Searching for More Efficient Dynamic Programs

Tim Vieira **Ryan Cotterell** **Jason Eisner**
Johns Hopkins University University of Cambridge ETH Zürich
tim.f.vieira@gmail.com ryan.cotterell@inf.ethz.ch
jason@cs.jhu.edu

Abstract

Computational models of human language often involve combinatorial problems. For instance, a probabilistic parser may marginalize over exponentially many trees to make predictions. Algorithms for such problems often employ dynamic programming and are not always unique. Finding one with optimal asymptotic runtime can be unintuitive, time-consuming, and error-prone. Our work aims to automate this laborious process. Given an *initial* correct declarative program, we search for a sequence of semantics-preserving transformations to improve its running time as much as possible. To this end, we describe a set of program transformations, a simple metric for assessing the efficiency of a transformed program, and a heuristic search procedure to improve this metric. We show that in practice, automated search—like the mental search performed by human programmers—can find substantial improvements to the initial program. Empirically, we show that many speed-ups described in the NLP literature could have been discovered automatically by our system.

1 Introduction

Algorithmic research in natural language processing (NLP) has focused—in large part—on developing dynamic programming solutions to combinatorial problems that arise in the field (Huang, 2009). Such algorithms have been introduced over the years for countless linguistic formalisms, such as finite-state transduction (Mohri, 1997; Eisner, 2002; Cotterell et al., 2014), context-free parsing (Stolcke, 1995; Goodman, 1999), dependency parsing (Eisner, 1996; Koo and Collins, 2010; Ma and Zhao, 2012) and mildly context-sensitive parsing (Vijay-Shanker and Weir, 1989, 1990; Kuhlmann et al., 2018). In recent years, the same algorithms have often been used for deep structured prediction, using a neural scoring function that decomposes over the structure (Durrett and Klein, 2015; Rastogi et al., 2016; Lee et al., 2016; Dozat and Manning, 2017; Stern et al., 2017; Kim et al., 2017; Hong and Huang, 2018; Wu et al., 2018; Wu and Cotterell, 2019; Qi et al., 2020; Rush, 2020).

When a dynamic programming algorithm for a new problem is first introduced in the literature, its runtime may not be optimal—faster versions are often published over time. Indeed, the process of introducing a first algorithm and subsequently finding improvements is common throughout computer science. In the case of dynamic programming, there are program transformations that may be exploited to derive algorithms with a faster runtime (Eisner and Blatz, 2007). These transformations map a program to another program with the same meaning (given the same inputs, it will produce the same outputs), but with possibly different running time. This paper shows how to search over program transformation sequences in order to automatically discover faster algorithms, automating the work of the NLP algorithmist.

Consider the following instances[1] of published dynamic programs whose runtime bounds were later improved using specific applications of the program transformations mentioned above.

- Projective dependency parsing: Collins (1996) gave an $\mathcal{O}(n^5)$ algorithm that was sped up to $\mathcal{O}(n^4)$ by Eisner and Satta (1999).
- Split-head-factored dependency parsing: implemented naïvely runs in $\mathcal{O}(n^5)$; with some effort, an $\mathcal{O}(n^3)$ algorithm can be derived (Eisner, 1996; Johnson, 2007; Eisner and Blatz, 2007).
- Linear index-grammar parsing: $\mathcal{O}(n^7)$ in Vijay-Shanker and Weir (1989), sped up to $\mathcal{O}(n^6)$ by Vijay-Shanker and Weir (1993).
- Lexicalized tree adjoining grammar parsing: $\mathcal{O}(n^8)$ in Vijay-Shankar and Joshi (1985), sped up to $\mathcal{O}(n^7)$ by Eisner and Satta (2000).

[1] Many of these examples were brought to our attention in the works of Eisner and Blatz (2007) and Gildea (2011); further discussion can be found therein.

Findings of the Association for Computational Linguistics: EMNLP 2021, pages 3812–3830
November 7–11, 2021. ©2021 Association for Computational Linguistics

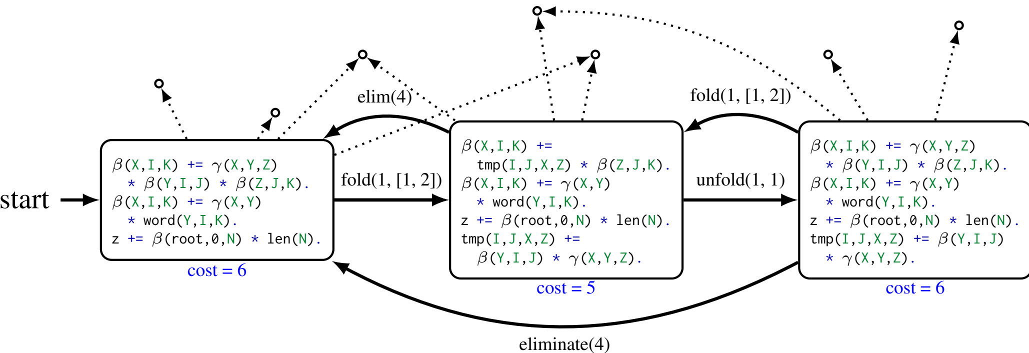

Figure 1: Depiction of the program optimization graph search problem (§5). The program used in this figure is our running example of speeding up CKY (Example 3). Nodes are Dyna programs (§2). The node pointed to by "start" indicates the user's program. Edges are program transformations (§4). Costs are derived by program analysis (§3). Only a tiny subset of the nodes and edges that exist in the search graph are shown. The dotted unlabeled outgoing edges represent additional transformations that we did not elaborate in the diagram to reduce clutter.

- Inversion transduction grammar: $\mathcal{O}(n^7)$ in Wu (1996), sped up to $\mathcal{O}(n^6)$ by Huang et al. (2005).
- CKY parsing (Cocke and Schwartz, 1970; Younger, 1967; Kasami, 1965) is typically presented in a suboptimal $\mathcal{O}(K^3n^3)$ form, but can be sped up to $\mathcal{O}(K^2n^3 + K^3n^2)$ (Lange and Leiß, 2009; Eisner and Blatz, 2007).
- Tomita's context-free parsing algorithm (1985) runs in $\mathcal{O}(n^{\rho+1})$ where ρ is the length of the longest right-hand side of a context-free production in the grammar (Johnson, 1989). However, it can be made to run in $\mathcal{O}(n^3)$ by binarizing the production rules.

In this paper, we ask a simple question: Can we *automatically* discover these faster algorithms? Typically, a dynamic programming algorithm can be regarded as performing inference in an semiring-weighted deduction system (Goodman, 1999). Eisner et al. (2005) provided a programming language, Dyna, for expressing such deduction systems, along with a compiler that produced fast inference code. All of the runtime improvements mentioned above are examples of source-to-source program transformations (Eisner and Blatz, 2007).[2]

Our work, depicted in Fig. 1, poses program optimization as a search over transformed versions of the initial program, an idea that was suggested as future work by Eisner and Blatz (2007). Our contribution is to show that two classic search algorithms—beam search (Reddy, 1977; Meister et al., 2020) and Monte Carlo tree search (Kocsis and Szepesvári, 2006)—are effective for this purpose, rapidly rediscovering many of the known optimizations listed above. To set up this solution, the following sections describe the elements of the search problem: our space of possible programs (§2), a simple cost function that serves as a proxy for program runtime (§3), and a set of directed edges that connect semantically equivalent programs (§4). Our search starts at the initial program and seeks a low-cost equivalent program that can be reached by traversing directed edges. In §5, we review the beam search and MCTS algorithms that we will use for this purpose in the experiments of §6.

2 Our Space of Dynamic Programs

We will consider programs that are expressed in the original version of the Dyna language (Eisner et al., 2005), which is essentially a way of writing down the recurrence relations of a dynamic programming algorithm. In this section, we only briefly describe the language and refer the reader to Eisner et al. (2005) for a more complete introduction. To start, consider the following examples.

Example 1. *The total weight of length-4 paths in a graph with edge weights* w*:*

```
20  z += w(Y1,Y2) * w(Y2,Y3) * w(Y3,Y4) * w(Y4,Y5).
```

This program defines the value of a derived **item** z in terms of input items w(...). The value of z is $\sum_{Y_1} \cdots \sum_{Y_5} \texttt{w}(Y_1,Y_2)\cdot\texttt{w}(Y_2,Y_3)\cdot\texttt{w}(Y_3,Y_4)\cdot\texttt{w}(Y_4,Y_5)$.[3] No-

[2]The closest work in the NLP literature is Gildea (2011), who proposed the junction-tree minimization to speed up dynamic programs, which corresponds to only considering the fold transformation. Outside of NLP, Mastria et al. (2020) learn to fold in the context of answer set programs. Our work considers a broader range of program transformations.

[3]We chose the name z as it is short and traditional for denoting the normalization constant of a probabilistic model, e.g., $\texttt{p}(Y_1, Y_2, Y_3, Y_4, Y_5) \propto \texttt{w}(Y_1, Y_2)\cdot\texttt{w}(Y_2, Y_3)\cdot\texttt{w}(Y_3, Y_4)\cdot\texttt{w}(Y_4, Y_5)$ would be have z as its normalization constant.

tice that there is no need to clutter the expression with explicit summations or control-flow constructs such as for-loops: all variables (denoted by capitalized letters) that appear only on the right-hand side of `+=` are summed over.

