Efficient Connectionist Representations of Syntactic Parse Trees for Grammatical Inference

HO, Kei Shiu Edward* CHAN, Lai Wan†

Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong

Abstract

In this paper, syntactic parsing is tackled in the context of connectionism. We present the Confluent Preorder Parser (CPP) in which syntactic parsing is achieved via a holistic transformation from the sentence representation to the desired parse tree representation. To achieve the generalization ability, these connectionist representations should be structural-preserving, that is, parse trees which look alike (structurally) in the symbolic forms should have their corresponding representations close to each other in the representational space. In this respect, the traditional convention to encode the parse tree as a hierarchical data structure is inappropriate. We show that by linearizing the parse tree via preorder traversal and applying confluent inference during training, the representations obtained are capable of preserving the structural characteristics of the parse trees they encode. Simulation results show that CPP has an excellent generalization performance of more than 90%. In addition, it is capable of correcting erroneous sentences and resolving syntactic ambiguities. We analyze CPP in two different ways. Firstly, hierarchical clustering is used to examine the parse tree representations developed. Then, an analysis is presented which elucidates the operations of CPP as governed by a finite state automata and holistic parsing is interpreted as a series of decision makings during the process. Based on this formalism, the syntactic parsing mechanism, the generalization and the error recovery capability of CPP can all be explained in terms of state transitions. In addition, we can observe a theoretical limit on the computational power of CPP in learning infinite grammars.

1 Introduction

1.1 Syntactic Parsing

In language processing, syntactic parsing addresses the problem of finding the hierarchical relationship between the terminals (or more specifically, words) in a sequential sentence. Traditionally, this relationship is manifested in the form of a tree – the parse tree. For example, by classifying the words in the sentence "the boy takes the apple on the table", we form a sequence $\langle$ D N V D N P D N $\rangle$ where the terminal D stands for determiner, N for noun, V for verb and P for preposition. Upon successful parsing of this sequence according to some grammatical rules, the parse tree in Figure 1 is produced, where np stands for noun phrase, vp for verb phrase, pp for prepositional phrase and s for sentence. Alternatively, the tree can also be represented by the Lisp-like structure ((D N) (V ((D N) (P (D N))))).

Using conventional AI approaches, the sentences and the parse trees are represented symbolically as character strings, lists, or pointer structures, etc., and the parsing problem is treated as a rational inference process which involves multiple intermediate steps and decision making (including backtracking) as governed by a well-defined algorithm. During parsing, the internal structure of the sentence is analyzed

*email: ho052@cs.cuhk.hk
†email: lwchan@cs.cuhk.hk

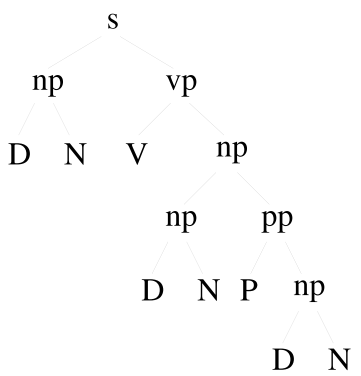


Figure 1: The Parse Tree of the Sentence ⟨ D N V D N P D N ⟩

step-by-step according to a predefined grammar, with the parse tree being built up incrementally at the same time (see Figure 2). Representative examples include the pushdown automata [13] and the chart parser [2].

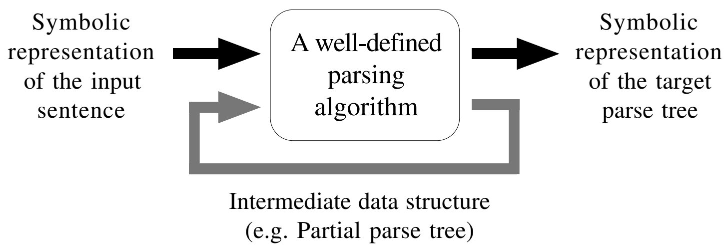


Figure 2: Syntactic parsing using symbolic techniques

1.2 Hybrid Parsers

Despite their popularity, symbolic parsing techniques have been criticized of their inadequacy in handling natural languages. In particular, the "crisp" nature of the symbolic rules simply fails to demonstrate the type of generative capability and flexibility (that is, robustness) as found in actual language usage [15]. Moreover, the formulation of the parsing algorithm requires a thorough knowledge of the characteristics of the target grammar. Unfortunately, parsing (particularly when used in natural language understanding) is among those problems of which the solution is not well defined, let alone talking about the details of its mechanism. Many of the deeply-entrenched assumptions may have little evidential background and the linguistic theory approaches are originally intended for explaining the competence of human language usage only, rather than for the purpose of efficient algorithmic realization (see also [15]).

Compared with the symbolic AI methods, connectionist approaches are inherently robust. In addition, they use inductive learning to acquire knowledge from examples and can generalize naturally to unseen sentences. To exploit these advantages, efforts have been paid to integrate the connectionist techniques into the symbolic processing framework, giving rise to various "hybrid" parsers.

For example, in the "Massively Parallel Parsing" system [21], a chart parser is used to generate all possible parses of the input sentence first. They are then translated into a connectionist network in which nodes are used to represent alternative syntactic classes and meanings of the words as well as the pragmatic constraints. Nodes leading to a consistent parse are connected via excitatory links, while nodes corresponding to mutually-exclusive parses are connected via inhibitory links. The overall interpretation of the sentence is obtained via parallel constraint satisfaction.

The Connectionist Deterministic Parser (CDP) [15] consists of a symbolic component which is a deterministic parser, and a subsymbolic component which is a feedforward network. During parsing, based on the contents of the parser's symbolic data structures (which include the stack and the lookahead buffer), the feedforward network outputs the action to perform and the parser manipulates its data structures accordingly. Effectively, the rigid rules are replaced by the feedforward network which has a more flexible decision boundary. Robustness in processing is thus achieved.

In SPEC (Subsymbolic Parser for Embedded Clauses) [19], a complex sentence is broken down into its constituent clauses by the *Segmenter* (which is a feedforward network). Each clause is parsed by the *Parser* (which is a Simple Recurrent Network) to generate the corresponding case-role representation. To deal with center-embeddings, information of the superordinate clause is held in the *Stack* (which is a RAAM network [22]) when the *Parser* has to switch to a subordinate clause. When the processing of the subordinate clause is done, this information is recovered/popped from the *Stack* and the parsing of the superordinate clause continues.

In the SCAN architecture (Symbolic/Connectionist Approach for Natural language phrases) [32], a *Plausibility Network* (which is basically a feedforward network) is trained to evaluate the "likelihood" that a syntactic relationship can hold between two nouns (or noun groups) by using the words' semantic features. A chart parser then exploits this decision/information to resolve syntactic ambiguity.

To summarize, these approaches still require the detailed specification of the parsing algorithm. In some sense, they are merely partial or complete neural network re-implementation of some symbolic algorithm. The inductive learning power of neural networks simply has not been fully utilized. What we target for is a system that exhibits a style of *Extensional Programming* as discussed in [15] – with access to the input and the output of the parsing process only, but with minimal knowledge of the actual algorithm, the system can induce the transformation/mapping from the sentence to the parse tree which effects syntactic parsing.

1.3 Holistic Parsing by the Confluent Preorder Parser

In this paper, syntactic parsing is discussed in the context of pure connectionism. The **Confluent Preorder Parser (CPP)** will be presented which exemplifies the **Holistic Parsing Paradigm**. Instead of a step-by-step algorithmic approach, syntactic parsing is accomplished via a **Holistic Transformation** [4, 5]. This transformation maps the connectionist representation encoding the input sentence to the connectionist representation encoding the target parse tree (see Figure 3). Effectively, the parsing mechanism is encapsulated in a "black box". With the knowledge of the ultimate input (the sentence) and the ultimate output (the parse tree) of the parsing process only, CPP learns inductively to acquire the necessary grammatical regularity underlying the training examples. This has the advantage that little knowledge has to be assumed for the actual, detailed parsing algorithm.

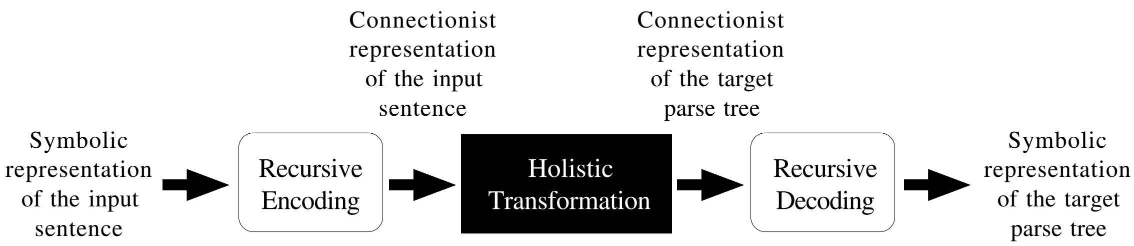


Figure 3: Holistic parsing paradigm

Compared with other grammatical inference approaches, CPP is unique in several ways :

- Syntactic parsing as addressed by CPP is significantly more difficult than the "classification problem" tackled by [8, 23, 31], and the "prediction problem" attempted by [7, 18, 26].

  Firstly, CPP aims at tackling context-free grammars instead of regular languages. In general, to parse context-free languages, external memory is required [13]. In contrast to other approaches (such as SPEC [19] and the Neural Network Pushdown Automata [28]), CPP is not equipped with a stack.

