Lecture 14: Graph-based dependency parsing
Announcements

- No recitation on Friday (Tartan Community Day).
Dependency parsing
Dependency parsing

- Transition-based (shift-reduce) parsing:
Dependency parsing

- Transition-based (shift-reduce) parsing:
  - Greedy choice of local transitions guided by a good classifier.
Dependency parsing

- Transition-based (shift-reduce) parsing:
  - Greedy choice of local transitions guided by a good classifier.
  - Examples: MaltParser [Nivre et al. 2008], Stack LSTM [Dyer et al. 2015]
Dependency parsing

- Transition-based (shift-reduce) parsing:
  - Greedy choice of local transitions guided by a good classifier.
  - Examples: MaltParser [Nivre et al. 2008], Stack LSTM [Dyer et al. 2015]

- Graph-based dependency parsing:
Dependency parsing

- Transition-based (shift-reduce) parsing:
  - Greedy choice of local transitions guided by a good classifier.
  - Examples: MaltParser [Nivre et al. 2008], Stack LSTM [Dyer et al. 2015]

- Graph-based dependency parsing:
  - Given scores for every pair of words, find the (globally) highest scoring set of edges.
Dependency parsing

- Transition-based (shift-reduce) parsing:
  - **Greedy** choice of local transitions guided by a good classifier.
  - Examples: MaltParser [Nivre et al. 2008], Stack LSTM [Dyer et al. 2015]

- Graph-based dependency parsing:
  - Given scores for every pair of words, find the (globally) highest scoring set of edges.
Unfortunately, this approach doesn’t always lead to a tree since the set of edges selected may contain cycles. Fortunately, in yet another case of multiple discovery, there is a straightforward way to eliminate cycles generated during the greedy selection phase.

Chu and Liu (1965) and Edmonds (1967) independently developed an approach that begins with greedy selection and follows with an elegant recursive cleanup phase that eliminates cycles.

The cleanup phase begins by adjusting all the weights in the graph by subtracting the score of the maximum edge entering each vertex from the score of all the edges entering that vertex. This is where the intuitions mentioned earlier come into play. We have scaled the values of the edges so that the weight of the edges in the cycle have no bearing on the weight of any of the possible spanning trees. Subtracting the value of the edge with maximum weight from each edge entering a vertex results in a weight of zero for all of the edges selected during the greedy selection phase, including all of the edges involved in the cycle.

Having adjusted the weights, the algorithm creates a new graph by selecting a cycle and collapsing it into a single new node. Edges that enter or leave the cycle are altered so that they now enter or leave the newly collapsed node. Edges that do not touch the cycle are included and edges within the cycle are dropped.

Now, if we knew the maximum spanning tree of this new graph, we would have what we need to eliminate the cycle. The edge of the maximum spanning tree directed towards the vertex representing the collapsed cycle tells us which edge to delete to eliminate the cycle. How do we find the maximum spanning tree of this new graph? We recursively apply the algorithm to the new graph. This will either result in a spanning tree or a graph with a cycle. The recursions can continue as long as cycles are encountered. When each recursion completes we expand the collapsed vertex, restoring all the vertices and edges from the cycle with the exception of the single edge to be deleted.

Putting all this together, the maximum spanning tree algorithm consists of greedy edge selection, re-scoring of edge costs and a recursive cleanup phase when needed. The full algorithm is shown in Fig. 15.13.

Fig. 15.14 steps through the algorithm with our Book that flight example. The first row of the figure illustrates greedy edge selection with the edges chosen shown in blue (corresponding to the set $F$ in the algorithm). This results in a cycle between...
Graph-based dependency parsing

- Edge-factored (or arc-factored) approaches:
Graph-based dependency parsing

- Edge-factored (or arc-factored) approaches:
  - Score of a tree decomposes as sum of edge scores:
    \[ \psi(y, w; \theta) = \sum_{i \rightarrow j \in y} \psi(i \rightarrow j, w, \theta) \]
Graph-based dependency parsing

- **Edge-factored** (or **arc-factored**) approaches:
  - Score of a tree decomposes as sum of edge scores:
    \[
    \Psi(y, w; \theta) = \sum_{i \rightarrow j \in y} \psi(i \rightarrow j, w, \theta)
    \]
  - Start with a fully-connected directed graph

![Initial rooted, directed graph for Book that flight.](image)

Figure 15.12

Unfortunately, this approach doesn’t always lead to a tree since the set of edges selected may contain cycles. Fortunately, in yet another case of multiple discovery, there is a straightforward way to eliminate cycles generated during the greedy selection phase.

