Approximate Bit Vectors for Fast Unification

Matthew Skala[1] and Gerald Penn[2]

[1] University of Manitoba, Winnipeg, Manitoba, Canada
mskala@cs.umanitoba.ca
[2] University of Toronto, Toronto, Ontario, Canada
gpenn@cs.toronto.edu

Abstract. Bit vectors provide a way to compute the existence of least upper bounds in partial orders, which is a fundamental operation needed by any unification-based parser. However, bit vectors have seen relatively little adoption because of their length and associated speed disadvantages. We present a novel bit vector technique based on allowing one-sided errors; the resulting approximate bit vectors can be much shorter than the minimum lengths required by existing techniques that would provide exact answers. We give experimental results showing that our approximate vectors give accurate enough answers to be useful in practice.

1 Introduction

In unification-based grammar, the operation *par excellence* is unification, even in parsing, where the sheer number of unifications outweighs any of the other basic operations that a chart parser must perform, including the maintenance of the chart itself.

In Head-driven Phrase Structure Grammar [10], unification is conducted over typed feature structures, and these unifications are driven primarily by the consistent combination of the types, which are drawn from a large partially ordered set. These types not only have the ability to cause unification to fail (if, that is, they do not have any common subtypes), but they also uniquely determine the work that must take place for the unification of the entire feature structure to suceed. This typically involves recursively combining the feature values that the feature structures share, since a type determines which features its feature structure has values for, as well as enforcing certain principles of grammar, since a principle of grammar is often stated as an implicational constraint in feature logic with an antecedent consisting of a type. To take advantage of the speed that this kind of strong typing can afford, we must build a $T \times T$ table to index these operations, and then look up the argument types in it during unification. Table lookup is fast in principle, but even though these tables are fairly sparse, they are large enough, and the operations involved are numerous enough, to lose an important property that good compiled code should have, called *locality of reference*. Disorganized references to this unification table cause memory pages to be swapped out of RAM, which can consume a great deal of time.

Kiefer et al. [5] and Penn [9] were the first to suggest that unification could be optimized by empirically gathering statistics on which feature paths are likely to produce type unification failures. The present paper is more related to the approach of Kiefer et

al., who used a vector of types gathered from the most common of these paths to filter out unifications that are destined to fail before making reference to the unification table. If the types in the corresponding dimensions of the vectors combine, then we must still refer to the table. But if a failure is caught by the vectors (and very modular grammars generally propose a large percentage of type-incompatible unifications), then we can fail without referring to the table, and better preserve locality of reference.

This is all made possible by a compact representation of types that does not itself require a table to answer the yes/no question of whether a finite set of types has a common subtype (as opposed to our table, which also enumerates all of the necessary operations incident to discovering that they do). This representation is based on a bit-vector encoding scheme for partial orders first proposed by Aït-Kaci et al. [1], in which two types have a consistent common subtype iff the bitwise AND of their vectors is not zero. The problem with this encoding is its size—there must be at least as many types in this encoding as the number of meet-irreducible subtypes in the partial order of types, which is always at least as large as the number of maximally specific subtypes. The number of meet-irreducible types can run into the thousands, occupying 100 or more machine words, and unlike our unification tables, the vector tables are not sparse, since the positions of the zeros must be preserved. Managing these is so onerous that Kiefer et al. resort to bit vectors only when they first encounter a pair of types that is not present in a page-limited cache of recent or frequent unifications [5].

Even in their initial paper, Aït-Kaci et al. [1] alluded to a modification of their encoding scheme that works for certain shapes of partial orders, but for which the unification operation is more complex than bitwise-AND and zero-checking. Fall [4] coined a sparse logical term data structure that also has a more complex unification operation. Skala et al. [11] used bit vectors with bitwise-AND, but generalized zero-checking to checking for $\lambda \geq 0$ or fewer 1 bits, and achieved a modest reduction in bit-vector size.

In this paper, we use bitwise-AND and λ-checking, but with the additional generalization that our unification operation sometimes makes mistakes. It makes them in only one direction, however: a "no" always means that two types are not unifiable, but a "yes" may not be correct. This kind of encoding is related to the one-sided error properties of Bloom filters [2] that have been used for efficiently encoding language models in statistical machine translation [12]. Since we need to ask the unification table what operations to perform in case of a "yes" anyway, this particular direction of error never affects the soundness of unification, and at worst only increases the time spent.

Once we accept that our filter may make mistakes, it is our choice how large to make the bit vectors, with the understanding that the number of mistakes increases as size decreases. Size for a given error rate, in turn, generally decreases as λ increases. Time cost increases as λ increases also, unless the CPU instruction set supports a primitive for checking for λ or fewer bits, which many now do.

To a great extent, it is also our choice as to *where* we make mistakes. We introduce a stochastic local search technique for selecting optimal bit vectors, given the parameters of the encoding scheme, which is capable of incorporating preference weights. These weights can be measured by counting the number of calls to the unifier for every pair of types over some sample of (possibly noisy) text, with the result that, if the sample is representative, the encoding will perform best where the unifier will use it most often.

