String Shuffling over a Gap between Parsing and Plan Recognition

John Maraist
Computer Science Department
University of Wisconsin-La Crosse

In the AAAI 2017 Workshop on Plan, Activity, and Intent Recognition, Feb. 4–5, San Francisco

Abstract

We propose a new probabilistic plan recognition algorithm YR based on an extension of Tomita's Generalized LR (GLR) parser for grammars enriched with the shuffle operator. YR significantly outperforms previous approaches based on top-down parsers, shows more consistent run times among similar libraries, and degrades more gracefully as plan library complexity increases. YR also lifts the restrictions on left-recursion imposed by approaches based on top-down parsing algorithms. We further propose that context-free shuffle grammars, more than traditional context-free grammars, should be seen as the appropriate analogue of HTN plan libraries in the correspondence of plan recognition and parsing.

1 Introduction

The connection between parsing and HTN plan recognition is well-established, but the mismatch between them is hard to ignore: plan recognition algorithms deal with multiple top-level goals, concurrent interleaving subgoals, and rich probability models unaddressed by parsing algorithms. So one can adopt a parsing algorithm directly, and severely restrict the plans and observations which can be recognized, or one can adopt an algorithm which bears only passing resemblance to known, proven parsers. Here we propose a new plan recognition algorithm at the same time as a new characterization of the relationship between parsing and plan recognition. Previous work imagined a link between plan recognition algorithms and parsing algorithms for context-free grammars (CFGs). However this choice of CFGs as an analogue is exactly the mismatch: the appropriate analogue is *context-free shuffle grammars* (CFSGs), CFGs enriched with a *shuffle operator*. Shuffle operations combine strings so that the order of symbols from each string is preserved, but interleaving of the shuffled strings is possible. For example, both m12np3r45 and mn123pr45 are shufflings of mnpr and 12345, but mp12nr345 is not because the n and p occur in a different order.

In this paper we show how a plan recognition algorithm based on CFSG parsing provides significant improvements over previous HTN plan recognition approaches. Much of the improvement derives from adapting LR parsing techniques. An algorithm like Yappr will start from an intended goal, first decompose it into subgoals, and then match expected against actual observations. Our algorithm YR avoids creating such separate internal explanations for a single partition of observations among distinct intentions, reducing the search space when considering each successive observation. Moreover, plan recognizers based on top-down parsers inherit the restrictions that these parsers place on their grammars. Transforming grammars/plan libraries around these restrictions (Geib and Goldman 2010, for example) can introduce an obfuscatory gap between the libraries designed by a user and those recognized by the system. Bottom-up parsers do have some restrictions, but include a much greater share of practical grammars.

Many earlier parsing-based approaches to plan recognition restricted partially-ordered plans, interleaving multiple intentions, and/or did not address relative probabilities of intended goals (Avrahami-Zilberbrand and Kaminka 2005; Bui, Venkatesh, and West 2002; Kaminka, Pynadath, and Tambe 2002; Kautz 1991; Vilain 1991; Pynadath and Wellman 2000; Huber, Durfee, and Wellman 1994). Geib and Goldman's PHATT (2009) was the first to impose no such restrictions; Yappr (Geib, Maraist, and Goldman 2008) subsequently refined PHATT for significant performance gains. Geib's ELEXIR (2009) offered further performance improvements but requires more detailed plans than simple HTNs to enable early commitment to goals. Ramírez and Geffner propose a different, domain-theoretic approach to plan recognition (2009) whose performance is comparable to Yappr's (Stoutenburg Tardieu 2015). More recently Mirsky and Gal proposed SLIM, which combines bottom-up and top-down techniques (2016).

In Section 2 we present background information on plan libraries, plan recognition, parsing and the shuffle operator. In Section 3 we present the YR plan recognition algorithm. In Section 4 we consider YR's theoretical and experimental performance. Finally in Section 5 we conclude.

2 Background

2.1 Plans and plan recognition

We consider this formal language of plans and plan libraries:

Definition 1 *A plan library is a tuple* (Σ, NT, R, P) *where* Σ *and* NT *are disjoint finite alphabets respectively of terminal* *events and nonterminal goals; $R \subseteq NT$ is the set of root or intendable nonterminal goals; and P is a set of production rules of the form $(n, \beta, \sqsubset)$ where $n \in NT$, β is a string of symbols in $\Sigma \cup NT$ where either $|\beta| = 1$ and $\beta \subseteq \Sigma$ or $|\beta| > 1$ and $\beta \subseteq NT$, and $\sqsubset$ is an antisymmetric relation over the indices $[1, |\beta|]$ into β.*

This formulation is equivalent to similar grammar-based approaches to HTN plan recognition (Geib, Maraist, and Goldman 2008; Geib and Goldman 2009), with the superficial change of representing disjunctive rules as several different rules for the same goal. It differs from traditional HTNs where an arbitrary condition determines the applicability of each rule (Ghallab, Nau, and Traverso 2004); this formulation is equivalent to HTN formulations without these conditions. A *frontier* is a set indexing subgoals of a rule yet to be achieved: for a rule $(M, \beta, \sqsubset)$, any $F \subseteq \{1, \cdots, |\beta|\}$ where if $x \in F$ and $x \sqsubset y$ then $y \in F$ as well. The *initial* frontier of a rule $(M, \beta, \sqsubset)$ is just the set of all indices into β. The *insertion points* of a frontier are those not blocked by an ordering requirement.