Secondly, in the classification problem, a single bit suffices to represent whether a string can be derived by the language or not. Similarly, for the prediction problem, it is only necessary to devise a unique representation for each terminal in the alphabet. For a regular language, the number of terminals is finite and often small. Therefore, a localist representation scheme will do in both cases. However, the data structures that CPP has to deal with, that is, the parse trees and the sentences, are far more complicated. They are structured, recursive and possibly infinite (since there is no prescribed maximum length for the sentences or maximum height for the parse trees). Thus to encode these data structures using fixed-length coding, distributed representations must be used and they have to be developed using functional composition techniques [29] (such as RAAM [22]). Needless to say, the overall performance of the parser will be intimately related to these representations. Unfortunately, encoding a recursive symbolic data structure, especially a hierarchical one (such as the parse tree), is by all means "hard" for neural networks. This thus has posed a major challenge.

- As revealed by the experiment to be presented in Section 4 (in which the length of the longest sentence used is 17), excellent generalization performance has been achieved by CPP (more than 90%). This is superior when compared with previous holistic parsing approaches. For example, in the Backpropagation Parsing Network [11], the generalization performance is 40% only and for the parser designed by Reily [24], the generalization performance is as low as 0%.
- CPP is also capable of parsing erroneous sentences and resolving syntactic ambiguities. This characteristic is especially appealing in natural language processing when incomplete or ungrammatical sentences have to be dealt with frequently. Using the symbolic AI techniques, it is necessary to anticipate the types of exceptional conditions that the parser may come across and a special recovery mechanism is then tailored and incorporated in the parser. Often, some of the grammar rules have to be re-designed at the same time. The approach is thus costly, "ad-hoc" and inflexible. In CPP, robustness in processing is achieved without an extra cost – neither is it necessary to teach CPP to parse negative training examples (that is, erroneous or ambiguous sentences), nor an explicit mechanism of any kind has to be installed (e.g. error production [1, 17]).

2 Structural-Preserving Representations

As mentioned in Section 1, distributed representations have to be developed for the sentences and the parse trees. In CPP, these representations are "learned" via training a neural network, rather than "designed" or "calculated" as in the Holographic Reduced Representations HRR [20]. Since neural networks generalize by interpolating and/or extrapolating between a novel pattern and the training patterns, the representations developed should be **Structural-Preserving** – their surface forms should reflect systematically the structural characteristics of the data structures they encode, such as the constituent components that are present as well as their relationship and alignment in the structure (see also [20]). Precisely, the coding corresponding to two similar sentences should have a short Euclidean distance in the connectionist representational space. In much the same way, the representations encoding two structurally similar parse trees should also be close to each other in the Euclidean space.

In this respect, the traditional convention that a parse tree should be represented as a hierarchical data structure (see [3, 11, 24]) has been found inappropriate. The structure of the parse tree is only implicitly reflected by the order and manner in which different components and terminals of it are combined together to give the fixed length representation. The coding thus obtained simply cannot facilitate the type of generalization needed in connectionist syntactic parsing.

In view of this, the parse tree is no longer represented as a hierarchical data structure in CPP. Instead, it is linearized[1] by preorder traversal to form a sequence and encoded using a SRAAM network

[1] Linearization has first been applied by Kwasny and Kalman [16] to facilitate the encoding of trees using a SRAAM network. Their approach and ours differ in the way of expressing the empty branches. In [16], null pointers (as the special terminal '*') are explicitly indicated for the leaves and those internal nodes with empty branch(es). These '*'s

(Sequential Recursive Auto-Associative Memory [22]). For example, the parse tree in Figure 1 gives rise to the sequence ⟨ s np D N vp V np np D N pp P np D N ⟩. CPP thus produces the preorder traversal sequence first, from which the target parse tree can be reconstructed[2]. With linearization, the two types of sequences – the sentences and the preorder traversals, are encoded by two SRAAM networks respectively, that is, SRAAM_1 and SRAAM_2 in Figure 4.

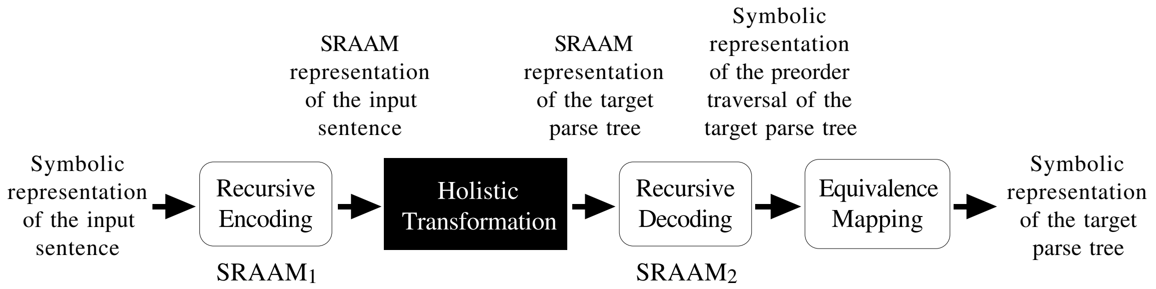


Figure 4: Holistic parsing with parse trees linearized by preorder traversal

By representing the internal nodes (e.g. np) also in the preorder traversal, the structure of the parse tree can be made explicit. Thus similar sentences having the same branch in the parse trees share the same sub-sequence in their preorder traversals. When the preorder traversal sequences are encoded using the SRAAM network, their similarity will be manifested in the coding developed, thus preserving the structural characteristics of their corresponding parse trees. This is confirmed by the experimental results shown in Section 5.

To further promote the generalization performance, confluent inference [6] is applied in training. This technique, instead of performing a mapping from one representational space to another one, develops the same SRAAM representation for the input sentence and the parse tree. Thus, the Holistic Transformation in Figure 4 is implemented as an identity mapping (which in fact, is effectively removed as a result).

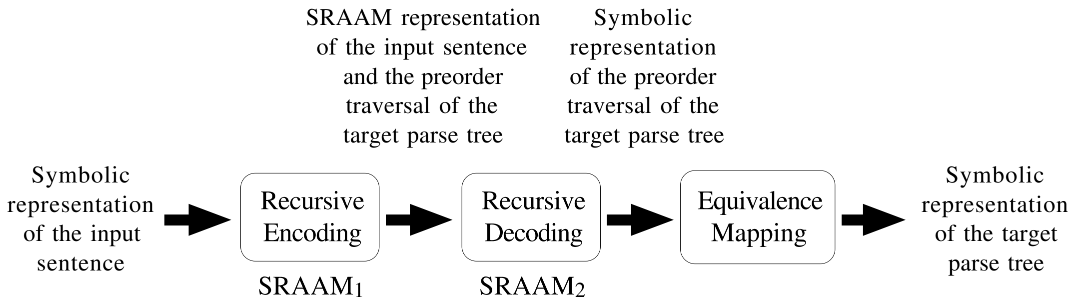


Figure 5: Holistic parsing by CPP

With confluent inference, two advantages are evident :

- Firstly, it eliminates an explicit transformation mechanism/function from one type of sequence representation (the sentence) to another (the preorder traversal). This transformation is an additional step linking two independent representations. It may corrupt the structural properties of the representations.

appear in the preorder traversals also. For example, the parse tree in Figure 1 now corresponds to the preorder traversal ⟨ s np D * * N * * vp V * * np np D * * N * * pp P * * np D * * N * * ⟩. In our approach, we do not explicitly include the null pointers. Obviously, CPP in general gives rise to a shorter preorder traversal sequence.

[2] To guarantee the equivalence mapping between a parse tree and its preorder traversal, each parse tree must be made "regular" in the sense that every internal node of it has the same number of children (that is, the tree is of fixed valence). Otherwise, we are unable to define a parse tree uniquely by using the preorder traversal alone. In case there is an empty branch in an internal node, a special terminal '@' is inserted as a dummy leaf, which is enumerated explicitly in the preorder traversal.

- Secondly, as a result of using confluent inference, the encoding mechanism and the decoding mechanism in Figure 5 are trained simultaneously and coherently (see Section 3.1 for a detailed discussion). In this way, the SRAAM coding that are developed take into account the characteristics of the sentences and the preorder traversals as well as the parsing function. It can therefore be expected that they are better adapted to the syntactic parsing problem.

3 Confluent Preorder Parser (CPP)

3.1 The Training Methodology

Using the linearization and confluent inference techniques discussed in the previous section, we design the Confluent Preorder Parser (CPP) as shown in Figure 6. It combines $SRAAM_1$ (the *Sentence Encoder*) and $SRAAM_2$ (the *Preorder Traversal Encoder*) in Figure 5 to form a dual-ported SRAAM [6], with one single hidden layer shared between the two SRAAMs. The equivalence mapping from the preorder traversal sequence to the parse tree is done separately.

