Efficient Parsing of Highly Ambiguous Context-Free Grammars with Bit Vectors

Helmut Schmid
Institute for Computational Linguistics
University of Stuttgart
Azenbergstr. 12
D-70174 Stuttgart
Germany
schmid@ims.uni-stuttgart.de

Abstract

An efficient bit-vector-based CKY-style parser for context-free parsing is presented. The parser computes a compact parse forest representation of the complete set of possible analyses for large treebank grammars and long input sentences. The parser uses bit-vector operations to parallelise the basic parsing operations. The parser is particularly useful when all analyses are needed rather than just the most probable one.

1 Introduction

Large context-free grammars extracted from treebanks achieve high coverage and accuracy, but they are difficult to parse with because of their massive ambiguity. The application of standard chart-parsing techniques often fails due to excessive memory and runtime requirements.

Treebank grammars are mostly used as probabilistic grammars and users are usually only interested in the best analysis, the Viterbi parse. To speed up Viterbi parsing, sophisticated search strategies have been developed which find the most probable analysis without examining the whole set of possible analyses (Charniak et al., 1998; Klein and Manning, 2003a). These methods reduce the number of generated edges, but increase the amount of time needed for each edge. The parser described in this paper follows a contrary approach: instead of reducing the number of edges, it minimises the costs of building edges in terms of memory and runtime.

The new parser, called BitPar, is based on a bit-vector implementation (cf. (Graham et al., 1980)) of the well-known Cocke-Younger-Kasami (CKY) algorithm (Kasami, 1965; Younger, 1967). It builds a compact "parse forest" representation of all analyses in two steps. In the first step, a CKY-style recogniser fills the chart with constituents. In the second step, the parse forest is built top-down from the chart. Viterbi parses are computed in four steps. Again, the first step is a CKY recogniser which is followed by a top-down filtering of the chart, the bottom-up computation of the Viterbi probabilities, and the top-down extraction of the best parse.

The rest of the paper is organised as follows: Section 2 explains the transformation of the grammar to Chomsky normal form. The following sections describe the recogniser algorithm (Sec. 3), improvements of the recogniser by means of bit-vector operations (Sec. 4), and the generation of parse forests (Sec. 5), and Viterbi parses (Sec. 6). Section 7 discusses the advantages of the new architecture, Section 8 describes experimental results, and Section 9 summarises the paper.

2 Grammar Transformation

The CKY algorithm requires a grammar in Chomsky normal form where the right-hand side of each rule either consists of two non-terminals or a single terminal symbol. BitPar uses a modified version of the CKY algorithm allowing also chain rules (rules with a single non-terminal on the right-hand side). BitPar expects that the input grammar is already epsilon-free and that terminal symbols only occur in unary rules. Rules with more than 2 non-terminals on the right-hand side are split into binary rules by applying a transformation algorithm proposed by Andreas Eisele[1]. It is a greedy algorithm which tries to minimise the number of binarised rules by combining frequently cooccurring symbols first. The algorithm consists of the following two steps which are iterated until all rules are either binary or unary.

1. Compute the frequencies of the pairs of neighboring symbols on the right-hand sides of rules. (The rule A → B C D, e.g., adds 1 to the counts of ⟨B,C⟩ and ⟨C,D⟩, respectively.)
2. Determine the most frequent pair ⟨A,B⟩. Add a new non-terminal X. Replace the symbol pair

[1] personal communication

A B in all grammar rules with X. Finally, add the rule X → A B to the grammar.

3 Computation of the Chart

In the first step, the parser computes the CKY-style recogniser chart with the algorithm shown in Figure 1. It uses the transformed grammar with grammar rules P and non-terminal symbol set N. The chart is conceptually a three-dimensional bit array containing one bit for each possible constituent. A bit is 1 if the respective constituent has been inserted into the chart and 0 otherwise. The chart is indexed by the start position, the end position and the label of a constituent[2]. Initially all bits are 0. This chart representation is particularly efficient for highly ambiguous grammars like treebank grammars where the chart is densely filled.

```
1  recognise(P,N,$w_1$,...,$w_n$)
2    allocate and initialise chart[1..n][1..n][N] to 0
3    for e ← 1 to n do
4      for each non-terminal A with A → $w_e$ ∈ P do
5        chart[e][e] ← chart[e][e] | chainvec[A]
6      for b ← e−1 down to 1 do
7        for each non-terminal A ∈ N do
8          if chart[b][e][A] = 0 and
               derivable(P,N,b,e,A) then
9            chart[b][e] ← chart[b][e] | chainvec[A]

10 derivable(P,N,b,e,A)
11   for each rule A → B C ∈ P do
12     for m ← b to e−1 do
13       if chart[b][m][B] = 1 and
            chart[m+1][e][C] = 1 then
14          return true
15   return false
```