Dyna programs elegantly enable recursive computation by allowing the value of an item to be defined in terms of other items of the same kind.

Example 2. *The Viterbi algorithm (Viterbi, 1967) finds the most probable path in a graph from a node labeled* `"a"` *to a node labeled* `"z"` *with edge probabilities (or weights)* `w`*:*

```
21  α("a") max= 1.
22  α(J) max= α(I) * w(I,J).
23  z += α("z").
```

The second rule is recursive. For each possible value of `J` on the left-hand side, it defines `α(J)` by maximizing over assignments to the *other* variables on the right-hand side (namely `I`). Maximization is specified by `max=`, whereas summation in Example 1 was specified by `+=`.

Example 3. *Weighted context-free parsing with CKY (Cocke and Schwartz, 1970; Younger, 1967; Kasami, 1965), or more precisely the inside algorithm (Baker, 1979; Jelinek, 1985):*[4]

```
24  β(X,I,K) += β(Y,I,J) * β(Z,J,K) * γ(X,Y,Z).
25  β(X,I,K) += γ(X,Y) * word(Y,I,K).
26  z += β(root,0,N) * len(N).
```

The values of the γ items should be defined to be the weights of the corresponding context-free grammar rules: for example, the item $\gamma(\texttt{s},\texttt{np},\texttt{vp}) = 0.7$ encodes the production $\texttt{s} \xrightarrow{0.7} \texttt{np}\,\texttt{vp}$). Also, the item `word(X,I,K)` should be 1 if the input word `X` appears at position `I` of the input sentence and $\texttt{K} = \texttt{I} + 1$, and should be 0 otherwise. Then for any nonterminal symbol `X` and any substring spanning positions [`I`, `K`) of the input sentence, the item β(`X,I,K`) represents the total weight of all grammatical derivations of that substring from `X`.

More generally, a Dyna **program** $\mathcal{P}$ is a collection of **rules**, each rule having the form $\texttt{h} \oplus\!= \texttt{b}_1 \otimes \cdots \otimes \texttt{b}_K$. Here $\langle \oplus, \otimes \rangle$ can be any pair of operations that form a **semiring** (Goodman, 1999; Huang, 2009), such as $\langle +, * \rangle$ in Example 1 and Example 3, or $\langle \texttt{max}, * \rangle$ in Example 2.[5] We call `h` the head, and $\texttt{b}_1, \cdots, \texttt{b}_K$ the body of the rule. Each $\texttt{b}_k$ in the body is called a **subgoal**.

Let $\mathsf{head}(r)$ and $\mathsf{body}(r)$ denote the head and body terms in a rule r. We assume that all rules in the program use the same semiring.[6] The structured names of items are **terms**, which are nested typed tuples as in Prolog. For example, `f(g(z,h(3)))` is a 1-tuple of type `f`, whose single element is a 2-tuple of type `g`, and so on. The rules use captialized variables such as `X` to pattern-match against subterms, where a variable that is repeated in a rule must have the same value each time. Let $\mathsf{vars}(\cdot)$ denote the set of variables contained in a term, e.g., $\mathsf{vars}(\texttt{f}(\texttt{g}(\texttt{X}), 4, \texttt{X})) \mapsto \{\texttt{X}\}$. The Dyna language allows logical **side conditions** on a rule, e.g., `goal += f(X) for X < 10`. This is syntactic sugar for `goal += f(X) * lessthan(X,10)`, where the value of each $\texttt{lessthan}(a,b)$ term is the one or zero element of the semiring, according to whether $a < b$ or not.

3 Program Analysis

Our goal is to search for a *fast* Dyna program. We will assume that the programs are executed using the forward chaining algorithm described by Eisner et al. (2005).[7] In principle, we could evaluate a candidate program's runtime by actually executing it, but this would be very expensive and would also require us to specify particular inputs to the program. Instead, as our search objective, we will use a simple asymptotic upper bound on the program's runtime, based on a folk theorem from the Datalog community that has a long history of use. Many NLP papers have analyzed the runtime of their algorithms using either this folk theorem or a more refined version given by McAllester (2002): Gildea (2011); Nederhof and Satta (2011); Gilroy et al. (2017); Melamed (2003); Kuhlmann (2013); Nederhof and Sánchez-Sáez (2011); Büchse et al. (2011); Lopez (2009); Eisner and Blatz (2007).

The folk theorem says that, under certain conditions (discussed later), the running time of forward chaining execution of a given program is at worst linear in the number of ways to instantiate its rules, i.e., bind the variables to constants. A relatively simple bound on rule instantiations is available if we can establish that each variable in the program can be bound in at most η different ways. In that case, given some other conditions discussed at the end of this section, the number of ways to instan-

[4] If the reader is not familiar with context-free parsing, we recommend Jurafsky and Martin (2020, chapters 12–13).

[5] Many other semirings are useful in NLP (Goodman, 1999; Huang, 2009; Eisner et al., 2005).

[6] We leave the extensions in Dyna 2 (Eisner and Filardo, 2011), which relaxes this restriction, to future work.

[7] This algorithm assumes that the program is range-restricted, i.e., $\mathsf{vars}(\mathsf{head}(r)) \subseteq \mathsf{vars}(\mathsf{body}(r))$. An example of a non-range restricted rule is `id(I,I) += 1`.

tiate a rule with k variables is bounded by $\mathcal{O}(\eta^k)$. Program $\mathcal{P}$'s total runtime is $\mathcal{O}(\eta^{\text{degree}(\mathcal{P})})$ where degree($\mathcal{P}$) is the maximum number of variables in any rule of $\mathcal{P}$. We therefore take degree to be the cost function to minimize during search.

Consider Example 1. Evaluating this program under the forward-chaining algorithm will instantiate the rule by binding the 5 variables X_1, X_2, X_3, X_4, X_5 to constants. Then, the number of rule instantiations is $\mathcal{O}(\eta^5)$.

Similarly, Example 3 runs in $\mathcal{O}(\eta^6)$, as the first rule must sum over 6 variables, X, Y, Z, I, J, K. Note that this is a coarse-grained analysis: the runtime is usually given more specifically as $\mathcal{O}(n^3 K^3)$ where n is the number of sentence positions, and K is the number of grammar symbols. (This finer-grained bound can be achieved by the theorem of McAllester (2002): the intuition is that the variables I, J, K can each be bound in $\mathcal{O}(n)$ ways while X, Y, Z can each be bound in $\mathcal{O}(K)$ ways.) However, the simpler analysis $\mathcal{O}(\eta^6)$ gives us a single exponent to reduce, namely 6.

To see the cost function in action, consider that Example 1 has a running time of $\mathcal{O}(\eta^5)$, whereas the following equivalent program runs in $\mathcal{O}(\eta^2)$.

Example 4. *Efficient factorization of Example 1*

```
27  z += rest₁(Y₁).
28  rest₁(Y₁) += w₁(Y₁,Y₂) * rest₂(Y₂).
29  rest₂(Y₂) += w₂(Y₂,Y₃) * rest₃(Y₃).
30  rest₃(Y₃) += w₃(Y₃,Y₄) * rest₄(Y₄).
31  rest₄(Y₄) += w₄(Y₄,Y₅) * rest₅(Y₅).
```

Similarly, in Example 3, we can sum over the variable Y separately from K as follows:

Example 5. *Faster CKY (Example 3)*

```
32  β(X,I,K) += tmp(I,J,X,Z) * β(Z,J,K).
33  tmp(I,J,X,Z) += β(Y,I,J) * γ(X,Y,Z).
```

which is more efficient as its running time is $\mathcal{O}(\eta^5)$. It is also more efficient under the finer-grained analysis, $\mathcal{O}(K^2 n^3 + K^3 n^2)$.

The degree analysis of a Dyna program only leads to a *valid* $\mathcal{O}$-expression under some conditions, which we will now discuss. (1) The degree bound requires the *grounded* program to be **acyclic** (Eisner et al., 2005). Cycles slow down forward chaining because it must iterate to a numerical fixed point. Generally, the number of iterations required to reach a fixed-point is data dependent.[8] (2) The degree bound assumes that all of the relations in the program are **bounded** in size. The degree bound requires that terms are not nested; this prevents the user from encoding infinite sets, such as the Peano integers. Additionally, it assumes that the program's rules are all range-restricted (footnote 7). (3) The degree bound also assumes that the semiring operations are constant time.

We will see in §6 that simply optimizing degree is sufficient to recover a number of asymptotic speedups noted in the NLP literature (see §1) as well as asymptotic speedups on synthetic programs.

That said, the degree analysis might be loose for many reasons. The upper bounds derived using our methodology assume that relations are dense. Often relations are statically known to be sparse. Many low-level details affect actual execution time, but do not matter for asymptotic complexity. For example, memory layouts (e.g., row-order or column-order layout of a dense array in memory), sparse vs. dense representations of relations (e.g., hash tables vs. arrays), and indexes on relations (including sorted order) can have a dramatic effect on the running time in practice. However, they will not manifest in the degree analysis (e.g., Bilmes et al. (1997); Dunlop et al. (2011); DeNero et al. (2009); Lopez (2007)). Such choices are out of the control of our specific search space, but they may interact with the program in ways that are not represented in the degree.

