Efficient Matching using Overlay Transducers

Clemens Marschner*
Fast Search & Transfer
Rablstr. 26
D-81669 Munich
clemens@fast.no

Abstract

We present algorithms and data structures for pattern matching on large corpora that are annotated with linguistic features, such as part-of-speech tags or other syntactic or semantic annotations. The results suggest that the proposed data structures, *Overlay Transducers*, form an effective filter on the search space of non-deterministic transducers as used for Local Grammar matching, reducing the number of comparisons per input token to roughly the number of feature names being used in the search patterns, independent of the number of search patterns in total.

Keywords

Overlay Transducers, Local Grammars, Finite-State Methods, Annotation Graphs

1 Introduction

Local Grammars are a way to describe language constructs by means of (possibly nested) syntax diagrams that may also contain output symbols. They have been used for describing the grammar of French [3], for extracting locations from German texts [7], for extracting biographical relations [2], for rule-based tagging [12], and much more. For linguistic applications they are the natural extension of regular expressions (as used in programming languages) that can take not only input characters, but a wide variety of linguistic features of the input into account.

Current implementations of this formalism, like the Unitex system[9], transform these grammars into transducers called (Extended) Recursive Transition Networks (RTNs)[14] and are used on preprocessed text whose set of input symbols (the text dictionary) is known. However, when these grammars are to be applied to very large, web-scale corpora, or to corpora of unknown size like streams, the input alphabet (in the form of tokens and feature values) cannot be enumerated, which presents a specific challenge for matching algorithms. We have developed a state-of-the-art matcher, henceforth called "LG-Matcher", for a web-scale search engine. In this context, the amount of analysis that can be done is usually tightly bounded by the latency of a document becoming searchable and translates directly into hardware costs, so efficient algorithms are paramount.

As RTNs in general are non-deterministic, the search space can become quite large, and grows significantly as the grammars evolve. Dealing with the sources of non-determinism is the first challenge. The use of output symbols and subgraphs limits the ability to determinize the transducers. On the other hand, most of the match attempts end up being unsuccessful, as most grammars cover only a small portion of the input text.

Connected to this is a second challenge, given by the demand for supporting larger and larger grammars. The unlimited input alphabet means that for an input search patterns can usually only be evaluated linearly, one after the other. In contrast, deterministic transducers, as used for lexicon lookup, can be traversed in time independent of their size.

Most of the match attempts occur near the initial state of the grammars, so optimising here promises a high return with little effort. In [6] we presented a filter method called "Prefix Overlay Transducer" (POT) which reduced the search space of the LG-Matcher by mapping finite sub sequences of input patterns to a set of paths for the LG-Matcher, similar to adding an overlay to the initial state of the grammar. By default, the POT-Matcher would first filter input sequences and only activate the LG-Matcher when the prefix of the grammar matched.

In this paper we generalize from this method by adding the ability to attach overlay transducers not only to the initial state, but to all states of the input grammar. We have implemented a significantly changed matching algorithm and tested it with various parameters. The experiments show that in contrast to the LG-Matcher we achieve elegant scalability and a significant speed-up compared to the prefix overlay transducers. In practice, growing a real-life local grammar from 1 to 43 sub graphs extends the running time only 0.57 times, allowing for searching several MB of richly annotated input text per second with the power of context-free grammars.

One might be tempted to restrict the expressiveness of the formalism, e.g. by abandoning recursion, or other means that transform the input grammars. In this paper, we argue that these kinds of transformations can be avoided, while at the same time the expressiveness of the formalism is not harmed at all.

The remainder of this document is organized as follows: We first give a brief overview of local grammars in their representation as RTNs and describe how the

*Formerly at CIS Center for Information and Speech Processing, University of Munich, Germany.

LG-Matcher works. Then we present how RTNs are mapped to a representation from which the OTs are built, and discuss their properties. We continue with describing how the OT-Matching algorithm works and how it compares to the POT-Matcher from [6]. We close with giving experimental results on a variety of practical grammars.

2 Local Grammars and Recursive Transition Networks

Local Grammars are a formalism for representing local syntactic or semantic rules. They are used for pattern matching on input text which may contain linguistic annotations.

An example of a local grammar is given in Fig. 1, which shows how a fraction of a grammar recognizing locatives. Fig. 3 shows the RTN that was generated from Fig. 1.

Input Texts. To facilitate the understanding of how Local Grammars work, we represent text as a sequence of possibly overlapping boolean or nominal *annotations* that bear syntactic or semantic *features*.

Definition 2.1 The input to the local grammar matcher consists of (1) a flat sequence of textual units (characters or tokens) and their position $0, ..., n$ that specify their boundaries, and (2) a (possibly empty) set of annotations. Every annotation is specified by an interval $[i, j]$, $0 \leq i < j \leq n$ and a set of key-value pairs bearing (nominal or boolean) features.

An example for the sentence "He lives in New York" is shown in Fig. 2. A feature would be the surface form of a token, or its part-of-speech (POS) tag. Annotations can span more than one textual unit, and they can overlap: The "location" feature could span both the phrase "New York" and "New York City". Annotations can be generated by linguistic modules such as gazetteers, grammars, or taggers. The set of feature names is usually small. E.g. for the rest of this paper we assume that only the features "surface", "lowercase", "capital", "pos" (part-of-speech) and "semantic" exist. We further assume that the result of this processing need not be unambiguous – contradictory annotations may still exist. We further assume that whitespace is simply ignored – to be precise we could think of adding transitions to all automata, matching zero or more whitespace between annotations. The result is equivalent to that of a directed *annotation graph*, with token boundaries as nodes and annotations as edges.

