Treegraft:
A Stochastic Transduction Chart Parser

NLP Lab Project Final Report
Spring 2008

Jonathan Clark
Advised by Alon Lavie

The goal of this project was to create a research tool that will enable future explorations of issues including the strong generative power of various types of grammars and the suitability of such grammars to applications such as parsing, paraphrasing, and machine translation. This goal culminated in the creation of a stochastic transduction chart parser. Efforts were made to ensure that this software was modular enough to allow for the dramatic and demanding code extensions often required by research.

In this report, I begin by exploring the research problem and motivations for creating this tool (Section 1). Next (Section 2), I provide a more formal description of the transduction chart parser that I implemented for monolingual and Synchronous Context-Free Grammars (CFG / SCFG). Then, I note a few of the more practical aspects of Software Engineering that I encountered during this project (Section 3) followed by an explanation of how I evaluated the formal correctness of my program (Section 4). Finally, I discuss some insights into Future Work that I would like to conduct using the tools and the knowledge that resulted from this project (Section 5) and I conclude (Section 6).

1 Problem and Motivation

1.1 Painful Decisions between Grammar Formalisms

In Grammars and Lexicons and in Grammar Formalisms, I learned that each Grammar Formalism provides its own strengths and weaknesses. Committing to a formalism is a big decision when writing a large grammar or devising a grammar induction strategy. However, while many of these formalisms differ strongly in their motivations, their mechanics share much in common. This leads me to question at exactly what points they become mechanically incompatible and which points are encompassed by some more general framework such that they need not be sacrificed by committing to only a single formalism.

Further, as became apparent to me when attempting to build a Japanese-to-English machine translation system using the Avenue Transfer Engine, language structures can be very non-isomorphic. This non-isomorphism goes well beyond the strong generative capacity of a SCFG. Often, languages do not express the same grammatical meanings via their word order. For instance, English encodes semantic roles while Japanese encodes new and old information (which participant is emphasized). Therefore, it might seem that we would expect there to be less correlation between the word ordering structures in these languages. However, in a statistical modeling, we are often not interested in outliers, and so to find out whether or not such grammatical non-isomorphism is truly a bottleneck in our models will require empirical evaluation.

1.2 Long-Term vs Short-Term Goals

This then begs the question, how do we go about pursuing these avenues of research? One way would be to modify existing software (namely, the Avenue Transfer Engine) to test these ideas. However, legacy C++ code can be difficult to modify. More importantly, the implementation process can be a path to a much deeper knowledge of concepts since holes in understanding typically produce holes in the implementation, forcing both to be addressed.

With this in mind, I took a second approach: write a research tool from the ground up that is built to enable the exploration of various types of grammars and parsing strategies. Since the goal of actually exploring these grammars is beyond the scope of this semester-long project, I have simply implemented this research tool as a first step toward addressing this larger research agenda.

2 Algorithm

2.1 Chart Parsing

Following the mechanics of the Avenue Statistical Transfer Engine[9], I implemented a transduction chart parser. As a stepping stone toward defining transduction chart parsing, we will first reproduce (with some minor additions) the algorithmic definition of Chart Parsing discussed in the Algorithms for NLP course[1].

For a parsing input $x = x_1 ... x_n$, we begin by processing the input from left-to-right with i corresponding to the input token being processed:

1. **Begin with the first input symbol**
   Set $i = 0$

2. **Initialize the Agenda with all POS of the current terminal input symbol**
   If the agenda is empty and $i < n$, then set $i = i + 1$, find all POS of x_i and add them as constituents $C(p_i, p_{i+1})$ to the agenda

[1] http://www.cs.cmu.edu/afs/cs.cmu.edu/project/cmt-55/lti/Courses/711/Class-notes/Chart-Parsing.pdf

3. **Pick a key from the Agenda**
   If the agenda is empty, stop. Otherwise, remove the next Key constituent $C(p_j, p_{i+1})$ from the agenda (note that $j \leq i$).

4. **Add all grammar rules that start with the Key as active arcs**
   For each grammar rule of the form $[A \rightarrow CX_1...X_m]$ (those that begins with the key C that we just got from the agenda), create an active arc $[A \rightarrow \bullet CX_1...X_m](p_j, p_j)$ and, iff such an arc does not already exist in the list of active arcs L, append it to L.