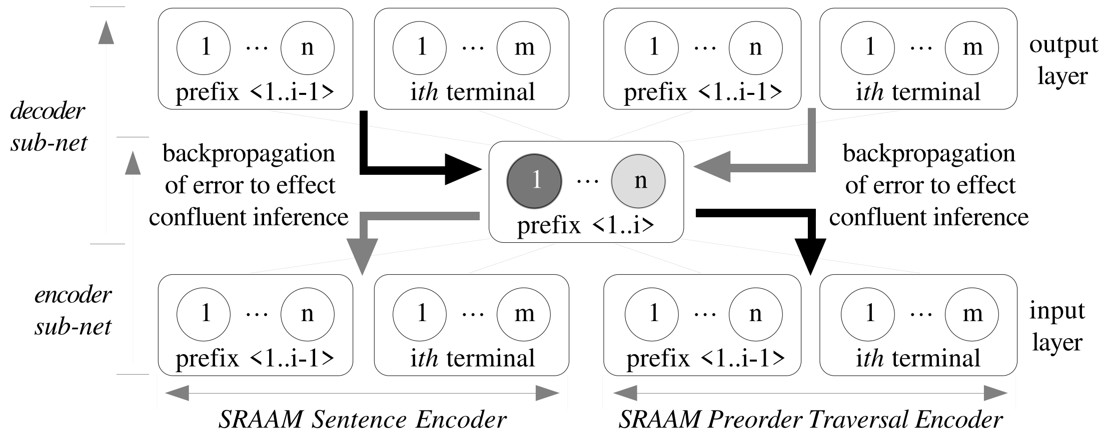
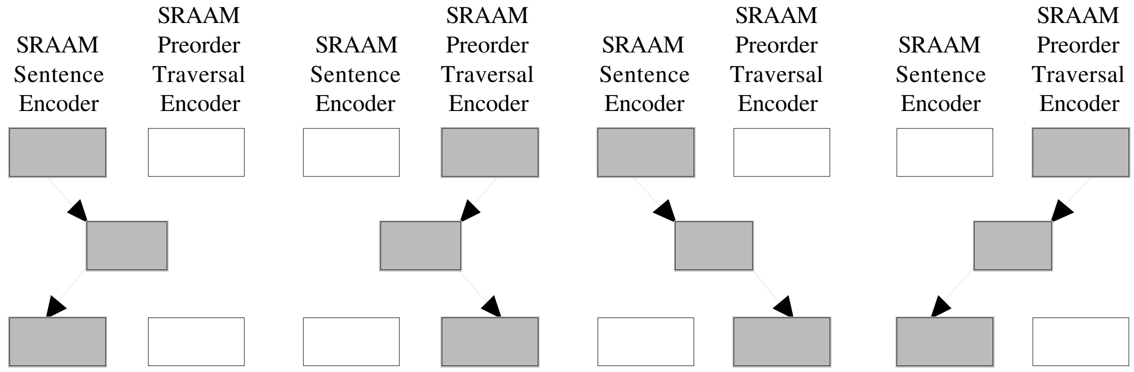


Figure 6: Confluent Preorder Parser (CPP)

Given a set of training sentences, we first generate the preorder traversals for their corresponding parse trees. The training sentences as sequences are then encoded by the *Sentence Encoder* on the left half of CPP (see Figure 7(a)), while the preorder traversals are encoded by the *Preorder Traversal Encoder* on the right half (see Figure 7(b)).

As explained in Section 2, confluent inference [6] is applied. In addition to the standard training method of SRAAM [22] (Figure 7(a) and 7(b)), an extra error term is backpropagated during training from the output layer of the *Preorder Traversal Encoder* to the input layer of the *Sentence Encoder*, and also from the output layer of the *Sentence Encoder* to the input layer of the *Preorder Traversal Encoder*, as illustrated in Figure 7(c) and Figure 7(d) respectively. In this way, the SRAAM representation of a sentence (as encoded by the *Sentence Encoder*) will give rise to the corresponding preorder traversal upon recursive decoding by the decoder sub-net of the *Preorder Traversal Encoder*. Similarly, when the SRAAM representation of a preorder traversal sequence (as encoded by the *Preorder Traversal Encoder*) is decoded by the decoder sub-net of the *Sentence Encoder*, the original sentence is obtained. When training succeeds, identical SRAAM representation is developed for each complete sentence (such as ⟨ D N V D N P D N ⟩) and the preorder traversal of its respective parse tree (correspondingly ⟨ s np D N vp V np np D N pp P np D N ⟩).

With confluent inference, syntactic parsing proceeds as follows. Treating a sentence as a sequence of terminals (e.g. ⟨ D N V D N P D N ⟩), its SRAAM representation is first derived by encoding it recursively from left to right using the encoder sub-net of the *Sentence Encoder*. After the whole sentence has been scanned, the SRAAM coding obtained which represents both the sentence and its respective



(a) Encoding sentences (b) Encoding preoder traversal sequences (c) Confluent inference from *Preorder Traversal Encoder* (d) Confluent inference from *Sentence Encoder*

Figure 7: Backpropagation of errors during the training of CPP. Different parts (shaded) of CPP are adapted for different types of training patterns. The arrows show the backpropagation of error during training. (a) and (b) are the backpropagation of error when the *Sentence Encoder* and the *Preorder Traversal Encoder* are trained separately. (c) and (d) are the backpropagation of the extra error term to implement confluent inference.

preorder traversal is fed directly to the decoder sub-net of the *Preorder Traversal Encoder*. Through recursive decoding, the output ⟨ s np D N vp V np np D N pp P np D N ⟩ is the preorder traversal of the parse tree. Exploiting the equivalence mapping between this preorder traversal and the parse tree it is derived from, the latter can be obtained.

3.2 Learning to Parse Phrases

In its original design, CPP uses complete sentences as the training examples only. No explicit syntactic groupings of the terminals are defined (e.g. np, vp, etc). This agrees with the basic philosophy of the holistic parsing paradigm that the minimum grammatical knowledge should be assumed. CPP has access to the ultimate input (the sentence) and the ultimate output (the parse tree) of parsing only. Relaxing this constraint slightly, phrases, in addition to complete sentences, can be fed to CPP as training examples also. Phrases are generated using internal nodes other than s as the root (e.g. np). Take for instance, the sentence s = ⟨ D N V D N ⟩ is in fact made up of two phrases – a noun phrase np = ⟨ D N ⟩ and a verb phrase vp = ⟨ V D N ⟩. They correspond to the sub-trees (D N) and (V (D N)), with preorder traversals ⟨ np D N ⟩ and ⟨ vp V np D N ⟩ respectively. To acquire this knowledge, CPP has to learn the association between the SRAAM representation of np = ⟨ D N ⟩ and that of ⟨ np D N ⟩, and also the association between the SRAAM representation of vp = ⟨ V D N ⟩ and that of ⟨ vp V np D N ⟩. Specifically, an identical representation has to be developed for np = ⟨ D N ⟩ and ⟨ np D N ⟩, as well as for vp = ⟨ V D N ⟩ and ⟨ vp V np D N ⟩.

As revealed by our simulations to be reported in Section 4.1, the generalization performance of CPP can be significantly promoted with this simple modification, at the expense of using more training patterns (given the same number of complete training sentences).

3.3 Implementation of CPP

CPP is implemented on two platforms. One is a sequential implementation on a SPARC-20 computer. For the experiment in Section 4.1, one single iteration (which accounts for a complete sweep of 864 training patterns) takes more than 12 minutes to complete. Another is a parallel version of CPP on the DECmpp computer which has an SIMD architecture. Tens of folds in speedup is achieved. The parallel

CPP takes less than 30 seconds to finish one epoch and totally, it needs about 2,000 cycles before training converges. See [12] for a detailed discussion of this parallel CPP.

4 Experiment

4.1 Generalization Performance of CPP

To evaluate the performance of CPP, we use the following context-free grammar as in [22] :

Sentence	Noun Phrase	Verb Phrase	Prepositional Phrase	Adjectival Phrase
s $\rightarrow$ np vp s $\rightarrow$ np V	np $\rightarrow$ D ap np $\rightarrow$ D N np $\rightarrow$ np pp	vp $\rightarrow$ V np vp $\rightarrow$ V pp	pp $\rightarrow$ P np	ap $\rightarrow$ A ap ap $\rightarrow$ A N

Each syntactically well-formed sentence (and its parse tree) is generated using s as the root. We apply the above grammar rules recursively to generate trees having 5 levels at most. A total of 112 sentences are obtained (Figure 8 shows the distribution of sentences of different lengths). Among them, 80 sentences are randomly selected for training while the remaining 32 sentences are reserved for the testing phrase.

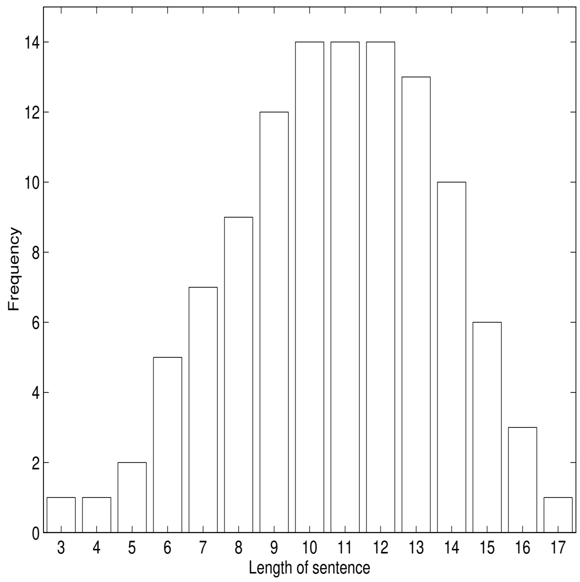


Figure 8: The distribution of sentences of different lengths which are used in the experiment. Totally, there are 112 sentences.

The representations of the terminals (including the internal nodes) are shown in Table 1. Orthogonal binary strings are used to minimize the mutual interference between the coding. The error injection method proposed in [27] is adopted – an extra bit is attached as the leftmost bit of each coding which is initialized to 0 for a terminal while for a non-terminal (a SRAAM coding), its value is forced to 1 during training.