A probabilistic plan recognizer generates explanations on which we make probabilistic queries; the typical query is the probability that a particular goal is being pursued under the given observations. Following Geib and Goldman (2009) we calculate this query as:

$$P(G|obs) = \frac{\sum_{e \in \text{Expls}(obs), G \in e} P(e_i \wedge obs)}{\sum_{e \in \text{Expls}(obs)} P(e \wedge obs)} \qquad (1)$$

where Expls(obs) is the recognizer's set of explanations for the given observations, and $G \in e$ qualifies that the goal G is intended under explanation e. Calculating these $P(e_i \wedge obs)$ relies on the model introduced by Goldman et al. (1999). We enrich the plan library with:

1. The probability $P(G)$ that a particular goal G is intended for its own sake.
2. The probability of choosing a particular alternative when there are several production rules for the same goal, which gives the probability $P(plan)$ that a particular plan will be used to achieve a given goal.
3. The probability $P(s|S)$ that the agent will choose a particular primitive action s from a set of pending primitive actions S at some point in plan execution, which gives the probability of a particular ordering of observed actions.

Then to calculate Equation 1 we quantify these factors,

$$P(e \wedge obs) = k \cdot \prod_G P(G)^{|G_e|} \cdot \prod_{plan \in e} P(plan) \qquad (2)$$
$$\cdot \prod_i P(obs_i | obs_0, \cdots, obs_{i-1}) \ .$$

$|G_e|$ is the number of distinct plans achieving G which the explanation identifies, and k is a factor accounting for no further plans achieving G being pursued. For simplicity in this presentation we assume that the latter two probabilities are uniform distributions, but nothing in the YR algorithm prevents a more complicated model. A noteworthy complication of calculating the $P(obs_i | obs_0, \cdots)$ is that it changes when we discover additional intentions. We cannot preemptively weigh pending actions for the new intention. So we store a *pending stack* describing the entire series of pending sets rather than only the most recent set. Often one simplifies the probability model by declaring all members of each pending set to be equally likely; in this case we must store only the sets' sizes, not the sets themselves.

2.2 LR and GLR parsing

The similarity in structure between plan libraries and context-free grammars is clear. Grammars have a single start symbol where plan libraries have multiple top-level intentions; grammar rules are totally ordered, where plan rules may be partially ordered or unordered.

Definition 2 *A context-free grammar (CFG) is a tuple (NT, Σ, R, S) where Σ and NT are disjoint finite alphabets respectively of terminals and nonterminals; $S \in NT$ is the* start *symbol; and R is a set of production rules (A, u) where $A \in NT$, and u is a string of symbols in $\Sigma \cup NT$.*

We write the pair (A, u) as the more intuitive $A \to u$. An *item* represents the state of matching inputs against a rule. Items for a rule are written by adding a dot at a position in the rule's right-hand side; a rule of length n will give rise to $1 + n$ items. The *initial item* of a rule with $A \to u$ is $A \to \bullet u$. In building a parser it is typical to select a nonterminal S' and a terminal \$ which are not part of the original grammar, and extend the grammar with those symbols and the production rule $(S', S\$)$. The \$ represents the end of input, and the additional rule makes the success condition for the parser unambiguous.

Our summary of LR parsing largely follows Grune and Jacobs (2008). We focus specifically on LR(0) parsing; unlike parsing an entire given string, in plan recognition we cannot expect knowledge of an agent's future actions ahead of time! The core of an LR parser is some technique for finding the next *handle*, a segment of the string which can be reduced to a particular nonterminal. Efficient parsing requires efficient handle-finding; we use a deterministic finite automaton (DFA). The automaton is more easily understood as a nondeterministic finite automaton (NFA), which is then trivially translated to a DFA. The NFA has as its states both individual items, plus one additional state for each nonterminal, the *station* of that nonterminal. The NFA's initial state is the station for the extending nonterminal S'. There are three sources of transitions in the NFA: From an item $A \to u \bullet av$ (where a is either a terminal or nonterminal, and for any strings u and v) there is a transition labeled a to item $A \to ua \bullet v$.[1] Moreover from every item $A_0 \to u \bullet A_1 v$ there is an ε-transition to the station for A_1. Finally, for each rule $A \to u$ there is an ε-transition from A's station to each initial item $A \to \bullet u$ of a rule for A. The DFA corresponding to this NFA becomes the basis for the parser's control tables. The states of the DFA are typically numbered rather than labeled with item sets, the initial state taking the lowest number.

[1]Although we do not explore the additional bookkeeping required in the parsing algorithm here, additionally including ε-transitions from $A \to u \bullet av$ to $A \to ua \bullet v$ for certain terminal a would give a simple extension for the case where a is present but not observed. Likewise, additional a-labeled transitions from $A \to u \bullet av$ to itself model false observations.

From the handle-finding DFA we can construct parser action tables. Table columns correspond to symbols; table rows correspond to item sets; and each entry contains zero or more actions. Starting with initially no actions we populate the table according to three rules,

1. If the DFA contains a transition $s_1 \underset{a}{\rightarrow} s_2$ for $a \in \Sigma$, then we add shift s_2 to row s_1, column a.
2. If a state s contains an item $A \rightarrow u\bullet$ for $A \in NT$ (so not S'), we add reduce $A \rightarrow u$ to every column in row s.
3. If a state s contains $S' \rightarrow S \bullet \$$ for the $S', \$ \notin NT \cup \Sigma$, we add accept to row s, column \$.