Chu and Liu (1965) and Edmonds (1967) independently developed an approach that begins with greedy selection and follows with an elegant recursive cleanup phase that eliminates cycles.

The cleanup phase begins by adjusting all the weights in the graph by subtracting the score of the maximum edge entering each vertex from the score of all the edges entering that vertex. This is where the intuitions mentioned earlier come into play. We have scaled the values of the edges so that the weight of the edges in the cycle have no bearing on the weight of any of the possible spanning trees. Subtracting the value of the edge with maximum weight from each edge entering a vertex results in a weight of zero for all of the edges selected during the greedy selection phase, including all of the edges involved in the cycle.

Having adjusted the weights, the algorithm creates a new graph by selecting a cycle and collapsing it into a single new node. Edges that enter or leave the cycle are altered so that they now enter or leave the newly collapsed node. Edges that do not touch the cycle are included and edges within the cycle are dropped.

Now, if we knew the maximum spanning tree of this new graph, we would have what we need to eliminate the cycle. The edge of the maximum spanning tree directed towards the vertex representing the collapsed cycle tells us which edge to delete to eliminate the cycle. How do we find the maximum spanning tree of this new graph? We recursively apply the algorithm to the new graph. This will either result in a spanning tree or a graph with a cycle. The recursions can continue as long as cycles are encountered. When each recursion completes we expand the collapsed vertex, restoring all the vertices and edges from the cycle with the exception of the single edge to be deleted.

Putting all this together, the maximum spanning tree algorithm consists of greedy edge selection, re-scoring of edge costs and a recursive cleanup phase when needed. The full algorithm is shown in Fig. 15.13.

Fig. 15.14 steps through the algorithm with our Book that flight example. The first row of the figure illustrates greedy edge selection with the edges chosen shown in blue (corresponding to the set \(F\) in the algorithm). This results in a cycle between

- **Score of a tree decomposes as sum of edge scores:**

\[
\Psi(y, w; \theta) = \sum_{i \rightarrow j \in y} \psi(i \rightarrow j, w, \theta)
\]

- **Start with a fully-connected directed graph**
Graph-based dependency parsing

- **Edge-factored** (or **arc-factored**) approaches:
  - Score of a tree decomposes as sum of edge scores:
    \[
    \Psi(y, w; \theta) = \sum_{i \xrightarrow{L} j \in y} \psi(i \xrightarrow{L} j, w, \theta)
    \]
  - Start with a fully-connected directed graph
  - How to infer the highest scoring tree?
Graph-based dependency parsing

- **Edge-factored** (or **arc-factored**) approaches:
  - Score of a tree decomposes as sum of edge scores:
    \[
    \Psi(y, w; \theta) = \sum_{i \rightarrow j \in y} \psi(i \rightarrow j, w, \theta)
    \]
  - Start with a fully-connected directed graph
  - How to infer the highest scoring tree?
  - Find a **maximum directed spanning tree**: Chu and Liu (1965) and Edmonds (1967) algorithm
Chu-Liu-Edmonds algorithm

function MaxSpanningTree(G=(V,E), root, score) returns spanning tree

\[ F \leftarrow [] \]
\[ T' \leftarrow [] \]
\[ score' \leftarrow [] \]

for each \( v \in V \) do
\[ bestInEdge \leftarrow \arg \max_{(u,v) \in E} \text{score}[e] \]
\[ F \leftarrow F \cup \text{bestInEdge} \]
for each \( e=(u,v) \in E \) do
\[ \text{score}'[e] \leftarrow \text{score}[e] - \text{score}[\text{bestInEdge}] \]
if \( T=(V,F) \) is a spanning tree then return it
else
\[ C \leftarrow \text{a cycle in } F \]
\[ G' \leftarrow \text{CONTRACT}(G, C) \]
\[ T' \leftarrow \text{MaxSpanningTree}(G', \text{root}, \text{score}') \]
\[ T \leftarrow \text{EXPAND}(T', C) \]
return \( T \)

function CONTRACT(G, C) returns contracted graph

function EXPAND(T, C) returns expanded graph
Chu-Liu-Edmonds algorithm

```plaintext
function MaxSpanningTree(G=(V,E), root, score) returns spanning tree

    F ← []
    T' ← []
    score' ← []
    for each v ∈ V do
        bestInEdge ← argmax_{e=(u,v) ∈ E} score[e]
        F ← F ∪ bestInEdge
    for each e=(u,v) ∈ E do
        score'[e] ← score[e] − score[bestInEdge]
    if T=(V,F) is a spanning tree then return it
    else
        C ← a cycle in F
        G' ← Contract(G, C)
    T' ← MaxSpanningTree(G', root, score')
    T ← Expand(T', C)
    return T

function Contract(G, C) returns contracted graph

function Expand(T, C) returns expanded graph
```

Figure 15.13

The Chu-Liu Edmonds algorithm for finding a maximum spanning tree in a weighted directed graph.

