Application of Search Algorithms to Natural Language Processing

Takeshi Matsumoto
School of Informatics and Engineering
Flinders University of South Australia
takeshi.matsumoto@flinders.edu.au

David M. W. Powers
School of Informatics and Engineering
Flinders University of South Australia
powers@infoeng.flinders.edu.au

Geoff Jarrad
Business Intelligence Group
CSIRO ICT centre
geoff.jarrad@csiro.au

Abstract

Currently, the most common technique for Natural Language parsing is done by using pattern matching through references to a database with the aid of grammatical structures models. But the huge variety of linguistical syntax and semantics means that accurate real time analysis is very difficult. We investigate several optimisation approaches to reduce the search space for finding an accurate parse of a sentence where individual words can have multiple possible syntactic categories, and categories and phrases can combine together in different ways. The algorithms we consider include mechanisms for ordering that reduce the search cost without loss of completeness or accuracy as well as mechanisms that prune the space and may result in eliminating valid parses or returning a suboptimal as the best parse. We discuss the development and benchmarking of the existing and proposed algorithms in terms of accuracy, search space and parse time. Speed up of an order of magnitude was achieved without loss of completeness, whilst decrease of over two orders of magnitude was achieved in the search space. A further order of magnitude reduction of both time and search space was achieved at the expense of some loss of accuracy in finding the most probable parse.

1 Introduction

The complexity and the sizes of the lexical databases and grammatical rules contribute to most of the behaviour of Natural Language parsers. By increasing the sizes of the database or including a more complex set of grammatical rules, the parser is able to handle the parsing of more complex sentences or is able to include more accurate information to the parsed sentences, but the introduction of these results in a more complex parsing procedure and the capability to compute for more cases is necessary for the parser. Even without the extended database or rules, parsing of long sentences is often avoided due to the extremely large amount of different possibilities in parsing the sentence.

To counteract the increase in the parse time form the application of complex grammatical rules, we explore the effects of applying search algorithms to a parser to reduce the search space and hence enhance the parsing speed. To measure the accuracy of the parse, we use a simple scoring system derived from the probability that a particular structure would exist. This scoring system does not always parse the sentence correctly, but it provides a good indication of the likeliness of the structure from a statistical point of view.

The purpose of the project is to provide a faster way of parsing sentences without losing the effect of grammatical structures, or the semantic and syntactic information that have been applied to or extracted from the parser. These areas being the key focus of most research done in NLP and will continue to increase in complexity in the future.

2 Parsing

The parser we are using was the probabilistic, lexicalised combinatory, categorical grammar (PLCCG) parser implemented by the CSIRO[1] (Jarrad et al., 2003) that incorporates a bottom-up search strategy. In the training stage, the parser builds up a statistical model of the grammatical structure by learning from a manually parsed corpus, which is used to assign the possible categories and the probabilities of the particular category for a word, and also the probabilities associated with the actual combination of two structures. The CCG[2] (Steedman, 1996) incorporated in the parser defines the rules and methods used in the combination stage of the parser, and implements an extended set of the standard CCG combinators (Jarrad et al., 2003) that makes the grammar more flexible. The nature in which a combination occurs is very much like using the link grammar rules to combine between the different states.

Initially, the individual words are given a set of potential categories that it has seen for the particular word in the training corpus. Due to the varieties in the training data and the increase in freedom gained from the extended grammar, some words are given a huge set of potential categories. This creates a more robust grammar, which can handle the parsing of complex sentences, but also contributes to an explosion in the search space. If the particular word was not seen in the training corpus, a lexical database called WordNet (Miller et al., 1993) is used to assign the possible categories for the word. This is done by extracting the part of speech (POS) tags for the unknown word and assigning all the possible categories for that POS to the word. Finally, if the word was not found in WordNet, the set of all possible categories are assigned to the unfamiliar word with the probability of the category being the frequency of the category form the entire training corpus. The difference in the number of initial categories for a word can be enormous, ranging from one to over one thousand.