An entry with more than one action is a *conflict*: a *shift-reduce* conflict when both a shift and one or more reduce actions are present, or a *reduce-reduce* conflict when two or more reduce actions are present. An entry with no actions is taken to be a shift to some error state.

For tables with no conflicts, LR(0) parsing is a straightforward reading of the transition tables in a push-down automaton: the initial stack holds the initial row number (or item set). Parsing involves iterating through the following loop:

- First we pop the next input symbol s
- While the action table has an entry reduce $A \rightarrow u$ at the row for the state on top of the stack, column s:
  - First we pop $|u|$ elements from the stack.
  - The row now atop the stack will have a shift entry for column A, and we push the indicated state onto the stack.
- Then we consider the entry reduce $A \rightarrow u$ at the row for the state on top of the stack, column s:
  - If it is the action shift s' then we push s' onto the top of the stack.
  - If it is accept, then we terminate successfully.

If at any point we have an error state at the top of the stack, parsing fails.

When there are conflicts in the parser table, the LR algorithm is correct but becomes nondeterministic, since any choice of actions leading to the accept state does still signify that a string is in the language generated by the grammar. GLR and similar approaches structure a breadth-first search over such choice points (Tomita 1985; Tomita and Ng 1991). Conceptually GLR keeps a separate stack for each possible action, deriving one set of stacks from another at each step. Individual stacks with no shift transition are simply discarded; parsing fails only when the set of current stacks becomes empty. Of course, the naïve implementation which copies the stack upon each ambiguity is absurd; GLR relies on two essential optimizations for its performance. The first optimization creates a *tree-structured stack*: rather than duplicating the entire stack at each conflict, we share the stack prefix among the several alternatives. The second optimization then gives us a *graph-structured stack* sharing graph suffixes as well: the shift and reduce actions for a single input symbol will collectively create at most one node for each table row, possibly sharing each new node among several previous stack tops. Future operations to these shared suffixes are common to all relevant stacks until a reduce step pops past the merge point. As transient as the shared suffixes can be, the optimization is nonetheless crucial to GLR's performance; although GLR's worst-case performance is exponential compared to the size of its grammar, this worst-case behavior arises only for very artificially ambiguous grammars. For grammars with limited ambiguities, GLR is nearly linear and represents the state of the art of parsing (Grune and Jacobs 2008, Sec. 11.3).

2.3 The shuffle operator

Although shuffling has long been an aspect of concurrent systems analysis, it is only in recent years a theory of languages with shuffling has developed (Restivo 2015). Much of the work to date focused on theoretical properties of language classes; practical approaches have been limited to regular expressions and languages with shuffling (Sulzmann and Thiemann 2015; submitted 2016). Recently we proposed GLR-S, an extension of GLR for CFSGs (2016), and offer a proof of its correctness. The shuffle operator is distinct from ID/LP grammars, which also relax the total order constraints on the right-hand sides of production rules (separating immediate dominance ID from linear precedence LP). However ID/LP grammars only allow reordering of the symbols of a rule's right-hand side at the point where the symbol at the left-hand side is expanded; no subsequent shuffling of the expansions of the symbols of the right-hand side is allowed. Shieber shows that an ID/LP grammar can always be converted to an equivalent CFG (1984), while context-free languages are not closed under shuffle.

CFGSs differ simply by allowing rules of the form $A \rightarrow A_1||A_2|| \cdots ||A_n$. Where each A_i rewrites to $u_i \in \Sigma^*$, A rewrites to any shuffling of the $u_1, \ldots, u_n$. GLR-S extends the core (nondeterministic) LR algorithm in four steps: extending first the notion of an item, then the handle-finding NFA, then the construction of the action table, and finally updating the processing of new action table entries. We describe each of these steps, and then the optimizations to the deterministic generalized algorithm.

GLR-S adds two sorts of items to the classical notion. Corresponding to a shuffle rule, there are items $A \rightarrow \bullet\{A_1, \cdots, A_n\}_m$ and $a \rightarrow \{A_1, \cdots, A_n\}_m\bullet$. Here m is an integer set to the initial number of shuffled subterms, serving the same role in reduction as the length of right-hand rule sides in the case of sequential rules. In addition to stations there is a placeholder item ∇A representing a transition to the subtask of recognizing a particular shuffled substring. For the handle-finding NFA there are additional transitions corresponding to the new items. From an item $A \rightarrow \bullet\{A_1, \ldots, A_n\}_n$ and for each i there is a transition labeled A_i to $A \rightarrow \bullet\{A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n\}_n$, plus an ϵ-transition from any $A \rightarrow \bullet\{\}_n$ to $A \rightarrow \{\}_n\bullet$. GLR-S annotates the NFAs with *hyperedges* linking from one state to several. Specifically the graph of states and edges forms an arc-labeled F-directed hypergraph (Gallo et al. 1993). Of course the hyperedges have no impact on the operation of an NFA (and in any event one does not "run" the handle-finding automaton in any real sense). Our interest in the hyperedges is strictly for accounting: translating them to the DFA and generating action table entries based on them.

