Dependency parsing

- Dependency parsing (DP) is a word centric parse that builds named, ordered relations between pairs of words in a sentence. The two words at either end of a relation are called respectively the **head** and the **dependent**.
- While DP has a long history it has become very widely used in the last approximately 5 years. A cross linguistic universal dependency tag list has been defined and tree banks labelled using the universal tag list are under active construction in over 60 languages.
- These tree banks can then be used as training sets to learn models that when given a sentence in the language will create a dependency parse.
- DPs turns out to be much more useful in the newer ML based NLP algorithms compared the traditional constituent parse.

A sample screenshot of UD trees

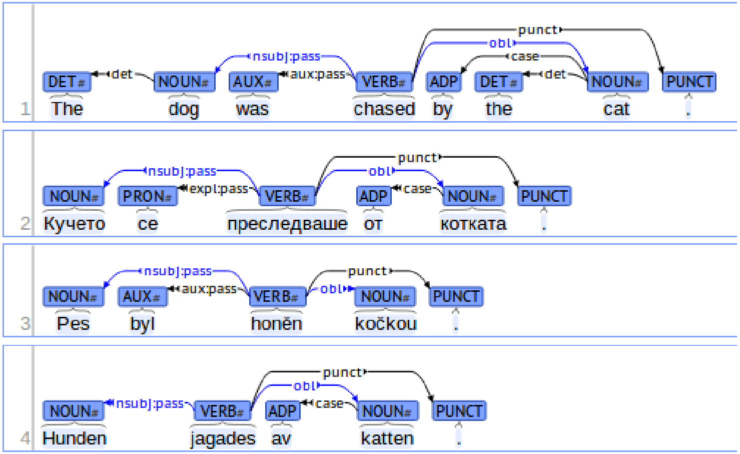

1 The dog was chased by the cat .
DET NOUN AUX VERB ADP DET NOUN PUNCT
det nsubj:pass aux:pass obl case det punct
2 Кучето се преследваше от котката .
NOUN PRON VERB ADP NOUN PUNCT
nsubj:pass expl:pass obl case punct
3 Pes byl honěn kočkou .
NOUN AUX VERB NOUN PUNCT
nsubj:pass aux:pass obl punct
4 Hunden jagades av katten .
NOUN VERB ADP NOUN PUNCT
nsubj:pass obl case punct

Figure: Parallel example in 4 languages, involving a passive verb, a nominal subj, oblique agent. The words are PoS tagged. Languages: English, Bulgarian, Czech, Swedish.

Dependency graph

Definition 1 (Dependency graph (DG))

For a sentence $w = w_0 w_1 \ldots w_n$ a dependency graph $G = (V, E)$ is a labelled digraph with label set R such that:

1. $V \subseteq \{w_0, \ldots, w_n\}$.
2. $E \subseteq V \times R \times V$.
3. *If* $(w_i, r, w_j) \in E$ *then* $(w_i, r') \notin E$ *for all* $r' \neq r$.

Condition 3) is called a mono-stratal DG where between any two nodes there can be at most one labelled edge. One can build multi-stratal DGs where different mono-stratal DGs are built for the same sentence. For example, meaning DG, syntax DG, morphology DG, etc.

Example DG

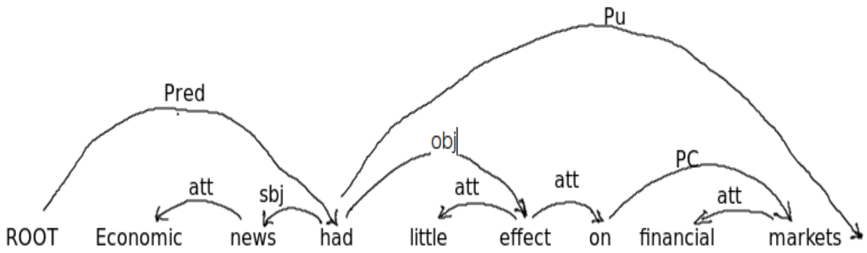


Figure: An example DG. By convention the graph originates at the ROOT and ends at the '.' ending the sentence.