Two CPPs were evaluated. For CPP1, the original design is realized and the parser has to learn to process complete sentences only. For CPP2, in addition to complete sentences, the network is also trained to parse phrases to produce the corresponding sub-trees (see Section 3.2). In either case, the network is trained using the back-propagation algorithm [25] with fixed rate and momentum until all sentences in the training set can be successfully parsed. Generalization capability is then measured by evaluating its performance on the testing set. Summarized in Table 2 are the training parameters and

Table 1: The representations of the terminals

	length = 23 units
N	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
V	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0
P	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0
D	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0
A	0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0
s	0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0
np	0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0
vp	0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
pp	0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ap	0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Table 2: The training parameters and the results of the experiment. The symbols n and m (the lengths of the representations) are interpreted as in Figure 6.

	Length of representations		Training parameters		Successful parse %	
	SRAAM	Terminal	Rate	Momentum	Training	Testing
CPP1	n = 220	m = 23	0.00225	0.9	100%	81.25%
CPP2	n = 230	m = 23	0.00225	0.9	100%	93.75%

the results. With the concept of phrases, CPP2 has a better understanding of the internal structure of a sentence and its parse tree and it outperforms CPP1 in generalization.

4.2 The Need to Integrate Linearization and Confluent Inference

The techniques of linearization and confluent inference are complementary to each other and omitting any one of them will significantly sacrifice the total performance of CPP. To demonstrate this, a control experiment has been conducted. Three prototype parsers were constructed. They are derivatives of CPP but omitting one or both of the two techniques (see Table 3). They were trained separately on parsing the same set of 80 complete sentences used in Section 4.1 :

- Prototype P1 has the same architecture as the Backpropagation Parsing Network (BPN) proposed in [11]. Linearization is not adopted. The parse trees as hierarchical data structures are encoded by a RAAM network, while the sentences as sequences are represented using a SRAAM network. No confluent inference is applied and the two networks are trained separately and independently, until both types of data structures can be perfectly encoded (and decoded). Syntactic parsing is then effected by transforming holistically from the SRAAM representation of a complete sentence to the RAAM representation of the desired parse tree via a 3-layered feedforward network (with one hidden layer of 75 units).
- In Prototype P2, linearization is excluded. As in P1, the hierarchical parse trees and the sentences are encoded using a RAAM network and a SRAAM network respectively. In contrast to the Prototype P1, confluent inference is applied. Extra backpropagation terms are introduced such that identical representation is developed for both the complete sentence and its corresponding parse tree. During parsing, an input sentence is encoded by the SRAAM network and the coding obtained is then decoded by the RAAM network to give its corresponding parse tree. Thus effectively, Prototype P2 is adapted from CPP by simply replacing its SRAAM Preorder Traversal Encoder by a RAAM network which serves as the Parse Tree Encoder.
- In Prototype P3, linearization is adopted but confluent inference is not included. Similar to CPP, two separate SRAAM networks are trained to encode the sentence sequences and the preorder

Table 3: The results of a comparative study showing the importance of integrating the two techniques of linearization and confluent inference

		P1	P2	P3	CPP1	CPP2
Techniques used	Confluent Inference	NO	YES	NO	YES	YES
	Linearization	NO	NO	YES	YES	YES
	Learn Parsing Phrases Also	NO	NO	NO	NO	YES
Length of Coding	Sentence Sequences (SRAAM)	100	125	100	220	230
	Preorder Traversals (SRAAM)	N/A	N/A	230	220	230
	Parse Trees (RAAM)	61	125	N/A	N/A	N/A
	Terminals	23 (SRAAM) 61 (RAAM)	125	23	23	23
Successful Parse %	Training	100%	100%	100%	100%	100%
	Testing	18.75%	28.125%	59.375%	81.25%	93.75%

traversals of the parse trees respectively. However, no confluent inference is applied and the two types of SRAAM representations are developed independently. Training persists until all training sequences of both types are successfully encoded (and decoded) by the respective SRAAM network. The holistic parsing function as depicted in Figure 3 is implemented by training a 3-layered feedforward network (with one hidden layer of 200 units) which acts as an explicit transformation to map from the SRAAM representation of a complete sentence to the SRAAM representation of the preorder traversal of the corresponding parse tree.

Upon successful training, the generalization performances of these three prototypes were evaluated using the same set of 32 testing sentences in Section 4.1. The results are then compared against those obtained by CPP1 and CPP2, as summarized in Table 3. Also shown are the lengths of the coding used to represent the various types of data structures. They are in fact determined by trial-and-error. For each model, different lengths of representations were tried and the optimal result (that is, best generalization) is shown in Table 3. As revealed by the results, linearization plays a more effective role in promoting the generalization performance. However, the best performance can only be obtained by integrating the two techniques.

4.3 Parsing Erroneous Sentences

CPP, as a connectionist model, exhibits robust performance when presented with incomplete or ungrammatical sentences. To demonstrate this error recovery capability, "error" is injected systematically to each of the 80 training sentences used in Section 4.1, giving rise to another set ERROR of erroneous sentences, which are then parsed by the two trained CPPs. Four types of errors have been considered – with one wrong terminal (SW), with one missing terminal (SM), with one extraneous terminal (SE) and with two neighbouring terminals swapped (SS). For each $S_E \in$ERROR, the error is said to have been corrected if (a) a syntactically well-formed parse tree T is generated such that its corresponding sentence S differs in length from S_E by at most one, and (b) S and S_E have no more than d terminals mismatched, where d is a predetermined parameter, which is usually equal to one or two.

The results of the error recovery ability of these four types of errors are summarized in Table 4. As shown, CPP is most proficient in handling erroneous sentences of type SW and SM. In natural language processing, these two types of errors correspond (in some sense) to the situations when words are wrongly spelled and there are missing words (especially the determiner such as "the") respectively. To better understand this error recovery capability of CPP, two factors which may affect its efficiency have been evaluated, namely, the "Location of the Error" and the "Length of the Erroneous Sentence". See [10] and [9] for a detailed discussion.

[3] We should account for the case when S_E is corrected by bringing its two swapped elements back to their original order. The number of mismatches in that case will then be 2.

Table 4: Error recovery capability of CPP – % of erroneous sentences which can be successfully parsed

		Maximum mismatches d = 1		Maximum mismatches d = 2	
Error type	Total	CPP1	CPP2	CPP1	CPP2
SW	853	90.15%	94.61%	91.21%	94.72%
SM	853	54.51%	54.63%	69.64%	70.93%
SE	933	32.79%	33.44%	46.62%	50.80%
SS	773	N/A[3]	N/A	63.65%	66.88%

Table 5: Syntactic disambiguation by CPP – % of ambiguities successfully resolved

		1 ambiguity		2 ambiguities		3 ambiguities	
	Total	CPP1	CPP2	CPP1	CPP2	CPP1	CPP2
2 alternatives	853	98.24%	99.65%	95.90%	98.36%	91.89%	97.31%
3 alternatives	4395	96.60%	98.83%	89.19%	95.31%	72.70%	91.37%
5 alternatives	14545	94.26%	96.13%	71.26%	90.17%	40.18%	77.34%

4.4 Syntactic Disambiguation

The syntactic category of the word "duck" is different for its two occurrences in "they eat a duck" and "they duck their heads". Syntactic ambiguity of this type can be resolved by considering the syntactic categories of those neighbouring words collocated with the ambiguous word. CPP exhibits similar capability. To show this, ambiguities are introduced to the 80 training sentences. For every training sentence $S_k = \langle s_1\ s_2\ \ldots\ s_p \rangle$, we produce a set of ambiguous sentences $S_{k,i}$ where i=1..p. Each sentence $S_{k,i}$ is constructed by replacing the ith terminal s_i in S_k by a new terminal s_i/t where t is a terminal randomly selected from the set {D, N, V, A, P} with $s_i \neq t$. This s_i/t thus represents the dual identity (or dual syntactic category) of an ambiguous word appearing as the ith terminal of the sentence S_k. For example, if $S_k = \langle$ D N V D N $\rangle$ and let t=V, an ambiguous sentence $S_{k,2} = \langle$ D N/V V D N $\rangle$ is produced. All these ambiguous sentences are then processed by the two trained CPPs. The representation of an ambiguous terminal such as N/V is obtained by OR-ing bit-by-bit, the respective representations of N and V. Now, the parse tree T obtained should should have its preorder traversal matches exactly with the original unambiguous sentence. In other words, each s_i/t should be resolved to give the desired syntactic interpretation s_i.

Furthermore, we have also considered sentences with 1, 2 and 3 ambiguous terminals respectively, and each ambiguity consists of 2, 3 and 5 alternatives. In the case of having 5 alternatives, the problem is effectively reduced to having a terminal of unknown class. The results of syntactic disambiguation are summarized in Table 5.

5 Analysis of Representations

We discussed in Section 2 the importance of structural-preservation in promoting the overall generalization of a holistic parser. Linearization of parse trees and confluent inference are suggested in training for this respect (see also Section 3.1). In this section, we examine the representations obtained in the experiments of Section 4.1 and 4.2 (that is, CPP1 and Prototype P1) to verify that these representations do preserve the structural characteristics of the data structures they encode.

Because the representations developed are of very high dimension (over 200), techniques must be applied to project them onto a lower-dimensional space for easy visualization. Two methods have been used. Firstly, hierarchical clustering [14] is applied to the final representations of the 80 parse trees obtained in training. For Prototype P1, local clustering does occur occassionally (see Figure 9). However, no regular and systematic grouping can be observed in the global sense, regardless of whether prefix or suffix is considered. In contrast, for CPP1, parse trees corresponding to sentences which share similar suffix have their respective representations clustered together, as shown in Figure 10. By inspecting Figure 10, roughly eight groups of data have been formed, with sentences in each group share a particular suffix (correspondingly, their parse trees consist of the same sub-tree).

