in the number of possible candidates with increasing sentence size. The rate of growth is $O(2^nT^n)$ for the joint system, where n is the length of the sentence and T is the number of chunk types. It is natural to use some greedy heuristic search algorithms for inference in some similar joint problems (Zhang and Clark, 2008; Zhang and Clark, 2010). However, the greedy heuristic search algorithms only explore a fraction of the whole space (even with beam search) as opposed to dynamic programming. Additionally, a specific advantage of the dynamic programming algorithm is that constraints required in a valid prediction sequence can be handled in a principled way. We show that dynamic programming is in fact possible for this joint problem, by introducing some effective pruning schemes.

To make the inference tractable, we first make a first-order Markov assumption on the features used in our model. In other words, we assume that the chunk c_i and the corresponding label t_i are only associated with the preceding chunk c_{i-1} and the label t_{i-1}. Suppose that the input sentence has n words and the constant M is the maximum chunk length in the training corpus. Let $V(b,e,t)$ denote the highest-scored segmentation and labeling with the last chunk starting at word index b, ending at word index e and the last chunk type being t. One way to find the highest-scored segmentation and labeling for the input sentence is to first calculate the $V(b,n\text{-}1,t)$ for all possible start position $b \in (n\text{-}M)..n\text{-}1$, and all possible chunk type t, respectively, and then pick the highest-scored one from these candidates. In order to compute $V(b,n\text{-}1,t)$, the last chunk needs to be combined with all possible different segmentations of words $(b\text{-}M)..b\text{-}1$ and all possible different chunk types so that the highest-scored can be selected. According to the principle of optimality, the highest-scored among the segmentations of words $(b\text{-}M)..b\text{-}1$ and all possible chunk types with the last chunk being word $b'..b\text{-}1$ and the last chunk type being t' will also give the highest score when combined with the word $b..n\text{-}1$ and tag t. In this way, the search task is reduced recursively into smaller subproblems, where in the base case the subproblems $V(0,e,t)$ for $e \in 0..M\text{-}1$, and each possible chunk type t, are solved in straightforward manner. And the final highest-scored segmentation and labeling can be found by solving all subproblems in a bottom-up fashion.

The pseudo code for this algorithm is shown in Figure 1. It works by filling an n by n *by* T table *chart*, where n is the number of words in the input sentence *sent*, and T is the number of chunk types. *chart*$[b,e,t]$ records the value of subproblem $V(b,e,t)$. *chart*$[0, e, t]$ can be computed directly for $e = 0..M\text{-}1$ and for chunk type t=1..T. The final output is the best among *chart*$[b,n\text{-}1,t]$, with $b = n\text{-}M..n\text{-}1$, and t=1..T.

Inputs: sentence *sent* (word segmented and POS tagged)
Variables:
word index b for the start of chunk;
word index e for the end of chunk;
word index p for the start of the previous chunk.
chunk type index t for the current chunk;
chunk type index t' for the previous chunk;
Initialization:
for e = 0.. M-1:
  for t =1..T:
    chart$[0,e,t]$ ←single chunk *sent*$[0,e]$ and type t
Algorithm:
for e = 0..n-1:
  for b = (e-M)..e:
    for t =1..T:
      chart$[b,e,t]$←the highest scored segmentation and labeling among those derived by combining *chart*$[p,b\text{-}1,t']$ with *sent*$[b,e]$ and chunk type t, for $p = (b\text{-}M)..b\text{-}1$, t' =1..T.
Outputs: the highest scored segmentation and labeling among *chart*$[b,n\text{-}1,t]$, for b=n-M..n-1, t =1..T.

Figure 1: A dynamic-programming algorithm for phrase chunking.

4.2 Pruning

The time complexity of the above algorithm is $O(M^2T^2n)$, where M is the maximum chunk size. It is linear in the length of sentence. However, the constant in the O is relatively large. In practice, the search space contains a large number of invalid partial candidates, which make the algorithm slow. In this section we describe three partial output pruning schemes which are helpful in speeding up the algorithm.