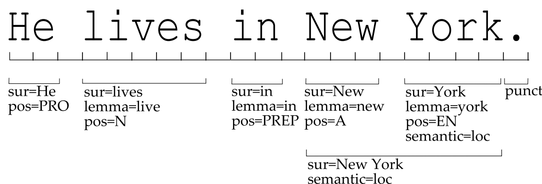

Fig. 2: *The sentence "He lives in New York" represented as a set of feature regions.*

Recursive Transition Networks. Each Local Grammar has an equivalent recursive transition network (RTN). In the specific sense of this paper they are defined as follows:

Definition 2.2 An RTN consists of a set of (named) graphs G with name $N(g)$ with a specific top graph g_0.

Each graph g consists of the tuple $\langle Q, \Sigma, \Gamma, q_0, F, \delta, \rangle$, where Q is the set of states, Σ is the input alphabet, Γ is the output alphabet, $q_0 \in Q$ is a special initial state, $F \subseteq Q$ is the set of final states, and $\delta \subset Q \times \{\Sigma \cup \epsilon \cup N(G)\} \times \{\Gamma \cup \epsilon\} \times Q$ is the transition function.

The input symbols of RTNs consist of search patterns which are composed from expressions of the kind "$feature = value$" and are closed under conjunction (&) and negation (!). As output, we only allow terms of the form "[$feature$" and "]" which specify a new annotation with a "semantic" feature to be created. The new annotation will span from the start of where the transition containing "[$feature$" has matched to the end of where the transition containing "]" has matched.

Disjunction (|) and concatenation (∘) can be used in Local Grammars as well, but in RTNs they are expressed through transitions being put in parallel or in sequence and are therefore not part of the search pattern algebra. Since transducers in general cannot be determinized, they temporarily need to be viewed as finite-state automata: Subgraph calls are treated like input symbols, and a single symbol is formed from an input–output pair. Each graph of a grammar is treated this way, and is determinized and minimized using an algorithm like Hopcroft minimization[4].

Matching With RTNs. After determinization, the RTN will contain different input and output symbols on all transitions leaving a state, but they may still select overlapping sets of the input, because the alphabet of the input is unknown. Furthermore, not all ϵ-transitions may have been removed, as they might still contain different output symbols. Finally, determinization takes place on the graph level only, and subgraphs may also match overlapping sets of words from the input. As a result, the matcher will be nondeterministic.

The LG-Matcher will therefore perform, for each input unit, a depth-first search through the transducer. Outgoing transitions at a state are ordered. For each transition, the search pattern is evaluated against all annotations starting at a given position. If the evaluation is successful, the parser needs to move to the end of the annotation and to the target state of the transition. If it is not successful, it will first try to match all further transitions at the state with all annotations at the position in the text, and then track back to the previous state. If it reaches a final state, it reports the path through the input annotations in connection with the path through the grammar. ϵ-transitions are always followed, regardless of the input. Calling and leaving a subgraph can be seen as special cases of ϵ-transitions, as described in Fig. 4.

Definition 2.3 A *path* through the RTN is a list p of numbers that describe which of the (ordered) transitions were followed from the initial state. During

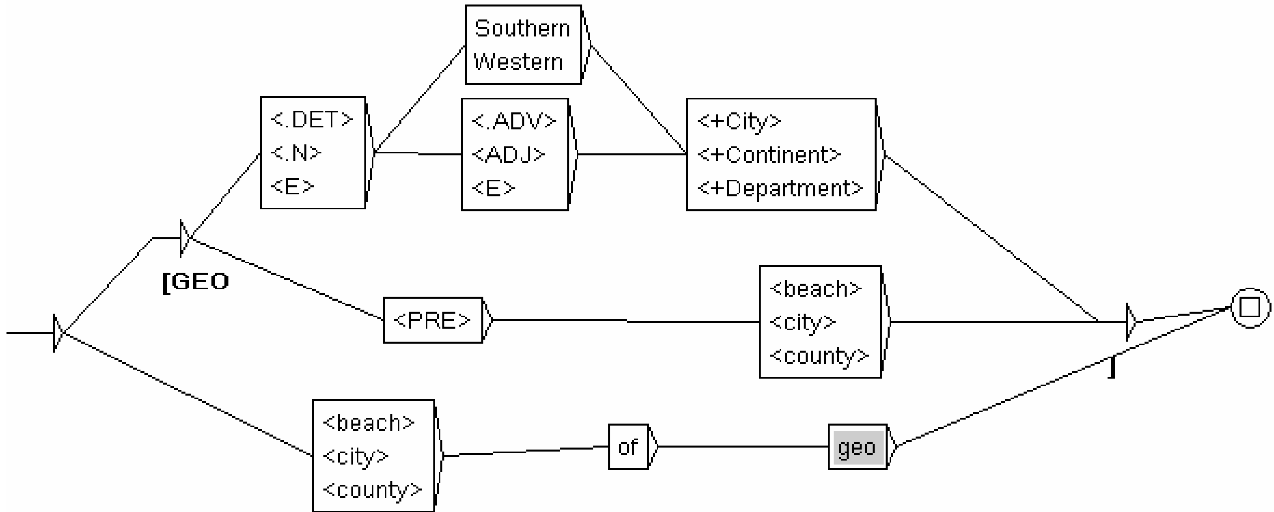

Fig. 1: *A graph of a Local Grammar in "Unitex" syntax.* $\langle E \rangle$ *denotes an* ϵ*-transition,* $\langle beach \rangle$, $\langle .DET \rangle$ *and* $\langle +City \rangle$ *define conditions for lemmas, part-of-speech tags, and semantic codes, respectively.* $\langle PRE \rangle$ *searches for capitalized words, the grey box forms a non-terminal calling the "geo" graph, and the terms* $[GEO$ *and* $]$ *are output symbols attached to an empty* ($\langle E \rangle$) *transition.*

matching, each entry of p is related to 1 or 0 annotations, depending on whether the transition is input-consuming or not. The sequence of annotations is continuous and covers a region of the input. Together they form the *control state* of the LG-matcher.