To further visualize the topological relationship among the parse tree representations, they are input to a 15×15 Self-Organizing Map. As shown in Figure 11(a), parse trees that are similar in their symbolic forms (consisting of the same sub-tree) are mapped to nearby neurons in the Kohonen Map, which implies that their corresponding SRAAM coding are close to each other in the Euclidean representational space. This result is consistent with that obtained by hierarchical clustering. But for Prototype P1, the result of a Kohonen Map trained in a similar way (Figure 11(b)) shows that no clear-cut boundary can be drawn to separate different clusters of parse trees.

From a global point of view, the sentences and the parse trees are mapped via the encoding function onto points of a high-dimensional representational space (constituted by the hidden units). The degree of structural similarity between two parse trees or sentences is translated into the Euclidean distance between their corresponding points in the space. Thus a distribution is resulted. In CPP, this distribution has created in the representational space a nested, hierarchical and more importantly, regular structure (see Figure 10). The sub-division of the space is syntactical, since the representations are developed as a result of performing the syntactic parsing function[4]. At the top-most level, the representational space is first divided into two clusters – one consists of points that correspond to trees/sentences which end with a "V", and the other consists of points that correspond to trees/sentences which end with a phrase of some kind (which acts as the complement of "V"). The latter cluster then further sub-divides according to the exact form of the phrase. Within a cluster where all points correspond to trees of identical right-branch (as headed by "V"), partitioning continues according to the difference in the left-branch. In this way, a nested structure is observed in which the further the sub-division goes, the finer details of the structure of the parse trees that count. A representational space resembling the *Conceptual Space* as put forth by van Gelder in [30] has thus been successfully realized by CPP, which is not achievable by Prototype P1 (c.f. Figure 9).

6 Holistic Parsing as a Step-by-Step Rational Inference

Holistic parsing as exemplified by CPP effectively encapsulates its underlying parsing mechanism in a "black box". It has the advantage that the necessary grammatical knowledge can be inferred via inductive learning. Instead of specifying "how to achieve", we simply specify "what to achieve". However, the approach has, at the same time, inhibited an easy understanding of how syntactic parsing is actually accomplished by CPP. In this section, we perceive CPP as a finite state automata. We analyze the characteristics of this finite state automata which are intimately related to the performance of CPP. This study can on one hand serve as an illustration that knowledge has really been abstracted by CPP. On the other hand, it can also shed light on how the performance of CPP can be further enhanced.

6.1 A Finite State Automata Model of Syntactic Parsing

From the network dynamics' point of view, CPP is an auto-associative recurrent network. During the recursive encoding of a sentence using the SRAAM Sentence Encoder (see Figure 6), the hidden layer activation of CPP is evolving as each terminal of the sentence is being read in. In some sense, it can be thought of manifesting CPP's temporary interpretation of the partial input sentence that has been seen so far. In each time-step, the activation pattern (much like a preliminary decision) is fed back to the input layer. This feedback, together with the next terminal of the sentence (which plays the role as a piece of new information), forms the next hidden layer activation (which represents a updated view of the sentence/problem at hand). Only after the whole sentence is scanned will the final hidden layer activation pattern be sent to the decoder sub-net. In Section 5, we analyzed the final representations of the complete parse trees (which are also the same as the corresponding sentences' representations). It is also worth examining the intermediate activation patterns to see if some form of rational inference or knowledge resolution is carrying out during the parsing process.

[4] By applying confluent inference in training, it can be expected that the distribution of the points in the representational space will be optimized for parsing.

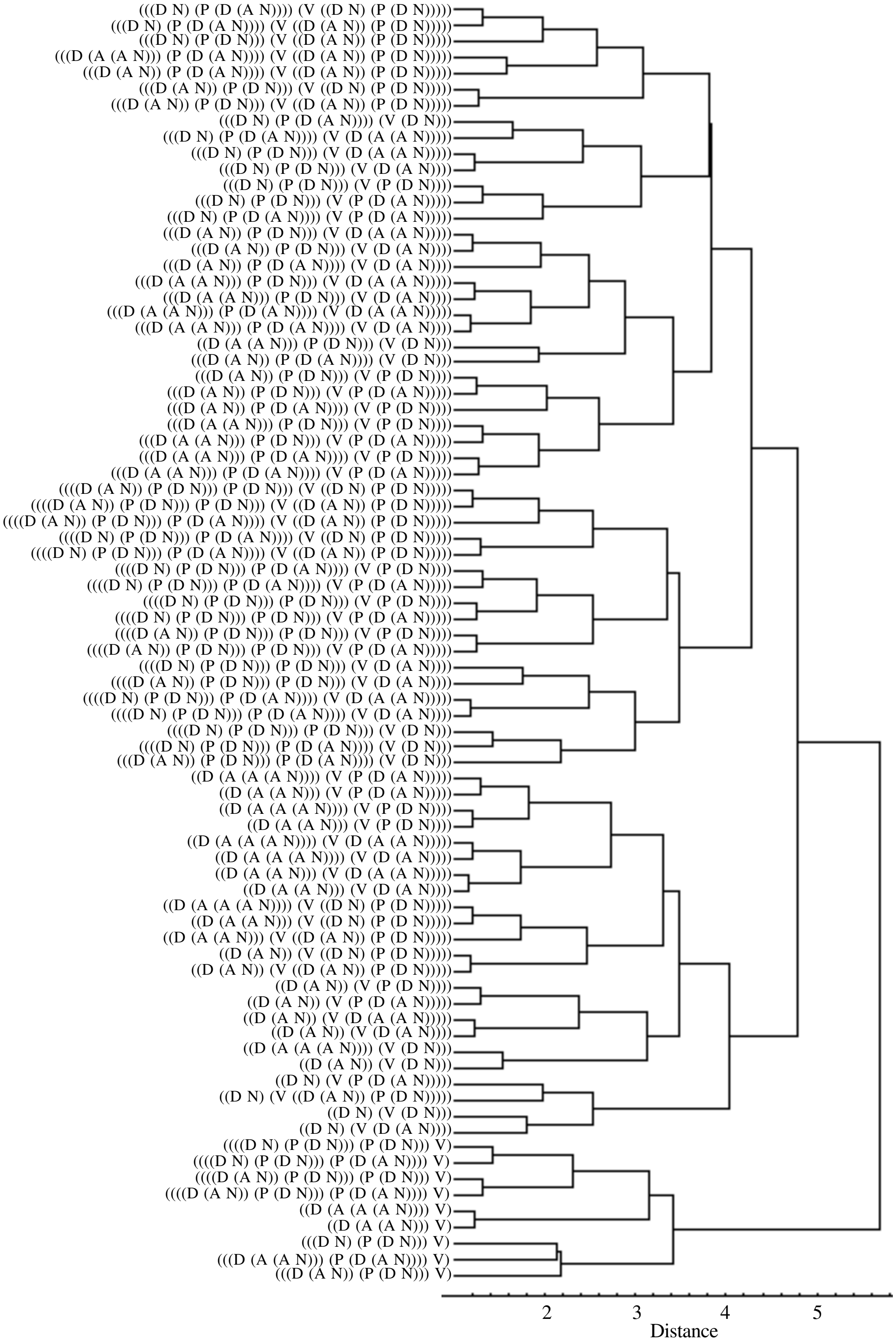


Figure 9: Hierarchical clustering of the parse tree representations obtained by Prototype P1