Where there is such an edge from an item I to items I_i in the NFA, we expect that any state containing I in the DFA would have a similar hyperedge to the least sets containing the I_i. From the initial item $a \to \bullet\{A_1, \dots, A_n\}_n$ of a shuffle node we add a hyperedge to the n indirection items $\nabla A_1, \cdots, \nabla A_n$; from each indirection item ∇A_i we have an ϵ-transition to the station for A_i. The hyperedges have their own translation to the action table; for each hyperedge $\mathcal{S} \to \mathcal{S}_1, \cdots, \mathcal{S}_n$ and simple edge $\mathcal{S}_i \underset{s}{\to} \mathcal{S}'_i$ where $s \in \Sigma$, the action $\mathsf{subs}(\mathcal{S}_i \to \mathcal{S}'_i; \mathcal{S}_1, \cdots, \mathcal{S}_{i-1}, \mathcal{S}_{i+1}, \mathcal{S}_n)$ is added to the table at row $\mathcal{S}_0$, column s.

The runtime state of the core nondeterministic GLR-S algorithm is a cactus stack — a tree which grows and shrinks at the leaves — of states from the handle finding DFA. We write $\beta : \mathcal{S}$ to name a stack top β containing the state $\mathcal{S}$. Figure 1 shows the core nondeterministic GLR-S algorithm. Step 1(c) of the algorithm shows the purpose of the ∇A nodes. The reduce operations in LR parsers rely on the number of states pushed onto the stack for the right-hand side of a rule being the same as the length of that right-hand side. But when splitting the stack for shuffled substrings, the substacks start with an initial state that does not correspond to any recognized symbol. The presence of an indirection to ∇A in an item set allows the parser to detect and correct this offset. But since the ∇A item would not appear in the state corresponding to a sequential use of A — which would follow an ϵ-link to A's station instead of to ∇A — there is no disruption to stack operations in sequential cases.

GLR-S adopts the graph-structured stack, but the mid-stack mutation performed at Step 1.(e).i of the algorithm complicates its use. In the nondeterministic parser it is simple enough to mutate the middle of a stack, but not all of the stacks superposed in graph-structuring will undergo the same mutations. To store stacks a deterministic implementation of GLR-S uses not one but two graphs, distinguished as *upper* and *lower*. The lower graph holds parser states as in Tomita's graph-structured stack, but disconnected at the junctions arising from shuffle operations. The relationship between these graph fragments, as well as the sets of stack tops, are maintained in the upper graph. The upper graph has the structure of an and/or tree, possibly with shared substructure. Or-nodes reflect different possible parses; and-nodes organize recognition of shuffled substrings. To chain together substacks in the lower graph, we bind *controllers* in and-nodes, and reference both a stack top and one of these controllers in each leaf node. The binding of a controller in an and-node associates the controller with both a prior stack top, and a parent controller. So essentially the controllers form a linked tree of stacks from the lower graph, which taken together assemble the cactus stacks of the nondeterministic algorithm. Separating the binding of a controller to a parent stack top from the stack tops in leaf nodes in this way allows local mutation of controller bindings in a way that restricts the scope of the effect.

So for each input symbol, GLR-S traverses the upper graph, constructing a new upper graph while reusing as much of the previous graph as possible. When traversing an or-node one discards any children for which there is no way to advance with the next input. An and-node requires exactly one of its children to advance for each input symbol; if more than one can advance, then there will be a disjunction for each possible evolving child, with the other child graphs in each case unchanged. The usual shift and reduce operations apply at leaf nodes. When a reduce operation exhausts a stack via Step 1(c) of the algorithm, GLR-S updates the controller binding, possibly to a disjunction of several different bindings. In each traversal GLR-S preserves the shared structure of the upper graph by caching a map from old upper graph nodes to new, thus traversing each shared subgraph once only.

Algorithm: Core GLR-S parsing. Initially the single state on the stack is the DFA's initial state. For each symbol s of the input string including the end-of-string marker \$:

```
1. For each reducible stack top $\alpha : \mathcal{S}$, if $\mathsf{reduce} a \to u\bullet \in \mathcal{S}$, then
   we may choose to reduce that rule:
   (a) Drop $\alpha$ as a stack top.
   (b) Pop $|u|$ nodes from $\alpha$ to node $\alpha'$.
   (c) If $\alpha' : \mathcal{S}'$ contains the indirection node for $a$, $\nabla a \in \mathcal{S}'$, pop
       one additional time to $\alpha''$, else take $\alpha''$ to be just $\alpha'$.
   (d) Let $\alpha'' : \mathcal{S}_0$, and choose some shift operation $\mathsf{shift}(\mathcal{S}'_0)$ from
       row $\mathcal{S}_0$, column $a$, or raise an error if there is no possible
       shift. Create $\beta : \mathcal{S}'_0$ with parent $\alpha''$.
   (e) If $\alpha''$ has other child nodes:
       i. Then update the other children of $\alpha''$ to have $\beta$ as their
          parent.
       ii. Else take $\beta$ as a stack top.
2. Choose a stack top $\alpha : \mathcal{S}$ with a stack or accept operation in row
   $\mathcal{S}$, column $s$ (or reject if there is no such $\alpha$), and choose one of
   those operations.
   (a) If the operation is $\mathsf{accept}$, then the parser accepts the string.
   (b) If the operation is $\mathsf{shift}(\mathcal{S})$, then create $\beta : \mathcal{S}'$ with parent $\alpha$,
       and replace $\alpha$ as a stack top with $\beta$.
   (c) If the operation is $\mathsf{subs}(\mathcal{S}_0 \to \mathcal{S}'_0; \mathcal{S}_1, \cdots, \mathcal{S}_n)$, then create
       $\beta_i : \mathcal{S}_i$ with parent $\alpha$ for each $0 \le i \le n$, and moreover create
       $\beta'_0 : \mathcal{S}'_0$ with parent $\beta_0$. Replace $\alpha$ as a stack top with $\beta'_0$ and
       the $\beta_1, \cdots, \beta_n$.
```

After the end-of-string marker \$ if we have not accepted the input, then we reject it.