The probability scores for the states are derived from the combination of 3 transition probabilities, the word category transition, categorial transition, and the lexical transition. The parser uses these probabilities to derive the scores of a parse to find the most probable parse, which is derived by exhaustively combining all possible states for the parsing sentence. The approach in which this is done is very similar to the chart parser (Charniak, 1993). This eventually results in the formation of a state combing all words in the sentence, which we call the terminal state. The scores of all the terminal states are compared and the parser returns the parse tree structure for the most probable state. If there are multiple states that are equivalent in how it was structured, the parser keeps the state with the higher probability. For duplicate scores, the first one it encounters is kept.

The task of finding all of the possible combinations is almost as difficult as the travelling salesman problem. The search space expands as the third power of the words, but due to the fact that states can only combine with adjacent states, some reduction in the search space occurs automatically. But on top of the possible combinations of the sequences of words, there is a squared factor for the number of possible categories each combination would need to consider, which results in the model:

$$\sum_{i=1}^{l-1} \sum_{j=i}^{l-1} \sum_{k=j+1}^{l} N_{i,j} \cdot N_{j+1,k}$$

Where i, j, and k represents the starting position of the left sequence, ending position of the left sequence, and the ending position of the right sequence, respectively. l represents the number of words in the sentence and $N_{i,j}$ represents the number of states for the sequence between i and j.

The above model clearly indicates that the task of parsing, especially when the number of categories for a word can be of the order of several hundreds, is a lengthy task. This value can reach up to several millions even on sentences with less than 10 words. Hence the need for a search algorithm that would prune the search space without any loss of accuracy.

3 Optimal-Search Algorithm

The major goal of this project was to explore alternative standard and novel algorithms that were appropriate to the task and could relatively easily be slotted into the existing parser framework. The

[1] Commonwealth Scientific and Industrial Research Organisation

[2] Combinatory Categorial Grammar

kind of algorithms and optimisations that are reasonable is tightly constrained by the nature of the CCG model and the PLCCG implementation. Another major constraint of the algorithm is one that is often ignored, which is the overhead in the execution of the algorithms. This factor plays an equally important role in the search problem, but has often been ignored due to the increase in the hardware performance rate. The algorithmic design was modularised, so that an easy switching of the algorithm could be done with a uniform interface to the rest of the original parser. This meant that the algorithm relied on some of the existing structure of the parser, which was the cause of some limitations in the algorithms and is an area that could be modified in the future to further increase the efficiency of the parser.

The first algorithm that was considered was Adaptive probing (Wheeler, 2001) and this was tested on a subset of the problem by simulation using a toy language (Kilby, 2002). This algorithm was considered due to the gain in search speed seen in the simplified search problem, but was rejected due mainly to the random nature of the search, which means that an exhaustive search was necessary to provide the most probable parse.

The first enhancement was to apply a different ordering of the combinations to allow the fast build up of the relevant sections of the parse tree. By ranking the states in order of their probabilities, the parse tree was built up in such a way that the most probable state in the tree was considered first. Due to the randomised build up of the parse tree in terms of extension of the branches in the search tree, the algorithm had to include an indicator to allow the extension of branches from nodes, even after it had been used to construct its children states already. This backtracking mechanism was implemented using a list containing all of the states, which was divided into two sections. A pointer into the list indicated the division point between the two sections, one of which contained all of the states that had been used to combine with other states, and the other section contained all of the states that had not been used to combine with other states. Whenever a state was used to combine with other states, it was placed in the used section and the states that resulted from the combination were placed in the non-explored section. This divided list ensured that no two same combinations would ever occur more than once.