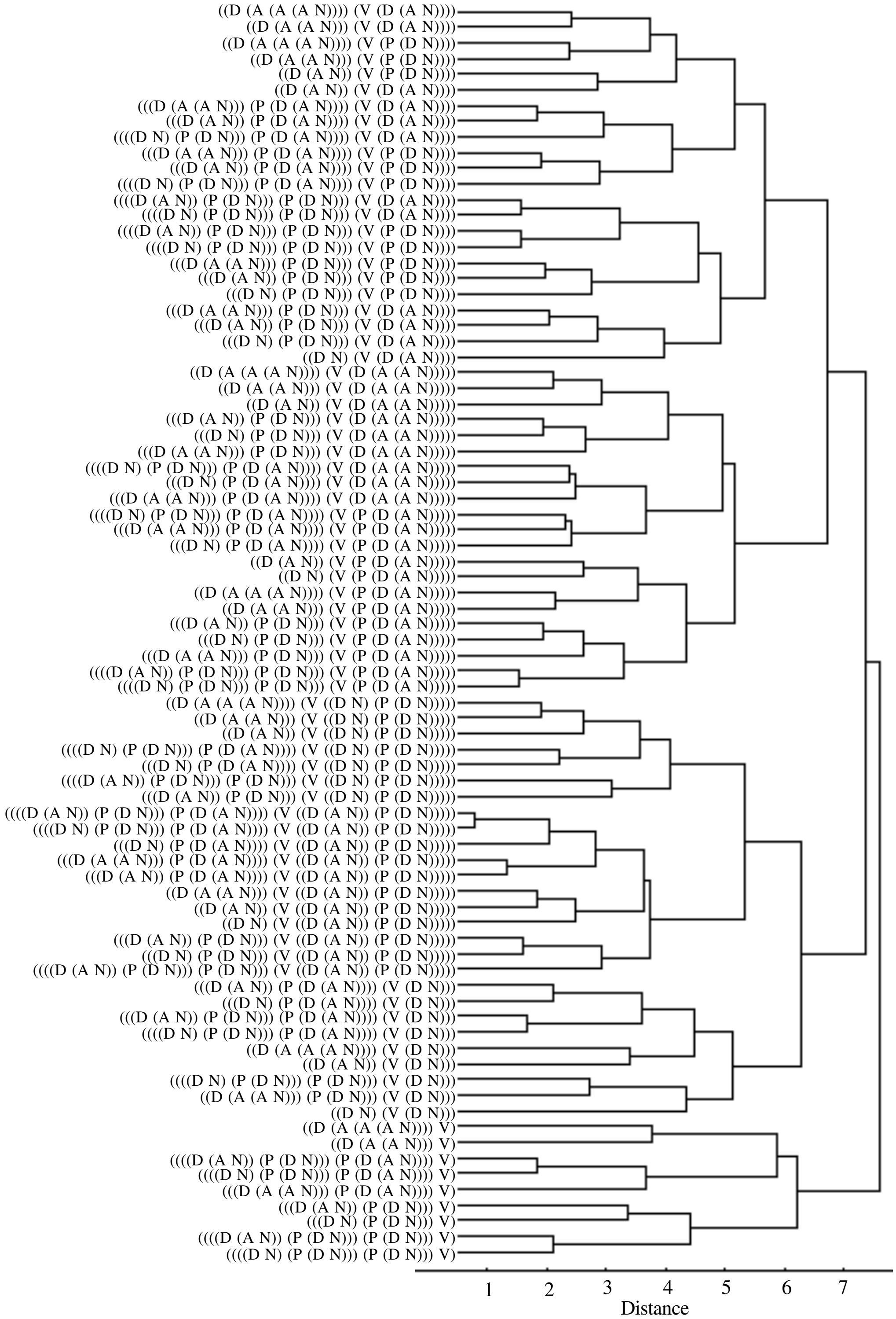
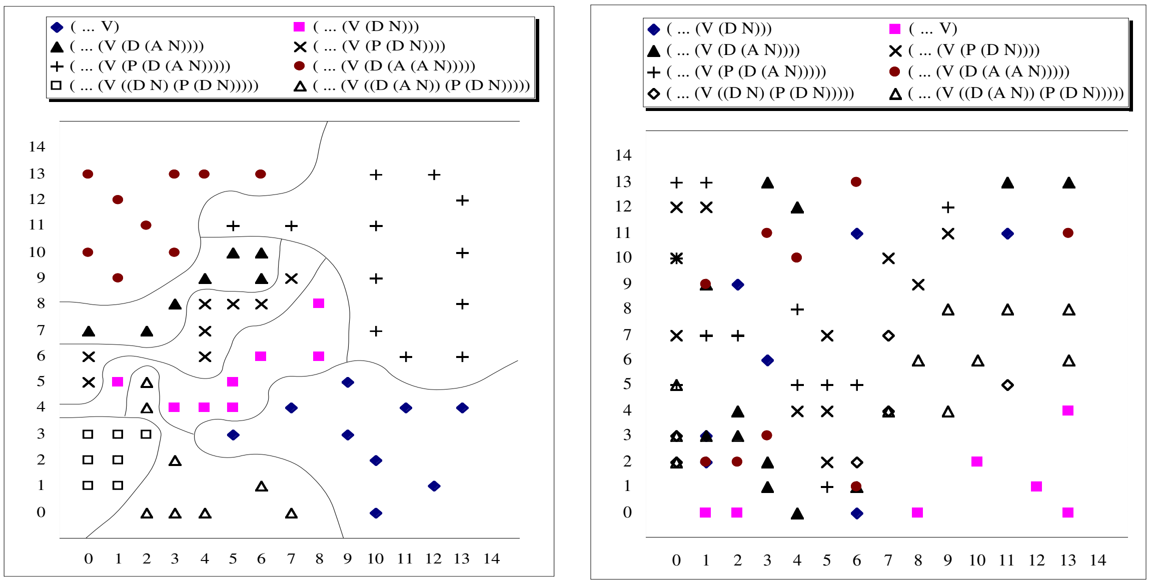


Figure 10: Hierarchical clustering of the parse tree representations obtained by CPP1



(a) Parse tree representations developed by CPP1 (b) Parse tree representations developed by P1

Figure 11: The analysis of the parse tree representations developed by CPP1 and P1 by using a 15×15 Kohonen Map. Each symbol locates the most responding neuron. In CPP1 (a), topological ordering is observed and artificial boundaries are drawn to separate different clusters of sub-trees for easy visualization. As shown, roughly 8 clusters can be identified. Each of them consists of parse trees which share some identical sub-tree, as labelled in the legend. In P1 (b), no topological ordering is observed.

Table 6: The number of states and transitions that were identified when the 80 training sentences were parsed by CPP1 and CPP2

	CPP1	CPP2
Total Number of Transitions	234	234
Starting State (#)	1	1
Intermediate States	89	77
Final States	92	106
Total Number of States	182	184

To achieve this, the two trained CPPs were tested on parsing each of the 80 training sentences again. In addition to the decoding of the SRAAM representation of a complete sentence, the intermediate hidden layer activation after reading each terminal (which represents a sentence prefix) was decoded also. Intuitively, this activation pattern can be perceived as the state of the parser. The identities of the states are defined by the decoding's results (which are sequences of terminals that may or may not represent meaningful preorder traversals of parse trees). When two states give the same result upon decoding, they are treated as the same state. A state is said to be a final state if it is decoded to give a syntactically well-formed parse tree or sub-tree. In this way, 182 states are identified for CPP1 which include 92 final states, 89 intermediate states and one starting state. For CPP2, there are 106 final states, 77 intermediate states and one starting state, giving a total of 184 states (see Table 6).

These states, together with the possible transitions between them, effectively define a finite state automata [13] which encodes the grammatical knowledge/regularity necessary for the parsing function. Syntactic parsing by CPP can therefore be re-formulated as follows. Starting from the null context # (that is, the starting state), the activation pattern changes as each input terminal is read and the parser can be considered as changing from one state to another during parsing (that is, state transitions). After scanning an entire training sentence, the parser should reside in a final state. Thus intuitively, the holistic parsing process of a sentence can be analyzed as a rational inference procedure governed by a finite state

Table 7: The identities of the states appearing in Figure 12, where '...' represents a sequence or sub-sequence that cannot be interpreted to give a well-formed tree or sub-tree

(a) CPP1

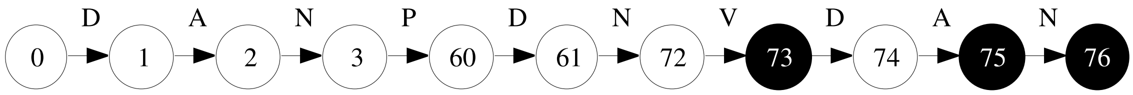
State	Decoded Sequence (Parse Tree)
0	the starting state
1	...
2	...
3	(D (A N)) ...
60	((D (A N)) V) ...
61	...
72	((D (A N)) (P (D N))) ...
73	(((D (A N)) (P (D N))) V)
74	...
75	(((D (A N)) (P (D N))) (V (D N)))
76	(((D (A N)) (P (D N))) (V (D (A N))))

(b) CPP2

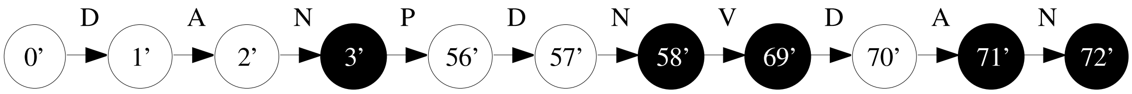
State	Decoded Sequence (Parse Tree)
0'	the starting state
1'	...
2'	(D N)
3'	(D (A N))
56'	((D (A N)) V) ...
57'	...
58'	((D (A N)) (P (D N)))
69'	(((D (A N)) (P (D N))) V)
70'	((D (A N)) (P (D (A N)))) ...
71'	(((D (A N)) (P (D N))) (V (D N)))
72'	(((D (A N)) (P (D N))) (V (D (A N))))

automata, with the intermediate decisions reflected by the outputs of the states "visited" by the parser during the processing of the sentence.

For example, the state transition sequence resulted in parsing the sentence ⟨ D A N P D N V D A N ⟩ by CPP1 and CPP2 has been depicted in Figure 12(a) and Figure 12(b) respectively, with the identities (or interpretations) of the states involved listed in Table 7. Darkened nodes are used to represent the final states. For convenience of illustration, each state is labeled by a unique number[5]. As shown, separate nomenclatures have been applied to CPP1 and CPP2 to emphasize that their states are in fact different.



(a) CPP1



(b) CPP2

Figure 12: Syntactic parsing perceived as state transitions. The final states 76 and 72' both output the parse tree (((D (A N)) (P (D N))) (V (D (A N)))).

6.2 Characteristics of the Finite State Automata Extracted

Several characteristics of this finite state automata merit further discussion :

- The total number of states in the finite state automata of CPP1 and that of CPP2 are comparable (see Table 6) , in spite of the fact that they are exposed to different training environments. CPP2, with additional knowledge of the phrases in the training sentences, develops more final states (but less intermediate states) than CPP1. This accounts for its better generalization performance. From

[5]These numbers indicate the relative order in which different states are extracted. For example, the first state found for CPP1 will be assigned the label 1.