1.1 Notation and Definitions

Let $\langle X, \sqsubseteq \rangle$ denote a partial order consisting of a set X and a reflexive, antisymmetric, and transitive binary relation $\sqsubseteq$. If u and v are elements of X such that $u \sqsubseteq v$ then we say that u *subsumes* v. Let $u \sqcup v$ denote the unique least upper bound or *join* of $u, v \in X$, if one exists, and $u \sqcap v$ the greatest lower bound or *meet*. If u and v are distinct elements of X such that $u \sqsubseteq v$ and there is no $w \in X$ with $w \neq u, w \neq v$, and $u \sqsubseteq w \sqsubseteq v$, i.e., v follows u in X with no other elements in between, then we say that v is a *successor* of u and u is a *predecessor* of v. A *maximal* element is one that has no successor.

Figure 1 shows a small example partial order, drawn (as is customary) with the maximal elements at the top. In this example, a subsumes itself, c, and d, whereas b subsumes only itself and d. The element $\perp$ (bottom) subsumes everything, but only a and b are its successors. The maximal elements are c and d. The join of a and b is d.

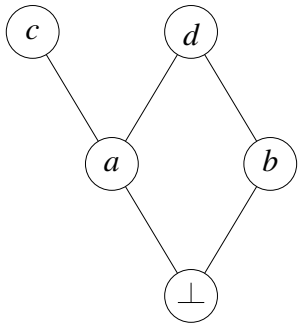

Fig. 1. A small partial order

Given two partial orders $\langle X, \sqsubseteq \rangle$ and $\langle Y, \preceq \rangle$, a pair of functions $f : X \to Y$ and $g : (Y \times Y) \to \{0, 1\}$ is called an *embedding* of X into Y. An embedding may have some of the following properties for all $u, v \in X$:

$$u \sqsubseteq v \Rightarrow f(u) \preceq f(v) \tag{1}$$

$$\mathit{defined}(u \sqcup v) \Rightarrow g(f(u), f(v)) = 1 \tag{2}$$

$$\neg \mathit{defined}(u \sqcup v) \Rightarrow g(f(u), f(v)) = 0 \tag{3}$$

$$u \sqcup v = w \Leftrightarrow f(u) \curlyvee f(v) = f(w)\,. \tag{4}$$

If it has the properties (1), (2), (3), and (4), an embedding is said to preserve order, success, failure, and joins, respectively.

In the present paper, we are interested in the case where desired properties hold for many, but not necessarily all, pairs of $u, v \in X$. In particular, we examine the case of embeddings that preserve order, success, and usually failure, but where (3) does not hold unconditionally. We would prefer for it to hold as much as possible; so we define,

for some *weight function* $w : (X \times X) \to \mathbb{R}$, the *weight* of an embedding $\langle f, g \rangle$ as

$$\sum_{\substack{u,v \\ \neg defined(u \sqcup v) \\ g(f(u),f(v))=1}} w(u,v)\,. \tag{5}$$

The weight is the sum of $w(u,v)$ for pairs on which the embedding violates (3). If $w(u,v)$ represents a measure of how undesirable it would be for our embedding to give a false positive for the joinability of u and v, then minimizing the weight of the embedding will give us the best possible embedding, the one closest to satisfying (3), under the reasonable assumption that our desires about different pairs are independent of each other. Note that the value of w is only relevant for pairs of u and v that do not have a join; we insist on (2), so there are no false negatives.

For consistency with other work we talk about joins, which are by definition unique, but our technique actually tests for the existence of least upper bounds, which are not necessarily unique. The partial orders of most interest to us are *meet semilattices*, in which it makes no difference. Meet semilattices are defined by the property that in a meet semilattice, every pair of distinct elements has a unique greatest lower bound (the meet). It is a well-known theorem that in a meet semilattice, every pair of elements also either has one unique least upper bound (the join) or no least upper bound at all. Nonetheless, our technique would also be applicable to more general partial orders, in which case the property (2) would be modified to guarantee $g(f(u), f(v)) = 1$ whenever there is at least one least upper bound, not only a unique join, and for (3) we would only consider pairs having no least upper bounds.

2 Bit-Vector Encoding

Two types in a type hierarchy have a least upper bound if and only if there is at least one maximal type that they both subsume. For instance, in Fig. 1, d is a maximal type subsumed by a and b; they have a least upper bound. But b and c do not both subsume any maximal type; they do not have a least upper bound.

Since it suffices to look for such maximal types, we can compute existence of least upper bounds by associating the types in the hierarchy with their sets of subsumed maximal types, and computing the intersections of those sets. Unification fails if the intersection is empty. Representing the sets as vectors of bits, and using bitwise AND to compute the intersections, yields the classical bit vector technique of Aït-Kaci et al. [1]. This embedding preserves order, success, and failure, and can be extended (by adding bits for unary-branching types) to preserve joins; but it has the significant disadvantage of using very long bit vectors—linear in the number of types in the worst case, and frequently so in practice.