5. **Extend any existing active arcs with the key**
   (Without target side packing nor transduction constraints)
   For all active arcs of the form $[A \rightarrow X_1...\bullet C_k...X_m](p_h, p_j)$ (those that to advance their dot further need a Key that provides C) where $h \leq j \leq i$ in the list of active arcs L, create a new active arc of the form $[A \rightarrow X_1...C_k\bullet...X_m](p_h, p_{i+1})$ (one with the dot advanced over C). If we later intend to recover the parses that have been produced, we must associate a backpointer α with each RHS constituent so that our arc now has the form $[A \rightarrow X_1(\alpha_1)...C_k(\alpha_k)\bullet...X_m](p_h, p_{i+1})$. Thus, we set α_k equal to $C(p_j, p_{i+1})$. (Note that since we are not yet doing any ambiguity packing there is a one-to-one correspondence between RHS symbols and backpointers).

6. **Add LHS constituents of newly completed rules to the agenda**
   (Without ambiguity packing nor target side packing)
   For all active arcs of the form $[A \rightarrow X_1...C\bullet](p_j, p_{i+1})$ (those arcs that have been completed), add a new key to the agenda of the form $[A \rightarrow X_1...C](p_j, p_{i+1})$.

7. **Add the key to the chart**
   Record that this key is a proven constituent for this input sequence.

8. **Test for termination conditions**
   If the key is $S(1, n)$, the input is grammatical and we can stop now if we only seek one grammatical parse. If the agenda is not empty, go to step 3. If there are more input tokens, go to step 2.

2.2 Local Ambiguity Packing

As chart parsing is a dynamic programming algorithm, it allows us to reduce an otherwise exponential task to $O(n^3)$ time. Ambiguity packing is one key way in which a chart parser can reduce the time to explore this huge space; multiple ways of deriving the same parse structure are represented by a single Key. Thus, we will use exactly one key to represent each source LHS covering a unique span.

More formally, if we wish to perform ambiguity packing, we must modify the above algorithm's step 6 as follows:

6. **Add LHS constituents of newly completed rules to the agenda**
(With ambiguity packing, without target side packing)
For all active arcs of the form $[A \rightarrow X_1...C\bullet](p_j, p_{i+1})$ (those arcs that have been completed), *add a new key to the agenda of the form* $A(p_j, p_{i+1})$.

2.3 Scoring

Another feature of Treegraft is a simple scoring mechanism that allows it to process stochastic grammars. Without ambiguity packing, implementing scoring would be nearly trivial since each Key in the Chart would correspond to a single (partial) parse. Thus, an aggregate score could be obtained by adding log probabilities up the tree. However, ambiguity packing complicates this only slightly. Now, instead of being able to score each Key, we must first unpack the ambiguities into individual parses before summing the log probabilities of all rules applied to form the derivation.

$$score(parse) = \sum_{rule \in derivation(parse)} \log P(rule)$$

Using a single rule probability is, of course, a very simple case of parse scoring. In the future, Treegraft can easily be modified to support multiple feature scores for each parse. Also, note that it is still possible to assign scores to individual Keys, but they would have to report information about groups of parses (e.g. the score of the best derivation rooted at the current key or the total probability mass of all derivations rooted at the current key).

2.4 Transduction

So far, I have discussed chart parsing only in a monolingual context. However, a few modifications must be made to allow the chart parser to act as a transduction machine from some "source language" into a "target language" (note that in the case of paraphrasing both the source and target languages might be the same). Broadly, these modifications fall into three categories:

1. ensuring that the grammatical constraints of the target RHS are satisfied (i.e. that the target tree is grammatical)
2. dealing with the increased ambiguity with the new possibility of having multiple target LHS's and RHS's for a single source LHS
3. generating the target language parse trees and strings after the source has been built

For the sake of simplicity, I will discuss the case of transduction with a Synchronous Context-Free Grammar (SCFG), since it is the formalism that I implemented in this project; however, the concepts discussed below can easily be extended to other formalisms such as Tree Substitution Grammar.