Alternate representation as a tree

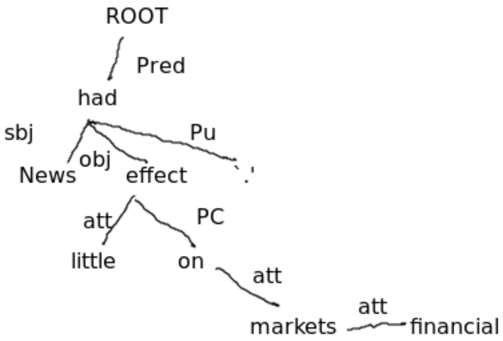


Figure: Earlier example represented as a directed tree.

Properties

A well formed DG for sentence S is a directed tree originating at $w_0 = ROOT$ and has spanning node set $V = V_S$. $w_i \rightarrow w_j$ is drawn if $(w_i, r, w_j) \in E$ for some $r \in R$. $w_i \overset{\star}{\Rightarrow} w_j$, the reflexive transitive closure of $\rightarrow$. $w_i \leftrightarrow w_j$ if either $w_i \rightarrow w_j$ or $w_j \rightarrow w_i$. Similarly, $w_i \overset{\star}{\leftrightarrow} w_j$. It also satisfies the spanning property - that is $V = V_S$.

A tree is a **projective** DT if there is a directed path from the head node to all nodes between the head and the dependent node of the head and this is true for each head in the tree. If a DT is not projective it is called **non-projective**.

Most common English sentences are projective. But non-projective examples are possible and legal. Though the non-projective example can be converted to a projective one by rearranging a few words in the sentence.

Root property: $\nexists w_i \in V$ such that $w_i \rightarrow w_0$.

Spanning property: $V = V_S$. V_S is the set of words in the sentence.

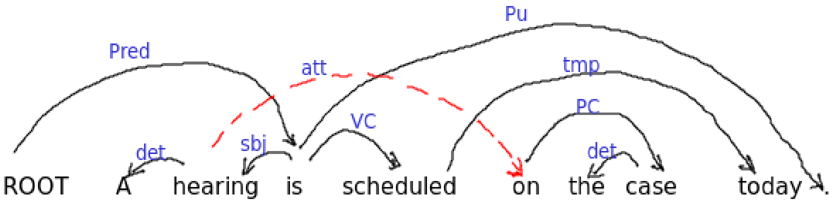
Weak connectedness: For every pair w_i, w_j where $w_i \neq w_j$ $w_i \overset{\star}{\leftrightarrow} w_j$. Every node is weakly connected to every other node.

Single head: For a pair $w_i \rightarrow w_j$, there is no node $w_k \neq w_i$ such that $w_k \rightarrow w_j$. That is every dependent has exactly one head.

Acylicity: There are no directed cycles in a dependency graph.

Nested property: A $\underline{\text{projective DG=(V,E)}}$ satisfies the nested property. For all nodes $w_i \in V$, the set of words $\{w_j | w_i \overset{\star}{\Rightarrow} w_j\}$ is a contiguous subsequence of the sentence w.

Non-projective example



Algorithms for dependency parsing

- Multiple algorithms exist. Variants of CYK and Earley.
- Graph based algorithms, spanning tree based algorithms.
- The algorithms based on transitions are widely used.
- Ensemble based parsers.
- Constituency based DPs.

Transition based algorithms

- Many transition systems are possible. A transition system defines transitions between defined configurations.
- One definition of a configuration is $c = (\sigma, \beta, E)$ where σ is a stack of words $w_i \in V_S$, β is a buffer of words $w_i \in V_S$ and E is a set of dependency edges $(w_i, r, w_j \in V_S \times R \times V_S)$, R is a set of dependency types or relations and V_S is the set of words $w_0, \ldots, w_n$.
  σ contains partially processed words, β contains the remaining input words and E is the partially built dependency graph.