Skala et al. describe a class of embeddings in which each maximal type corresponds to a vector with more than one nonzero bit; specifically $\lambda + 1$ bits for some parameter λ [11]. The vector for a non-maximal type is the union (bitwise OR) of the vectors for all types it subsumes. This technique reduces to the classical one when $\lambda = 0$. Instead of counting only the all-zero vector as unification failure, they count any vector with λ

or fewer nonzero bits as failure. Success is preserved because any two types that both subsume some maximal type must contain all its nonzero bits, which is more than λ. Skala et al. show that they can design such encodings using constraint programming over sets, to preserve failure as well success, and optionally joins.

The work of Skala et al. takes its inspiration from the well-known technique called Bloom filtering [2, 11], which stores a sparse array of bits in reduced space by mapping each index in the large array through a set of pseudorandom hash functions into several indices in the small array, as shown in Fig. 2. To store a bit, they find all its hashed locations and store bits in all of them. To check whether a bit was stored, they check all its hashed locations and say "yes" only if they *all* contain bits. There is some chance that looking up a bit could give a false positive—a "yes" answer even though that bit was not stored in the filter. As the number of bits stored increases, the risk of these false positives increases. But the Bloom filter will never give a false negative: if queried on a bit that was in fact stored, it will always return a "yes" answer.

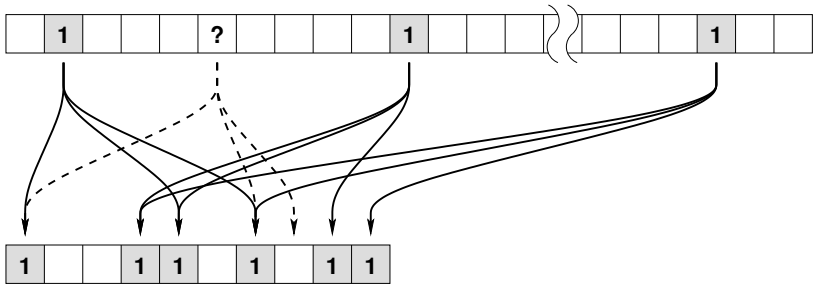

Fig. 2. A Bloom filter

Bloom filters can also be used for more elaborate operations on sets of bits; in particular, the operation of interest here, which consists of testing for non-empty intersection. Given two Bloom filters in which each bit stored corresponds to $\lambda + 1$ distinct indices, if we take the bitwise AND of the two shortened vectors and find it contains λ or fewer bits, we know that there can be no stored bits in common between the two filters. The single-bit lookup is simply a special case of that operation where one of the sets is a singleton. If we use the Bloom filtering technique to abbreviate the long vectors of Aït-Kaci et al. [1], then testing for the existence of a least upper bound is the more general case.

Using several hashed locations per bit reduces false positives: even if there is a collision at one index, there would have to be collisions at all of them in order for a false positive to occur. Bloom discusses this issue in detail and derives formulas relating the parameters of the data structure to its error probability [2]. The multiple-location feature ($\lambda > 0$) allows significant reduction in vector length for our application. However, the better-known feature of Bloom filters, not exploited in the previous work of Skala

et al. [11], is the one-sided error property: by allowing the least upper bound test to sometimes violate failure preservation, we can reduce the vector lengths much further.

It might seem that losing failure preservation renders the least upper bound test useless, since any logic programming system should eventually produce correct answers. However, even an approximate unification test has value if, like ours, it can give its answers *quickly*. As Kiefer et al. describe [5], a fast test that quickly rejects many failed unifications can be run first; if it fails (bearing in mind that success is preserved, so failure results are always correct), then we can skip doing the more expensive test that would give guaranteed correct results. In a parsing system we could even defer complete rejection of failed unifications until after several unifications have been performed. This usage of the cheap tests first combines well with existing techniques that look first, in a complicated data structure, at the parts for which unification is most likely to fail. It is also in line with traditional uses of the Bloom technique in things like programming language compilers: if the Bloom filter can reject most failed symbol matches quickly, it reduces the cost of a more expensive hash lookup that can verify the relatively few matches passing the initial filter.

The most obvious way to apply Bloom filtering to our application would be to simply assign a randomly-selected choice of $\lambda + 1$ bits to each maximal type, and give each non-maximal type the union (bitwise OR) of all the vectors for types it subsumes. However, we can argue intuitively (and test by experiment) that that approach will not produce optimal results. Any non-maximal type that subsumes a significant number of maximal types will be expected to subsume at least one type containing a nonzero value at each of the indices in the vector; then that non-maximal type will end up having all its bits set. In a type hierarchy like that of the English Resource Grammar (ERG) [3], there are clusters of hundreds or thousands of maximal types sharing a single predecessor. That predecessor, and everything subsuming it, will end up with the all-ones vector; then the unification test will rate it as unifying with everything, and the filter will be useless on that type. It seems that to expect good results, we should somehow optimize the assignment of bits to encourage non-maximal types to have less dense vectors, even if doing so causes more sharing of bits, and thus more collisions, among maximal types.

