Language, Information and Computation (PACLIC12), 18-20 Feb, 1998, 316-327

The Advantages of 3D-Trees in Modeling Human Sentence Processing

Charles C. Lee[*]
National Chiao Tung University

A 3D-Tree is an efficient representation of a set of regular syntactic trees (2D-trees) that have the same tree skeleton but which may have different labels for the nodes. A node of a 3D-tree may have one or more sets of children as well as one or more parent nodes as long as the 2D-trees represented have the same tree skeleton. A 3D-tree is therefore able to efficiently represent the type of ambiguous structures which never seem to cause the Human Sentence Processor (HSP) to garden path. The advantages of 3D-trees over other parallel structures and the computational properties of building, manipulating, and maintaining, 3D-trees are presented in this paper. In addition, when the number and accessibility of 3D-trees are limited by the memory architecture of a parser in order to model certain psycholinguistic phenomena, the parser has the desirable computational properties of linear space requirements for parsing any sentence, and linear time requirements for parsing non-garden path sentences.

1. INTRODUCTION

In modeling human sentence processing performance, many models appeal to limitation in the memory architecture of the Human Sentence Processor (HSP) to explain the HSP's performance in processing non-garden path and garden path sentences (Frazier and Fodor, 1978; Church, 1982; Gorrell, 1987; Gibson, 1991; Jurafsky, 1992). However, upon closer examination, the memory architectures proposed by many of the parallel models (e.g. Gibson, 1991; Jurafsky, 1992) do not *inherently* explain nor motivate (separate from their parsing algorithms or evaluation measures) the differences between local syntactic ambiguities which may cause the HSP to garden path, and those that never do (Lee, 1996). In addition, the HSP seems to require linear time for processing non-garden path sentences, and non-linear time for garden path sentences (MacDonald, 1992).[1] Unrestricted parallel memory architectures do not inherently restrict a parser to these time properties.

The dilemma faced when modeling the HSP is that the parser must at times be able to easily pursue multiple analyses in parallel, but at other times be unable to do so or to have difficulty in doing so. In order to model the HSP's ability to pursue multiple analyses, some models have very powerful memory architectures. Rather than have the memory architecture limit the parser's ability to pursue multiple analyses in parallel when garden pathing is required, they use the parsing algorithm and an evaluation measure (Gorell, 1987; Gibson, 1991; Jurafsky, 1992). However, the memory architecture of these parsers alone does not explain why the parser should have to choose between multiple analyses at all, especially when there are only two possible analyses. That is, at times, the parser is able to easily maintain two alternate structures in memory (in order to avoid spurious garden pathing), and so it is unclear from the memory architecture alone why at other times (when garden pathing is required) the parser is suddenly unable to do so. In addition, when the memory architecture is considered alone, the inherent time and space properties of a parser with these types of powerful memory architectures are non-linear on even non-garden path sentences.

So a desirable memory architecture should inherently allow the parser to easily pursue only the kind of structural ambiguities that the HSP finds easy to pursue, and should inherently constrain the parser to have linear time (and space) properties on non-garden path sentences. At the same time, it should also inherently explain why the parser should have to choose between two analyses at all (i.e. why limited memory should cause the parser to garden path at all). A parser which uses *3D-trees*, a very restricted parallel memory structure, and limits the number and accessibility of 3D-trees by its memory architecture, can adequately model the psycholinguistic data that originally motivated these parallel models, and has the desirable properties of linear

[*] Department of Foreign Languages and Literatures, National Chiao Tung University, 1001 Ta Hsueh Road, Hsinchu, Taiwan R.O.C. 30050, Email: clee@cc.nctu.edu.tw

[1] It is unclear what the space requirements might be for non-garden path and garden path sentences, though it seems that if memory is limited, then linear requirements would seem very reasonable, at least for non-garden path sentences, and maybe even for the processing of garden path sentences.

time and space requirements when processing any non-garden path sentences. Therefore, these unrestricted parallel memory architectures are not required to model the psycholinguistic data which motivated them.

This paper describes the 3D-tree structure, showing how it allows the parser to easily pursue those types of ambiguities that never seem to cause the HSP to garden path. In addition, the computational properties of building and maintaining 3D-trees are presented. Finally, it shows how 3D-trees, a performance grammar, and the memory architecture proposed by Lee (1996) gives the parser inherent linear time and space requirements for processing non-garden path sentences.

2. THE 3D-TREE

A 3D-tree is an efficient representation of a set of regular syntactic trees (2D-trees) that all have the same tree skeleton, but which may have different labels for the nodes. Figure1 illustrates how two 2D-trees that have the same skeletal tree structure can be merged into one 3D-tree.

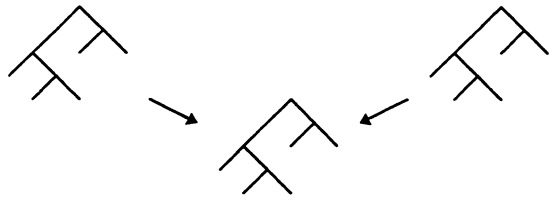


Figure 1: A Possible 3D-Tree

On the other hand, Figure 2 illustrates how a 3D-tree cannot represent both a left-branching and a right-branching structure because they do not have the same tree skeleton.