Table 8: The number of additional states of each type that were identified when the 32 testing sentences were parsed by each of CPP1 and CPP2

	CPP1	CPP2
Starting State (#)	+0	+0
Intermediate States	+22	+20
Final States	+16	+19
Total	+38	+39

our experience, the total number of states of both CPP1 and CPP2 does not vary a lot (within a range of ±10) in different runs of the experiments, and the same applies to the number of final states.

- If the parser simply remembered the training cases as independent sentences and parse trees by brute force, the total number of states should be 234. In CPP1 and CPP2, we found that the total number of states were only 182 and 184 respectively (see Table 6). The state transition sequences corresponding to the parsing of two similar sentences have overlapped in certain degree, thus some intermediate states have been shared. This implies that a certain degree of abstraction of grammatical knowledge has been achieved.

- Given a 100% success rate in training, each of the 80 training sentences should correspond to a unique final state, giving a total of 80 final states. For CPP1, in addition to these 80 states, 12 extra final states have been identified in our experiment, each corresponds to a syntactically well-formed parse tree or constituent phrase of some kind. When we interpret the intermediate states of CPP1 (e.g. states 3, 60 and 72 in Table 7), some of the states show traces of partial sub-trees. These are the evidence that CPP1 has in some extent induced the sentences' internal structure. Thus, grammatical knowledge has been acquired.

- The method by which the finite state automata is constructed is deterministic. The definitions of the states are uniquely defined by the set of weights of the trained CPP.

6.3 Generalization

Interestingly, the generalization capability of CPP can be explained by the finite state automata model. In addition to the final states corresponding to the training sentences, the training process also organizes the representational space (which is the hidden layer of CPP) in such a way that another set of states can be decoded to output the correct parse trees of the testing sentences. In our case, when the 32 testing sentences (26 of which can be correctly generalized) were parsed by CPP1, 16 additional final states were identified. The other 10 final states were among the 12 extra final states that were obtained during the parsing of the 80 training sentences. For example, when the testing sentence ⟨ D A N P D N V D N ⟩ was parsed by CPP1, the final state 75 had already been identified during the parsing of the training sentence ⟨ D A N P D N V D A N ⟩ (see Figure 13). For CPP2, 19 additional final states were identified by parsing the 32 testing sentences (see Table 8).

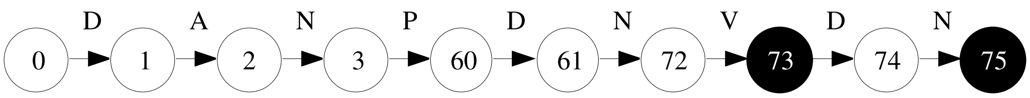


Figure 13: Parsing the testing sentence ⟨ D A N P D N V D N ⟩ by CPP1. The final state 75 had already been identified when the training sentence ⟨ D A N P D N V D A N ⟩ was parsed (see Figure 12a).

6.4 Error Recovery

The error-recovery capability of CPP can also be accounted for in a similar way. In parsing an erroneous sentence, the state transition sequence is deviated from its normal track (which is followed when the original sentence is parsed). The error is then corrected until the normal track is finally resumed and the parse tree corresponding to the original intact sentence is derived.

In Figure 14, 15, 16 and 17, an erroneous sentence of each of the four types of errors (SS, SW, SM and SE) has been depicted respectively. In each case, the lower track is followed when the original correct sentence is input whereas the upper state transition sequence traces the parsing of the erroneous sentence. As in the case of generalization, new states are created (they are represented by nodes with a shadow in the Figures). Unlike the cases in Section 6.3, these new states are intermediate states only and the parser tends to reside in final states that were discovered before.

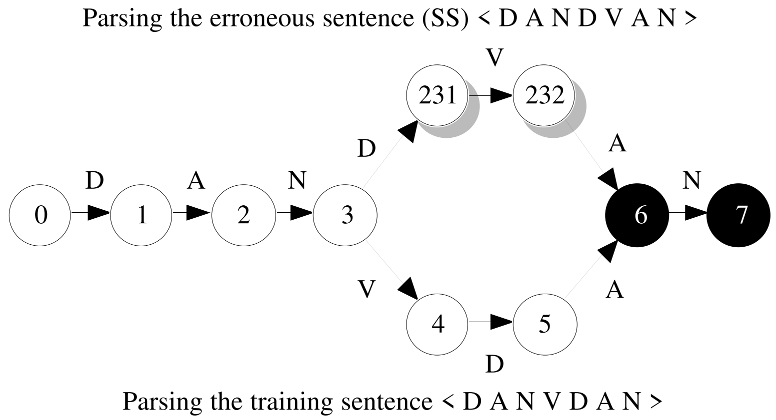


Figure 14: Error Recovery of an erroneous sentence of type SS (with the $4th$ and $5th$ elements swapped) perceived as state transitions. The final state 7 outputs the parse tree ((D (A N)) (V (D (A N)))).

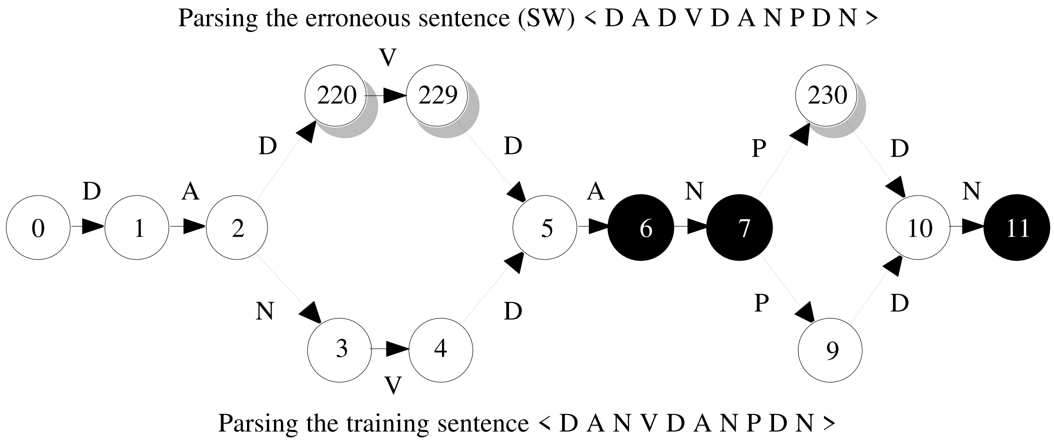


Figure 15: Error Recovery of an erroneous sentence of type SW (the $3rd$ element should be "N" instead of "D") perceived as state transitions. The final state 11 outputs the parse tree ((D (A N)) (V ((D (A N)) (P (D N))))).

In certain sense, the states of the finite state automata (and also CPP) are not discrete. Each of them behaves like an "attractor" and represents a wide range of SRAAM coding (or hidden layer activations) which are decoded to the same parse tree. This characteristic equips CPP with a certain tolerance to noise.

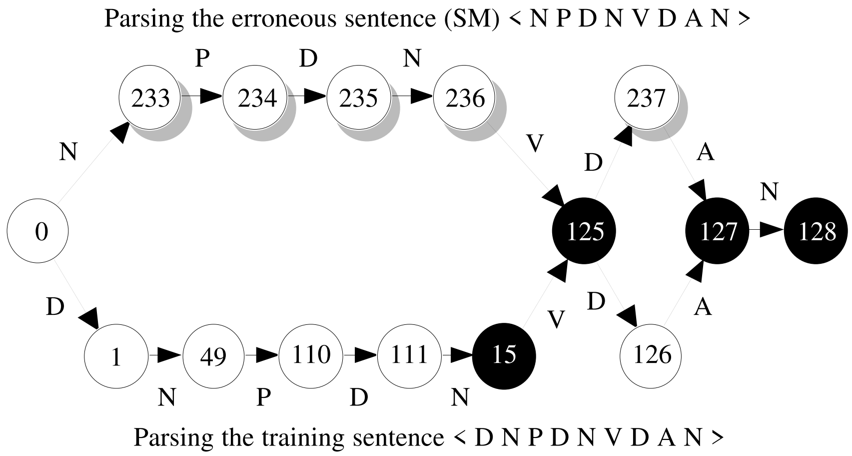


Figure 16: Error Recovery of an erroneous sentence of type SM (the $1st$ element "D" is missing) perceived as state transitions. The final state 128 outputs the parse tree (((D N) (P (D N))) (V (D (A N)))).

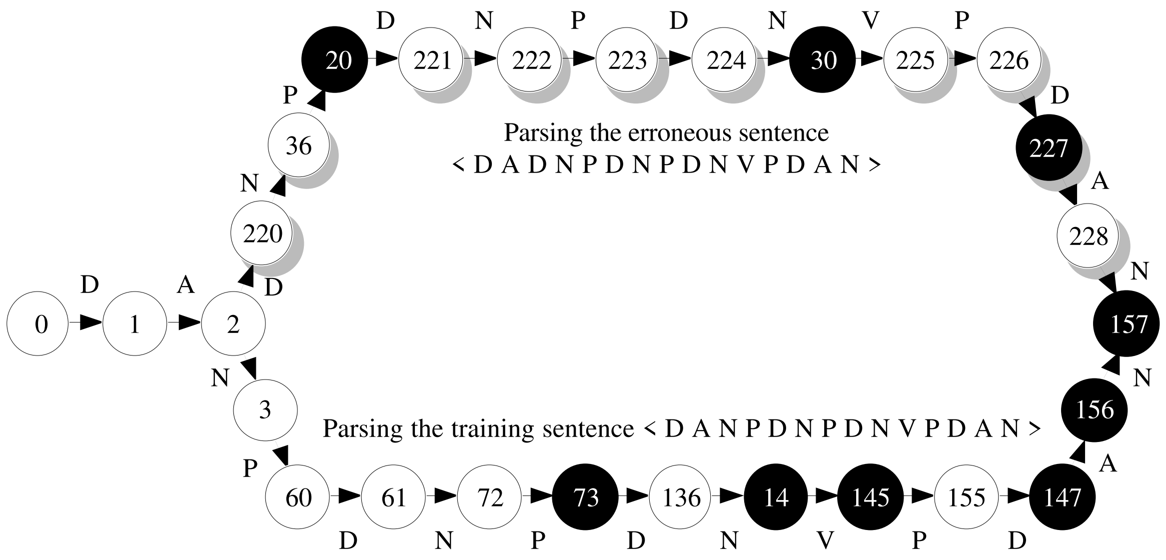


Figure 17: Error Recovery of an erroneous sentence of type SE (the $3rd$ element "D" is superfluous) perceived as state transitions. The final state 157 outputs the parse tree ((((D (A N)) (P (D N))) (P (D N))) (V (P (D (A N))))).