Figure 1: CKY-recogniser

Like other CKY-style parsers, the recogniser consists of several nested loops. The first loop (line 3 in Fig. 1) iterates over the end positions e of constituents, inserts the parts of speech of the next word (lines 4 and 5) into the chart, and then builds increasingly larger constituents ending at position e. To this end, it iterates over the start positions b from e-1 down to 1 (line 6) and over all non-terminals A (line 7). Inside the innermost loop, the function `derivable` is called to compute whether a constituent of category A covering words w_b through w_e is derivable from smaller constituents via some binary rule. `derivable` loops over all rules A → B C with the symbol A on the left-hand side (line 11) and over all possible end positions m of the first symbol on the right-hand side of the rule (line 12). If the chart contains B from position b to m and C from position m+1 to e (line 13), the function returns true (line 14), indicating that w_b through w_e are reducible to the non-terminal A. Otherwise, the function returns false (line 15).

In order to deal with chain rules, the parser precomputes for each category C the set of non-terminals D which are derivable from C by a sequence of chain rule reductions, i.e. for which D $\overset{*}{\Rightarrow}$ C holds, and stores them in the bit vector chainvec[C]. The set includes C itself. Given the grammar rules NP → DT N1, NP → N1, N1 → JJ N1 and N1 → N, the bits for NP, N1 and N are set in chainvec[N]. When a new constituent of category A starting at position b and ending at position e has been recognised, all the constituents reachable from A by means of chain rules are simultaneously added to the chart by or-ing the precomputed bit vector chainvec[A] to chart[b][e] (see lines 5 and 9 in Fig. 1).

The first parsing step is a pure recogniser which computes the set of constituents to which the input words can be reduced, but not their analyses. Therefore it is not necessary to look for further analyses once the first analysis of a constituent has been found. The function `derivable` therefore returns as soon as the first analysis is finished (line 13 and 14), and `derivable` is not called if the respective constituent was previously derived by chain rules (line 8).

Because only one analysis has to be found and some rules are more likely than others, the algorithm is optimised by trying the different rules for each category in order of decreasing frequency (line 11). The frequency information is collected online during parsing.

Derivation of constituents by means of chain rules is much cheaper than derivation via binary rules. Therefore the categories in line 7 are ordered such that categories from which many other categories are derivable through chain rules, come first.

The chart is actually implemented as a single large bit-vector with access functions translating index triples (start position, end position, and symbol number) to vector positions. The bits in the chart are ordered such that chart[b][e][n+1] follows after chart[b][e][n], allowing the efficient insertion of a set of bits with an or-operation on bit vectors.

[2] Start and end position of a constituent are the indices of the first and the last word covered by the constituent.