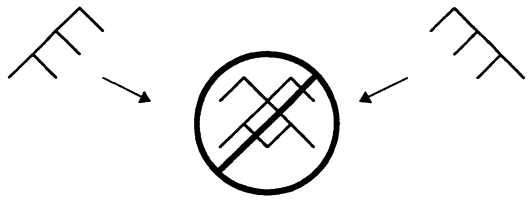


Figure 2: An Impossible 3D-Tree

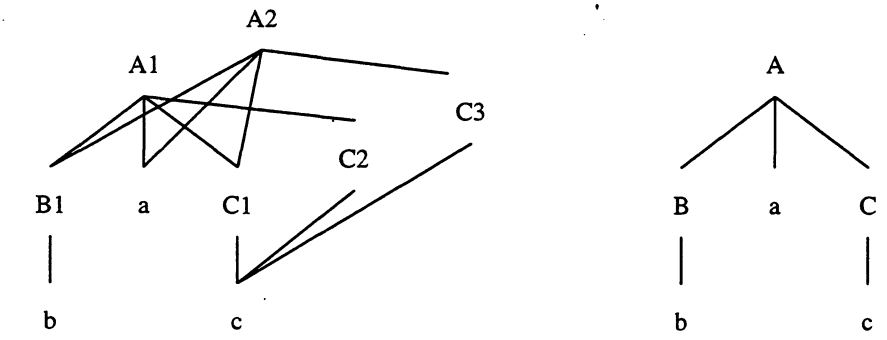
A node in a 3D-tree is a *3D-node* and consists of a set of *3D-nodettes*. Like a node in a 2D-tree, a 3D-node may only have one parent 3D-node, and a single set of 3D-nodes as children (i.e. there is a single skeleton structure associated with each 3D-node). The fact that a 3D-node consists of a set of 3D-nodettes gives a 3D-tree an additional dimension (a third dimension) over a traditional 2D-tree. It is this third dimension (i.e. 3D-nodettes) which gives a 3D-tree its unique properties. 3D-nodettes are unlike 2D-nodes since a 3D-nodette may have multiple parents (as long as all parent 3D-nodettes belong to the same 3D-node), and may have multiple sets of child 3D-nodettes (as long as the constraint regarding the same tree skeleton is maintained). This structure is therefore able to represent any set of 2D-trees which have the same tree skeleton, but which may have different labels. A 3D-tree efficiently represents a set of 2D-trees because it in fact does *not* represent them with a set of 2D-trees like the parallel models.

Although 3D-trees are similar to the Tomita's (1987) packed shared forest in that a 3D-node and a packed node of a packed shared forest may consist of more than one node of a regular 2D-tree, it differs in two important ways. A 3D-tree cannot represent 2D-trees that have different skeletal tree structures while a packed shared forest can. The other difference is that a 3D-node can consist of subnodes that have different labels while a packed node cannot. The significance of these differences is presented later.

2.1 An Example 3D-tree

Figure 3 gives an example of a possible 3D-tree which illustrates the unique features of a 3D-tree. The 3D-nodettes are labeled with XN, where the number N identifies the 3D-nodette as the N^{th} 3D-nodette of the 3D-node identified by the letter X. Notice that a 3D-node may or may not have more than one 3D-nodette. The 3D-nodes A and C have more than one 3D-nodette while the 3D-node B has only one 3D-nodette. The lexical nodes

a, b, and c are the *3D-heads* of the 3D-nodes A, B and C, respectively. Also notice that a 3D-nodette may have more than one set of children. The 3D-nodette A1 has B1 and C1, and B1 and C2 as children while A2 has B1 and C1, and B1 and C3 as children. Finally, a 3D-nodette may also have more than one parent. The 3D-nodettes B1 and C1 have two parent 3D-nodettes each, while the others have only one parent each.



Viewed from the upper right

Viewed straight on

Figure 3: An Example 3D-tree

A significant difference between a set of 2D-trees and a single 3D-tree is in the way ambiguity is represented. A set of 2D-trees " multiplies out'" or enumerates all ambiguity, while a 3D-tree does not. With a set of 2D-trees, a new tree must be created for each analysis. Multiple attachments between the 3D-nodettes of two 3D-nodes is kept local to just the two nodes. Multiple attachments between other 3D-nodes, either above or below this point in a 3D-tree, does not affect the attachment between these two nodes. Therefore, ambiguity in a 3D-tree is not propagated either up or down the tree. That is, ambiguity is not multiplied out in a 3D-tree.

2.2 Why a 3D-tree?

Why is a tree structure which can easily represent ambiguity between trees that have the same tree skeleton useful? It is because it seems that ambiguity which requires the HSP to build and maintain analyses which have the same tree skeleton is easy for the HSP and never seems to cause the HSP to garden path.[2]

(1) a. John wants...
b. John wants the dog.
c. John wants to leave.
d. John wants the dog...
e. John wants the dog to leave.

For example, when the HSP is processing (1a), there is syntactic ambiguity with respect to the verb *wants*. It could be a simple transitive verb (as in (1b)), a subject control verb (as in (1c)), or an object control verb (as in (1e)). Regardless of how the sentence finishes, the HSP does not garden path nor does it have any difficulty processing the sentence. Therefore, when the HSP has only seen (1a), it must be able to easily pursue all three analyses. Since the three analyses that span (1a) have the same tree skeleton, these three analyses can be easily pursued with one 3D-tree. Similarly, when the HSP has only seen (1d), it must also be able to pursue two of the analyses since the sentence fragment is still ambiguous between (1b) and (1e). Again, a single 3D-tree representing both analyses can span (1d).

While the example in (1) involves syntactic ambiguity that is eventually resolved, the example in (2) shows that the same is true for syntactic ambiguity that may not be eventually resolved. When the HSP has seen (2a), it is faced with the syntactic ambiguity between at least two senses of the word *rent*. One sense has *John* as the agent-source (as in (2c,d)) and the other has *John* as the agent-recipient (as in (2f)). The sentence in (2b) is ambiguous between these two readings. Regardless of how the sentence fragment in (2a) or (2d) ends, the HSP does not garden path. The use of a 3D-tree would allow the parser to easily pursue both analyses that span (2a) or (2d) since both analyses can be easily represented with one 3D-tree.