An obvious alternative cost function would be the empirical execution time of executing the transformed program on a workload of representative inputs (e.g., running a transformed parser on actual sentences from the Penn Treebank (Marcus et al., 1993)). But as we noted earlier, such a cost function might be impractically expensive. For example, evaluating the degree of a degree-1000 program is linear in the size of the program, whereas evaluating the wallclock time is $\mathcal{O}(\eta^{1000})$. Optimizing the program degree is a crucial design choice as it enables a more exhaustive search in practice. Additionally, it sidesteps the need to optimize for a specific workload. However, in future work, we would like to investigate hybrid search algorithms (e.g., Song et al. (2019)) that do attempt to minimize empirical execution time, but replace some of the expensive evaluations of that with cheaper approxi-

[8] In the Boolean case (i.e., a Datalog program), the cycles do not affect the running time because finding "new" values for an item that is already true does not trigger further propagation to items that depend on it. Unfortunately, this is not true of general semirings. For example, the program `a += r * a. a += 1.` encodes a geometric series; it may take many iterations to converge if $|\mathtt{r}|$ is close to 1, and will diverge if $|\mathtt{r}| \geq 1$. The efficiency of this cyclic program thus depends on the value of the input parameter `r`.

mations: empirical execution time but with a time-out, fine-grained bounds obtained by abstract interpretation, and—most cheaply—estimates derived from worst-case asymptotic analysis as above.

4 Program Transformations

This section details the set of transforms that we consider in this paper. None of the transforms are novel to this work. They have been detailed and proved correct in Eisner and Blatz (2007). We include them in our discussion for completeness of presentation and to illuminate the challenges of our search problem. We provide pseudocode for the transforms in App. B, but defer to Eisner and Blatz (2007) for a more thorough discussion.

Input and output declarations. We assume that the initial program declares some items as input and/or output items. The rest are considered intermediate items. A program transform must preserve the mapping from a valuation of the input items to a valuation of the output items. (A valuation is an assignment of a value to each item.) However, a program transform is free to introduce, destroy, or alter intermediate items.

For example, for CKY, the input and output items are declared as follows:

```
34 input word(X,I,K); γ(X,Y,Z); γ(X,Y).
35 output goal.
```

4.1 Fold

Examples 1 to 3 were examples of the folding transform. Our candidate fold actions are based on **variable elimination**, as these are the only valid folding actions that reduce the rule's degree. For a given rule r, the variable $v \in \mathsf{vars}(r)$ *can be eliminated* if it does not appear in *all* of the factors in the rule's body and it does not appear in the head of the rule. Formally, the set of such variables is $\mathsf{elim}(r) \overset{\text{def}}{=} \{v \mid \mathsf{factors}(r, v) \neq \mathsf{body}(r), v \notin \mathsf{head}(r)\}$ where $\mathsf{factors}(r, v) \overset{\text{def}}{=} \{b \mid b \in \mathsf{body}(r), v \in \mathsf{vars}(b)\}$.

If any rule r of the program $\mathcal{P}$ contains variables that can be eliminated ($|\mathsf{elim}(r)| > 0$), then eliminating any variable $v \in \mathsf{elim}(r)$ by folding $\mathsf{factors}(r, v)$ out of r reduces that r's degree, which may reduce (and never increases) the degree of the program. Therefore, no final program benefits from having rules with variables that can be eliminated. However, when more than one variable can be eliminated ($|\mathsf{elim}(r)| > 1$), the order in which the variables are eliminated will affect the eventual degree. Finding an optimal sequence of variable-elimination steps is NP-hard, by reduction from variable elimination ordering in probabilistic graphical models (Gildea, 2011).

We briefly note that folding can increase the space complexity of the program, since it introduces intermediate items that will be stored. We do not consider optimizing the space–time tradeoff, but it could be done with methods similar to ours.

4.2 Unfold and Rule Elimination

Suppose the user provided an inefficient program, such as Example 6, which could have been obtained by folding Example 1 with a suboptimal variable-elimination ordering.

Example 6. *Bad ordering for Example 1*

```
36 goal += tmp₁(X₁,X₄,X₅).
37 tmp₄(X₁,X₂) += w₁(X₁,X₂).
38 tmp₃(X₂,X₄) += w₂(X₂,X₃) * w₃(X₃,X₄).
39 tmp₁(X₁,X₄,X₅) += tmp₂(X₁,X₄) * w₄(X₄,X₅).
40 tmp₂(X₁,X₄) += tmp₄(X₁,X₂) * tmp₃(X₂,X₄).
```

While the above program is correct, its degree is 3, which is worse than the optimal variant, which has degree 2. It has no variables that can be eliminated, so there are no fold actions that can improve its degree. To improve it, we first have to *undo* the poor choices. There are two transformations for "undoing" folds: unfold and rule elimination, which we will describe in this section.

The **unfold** transform is essentially the inverse of the fold transformation, and corresponds to inlining code. It takes as input a specific subgoal $\mathtt{b_k} \in \mathsf{body}(r)$ of some rule r. The goal is to replace $\mathtt{b_k}$ by its definition. We will remove r from the program, and replace it by adding a specialized version of r for each rule r' whose head unifies with $\mathtt{b_k}$. These rules r' define $\mathtt{b_k}$—except in the special case where $\mathtt{b_k}$ matches any input items, in which case we cannot unfold $\mathtt{b_k}$ because its complete definition is not available.

As a simple example, consider unfolding the first subgoal of the first rule of Example 5,

```
41 β(X,I,K) += tmp(I,J,X,Z) * β(Z,J,K).
42 tmp(I,J,X,Z) += β(Y,I,J) * γ(X,Y,Z).
```

This becomes

```
43 β(X,I,K) += β(Y,I,J) * γ(X,Y,Z) * β(Z,J,K).
44 tmp(I,J,X,Z) += β(Y,I,J) * γ(X,Y,Z).
```

Notice that the second rule is now defunct.[9] This

[9] We make use of a simple dead rule detection strategy to identify rules that cannot fire based on the declared inputs, or are unused by any of the declared outputs. Determining which rules are dead is possible with a straightforward graph

transformation is correct by the distributive rule. Notice that the degree increased from 5 to 6.

An unfold will usually increase or preserve the program's degree—but there are exceptions due to repeated variables or constants in rules:

```
45  a(I,K) += b(I,J) * c(J,K).
46  trace += a(L,L).
```

Unfolding the subgoal of the second rule decreases the program's degree from 3 to 2:

```
47  trace += b(L,J) * c(J,L).
```

Thus, this is an example of an immediately useful unfold. The program in this case computes the trace of a matrix product, and the role of unfold is to *specialize* the sub-program that computes the entire matrix product a(I,K) to the site where the product is used, which only seeks its diagonal a(L,L). Such optimizations are easy for programmers to miss.

Rule elimination is an alternative transformation, also described by Eisner and Blatz (2007), that happens to achieve the same result in the above examples. As the name suggests, rule elimination targets rules instead of subgoals. The transform takes as input a rule r', removes it from the program, and adds specialized versions of all of the rules r whose subgoals match the head of r'. Rule r' cannot be eliminated if its head matches any output items, since that would change the value of those output items. Notice that attempting to eliminate recursive rules is futile, as a rule cannot be eliminated until it reaches its base case.

Rule elimination and unfold are especially useful for eliminating non-range-restricted rules (footnote 7). For example, eliminating the first rule from

```
48  f(I) += 1.
49  f(I) += g(I) * m(I,J).
50  goal += f(I) * h(I).
```

yields the range-restricted program

```
51  f(I) += g(I) * m(I,J).
52  goal += f(I) * h(I).
53  goal += 1 * h(I).
```

Another useful case of rule elimination is for propagating constants throughout the program. For example, if the grammar in Example 3 is known in advance, i.e., $\gamma(\mathtt{X},\mathtt{Y},\mathtt{Z}) \notin \mathsf{inputs}$, then we can propagate them ahead of time. Yielding a highly specialized program with no X, Y, Z variables and having an overall reduced degree of 3.

In order to recover the original version of Example 1 given Example 6, we can eliminate all rules except for the one defining the output item z, and then fold to eliminate variables in a different order. This poses search challenges because all of the unfold or rule elimination actions needed to reach Example 1 are "uphill": they increase the degree, until the folds are applied. This means that finding useful unfold and rule elimination moves can be challenging (i.e., take a fair amount of exploration).

reachability analysis on a coarsened program, such as the program resulting from dropping the arguments to all relations, known as the predicate graph.

5 Program Improvement

Our goal is to find a sequence of transformations to the user's program $\mathcal{P}_0$ that gives the lowest cost, $\mathsf{cost}(\mathcal{P}) \overset{\text{def}}{=} \mathsf{degree}(\mathcal{P})$. In this section, we provide two effective search algorithms for approaching this goal: beam search and Monte Carlo tree search.

5.1 The Graph Search Problem

We consider an abstract **graph search problem**, $\langle \mathcal{S}, \mathcal{A}, s_0, \mathcal{T}, \mathsf{transition}, \mathsf{cost}\rangle$, where $\mathcal{S}$ is a state space, $\mathcal{A}$ is an action space, $\mathsf{transition} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is a transition function, $s_0 \in \mathcal{S}$ is an initial state, $\mathcal{T} \subseteq \mathcal{S}$ is a set of terminal states, and $\mathsf{cost} : \mathcal{T} \to \mathbb{R}_{\geq 0}$ is an cost function on the terminal states. The cost and transition functions will be treated by MCTS and beam search as black boxes.[10] The goal of the search problem is to find the terminal state in the graph that has the lowest cost, $s^* = \min_{s \in \mathcal{T}} \mathsf{cost}(s)$.