3 Stochastic Local Search

The problem of finding an assignment of vectors to types while satisfying the desired embedding properties is essentially a constraint programming problem over the domain of finite sets. Skala et al. [11] describe using a constraint propagation technique to optimize the number of bits while preserving order, success, failure, and optionally joins. Our current problem is similar, but we fix the number of bits, allow some constraints (those corresponding to preservation of failure) to be possibly violated, and optimize on the total weight of violated constraints.

The constraint propagation solution technique is not well-suited to this version of the problem, because it would require introducing a solver variable to represent each of the millions of possibly-violated constraints. The resulting problem would be far beyond the limits of the constraint propagation solver used by Skala et al. [11] for the failure-preserving bit vector problem; the instances they solved were only within reach

because they used a bespoke "intersection not too large" constraint with much lower time and memory overhead (in particular, without introducing a new solver variable per type pair) than the obvious implementation based on solver primitive constraints. They also proved mathematical properties of the exact problem that allowed them to factor the instances and eliminate many constraints entirely. Such properties do not seem to apply easily to the non-exact problem, where each constraint must be retained in order to count the violated ones.

However, the constraint propagation solver's main advantage of providing an exact solution is not so important here, where the best possible solution is not expected to be perfect anyway. We instead use a stochastic local search technique, which has the advantage of being a simpler algorithm amenable to efficient implementation in C. The key, as in the earlier work of Skala et al. [11], is to store and process as little data per constraint as possible, because the number of constraints is quadratic in the number of types, growing faster than anything else in the problem.

For some vector length and parameter λ (for which we will try different values and see which one works best), we start with a random assignment of vectors to the maximal types, giving each maximal type a vector with ones at $\lambda + 1$ distinct indices. Non-maximal types always have vectors consisting of the bitwise OR of the vectors for all the types they subsume. It is easy to show that all assignments of this kind must be embeddings that preserve order and success; the question is how much failure an embedding preserves, and we measure that using the weight defined earlier.

We evaluate the weight of the initial embedding, make a small random change (notionally changing the vector for one type and making any necessary changes that result from that) and evaluate the weight of the result. If it is an improvement (equal or lesser weight) then we retain that change; otherwise we undo the change. Then we repeat this process through some number of iterations, hoping to find an optimal or near-optimal embedding. Note that we retain random moves that leave the total weight unchanged (no strict improvement) in order to encourage the search to explore more of the search space, reducing any tendency for it to get stuck in local minima. Such moves do seem to be common in practice.

This kind of stochastic local search is very simple, but in our experiments it is shown to work well on this problem when coupled with an appropriate heuristic for suggesting the random changes to test. More elaborate local search algorithms, such as simulated annealing, seem to provide little if any improvement in the final search result. Simulated annealing trades off avoidance of local minima for slower convergence; our observation is that although local minima exist, our heuristic renders them shallow: so many different local moves are possible, and so many local moves are good, that the search will almost always escape the local minimum eventually. The fitness smoothly decreases over the course of the optimization without seeming to become "stuck." As a result, simply trying the entire search a few times with different random starting points and taking the best one at the end generally produces better results than a single slower-converging run of simulated annealing, as well as being more easily amenable to parallelization.

The heuristic for choosing random moves is critical to the success of the technique. For each random move, we choose a type uniformly from the collection of all types. Depending on the type chosen and its current vector, we assign new vectors as follows.

- If we chose a maximal type, we assign it a completely new vector containing ones at $\lambda + 1$ distinct indices, chosen uniformly at random from the set of all such vectors.
- If we chose a non-maximal type which is currently assigned a vector containing greater than $\lambda + 1$ ones, then we will "shrink" that vector: remove a bit and make necessary changes to keep the other types consistent. We choose one of the selected type's one bits uniformly at random and set that bit to zero. Then for any maximal types subsumed by our selected type, if they contain a one at the zeroed index, we give them new values, choosing $\lambda + 1$ distinct indices uniformly at random from those that contain one bits in the selected type's new vector.
- If we chose a non-maximal type whose vector currently contains exactly $\lambda + 1$ ones (which implies all the types it subsumes must have the same vector), then we give it a new value of $\lambda + 1$ bits chosen from all indices, just as if it were a maximal type, and we set the vectors of all types it subsumes to that new value.

After any of those cases we then make all necessary changes to non-maximal type vectors to ensure that every non-maximal type's vector is equal to the bitwise OR of the vectors of all types it subsumes.

This heuristic may seem unintuitive, but it is motivated by the issue with random assignments described in the previous section. A random assignment will tend to contain many all-ones vectors on non-maximal types. If we change the value of just one maximal type, which is the obvious kind of random move to make, it is quite possible that any bits we remove will still be covered by other maximal types: so that the random assignment plus this kind of move will still contain all-ones vectors in the same places. It is difficult or impossible to remove an all-ones vector from the assignment by this kind of move. Thus we would be faced with local minima from which the search cannot easily escape. In order to explore assignments with less dense vectors on the non-maximal types, we must allow moves with a significant chance of removing all-ones (or mostly-ones) vectors where they exist. The stated heuristic does that in a straightforward way.