[2] The argument presented here assumes that the HSP builds an interpretation of a sentence in generally a word-by-word or constituent-by-constituent manner without waiting for the end of the sentence or clause (Marslen, 1975; Swinney, 1979; Tanenhaus, 1979; Potter, 1979; Tyler, 1982; Marslen, 1988; Altmann, 1988).

(2) a. John rented...
b. John rented an apartment.
c. John rented Mary an apartment.
d. John rented an apartment...
e. John rented an apartment to John.
f. John rented an apartment from John.

The two examples above involve lexical ambiguity of a word between senses with the same category (verbs in these examples). However, there are also examples of lexical ambiguity between senses with different categories which never seem to cause the HSP to garden path.

(3) a. The claim that Mary made a mistake was known by everyone.
b. The fact that Mary knows the man was not obvious to me.
c. The knowledge that Mary wants a car was kept a secret from me.

Take the noun complement clauses in (3) and the relative clauses in (4). Up to the fifth word in each of these sentences, the HSP does not know whether the post-nominal modifier is a relative clause or a noun complement clause. That is because the word *that* is ambiguous between a relative clause marker and a sentential complement marker (i.e. whether there is a filler and gap relationship, or not). However, regardless of whether the sentences end as in (3) or as in (4), the HSP never seems to garden path. Since both analyses can be represented by 2D-trees with the same skeleton structure, they can both be represented in a single 3D-tree and so a parser which builds and maintains 3D-trees can easily pursue both analyses in parallel.

(4) a. The claim that Mary made was known by everyone.
b. The fact that Mary knows was not obvious to me.
c. The knowledge that Mary wants was kept a secret from me.

While there are other memory architectures that strive to achieve efficient representation of multiple trees, the fact that a 3D-tree does not multiply out ambiguity either up or down a tree (when they have the same tree skeleton) sets it apart from them. For example, a chart (Kay, 1986) allows multiple parent edges to share the same daughter edge (i.e. it does not multiply ambiguity down a tree). However, a chart still multiplies ambiguity up a tree because an ambiguous constituent that spans the same words is represented with distinct edges. So even if the tree structure above this constituent is unambiguous, the structure above this edge must be duplicated for each inactive edge of the ambiguous constituent. While packed shared forests (Tomita, 1987) improve upon the chart by preventing multiplication of some ambiguities up a tree (i.e. ambiguity between nodes which span the same input and have the same node label), a packed shared forest does not efficiently represent the kind of ambiguity just mentioned regarding the word *that* since the ambiguity is between different node labels and so they cannot be represented by a single packed node.

Another advantage that a 3D-tree has over other parallel memory architectures (Kay, 1986; Gorrell, 1987; Tomita, 1987; Gibson, 1991; Jurafsky, 1992) is that it inherently explains why the HSP should garden path at all. Since a 3D-tree cannot represent analyses that have different tree skeletons, the only way it can pursue such analyses in parallel is to leave two 3D-trees unattached. However, this requires the HSP to keep track of an additional 3D-tree. Therefore, there is a natural memory cost to pursuing analyses that do not have the same tree skeleton. This explains why the HSP would need to eventually commit to one analysis over another (i.e. to reduce the number of 3D-trees in memory).

(5) a. The horse raced past the barn.
b. The man gave the girl a ring impressed a watch.
c. After the man drank the water proved to be poisoned.
d. Without her donations to the charity failed to appear.

For example, all of the sentences in (5) have a local syntactic ambiguity that would require a parser that could only build 3D-trees to eventually choose to commit to one analysis over another in order to reduce the memory load of having an additional 3D-tree in memory. That is, unlike the non-garden path examples above, a single 3D-tree cannot represent both analyses and therefore the parser cannot easily pursue both analyses. Since the HSP garden paths on these sentences, the parsing algorithm of the HSP must eventually commit to the wrong analysis over the correct one.

If the memory architecture also restricts the number of unattached 3D-trees that are allowed in memory, then the HSP can only postpone a bounded number of attachments at any time. This also naturally explains why there do not seem to be any ambiguities between a purely left- and a purely right-branching structure (with more than one or two branching nodes) that are easy to pursue and do not cause garden pathing. That is, pursuing a purely right-branching and a purely left-branching structure at the same time would require the parser to leave a 3D-

tree unattached for each word. Doing so becomes increasing difficult with each word and its ability to do so is bounded.

3. The Computational Properties of 3D-Trees

There are several assumptions made in determining the computational properties of 3D-trees. One main assumption is that the grammar and lexicon are considered to be constant and the size of the grammar/lexicon is bounded. It is critical to the argument presented that the grammar is constant (i.e. it doesn't change in size while processing a sentence). If processing a word was allowed to increase the size of the grammar or lexicon, then the time required to process a sentence in the worst case would not be linear since the time to look up a word or retrieve the information for a word could grow with the length of the sentence processed. Therefore the time required for lexical look up when processing a sentence of length n would be on the order of n^2. The size of the grammar/lexicon must also be bounded (being constant doesn't necessarily mean it is unbounded) otherwise simply processing the lexical entry of a word would require unbounded time and/or unbounded space. This is a reasonable assumption since we are concerned with modeling human performance and not competence. For example, while a verb can theoretically take an infinite number of adverbial modifiers (i.e. have an unbounded number of children), a performance grammar can place a bound on the number of such modifiers since the HSP is unable to process a sentence with an unbounded number of adverbials.