That and flight. The scaled weights using the maximum value entering each node are shown in the graph to the right.

Collapsing the cycle between that and flight to a single node (labelled tf) and recursing with the newly scaled costs is shown in the second row. The greedy selection step in this recursion yields a spanning tree that links root to book, as well as an edge that links book to the contracted node. Expanding the contracted node, we can see that this edge corresponds to the edge from book to flight in the original graph.

On arbitrary directed graphs, this version of the CLE algorithm runs in \( O(mn) \) time, where \( m \) is the number of edges and \( n \) is the number of nodes. Since this particular application of the algorithm begins by constructing a fully connected graph \( m = n^2 \) yielding a running time of \( O(n^3) \).

Gabow et al. (1986) present a more efficient implementation with a running time of \( O(m + n \log n) \).
Chu-Liu-Edmonds algorithm

function `MaxSpanningTree(G=(V,E), root, score)` returns spanning tree

```
F ← []
T’ ← []
score’ ← []
for each v ∈ V do
    bestInEdge ← argmax_e=(u,v)∈E score[e]
    F ← F ∪ bestInEdge
for each e=(u,v) ∈ E do
    score’[e] ← score[e] − score[bestInEdge]
if T=(V,F) is a spanning tree then return it
else
    C ← a cycle in F
    G’ ← Contract(G, C)
    T’ ← MaxSpanningTree(G’, root, score’)
    T ← Expand(T’, C)
return T
```

function `Contract(G, C)` returns contracted graph

function `Expand(T, C)` returns expanded graph
Chu-Liu-Edmonds algorithm

function MaxSpanningTree(G=(V,E), root, score) returns spanning tree

F ← []
T′ ← []
score′ ← []

for each v ∈ V do
    bestInEdge ← argmax_{e=(u,v) ∈ E} score[e]
    F ← F ∪ bestInEdge

for each e=(u,v) ∈ E do
    score′[e] ← score[e] − score[bestInEdge]

if T=(V,F) is a spanning tree then return it
else
    C ← a cycle in F
    G′ ← CONTRACT(G, C)
    T′ ← MaxSpanningTree(G′, root, score′)
    T ← EXPAND(T′, C)
return T

function CONTRACT(G, C) returns contracted graph

function EXPAND(T, C) returns expanded graph
Chu-Liu-Edmonds algorithm

function MaxSpanningTree(G=(V,E), root, score) returns spanning tree

    F ← []
    T' ← []
    score' ← []
    for each v ∈ V do
        bestInEdge ← argmax e=(u,v)∈ E score[e]
        F ← F ∪ bestInEdge
        for each e=(u,v) ∈ E do
            score'[e] ← score[e] − score[bestInEdge]
    if T=(V,F) is a spanning tree then return it
    else
        C ← a cycle in F
        G' ← CONTRACT(G, C)
        T' ← MaxSpanningTree(G', root, score')
        T ← EXPAND(T', C)
        return T

function CONTRACT(G, C) returns contracted graph

function EXPAND(T, C) returns expanded graph
Chu-Liu-Edmonds algorithm

function MaxSpanningTree(G=(V,E), root, score) returns spanning tree

F ← []
T' ← []
score' ← []
for each v ∈ V do
    bestInEdge ← argmax e=(u,v)∈ E score[e]
    F ← F ∪ bestInEdge
    for each e=(u,v) ∈ E do
        score'[e] ← score[e] − score[bestInEdge]
if T=(V,F) is a spanning tree then return it
else
    C ← a cycle in F
    G’ ← CONTRACT(G, C)
    T’ ← MaxSpanningTree(G’, root, score’)
    T ← EXPAND(T’, C)
return T