The approach contains three causes of inefficiencies: (1) non-deterministic parts, (2) infinite alphabet and, as a consequence, (3) a potentially large temporary search space.

Non-determistic parts, introduced by subgraph calls, output symbols, or ϵ-transitions, cause that the same expression needs to be evaluated more than once per input unit. Determining which transitions to follow requires the evaluation of all search patterns, which takes $O(n)$ comparisons per state, where n is the number of outgoing transitions. This may require several hundreds or even thousands of evaluations per input unit in a practical grammar, which is usually not acceptable. At last, while the search space may initially be prohibitively large, it usually narrows down significantly when paths are followed. In most cases, no final state will be reached, as Local Grammars usually cover only a small portion of the input text. It therefore makes sense to move costly computations to a very late time where the search space is already small.

3 Optimizing RTNs Using Lexical Transducers

To overcome these three problems, the central idea of this paper is to use a *lexical transducer* to map a string representation of the search patterns to a set of paths in the RTN. These are usually used to represent large dictionaries.

Definition 3.1 A lexical transducer is a deterministic, acyclic, minimal transducer which can contain output symbols only at final states and whose input symbols are single characters.

Building the OT. In a pre-processing step for the RTN we translate the search patterns that label outgoing transitions of the initial state and its successors into a string representation. These string representations are compiled into a lexical transducer, called *overlay transducer*. For matching, input units and annotations are also treated as strings and are first processed by the overlay transducer. When reaching a final state, the output of the transducer defines a set of paths of the RTN. Starting from the end points of these paths we continue with the usual RTN matching.

This is efficient as for a deterministic automaton, if the alphabet is known in advance, its symbols can be enumerated, and an array can be used for each state, whose indices represent the symbols, and whose entries signify whether a transition exists or not. This means, for a given input character, the matching outgoing transition can be found in $O(1)$, regardless of the size of the automaton[5], in contrast to O(n) with RTNs.

Since cycles or recursion cannot be represented in lexical transducers, only the prefix of all words of the language recognized by the RTN will be represented. The algorithm is shown in Algorithms 1 and 2. It is guaranteed to terminate, as three constraints may be specified: The maximum number of states the transducer is allowed to have, the number of cycles that may be unfolded, and the maximum "level" which specifies the number of input-consuming transitions to be followed from the initial state. In [6] we showed that the "level" forms the most effective of these criteria, and we will limit further discussion to this parameter.

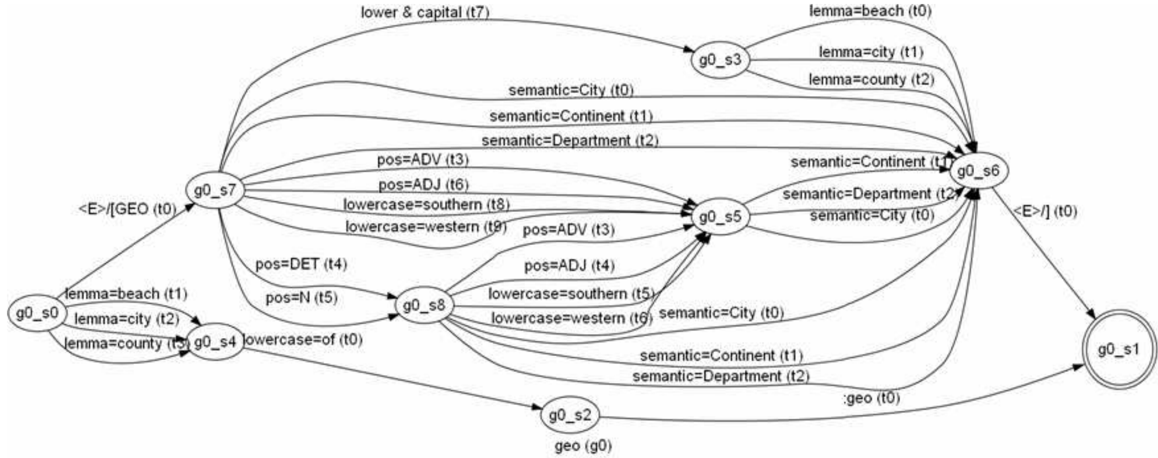

Fig. 3: *A Recursive Transition Network, generated from the graph in Fig. 1. Boolean features are abbreviated. Transitions at each state are ordered (tn).*

A level of 2 means that each entry of the transducer consists of two concatenated expressions of the form "$feature1 = value1.feature2 = value2$".