  Example: `ROOT Economic news had little effect on financial markets.`

  ([ROOT, news], [had,little,effect,on,financial,markets,.], $\{(news, ATT, Economic)\}$)
- For any sentence $w = w_0, w_1, \ldots, w_n$, the initial configuration is: $([w_0], [w_1, \ldots, w_n], \emptyset)$. A terminal configuration is: $(\sigma, [\,], E)$ for any σ, E.

Shift-reduce dependency parsing

Notation: $\sigma|w_i$: w_i at top of stack; $w_j|\beta$: w_j at head of buffer.

- left-arc(r): $(\sigma|w_i, w_j|\beta, E) \rightarrow (\sigma, w_j|\beta, E \cup \{(w_j, r, w_i)\})$, where $i \neq 0$ - ensures root property.
- right-arc(r): $(\sigma|w_i, w_j|\beta, E) \rightarrow (\sigma, w_i|\beta, E \cup \{w_i, r, w_j\})$, both stack and buffer must be non-empty.
- shift: $(\sigma, w_i|\beta, E) \rightarrow (\sigma|w_i, \beta, E)$, the buffer must be non-empty

The above is one transition system. Others are possible.

Transition sequence

- Let $\mathcal{T}$ be the set of all permitted transitions in a given transition system.
- A transition sequence for $w = w_0, \ldots, w_n$ is sequence of configurations $C_{0,m} = (c_0, c_1, \ldots, c_m)$ such that:
  - c_0 is the initial configuration.
  - c_m is a final configuration.
  - For all $1 \leq i \leq m$, $c_{i+1} = t(c_i)$, $t \in \mathcal{T}$
- Every transition sequence defines a DG with spanning, root and single-head property. But it may not be connected - that is may not be a DT. A trivial example: a sequence of shifts. Leads to dependency forest that can be converted to a DT.

Example

Operation	**Configuration**
	([ROOT],[Economic,...],$\emptyset$)
$\overset{sh}{\Rightarrow}$	([ROOT,Economic],[news,...],$\emptyset$)
$\overset{la(ATT)}{\Rightarrow}$	([ROOT],[news,...],$E_1 = \{(news, ATT, Economic)\}$)
$\overset{sh}{\Rightarrow}$	([ROOT,news],[had,...],E_1)
$\overset{la(SBJ)}{\Rightarrow}$	([ROOT],[had,...],$E_2 = E_1 \cup \{(had, SBJ, news)\}$)
$\overset{sh}{\Rightarrow}$	([ROOT,had],[little,...],E_2)
$\overset{sh}{\Rightarrow}$	([ROOT,had,little],[effect,...],E_2)
...	
$\overset{la(ATT)}{\Rightarrow}$	([ROOT,...,on],[markets,...],$E_4 = E_3 \cup \{(markets, ATT, financial)\}$)
$\overset{ra(PC)}{\Rightarrow}$	([ROOT,had,effect],[on,...],$E_5 = E_4 \cup \{(on, PC, markets)\}$)
...	
$\overset{ra(PRED)}{\Rightarrow}$	([ ],[ROOT],$E_9 = E_8 \cup \{(ROOT, PRED, had)\}$)
$\overset{sh}{\Rightarrow}$	([ROOT],[ ], E_9)

Parsing alg.

- The transition system defined earlier is non-deterministic since more that one operation is possible in a configuration.
- To obtain a deterministic parse assume there is an oracle o that given a configuration gives a transition operation t. That is: $t = o(c)$ iff t is the correct transition operation at c.
- The following simple alg. is then possible:

Algorithm 0.1: PARSE(w, o)

```
$c \leftarrow c_0(w)$
**while** (c is not terminal)
  **do** $\begin{cases} t \leftarrow o(c) \\ c \leftarrow t(c) \end{cases}$
**return** $(DG_c)$
```

Formalizing the oracle

- Let C be the set of all configurations for any sentence w. Assume we have a feature function $f : C \rightarrow \mathcal{Y}$ where $\mathcal{Y}$ is an $m-$dimensional feature space.
- We want to learn a classifier $g : \mathcal{Y} \rightarrow \mathcal{T}$ such that $g(f(c)) = o(c)$ for any configuration c.
- To do the above we must:
  - Represent configurations by feature vectors. That is define f.
  - Define/construct the training data.
  - Define a learning algorithm that constructs the classifier g.