6.5 Implications

From the encoding point of view, the vector of hidden layer units of CPP constitutes a representational space for sequences (whether sentences or preorder traversals) and their sub-sequences. Each point in this space can be interpreted in two different ways – as a sentence sequence or a preorder traversal sequence. As the hidden layer units are real-valued, an infinite capacity is provided in principle. But obviously, because of the discretization due to encoding and decoding, only a finite number of them can be configured by training to give a syntactically well-formed parse tree upon decoding.

On the other hand, given a context-free grammar which is recursive, an infinite number of different sentences can be generated. This implies that the correct finite state automata should have an infinite number of final states, with the output of each represents a unique parse tree. Needless to say, it is impossible to achieve practically given only a limited number of training examples. Apparently, this has placed a theoretical limit on the computation power of CPP. What training should strive for (or can strive for) is to organize the weight space (and also the representational space) of the parser network in such a way that the maximum number of extra final states can be realized/explored in addition to those defined by the training sentences. Doubtless, this at least requires the selection of a training set that is representative of the possible types of structures and their possible modes of syntactic combinations.

7 Conclusion

In this paper, we have successfully tackled a symbolic AI problem that considered to be "hard" for neural networks, namely syntactic parsing. As revealed by the simulation, CPP has achieved excellent generalization performance and robust processing. By integrating the two techniques of linearization and confluent inference, CPP has imposed a topological ordering on the representations developed, thus preserving the structural characteristics of the data structures (that is, the sentences and the parse trees) they encode. The resulting representational space exhibits a nested, hierachical and regular structure resembling the idea of Conceptual Space as proposed by van Gelder [30].

To have a better understanding of the underlying parsing mechanism of CPP, an analysis has been presented in which a finite state automata is constructed upon successful training of the network. Based on this formalism, syntactic parsing, generalization and error recovery can all be explained in terms of state transitions. Interestingly, holistic parsing can also be perceived as if a rational inference process is involved (as manifested by the states which CPP or the finite state automata comes across during the processing of a sentence). Although this analysis can in some sense serve as an illustration that an abstraction of grammatical knowledge has really occurred during training, it has at the same time revealed a theoretical limit on the computational power of CPP in learning an infinite grammar perfectly.

To a certain extent, the performance of a trained CPP can be thought of being reflected by the characteristics of the finite state automata extracted. For example, the generalization performance is intimately related to the total number of final states that can be realized as well as their distribution (as "points") in the representational space, while its error recovery capability is affected by the "attraction basin" of each of these final states (perceived as "attractors"). The definition of the finite state automata is deterministic and its characteristics evolve as a result of training only. It is thus worth exploring their relationships with factors such as the number of hidden units employed (which determines the maximum representational capacity of CPP), the training environment (including the parameters used such as the learning rate and whether sequential training or parallel training is adopted), the coding for the terminal symbols as well as the training patterns used (their variety and combination). This type of study may shed light on the further improvement on the performance of CPP.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading:Mass, 1986.

[2] J. Allen. *Natural Language Understanding.* The Benjamin/Cummings Publishing Company, Inc., second edition, 1995.

[3] G. Berg. Learning recursive phrase structure : Combining the strengths of PDP and X-bar syntax. Technical Report TR91-5, University at Albany, 1991.

[4] D. S. Blank, L.A. Meeden, and J. B. Marshall. Exploring the symbolic/subsymbolic continuum: A case study of RAAM. In J. Dinsmore, editor, *The Symbolic and Connectionist Paradigms: Closing the Gap*, pages 113–148. Lawrence Erlbaum Associates, Hillsdale, NJ, 1992.

[5] D. J. Chalmers. Syntactic transformations of distributed representations. In N. Sharkey, editor, *Connectionist Natural Language Processing*, pages 46–55. Intellect Books, 1992.

[6] L. Chrisman. Learning recursive distributed representations for holistic computation. *Connection Science*, 3:345–366, 1991.

[7] J. L. Elman. Finding structure in time. *Cognitive Science*, 14:179–211, 1990.

[8] C. Giles, C. Miller, D. Chen, H. Chen, G. Sun, and Y. Lee. Learning and extracting finite state automata with second-order recurrent neural networks. *Neural Computation*, 4(3):393–405, 1992.

[9] K. S. Ho and L. W. Chan. Confluent Preorder Parsing. Technical Report CS-TR-95-03, Department of Computer Science, The Chinese University of Hong Kong, 1995.

[10] K. S. Ho and L. W. Chan. Syntactic parsing using RAAM. In *Proc. World Congress on Neural Networks*, volume 1, pages 485–488, 1995.

[11] K. S. Ho and L.W. Chan. Representing sentence structures in neural networks. In M. W. Kim and S. Y Lee, editors, *Proc. International Conference in Neural Information Processing Systems*, volume 3, pages 1462–1467, 1994.

[12] K. S. Ho, W. S. Luk, and L. W. Chan. Parallel implementation of the Confluent Preorder Parser on the DECmpp. in preparation.

[13] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading:Mass, 1979.

[14] R. A. Johnson and D. W. Wichern. *Applied Multivariate Statistical Analysis.* Prentice-Hall, Inc., third edition, 1992.

[15] S. C. Kwasny and K. A. Faisal. Symbolic parsing via subsymbolic rules. In J. Dinsmore, editor, *The Symbolic and Connectionist Paradigms: Closing the Gap*, pages 209–236. Lawrence Erlbaum, Hillsdale, NJ, 1992.

[16] S. C. Kwasny and B. L. Kalman. Tail-recursive distributed representations and simple recurrent networks. Technical Report WUCS-93-52, Department of Computer Science, Washington University, 1993.

[17] J. R. Levine, T. Mason, and D. Brown. *lex & yacc.* O'Reilly & Associates, Inc., second edition, 1992.

[18] A. Maskara and A. Noetzel. Forced simple recurrent neural networks and grammatical inference. In *Proc. the Fifteenth Annual Conference of the Cognitive Science Society*, pages 420–425, Hillsdale, NJ, 1993. Lawrence Erlbaum Associates.

[19] R. Miikkulainen. Subsymbolic case-role analysis of sentences with embedded clauses. Technical Report AI93-202, Department of Computer Sciences, the University of Texas at Austin, 1993.

[20] T. A. Plate. *Distributed Representations and Nested Compositional Structure.* PhD thesis, Graduate Department of Computer Science, University of Toronto, 1994.

[21] J. Pollack and D. Waltz. Massively parallel parsing: A strongly interactive model of natural language interpretation. *Cognitive Science*, 9:51–74, 1985.

[22] J. B. Pollack. Recursive distributed representations. *Artificial Intelligence*, 46:77–105, 1990.

[23] J. B. Pollack. The induction of dynamical recognizers. *Machine Learning*, 7:227–252, 1991.

[24] R. Reily. Connectionist techniques for on-line parsing. *Network*, 3:37–45, 1990.

[25] D. E. Rumelhart, G. E. Hinton, and R.J. Williams. Learning internal representations through error propagation. In D. E. Rumelhart, J. L. McClelland, and The PDP Research Group, editors, *Parallel distributed processing: Experiments in the microstructure of cognition*, volume 1, pages 318–362. MIT Press, Cambridge, MA, 1986.

[26] D. Servan-Schreiber, A. Cleeremans, and J. L. McCleeland. Graded state machines: The representation of temporal contingencies in simple recurrent networks. *Machine Learning*, 7:161–193, 1991.

[27] A. Stolcke and D. Wu. Tree matching with recursive distributed representations. Technical Report TR-92-025, International Computer Science Institute, Berkeley, 1992.

[28] G. Z. Sun, C. L. Giles, H. H. Chen, and Y. C. Lee. The neural network pushdown automaton: Model, stack and learning simulations. Technical Report UMIACS-TR-93-77 & CS-TR-3118, University of Maryland, 1993.

[29] T. van Gelder. Compositionality: a connectionist variation on a classical theme. *Cognitive Science*, 14:355–384, 1990.

[30] T. van Gelder. Making conceptual space. In S. Davis, editor, *Connectionism: theory and practice*, pages 179–194. Oxford University Press, New York, 1992.

[31] R. Watrous and G. Kuhn. Induction of finite state languages using second-order recurrent networks. In J. Moody, S. Hanson, and R. Lippmann, editors, *Advances in Neural Information Processing Systems 4*, pages 309–316, San Mateo, CA, 1992. Morgan Kaufmann.

[32] S. Wermter. *Hybrid Connectionist Natural Language Processing*. Chapman & Hall, London, 1995.