One other issue is important to the practical implementation. Fully evaluating the weight of an assignment requires testing $O(n^2)$ pairs of types and adding up the weights for the pairs that are constraint violations. Rather than doing that on every iteration, we use an incremental approach. In most cases only one or a few maximal types, plus the non-maximal types that subsume them, are given new vectors by a random move. We keep track of the vectors affected by a move and only recalculate the weights for constraints involving those vectors. Incremental calculation results in a significant improvement in the time per iteration.

4 Evaluation

The goal of our evaluation was to test, first, the optimization process, with such questions as how well shortened bit vectors could approximate the joinability function; and second, the effect on speed of using these vectors in an actual parsing situation. We tested the technique on the type signatures of three unification-based grammars: the ALE HPSG grammar, MERGE, and the ERG.

The ALE HPSG grammar is a simple grammar included with the Attribute Logic Engine as a demonstration of the system [8]. It is a very literal encoding of Chapters 1–5

and 8 of Pollard and Sag's HPSG text [10]. This grammar's type signature contains 132 types, of which 87 are maximal; thus the classical bit vector technique would require 87-bit vectors to preserve order, success, and failure. We did not explicitly repeat the constraint propagation experiments of Skala et al. [11] for this grammar, but it happens that the grammar is so small and easy to represent that our optimizations produced perfect (weight zero) results in some cases, allowing us to put an upper bound of 24 on the number of bits needed to preserve order, success, failure, and joins with their techniques, using a λ value of 2.

ERG, the English Resource Grammar [3], is the largest grammar we tested, with 45 rules, 155 features, 1314 lexical entries, no lexical rules, and 3412 types, plus a most general type, $\perp$, a built-in type for strings, and 893 "completion" types to guarantee the existence of unique most general unifiers. We used the same ALE port of this grammar tested earlier by Skala et al. [11]. As they describe, the classical bit-vector embedding would require 2788 bits, and the shortest vectors they found were 985 bits long, with further improvements possible either by modularization or potentially by improving the constraint propagation solver.

MERGE is a reimplementation [6] of the ERG that was designed to showcase the features of the TRALE system [7]. Its type signature contains 1231 types, of which 502 are maximal, thus requiring 502 bits to preserve order, success, and failure with the classical bit vector technique.

4.1 Optimization

We implemented the stochastic local search in C and ran it on a cluster of five computing servers each containing eight AMD Opteron dual-core CPUs at 1GHz clock frequency, for a total of 80 cores, with 64-bit Linux 2.6.18 operating system kernels. The optimizer itself was not written as a parallel program, but the multiple CPUs meant we could run multiple instances of it simultaneously. Each server had 32G of RAM, of which our experiments only used a small fraction (up to a few hundred megabytes per optimization instance in the largest case, which was the ERG).

In each case we did three optimization runs with different random number seeds for each combination of parameters, and kept the best of the three. For the ALE HPSG and MERGE, we did one million iterations of the local search for each run. For the ALE HPSG each such run took less than a minute on one core. For MERGE, it was approximately ten hours on one core per run. Optimizations for the ERG ran much slower, apparently due both to its sheer size (recall that the number of constraints increases with the square of the number of types) and differences in its design leading to a more difficult optimization problem. We ran fewer iterations, only one hundred thousand per run; ERG optimization run time varied significantly with the parameters chosen (much more so than for the other grammars) but was typically two or three hours on one core.

We tested three different kinds of weight functions. First the uniformly-weighted case, where each non-joinable pair of types is assigned weight 1. This corresponds to simply minimizing the number of type pairs for which the vectors will give a false positive, without regard for the relative importance of different pairs. To provide a more realistic weighting, we also parsed sentences from a corpus appropriate to each grammar using an instrumented version of ALE that counted the number of unification attempts

for each pair. This instrumented ALE ran very slowly, especially on MERGE, but we were able to parse 1724, 217, and 1043 sentences for the ALE HPSG, MERGE, and the ERG respectively, in each case using the test corpus associated with the grammar. We smoothed the counts by adding one to each, and used the resulting function to run weighted optimizations.

Finally, we exploited ALE's existing modularization. ALE splits each type signature into non-joining modules and will never attempt a unification of types in distinct modules (which would always fail). As a result, it is not necessary for our bit vectors to correctly handle unification between modules, and it might be possible to get better results within modules by sacrificing some accuracy between modules. To test that possibility, we created a third weight function called the modular weight function, equal to the smoothed-count function except with the weight set to zero for each pair of types not in the same module.

Figure 3 shows the progress of a typical set of three optimization runs, in this case for MERGE with uniform weights, a vector length $b = 32$, and $\lambda = 2$.