Features

- Features of a configuration can be defined in multiple ways. They can be word position + word based features from both the stack, the buffer and the current edge set. Alternately, they can be dense features in the form of embeddings in a vector space (similar to word embeddings). Such dense feature vectors are much more recent[1]
- Features constructed from word position and word features are called *configurational word features*.

[1] Chen, Manning, A Fast and Accurate Dependency Parser using Neural Networks, EMNLP 2014.

Address functions

- Configurational word features are broken into two parts - an address or position part which locates the word and one or more feature of the word.
- Address/position can be related to stack or buffer or graph.
  $\sigma[i]$ - i^{th} word in the stack top of stack is $\sigma[0]$.
  $\beta[i]$ - i^{th} word in the buffer where $\beta[0]$ is the front of the buffer.
  $lc(\sigma[i])$ - left-most child in DG of the i^{th} word in the stack.
  The above are called address functions that take a configuration c and output a word $w_i \in V_S$. In case such a w_i does not exist we use NULL.

Feature values

- Given an address function that returns a word w_i we can define specific features of w_i. For example: lemma, PoS tag, head or dependent in DG, dependency relation.
- Some features can be static like PoS tag. Others can be dynamic for example a dependency relation in a particular configuration.
- Other typical features can be:
  - Distance between $\sigma[0]$ and $\beta[0]/\beta[1]$ etc.
  - Number of children in DG.
  - Number of left children in DG.
  - Number of right children in DG.

  etc.

Example

Suppose we have the following address functions:

$\sigma[0]$, $\beta[0]$, $\beta[1]$, $ldep(\sigma[0])$, $rdep(\sigma[0])$, $ldep(\beta[0])$, $rdep(\beta[0])$. And let the features be: word/lemma, PoS, dependency relation. Then we have a vector that has length 21 assuming that PoS, dependency relation are categorial variables. Often such features are one-hot vectors which will considerably increase the vector size. Using the configurations from the transition sequence in slide 13:

Example: `ROOT Economic news had little effect on financial markets.`

c_0=([ROOT],[Economic,...],$\emptyset$)

(ROOT,NULL,NULL,Economy,JJ,NULL,news,NN,NULL,NULL,NULL,NULL,NULL,NULL,
NULL,NULL,NULL,NULL,NULL,NULL,NULL)

c_4=([ROOT],[had,...],$E_2 = \{(had, SBJ, news), (news, ATT, economic)\}$)

(ROOT,NULL,NULL,had,VB,SBJ,little,JJ,NULL,NULL,NULL,NULL,NULL,NULL,NULL,
news,NN,ATT,NULL,NULL,NULL)

Actually, we need a 1-hot representation for dependency tags since ‘news’ has two dependency tags. This will immediately increase the vector size to 57.

Training data

- Having defined a feature vector for any configuration we need to get training data to learn a classifier.
- Data available is DG treebanks in the form (w, DG) where w is a sentence (possibly PoS tagged) and DG is the dependency grapy of w.
- The above data must be converted to a form such that we learn a classifier/oracle (o) which given a configuration c outputs a transition operation t - $o : C \rightarrow \mathcal{T}$.
- Convert each (w, DG) pair to a transition sequence $C_{0,m}$ where c_0 is the starting configuration and c_m is the terminal configuration where the edge set E (in the terminal configuration) is the dependency graph DG.
- This will produce a learning set of pairs: $(f(c), t)$.

Training data - contd.