The question remains how expressions can be represented as strings in a way that the transducer can be traversed efficiently. We distinguish between the set R of representable and N of non-representable expressions. In the simplest case only transitions of the type "$feature = value$" are in R and the set of features is known and is mapped to a set of characters, while other types of expressions like "$feature \neq value$" are not[1]. The overlay maps string representations of finite RTN paths, consisting of a sequence of concatenated *keys* being computed from the patterns, to their numerical representations. Non-representable transitions are represented using the key "+", which means that the transition at the end of the path needs to be evaluated "by hand". For ϵ-transitions (which includes sub graph calls and -returns) the key is empty. Patterns of the form "feature=value" are represented by appending a delimiter symbol "$", e.g. "*pos=N$*", or even shorter using single-character feature identifiers such as "*pN$* when the set of features is known and small.

The OT is built through a breadth-first search through the RTN, which allows for applying any of the mentioned termination criteria. The overlay transducer is built in parallel to the traversal. First assume that subgraphs are not present. Then, for each state s in the RTN, we first seek the set of ϵ-reachable states from s and remember the path to these states, respectively. The set of non-ϵ-transitions leaving these states forms a candidate set for inclusion in the OT. Candidates from R are added directly to the OT. Candidates from N or ones that are left when one of the termination conditions are reached are added the list of "remaining" paths at the "+" symbol in a finalization step.

Algorithm 1 assumes that the OT can be built breadth-first in main memory. However, by changing $finalize()$ and $addPlusTransition()$ to write the whole key of the OT into a sink (like a file) it would be possible to use an external algorithm as well, though then it becomes harder to use a termination criterion like "maximum number of states" in a way that it remains guaranteed that all- or none of the outgoing transitions of a state are covered.

[1] In fact it is also possible to represent expressions containing "&" operators as well, as long as negation is not present.

max level 1		max level 2			
pA$	(0),(2,0)	pA$	(2,0,-)	pAsC	(2,0,0)
pV$	(1)	pAsb	(2,0,2)	pA$+	(2,0,1)
		pAsc	(0,0)	pVsc	(1,0)

Table 1: *Two prefix overlay transducers for Fig. 4. The entries map string representations of the search patterns to paths in the RTN.*

Calls to subgraphs need to be treated as if these transitions were replaced by ϵ-transitions, one connecting the source state to the initial state of the subgraph, and one for each final state to the target state of the subgraph call. In the path the transitions returning from the subgraph are denoted with "-". See the dotted lines in Fig. 4, which replace the dashed line labeled ":sub". An example of OTs for max level 1 and 2 for this the initial state of this graph is shown in table 1.

Properties of Overlay Transducers. As can be seen in Fig. 5, the resulting OT consists of alternating parts, (A) and (B): a state whose outgoing transitions enumerates the feature names, and a following subtrie which represents the set of possible feature values followed by delimiter symbols.

Final states can occur in three places: leaf nodes (1), inner nodes (2), or following "+" (3), the special symbol for "non-representable" transitions. In cases 1 and 2, the set of paths describe the *states* of the RTN following the last transition, while in case 3 they describe the last *transition*. For inner nodes, the construction algorithm guarantees that *all* outgoing transitions of the states represented by these nodes *are contained in the OT*. Likewise, of course, in case 1 none of the outgoing transitions is represented.

Algorithm 1 buildOT(ot: Automaton, lg: LocalGrammar, maxStates, maxCycles, maxLevel)

```
1: bftQueue ← {⟨otState : initial(ot), lgState : initial(lg), path : ∅, level : 1⟩}
2: {a possibly long FIFO breadth-first traversal queue of records}
3: while bftQueue is not empty do
4:   e ← dequeue(bftQueue)
5:   for all c in candidates(e.otState, e.lgState, e.path, e.level) do
6:     if c.lgTrans is representable in overlay transducer then
7:       newOtState = insert(c.otState, convertToString(c.lgTrans))
8:       if (c.lgState is final and c.stack is empty) or cyclesExceeded(c.path, maxCycles) then
9:         finalize(otState, c.path) {sets state final and adds path}
10:      else
11:        enqueue(bftQueue, ⟨newOtState, c.path, c.lgState, c.level, c.stack⟩)
12:    else
13:      addPlusTransition(c.otState, c.path) {not representable}
14:    if max states exceeded then
15:      generate remaining candidates and add all as "+" transitions
```

Algorithm 2 candidates(otState, path, lgState, stack, level)

```
1: {performs ε-removal and generates candidates for one otState}
2: results ← ∅
3: if lgState is final then
4:   {follow transition to upper graph, if necessary}
5:   if stack is not empty then
6:     ⟨topstack, topstate⟩ ← pop(stack)
7:     results ← candidates(otState, path∘ "-" , topstate, topstack, level)
8:   else
9:     {we have reached a final state. Add the path, as a side-effect.}
10:    finalize(otState, path) {may already be final}
11: for i = 0...numOutgoing(lgState) do
12:   tr ← outgoing(lgState, i)
13:   if tr is input consuming then
14:     results ← results ∪ {⟨otState, path ∘ i, tr, target(tr), stack, level + 1⟩}
15:   else
16:     if tr is subgraph call then
17:       {follow "ε" transition down, but remember where we came from.}
18:       subinitial ← initial(tr)
19:       substack ← push(stack, target(outgoing(lgState, i)))
20:       results ← results ∪ candidates(otState, path ∘ i, subinitial, substack, level)
21:     else
22:       targetState = target(tr) {skip epsilon transition}
23:       results ← results ∪ candidates(otState, path ∘ i, targetState, stack, level)
   return results
```

Matching With Prefix Overlay Transducers. First, assume that the generated transducer is used only as an overlay to the initial state, as depicted in Fig. 6. When reaching a final state, the POT-Matcher would build the control state of the LG-Matcher and then run this matcher to evaluate the rest of the grammar. The overlay transducers are still non-deterministic at the states that enumerate possible input features, so the matching algorithm again needs to follow all transitions where annotations, starting at a given position, exist that contain the feature that is represented at this transition. The feature value can then be evaluated in time linear to its length, but independently of the number of feature values contained in the transducer.