To apply the ranked ordering, the list was maintained in a sorted manner by their probability scores and the pointer simply moved along the list, as more states were used to combine with other states. The state being pointed to by the pointer, which was the state being used to combine with other states of higher scores, was called the pivot state. By combining the pivot state with states of higher scores, the algorithm guaranteed that resulting state of the combination would be equal or lower scored than the pivot state. This allowed for a simple algorithm for maintaining the ordered list. The ranking algorithm is essentially embodied by the following pseudo-code:

1. Populate the list with every state for every word.
2. Sort the list by their probability scores.
3. Set pointer at the first state in the list.
4. While the list contains un-combined states:
5. Set pivot as the next most probable state.
6. Return if pivot state is a terminal state.
7. Combine pivot with all adjacent states with higher probability.
8. Insertion sort all newly created states in to the list.
9. Return failure

With the application of this ordering, the algorithm allowed for early termination of the search, since the newly created states (being of equal or lesser probability) must be inserted below the pivot state due to the cascading effect of the product of the probability. Any terminal state found later would have a lower probability than the first one that was found, so the algorithm guarantees the retrieval of the most probable state without having to exhaustively search all possible combinations.

By only using a single list to maintain all possible derivation of the states, traversals and maintenance of the ordering of the list used up a lot of valuable time. To counteract this, we re-introduce a charting behaviour as the second improvement to the algorithm. We implemented a table, called the indexed table, in which all the states that were in the used section were placed, rather than keeping them in the same list. The table also grouped together the states that occupied the same starting and ending positions, to simplify the decision process in determining which states were adjacent to the pivot state. The ranked list was replaced by a

table, which we called the sorted table that handled the push and pop manipulations to simplify and to modularise the algorithm for future use.

The third major step involved the use of a critical score, which is the score of the currently most probable terminal state in the sorted table. By not operating on states that are going to produce a lower probability than the critical score, it allowed for a large pruning of the search tree, weeding out states with very low probability that would not contribute to the most probable terminal state. The algorithm also provides a pre-processing stage before a combination between states took place, which contributed to a little overhead, but managed to cut down the amount of unnecessary combinations and avoided the lengthy combination stage of two states.

The scoring system, as it stood, meant that combined states of large length would have a very low score, even if they consisted of very probable sub-structures. There was a necessity to allow larger sized states a better score to indicate their higher desirability. The next major step in the evolution of the algorithm was to alter the scoring system to allow larger sized states higher ranking than by the use of the raw probability scores. This was achieved by normalising the scores to the most probable scores of the corresponding positions of the states and hence altered the ranking system so that states that occupied different sections in the sentence were compared relative to other states that occupied the same sequence of words. The scores of a potential perfect combination of the most probable states for each word were used to derive the normalisation scores for the particular sequence. This is not the most accurate way of determining the normalisation scores, but it provided an efficient way to change to ordering while not causing too much overheads in the pre-processing stage. The normalisation scores and the scores used for ranking are derived by:

$$S_{i,j}^{normal} = \prod_{k=i}^{j} S_{k,k}^{\max}$$

$$S_{i,j}^{rank} = S_{0,i-1}^{normal} \cdot S_{i,j} \cdot S_{j+1,l}^{normal} = S_{0,l}^{normal} \cdot S_{i,j} \Big/ S_{i,j}^{normal}$$

In the above model, i and j represents the starting and ending indexes of the state and l represents the length of the sentence. $S_{i,j}{}^{normal}$ represents the normalization score for the sequence in the range between i and j, $S_{k,k}{}^{max}$ represents the score of the most probable states for word at k. $S_{i,j}{}^{rank}$ represents the score used for ranking, but it also represents the heuristical score of the state. $S_{i,j}$ represents the raw probability score of the particular sequence which starts and ends at positions i and j. Note that the $S_{0,l}{}^{normal}$ is a constant for the same sentence and can be factored out for the purpose of ranking, which gives:

$$S_{i,j}^{rank} = S_{i,j} \Big/ S_{i,j}^{normal}$$

The combined algorithm still maintains the retrieval of the parse tree with the same probabilistic score as the exhaustive algorithm, but has managed to prune a very large section of the search tree without creating too much overhead in the execution of the algorithm. The pseudo–code for this new algorithm is:

1. Construct the normalisation mapping.
2. Initialise the critical score to zero.
3. Populate and sort the sorted table with all states for all words using the normalised scores.
4. Remove the most probable state and insert into the indexed table.
5. While the sorted table contains uncombined states:
6. Remove the most probable from the sorted table as the pivot.
7. Return if the pivot is a terminal state.
8. Combine pivot with all adjacent states in the indexed table that don't fall below the critical score.
9. For every state that has been created:
10. Adjust the critical score if the produced state is a terminal state and the score is better.
11. Insert the created states into the sorted table with the normalised score.
12. Insert the pivot into the indexed table.
13. Return failure.