Figure 1: The core GLR-S nondeterministic parsing algorithm.

3 From shuffle parsing to plan recognition

With CFSGs rather than CFGs as an analogue, we have a much closer connection between plans and grammars. But the match is not perfect: first, the partial order of HTN plans is more expressive than the ordering allowed under CFSGs. In HTN plan recognition there may be multiple active goals drawn from multiple top-level intentions, whereas in parsing there is a single instance of a unique start symbol. There is not necessarily a terminator \$ to the observed actions for each plan. And finally, we are interested in probabilistic plan recognition. We consider these issues in turn in this section, and additionally consider the issue of evaluating explanation

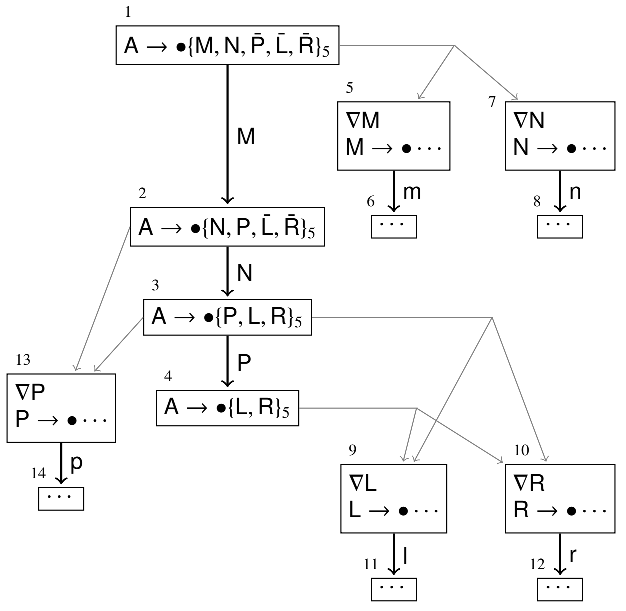

Figure 2: Portion of the handle-finding DFA for a library containing Plan A of Section 3.

probabilities.

The partial order among HTN plan steps is more expressive than the the total-or-no order of CFSGs. For example, a plan for A involving five steps L, M, N, P, R may require $M < P$, $N < R$ and both $M, N < L$. This ordering cannot be expressed with the tools of a CFSG. However the implementation in YR is straightforward. In cases where a new concurrent subgoal becomes unblocked by the completion of an existing subgoal, we add additional hyperedges to the handle-finding automaton from states where the one of new concurrent subgoals may first be oberved. To illustrate with A, let us assume that m is a possible first observation towards M, n towards N, and so on; we use overbars to indicate subterms of a shuffle item which are "blocked" by ordering constraints, as in $\mathsf{A} \rightarrow \bullet\{\mathsf{M}, \mathsf{N}, \bar{\mathsf{L}}, \bar{\mathsf{P}}, \bar{\mathsf{R}}\}_5$. So the handle-finding DFA for A's library would include the fragment shown in Figure 2 and the action table would include rows as shown in Table 1.

In a sense it is trivial to address the multiple top-level goals of HTNs; in the handle-finding NFA we can simply take the stations of all of these goals to be initial in the NFA, such that all of their initial items are in the DFA's initial state. We can also admit multiple intentions with ease, as a top-level set of concurrent states with interleavable observations. Whenever YR address a new observation, it is always an option to start a new substack from the initial state, perhaps in addition to the possibility of advancing the previously existing stacks. By allowing these new goals to re-use lower graph structure, YR further optimizes space. The lack of a terminator is not a problem *per se*; we simply do not generate the additional wrapping rule. A long-running plan recognizer will need some mechanism to detect and retire long-satisfied goals, but this question is separate from the scope of this work.

Probabilities. Our model of HTN plan recognition probabilities contains three distinct sources of probability information, and we account for each in YR. In top-down methods such as Yappr, the intended goal behind any particular explanation is clear since each explanation begins with the hypothesized intention. But in a bottom-up method one state may correspond to multiple alternative intended goals. Different transitions may be possible under different goals, which in turn gives different values for $P(obs_i|o_0, \ldots o_{i-1})$ in Equation 2 for different intentions. For this reason, in the handle-finding NFA for YR it is necessary to annotate items with an intention. Each state (each item set) can then be associated with the union of the goals annotating its items. These goal sets give us not only the base probability of the intention behind an explanation, but as we will see below they are essential for discerning the pending sets of possible actions.

The second aspect of probability information from HTN plan recognition models is that associated with different ways for achieving a goal. These are just the same as in probabilistic CFGs, and we can use similar techniques, such as of Wright and Wrigley (1991). We have a probability associated with each action table entry, and a running product at each stack top. When the stack corresponding to one of several shuffled strings is exhausted and the controller's stack top is advanced, the probability associated with the exhausted stack is merged (multiplied) into the controller as well. Both the intention annotations and the incorporation of probabilities in the parser table tends to increase the size of the tables, but in practical plan libraries the increase is not prohibitive, as we discuss in Section 4.