4 Using Bit-Vector Operations

The function `derivable` is the most time-consuming part of the recogniser, because it is the only part whose overall runtime grows cubically with sentence length. The inner loop of the function iterates over the possible end positions of the first child constituent and computes an and-operation for each position. This loop can be replaced by a single and-operation on two bit vectors, where the first bit vector contains the bits stored in chart[b][b][B], chart[b][b+1][B] ... chart[b][e-1][B] and the second bit vector contains the bits stored in chart[b+1][e][C], chart[b+2][e][C] ... chart[e][e][C]. The bit-vector operation is overall more efficient than the solution shown in Figure 1 if the extraction of the two bit vectors from the chart is fast enough. If the bits in the chart are ordered such that chart[b][1][A] ... chart[b][N][A] are in sequence, the first bit vector can be efficiently extracted by block-wise copying. The same holds for the second bit vector if the bits are ordered such that chart[1][e][A] ... chart[n][e][A] are in sequence. Therefore, the chart of the parser which uses bit-vector operations, internally consists of two bit vectors. New bits are inserted in both vectors.

```
recognise(P,N,$w_1$,...,$w_n$)
  allocate and initialise chart[1..n][1..n][N] to 0
    allocate vec[N]
  for e ← 1 to n do
    initialise vec[N] to 0
    for each non-terminal A with $A \rightarrow w_e \in$ P do
      vec ← vec | chainvec[A]
    chart[e][e] ← chart[e][e] | vec
    for b ← e−1 down to 1 do
      initialise vec[N] to 0
      for each non-terminal A ∈ N do
        if vec[A] = 0 and derivable(P,N,b,e,A) then
          vec ← vec | chainvec[A]
      chart[b][e] ← chart[b][e] | vec

derivable(P,N,b,e,A)
  for each rule A → B C ∈ P do
    vec1 ← chart[b][b...e-1][B]
    vec2 ← chart[b+1...e][e][C]
  return vec1 & vec2 ≠ 0
```

Figure 2: optimised CKY-recogniser

Due to the new representation of the chart, the insertion of bits into the chart by means of the operation chart[b][e] ← chart[b][e] | vec cannot be done with bit vector operations, anymore. Instead, each 1-bit of the bit vector has to be set separately in both copies of the chart. Binary search is used to extract the 1-bits from each machine word of a bit vector. This is more efficient than checking all bits sequentially if the number of 1-bits is small. Figure 3 shows how the 1-bits would be extracted from a 4-bit word v and stored in the set s. The first line checks whether any bit is set in v. If so, the second line checks whether one of the first two bits is set. If so, the third line checks whether the first bit is 1 and, if true, adds 0 to s. Then it checks whether the second bit is 1 and so on.

```
1   if $v \neq 0$ then
2     if $v$&1100 $\neq 0$ then
3       if $v$&1000 $\neq 0$ then
4         s.add(0)
5       if $v$&0100 $\neq 0$ then
6         s.add(1)
7     if $v$&0011 $\neq 0$ then
8       if $v$&0010 $\neq 0$ then
9         s.add(2)
10      if $v$&0001 $\neq 0$ then
11        s.add(3)
```

Figure 3: Extraction of the 1-bits from a bit vector

5 Parse Forest Generation

The chart only provides information about the constituents, but not about their analyses. In order to generate a parse forest representation of the set of all analyses, the chart is traversed top-down, reparsing all the constituents in the chart which are part of a complete analysis of the input sentence. The parse forest is stored by means of six arrays named `catname`, `catnum`, `first-analysis`, `rule-number`, `first-child`, and `child`. `catnum[n]` contains the number of the category of the n^{th} constituent. `first-analysis[n]` is the index of the first analysis of the n^{th} constituent, and `first-analysis[n+1]-1` is the index of the last analysis. `rule-number[a]` returns the rule number of analysis `a`, and `first-child[a]` contains the index of its first child node number in the `child` array. The numbers of the other child nodes are stored at the following positions. `child[d]` is normally the number of the node which forms child `d`. However, if the child with number `d` is the input word w_e, the value of `child[d]` is $-e-1$ instead. A negative value in the child array therefore indicates a terminal node and allows decoding of the position of the respective word in the sentence. catname[catnum[child[first-child[first-analysis[n]]]]] is therefore the name of the category of the first child of the first analysis of the n^{th} constituent. The rule-number array is not needed to represent the structure of the parse forest, but speeds up the retrieval of rule probabilities and similar information.