As a result of stepping through the overlay transducer, a set of paths is generated that describe the state from which to procees within the LG-Matcher. Furthermore, final states in the OT can be reached by different annotation paths. Given the information which RTN-transition is input-consuming or not, the OT-Matcher can build the control state of the LG-Matcher.

To avoid that the LG-Matcher evaluates transitions that are already covered by the OT, it needs to be run in one of three modes: In case (1) mentioned above (leaf node), check for final state, then let the LG-Matcher follow all outgoing transitions. For case (2), just check for final state and return, as the OT is guaranteed to include all outgoing transitions. For case (3), the path represents a transition with a search pattern that still needs to be evaluated. Evaluate it and, if it matches, proceed like in case (1). When backtracking reaches the point where the LG-Matcher started, remove the remaining path from the stack and return to the POT-Matcher.

"Generalized" Overlay Transducers. Using the POT-Matcher as an overlay to the LG-Matcher

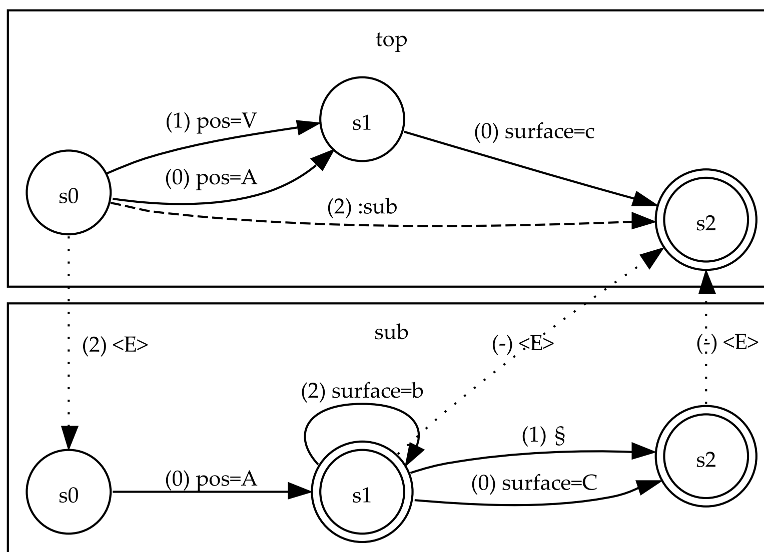

Fig. 4: *Example of different phenomena encountered in RTNs during OT construction. The picture shows two graphs, called "top" and "sub", where top calls sub on the transition denoted ": sub". During the construction this transition is replaced by the dashed ϵ-transitions. The result can be determinized, as both initial states of top and sub have a "pos = A" transition. sub contains a cycle of the form "sur = b *" that needs to be taken care of. It also contains a transition with an unknown operator "§" that cannot be represented in a OT. Transitions are assigned a number that denotes the local order at the respective state. Transitions returning from subgraphs receive the id "-".*

achieves good results with small- to medium-sized grammars, but it becomes less effective once the grammars grow and more time is spent in the "suffix" parts. Growing the POT to more than two or three levels is inefficient, since it usually grows exponentially with every level added, while its effect diminishes (except if the coverage of the grammar is really large). Therefore, we build overlay transducers for all states. This works equivalently as with the initial state, with two exceptions in connection with sub graphs: (1) When building an OT of a sub graph, in the general case one cannot follow a transition from a final state "up", because the sub graph may have been called from different parent graphs. (2) Some OTs may occur at final states, i.e. the root state of the OT may be final as well. This is different to the POT as the grammar compiler usually avoids that the generated grammar accepts the empty string.

If each generated key of the OT is prefixed by an identifier of the state it belongs to, all OTs of all states can be merged into a single transducer, and minimization can capture regularities in the grammar and achieve compression.

For matching, the original algorithm was turned "upside down" (see Fig. 7). The LG-Matcher is enhanced by the ability to let the OT-Matcher generate all possible paths for each encountered state, instead of simply iterating through its outgoing transitions. As a result, when reaching a final state in the top graph, the path through the local grammar will consist of snippets generated by the OT matcher. By marking up start and end of these snippets, the LG-Matcher is able to clear complete sub paths during backtracking, to avoid following paths that would be covered by the OT.

4 Evaluation

We have conducted several experiments using an implementation of the combined OT- and LG-Matchers. In the following, we used OTs of different sizes, written "ot x.y", where x denotes the level at the initial state, and y the one at all other states. A level of 0 means that no overlay is used. All experiments were conducted on an average Intel-laptop.

The first experiment measures the overhead of combining the two matchers (see Table 4), run on a 300000 token subset of the Reuters corpus (1,9 MB text, 9,58 MB with annotations). The left column shows runtimes for the LG-Matcher alone. In the right column an OT of level 1.0 is used. Very primitive patterns are used, one that doesn't match anything, one that matches all tokens, and so forth. The results indicate a slight constant overhead for switching back and forth, which gets a bit larger when more results are reported. As can also be seen, a certain overhead is involved with reporting a match. This is different to using "pure" transducers that generate output for every encountered input.