The final aspect of the HTN probability model is the selection of one particular action from the several which may be available at any one time. The compression of intentions and explanations which comes with the bottom-up LR style also impacts our access to pending sets. Since the parser states can correspond to multiple intentions, and since within an item set certain intentions may be relevant to only a subset of items, we populate the pending stack with item sets (or their indices). Even in the case where we deem all pending set members to be equally likely, in general we can fix the size of the pending sets only when considering one single intention at a time. So when we traverse the upper graph, we record the parse state number into the pending stack. Then when we later hypothesize a particular goal for an intention, we can translate this stack into the number of actions available at each step.

One complication would arise from adopting a stack structure with one entry per observation: we cannot re-use upper graph fragments from observation to observation if we represent them as a simple stack. The stack of *every* leaf node would then need to be updated for each observation, even when a leaf continues to reference the same lower graph node. In these cases the same actions are possible, so the state numbers added to a naïve stack would be the same as the current top-of-stack value. We can take advantage of

State	M	N	P	m	n	l	r	p
1	shift 2			subs(5→7;6)	subs(6→8;5)			
2		shift 3						subs(13→14)
3			shift 4			subs(9→11;10)	subs(10→12;9)	subs(13→14)
4						subs(9→11;10)	subs(10→12;9)	

Table 1: Portion of the YR action table derived from the handle-finding automaton of Figure 2, where concurrent subgoals become unblocked.

the fact that YR processes one observation at a time, "timestamping" each stack entry with the observation number current when it was pushed, and taking that entry to apply at all times forward up to the next timestamp. In this way re-using an unchanged subset of the upper graph implicitly pushes the same state onto its pending stack — so essentially, the pending stack is a sparse stack which only records changes.

A pending stack of single states as we have described it so far is sufficient for the non-branching stack of a grammar fragment which does not have shuffling. But a single state will not capture the possible actions across the branched states of recognizing shuffled strings. When a substack is exhausted and the probability information is incorporated into the controller stack, we will actually need a pending stack of *multisets* of states. Incorporating the pending stack from an exhausted shuffle substack entails the pairwise multiset union of corresponding pending stack entries.

Querying across explanations. In the YR state, the upper graph encodes all explanations for observed actions. To answer the typical probabilistic query of whether the observations suggest that a particular goal is intended, we must iterate across YR's compression of multiple goals and plans into the same upper graph nodes. We traverse the upper graph, associating prior probabilities and pending stacks to particular distinct intentions, and enumerating the different combinations of these distinct intentions to cover all observations. The states in the pending stack associated with each distinct intention tell us both the actual goals which might be associated with an intention, and the choices of alternatives from disjunction rules. Each pair of an actual goal plus choices of alternatives denotes a distinct plan for the intention; the cross products of the plans for the different intentions in a covering combination comprises the explanations (in the sense of Equation 2) for that combination. Note that it is only at these final steps that we can translate the pending stack's table row number multisets into the number of possible actions available at each step, since it is only then that we identify the goal associated with the intention, and thus can rule out actions associated only with other goals. In traversing the upper graph we can calculate simultaneously the probability of each intention, plus the total probability mass, for Equation 1.

4 Performance

We have not completed a complexity analysis of GLR-S, but we do not expect to find that it will run in worst-case polynomial time: Berglund *et al.* (2013) showed that determining membership in a shuffle of context-free languages is NP-complete. Although exponential worst-case times are typical of HTN plan recognizers, they tend to behave much more reasonably on all but the most eccentric examples. For testing we generated libraries with two "generations" of alternating or- and and-rules. Each has 100 top-level intendable goals naming an or-rule; each or-rule has two children, and each and-rule, three. Children are randomly chosen from a pool of rule names, so there is a possibility of ambiguity in plan derivations. We vary the ordering relationship on and-node children, considering six different groups of libraries: totally ordered children, unordered children, the "head" (resp. "tail") order where one child must precede (follow) the others, and randomized orders where there is a 50% or 25% chance when generating the library for each pair of children that the order between them will be enforced. Except for the unordered group, each input set interleaves the observations for three intendable goals, and we imposed a five-minute maximum run time. For the unordered group, we ran only a single goal's observations; since every observation can always signify a new intention, the number of observation partitions grows much faster than with the other groups.

We compare YR to Yappr, and based on those results argue that YR provides a dramatic improvement in performance. Although Yappr is not the most recent probabilistic plan recognizer, it is the most recent for which we can make an "apples-to-apples" comparison with YR, and its performance still ranks high (Stoutenburg Tardieu 2015). ELEXIR offers considerable performance improvements, but it is based on combinatory categorical grammars (CCGs) rather than HTN plans. Automatically and sensibly translating between HTN libraries and CCG lexicons remains an open problem, so it is not clear how to include ELEXIR in the sort of comparison based on randomized libraries which we use here. A further functional difference is that the suggested best-performance use of ELEXIR is to structure lexicons such that association of a goal with actions is postponed until late in the sequence of actions for that goal, while YR is designed to provide probabilities for all possible goals upon the first observation. With regard to SLIM, our focus with YR is on inferring the *goal* behind observations — as with Yappr we do not retain detailed plan information, and instead assume that the full plan details can be reconstructed from the goals and observations. In fact, much of the improvement in Yappr over its predecessor PHATT arose from discarding these details. The goal of SLIM, on the other hand, is to improve performance while retaining all plan details. As such, a comparison between SLIM and YR would be sharply weighted towards YR, as