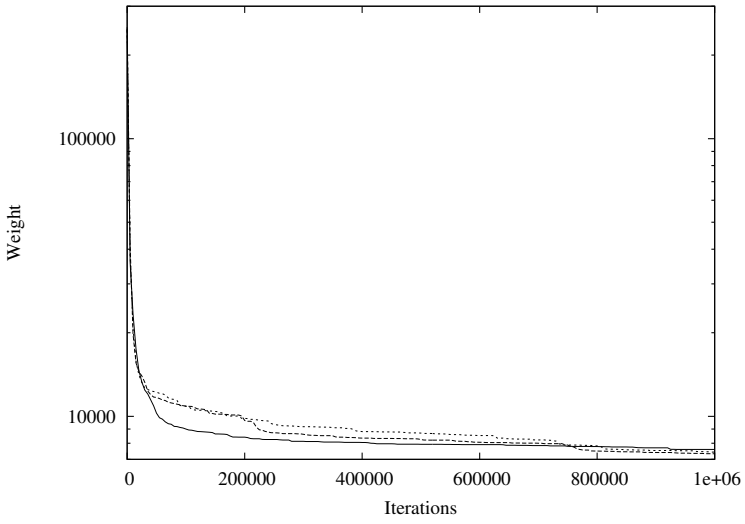

Fig. 3. Typical optimization progress (MERGE, 32 bits, $\lambda = 2$).

Tables 1, 2, and 3 show results from the optimization, including for each weight function the total weight (corresponding to a function that simply says "yes" in all cases), the weight of the initially chosen random vectors, and the weight after optimization. Both the random and optimized weights are, as described above, the best seen in the three runs. Not all combinations of parameters were tested for all grammars, and not all tested parameters are shown; because of resource limitations on the parsing experiments described in the next section, we prioritized the optimizations that seemed to

provide the best parsing results, and did additional optimization runs to include shorter vectors than originally planned.

Table 1. Optimization results for ALE HPSG.

		uniform		weighted		modular	
bits	λ	rand	opt	rand	opt	rand	opt
4	0	3663	1901	3352404	2986	929	6
4	1	3945	1197	1138828	2489	86113	7
4	2	5132	1869	3245387	18952	96497	376
8	0	2242	815	157700	1367	928	0
8	1	1879	160	2164885	378	3	0
8	2	2123	87	961674	199	8	0
8	3	2701	92	962094	243	6	0
8	4	3100	114	3025163	436	409	1
8	5	3850	203	3112821	1188	412	6
16	0	1217	308	55063	550	0	0
16	1	894	20	894	24	0	0
16	2	960	6	960	9	0	0
16	3	1077	2	1157	2	1	0
16	4	1196	2	1308	2	2	0
16	5	1482	3	1664	4	2	0
24	0	825	173	1676	293		
24	1	542	1	542	1		
24	2	629	0	629	0		
24	3	688	0	688	0		
24	4	804	0	1760	0		
24	5	909	0	931	0		
total weight		8304		8014419		2012974	

The most important observation from these results is that the optimized bit vectors are very good even at short bit lengths. For the ALE HPSG type signature with modularization, eight bits were sufficient to give zero weight, perfect results; that compares favorably with the 87 bits required by the classical technique of Aït-Kaci et al. [1]. On the much larger ERG, where an exact solution (as described by Skala et al. [11]) would require hundreds of bits, a single 32-bit machine word is a long enough vector to bring the weight in the modular case down to 56477 out of 8831793 with $\lambda = 1$. In other words, for the smoothed distribution of unifications actually performed during parsing, 99.36% of checks to the unification table are eliminated by the bit vector test on 32-bit vectors.

4.2 Parsing

For each grammar, we parsed a held-out section of the relevant corpus using a version of ALE 4.0 beta modified to support bit vectors for unification, and we measured the mean

Table 2. Optimization results for MERGE.

		uniform		weighted		modular	
bits	λ	rand	opt	rand	opt	rand	opt
4	0	478264	171326	632923	217782	272771	74350
4	1	502991	140701	720863	203209	316784	72436
4	2	550343	217287	2050686	327957	348983	150810
8	0	387563	90037	472952	111694	180261	37073
8	1	392032	47398	478683	62555	205231	29693
8	2	413042	42728	537685	55432	222813	30943
8	3	438898	53143	582860	63317	219105	38584
8	4	470275	60685	636929	86620	247299	43690
16	0	305157	45041	358722	55106	135786	17519
16	1	302719	19605	346807	24010	146375	11851
16	2	318062	19752	348421	21940	155394	12617
16	3	334066	20090	388130	24059	156956	16503
16	4	351224	21097	401921	26265	165616	20191
24	0	260450	28904				
24	1	255756	10342	287850	13521	126340	6590
24	2	274857	10996	302617	11845	132106	7176
24	3	286147	12570	323197	13180	136827	9373
24	4	306390	16199	337226	18480	145621	11280
32	0	235734	20897			110096	7893
32	1	232300	7584	263404	8435	107306	4209
32	2	246077	7331	274537	8882	120919	4436
32	3	262801	8064	290640	8917	128384	6056
32	4	276403	11036	303034	11000	131476	7843
total weight		695724		2378477		555131	

Table 3. Optimization results for the ERG.