function CONTRACT(G, C) returns contracted graph

function EXPAND(T, C) returns expanded graph
Chu-Liu-Edmonds algorithm

function MaxSpanningTree(G=(V,E), root, score) returns spanning tree

\[
\begin{align*}
F &\leftarrow [] \\
T' &\leftarrow [] \\
\text{score}' &\leftarrow [] \\
\text{for each } v \in V \text{ do} \\
& \quad \text{bestInEdge} \leftarrow \text{argmax}_{e=(u,v) \in E} \text{score}[e] \\
& \quad F \leftarrow F \cup \text{bestInEdge} \\
& \quad \text{for each } e=(u,v) \in E \text{ do} \\
& \quad \quad \text{score}'[e] \leftarrow \text{score}[e] - \text{score}[\text{bestInEdge}] \\
\text{if } T=(V,F) \text{ is a spanning tree then return it} \\
\text{else} \\
& \quad C \leftarrow \text{a cycle in } F \\
& \quad G' \leftarrow \text{Contract}(G,C) \\
& \quad T' \leftarrow \text{MaxSpanningTree}(G', \text{root}, \text{score}') \\
& \quad T \leftarrow \text{Expand}(T', C) \\
& \quad \text{return } T
\end{align*}
\]

function Contract(G, C) returns contracted graph

function Expand(T, C) returns expanded graph
Chu-Liu-Edmonds algorithm

- Select best incoming edge for each node

![Graph Example]

Given this formulation, we are faced with two problems in training our parser: identifying relevant features and finding the weights used to score those features. The features used to train edge-factored models mirror those used in training transition-based parsers (as shown in Fig. 15.9). This is hardly surprising since in both cases we’re trying to capture information about the relationship between heads and their dependents in the context of a single relation. To summarize this earlier discussion, commonly used features include:

- Wordforms, lemmas, and parts of speech of the headword and its dependent.
- Corresponding features derived from the contexts before, after and between the words.
- Word embeddings.
- The dependency relation itself.
- The direction of the relation (to the right or left).
- The distance from the head to the dependent.

As with transition-based approaches, pre-selected combinations of these features are often used as well.

Given a set of features, our next problem is to learn a set of weights corresponding to each. Unlike many of the learning problems discussed in earlier chapters,
Chu-Liu-Edmonds algorithm

- Subtract its score from all incoming edges
Chu-Liu-Edmonds algorithm

- Subtract its score from all incoming edges

Given this formulation, we are faced with two problems in training our parser: identifying relevant features and finding the weights used to score those features. The features used to train edge-factored models mirror those used in training transition-based parsers (as shown in Fig. 15.9). This is hardly surprising since in both cases we’re trying to capture information about the relationship between heads and their dependents in the context of a single relation. To summarize this earlier discussion, commonly used features include:

- Wordforms, lemmas, and parts of speech of the headword and its dependent.
- Corresponding features derived from the contexts before, after and between the words.
- Word embeddings.
- The dependency relation itself.
- The direction of the relation (to the right or left).
- The distance from the head to the dependent.

As with transition-based approaches, pre-selected combinations of these features are often used as well.
Chu-Liu-Edmonds algorithm

- Contract nodes if there are cycles

Given this formulation, we are faced with two problems in training our parser: identifying relevant features and finding the weights used to score those features.

The features used to train edge-factored models mirror those used in training transition-based parsers (as shown in Fig. 15.9). This is hardly surprising since in both cases we're trying to capture information about the relationship between heads and their dependents in the context of a single relation. To summarize this earlier discussion, commonly used features include:

- Wordforms, lemmas, and parts of speech of the headword and its dependent.
- Corresponding features derived from the contexts before, after and between the words.
- Word embeddings.
- The dependency relation itself.
- The direction of the relation (to the right or left).
- The distance from the head to the dependent.

As with transition-based approaches, pre-selected combinations of these features are often used as well.
Chu-Liu-Edmonds algorithm

- Contract nodes if there are cycles

Given this formulation, we are faced with two problems in training our parser: identifying relevant features and finding the weights used to score those features. The features used to train edge-factored models mirror those used in training transition-based parsers (as shown in Fig. 15.9). This is hardly surprising since in both cases we're trying to capture information about the relationship between heads and their dependents in the context of a single relation. To summarize this earlier discussion, commonly used features include:

- Wordforms, lemmas, and parts of speech of the headword and its dependent.
- Corresponding features derived from the contexts before, after and between the words.
- Word embeddings.
- The dependency relation itself.
- The direction of the relation (to the right or left).
- The distance from the head to the dependent.