There are no restrictions on the type of grammar used as long as the grammar can be reduced to a constraint based representation of the constituents licensed as siblings of the head. That is, it is assumed that the different senses of a word are retrieved from the lexicon, and with each sense, a set of licensing frames. A licensing frame specifies the possible daughter 3D-nodettes and their order. Since the grammar is assumed to be constant and bounded, let e be the maximum number of senses a word may have, and let f be the maximum number of licensing frames per sense, and let i be the maximum number of items in a licensing frame. The values for e, f, and i are constants.

To facilitate the presentation below, let the list of parent 3D-nodettes for a 3D-nodette be called the *PList*, and a set of daughter 3D-nodettes the *DSet*, and the set of DSets (i.e. the list of the sets of children), the *CList*. Also, let the attachment between 3D-nodettes to be called a *link*. A DSet also contains the current state of the associated licensing frame as well as whether the DSet is complete or not.

Another assumption is that each 3D-node in a 3D-tree is headed by only one word and each word is associated with a bounded number of 3D-nodes. To simplify the presentation below, it is assumed that the grammar only allows each word to head only one 3D-node. This last assumption is not necessary for the computational properties of a 3D-tree as long as the two assumptions about the grammar and lexicon are still maintained. If a word can head more than one 3D-node, there would still be an upper bound to the number of such 3d-nodes since the grammar is bounded, and the argument below would still hold.

3.1. Properties of a 3D-node

When a 3D-node is attached as a daughter to a 3D-nodette, the maximum number of links (l_1) is the maximum number of licensing frame items in the parent 3D-nodette (fi) multiplied by the maximum number 3D-nodettes in the daughter node (e), or $l_1 = fie$. After every attachment, there is one DSet for each link in the parent 3D-nodette, and so after the first attachment, there would be l_1 DSets which each could have at most $i - 1$ more links. Therefore, after m attachments:

$$l_m = l_{m-1}(i - m + 1)e = f(i)(i - 1)\ldots(i - m + 1)e^m$$

The maximum number of links (l) that is possible between a 3D-nodette and a daughter 3D-node (also the maximum number of DSets in a 3D-nodette) is at the *ith* attachment (l_i) and so l is the constant:

$$l = l_i = l_{i-1}(i - i + 1)e = f(i)(i - 1)\ldots(1)e^i = f(i!)e^i$$

The fact that l is a constant means that the maximum number of links between a 3D-nodette and a daughter 3D-node does not depend on the number of words in the sentence.[3] Therefore, the maximum number of links between two 3D-nodes (L) is:

[3] While this is the theoretical maximum that is used for argumentation purposes, in practice, the maximum links l is around 10 and the average is around 3 even though i in practice is around 20 and e is around 10. So the amount of ambiguity possible in a real grammar is no where near this theoretical maximum.

$$L = el$$

Given this, the maximum number of links between a 3D-nodette and a parent 3D-node is *el* divided by the maximum number of 3D-nodettes in a 3D-node (*e*), or in other words *l*. The important thing to note is that the maximum number of links between a 3D-nodette and a parent 3D-node or a 3D-nodette and a daughter 3D-node is *l* and is a constant. Note that *L* is also the maximum number of DSets in a 3D-node.

The maximum space required for a 3D-nodette is the space required for the PList and CList plus the node label. The space required for the node label is constant. The space required for the PList is proportional to the maximum number of links between a 3D-nodette and a parent 3D-node which was shown above to be *l*. The space required for the CList is proportional to the maximum number of DSets and the maximum space required for a DSet. The maximum number of DSets was shown above to be *l*, and the maximum space required for a DSet is proportional to the maximum number of items in a licensing frame *i*. So the maximum space required for a 3D-nodette is proportional to $l + li + 1$. Therefore, the maximum space required for a 3D-node (*d*) is also constant:

$$d = e(l + li + 1) = L + Li + e = L(1 + i) + e$$

The fact that *L* (the maximum number of links between two 3D-nodes) and *d* (the maximum space required for a 3D-node) are constant is central to the computational properties of building and maintaining 3D-trees.

3.2. Linear Space Requirements

Recall that each 3D-node is headed by one word and each word heads only one node. Therefore, the maximum space *s* required by a 3D-tree to span *w* words is equal to the sum of the maximum space required by each 3D-node of the tree.

$$s = dw$$

This means that the amount of structural information in a 3D-tree is directly proportional to *w*. A useful fact to note here is that *w* is also directly proportional to the number of attachments required to build a 3D-tree, which is $w - 1$. These two facts are crucial to the discussion of the time requirements of building and maintaining 3D-trees.

3.3. Time Requirements for Attaching

Since the maximum number of licensing frame items (*L*) and the maximum number of possible links (*L*) for a 3D-node are constant, the time required to check and attach two 3D-trees (2*L*) is constant. However, attaching two 3D-trees also requires the parent and daughter 3D-trees to be made consistent with the links just made. If a daughter 3D-nodette is not linked (i.e. the PList is empty), then that 3D-nodette must be pruned. If a parent DSet did not have a daughter 3D-nodette linked to it, then that DSet must be pruned. Therefore, the daughter 3D-tree needs to be pruned downwards, and the parent 3D-tree needs to be pruned upwards to make the resulting 3D-tree coherent. The time required to perform these two types of pruning actions is discussed next.

3.4. Time Requirements for Pruning

A *pruning pass* prunes a 3D-tree from bottom up or from top down. Since pruning a 3D-tree removes structural information from a 3D-tree (i.e. removes 3D-nodettes, DSets, and PList items), the time required for a single pruning pass of a 3D-tree is bounded by the amount of structural information in the 3D-tree, since it can at most remove all of the structural information in the 3D-tree. Therefore, the time required to perform a single pruning pass of a 3D-tree is at most proportional to the number of words it spans (*w*). If nothing is pruned, then a pruning pass must only check at most *L* attachments. Therefore, the minimum amount of time required by a pruning pass is constant time, since *L* is constant.