2.4.1 Packing Target Sides

The above algorithm can also result in having more active arcs than are necessary when we consider that now we can have multiple target sides (both target LHS's and target RHS's) for a single source side. This can result in having arcs that are redundant on the source side, and, until the arcs are completed, there is no guarantee they will ever lead to a valid target-side node. Thus, we wish to pack these arcs as well, especially since a large number of them will be created during parsing.

We modify the above algorithm's step 4 and 6 as follows:

4. **Add all grammar rules that start with the Key as active arcs**
For each grammar rule R_k of the form $[A \rightarrow CX_1...X_m]$ (those that begins with the key C that we just got from the agenda), create an active arc $[A \rightarrow \bullet CX_1...X_m](p_j, p_j)$ and, iff such an arc is not already in the list of active arcs L, append it to L *and append a pointer to the rule* R_k *to the arc's list of transduction rules* ρ*. If such an arc* A *already exists in* L*, append* R_k *to* ρ_A*.*

6. **Add LHS constituents of newly completed rules to the agenda**
(With ambiguity packing and target side packing)
For all active arcs of the form $[A \rightarrow X_1...C\bullet](p_j, p_{i+1})$ (those arcs that have been completed), *create a new key of the form* $A(p_j, p_{i+1})$*. If such a key already exists in the agenda or chart, call it* B *otherwise call our new key* B*. Append a pointer to the completed active arc* A *to* B*'s list of active arcs .*

The addition of these rule backpointers allow us to later transduce in a one-to-many fashion from the source-side LHS and RHS to the target-side LHS and RHS.

2.4.2 Checking Constraints of Transduction Rules

For a SCFG, we must ensure that target-side RHS non-terminals match the target-side LHS of the key that is attempting to extend a rule since we allow the source and target LHS non-terminal symbols to be different in the grammar. More formally, we check the requirements of the target RHS of a SCFG by modifying step 5 in the above algorithm (cumulative with the changes made for dealing with ambiguous backpointers):

5. **Extend any existing active arcs with the key**

(With target side packing and transduction constraints)
For all active arcs of the form $[A \rightarrow X_1... \bullet C_k...X_m](p_h, p_j)$ (those that to advance their dot further need a Key that provides C) where $h \leq j \leq i$ in the list of active arcs L: *If* C *is a non-terminal, ensure that the target-side RHS symbol* T *aligned to the source-side RHS symbol* C *matches the target-side LHS for the current Key; if it does not, do not proceed any further with this step.* Otherwise, create a new active arc of the form $[A \rightarrow X_1...C_k \bullet ...X_m](p_h, p_{i+1})$ (one with the dot advanced over C). If such an arc does not already exist in L, add this arc and call it x, otherwise call the existing arc of this form x. If we later intend to recover the parses that have been

produced, we must associate a backpointer α with each RHS constituent so that our arc now has the form $[A \rightarrow X_1(\alpha_1)...C_k(\alpha_k) \bullet ...X_m](p_h, p_{i+1})$. Thus, we set α_k equal to $C(p_j, p_{i+1})$. (Note that since we are not yet doing any ambiguity packing there is a one-to-one correspondence between RHS symbols and backpointers).

This preemptive check will prevent many extraneous active arcs from being created, but it still does not guarantee that the target side trees will be corrext. In addition to this preemptive check, we must re-check that the target-side non-terminals match as we transduce using each rule. This process is described in the following section.

2.4.3 Transducing into the Target-side Parse Forest

For transducing the source-side parse forest (also called the Chart or source hypergraph) to the target-side parse forest, we add an additional step to be executed for each Key C_i in the Chart:

9. **Create the target-side parse forest**
For each source-side Key $C(p_v, p_w)$, for each active arc $[A \rightarrow X_1(\alpha_1)...X_m(\alpha_m)](p_v, p_w)$ packed within that C, for each rule R packed within A, create a new target-side Key $[T_i \rightarrow Y_1(\beta_1)...Y_q(\beta_q)](p_v, p_w)$ (recall that α and β are backpointers). Now, set the non-terminal type of T_i to the target-side LHS of the rule R. If there already exists a target-side key of the form $T(p_v, p_w)$ (one with the same target-side non-terminal symbol and the same source-side span), then call it t, otherwise call our newly created target-side key t. Next, we populate t's list of target backpointers β, in which each element corresponds to one target RHS symbol. For each terminal a_j at the jth position in the target RHS of the rule R, assign a_j to β_j. For each non-terminal symbol B_j as the jth position of the target RHS, find the source-side non-terminal X_k to which it is aligned and, using the rule R, ensure that the target-side RHS symbol T aligned to the source-side RHS symbol C matches the target-side LHS for the current Key; if it does not, do not proceed any further with this iteration for R, discarding T_i if it does not have backpointers $\beta_g \forall 1 \leq g \leq q$. Otherwise, for B_j's source-side backpointer α_k (a pointer to another source Key), store a pointer to the corresponding target-side Key T_w as the target-side backpointer β_w.