In the second experiment, a grammar recognizing time adverbials (like "in the first days of September" or "on May 20") is applied to a 9.4 MB part of the Reuters corpus (1.5 Mill. tokens, 48.8 MB with annotations). It consists of 65 graphs with 1277 states and 4209 transitions and is largely lexicalized. 43 of these graphs are called from a top graph in a large "OR" construction. We created 7 new grammars that include 1, 7,..., 43 of these sub graphs and combined them with different overlay transducers (see Fig. 8). When using OTs of depth 2 or more, the performance is largely determined by the number of comparisons at

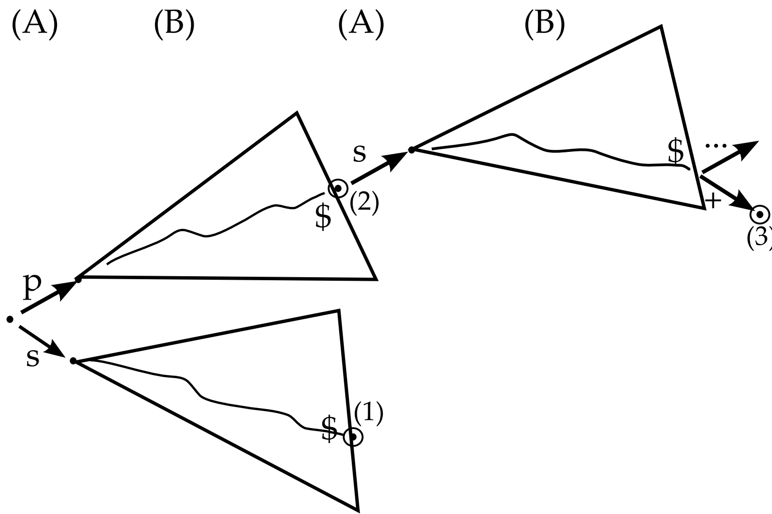

Fig. 5: *Schema of an OT, displayed as a trie. It consists of two parts: states whose outgoing transitions enumerate the features being looked at, and their arguments, represented through a sub-trie that can be traversed linearly. The sub trie is left through the end symbol denoted "$". The OT contains final states at three different places: (1) leaf nodes, (2) inner nodes, or (3), end nodes at "+" transitions, denoting the set of paths that end with non-representable transitions.*

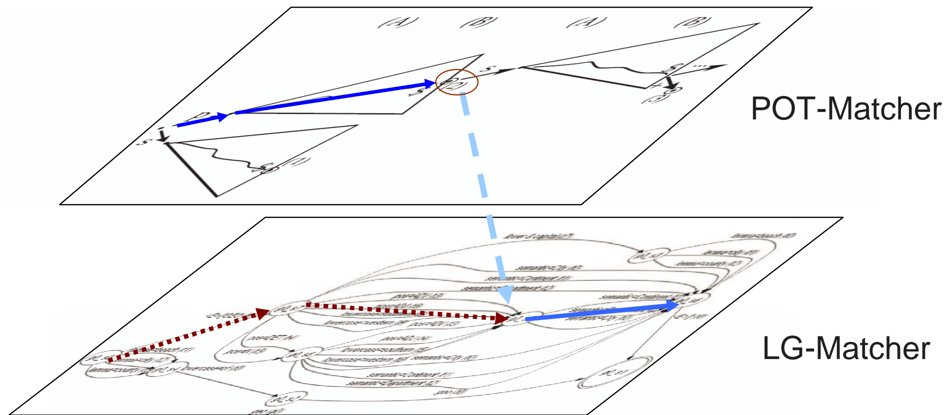

Fig. 6: *Schema of the POT-matcher. Solid lines indicate traversal through the POT. At a final state, the path in the local grammar is built (dotted lines) and matching continues using the LG-Matcher.*

the initial state. Scaling from 7 to 31 graphs leaves the runtimes practically constant. Graph 37 introduces a pattern that searches a different feature at the initial state and causes a sudden step upward, caused by the extra comparison that most be done at each input token.

The last experiment used the same grammar, but combined overlay transducers with a graph transformation algorithm called "Flattening" [10]. Graphs called by the top graph were embedded into the top graph. This was repeated a number of times, and these "flattened" graphs were combined with overlay transducers of different sizes. The results indicate that overlay transducers dominate the effect of flattening. The Flatten operation has the disadvantage that it can blow up the grammar once the costs of embedding a graph starts to dominate the gains of obtaining better determinization (which usually starts at flatten level 1 or 2). Besides, the original structure (and therefore information) is lost – a property which it shares with other transformations like weak Greibach normal form [8]. With overlay transducers it is not necessary to make these compromises.

graph	no OT	OT, level 1.0
no match	2.312	2.756
one match	2.211	2.554
some matches	2.885	2.840
match adverbs	2.585	3.078
match all tokens	2.899	3.513
match and annotate all tokens	4.172	4.990
Det-Adj*-N	2.763	3.256

Table 2: *Runtimes of the combined algorithms on a 300.000 token input using different trivial graphs and an overlay transducer of level 1.0.*