Our problem (depicted in Fig. 1) can be easily mapped into this notation. Our states $\mathcal{S}$ are programs (§2). The initial state s_0 is the initial program $\mathcal{P}_0$. The transitions are applications of any valid program transformation, which we discussed in §4. In our setting, every state is a terminal. For the cost function, we use program's degree (§3). To ensure termination, we only explore up to a distance of 100 from the initial state. We will discuss in §5.4 how to structure the state and action spaces to make search more effective.

5.2 Beam Search

Beam search is a common heuristic search algorithm, which is easy to implement (Fig. 2) and often works well in practice. A terse description of the algorithm is that it is a variant of breadth-first search (Russell and Norvig, 2020) which prunes the search frontier (FIFO queue) to only keep the

[10]MCTS can be used in the more general case where cost and transition are stochastic or adversarial functions, but this is not the case in our setting.

```
1.  def beam_search(s_0, B):
2.    beam ← [s_0]
3.    while beam :
4.      beam′ ← [ ]
5.      for s ∈ beam :
6.        for a ∈ A(s) :
7.          s′ ← transition(s, a)
8.          beam′.append(s′)
9.      beam ← B lowest-cost elements of beam′
10.   return lowest-cost state ever to appear in beam
```

Figure 2: Beam search algorithm

B lowest cost states so far.[11] However, the pruning also robs breadth-first search of any guarantees. Increasing the beam size B generally returns a lower-cost terminal state, but occasionally it may result in incorrectly pruning a good state and thus returning a higher-cost terminal state (as we see in our experiments). We do recover exhaustive breadth-first search, which is guaranteed correct, when the beam size is large enough that *no* pruning is done.

5.3 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a learning based algorithm (Kocsis and Szepesvári, 2006; Coulom, 2007), is considered a "major breakthrough" in computer Go (Swiechowski et al., 2021; Gelly et al., 2012; Silver et al., 2016). MCTS is an uninformed search algorithm, which means that it is classified alongside well-known algorithms such as breadth-first search, beam search, and iterative deepening search. However, since MCTS is based on learning, it has some of the benefit of an informed search algorithm, such as A* (Hart et al., 1968), without the burden of designing or evaluating a heuristic function. Essentially, it *learns* its heuristic by sampling sequences of actions. For our specific search problem, designing an A* heuristic that works well with all of our search actions (especially unfold and rule elimination) is challenging. We discuss our choice further in §5.5.

The application of MCTS to graph search is summarized in Fig. 3. For thorough surveys on MCTS, we refer the reader to the surveys by Swiechowski et al. (2021) and Browne et al. (2012).

MCTS searches by estimating the expected cost-to-go for taking action a in a given state s, $\frac{\widehat{c}(s,a)}{\widehat{n}(s,a)}$ where $\widehat{c}(s,a)$ is the total cost of previous attempts

```
1.  def mcts(s_0, C, R):
2.    mode ← explore
3.    repeat R times: mcts′(s_0)
4.    mode ← deploy
5.    return mcts′(s_0)
6.  def mcts′(s):
7.    ▷ Terminal state, return cost and final state
8.    if s ∈ T : return ⟨s, cost(s)⟩
9.    a ← π(s)
10.   ▷ Transition to new state
11.   ⟨s*, c⟩ ← mcts′(transition(s, a))
12.   update(s, a, c)
13.   return ⟨s*, c⟩
14. def π(s, A):
15.   if n̂(s) = 0 :          ▷ Novel state, follow initial policy
16.     return π_0(s)
17.   else if mode = explore :
18.     return argmin_{a∈A} ĉ(s,a)/n̂(s,a) − C sqrt(log n̂(s) / n̂(s,a))
19.   else                           ▷ Deploy mode
20.     return argmax_{a∈A} n̂(s, a)
21. def update(s, a, c):
22.   ▷ Update MCTS statistics after observing cost
23.   ĉ(s, a) += c ; n̂(s, a) += 1 ; n̂(s) += 1
```

Figure 3: Search algorithm

of action a, and $\widehat{n}(s,a)$ is the total number of such attempts. In order to *learn* a **policy** π that maps states to actions, MCTS selects the action a in state s that minimizes the **lower-confidence bound**,

$$\frac{\widehat{c}(s,a)}{\widehat{n}(s,a)} - C\sqrt{\frac{\log \widehat{n}(s)}{\widehat{n}(s,a)}} \tag{1}$$

In state s, MCTS chooses the action a that minimizes (1). This bound treats actions optimistically in the face of uncertainty: if an action in state s has been under-explored, its cost *might* be rather lower than the noisy average cost observed for it so far, and so MCTS may be willing to try it again.

The constant $C > 0$ is a tunable constant that controls the exploration–exploitation tradeoff.[12] If $\widehat{n}(s,a) = 0$, the lower confidence bound is defined to be $-\infty$; thus, novel actions are always explored if there are any. In this paper, we use MCTS as a batch search algorithm. That is why the top-level MCTS routine includes a repeat-loop (line 3) and switches the policy into deployment mode (line 4). Notice that when the policy is in deployment mode, it *exploits* by selecting the most frequently explored action (line 20).[13] Lastly, we note if MCTS is run for sufficiently long and the

[11] In our experiments, we break ties encountered on line 9 by comparing programs according to following sort key: $\langle d_1, \ldots, d_N \rangle$ where d_i is i^{th} largest in the program with N rules. This comparison has the benefit that it will minimize lower degree terms as well, which may better guide search.

[12] In our experiments, we set C to equal the degree of the initial program. This is close to the theoretical requirement of an upper bound on the range (max - min) of the cost function.

[13] It is also reasonable to use the action with the lowest estimated cost. However, this choice is less stable in practice.

constant C is set appropriately, it will converge to an optimal transformation sequence (Kocsis and Szepesvári, 2006).

Initial Policy Design MCTS can be greatly sped up using the following strategy: on the first visit to a state (i.e., $\widehat{n}(s) = 0$), we redirect control to an **initial policy** π_0 rather than *uninformed* exploration (that which results from following the lower confidence bound). This results in a sensible initial value for the cost-to-go estimate. For our initial policy, we randomly fold all rules until there are no more fold actions available.

5.4 Refinements to the Search Graph

In the sections describing each of the transforms, we discussed conditions for the transforms to be valid. The basic version of the search graph would simply say that all valid transforms from §4 are available at all times. However, that would ignore some useful problem structure. In this section, we propose two refinements to the search space: rule to-do lists and macro folding. We validate their empirical utility in §6.

Rule to-do list. Each of the transforms we consider is centered around a specific rule in the current version of the transformed program. Applying transforms to rules r and r' in either order will get the same result if neither transform makes the other one impossible. Thus, we consider transforms in a canonical order. Each state will now consist of a program together with **to-do list** of rules that can still be transformed. The possible actions at that state consist of either removing the top rule r from the list (declining to transform r) or applying a program transform (fold, unfold, or elimination) that is centered on r. Applying such a transform may delete and/or add program rules, which are correspondingly deleted from the list and/or added at the bottom of the list. (If the list is empty, no more actions are possible.) This design reduces the branching factor by a factor of the number of rules, and improves the sharing of statistics at nodes in the MCTS search tree. Of course, a potential downside is that it makes the search tree deeper by a factor of the number of rules.

Macro folding. Our most important refinement is to use macro folding actions. These actions will take a given rule r and completely fold it *independently* from the main program allowing us to memoize it. More precisely, macro-folding runs the program containing the single rule r through search $\mathcal{P}'_r \leftarrow \mathsf{search}_{\text{fold-only}}([r])$, and then merges the solution into the main program, $(\mathcal{P} - r) \cup \mathcal{P}'_r$. Macro folding provides an exponential reduction in the size of the search space because it allows any given rule to be optimized by folding independently of the other rules in the program. Thus, if a given rule appears in multiple program variants, we can re-use knowledge acquired from folding it in other contexts to fold it in the current context—analogous to memoizing the best folding sequence for each rule. The macro folding action is implemented in our graph search instance as yet another action.[14] However, unlike the other transforms, macro folding is useful to memoize as it is reusable across many of the programs explored during search.

5.5 Discussion

Our goal in this work was to exhibit a working method (not necessarily the best one). But since our search problem is just graph search, why not simply use a classical method like A* (Hart et al., 1968)? The challenge is in designing an admissible and effective A* heuristic. The role of the heuristic is to approximate lookahead. MCTS does not need a hand-designed heuristic because it instead performs lookahead by actual rollouts. The average cost of these rollouts is still only an approximation of the optimal cost-to-go, because the rollouts use the current exploration policy—but it approaches the optimal cost-to-go as the algorithm continues to run. MCTS has been previously used for graph search (Wang et al., 2020; Negrinho et al., 2019).