We can now extract the individual parses from this target-side parse forest via ambiguity unpacking just as we would for the source-side parse forest. As an aside, we can see from this algorithm that source-side non-terminals must always be aligned to target-side non-terminals and vice versa, but terminals should never be aligned to anything.

3 Software Engineering

In the spirit of the hands-on experience that this NLP Lab Project is intended to provide, I briefly leave the context of research and touch on the more practical aspects of how Treegraft was implemented.

3.1 Implementation

Treegraft is implemented in pure Java and its source code is available at code.google.com, released under a BSD license[2]. During development Google's SVN code repository was used for version control so that any disastrous changes or loss of data could be recovered. All source code can be accessed from a web interface at Treegraft's Google Code homepage[3].

3.2 Documentation

To document the contract each component of the system fulfills, I used Java's own JavaDoc tool[4], which places the documentation alongside the code it refers to. This documentation includes high-level comments for every class, method, parameter, and return type. The resulting documentation can be on Treegraft's JavaDoc API page[5].

3.3 Object Oriented Modules

One consideration that was on my mind while developing Treegraft was that it should be easily extensible and robust to future changes in our approach; simply put, I sought to design a research tool rather than just an implementation of a particular idea. This consideration is implemented using Java generics and interfaces. For readers familiar with C++, generics are roughly equivalent to C++ templates. Interfaces are roughly equivalent to C++ abstract classes in which a set of methods and their expected input and output is well-defined (and well-documented), but multiple particular implementations are allowed; the code that references these interfaces is agnostic to the particulars of each interface implementation. For example, Treegraft has a "Rule" interface, which is implemented by both a monolingual CFG rule class and a synchronous CFG rule class. Their construction is such that a Tree Substitution Grammar (TSG) rule implementation should be relatively straightforward. Another instance of interfaces in Treegraft is the "Scorer," which allows for multiple scoring methods (with various feature sets) to be independently developed and then tested using the same framework.

3.4 JUnit

To test the formal correctness (discussed further in Section 4), I used JUnit[6], a Java unit testing framework. This allowed for the creation of an automated test suite in which known inputs can be fed to the system so that assertions about various aspects about actual versus expected program output. Feedback is provided in a simple pass or fail fashion with a detailed error trace when an assertion fails. In this way, we avoid

[2] http://en.wikipedia.org/wiki/BSD_license
[3] http://code.google.com/p/treegraft/
[4] http://java.sun.com/j2se/javadoc/
[5] http://treegraft.googlecode.com/svn/treegraft/java/doc/index.html
[6] http://www.junit.org/

both the time it takes a human to read over testing output, the chance for human error in comparisons, and the dread of having to spend one's time reading such outputs. In practice, this lead me to test my code much more frequently and develop with greater confidence.

3.5 Speed Considerations

Though a long-term goal of this code base includes time-expensive processes such as parsing of Tree Adjoining Grammars (TAG's), Java has still been shown to give reasonable runtimes for intensive NLP tasks including examples such as Marian Olteanu's Phramer[7], a MT decoder that is reportedly ten times faster than Moses or Pharaoh, and Joshua[8], Johns Hopkins's Reimplementation of the Hiero decoder.

3.6 Optimizations

In the world of algorithms, computer scientists often focus strongly on the worst-case or average case complexity of a program while throwing away the constants. However, in practice, these constants can be the different between 12 hours and 24 hours to get the results of an experiment. With this in mind, Treegraft makes heavy use of efficient hashing and efficient equality comparisons. The best example of this is in the handling of tokens. Rather than doing string comparisons during the parsing process, Treegraft instead hashes all unique input strings to unique integers so that all comparisons are fast integer operations.