		uniform		weighted		modular	
bits	λ	rand	opt	rand	opt	rand	opt
4	0	5242112	2265973	8721449	2588924	5673212	1040490
4	1	5383985	1494710	9301061	1915308	5646780	890570
4	2	6183247	2300090	11040021	3363294	6829783	1784741
4	3			14588111	14588111	8831793	8831793
8	0	3894775	1107982	6825571	1321100	4326082	532734
8	1			6761076	541630	4257009	301213
8	2			7148380	419644	4437374	267378
8	3			7861117	468618	4969842	274154
16	0	2929784	550199	5502479	663181	1397396	278070
16	1			4375163	209905	3410728	135560
16	2			5527272	203264	3318734	123529
16	3			5744620	238163	3688512	121298
32	0	2116009	272920	2590378	340666	962926	123846
32	1			2407795	138851	903752	56477
32	2			4201044	132072	2949600	84704
32	3			4506346	154472	3191455	59562
total weight		9105928		14588111		8831793	

time in milliseconds per sentence. Parse times were measured using the `walltime` clock in SICStus Prolog 3.12.10 on an Intel-based system with a 3.6 GHz Xeon processor and 3 GB of RAM running the Ubuntu 6.06.2 Linux operating system. The results are shown in Table 4. As well as the times for our bit vector technique implemented in Prolog, we list as a control the times obtained using SICStus Prolog's native and heavily optimized PJW hash function.

Adding instrumentation to a basic language primitive as we did resulted in very slow parsing overall. Combining that with the limited availability of machines licensed to run SICStus Prolog, ALE's requirement for a no-longer-current version of the interpreter, and memory and address space issues on the 32-bit parsing computers, it was very difficult to collect parse timing data at all and we could run only a few of the most interesting parameter combinations.

However, this data remains valuable as the most pessimistic possible test of our technique: the fact that bit vectors can produce comparable times to the native hash under such conditions at all, combined with the low weights described in the previous sections, suggests that a parser designed from the beginning to use bit vectors and running on a CPU with a bit counting feature could see significant improvement in vector size and overall speed by accepting a limited number of false positives; and that limited number need not be very large. We emphasize that the approximate bit vector technique is not specific to ALE or specific to Prolog, but a general technique applicable to any system that performs unification in a partial order of types. A customized parser designed specifically to use bit vectors would no doubt run faster, but would not provide

Table 4. Parse time results

grammar	weights	bits	λ	time (ms)
ALE HPSG	PJW hash			26.6
ALE HPSG	modular	4	0	27.3
ALE HPSG	modular	8	0	27.3
ALE HPSG	modular	8	1	28.6
ALE HPSG	modular	8	2	30.0
ALE HPSG	modular	32	1	33.7
MERGE	PJW hash			28696
MERGE	modular	4	0	29505
MERGE	modular	8	0	29176
MERGE	modular	8	1	29646
MERGE	modular	16	0	29112
MERGE	modular	16	1	30343
MERGE	modular	32	1	33235
ERG	PJW hash			1966
ERG	Aït-Kaci	2788	0	2507
ERG	modular	32	1	2194
ERG	modular	32	2	2377
ERG	modular	64	1	2310
ERG	uniform	32	1	2335
ERG	weighted	32	1	2213

a well-controlled evaluation of the approximation technique being tested in the present work.

When looking at the data it is clear that smaller values of λ resulted in faster parsing, despite the results in the previous section suggesting that larger λ may allow for better-optimized weights. That effect likely stems from the use of Intel Xeon processors for the parsing test, which (although they may implement bit count in their superscalar extensions at the assembly language level) do not have a bit count operation for the main registers easily accessible from Prolog. The increased time for larger λ reflects the cost of doing the bit count at the Prolog level by way of more expensive arithmetic operations. It also suggests that as long as we operate in this environment without a fast bit count, it is unnecessary to make heroic efforts on the vector optimization because the effect of λ overwhelms the differences in weight among roughly-optimized vector sets.

For the smaller grammars (ALE HPSG and MERGE), the most effective bit vector lengths in this experiment are very short: only four or eight bits. Again, the advantages of keeping the vectors short outweigh the increased error rates. For the ERG, we tested longer vectors including the classical vectors of Aït-Kaci et al. [1]. The advantage of our short vectors over those much longer ones is evident. We also tested, at the 32-bit vector length, the differences between vectors optimized with uniform weights, with weights derived from test parsing, and with modularity taken into account. As expected,

the modular vectors give better times than the weighted vectors which give better times than the uniform vectors.

Figure 4 presents a different view of the parsing times: it shows the time for each individual sentence in the ERG test, sorted in decreasing order of the PJW hash time. Figure 5 is a similar chart for MERGE. These comparisons show that the time differences associated with longer vectors or greater λ are generally consistent across the corpus.

