivation because of an indispensable incomplete item that is blocked on the left side (like the item labeled $a$ in fig. 3). If this is the case, there must be ancestor items of a whose creation caused the blocking. One of these ancestors, ultimately the one that

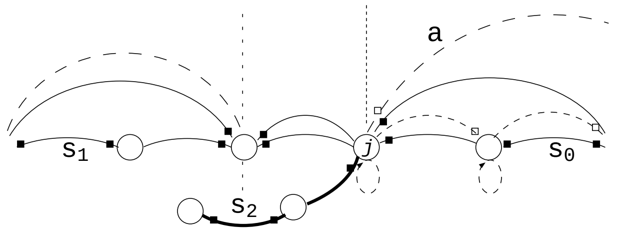

Figure 3: A new left substring ends in a right substring. The thick arcs are items that are expanded later in the parsing process.

ends in the sink node[1], is available for combination at node $j$, which is a contradiction.

A special case of the configuration described in this section is given when the new sub-path hits the old border node (e.g., node $j - 1$ in fig. 3). In this case, it is obvious that all, and only the correct derivations will be created.

### 4.2 A new left substring ends in a left substring

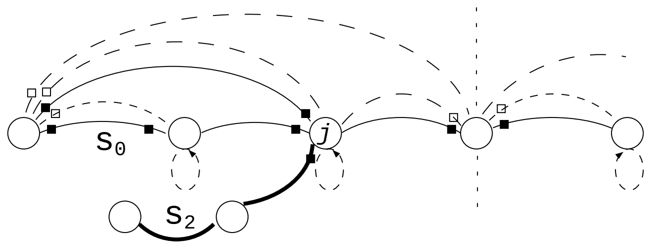

Figure 4: A new left substring ends in a left substring

All complete items starting at node $j$ are available to the new sub-path. Every derivation starting at $j$ that is compatible with the new seed $s_2$ but not with the old one ($s_0$) will be constructed by the appropriate predictions and expansions, and since the predict methods keep track of which nonterminals have already been predicted, no work is duplicated. For blocked incomplete items, the same argumentation as in 4.1 applies.

### 4.3 An alternative left substring path overruns a seed

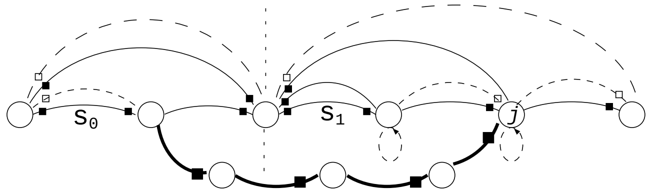

Figure 5: A seed is overrun by an alternative left substring

This is almost the same situation as in 4.2, except that the new items ending in node $j$ do not come from a new seed on

[1]Incomplete items ending in the sink node can not be blocked at the left side because there is no item to the right they can combine with.

the parallel path, but from an alternative path from seed $s_0$. Therefore, the same argumentation applies as in 4.2 above.

## 5 Conclusion and further considerations

An efficient island parsing algorithm for string input was generalized to make it more feasible for the use in speech applications. The new version deals with word graphs as input without losing the beneficial properties of the original. It also integrates the selection of seeds into the parser's search process, which, in addition to more uniformity, provides the user with more flexibility in the design of the search strategy.

The data structures for blocking and keeping the state of an item can be implemented as bit vectors, which produces minimal space and time overhead for all the blocking and state conditionals.

The modified algorithm has been implemented for context free grammars with annotated feature structures. This implementation also provides pluggable search strategies to facilitate experimentation.

From the point of view of the search strategy, the atomic action of the modified algorithm (one *parsing task*) is the expansion of an item. To be able to define a more fine grained strategy, the parser could be changed such that the tasks are instead combination of two items, projection and prediction, or a subset of the three ([Kay, 1986], [Erbach, 1991]).

The price to pay for the increased flexibility is a larger agenda, maybe prohibitively large, if the word graphs are big and/or the grammar is highly ambiguous. The changes to the algorithm are obvious, and it will depend on the specific task, whether the more elaborate search strategy will achieve better results or improved parsing efficiency.

## Acknowledgments

I am very indebted to Giorgio Satta for his help in fully understanding the bits and pieces of the original algorithm, for the discussions and for his patience. I also want to thank my proofreaders for their help in preparing this paper. All remaining errors are of course mine.

## References

[Ageno, 2003] Alicia Ageno. *An Island-Driven Parsing System*. PhD thesis, Universitat Politècnica de Catalunya, 2003.

[Brietzmann, 1992] Astrid Brietzmann. “Reif für die Insel”. Syntaktische Analyse natürlich gesprochener Sprache durch bidirektionales Chart-Parsing. In Helmut Mangold, editor, *Sprachliche Mensch-Maschine-Kommunikation*. Oldenbourg, München; Wien, 1992.

[Erbach, 1991] Gregor Erbach. An environment for experimentation with parsing strategies. In *Proceedings of the 12th International Conference on Artificial Intelligence*, pages 931–936, 1991.

[Gallwitz *et al.*, 1998] F. Gallwitz, M. Aretoulaki, M. Boros, J. Haas, S. Harbeck, R. Huber, H. Niemann, and E. Nöth. The Erlangen Spoken Dialogue System EVAR: A State–of–the–Art Information Retrieval System. In *Proceedings of 1998 International Symposium on Spoken Dialogue (ISSD 98)*, pages 19–26, Sydney, Australia, 1998.

[Kay, 1986] Martin Kay. Algorithm schemata and data structures in syntactic processing. In B. J. Grosz, K. Sparck Jones, and B. L. Webber, editors, *Natural Language Processing*, pages 35–70. Kaufmann, Los Altos, CA, 1986.

[Mecklenburg *et al.*, 1995] Klaus Mecklenburg, Paul Heisterkamp, and Gerhard Hanrieder. A robust parser for continuous spoken language using prolog. In *Proceedings of the Fifth International Workshop on Natural Language Understanding and Logic Programming (NLULP 95)*, pages 127–141, Lisbon, Portugal, 1995.

[Satta and Stock, 1994] Giorgio Satta and Oliviero Stock. Bidirectional context-free grammar parsing for natural language processing. *Artifical Intelligence*, 69:123–164, 1994.

[Thanopoulos *et al.*, 1997] A. Thanopoulos, N. Fakotakis, and G. Kokkinakis. Linguistic processor for a spoken dialogue system based on island parsing techniques. In *Proc. of 5th Eurospeech*, volume 4, pages 2259–2262, 1997.