Could an A* heuristic be designed in our setting? There are many good search heuristics (e.g., Gogate and Dechter (2004)) in the special case where only folding actions are allowed. However, for unfold and rule elimination, the heuristics are difficult to derive. The challenge with these actions is that they are always uphill moves with delayed benefits: it is often the case that we require several unfolds, each increasing the degree, followed by several (potentially tricky) folds.

6 Experiments

The goal of this paper was to devise a system for automatically improving typical dynamic programming problems. To evaluate whether we achieved this goal, we devised a set of *unit tests* and *stress*

[14]When we enable macro folding in our experiments, we disable the basic fold action since they are redundant.

benchmark	avg rel degree		% optimal	
	beam	mcts	beam	mcts
bar-hillel	1.00	1.00	100	100
bilexical-labeled	0.97	1.00	90	100
bilexical-unlabeled	1.00	0.99	100	90
chain-10	1.00	1.00	100	100
chain-20	1.00	1.00	100	100
chain-expect	1.00	1.00	100	100
cky+grammar	0.74	0.68	40	40
cky3	0.99	0.99	90	90
cky4	0.97	0.96	90	80
edit	1.00	0.99	100	90
hmm	1.00	1.00	100	100
itg	0.98	0.95	90	60
path	1.00	1.00	100	100
semi-markov	1.00	1.00	100	100
split-head	0.99	0.99	90	90

Table 1: Experimental results for stress test experiments. Each row is a class of 10 randomly constructed, semantically equivalent input programs. • % optimal: the percentage of the 10 random programs for which we find the optimal degree within the search budget ($R = 300K$ iterations for MCTS, $B = 1000$ for beam search). In many rows, this is 100%. However, in some recursive cases with two or more recursive subgoals, the randomly applied unfolds make the programs very big, which makes them difficult to optimize. Both methods performed poorly on the cky+grammar benchmark, which is by far the longest program we consider as it contains 35 rules in its original form. • Average relative degree: The relative degree achieved by search, averaged over the 10 random programs. We see that this metric follows the same trend as % optimal. • Overall, we see a small but consistent improvement of beam_search over mcts (under both metrics), except on bilexical-labeled.

tests to see how well our proposed approach works.

Unit tests. Our unit tests include most of the faux pas mentioned in §1; the precise set of programs is provided in App. A. However, recovering these instances is relatively easy as they typically only require a few fold transformations. Thus, they are not a good stress test for our automated system. In all of our test cases, we know the optimal degree, and we have verified that both MCTS (with $R = 100$ iterations) and beam search (with a beam of size $B = 10$) successfully find it within a few seconds.

Stress tests. The inspiration for our stress tests is to imagine that a naïve programmer produces a suboptimal program. For example, they may have chosen the poor variable-elimination order in Example 6, and, now, we would like to "undo" their handiwork via unfold, elimination, and fold. We operationalize such a naïve programmer by imaging that they have applied a sequence of random folds and unfolds to a starting point program. More precisely, for each program $\mathcal{P}_0$ in our unit test suite with known optimal degree d^*, we generate an inefficient variant $\mathcal{P}_1$ for search to improve. The variant $\mathcal{P}_1$ is generated by applying a random sequence of transformations to $\mathcal{P}_0$, but we reject $\mathcal{P}_1$ if the optimal degree d^* can be "trivially" achieved by applying the greedy fold-only algorithm to $\mathcal{P}_1$. The results of this experiment are summarized in Table 1.

We compare the search algorithms according to the percentage of stress tests that they are able to solve optimally. We also consider the **relative degree**, i.e., the fraction of the possible improvement that was actually achieved: $\frac{\text{degree}(\mathcal{P}_1)-\text{degree}(\mathcal{P})}{\text{degree}(\mathcal{P}_1)-d^*}$.

search	B	todo	macro	avg rel deg	% optimal
beam	100	−	−	96	87
		+	−	93	75
		+	+	97	93
	1000	−	−	90	74
		+	−	94	77
		+	+	98	93
mcts		−	−	97	83
		+	−	96	80
		+	+	97	89

Table 2: Ablation of search-space refinements. Like Table 1, we show % optimal and average relative degree, except here we average together all benchmarks to see an overall picture. The rows are labeled by their search method and whether that search method operates on a search graph with a to-do list and/or macro folding. Overall, we see that the proposed refinements improve overall performance under both metrics, except that smaller beams are disadvantaged by the increased depth of the to-do list refinement. When the macro folds are added to the small beam, performance hits the same peak as the system with a larger beam.

Ablation analysis. We explore the utility of our propose search space refinements (§5.4) in Table 2.

7 Conclusion

We have presented a system for automatically analyzing and improving dynamic programming algorithms. Expressing those algorithms in the Dyna programming language allows us to successively apply program transformations introduced by Eisner and Blatz (2007). We showed that Monte Carlo tree search and beam search allows us to automatically discover asymptotically faster algorithms.

Acknowledgments

We wish to thank Matthew Francis-Landau, Ran Zmigrod, Nathaniel Wesley Filardo, and the anonymous reviewers for their helpful feedback and suggestions that improved this paper.

References

J. K. Baker. 1979. Trainable grammars for speech recognition. In *Speech Communication Papers Presented at the Meeting of the Acoustical Society of America*.

Yehoshua Bar-Hillel, Micha Asher Perles, and Eli Shamir. 1961. On formal properties of simple phrase-structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14.

Jeff Bilmes, Krste Asanović, Chee-Whye Chin, and Jim Demmel. 1997. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of the International Conference on Supercomputing*.

Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. 2012. A survey of Monte Carlo tree search methods. *IEEE Transations on Computational Intelligence and AI Games*, 4(1).

Matthias Büchse, Mark-Jan Nederhof, and Heiko Vogler. 2011. Tree parsing with synchronous tree-adjoining grammars. In *International Workshop on Parsing Technologies*.

John Cocke and Jacob T. Schwartz. 1970. Programming languages and their compilers: Preliminary notes. Technical report, Courant Institute of Mathematical Sciences, New York University.

Michael John Collins. 1996. A new statistical parser based on bigram lexical dependencies. In *Proceedings of the Conference of the Association for Computational Linguistics*.

Ryan Cotterell, Nanyun Peng, and Jason Eisner. 2014. Stochastic contextual edit distance and probabilistic FSTs. In *Proceedings of the Conference of the Association for Computational Linguistics*.

Rémi Coulom. 2007. Efficient selectivity and backup operators in Monte-Carlo tree search. In *Computers and Games*. Springer Berlin Heidelberg.

John DeNero, Mohit Bansal, Adam Pauls, and Dan Klein. 2009. Efficient parsing for transducer grammars. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics and Human Language Technology*, pages 227–235.

Timothy Dozat and Christopher D. Manning. 2017. Deep biaffine attention for neural dependency parsing. In *Proceedings of the International Conference on Learning Representations*.

Aaron Dunlop, Nathan Bodenstab, and Brian Roark. 2011. Efficient matrix-encoded grammars and low latency parallelization strategies for CYK. In *International Workshop on Parsing Technologies*.

Greg Durrett and Dan Klein. 2015. Neural CRF parsing. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics and the International Joint Conference on Natural Language Processing*.

Jason Eisner. 2002. Parameter estimation for probabilistic finite-state transducers. In *Proceedings of the Conference of the Association for Computational Linguistics*.

Jason Eisner and John Blatz. 2007. Program transformations for optimization of parsing algorithms and other weighted logic programs. In *Proceedings of the Conference on Formal Grammar*.

Jason Eisner and Nathaniel W. Filardo. 2011. Dyna: Extending Datalog for modern AI. In Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors, *Datalog Reloaded*, volume 6702 of *Lecture Notes in Computer Science*. Springer. (Longer version available as tech report).

Jason Eisner, Eric Goldlust, and Noah A. Smith. 2005. Compiling comp ling: Weighted dynamic programming and the Dyna language. In *Proceedings of the Joint Conference on Human Language Technology Conference and Empirical Methods in Natural Language Processing*.

Jason Eisner and Giorgio Satta. 1999. Efficient parsing for bilexical context-free grammars and head automaton grammars. In *Proceedings of the Conference of the Association for Computational Linguistics*.

Jason Eisner and Giorgio Satta. 2000. A faster parsing algorithm for lexicalized tree-adjoining grammars. In *Proceedings of the International Workshop on Tree Adjoining Grammar and Related Frameworks*.

Jason M. Eisner. 1996. Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of the Conference of the Association for Computational Linguistics*.

Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michèle Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. 2012. The grand challenge of computer Go: Monte Carlo tree search and extensions. *Communications of the ACM*, 55(3).

Daniel Gildea. 2011. Grammar factorization by tree decomposition. *Computational Linguistics*, 37(1).

Sorcha Gilroy, Adam Lopez, and Sebastian Maneth. 2017. Parsing graphs with regular graph grammars. In *Proceedings of the Joint Conference on Lexical and Computational Semantics (*SEM 2017)*.

Vibhav Gogate and Rina Dechter. 2004. A complete anytime algorithm for treewidth. In *Proceedings of the Converence on Uncertainty in Artificial Intelligence*.

Joshua Goodman. 1999. Semiring parsing. *Computational Linguistics*, 25(4).

Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2).

Jacques Herbrand. 1930. *Recherches sur la Théorie de la Demonstration*. Ph.D. thesis, Université de Paris.