	Run times							Mean
	Mean		Std. dev.		% YR	Timeouts		Yappr/
Group	YR	Yappr	YR	Yappr	faster	YR	Yappr	YR ratio
Total	14ms	305ms	0.08ms	3.7ms	99.0	0	0	15.5
Head	35ms	881ms	0.64ms	36ms	95.2	0	0	11.0
Tail	2.0s	113s	3.5ms	6.0s	94.8	0	149	45.4
50%	5.0s	118s	200ms	9.0s	97.8	0	172	70.5
25%	88s	204s	5.0s	-	52.4	206	427	68.5
Unord	2.0s	231s	327ms	5.0s	100.0	0	203	103.9

Table 2: Summary of performance data. Means, std. devs. and ratios include only runs which terminated; where timeouts are reported the actual values would be larger.

a rough examination of SLIM's reported improvement over PHATT compared to Yappr's reported improvement over PHATT suggests. It is likely for these same reasons that neither ELEXIR nor SLIM compared themselves to Yappr in the presentation of their performances.

Table 2 shows a summary of the run times by YR and Yappr of all six groups. YR performed more quickly than Yappr in all but a handful of cases, between 94.8% and 100% of the runs in the groups. The mean run time entries in the table exclude runs which timed out, so performance is actually worse than the table would suggest for systems and groups with a substantial number of timeouts. The figure also shows that YR's run times are much less variable than Yappr's (and again, the presence of timeouts indicates a greater actual variance than reported). The rightmost column shows the mean ratio of Yappr and YR's run times on the same input; YR consistently runs between one and two orders of magnitude faster than Yappr. Figure 3 plots the run times of YR and Yappr on the individual runs.

Much of YR's performance advantage is due to its compact representation of expressions, and to a lesser extent its sharing of graph structures. The upper graph allows differences among explanations to be expressed locally within that graph. Crucially, the bottom-up approach naturally allows us to avoid creating separate representations for explanations which differ only by the particular goal attributed to an explanation, or by alternate plans for (sub)goals: every explanation with a particular partition of observations among intentions shares its representation in the upper graph. In the 25%-ordered and unordered cases, we see YR's behavior degrade as the number of partitions spikes, but still outperforming Yappr.

One area of YR's performance which does not improve over Yappr is the compilation of plan libraries. The final size of the compiled artifacts (comparing the number of rows in YR's table to the number of plan frontier fragments) is not consistently greater in either YR or Yappr. But our adoption of LR(0) table construction does require considerably longer than Yappr's compilation — up to several minutes instead of rarely over a second. Of course we have not attempted to adapt the many improvements to LR table construction to YR, and we believe it is likely that table construction times can be significantly improved.

5 Conclusions and future work

We have presented a new algorithm for probabilistic HTN plan recognition based on generalized LR shuffle parsing, and argued for the alignment of HTN plan libraries with context-free shuffle grammars rather than context-free grammars in the correspondence of plan recognition to parsing. Applying bottom-up parsing techniques allows dramatic performance improvements over previous algorithms derived from top-down parsing techniques.

In the previous section we noted a difference in scope between YR and SLIM: YR focuses on deducing the top-level goals behind observed actions, while SLIM retains full plan details. Along with the basic GLR algorithm Tomita gave an approach for efficiently representing a *parse forest*, a collection of parse trees, of all possible derivations. An interesting extension of YR (and GLR-S) would adopt Tomita's parse forest to accommodate shuffle expressions, which in YR would correspond to full plan recognition.

References

Avrahami-Zilberbrand, D., and Kaminka, G. A. 2005. Fast and complete symbolic plan recognition. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Berglund, M.; Björklund, H.; and Björklund, J. 2013. Shuffled languages — representation and recognition. *Theoretical Computer Science* 489–490:1–20.

Boutilier, C., ed. 2009. *Proceedings of the 21st International Joint Conference on Artificial Intelligence*. Morgan Kaufmann.

Bui, H. H.; Venkatesh, S.; and West, G. 2002. Policy recognition in the abstract hidden Markov model. *Journal of Artificial Intelligence Research* 17:451–499.

Dediu, A.-H.; Formenti, E.; Martín-Vide, C.; and Truthe, B., eds. 2015. *Proc. 9th International Conference on Language and Automata Theory and Applications (LATA)*, volume 8977 of *Lecture Notes in Computer Science*. Nice: Springer.

Gallo, G.; Longo, G.; Pallottino, S.; and Ngyyen, S. 1993. Directed hypergraphs and applications. *Discrete Applied Mathematics* 42:177–201.

Geib, C. W., and Goldman, R. P. 2009. A probabilistic plan recognition algorithm based on plan tree grammars. *Artificial Intelligence* 117(11):1101–1132.

Geib, C. W., and Goldman, R. P. 2010. Handling looping and optional actions in YAPPR. In *Plan, Activity, and Intent Recognition*.

Geib, C. W.; Maraist, J.; and Goldman, R. P. 2008. A new probabilistic plan recognition algorithm based on string rewriting. In Rintanen et al. (2008), 91–98.

Geib, C. W. 2009. Delaying commitment in plan recognition using combinatory categorial grammars. In Boutilier (2009).