- Assuming DG is projective we can do it as follows:

  For the transition sequence $C_{0,m}$

  - $c_0 = c_0(w)$.
  - $DG = (V_s, E_m)$.
  - For every non-terminal configuration $c_i \in C_{0,m}$ add pair $(f(c_i), t_i)$ where $t_i(c_i) = c_{i+1}$ and $f(c_i)$ is the feature vector for configuration c_i.
- Above assumes given (w, DG) we can construct a transition sequence for w that results in DG.
- This is possible for projective DG using the parsing algorithm (and transition system) defined earlier (slide 11) where oracle o is defined by:

$$o(c = (\sigma, \beta, E)) = \begin{cases} la(r) & \text{if } (\beta[0], r, \sigma[0]) \in DG \\ ra(r) & \text{if } (\sigma[0], r, \beta[0]) \in DG \text{ and } (\forall w_k, r' \text{ if } (\beta[0], r', w_k) \in DG \\ & \quad \text{then } (\beta[0], r', w_k) \in E) \\ sh & \text{Else} \end{cases}$$

$ra(r)$ says $(\sigma[0], r, \beta[0])$ added only if all outgoing arcs from $\beta[0]$ are already in E - because $\beta[0]$ will exit from σ and β and cannot be part of any more edges.

Oracle or classifier

- Different classifiers can be learnt. We need a multi-class classifier:
  - Neural network.
  - Decision tree.
  - SVM.
  - Memory table with a similarity function on configurations.

Problems with configurational word features

- Size of the feature vector can become very large when we try to capture higher order interactions. This leads to very sparse vectors with very little data support for proper learning.
- The feature template is manually constructed so it can be incomplete.
- Empirically most of the time in a parse is taken up by feature vector computation.

Dense vectors (embedded vector)

- Use ideas from word2vec to create dense feature vectors.
- Architecture of neural network:

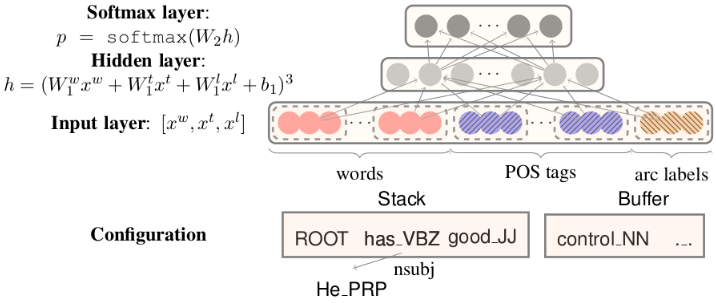


Figure: Neural n/w for embedded configuration feature vector. (Src: Chen, Manning)

Neural n/w

- 3 layer n/w with one hidden layer of size d_h.
- Input layer contains embeddings for words, their PoS tags, relation labels of address words in DG. (see next slide for details)
- Hidden layer values computed by:
  $h = (W_w x_w + W_t x_t + W_r x_r + b)^3$ Note: cubic activation.
  where $x_w = (e_1; \ldots; e_{n_w})$ input word embeddings, x_t input tag embeddings corresponding to words in x_w and x_r relation label embeddings of labels corresponding to the addresses chosen from the DG.
- Cubic activation allows interaction between all the all three parts and between all possible pairs.
- Assuming $n_w = |x_w|$, similarly n_t and n_r the dimensions of the weight matrices are: $dim(W_w) = d_h \times (d.n_w)$, $dim(W_t) = d_h \times (d.n_t)$, $dim(W_r) = d_n \times (d.n_r)$.
- A softmax output layer with a weight matrix $dim(W_2) = \mathcal{T} \times d_h$

Input

- Choose stack addresses, buffer addresses, DG addresses manually.
- Following Nivre; Chen and Manning chose the following addresses: $\sigma[0]$ to $\sigma[2]$, $\beta[0]$ to $\beta[2]$ - 6; the first two leftmost/ rightmost children of $\sigma[0], \sigma[1]$ - 8; leftmost of leftmost and rightmost of rightmost of $\sigma[0], \sigma[1]$ - 4. This gives a total of 18 words, so 18 PoS tags and relation labels corresponding to the 12(=8+4) elements in the DG.
- Input made up of word2vec vectors of address words, corresponding vectors for PoS tags, label vectors for the tree addresses chosen from the embedding matrices E_w ($d \times |\mathcal{V}|$ for words), E_t ($d \times |PoS\ tags|$ for PoS tags), E_r ($d \times |R|$ for relation labels).
- Initialization for embedding matrices: For E_w word2vec vectors; for E_t, E_r random values from [-0.01,0.01].
- Weight matrices are intialized randomly to similar small values.