As with transition-based approaches, pre-selected combinations of these features are often used as well. Given a set of features, our next problem is to learn a set of weights corresponding to each. Unlike many of the learning problems discussed in earlier chapters,
Chu-Liu-Edmonds algorithm

- Recursively compute MST

The Chu-Liu-Edmonds algorithm is a graph-based algorithm for finding the minimum spanning tree (MST) of a weighted directed graph. It is particularly useful in natural language processing for tasks such as dependency parsing.

Given a set of features, our next problem is to learn a set of weights corresponding to each. Unlike many of the learning problems discussed in earlier chapters, the weights in edge-factored models are learned directly from the features, rather than being estimated from training data.

Commonly used features include:
- Wordforms, lemmas, and parts of speech of the headword and its dependent.
- Corresponding features derived from the contexts before, after and between the words.
- Word embeddings.
- The dependency relation itself.
- The direction of the relation (to the right or left).
- The distance from the head to the dependent.

As with transition-based approaches, pre-selected combinations of these features are often used as well.
Chu-Liu-Edmonds algorithm

- Expand contracted nodes

Figure 15.14 Chu-Liu-Edmonds graph-based example for Book that flight

Or more succinctly.

\[
\text{score}(S, e) = w \cdot f
\]

Given this formulation, we are faced with two problems in training our parser:

- identifying relevant features and finding the weights used to score those features.
- The features used to train edge-factored models mirror those used in training transition-based parsers (as shown in Fig. 15.9). This is hardly surprising since in both cases we're trying to capture information about the relationship between heads and their dependents in the context of a single relation. To summarize this earlier discussion, commonly used features include:
  - Wordforms, lemmas, and parts of speech of the headword and its dependent.
  - Corresponding features derived from the contexts before, after and between the words.
  - Word embeddings.
  - The dependency relation itself.
  - The direction of the relation (to the right or left).
  - The distance from the head to the dependent.
- As with transition-based approaches, pre-selected combinations of these features are often used as well.

Given a set of features, our next problem is to learn a set of weights corresponding to each. Unlike many of the learning problems discussed in earlier chapters,
Chu-Liu-Edmonds algorithm

function `MaxSpanningTree(G=(V,E), root, score)` returns spanning tree

\[
\begin{align*}
F &\leftarrow [] \\
T' &\leftarrow [] \\
\text{score}' &\leftarrow [] \\
\text{for each } v \in V \text{ do} & \\
\quad \text{bestInEdge} &\leftarrow \text{argmax}_{e=(u,v) \in E} \text{ score}[e] \\
\quad F &\leftarrow F \cup \text{bestInEdge} \\
\text{for each } e=(u,v) \in E \text{ do} & \\
\quad \text{score}'[e] &\leftarrow \text{score}[e] - \text{score}[\text{bestInEdge}] \\
\text{if } T=(V,F) \text{ is a spanning tree then return it} & \\
\text{else} & \\
\quad C &\leftarrow \text{a cycle in } F \\
\quad G' &\leftarrow \text{Contract}(G, C) \\
\quad T' &\leftarrow \text{MaxSpanningTree}(G', \text{root, score'}) \\
\quad T &\leftarrow \text{Expand}(T', C) \\
\text{return } T
\end{align*}
\]