The parse forest shown in Figure 4 is represented by

catname = [A,B,C,D]
catnum = [0,1,2,3]
first-analysis = [0,2,3,4]
rule-number = [1,2,3,4,5]
first-child = [0,2,4,5,6]
child = [1,2,1,3,-1,-2,-2]

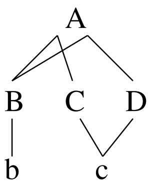

Figure 4: Parse forest with two analyses for A

The parse forest is built by the function parse shown in Figure 5. The function new-node(b,e,A) adds the number of A at the end of the catnum array. It also adds the currently biggest index of the first-child array plus 1 to the first-analysis array. It returns the largest index of the catnum array as node number. new-node also stores a mapping from the triple ⟨b,e,A⟩ to the respective node number n in a hash table. The hash table is used by get-node(b,e,A) to checks whether a constituent has already been added to the parse forest and, if true, returns its number. add-analysis(n,r,m) increments the size of the child array by 2 and adds the index of the first new element to the first-child array. It further adds the number of rule r to the rule-number array and stores the pair ⟨r,m⟩ in a temporary array which is later accessed in lines 17, 19, and 22. add-analysis(n,r) is similar, but adds just one element to the child array. Finally, the function add-child inserts the child node indices returned by recursive calls of build-subtree. The optimisation with bit-vector operations described in section 4 is also applicable in lines 14 and 15.

```
1  parse(P,N,S,$w_1$,...,$w_n$)
2     recognise(P,N,$w_1$,...,$w_n$)
3     return build-subtree(P,N,1,n,S)

4  build-subtree(P,N,b,e,A)
5     n ← get-node(b,e,A)
6     unless defined n do
7        n ← new-node(b,e,A)
8        if b = e and $r$ = A → $w_e$ ∈ P do
9           add-analysis(n,r)
10       for each rule $r$ = A → B ∈ P do
11          if chart[b][e][B] = 1 then
12             add-analysis(n,r)
13       for each rule $r$ = A → B C ∈ P do
14          for m ← b to e-1 do
15             if chart[b][m][B] = 1 and
                     chart[m+1][e][C] = 1 then
16                add-analysis(n,r,m)
17       for each analysis a = ⟨A → $w_e$⟩ of node n do
18          add-child(n,a,-e)
19       for each analysis a = ⟨A → B⟩ of node n do
20          d ← build-subtree(P,N,b,e,B)
21          add-child(n,a,d)
22       for each analysis a = ⟨A → B C, m⟩ do
23          d ← build-subtree(P,N,b,m,B)
24          add-child(n,a,1,d)
25          d ← build-subtree(P,N,m+1,e,C)
26          add-child(n,a,2,d)
27    return n
```

Figure 5: Parse forest generation

6 Viterbi Parsing

Viterbi parses for probabilistic context-free grammars (PCFGs) could be extracted from context-free parse forests, but BitPar computes them without building the parse forest in order to save space. After building the recogniser chart, the Viterbi version of BitPar filters the chart as shown in Figure 6 in order to eliminate constituents which are not part of a complete analysis.

After filtering the chart, the Viterbi probabilities of the remaining constituents are computed by the algorithm in figure 7. p[b][e][A] is implemented with a hash table. The value of prob(r) is 1 if the left-hand side of r is an auxiliary symbol inserted during the grammar transformation and otherwise the probability of the corresponding PCFG rule.

Finally, the algorithm of figure 8 prints the Viterbi parse.

7 Discussion

BitPar was developed for the generation of parse forests with large treebank grammars. It saves memory by splitting parsing into two steps, (1) the gen-

```
filter(P,S,chart)
  allocate and initialise chart2[1..n][1..n][N] to 0
  if chart[1][n][S] = 1 then
    filter-subtree(P,1,n,S,chart,chart2)
  chart $\leftarrow$ chart2

filter-subtree(P,b,e,A,chart,chart2)
  if chart2[b][e][A] = 1 then
    return  // chart2[b][e][A] was processed before
  chart2[b][e][A] $\leftarrow$ 1
  for each rule A $\rightarrow$ B $\in$ P do
    if chart[b][e][B] = 1 then
      filter-subtree(P,b,e,B,chart,chart2)
  for each rule r = A $\rightarrow$ B C $\in$ P do
    for m $\leftarrow$ b to e-1 do
      if chart[b][m][B] = 1 and
          chart[m+1][e][C] = 1 then
        filter-subtree(P,b,m,B,chart,chart2)
        filter-subtree(P,m+1,e,C,chart,chart2)
```