We also investigated alternative algorithms that included more pruning in the search tree, and also the effects of prematurely ending the search when an approximate result was found. We experi-

mented with ideas like pruning lower scored states at the start of the algorithm (beam search), approximating the correct parse to be the first terminal state it found, and applying a different priority system that encouraged the build up of larger sized states without first building up the sub-structures. The beam search had the same effects to the parser as a reduced set of categories and combinators, in that, some valid sentences could not be parsed because of the reduced amount of ways in forming the valid parse. This is a very common approach taken to optimise a searching task (Goodman, 1997), but was not the desired approach for this project since the task of the algorithm was to find the most probable parse for the sentence.

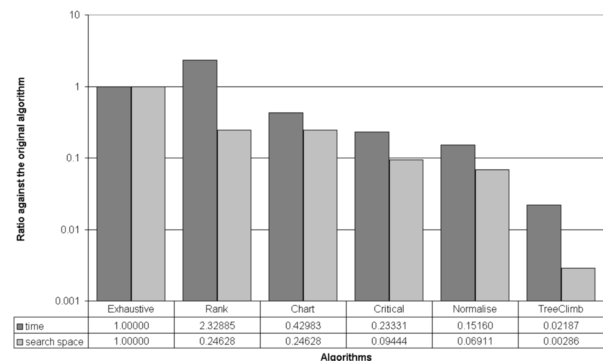
By terminating the algorithm prematurely, the parser sometimes retrieved the non-optimal result and also did not contribute much to the reduction of the parsing time, due to the improved ordering of the search algorithms.

By re-ordering the search, so that the build up of larger sized states were prioritised, the effects of the ordering by their scores were lost and hence the algorithm had to either exhaustively search all combinations to determine the most probable parse, or it had to end the algorithm after the production of some terminal state was made. This did not guarantee the retrieval of the most probable parse and it also meant that some unlikely combinations that could have been avoided by the ranking had to be done.

When implementing most of the experimental algorithms, some of the core structure to the algorithm had to be modified, but an interesting algorithm was discovered in the process. This was the product of the beam search and the prioritising of larger states, which we called the tree-climbing algorithm. The beam search stage, which we called the seeding stage, involved building of the parse with only the most probable state for each of the word, and the tree-climb approach, which was applied in the subsequent stage, resulted in an algorithm that was faster than the combined optimal algorithm, but was not as accurate when it came to the retrieval of the most probable parse. However, the proportionality of the incorrect parse was significantly lower than the application of just the beam search or the tree-climb algorithm. The tree-climb approach was not attractive in terms of both parsing accuracy and time in the cases where the sub-structures had to be built up first when it was applied by itself, but the seeding stage constructed the majority of the necessary sub-structures in the search tree, and hence allowed the tree climb algorithm to connect up the un-combined sections and quickly form a parse for a sentence. Although this algorithm did not guarantee the retrieval of the most probable parse, it provided alternative points of view in the relevance of the scoring system which was used to determine the 'correct parse' of the sentence; some parses which were retrieved were structured more similarly to the humanly evaluated parse than the most probable parse, even though they were assigned lower probabilities.

4 Results

The algorithms were trained and tested on both the Susanne corpus and the Penn Treebank corpus (Mitchell et al., 1992), approximately 95% of each was included in the training sets (sections 02 to 21 for the Penn Treebank) and a randomly chosen subset from the rest of the corpus was used for the testing sets (section 23 for the Penn Treebank corpus). This corresponded to 50 sentences in the Susanne corpus and just over 580 sentences in the Penn Treebank corpus. The major difference between the two corpora is the number of possible categories it contains. Where the Susanne corpus contains just over 500 categories, the Penn Treebank corpus contains over 1200 categories it can assign to each of the words. The two corpora were used to test the performance of the developed algorithms; hence the parsing accuracy is not the intended matter being evaluated here.