An interesting property of a 3D-tree is that the sum effect of pruning a 3D-tree while building it is directly proportional to the total number of attachments made during the parse. If a parser did not garden path while building a 3D-tree, then it made $w - 1$ attachments. There are two cases that need to be examined to prove this. The first case is if each pruning pass were to remove some information from a 3D-tree, then the sum of the time that would be required by all of the pruning passes would be at worst proportional to the amount of structural information in the 3D-tree (*w*). But the number of attach transitions required to build this 3D-tree is also proportional to *w*, so the sum contribution of pruning is directly proportional to the number of attachments.

The second case is if the pruning passes never remove any information from a 3D-tree. Then for each pruning pass, only constant time would be required. The number of times this check could be performed is at each attachment, and so the time required by all of the pruning passes is also proportional to the number of attachments, $w - 1$. So while it is possible for a single pruning pass to require time proportional to w, the sum effect of pruning while building a 3D-tree which spans w words is proportional to the number of attachments made.

Even if a parser garden paths while building a 3D-tree, the time required for pruning is still proportional to the number of attachments since undoing an attachment requires no pruning time. The key to both the garden path and non-garden path situation is that the maximum amount of structural information that can be pruned during a parse is proportional to the number of attachments made, and pruning can at most remove all of the structural information in a 3D-tree.

3.5. Time for Closing

Closing a 3D-tree removes all the incomplete DSets in the 3D-tree from the bottom nodes on up. At each 3D-nodette, DSets which are pruned because daughter 3D-nodettes were pruned are also removed in order to make the 3D-tree coherent. Since all of the 3D-nodes of a 3D-tree must be checked when closing, the time required to close a 3D-tree is proportional to w, the number of words the tree spans. Once a 3D-tree is closed, no further processing is required or possible on this 3D-tree fragment. The sum effect of closing 3D-trees during a parse is proportional to w since during a parse, the number of 3D-nodes that need to be and can be closed is proportional to the number of words w in the input sentence.

4. LIMITED NUMBER OF AND LIMITED ACCESS TO 3D-TREES

The computational properties of 3D-trees does not alone guarantee that a parser which uses 3D-trees will have desirable computational properties. The arguments above were for building a single 3D-tree that spans the whole input. However, if a parser is allowed to build an unbounded number of 3D-trees which each span the whole input (i.e. it can build and maintain analyses of the input which cannot be represented by a single 3D-tree by using multiple 3D-trees), then although the amount of time and space required is still proportional to the number of attachments made, the number of attachments is no longer necessarily proportional to the number of words w seen (even on non-garden path parses) since the number of 3D-trees which span the input may grow with the number of words seen. Therefore, in order to benefit from the computational properties of 3D-trees, a parser must limit the number of 3D-trees that span the whole input.

In addition, the parser must also limit the accessibility of the 3D-nodes to the parsing algorithm otherwise the time required by the parsing algorithm may be on the order of w^2, even on non-garden path parses. That is, for each word, if the parsing algorithm were allowed to examine every 3D-node, then at each step it would require time proportional to w, making the time required for w words proportional to w^2.

4.1 The TriSer Model

A specific parser model which limits the number and accessibility of 3D-trees is presented below in order to make the arguments, about the time and space properties of a parser which uses 3D-trees, more concrete. The model is the TriSer model proposed by Lee (1996). There are a number of motivations for the specific design TriSer that are related to various psycholinguistic phenomena (e.g. center-embedded sentences, high attachment sentences, garden path sentences) but they will not be presented here. What is important is that TriSer, using 3D-trees, is able to model the same psycholinguistic phenomena that motivated the unrestricted parallel memory models, and was in fact successfully tested on a suite of over 600 sentences (Lee, 1996).

TriSer has a *workspace* which contains unattached 3D-trees. The 3D-trees serve as a single linear representation of the constituents that completely span the words that have been seen. Each 3D-tree in the workspace is contained in a memory structure called a *chunk* which restricts the accessibility of the nodes of a 3D-tree. There can be at most five chunks in the workspace, and only the three rightmost chunks are immediately accessible. One of the immediately accessible chunks is always in *focus* and is referred to as the F1 chunk, with the two chunks to its left referred to as F2 and F3.

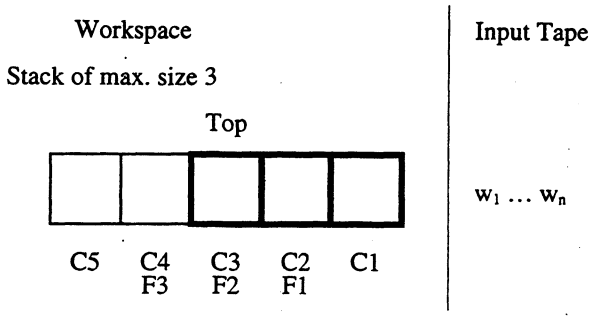


Figure 4: The Workspace Structure

The workspace structure is illustrated in Figure 4, with the immediately accessible chunks indicated by the thick squares. For this configuration, C2 is in focus. Operations that attach two 3D-trees can only operate on F1 and F2. An example configuration with only three chunks is shown in Figure 5, where the rightmost chunk is ZP, and is also the chunk in focus; and the leftmost chunk is XP. Unlike some parsing models, TriSer does not lookahead into the input stream.