Juneki Hong and Liang Huang. 2018. Linear-time constituency parsing with RNNs and dynamic programming. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

Liang Huang. 2009. Dynamic programming-based search algorithms in NLP. In *Proceedings of Human Language Technologies: The Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts*.

Liang Huang, Hao Zhang, and Daniel Gildea. 2005. Machine translation as lexicalized parsing with hooks. In *International Workshop on Parsing Technologies*.

Frederick Jelinek. 1985. Markov source modeling of text generation. In *NATO Advanced Study Institute: Impact of Processing Techniques on Communication*.

Mark Johnson. 1989. The computational complexity of Tomita's algorithm. In *International Workshop on Parsing Technologies*.

Mark Johnson. 2007. Transforming projective bilexical dependency grammars into efficiently-parsable CFGs with unfold-fold. In *Proceedings of the Conference of the Association for Computational Linguistics*.

Dan Jurafsky and James H. Martin. 2020. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, Third Edition*.

Tadao Kasami. 1965. An efficient recognition and syntax algorithm for context-free languages. Technical report, Air Force Cambridge Research Laboratory.

Yoon Kim, Carl Denton, Luong Hoang, and Alexander M. Rush. 2017. Structured attention networks. In *Proceedings of the International Conference on Learning Representations*.

Kevin Knight. 1989. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1).

Levente Kocsis and Csaba Szepesvári. 2006. Bandit based Monte-Carlo planning. In *Machine Learning*. Springer Berlin Heidelberg.

Terry Koo and Michael Collins. 2010. Efficient third-order dependency parsers. In *Proceedings of the Conference of the Association for Computational Linguistics*.

Marco Kuhlmann. 2013. Mildly non-projective dependency grammar. *Computational Linguistics*, 39(2).

Marco Kuhlmann, Giorgio Satta, and Peter Jonsson. 2018. On the complexity of CCG parsing. *Computational Linguistics*, 44(3).

John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. 2001. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the International Conference on Machine Learning*.

Martin Lange and Hans Leiß. 2009. To CNF or not to CNF? An efficient yet presentable version of the CYK algorithm. *Informatica Didactica*, 8.

Kenton Lee, Mike Lewis, and Luke Zettlemoyer. 2016. Global neural CCG parsing with optimality guarantees. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Vladimir I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 8.

Zhifei Li and Jason Eisner. 2009. First- and second-order expectation semirings with applications to minimum-risk training on translation forests. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*.

Adam Lopez. 2007. Hierarchical phrase-based translation with suffix arrays. In *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*.

Adam Lopez. 2009. Translation as weighted deduction. In *Proceedings of the Conference of the European Association for Computational Linguistics*.

Xuezhe Ma and Hai Zhao. 2012. Fourth-order dependency parsing. In *Proceedings of the International Conference on Computational Linguistics*.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330.

Alberto Martelli and Ugo Montanari. 1982. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282.

Elena Mastria, Jessica Zangari, Simona Perri, and Francesco Calimeri. 2020. A machine learning guided rewriting approach for ASP logic programs. In *Proceedings of the International Conference on Logic Programming (Technical Communications)*, volume 325 of *EPTCS*.

David McAllester. 2002. On the complexity analysis of static analyses. *Journal of the ACM*, 49(4).

Clara Meister, Tim Vieira, and Ryan Cotterell. 2020. Best-first beam search. *Transactions of the Association for Computational Linguistics*, 8:795–809.

I. Dan Melamed. 2003. Multitext grammars and synchronous parsers. In *Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*.

Mehryar Mohri. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2).

Mark-Jan Nederhof and Ricardo Sánchez-Sáez. 2011. Parsing of partially bracketed structures for parse selection. In *International Workshop on Parsing Technologies*.

Mark-Jan Nederhof and Giorgio Satta. 2011. Prefix probability for probabilistic synchronous context-free grammars. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics.

Renato Negrinho, Matthew R. Gormley, Geoffrey J. Gordon, Darshan Patil, Nghia Le, and Daniel Ferreira. 2019. Towards modular and programmable architecture search. In *Advances in Neural Information Processing Systems*.

Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, and Christopher D. Manning. 2020. Stanza: A python natural language processing toolkit for many human languages. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics: System Demonstrations*.

Lawrence R Rabiner. 1989. A tutorial on hidden Markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2).

Pushpendre Rastogi, Ryan Cotterell, and Jason Eisner. 2016. Weighting finite-state transductions with neural context. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*.

Raj Reddy. 1977. Speech understanding systems: A summary of results of the five-year research effort at Carnegie Mellon University. Technical report, Carnegie Mellon University.

John Alan Robinson. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1).

Alexander Rush. 2020. Torch-Struct: Deep structured prediction library. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics.

Stuart J. Russell and Peter Norvig. 2020. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson.

Sunita Sarawagi and William W. Cohen. 2004. Semi-markov conditional random fields for information extraction. In *Advances in Neural Information Processing Systems*, pages 1185–1192.

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587).

Jialin Song, Yuxin Chen, and Yisong Yue. 2019. A general framework for multi-fidelity Bayesian optimization with Gaussian processes. In *Proceedings of the Conference on Artificial Intelligence and Statistics*.

Mitchell Stern, Jacob Andreas, and Dan Klein. 2017. A minimal span-based neural constituency parser. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

Andreas Stolcke. 1995. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Linguistics*, 21(2).

Maciej Swiechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mandziuk. 2021. Monte Carlo tree search: A review of recent modifications and applications. *CoRR*, abs/2103.04931.

Masaru Tomita. 1985. An efficient context-free parsing algorithm for natural languages. In *Proceedings of the International Joint Conference on Artificial Intelligence*.

Masaru Tomita, editor. 1991. *Generalized LR Parsing*. Springer, Boston, MA.

K. Vijay-Shankar and Aravind K. Joshi. 1985. Some computational properties of tree adjoining grammars. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

K. Vijay-Shanker and David J. Weir. 1989. Recognition of combinatory categorial grammars and linear indexed grammars. In *International Workshop on Parsing Technologies*.

K. Vijay-Shanker and David J. Weir. 1990. Polynomial time parsing of combinatory categorial grammars. In *Proceedings of the Conference of the Association for Computational Linguistics*.

K. Vijay-Shanker and David J. Weir. 1993. Parsing some constrained grammar formalisms. *Computational Linguistics*, 19(4).

Andrew J. Viterbi. 1967. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2).

Linnan Wang, Yiyang Zhao, Yuu Jinnai, Yuandong Tian, and Rodrigo Fonseca. 2020. Neural architecture search using deep neural networks and Monte Carlo tree search. In *The AAAI Conference on Artificial Intelligence*.

Dekai Wu. 1996. A polynomial-time algorithm for statistical machine translation. In *Proceedings of the Conference of the Association for Computational Linguistics*.

Shijie Wu and Ryan Cotterell. 2019. Exact hard monotonic attention for character-level transduction. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*.

Shijie Wu, Pamela Shapiro, and Ryan Cotterell. 2018. Hard non-monotonic attention for character-level transduction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Daniel H. Younger. 1967. Recognition and parsing of context-free languages in time n^3. *Information and Control*, 10(2).

A Programs Used for Experiments

A.1 Chain Structures (chain-10, chain-20)

Chain structures are common in NLP as they model interaction between adjacent words or label in a sequence (e.g., Lafferty et al. (2001)).
The chain-N programs are adaptations of Example 1,

```
z += w(Y1,Y2) * w(Y2,Y3) * w(Y3,Y4) * w(Y4,Y5).
```

which is for the specific case of length 4, to have length N. The original program's degree is N. The optimal degree for chains is 2 regardless of N.

A.2 Projective Dependency Parsing

This program performs a version of CKY (App. A.3) where the non-terminals have been lexically annotated with their head word. In contrast to CKY's $\mathcal{O}(n^3)$ runtime, the original presentation of this algorithm ran in $\mathcal{O}(n^5)$ (Collins, 1996). In subsequent work Eisner and Satta (1999) gave a faster version of the algorithm that runs in $\mathcal{O}(n^4)$.

A.2.1 Bilexical Unlabeled (bilexical-unlabeled)

```
phrase(I,H,K) += phrase(I,H,J) * phrase(J,H',K) * score(H,H',left).
phrase(I,H',K) += phrase(I,H,J) * phrase(J,H',K) * score(H,H',right).
phrase(I,I,K) += word(I,I,K).
goal += phrase(0,_,N) * len(N).
input word(_,_,_); len(_); score(_,_,_).
output goal
```

Degree: 5. Optimal: 4.

A.2.2 Bilexical Labeled (bilexical-labeled)

Extends bilexical-unlabeled with labels (i.e., grammar relations).

```
% 0 right children so far
rconstit(X,H,I,K) += rewrite(X,H) * word(H,I,K).
% add right child
rconstit(X,H,I,K) += rewrite(X,H,Y,H,Z,H') * rconstit(Y,H,I,J) * constit(Z,H',J,K).
% 0 left children so far
constit(X,H,I,K) += rconstit(X,H,I,K).
% add left child
constit(X,H,I,K) += rewrite(X,H,Y,H',Z,H) * constit(Y,H',I,J) * constit(Z,H,J,K).
goal += constit(s,H,0,N) * len(N).
input word(_,_,_); len(_); rewrite(_,_,_,_,_,_).
output goal.
```

Degree: 8. Optimal: 7.