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Series on Agent Technology. Morgan Kaufmann Publishers.

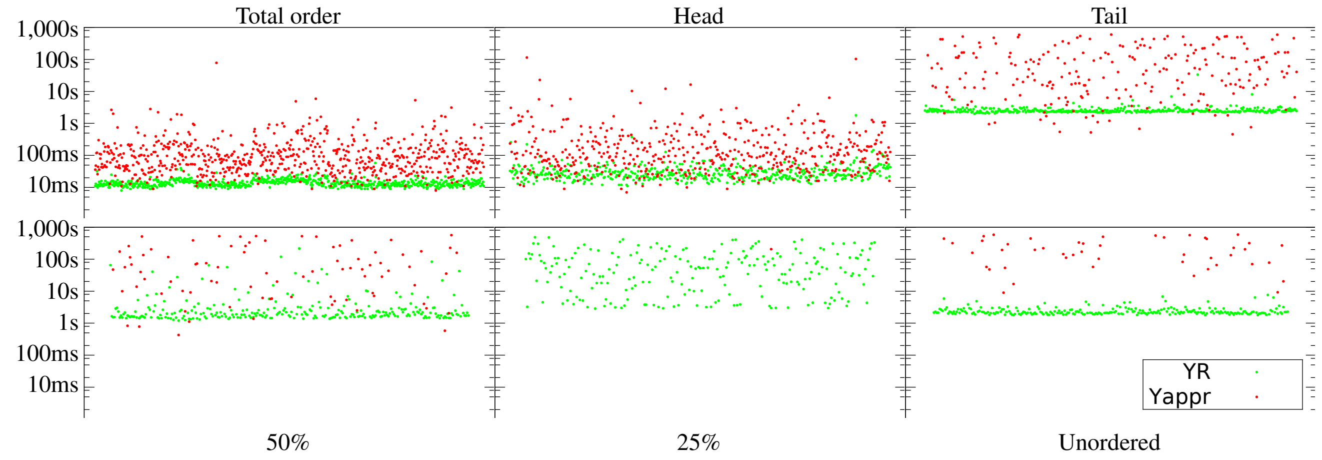

Figure 3: Comparison of YR and Yappr runtimes for different plan library profiles. Each vertical pair of points represents a run of YR and Yappr on the same plan library and input data. Missing points indicate runs terminated upon exceeding the timeout.

Goldman, R. P.; Geib, C. W.; and Miller, C. A. 1999. A new model of plan recognition. In Laskey and Prade (1999), 245–254.

Grune, D., and Jacobs, C. J. 2008. *Parsing Techniques: A Practical Guide*. Springer, 2 edition.

Huber, M. J.; Durfee, E. H.; and Wellman, M. P. 1994. The automated mapping of plans for plan recognition. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 344–351.

Kaminka, G. A.; Pynadath, D. V.; and Tambe, M. 2002. Monitoring teams by overhearing: A multi-agent plan-recognition approach. *Journal of Artificial Intelligence Research* 17(1):83–135.

Kautz, H. 1991. *A Formal Theory of Plan Recognition and its Implementation*. Ph.D. Dissertation, University of Rochester.

Laskey, K., and Prade, H., eds. 1999. *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*. San Francisco, California: Morgan Kaufmann.

Maraist, J. 2016. Generalized LR parsing and the shuffle operator. On *arxiv.org*, arXiv:1611.05831.

Mirsky, R., and Gal, Y. K. 2016. SLIM: Semi-lazy inference mechanism for plan recognition. In Kambhampati, S., ed., *Proc. 25th International Joint Conference on Artificial Intelligence*.

Pynadath, D., and Wellman, M. P. 2000. Probabilistic state-dependent grammars for plan recognition. In *Proceedings of the Sixteenth Conference on Uncertainty in Artificial Intelligence*, 507–514.

Ramírez, M., and Geffner, H. 2009. Plan recognition as planning. In Boutilier (2009), 1778–1783.

Restivo, A. 2015. The shuffle product: New research directions. In Dediu et al. (2015), 70–81.

Rintanen, J.; Nebel, B.; Beck, C.; and Hansen, E., eds. 2008. *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS-2008)*.

Shieber, S. M. 1984. Direct parsing of ID/LP grammars. *Linguistics and Philosophy* 7(2):135–154.

Stoutenburg Tardieu, C. 2015. Une comparaison d'algorithmes de reconnaissance de plan à l'aide d'observations in situ. Master's thesis, University of Sherbrooke.

Sulzmann, M., and Thiemann, P. 2015. Derivatives for regular shuffle expressions. In Dediu et al. (2015).

Sulzmann, M., and Thiemann, P. submitted 2016. Derivatives and partial derivatives for regular shuffle expressions. *Journal of Computer and System Sciences*.

Tomita, M., and Ng, S.-K. 1991. The generalized LR parsing algorithm. In Tomita (1991). chapter 1, 1–16.

Tomita, M. 1985. An efficient context-free parsing algorithm for natural languages. In Joshi, A. K., ed., *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, 756–764.

Tomita, M., ed. 1991. *Generalized LR Parsing*. Kluwer Academic Publishers.

Vilain, M. 1991. Deduction as parsing. In *Proceedings of the Conference of the American Association of Artifical Intelligence*, 464–470.

Wright, J., and Wrigley, E. 1991. GLR parsing with probability. In Tomita (1991). chapter 8, 113–128.