	Exhaustive	Rank	Chart	Critical	Normalise	TreeClimb
time	1.00000	2.32885	0.42983	0.23331	0.15160	0.02187
search space	1.00000	0.24628	0.24628	0.09444	0.06911	0.00286

Figure 1: Benchmarked logarithmic plot comparing the exhaustive algorithms to the developed algorithms on the Susanne corpus.

The PLCCG parser and the developed algorithms were implemented on Python, the reason being that the parser is still under development in the areas of syntactic and semantic accuracy.

The progressive increments to the proposed algorithm all contributed to large areas of the search space being pruned, but due to the overheads in the execution of the algorithm, some of the benefits were not as much as first expected.

Figure 1 indicates the ratio differences between the original exhaustive algorithm and the proposed algorithms on the Susanne corpus. The results from the Penn Treebank corpus are not shown, since we were unable to obtain the results for some algorithms due to memory problems. The left column indicates the parsing time and the right column indicates the amount of search space it explored, or the number of combinations made during the parse. The difference between the two indicates a rough estimate of the overhead in the execution of the algorithm compared to the original algorithm. As the plot indicates, the parse time of the ranked algorithm was excessive. The overhead in determining the adjacent states contributed to most of the parse time and resulted in a worse parse time, even though it was only exploring a quarter of the search space.

After the charting was implemented, the benefits of the algorithms became more apparent, even through it was still exploring the same search space. The reduction of the search space by 75% from the use of the ranked ordering indicates that most of the categories assigned to the initial words did not make much contribution, due to the rareness of its own category and the corresponding derived states.

The inclusion of the critical score made another dramatic reduction in the search space, indicating that a lot of unnecessary searching was occurring after the terminal state was produced, which could be carefully pruned out without affecting the final result. The proportionality between the search time and the search space increased with the inclusion of the critical value, which was contributed by the overheads in the pre-processing stage before the combination between the states occurred.

The use of the normalised scores contributed to yet another reduction in the parse time and search space. This algorithm did not make as much use of the critical value compared to the raw critical algorithm due to better ordering of the states, but since the normalisation scores are not the perfect representation of the relative scores to each position, which is impossible to predict, the critical value still plays an important role in the algorithm. This algorithm introduces extra processing to calculate the normalised scores and to re-order the states with the same scores, but the overheads is still a lot less than the raw critical scored algorithm, due to the repeated pre-processing overheads.

The experimental tree-climb algorithm result seen on the far right shows an impressive parse time and huge reduction in the search space, but has slight inaccuracies parse compared to the other algorithms, which can be seen in Table 1.

	Exhaustive	Optimal	Suboptimal
Susanne			(%)
Parse time	100.0	15.2	1.7
Search space	100.0	6.9	0.3
Most probable	100.0	100.0	84.0
Penn Treebank			(%)
Parse time	100.0	10.4	0.7
Search space	100.0	4.7	0.1
Most probable	100.0	100.0	66.7

Table 1: Statistics of parsing of the optimal and suboptimal algorithms for both the Susanne and Penn Treebank corpora.

The parse time and the search space are represented as the proportionality compared to the exhaustive algorithm and the percentage that the algorithm retrieved the most probable parse is indicated in the last row. The optimal algorithm is the combined algorithm of all the algorithms that provided benefits to the parsing speed without the loss of accuracy and the suboptimal algorithm is the tree-climb algorithm, which provided a fastest and also a reasonably accurate result from all tested suboptimal algorithms.

The results from the 2 corpora indicate similar trends in the characteristics of the algorithms, which indicate a consistent improvement from the application of the algorithms. The parse time and the search space showed a bigger improvement from the larger Penn Treebank corpus, even though there is over twice the number of categories to choose from. This is suspected to be the fact that more trivial sentences exists within the Penn Treebank testing set. Another contributing factor to this is the fact that the training set is a lot larger in the

Penn Treebank test. This means that the algorithm does not need to look up unknown words from WordNet or spend time assigning all possible categories for the word.