3.7 Lessons Learned

Initially, I proposed that Treegraft should be written in C for the reason of speed discussed above. However, after developing a good deal of the project in C, several points became clear: 1) that the code was becoming very brittle with regard to being extensible during future research explorations and 2) that the large increase in development time was likely not worth the potential gains in system execution times.

For these reasons, I began developing the project and Java and found that my productivity increased by a very large factor. One of the biggest benefits of developing in Java has little to do with the language itself, but the development tools (e.g. the Eclipse IDE) available for it (granted, the power of Java development tools comes from the lack of a C/C++ style pre-processor, which enables tools to do deeper processing). These tools provide features such as code templates, automatic compilation and error analysis as-you-type, project-wide code refactoring, jump to method/class definition, and an integrated in-code-documentation to HTML generator. Also, the ease with which Java can dynamically link new code at runtime means that testing new research ideas does not involve long compilation times. Overall, the power of the development tools to

[7] http://www.phramer.org

[8] http://www.clsp.jhu.edu/wiki2/Joshua_Lab

protect programmers from themselves make the “typical” benefits of the Java language (cross-platform execution and detailed error messages for exceptions) only a secondary consideration.

4 Evaluation

The formal correctness of the parser was evaluated using the JUnit testing framework for the following two scenarios:

4.1 Correctness of Monolingual CFG Source Forest

To test the correctness of Treegraft's generated source forest (the Chart), I used the known grammar and gold standard from the NLP Lab's Chart Parser module[9]. This included 21 input sequences including 278 individual assertions for which keys should have been created.

For each of these inputs, I verified that:

- the correct number of Keys was created for each input
- each Key covered the correct span
- each Key had the correct non-terminal constituent type

4.2 Correctness of Transduction SCFG Target Partial Parses

To test the correctness of Treegraft's generated partial parses for each Key in the target forest, I used a subset of the grammar and gold standard from the NLP Lab's monolingual Chart Parser module. Due to time constraints, I used the first 10 input sequences, which contain a total of 63 Keys. For each of the source-side Keys (which have a one-to-one correspondence with target-side Keys, but a one-to-many relationship with target-side parses), I manually added the expected target-side partial parses including the expected score at each non-terminal node. Likewise, I augmented the monolingual grammar into a synchronous grammar by adding target-side constituents and alignments. Several potential points of failure were targeted by this SCFG grammar:

- ambiguous source LHS's for the same source terminal symbol (tests ambiguity packing)
- ambiguous target RHS's for the same source terminal symbol / source LHS pair
- insertion of lexical items on the target side
- reordering between the source and target non-terminals

[9] http://www.cs.cmu.edu/afs/cs.cmu.edu/project/cmt-55/lti/Lab/Modules/NLP-712/chart/index.html

- having a successful source-side parse that fails due to a candidate Key's target LHS not matching the target RHS item that is being triggered
- parse trees coming from the same Key having different scores
- the span of each Backpointer Key falling within the span of its parent Key

For each of these inputs, I verified that:

- the correct number of Keys was created for each input
- each Key covered the source correct span
- each Key had the correct source and target non-terminal constituent type
- each Key had the expected number of partial parses
- each partial parse matched the expected partial parse (including both the structure and its score at each non-terminal)

5 Future Work

5.1 Constraining the Search Space

Given that many research applications of Treegraft might involve very large search spaces, it follows that we might want to prune parts of this search space that do not appear to be useful so that the system's run time can be reduced. One natural solution to this is to use a traditional beam search, which keeps only the top k entries at each step in the parsing process based on the scores at that step. For slightly more programming effort, we could use Liang Huang's technique of Cube Pruning or Cube Pruning and achieve even greater speed gains.

5.2 Unification-Based Feature Constraints

Though Treegraft already includes many of the features of the Avenue Transfer Engine's parsing technology, it does not yet include unification-based feature constrains via peudo-unification. Though these constraints do not provide any additional generative power, they do allow for rules to be constrained such that they only apply in very specific circumstances. Linguistic theories such as Lexical Functional Grammar (LFG) have shown such unification-based feature constraints to capture many grammatical phenomena across languages. Therefore, it could be beneficial to support their use in Treegraft in the future.