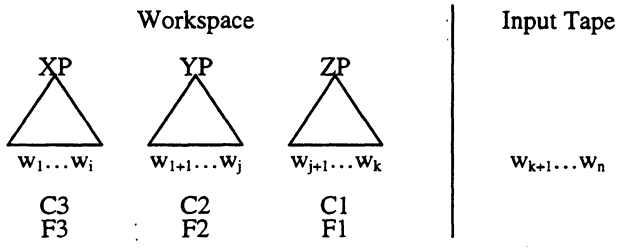


Figure 5: An Example Configuration

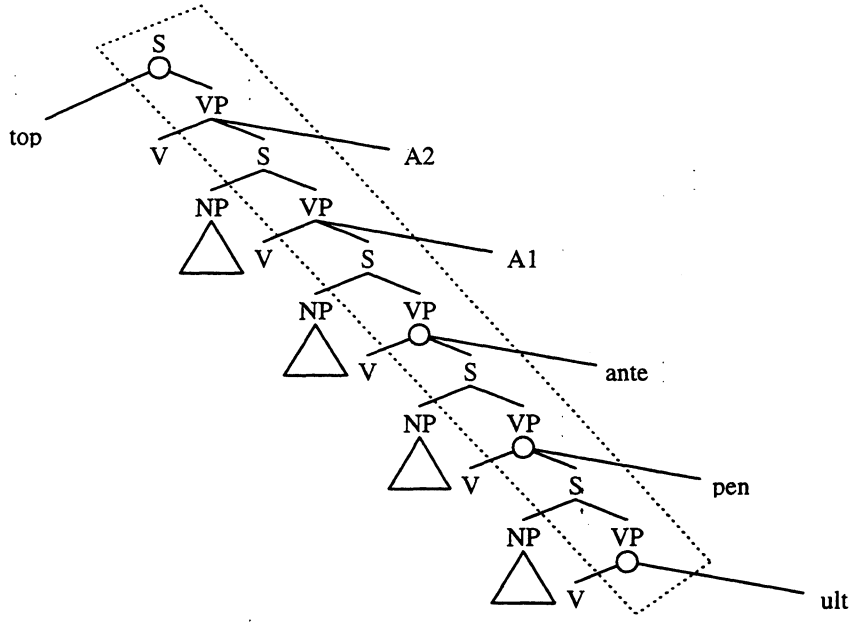


Figure 6: The Graded Accessibility of a Chunk

For TriSer, the chunk structure restricts the nodes of a 3D-tree which are accessible to the parsing algorithm. The tree in Figure 6 illustrates how access to the nodes are restricted by the chunk structure. The nodes marked by a circle are the only immediately accessible nodes in the chunk, and the two frontier nodes (A1 and A2) are accessible, although not immediately accessible. The top node is also immediately accessible, but only for attachment as a daughter or for attachment of leftward daughters. All other nodes are inaccessible. The *right frontier* of a 3D-tree consists of the *unsaturated nodes* along the right edge of the tree. Unsaturated nodes are those nodes that can have additional 3D-trees attached to their right. In this example, the right edge of the tree is enclosed by a dashed rectangle.

Therefore, the chunk structure consists of two basic parts: the *top node* (TOP) and the *frontier*. The frontier represents the right frontier of the 3D-tree. The three immediately accessible nodes of the frontier, from bottom on up, are the *ultimate node* (ULT), the *penultimate node* (PEN), and the *antepenultimate node* (ANTE). Since these three nodes are the only ones that can take right attachments and must often be referred to as a group, they shall be referred to as the *UPA* nodes.

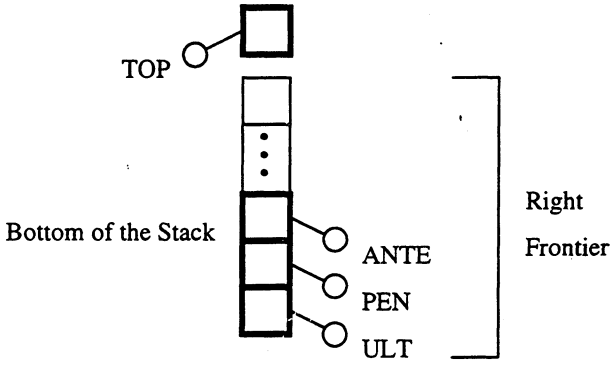


Figure 7: The Chunk Structure

Figure 7 illustrates the chunk structure. The memory structure of the frontier is a list of two nodes for the ULT and PEN nodes, as well as a pushup stack of nodes with the bottom of the stack as the ANTE node.

The memory architecture of TriSer is similar to the one proposed by Abney (1989) except for several key differences. One difference is that the trees built and maintained by the parser are *3D-trees*. The second difference is that access to the nodes in the 3D-trees, through the workspace and chunk structure, are restricted by a memory architecture that limits the total number of 3D-trees in memory, the number of immediately accessible 3D-trees, and the number of immediately accessible nodes in the immediately accessible 3D-trees. Abney (1989) does not have these restrictions.

4.2 The Operation of TriSer

TriSer is a non-deterministic parser that performs a depth-first search. There are a bounded number of transitions out of each state and the transitions out of a state are ordered. There are a total of eleven types of transitions, one of them being the *read* transition which reads in the next word from the input tape, and the rest are *epsilon* transitions (i.e. they do not read any input but just take the parser from one state to another). The transitions are designed in such a way that there can be at most seven epsilon transitions between each read transition.

4.3 The Time Requirements of TriSer

The time required by TriSer is the sum of the time required for taking or backtracking over each transitions and the time required by the parsing algorithm at each state to determine the set of transitions that are possible out of that state and the order in which to take them. The time required by TriSer for taking or backtracking over each transition is at best proportional to the number of transitions taken since:

1. a transition minimally takes constant time,
2. the number of transitions backtracked over is at most proportional to the number of transitions taken, and
3. the time required to backtrack over a transition is equivalent to the amount of time to take a transition.