The optimal search algorithm returns the most probable parse tree, but sometimes varied in the tagging and bracketing of the parse due to the cases when multiple parses have the same probability. The tree-climb algorithm's performance in the accuracy domain is relatively poor, but some of the loss in the accuracy can be recovered by altering the amount of states used in the seeding stage. However, because the algorithm loses track of the ranking of the states, the algorithm must exhaustively combine all states to determine the most probable parse.

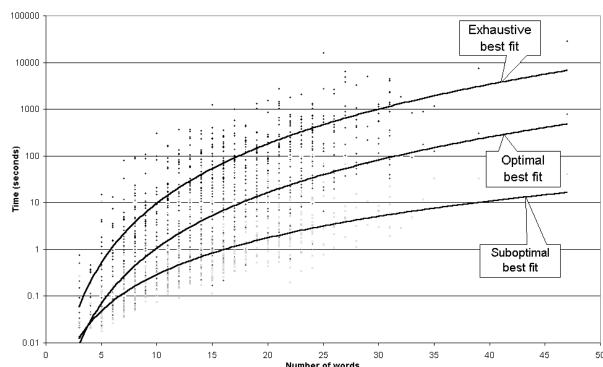


Figure 2: Number of words in the sentence versus parsing time on the Penn Treebank corpus for the exhaustive, optimal and the suboptimal algorithm.

On a corpus based comparison, it is fairly easy to see the improvements of the developed algorithms, but for the task of NLP, it is probably more important to look at a per sentence comparison, especially if it is in an environment where human interaction is required. Figure 2 indicates the relationship between the parsing time and the number of words in the sentence for the exhaustive, optimal and the suboptimal search algorithms. There is a huge reduction in the parse time from the algorithm with the optimal algorithm, and an even greater reduction from the suboptimal algorithm. Both these algorithms possesses another great feature in that the exponential coefficient factor for the parse time is a lot less than the exhaustive algorithm. This means that the algorithm works more efficiently with longer sentences, but the plot still indicates that the new algorithms are still better than the exhaustive algorithm for short sentences, even with the extra overheads from various forms of initialisation.

The long parsing times are the consequences of using a scripting language for the development and testing of the parser. The results should reduce by a factor of several tens or even hundreds if the parser was implemented on a natively compilable language.

Figure 3 describes the overall efficiency of the algorithms, which displays 4 different dimensions about the algorithms. The first is the linear slope seen in all the algorithms. This indicates that the non-searching processes in the individual algorithms (overheads), like the initialisation stage do not contribute greatly to the parse time, with the exception of the set of plots around the 10 second range, which has deviated from the other results. This is due to the extra time taken for the algorithms to fetch the relevant categories form WordNet and also the assignment of all possible categories. This is not apparent in the exhaustive algorithm, because they perform a lot more combinations if the number of categories assigned to each of the words is large, where as the optimal and the suboptimal algorithms does not take most of them into account. The actual slope of the plots indicates the sizes of the coefficient of the relationship, shown more clearly in Figure 2.

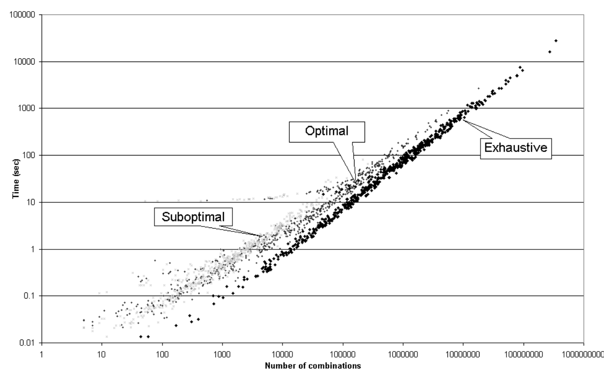


Figure 3: Efficiency of the algorithms measured on the Penn Treebank corpus.