A.2.3 Split-Head-Factored (split-head)

References: (Johnson, 2007; Eisner and Blatz, 2007)

```
goal += x(0,_,N) * len(N).
% words are "duplicated" as in Johnson (2007).
l(I,K) += word(left,I,K).
r(I,K) += word(right,I,K).
% (I, K) is a span with K as the head
l(I,K) += x(I,V,J) * l(J,K) * score(left, V, K).   % V -> K
% (I, K) is a span with I as the head
r(I,K) += r(I,J) * x(J,V,K) * score(right, V, I).  % I <- V
% J is the head of l(I, J) and J is the head of r(J, K)
x(I,J,K) += l(I,J) * r(J,K).
input word(_,_,_); score(_,_,_). len(_).
output goal(_).
```

Degree: 4, Optimal: 3.

A.3 CKY

The following programs are variants of CKY (Cocke and Schwartz, 1970; Younger, 1967; Kasami, 1965; Lange and Leiß, 2009; Eisner and Blatz, 2007; Tomita, 1985, 1991; Johnson, 1989; Baker, 1979; Jelinek, 1985). We briefly discussed CKY in Example 3 of the main text.

A.3.1 CKY (cky3)

```
1  β(X,I,K) += γ(X,Y,Z) * β(Y,I,J) * phrase(Z,J,K).
2  β(X,I,K) += γ(X,Y) * β(Y,I,K).
3  β(X,I,K) += γ(X,Y) * word(Y,I,K).
4  z += β(root, 0, N) * len(N).
5  input word(W,I,K); len(N); γ(X,Y,Z); γ(X,Y)
6  output z
```

Degree: 6, Optimal: 5.

A.3.2 CKY with 4-ary Productions (cky4)

The following program implements Tomita (1985)'s algorithm where the grammar can have a production rule of length up to 4. It is a simple modification to CKY4. We just add the following rule, and input declaration.

```
7  phrase(X,I1,I4) += γ(X,Y1,Y2,Y3) * phrase(Y1,I1,I2) * phrase(Y2,I2,I3) * phrase(Y3,I3,I4).
8  input γ(X,Y1,Y2,Y3)
```

Degree: 7, Optimal: 6.

A.3.3 CKY with a Fixed Grammar (CKY+grammar)

For CKY+grammar, we use CKY3, but we remove the input declaration for γ and declare the following grammar for the program to specialize to.

```
9   γ("S", "NP", "VP") += 1.0.
10  γ("NP", "Det", "N") += 1.0.
11  γ("NP", "NP", "PP") += 1.0.
12  γ("VP", "V", "NP") += 1.0.
13  γ("VP", "V") += 1.0.
14  γ("VP", "VP", "PP") += 1.0.
15  γ("PP", "P", "NP") += 1.0.
16  γ("<.>", ".") += 1.0.
17  γ("NP", "Papa") += 1.0.
18  γ("N", "caviar") += 1.0.
19  γ("N", "spoon") += 1.0.
20  γ("N", "fork") += 1.0.
21  γ("N", "telescope") += 1.0.
22  γ("N", "boy") += 1.0.
23  γ("N", "girl") += 1.0.
24  γ("N", "baby") += 1.0.
25  γ("N", "man") += 1.0.
26  γ("N", "woman") += 1.0.
27  γ("N", "moon") += 1.0.
28  γ("N", "cat") += 1.0.
29  γ("V", "ate") += 1.0.
30  γ("V", "saw") += 1.0.
31  γ("V", "fed") += 1.0.
32  γ("V", "said") += 1.0.
33  γ("V", "jumped") += 1.0.
34  γ("P", "with") += 1.0.
35  γ("P", "over") += 1.0.
36  γ("P", "under") += 1.0.
37  γ("P", "above") += 1.0.
38  γ("P", "below") += 1.0.
39  γ("P", "on") += 1.0.
40  γ("P", "in") += 1.0.
```

Degree: 6, Optimal: 3.

A.3.4 Inversion Transduction Grammars (itg)

Inversion transduction grammars were introduced by Wu (1996), who gave an $\mathcal{O}(n^7)$ algorithm, which was later sped up to $\mathcal{O}(n^6)$ by Huang et al. (2005). The model is one that simultaneously parses a pair of related sentences—typically a target language sentence and a source language sentence as in machine translation. The model allows for a restricted form of syntactic reordering of phrases called inversion.

```
1  constit(A,I,K,I',K') += word(X,I,K) * word'(X',I',K') * transduce(A,X,X').
2  constit(A,I,K,I',K') += constit(B,I,J,I',J') * constit(C,J,K,J',K') * rewrites(A,B,C).
3  constit(A,I,K,I',K') += constit(B,J,K,I',J') * constit(C,I,J,J',K') * rewrites_inv(A,B,C).
```

```
4  goal += constit("A",0,M,0,N) * lenM,N).
5  input word(_,_,_); word'(_,_,_); transduce(_,_,_); rewrites(_,_,_); rewrites_inv(_,_,_); len_,_).
6  output goal.
```

Degree: 9, Optimal: 8.

A.4 Edit Distance (edit)

The following program implements a weighted generalized monotonic alignment between two sequences `word` and `word'`. It is essentially the well-known Levenstein distance (Levenshtein, 1966).

```
1  align(0,0) min= 1.
2  align(J,J') min= align(I,I') * word(W,I,J) * word'(W',I',J') * score(W,W').
3  align(I,J') min= align(I,I')               * word'(W',I',J') * score(ε,W').
4  align(J,I') min= align(I,I') * word(W,I,J)                   * score(W,ε).
5  goal min= align(N,N') * len(N) * len'(N').
6  input word(_,_,_); word'(_,_,_); score(_,_); len(_); len'(_).
7  output goal.
```

Degree 6, Optimal: 4.

A.5 Bar-Hillel Construction (bar-hillel)

The following program implements a parser for (weighted) intersection of a context-free parser and a bigram model on the part-of-speed sequences. It is essentially Bar-Hillel et al. (1961)'s construction of a context-free language that accepts the intersection of a regular language and a context-free language.

```
1  goal += β(0,_,root,_,N) * len(N).
2  β(I,A,X,D,K) += β(I,A,Y,B,J) * β(J,C,Z,D,K) * γ(X,Y,Z) * bigram(B,C).
3  β(I,X,X,X,K) += tag(X,W) * word(W,I,K).
4  input len(_); word(_,_,_); bigram(_,_); γ(_,_,_); tag(_,_).
5  output goal.
```

Degree 10, Optimal: 8.

A.6 Expectations under a Linear-Chain Conditional Random Field (chain-expect)

This example implements the inside-outside speedup (Li and Eisner, 2009) for computing the expectation of an additively decomposable function $f : \mathsf{S} \times \mathsf{S}' \to \mathbb{R}^d$ over randomly drawn sequences from a weighted graph (e.g., a conditional random field (Lafferty et al., 2001)). The graph is specified as a collection of weights `w`, as well as `start` and `end` nodes. The relations α and β implement the forward-backward algorithm (discussed in Rabiner (1989)), and `z` is the normalization constant of the distribution. The expectation of the ith dimension of `f` is `fbar(I)/z`.

```
1  % forward algorithm
2  α(S) += start(S).
3  α(S') += α(S) * w(S,S').
4  % backward algorithm
5  β(S) += end(S).
6  β(S) += w(S,S') * β(S').
7  % normalization constant
8  z += α(S) * end(S).
9  % unnormalized expected value via inside-outside speedup
10 fbar(R) += α(S) * w(S,S') * β(S') * r(S,S',R).
11 input w(_,_); r(_,_,_); start(_); end(_).
12 output fbar(_). z.
```

Degree 3, Optimal: 3.

A.7 Hidden Markov Models (hmm)

Hidden Markov models (HMMs) are the generative and locally normalized analogue of CRFs, which are discussed in App. A.6. Rabiner (1989) provides a classic tutorial.

```
1  v(0,start) += 1.
2  v(T',Y') += v(T,Y) * emission(Y,X) * transition(Y,Y') * obs(T,X,T').
3  goal += v(N,stop) * len(N).
4  input obs(_,_,_); len(_); emission(_,_); transition(_,_).
5  output goal.
```

Degree 5, Optimal: 4.

A.8 Semi-Markov Model (semi-markov)

The a semi-Markov model (Sarawagi and Cohen, 2004) generalizes a Markov model to score spans rather than individual words. In terms of runtime, one can perform inference in a Markov model in $\mathcal{O}(n)$ time (omitting dependence on the number of tags). In contrast, inference in a semi-Markov model takes $\mathcal{O}(n^2)$.

```
β(start, 0) += 1.
β(Y, J) += β(X, I) * transition(X, Y) * chunk(Y, I, J).
goal += β(_, N) * len(N).
input transition(_,_); chunk(_, _, _); len(_).
output goal.
```

Degree 4, Optimal: 3.

A.9 Most Probable Path (path)

Example 2 of the main text briefly discussed the most-probable path algorithm (Viterbi, 1967). We give a slightly more general version here that finds the most probable path from a set of `start` states to a set of `end` states.

```
v(S) max= start(S).
v(S') max= v(S) * w(S, S').
goal max= v(S) * stop(S).
input w(_,_); start(_); stop(_).
output goal.
```

Degree 2, Optimal: 2.