function `Contract(G, C)` returns contracted graph

function `Expand(T, C)` returns expanded graph
Chu-Liu-Edmonds algorithm

function MaxSpanningTree($G=(V,E)$, root, score) returns spanning tree

$$F \leftarrow \[]$$

$$T' \leftarrow \[]$$

$$score' \leftarrow \[]$$

for each $v \in V$ do

$$bestInEdge \leftarrow \text{argmax}_{(u,v) \in E} score[e]$$

$$F \leftarrow F \cup bestInEdge$$

for each $e=(u,v) \in E$ do

$$score'[e] \leftarrow score[e] - score[bestInEdge]$$

if $T=(V,F)$ is a spanning tree then return it
else

$C \leftarrow$ a cycle in $F$

$G' \leftarrow \text{contract}(G, C)$

$T' \leftarrow \text{MaxSpanningTree}(G', root, score')$

$T \leftarrow \text{expand}(T', C)$

return $T$

function Contract($G, C$) returns contracted graph

function Expand($T, C$) returns expanded graph

runtime? naive: $O(n^3)$
Chu-Liu-Edmonds algorithm

function **MaxSpanningTree**\((G=(V,E), root, score)\) returns spanning tree

\[
F \leftarrow [] \\
T' \leftarrow [] \\
score' \leftarrow [] \\
\text{for each } v \in V \text{ do} \\
\quad bestInEdge \leftarrow \arg\max_{(u,v) \in E} score[e] \\
\quad F \leftarrow F \cup bestInEdge \\
\text{for each } e=(u,v) \in E \text{ do} \\
\quad score'[e] \leftarrow score[e] - score[bestInEdge] \\
\text{if } T=(V,F) \text{ is a spanning tree then return it} \\
\text{else} \\
\quad C \leftarrow \text{a cycle in } F \\
\quad G' \leftarrow \text{Contract}(G, C) \\
\quad T' \leftarrow \text{MaxSpanningTree}(G', root, score') \\
\quad T \leftarrow \text{Expand}(T', C) \\
\text{return } T
\]

function **Contract**\((G, C)\) returns contracted graph

function **Expand**\((T, C)\) returns expanded graph

runtime?  
naive: \(O(n^3)\)  
fancy: \(O(n^2 + n\log n)\)
Chu-Liu-Edmonds algorithm

function MaxSpanningTree(G=(V,E), root, score) returns spanning tree

F ← []
T' ← []
score' ← []

for each v ∈ V do
    bestInEdge ← argmax_e=(u,v)∈ E score[e]
    F ← F ∪ bestInEdge

for each e=(u,v) ∈ E do
    score'[e] ← score[e] − score[bestInEdge]

if T=(V,F) is a spanning tree then return it
else
    C ← a cycle in F
    G' ← CONTRACT(G, C)
    T' ← MaxSpanningTree(G', root, score')
    T ← EXPAND(T', C)
return T

function CONTRACT(G, C) returns contracted graph

function EXPAND(T, C) returns expanded graph

runtime? naive: O(n^3)
fancy: O(n^2 + nlogn)

what about labeled parsing?
Graph-based dependency parsing

- **Edge-factored** (or arc-factored) approaches:

  - Score of a tree decomposes as sum of edge scores:
    \[
    \psi(y, w; \theta) = \sum_{i \xrightarrow{\ell} j \in y} \psi(i \xrightarrow{\ell} j, w, \theta)
    \]
  
  - Start with a fully-connected directed graph
  
  - How to infer the highest scoring tree?

- Find a **maximum directed spanning tree**: Chu and Liu (1965) and Edmonds (1967) algorithm
Graph-based dependency parsing

- **Edge-factored** (or **arc-factored**) approaches:
  - Score of a tree decomposes as sum of edge scores:
    \[
    \psi(y, w; \theta) = \sum_{i \xrightarrow{} j \in y} \psi(i \xrightarrow{} j, w, \theta)
    \]
  - Start with a fully-connected directed graph
  - How to infer the highest scoring tree?
  - Find a **maximum directed spanning tree**: Chu and Liu (1965) and Edmonds (1967) algorithm
  - For projective trees: **Eisner’s algorithm** [Eisner 1996]
Graph-based dependency parsing

- **Edge-factored** (or arc-factored) approaches: score of a tree decomposes as sum of edge scores.

\[
\psi(y, w; \theta) = \sum_{i \rightarrow j \in y} \psi(i \rightarrow j, w, \theta)
\]
Graph-based dependency parsing

- **Edge-factored** (or **arc-factored**) approaches: score of a tree decomposes as sum of edge scores.

\[ \psi(y, w; \theta) = \sum_{i \rightarrow j \in y} \psi(i \rightarrow j, w, \theta) \]

- Can also define **higher-order** models: score decomposes as a sum of scores of local subgraphs.

First order

\[ h \rightarrow m \]

Second order

\[ h \rightarrow s \rightarrow m \quad g \rightarrow h \rightarrow m \]

Third order

\[ g \rightarrow h \rightarrow s \rightarrow m \quad h \rightarrow t \rightarrow s \rightarrow m \]
Graph-based dependency parsing

- **Edge-factored** (or **arc-factored**) approaches: score of a tree decomposes as sum of edge scores.

\[
\psi(y, w; \theta) = \sum_{i \rightarrow j \in y} \psi(i \rightarrow j, w, \theta)
\]

- Can also define **higher-order** models: score decomposes as a sum of scores of local subgraphs.

Graph-based dependency parsing

- **Edge-factored** (or **arc-factored**) approaches: score of a tree decomposes as sum of edge scores.

\[ \psi(y, w; \theta) = \sum_{i \rightarrow j \in y} \psi(i \rightarrow j, w, \theta) \]

- Can also define **higher-order** models: score decomposes as a sum of scores of local subgraphs.