The second dimension is the y-intercepts of the plots, which describes the efficiency of the execution of the algorithm. The smaller the y-intercept, the more efficient it is in executing each combination. The exhaustive algorithm clearly outperforming the others, due to the simple way it needs to be implemented. Due to the large over-

head in applying the suboptimal algorithm, there is a large overhead in the algorithm, meaning that if the states of a very low probability had to be used to produce a terminal state, this algorithm would run the slowest.

The third dimension is the spanning distance between the plots, which indicates the size of the exponential coefficient. The longer the distance between the quicker parses and the longer parses, the larger the exponential coefficient it has. The effect of which can be seen more clearly on Figure 2. The fourth and the final dimension is the average speed of the parse time, which is indicated by the average height of the points.

5 Conclusion and Future Work

Unlike most modern search algorithms that take advantage of the continuously increasing processing power of the modern day computers and hence lose elegance in the search technique, the developed search algorithm allows for the retrieval of the best possible solution in a very efficient manner while also taking into account of the overheads involved in execution of the algorithm. The implementation of the algorithm as the searching mechanism to find the most probable parse for the target parser has dramatically reduced the parsing time required to retrieve the same result as an exhaustive search mechanism.

The characteristics of the algorithm has the potential to be converted into a simple chunk parser, which is sometimes enough to extract the relevant information from the sentences. The proposed algorithm encourages the quick build up of subparses, rather than the linear build-up algorithm of the exhaustive algorithms, hence the order in which the combination occurs allows for the splitting of the sentence into sections or chunks by early termination of the algorithm.

The tree-climb algorithm needs further investigation, as the algorithm may possesses characteristics which may end up being more beneficial to the accuracy of the parser. This is due to the fact that the most probable parse is not always the correct parse. Further investigation techniques might include getting the algorithm to find multiple solutions before it is returned, or to measure the accuracy after altering the amount of states used in the seeding stage. Primitive experiments done on adjusting the seeding amount has decreased the error rate, but further tests are required to understand the effects on this.

By modifying the probability scores to include information on things like the syntactic and semantic context to provide a better indication of the grammar which will provide a better scoring system, the parser should be able to provide better results and still rely on the developed optimal algorithm to retrieve the most probable parse. This also means that the algorithm is generic enough to be applied to other kinds of search problems. The 4 main implemented techniques; ranking the nodes in the search tree to allow for early search termination, using charting to avoid processing of unwanted search space, applying the critical point scores and a quick pre-processing stage to avoid lengthy computation, and the use normalised scores to provide a better heuristical indication of the node being searched all provide vital ways to reduce the search space of the problem.

Acknowledgement

This project was supported in part by a scholarship from the CSIRO.

References

Eugene Charniak. 1993. *Statistical Language Learning*. MIT Press, Cambridge, England.

Jushua Goodman. 1997. *Global Thresholding and Multiple-Pass Parsing*. Proc. EMNLP-2.

Geoff Jarrad, Simon Williams, and Daniel McMichael. 2003. *A Framework for Total Parsing*. CMIS technical report 03/10.

Phil Kilby. 2002. *Applying Adaptive Probing to Achieve Fast Parsing*. Project proposal paper.

Mitchell P. Marcus, Beatrice Santorini, Mary A. Marcinkiewicz. 1992. *Building a large annotated corpus of English: the Penn Treebank*. http://www.cis.upenn.edu/~treebank/home.html.

George A. Miller, Richard Beckwith, Christiane Fellbaum, Derek Gross, and Katherine Miller. 1993. *Introduction to WordNet: An On-line Lexical Database*. http://www.cogsci.princeton.edu/~wn/.

Mark Steedman. 1996. *A Very Short Introduction to CCG*. ftp://ftp.cis.upenn.edu/pub/steedman/ccg/ccgintro.ps.gz

Ruml Wheeler. 2001. *Incomplete Tree Search using Adaptive Probing*. Proc. IJCAI-01.