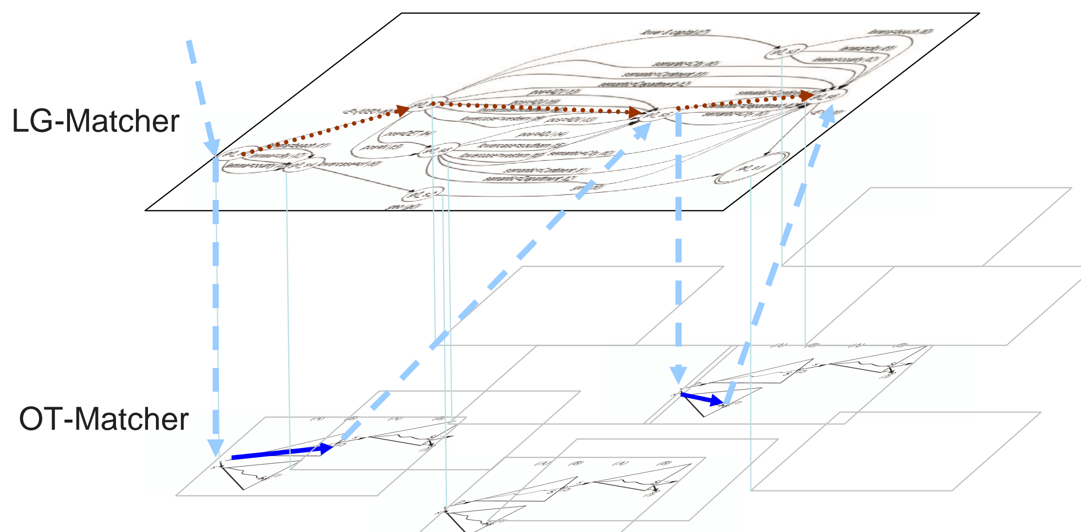

Fig. 7: *Schema of the OT-matcher. At each state, the OT returns paths that are followed in the grammar.*

5 Conclusion and Future Work

In this paper we have introduced overlay transducers as the guiding data structure for matching with non-deterministic, recursive and potentially large local grammars, in an attempt to combine the runtime advantages of lexical transducers with the power of context-free grammars. The results suggest that OTs can take over a large part of the matching process and are an effective filter on the search space of RTNs. Empirically the number of different features queried near the initial state has the largest influence on the runtime.

The method could be extended in several directions: First, it is important to maximize the class R of representable transitions. For complex expressions we are thinking of introducing a third class, whose R-part is evaluated using the OT, so that the number of times the N-part is evaluated is minimized. One guiding principle of this work is that it tries to move expensive operations to the back. Maintaining more complex feature output structures and unification [1] could benefit from this fact.

References

[1] Blanc, O., M. Constant. Lexicalization of grammars with parameterized graphs. In Proc. RANLP 2005, Borovets, Bulgaria, 117-121,

[2] Geierhos, M.: Grammatik der Menschenbezeichner in biographischen Kontexten. Master's Thesis, Center for Information and Speech Processing, Munich (2006)

[3] Gross, M.: Lexicon-Grammar And The Syntactic Analysis Of French. Proc. COLING 1984: 275–282

[4] Hopcroft, J.E., R. Motwani, J.D.Ullman: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, 2001.

[5] Liang, F.M.: Word Hy-phen-a-tion by Com-put-er. Ph.D. thesis, Stanford (1983)

[6] Marschner, C.: Efficiently Matching with Local Grammars Using Prefix Overlay Transducers. Proc. CIAA 2007, Prague, Czech Republic.

[7] Nagel, S.: An Ontology of German Place Names. Corela, Numéros spéciaux, Le traitement lexicographique des noms propres. (2005)

[8] Paumier, S.: Weak Greibach Normal of Recursive Transition Networks. Proc. Journées Montoises d'Informatique Théorique (2004)

[9] Paumier, S.: Manuel d'utilisation du logiciel Unitex. (2003)

[10] Paumier, S.: De la reconnaissance de formes linguistiques l'analyse syntaxique, Ph.D. thesis, Université de Marne-la-Valle (2003)

[11] Roche, E.: Parsing with Finite-State Transducers. In: Roche, E., Schabes, Y. (eds.): Finite-State Language Processing (1997) 241–282

[12] Roche, E., Y. Schabes: Deterministic Part-of-Speech Tagging with Finite-State Transducers. In: Roche, E., Schabes, Y. (eds.): ibd. 205–237

[13] Silberztein, M.: INTEX: A Corpus Processing System. COLING (1994) 579–583

[14] Woods, W. A.: Transition network grammars for natural language analysis. Comm. ACM **13** 10 (1970) 591–606