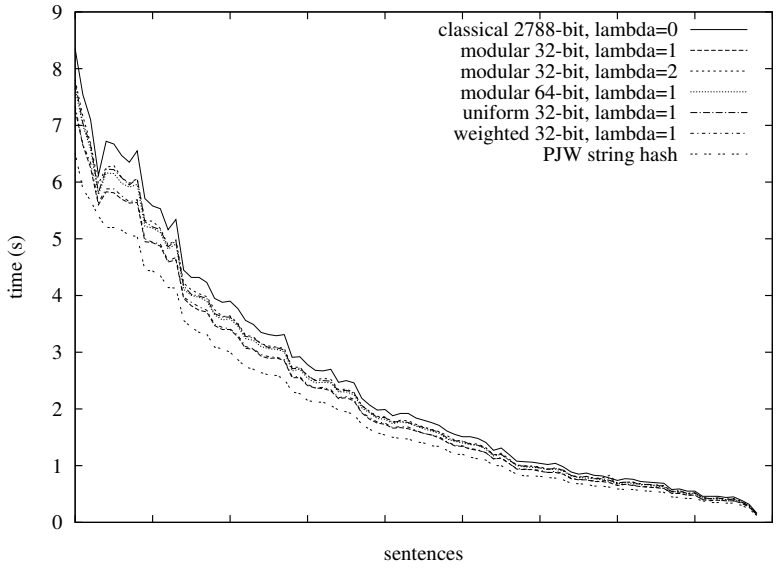

Fig. 4. Per-sentence parsing times for the ERG.

5 Conclusion

We have described an approximate bit vector technique for fast unification, based on hashing bits into multiple locations and permitting one-sided errors, as in Bloom filtering. We have also described a stochastic local search algorithm with an appropriate heuristic for optimizing the bit vector encodings; and we have presented experimental results evaluating both the local search and the speed of the unification for a variety of parameter values. Our technique allows the use of much shorter vectors than any previously-known bit vector techniques, making bit vectors a practical option in applications where they might not otherwise be considered.

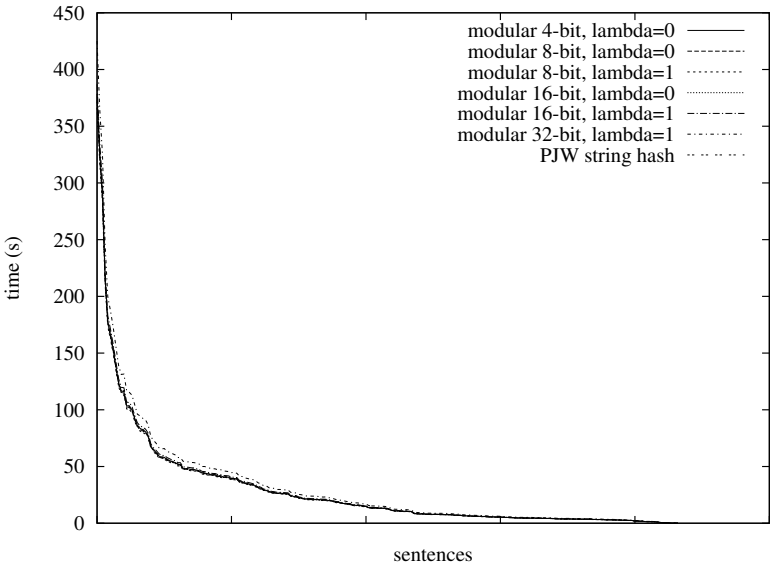

Fig. 5. Per-sentence parsing times for MERGE.

References

1. Aït-Kaci, H., Boyer, R.S., Lincoln, P., Nasr, R.: Efficient implementation of lattice operations. ACM Transactions on Programming Languages and Systems 11(1), 115–146 (Jan 1989)
2. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (Jul 1970)
3. Copestake, A., Flickinger, D.: An open-source grammar development environment and broad-coverage English grammar using HPSG. In: Proceedings of the Second Conference on Language Resources and Evaluation (LREC 2000) (2000)
4. Fall, A.: Reasoning with Taxonomies. Ph.D. thesis, Simon Fraser University (1996)
5. Kiefer, B., Krieger, H.U., Carroll, J., Malouf, R.: A bag of useful techniques for efficient and robust parsing. In: Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (ACL-ANNUAL'99). pp. 473–480. ACL (1999)
6. Meurers, D., De Kuthy, K., Metcalf, V.: Modularity of grammatical constraints in hpsg-based grammar implementations. In: Proceedings of the ESSLLI Workshop on Ideas and strategies for multilingual grammar Engineering (2003)
7. Meurers, D., Penn, G., Richter, F.: A web-based instructional platform for constraint-based grammar formalisms and parsing. In: Proceedings of the ACL Workshop on Effective Tools and Methodologies for Teaching NLP and CL (2002)
8. Penn, G.: A Utility for Feature-based Grammatical Theories. Master's thesis, Carnegie Mellon University (1993)
9. Penn, G.: Optimising don't-care nondeterminism with statistical information. Tech. Rep. 140, SFB 340, Tübingen (1999)
10. Pollard, C., Sag, I.: Head-driven Phrase Structure Grammar. Chicago (1994)

11. Skala, M., Krakovna, V., Kramár, J., Penn, G.: A generalized-zero-preserving method for compact encoding of concept lattices. In: 48th Annual Meeting of the Association for Computational Linguistics (ACL 2010). pp. 1512–1521. ACL (2010)
12. Talbot, D., Osborne, M.: Smoothed Bloom filter language models: Tera-scale LMs on the cheap. In: Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL). pp. 468–476 (2007)