5.3 Tree Substitution Grammars

Tree Substitution Grammar (TSG) is a subset of Tree Adjoining Grammar (TAG) without the adjunction operation. A Synchronous TSG (STSG) grammar provides us with strictly more strong generative power (though the weak generative power remains the same) than SCFG [7] since it provides more *tree relations*, pairs of source and target trees[8]. In the area of translation, preliminary results have shown gains using STSG as a tree-to-string transducer on 1-best source-side parses on a small test set[8].

5.4 Tree Adjoining Grammars

In the future, I would like to explore how Tree Adjoining Grammar can be used as a transduction mechanism between human languages, specifically in a Regular Form 2-Level TAG (RF-2LTAG) [2, 1]. By using a grammar formalism with strictly greater strong and weak generative power (in the synchronous case) than many other common formalisms, the functionality of those formalisms is still available to the grammar writer (whether it be a human or a machine). Finding an elegant syntax for exposing such features is yet another matter for follow-up work.

However, building a parser for RF-2LTAG would be non-trivial as it requires the construction of a meta-grammar[3, 4]. Still, many examples of natural language violating the iso-morphic constraints imposed by lesser grammar formalisms have been discussed[5] though alternative methods of grouping tree structures for achieving non-isomorphism have been proposed[6].

6 Conclusion

In this report, I have presented Treegraft, a stochastic transduction chart parser. Treegraft will serve as a research tool and enable future explorations of issues including the strong generative power of various types of grammars and their suitability of such grammars to applications such as paraphrasing and machine translation. These goals culminated in the creation of software deliverable: a stochastic transduction chart parser. Efforts were made to ensure that this software was modular enough to allow for the dramatic and demanding code extensions often required by research.

References

[1] David Chiang. Evaluation of Grammar Formalisms for Applications to Natural Language Processing and Biological Sequence Analysis. PhD thesis, University of Pennsylvania, 2004. Available from: http://www.isi.edu/~chiang/papers/thesis.pdf.

[2] David Chiang, William Schuler, and Mark Dras. Some remarks on an extension of synchronous tag. In International Workshop on Tree Adjoining Grammars and Related Frameworks (TAG+5), 2000. Available from: http://www.isi.edu/~chiang/papers/tag+5.pdf.

[3] Mark Dras. A meta-level grammar: Redefining synchronous tag for translation and paraphrase. In Association For Computational Linguistics, 1999. Available from: http://www.ics.mq.edu.au/~madras/papers/acl99v2.ps.

[4] Mark Dras, David Chiang, and William Schuler. A multi-level tag approach to dependency. In Workshop on Linguistic Theory and Grammar Implementation, 2000. Available from: http://www.ics.mq.edu.au/~madras/papers/mltag.ps.

[5] Mark Dras and Chung hye Han. Korean-english mt and s-tag. In International Workshop on Tree Adjoining Grammars and Related Frameworks (TAG+6), 2002. Available from: http://www.ics.mq.edu.au/~madras/papers/tag+6.pdf.

[6] Mark Dras and Chung hye Han. Non-contiguous tree parsing. In International Conference on Theoretical and Methodological Issues in Machine Translation, 2004. Available from: http://www.ics.mq.edu.au/~madras/papers/tmi04.pdf.

[7] Jason Eisner. Learning non-isomorphic tree mappings for machine translation. In Association for Computational Linguistics, 2003. Available from: www.aclweb.org/anthology-new/P/P03/P03-2041.pdf.

[8] Liang Huang, Kevin Knight, and Aravind Joshi. Statistical syntax-directed translation with extended domain of locality. In Association for Machine Translation in the Americas, 2006. Available from: www.cis.upenn.edu/~lhuang3/amta06-sdtedl.pdf.

[9] Alon Lavie, Katharina Probst, Erik Peterson, Stephan Vogel, Lori Levin, Ariadna Font-Llitjos, and Jaime Carbonell. A trainable transfer-based machine translation approach for languages with limited resources. In European Association for Machine Translation, 2002. Available from: http://www.cs.cmu.edu/~alavie/papers/EAMT-XFER-Apr04.pdf.