  First order

  \[ h \rightarrow m \]

  Second order

  \[ h \rightarrow s \rightarrow m \quad g \rightarrow h \rightarrow m \]

  Third order

  \[ g \rightarrow h \rightarrow s \rightarrow m \quad h \rightarrow t \rightarrow s \rightarrow m \]


- Second order parsing is NP-hard for *non-projective* dependency graphs [McDonald and Pereira, 2006]!
Graph-based dependency parsing

- **Edge-factored** (or **arc-factored**) approaches: score of a tree decomposes as sum of edge scores.

\[
\psi(y, w; \theta) = \sum_{i \rightarrow j \in y} \psi(i \rightarrow j, w, \theta)
\]

First order

- **Second order**

- **Third order**

Figure 11.6: Feature templates for higher-order dependency parsing
Graph-based dependency parsing

- **Edge-factored** (or **arc-factored**) approaches: score of a tree decomposes as sum of edge scores.
  
  \[ \psi(y, w; \theta) = \sum_{i \rightarrow j \in y} \psi(i \rightarrow j, w, \theta) \]

  First order

- How to parameterize \( \psi \) ?
Classic graph-based parsing features

- Word forms, lemmas, parts of speech of the head word and its dependent
- Corresponding features derived from the contexts before, after, between words
- Word embeddings
- Dependency relation
- Direction of the relation (right or left)
- Distance from the head to the dependent
- Combinations of all of the above
Graph-based neural network parsers

per-token features

word embeddings

neural network

Janet will back the bill
Graph-based neural network parsers

per-token features

neural network

word embeddings

pos embeddings

Janet/NNP  will/MD  back/VB  the/DT  bill/NN
Graph-based neural network parsers

edge features

per-token features

word embeddings

pos embeddings

neural network

Janet/NNP
will/MD
back/VB
the/DT
bill/NN
Graph-based neural network parsers

edge features

per-token features

neural network

word embeddings

pos embeddings
Graph-based neural network parsers
Graph-based neural network parsers

- Concat + feed-forward [Kiperwasser and Goldberg, 2016]
Graph-based neural network parsers

- Concat + feed-forward [Kiperwasser and Goldberg, 2016]
Graph-based neural network parsers

- Concat + feed-forward [Kiperwasser and Goldberg, 2016]
Graph-based neural network parsers

- Concat + feed-forward [Kiperwasser and Goldberg, 2016]
- Hinge loss: \[ \max \left( 0, 1 - \max_{y' \neq y} \sum_{(h, m) \in y'} \text{MLP}(v_h \circ v_m) + \sum_{(h, m) \in y} \text{MLP}(v_h \circ v_m) \right) \]

- Arc-factored parsing decomposes the score of a tree based approach presented in McDonald et al. (2005).
- Loss augmented inference is used to quickly parse longer sentences. In initial experiments, the number of parts stays small, enabling faster parsing. With the arc-factored parser: the same BiLSTM encoder with the unlabeled parser. The BiLSTM encoder responsible for producing labels is trained on the gold trees. The advantage of training the MLP on the arc representations is that the gradients of the entire network (including to the BiLSTM encoder) can be reduced to matrix-vector multiplications from the input to the MLP. The figure depicts a single-layer BiLSTM, while in practice we use two layers. When parsing a sentence, the BiLSTM encoding of the words at the arc's end points (the colors of the arcs correspond to colors of the words) is fed into a BiLSTM encoder and word embeddings. Using this approach, the number of parts stays small, enabling fast parsing.

The arc-factored model requires the scoring of different pairs. We evaluated our parsing model on English and Chinese data. For comparison purposes we follow the different pairs.

The intuition behind loss augmented inference is to update the parameters of the BiLSTM feature encoder to be good at predicting arc-labels significantly improves the parser's unlabeled accuracy. The forward first-order parser, trained with a margin-loss, is the first and second half of the syntax tree.

\[ M(x_{\text{the}}, x_{\text{brown}}, x_{\text{fox}}, x_{\text{jumped}}, x_{\ast}) = \arg \max (\text{LSTM}) \]

\[ \text{concat} \hspace{1cm} \text{concat} \hspace{1cm} \text{concat} \hspace{1cm} \text{concat} \hspace{1cm} \text{concat} \]

\[ \begin{align*}
& \text{MLP} \\
& \text{V_{the}} \\
& \text{concat} \\
& \text{MLP} \\
& \text{V_{brown}} \\
& \text{concat} \\
& \text{MLP} \\
& \text{V_{fox}} \\
& \text{concat} \\
& \text{MLP} \\
& \text{V_{jumped}} \\
& \text{concat} \\
& \text{MLP} \\
& \text{V_{*}}
\end{align*} \]