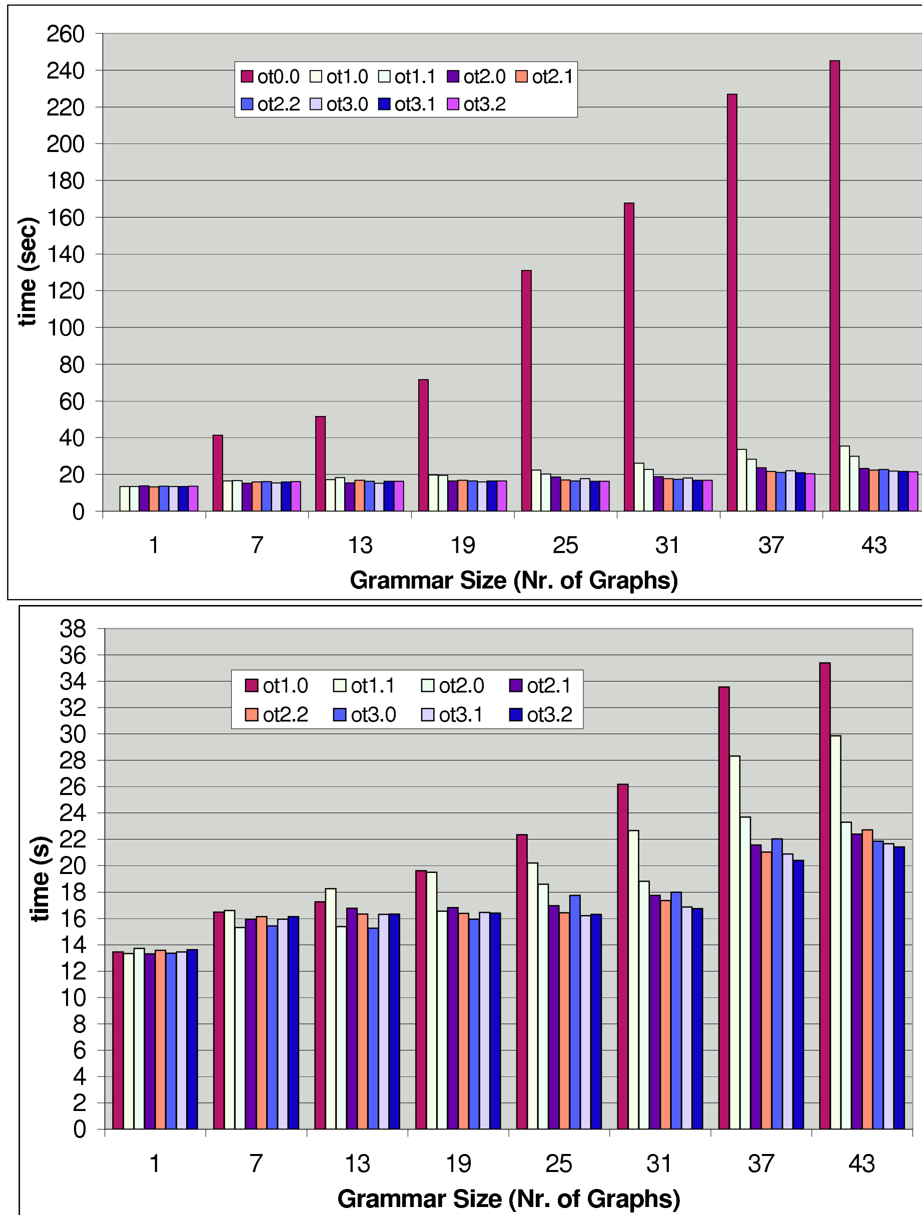

Fig. 8: *Performance comparison of different grammar and transducer sizes on 1.5 million tokens input text, with and without the original matching algorithm. E.g. "ot2.1" is the running time using the overlay transducer with depth 2 at the root state and depth 1 at all other states.*

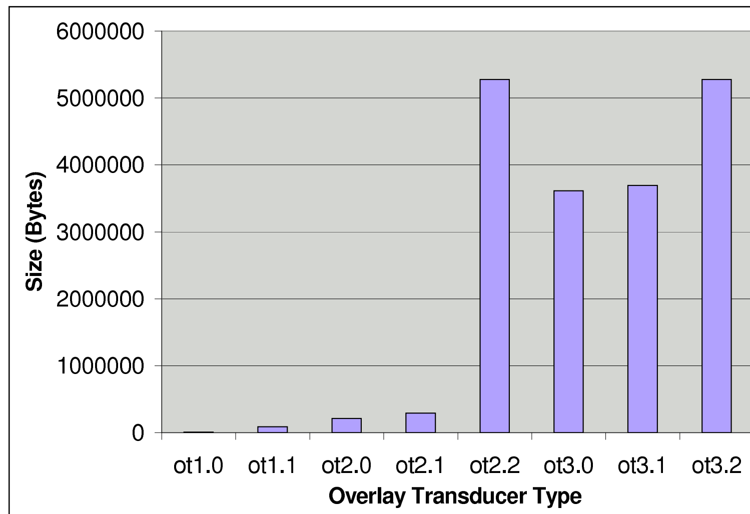

Fig. 9: *Sizes of the Overlay Transducers of Fig. 8, in Bytes.*

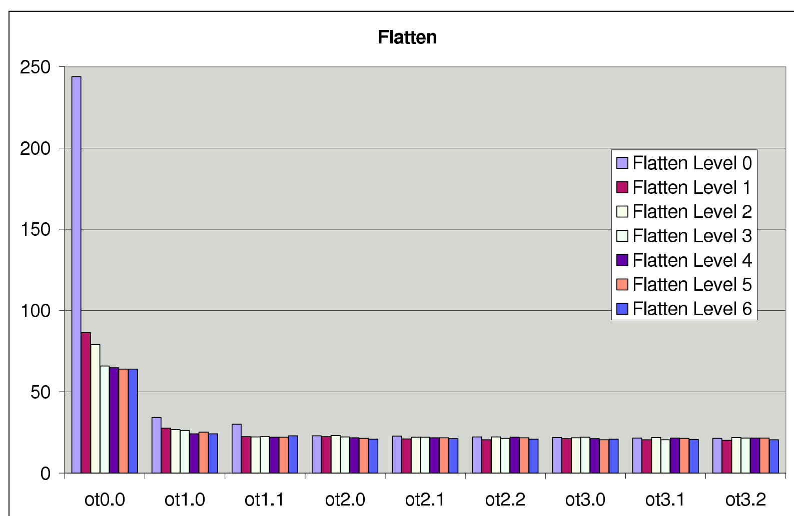

Fig. 10: *Effects on the runtime of the time adverbial graph when embedding subgraphs in a "Flatten" operation. At each iteration sub graphs are embedded into the top graph, and the result is determinized and minimized. Overlay transducers achieve a larger effect than the flatten operation and in most cases do not profit from a combination with the flatten operation.*