Figure 6: Filtering algorithm

```
viterbi(P,N,$w_1$,...,$w_n$,chart)
  for e $\leftarrow$ 1 to n do
    for each A $\in$ N with r = A $\rightarrow$ $w_e$ $\in$ P do
      if chart[e][e][A] = 1 then
        add-prob(e,0,e,A,r)
    for b $\leftarrow$ e$-$1 down to 1 do
      for each non-terminal A $\in$ N do
        if chart[b][e][A] = 1 then
          for each rule r = A $\rightarrow$ B C $\in$ P do
            for m $\leftarrow$ b to e$-$1 do
              if chart[b][m][B] = 1 and
                  chart[m+1][e][C] = 1 then
                add-prob(b,m,e,A,r)

add-prob(b,m,e,A,r)
  if r = A $\rightarrow$ w then
    p $\leftarrow$ prob(r)
  else if r = A $\rightarrow$ B then
    p $\leftarrow$ prob(r) * p[b][e][B]
  else if r = A $\rightarrow$ B C then
    p $\leftarrow$ prob(r) * p[b][m][B] * p[m+1][e][C]
  if undefined p[b][e][A] or p[b][e][A] < p then
    p[b][e][A] $\leftarrow$ p
    for each r = D $\rightarrow$ A $\in$ P do
      add-prob(b,0,e,D,r)
```

Figure 7: Computation of Viterbi probabilities

```
vparse(P,S,chart,$w_1$,...,$w_n$)
  return build-vparse(P,1,n,S)

build-vparse(P,b,e,A)
  print "(" A
  if b = e and $r$ = A $\rightarrow$ $w_e$ $\in$ P and
      p[b][e][A] = prob(r) then
    print $w_e$ ")" and return
  for each rule $r$ = A $\rightarrow$ B $\in$ P do
    if chart[b][e][B] = 1 and
        p[b][e][A] = p[b][e][B] * prob(r) then
      build-vparse(P,b,e,B)
      print ")" and return
  for each rule $r$ = A $\rightarrow$ B C $\in$ P do
    for m $\leftarrow$ b to e-1 do
      if chart[b][m][B] = 1 and chart[m+1][e][C] = 1
          and p[b][e][A] = p[b][m][B] * p[m+1][e][C]
                * prob(r) then
        build-vparse(P,b,m,B)
        build-vparse(P,m+1,e,C)
      print ")" and return
```

Figure 8: Generation of Viterbi parse

eration of a recogniser chart which is compactly stored in a bit-vector, and (2) the generation of the parse forest. Parse forest nodes are only created for constituents which are part of a complete analyses, whereas standard 1-pass chart parsers create more nodes which are later abandoned.

Viterbi parsing involves four steps. About 15 % of the parse time is needed for building the chart, 28 % for filtering, and 57 % for the computation of the Viterbi probabilities. The time required for the extraction of the best parse is negligible (0.04 %). The Viterbi step spends about 80 % of the time (45 % of the total time) on the computation of the probabilities and only about 20 % on the computation of the possible analyses. So, although Viterbi probabilities are only computed for nodes which are part of a valid analysis, it still takes almost half of the time to compute them, and the proportion increases with sentence length.

In contrast to most beam search parsing strategies, BitPar is guaranteed to return the most probable analysis, and there is no need to optimise any scoring functions or parameters.

8 Experiments

The parser was tested with a grammar containing 65,855 grammar rules, and 4,444 different categories. The grammar was extracted from a version of the Penn treebank which was annotated with additional features similar to (Klein and Manning, 2003b). The average rule length has 3.7 (without parent category). The experiments were conducted on a Sun Blade 1000 Model 2750 server with 750 MHz CPUs and 4 GB memory.

In a first experiment, 1000 randomly selected sentences from the PENN treebank containing 24,595 tokens were parsed. Viterbi parsing of these sentences took 27,596 seconds (1.14 seconds per word). The generation of parse forests[3] for the same sentences took 26,840 seconds (1.09 seconds per word).