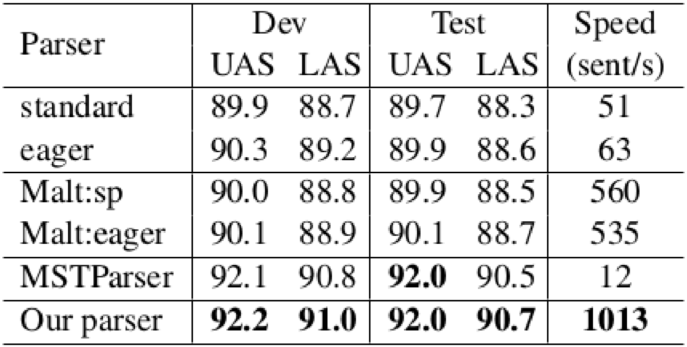
Training

- The training set is $\mathcal{L} = \{(c_i, t_i)\}$ generated from the treebank.
- Loss function: $L(\theta) = -\sum_i \log(p_{t_i}) + \frac{\lambda}{2} \parallel \theta \parallel^2$
  Cross entropy plus regularizer. θ is set of all parameters - Ws and Es.
- The error is backpropagated to change Ws and Es which learns the embedding for all three components of a configuration.
- Softmax calculated only over feasible transitions.
- Parameter values that give the best unlabelled attachment score are used for the final model.

Parsing

- To predict the transition operation an input vector is created from the embedding matrices for the configuration c_i and then fed forward through the learnt network to predict the transition t_i at the softmax layer. The next configuration is obtained by $c_{i+1} = t_i(c_i)$.

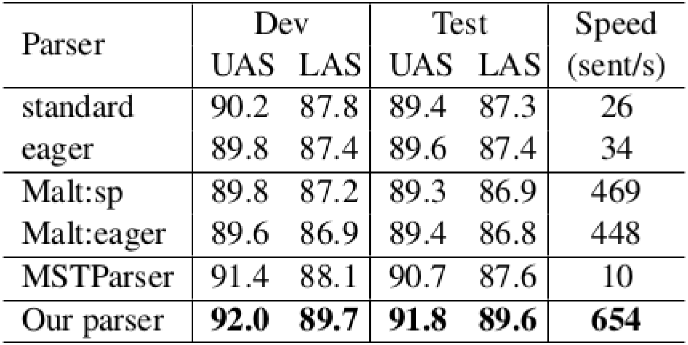
Performance



Parser	Dev		Test		Speed
	UAS	LAS	UAS	LAS	(sent/s)
standard	89.9	88.7	89.7	88.3	51
eager	90.3	89.2	89.9	88.6	63
Malt:sp	90.0	88.8	89.9	88.5	560
Malt:eager	90.1	88.9	90.1	88.7	535
MSTParser	92.1	90.8	**92.0**	90.5	12
Our parser	**92.2**	**91.0**	**92.0**	**90.7**	**1013**

Figure: Comparative performance of different parsers on Penn treebank using CoNLL dependencies.

Performance contd.



Parser	Dev		Test		Speed
	UAS	LAS	UAS	LAS	(sent/s)
standard	90.2	87.8	89.4	87.3	26
eager	89.8	87.4	89.6	87.4	34
Malt:sp	89.8	87.2	89.3	86.9	469
Malt:eager	89.6	86.9	89.4	86.8	448
MSTParser	91.4	88.1	90.7	87.6	10
Our parser	**92.0**	**89.7**	**91.8**	**89.6**	**654**

Figure: Comparative performance of different parsers on Penn treebank using Stanford dependencies.