B Pseudocode for Program Transformations

In this section, we will make more extensive use of manipulations of terms. Terms can be equated with—or, matched against—other terms via **structural equality constraints** (Herbrand, 1930; Robinson, 1965; Martelli and Montanari, 1982; Knight, 1989), which are denoted by the equality operator, e.g., `f(X) = f(3)`. Systems of structural equality constraints have a unique minimal solution (up to variable renaming) when a solution exists, known as the most general unifier. For example, `f(Y,Z) = f(g(X),4)` has the solution `{Y ↦ g(X), Z ↦ 4}`, whereas `f(X,g(X)) = f(3,g(4))` has no solutions (is unsatisfiable). We assume access to a subroutine unify that can find a **substitution** mapping $\boldsymbol{\theta}$ to that makes the terms equal, or returns $\boldsymbol{\theta} = \emptyset$ if no substitution exists. For example, $\mathsf{unify}(\mathtt{f}(\mathtt{Y},\mathtt{Z}), \mathtt{f}(\mathtt{g}(\mathtt{X}),4)) \mapsto \boldsymbol{\theta} = \{\mathtt{Y} \mapsto \mathtt{g}(\mathtt{X}), \mathtt{Z} \mapsto 4\}$. We can apply the substitution with $\mathsf{subst}(\mathtt{f}(\mathtt{Y},\mathtt{Z}), \boldsymbol{\theta}) = \mathtt{f}(\mathtt{g}(\mathtt{X}),4)$. We will make use of the following utility method: $\mathsf{fresh}(x) \mapsto x'$ which returns a term x', which denotes the same set as the term x, but has distinct variable names, $\mathsf{vars}(x) \cap \mathsf{vars}(x') = \emptyset$. This operation is useful to prevent variable naming conflicts. For example, $\mathsf{fresh}(\mathtt{f}(\mathtt{g}(\mathtt{X}),\mathtt{X})) = \mathtt{f}(\mathtt{g}(\mathtt{X}_*),\mathtt{X}_*)$ where $\mathtt{X}_*$ is a novel variable name.

1. **def** $\mathsf{fold}(\mathcal{P}, i, \boldsymbol{\alpha})$:
2. ▷ *input: rule index i, subgoal indices to fold into a new subgoal* $\boldsymbol{\alpha}$.
3. $(\mathtt{h} \oplus\!= \mathtt{b}_1 \otimes \ldots \otimes \mathtt{b}_K) \leftarrow \mathcal{P}_i$
4. $\boldsymbol{\beta} \leftarrow \{1, \ldots, K\} \setminus \boldsymbol{\alpha}$ ▷ *Remaining factors*
5. $\{\mathtt{X}_1, \ldots, \mathtt{X}_K\} \leftarrow \mathsf{vars}(\mathtt{b}_{\boldsymbol{\alpha}}) \setminus (\mathsf{vars}(\mathtt{b}_{\boldsymbol{\beta}}) \cup \mathsf{vars}(\mathtt{h}))$
6. ▷ *Generate a new relation with a unique name, provided by the* gensym() *utility method. Note that the ordering of the arguments is arbitrary, but it is important for it to be used consistently.*
7. $\mathtt{gen}_\star \leftarrow \mathsf{gensym}()$
8. $h' \leftarrow \mathtt{gen}_\star(\mathtt{X}_1, \ldots \mathtt{X}_K)$
9. $\mathcal{P}' \leftarrow \mathcal{P}$ ▷ *Copy rules*
10. $\mathcal{P}'_i \leftarrow \left(\mathtt{h} \oplus\!= h' \otimes \prod_{j \in \boldsymbol{\beta}} \mathtt{b}_j\right)$
11. $\mathcal{P}'.\text{append}\left(h' \oplus\!= \prod_{j \in \boldsymbol{\alpha}} \mathtt{b}_j\right)$ ▷ *Add new rule that defines* h'
12. **return** $\mathcal{P}'$

1. **def** $\mathsf{unfold}(\mathcal{P}, i, k)$:
2. $(\mathtt{h} \oplus\!= \mathtt{b}_1 \otimes \ldots \otimes \mathtt{b}_K) \leftarrow \mathcal{P}_i$
3. $\mathcal{P}' \leftarrow \text{remove}(\mathcal{P}, i)$
4. **for** $j = 1 \ldots |\mathcal{P}|$:
5. ▷ *In the case of recursion, we rename variables in* s' *to avoid variable-name collisions.*
6. $s \leftarrow \mathsf{fresh}(\mathcal{P}_j)$ **if** $i = j$ **else** $\mathcal{P}_j$
7. ▷ *Solve for a substitution to make the head match* h
8. $\boldsymbol{\theta} \leftarrow \mathsf{unify}(\mathsf{head}(s), \mathtt{b}_k)$
9. **if** $\boldsymbol{\theta} = \emptyset$: **continue**
10. ▷ *Copy rule body*
11. $r' \leftarrow (\mathtt{h} \oplus\!= \mathtt{b}_1 \otimes \ldots \otimes \mathtt{b}_{k-1} \otimes \mathsf{body}(s) \otimes \mathtt{b}_{k+1} \otimes \ldots \otimes \mathtt{b}_K)$
12. ▷ *Apply substitution; copy rule*
13. $\mathcal{P}' \leftarrow \mathcal{P}' \cup \{\mathsf{fresh}(\mathsf{subst}(r', \boldsymbol{\theta}))\}$
14. **return** $\mathcal{P}'$

1. **def** eliminate$(\mathcal{P}, s)$:
2. ▷ *Transform assumes that all rules are range-restricted.*
3. $\mathcal{P} \leftarrow$ linearize$(\mathcal{P},$ head$(s))$
4. $\mathcal{P}' \leftarrow$ `[]`
5. **for** $r' \in \mathcal{P}$:
6. $r \leftarrow$ fresh(r') **if** r' **is** s **else** r'
7. count = 0
8. $(\mathtt{h} \oplus\!= \mathtt{b}_1 \otimes \ldots \otimes \mathtt{b}_k) \leftarrow r$
9. **for** $i = 1 \ldots k$:
10. $\boldsymbol{\theta} \leftarrow$ unify(head$(s), \mathtt{b}_i)$
11. **if** $\boldsymbol{\theta} \neq \emptyset$:
12. count += 1
13. $r' \leftarrow (\mathtt{h} \oplus\!= \mathtt{b}_1 \otimes \ldots \otimes \mathtt{b}_{i-1} \otimes$ body$(s) \otimes \mathtt{b}_{i+1} \otimes \ldots \otimes \mathtt{b}_k)$
14. $\mathcal{P}'$.append(fresh(subst$(r', \boldsymbol{\theta})))$
15. **assert** count > 1 ▷ *ensured by linearize on line 3*
16. **if** r' **is** s : **continue**
17. $\mathcal{P}'$.append(r)
18. **return** $\mathcal{P}'$

The linearize utility method transforms a program with repeated subgoals. For example, it transforms

```
1 goal += f(X) * g(X,Y) * f(Y).
```

into the following equivalent program that that does not repeat the `f` subgoal—or, more precise, it does not have any pair of subgoals that unify.

```
2 goal += f(X) * g(X,Y) * gen(Y).
3 gen(Y) += f(Y).
```

This transformation is useful as pre-processing in the eliminate function, which assumes the there are no repeated subgoals. If we do not want the entire program to be linearized, but just sufficiently linearize to eliminate a specific rule, we can specify what term we want to be linear with respect to. This is used in line 3 of the eliminate pseudocode.

1. **def** linearize$(\mathcal{P}, \mathtt{z})$:
2. $\mathcal{P}' \leftarrow \mathcal{P}$ ▷ *copy rules*
3. **for** $i = 1 \ldots |\mathcal{P}|$:
4. $(\mathtt{h} \oplus\!= \mathtt{b}_1 \otimes \ldots \otimes \mathtt{b}_K) \leftarrow \mathcal{P}_i)$
5. **for** $j = 1 \ldots K$:
6. **if** unify$(\mathtt{b}_j, \mathtt{z}) = \emptyset$: **continue**
7. **for** $k = j{+}1 \ldots K$:
8. **if** unify$(\mathtt{b}_j, \mathtt{b}_k) = \emptyset$: **continue**
9. ▷ *fold subgoal k out of rule i*
10. $\mathtt{b}_k(\mathtt{X}_1, \ldots, \mathtt{X}_L) \leftarrow \mathtt{b}_k$
11. $\mathtt{gen}_* \leftarrow$ gensym()
12. ▷ *replace $\mathtt{b}_k$ with the $\mathtt{gen}_*$ subgoal so that it won't appear twice.*
13. $\mathcal{P}'_i \leftarrow$ fresh$(\mathtt{h} \oplus\!= \mathtt{b}_1 \otimes \ldots \otimes \mathtt{gen}_*(\mathtt{X}_1, \ldots, \mathtt{X}_L) \otimes \ldots \otimes \mathtt{b}_K)$
14. $\mathcal{P}'$.append(fresh$(\mathtt{gen}_*(\mathtt{X}_1, \ldots, \mathtt{X}_L) \oplus\!= \mathtt{b}_k(\mathtt{X}_1, \ldots, \mathtt{X}_L)))$
15. **return** $\mathcal{P}'$