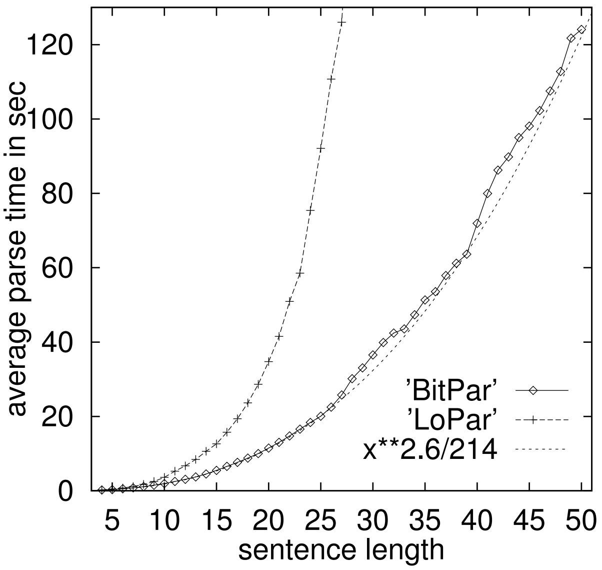

Figure 9: Average parse times

In another experiment, we examined how parse times increase with sentence length. Figure 9 shows the average Viterbi parse times of BitPar for randomly selected sentences of different lengths[4]. For comparison, the average parse times of the LoPar parser (Schmid, 2000) on the same data are also shown. LoPar is a 1-pass left-corner chart parser which computes the Viterbi parse from the parse forest. BitPar is faster for all sentence lengths and the growth of the parse times with sentence length is smaller than for LoPar. Although the asymptotic runtime complexity of BitPar is cubic, figure 9 shows that the exponent of the actual growth function in the range between 4 and 50 is about 2.6. This can be explained by the fact that the bit-vector operations become more effective as the length of the sentence and therefore the length of the bit-vectors increases.

The memory requirements of BitPar are far lower than those of LoPar. LoPar needs about 1 GB memory to parse sentences of length 22, whereas BitPar allocates 180 MB during parse forest generation and 55 MB during Viterbi parsing. For the longest sentence in our 1000 sentence test corpus with length 55, BitPar needed 113 MB to generate the Viterbi parse and 3,185 MB to compute the parse forest. LoPar was unable to parse sentences of this length.

We are planning to evaluate the influence of the different optimisations presented in the paper on parsing speed and to compare it with other parsers than LoPar.

9 Summary

A bit-vector based implementation of the CKY algorithm for large highly ambiguous grammars was presented. The parser computes in the first step a recogniser chart and generates the parse forest in a second step top-down by reparsing the entries of the chart. Viterbi parsing consists of four steps comprising (i) the generation of the chart, (ii) top-down filtering of the chart, (iii) computation of the Viterbi probabilities, and (iv) the extraction of the Viterbi parse. The basic parsing operation (building new constituents by combining two constituents according to some binary rule) is parallelised by means of bit-vector operations.

The presented method is efficient in terms of runtime as well as space requirements. The empirical runtime complexity (measured for sentences with up to 50 words) is better than cubic.

The presented parser is particularly useful when the whole set of analyses has to be computed rather than the best parse. The Viterbi version of the parser is guaranteed to return the most probable parse tree and requires no parameter tuning.

References

Charniak, E., Goldwater, S., and Johnson, M. (1998). edge-based best-first chart parsing. In *Proceedings of the Sixth Workshop on Very Larger Corpora*, pages 127–133. Morgan Kaufmann.

Graham, S., Harrison, M., and Ruzzo, W. (1980). An improved context-free recognizer. *ACM Transactions on Programming Languages and Systems*, 2(3):415–462.

[3] The parse forest were only generated but not printed.

[4] The two bulges of the BitPar curve were probably caused by a high processor load. The experiment will be repeated for the final version of the paper.

Kasami, T. (1965). An efficient recognition and syntax analysis algorithm for context-free languages. Technical Report AFCRL-65-758, Air Force Cambridge Research Laboratory.

Klein, D. and Manning, C. D. (2003a). A* parsing: Fast exact viterbi parse selection. In *Proceedings of HLT-NAACL 03*.

Klein, D. and Manning, C. D. (2003b). Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*.

Schmid, H. (2000). *LoPar: Design and Implementation*. Number 149 in Arbeitspapiere des Sonderforschungsbereiches 340. Institute for Computational Linguistics, University of Stuttgart.

Younger, D. H. (1967). Recognition of context-free languages in time n^3. *Information and Control*, 10:189–208.