The transitions with potentially the greatest time requirements are those that involve pruning 3D-trees since they may at worst have time requirements proportional to w. For TriSer, these include transitions other than attachment transitions. However, using a similar argument to the one above which showed that the time requirements of a parser to build and maintain 3D-tress is proportional to the number of attachments, it can be argued that the time required by TriSer to build and maintain 3D-trees is also at worst proportional to the number of transitions taken.[4]

[4] Since the amount of structural information in the 3D-trees is proportional to the number of attachments, the total amount of information that can be removed by all of the transitions which prune 3D-trees is also

The time required by the parsing algorithm during a parse is also proportional to the number of transitions taken. At each new state, the parsing algorithm must decide the set of possible transitions to take and the order in which to try them. At worst, the parsing algorithm must evaluate a state before each transition (i.e. if it takes only one transition out of each state), so the parsing algorithm must do this evaluation at most once for each transition. But since the number of transitions is proportional to the number of words w spanned by the workspace, the number of times the parsing algorithm must do this evaluation is also proportional to w.

Limiting the number of immediately accessible 3D-nodes in the workspace makes the amount of information to be considered by the parsing algorithm bounded (i.e. constant). Therefore, the amount of time required by the parsing algorithm to examine or close the immediately accessible 3D-nodes at each state is constant. Upon backtracking, no additional time is required by the parsing algorithm since all possible transitions out of a state and their order are determined only once for each state. Therefore the time required by the parsing algorithm is proportional to the number of transitions taken.

The parsing algorithm also needs to know if an immediately accessible chunk in the workspace is closeable and which of the top 3D-nodettes would remain if the chunk were closed. If the parsing algorithm were to actually close a chunk in order to generate this closure information, then the time required would be proportional to w at each new state, making the time required for a parse to be proportional to w^2. But it is not necessary to close a chunk at each state in order to have access to this closure information. Instead, this closure information can be updated during each attachment since this closure information will only change after an attachment. However, if the grammar has licensing dependencies between a node and a descendant node at an unbounded depth (i.e. the dependencies can span the full height of a 3D-tree), then maintaining this closure information at each attachment would still at worst require time proportional to w. But one feature of natural languages is that licensing constraints are local in nature and so there are no such unbounded licensing dependencies (Pollard and Sag, 1994).[5] Therefore, the number of ancestor 3D-nodes whose closure information must be updated above the point of attachment is bounded by the grammar and so the time required to update this information at each attachment is constant. While gap-filler dependencies are long distance dependencies that are unbounded, determining the closure information of a 3D-tree that has such a long distance dependency only requires constant time since it only requires checking if the filler has found a gap. Therefore, the total time required by the parser to maintain the closure information of the 3D-trees is at worst proportional to the number of transitions taken, which again means it is proportional to the number of words w which is spanned by the workspace.

So for a parse, the time required by TriSer is proportional to the number of transitions taken since the time required by the parsing algorithm is at worst proportional to the number of transitions taken and the time required for taking or backtracking over the transitions is proportional to the number of transitions taken. Recall that the number of transitions taken is proportional to the number of read transitions taken. And so since each word is read once and only once during a non-garden path parse of a sentence, the number of transitions taken in parsing a non-garden path sentence is proportional to the number of words in the sentence. Therefore, TriSer has linear time requirements for parsing non-garden path sentences.

Since TriSer uses a depth first search, the number of transitions taken for a garden path parse (i.e. a parse that requires backtracking) is at worst non-linear with respect to the number of words in the sentence, and so TriSer has non-linear time requirements for parsing garden path sentences.

4.4 The Space Requirements of TriSer

---

proportional to the number of attachments. Therefore the time required to prune 3D-trees during a parse is still proportional to the number of attachments. For TriSer, since the workspace size is limited, the number of attachments is proportional to the number of read transitions. But the number of read transitions is proportional to the total number of transitions since there can be at most seven transitions for each read transition. Therefore, the time required for pruning 3D-trees is proportional to the number of transitions, and so the time required by TriSer to build and maintain 3D-trees is proportional to the number of transitions taken. This results guarantees that at worst, the time required to take or backtrack over transitions during a parse is proportional to the number of transitions taken.

[5] For example, how the ambiguity of the exact internal structure of a subordinate verb phrase is resolved does not affect any ambiguity of the matrix verb phrase. This argument could have also been used to show that pruning a 3D-tree actually requires constant time for each pass. However, for pruning, this requirement of the grammar was not necessary to show that the sum effect of pruning is to require time proportional to the number of transitions taken.

The space required by TriSer is the amount of space required by the workspace plus the amount of space required for the control structure. The amount of space required by the workspace is the sum of the amount of space required by the 3D-trees in the workspace. But it has already been shown that the amount of space required by a 3D-tree is always proportional the number of words that it spans. Therefore, the amount of space required by the workspace is proportional to the number of words it spans. Therefore, TriSer's workspace has linear space requirements regardless of whether it is processing a non-garden path or a garden path sentence.

The control structure consists of two parts. One is the list of transitions that have already been taken to reach the current state (so it can backtrack) and the changes made to the state by each of these transitions (so it can undo these changes upon backtracking). As mentioned before, there are at most seven epsilon transitions between each read transition, so the number of transitions along the path from the starting state to the current state is proportional to the number of read transitions. But the number of read transitions is equal to the number of words spanned by the 3D-trees in the workspace, and so the number of transitions along this path is proportional to the number of words w. While the number of changes made to the state (due attaching two 3D-nodes and pruning) is at most proportional to w, similar arguments to the ones made above can be made that the amount of space to record this information for the transitions along the path to the current state is at most proportional to the number of attachments made and therefore proportional to w, the number of words spanned by the workspace. Therefore, this structure requires space proportional to w.

The other structure is the list of transitions that are possible out of the states along the path to the current state. Since there are a bounded number of transitions out of a state, the space required for each state along the path to the current state is constant. The number of states along the path to the current state is proportional to the number of transitions along the path leading from the starting state to the current state, which was just shown to be proportional to the number of words w spanned by the workspace.

Since the space requirements of the workspace and control structures are proportional to the number of words w spanned by the workspace and not the number of transitions taken, the space required by TriSer is at all times proportional to w. Therefore, TriSer has linear space requirements, regardless of whether it is processing a non-garden path or garden path sentence, and regardless of the particular type of parsing algorithm used (i.e. it only depends on the number of words spanned by the workspace).

4.5. Space and Time Requirements of a Set of 2D-trees

The problem with using sets of 2D-trees is that they allow the parser to represent the exponentially many readings which are theoretically possible by performance grammars. For example, enumerating all of the analyses possible due to PP attachment ambiguity using a set of 2D-trees can result in the number of analyses growing exponentially with the length of the sentence. The time and space requirements of such a parser is exponentially proportional to the length of the sentence.

Even if sets of 2D-trees were restricted to represent only those analyses with the same tree skeleton (like a 3D-tree), the time requirements would still be non-linear for parsing non-garden path sentences. This is because ambiguity is still multiplied out. If we assume each word introduces a constant number of ambiguous attachments (as was done above for 3D-trees), the number of 2D-trees would still grow exponentially with the number of words and so time and space requirements would grow exponentially. If we assume that each word introduces a constant number (c) of ambiguous attachments but also disambiguates the previous ambiguity (a situation more realistic in parsing), the space requirements of the parser would be linear (cw), but the time requirements would still be proportional to w^2 (i.e. non-linear) since for each word, the previous structure would have to be replicated, requiring time proportional to cw.

5. CONCLUSIONS

This paper has shown that memory architectures with unconstrained parallelism is not necessary to model certain psycholinguistic phenomena that have motivated such parallelism. Using only 3D-trees, which provide a very restricted form of parallelism, is sufficient. The benefits of using 3D-trees (in combination with a memory architecture that limits the number of and accessibility to the 3D-trees) are that the time requirements of the parser are linear for processing non-garden path sentences and the space requirements are linear for processing both non-garden path and garden path sentences. An important thing to note is that these benefits are independent of the specific parsing algorithm used. This means that parsers that use 3D-trees and which model the Human Sentence Processor have computational requirements that are practical.

REFERENCES

Abney, S. P. 1989. A computational model of human parsing. *Journal of Psycholinguistic Research* 18 (1):129-144.

Altmann, G., and M. Steedman. 1988. Interaction of context during human sentence processing. *Cognition* 30:191-238.

Church, K. W. 1982. *On Memory Limitations in Natural Language Processing*. Bloomington, Indiana: Indiana University Linguistics Club.

Frazier, L., and J. D. Fodor. 1978. The sausage machine: A new two-stage parsing model. *Cognition* 6:291-325.

Gibson, E. A. F. 1991. *A Computational Theory of Human Linguistic Processing: Memory Limitations and Processing Breakdown*. Ph.D. thesis, Carnegie Mellon University, May.

Gorrell, P. 1987. *Studies of Human Syntactic Processing: Ranked-Parallel versus Serial Models*. Ph.D. thesis, University of Connecticut, Storrs.

Jurafsky, D. 1992. *An On-line Computational Model of Human Sentence Interpretation: A Theory of the Representation and Use of Linguistic Knowledge*. Ph.D. thesis, University of California, Berkeley, March.

Kay, M. 1986. Algorithm schemata and data structures in syntactic processing. In B. J. Grosz, K. S. Jones, and B. L. Webber (Eds.), *Readings in Natural Language Processing*, 35-69. Los Altos, California: Morgan Kaufmann Publishers, Inc.

Lee, C. C. 1996. *The Human Sentence Processor: Memory Structure and Accessibility*. Ph.D. thesis, Stanford University, August.

MacDonald, M.C., M. A. Just, and P. A. Carpenter. 1992. Working memory constraints on the processing of syntactic ambiguity. *Cognitive Psychology* 24:56-98.

Marslen-Wilson, W. D. 1975. Sentence perception is an interactive parallel process. *Science* 189:226-228.

Pollard, C., and I. A. Sag. 1994. *Head-Driven Phrase Structure Grammar*. Chicago, Illinois: The University of Chicago Press.

Potter, M. C., and B. A. Faulconer. 1979. Understanding noun phrases. Journal of Verbal Learning and Verbal Behavior 18:509-521.

Swinney, D. A. 1979. Lexical access during sentence comprehension: (re)consideration of context effects. *Journal of Verbal Learning and Behavior* 18:645-659.

Tanenhaus, M.K., J. M. Leiman, and M. S. Seidenberg. 1979. Evidence for multiple stages in the processing of ambiguous words in syntactic contexts. *Journal of Verbal Learning and Verbal Behavior* 18:427-440.

Tomita, M. 1987. An efficient augmented-context-free parsing algorithm. *Computational Linguistics* 13:31-46.

Tyler, L. K., and W. Marslen-Wilson. 1982. Speech comprehension processes. In J. Mehler, E. C. Walker, and M. Garrett (Eds.), *Perspectives on Mental Representation*, chapter 9, 169-184. Hillsdale, New Jersey: Lawrence Erlbaum Associates, Inc.