**concat head, dependent as input to MLP**

stacked LSTM token encoder
Graph-based neural network parsers

- Concat + feed-forward \cite{Kiperwasser and Goldberg, 2016}

- Hinge loss: \( \max(0, 1 - \max_{y' \neq y} \sum_{(h, m) \in y'} MLP(v_h \circ v_m) + \sum_{(h, m) \in y} MLP(v_h \circ v_m)) \)

- w/ loss augmented inference: \( \max(0, 1 + \text{score}(x, y) - \max_{y' \neq y} \sum_{\text{part} \in y'} (\text{score}_{\text{local}}(x, \text{part}) + \mathbb{1}_{\text{part} \neq y})) \)

![Diagram of graph-based neural network parsers]

- MLP
- Concat head, dependent as input to MLP
- Stacked LSTM
- Token encoder
Graph-based neural network parsers
Graph-based neural network parsers

- Biaffine classifier [Dozat and Manning, 2017]
Graph-based neural network parsers

- Biaffine classifier [Dozat and Manning, 2017]
Graph-based neural network parsers

- Biaffine classifier [Dozat and Manning, 2017]
Graph-based neural network parsers

- Biaffine classifier [Dozat and Manning, 2017]

\[
\begin{align*}
    s_i &= Wr_i + b \\
    s_i^{(\text{arc})} &= (RU^{(1)}) r_i + (Ru^{(2)})
\end{align*}
\]

Fixed-class affine classifier
Variable-class biaffine classifier

\[
H^{(\text{arc-dep})} \oplus 1 \quad U(\text{arc}) \quad H^{(\text{arc-head})} \quad S(\text{arc})
\]

\[
\text{MLP: } h_i^{(\text{arc-dep})}, h_i^{(\text{arc-head})}
\]

MLP\_head  MLP\_dep

\[
\text{BiLSTM: } r_i
\]

stacked LSTM token encoder
Graph-based neural network parsers

- Biaffine classifier [Dozat and Manning, 2017]

- Locally normalized log loss.

\[
s_i = W r_i + b
\]

\[
s_i^{(arc)} = \left( R U^{(1)} \right) r_i + \left( R u^{(2)} \right)
\]

- Fixed-class affine classifier
- Variable-class biaffine classifier

**MLP:** \(h_i^{(arc-dep)}, h_i^{(arc-head)}\)

**BiLSTM:** \(r_i\)

**Embeddings:** \(x_i\)

\(H^{(arc-dep)} \oplus 1\)

\(U^{(arc)}\)

\(H^{(arc-head)}\)

\(S^{(arc)}\)

\(T\)

**biaffine classifier**

**MLP\_head**  **MLP\_dep**

**stacked LSTM**

**token encoder**
Graph-based dependency parsing

- **Edge-factored** (or **arc-factored**) approaches: score of a tree decomposes as sum of edge scores.

\[ \psi(y, w; \theta) = \sum_{i \rightarrow j \in y} \psi(i \rightarrow j, w, \theta) \]  

- First order

- How to parameterize \( \psi \)?
Graph-based dependency parsing

- **Edge-factored** (or **arc-factored**) approaches: score of a tree decomposes as sum of edge scores.

\[
\psi(y, w; \theta) = \sum_{i \rightarrow j \in y} \psi(i \rightarrow j, w, \theta)
\]

First order

- How to parameterize \(\psi\)?
- How to learn \(\theta\)?
Graph-based dependency parsing

- **Edge-factored** (or **arc-factored**) approaches: score of a tree decomposes as sum of edge scores.

\[
\psi(y, w; \theta) = \sum_{i \rightarrow j \in y} \psi(i \rightarrow j, w, \theta)
\]

- First order

- How to parameterize \( \psi \)?

- How to learn \( \theta \)?

  - Locally normalized: predict each head and its label, softmax, log loss.
Graph-based dependency parsing

- **Edge-factored** (or **arc-factored**) approaches: score of a tree decomposes as sum of edge scores.

\[
\psi(y, w; \theta) = \sum_{i \rightarrow j \in y} \psi(i \rightarrow j, w, \theta) \]

- How to parameterize \(\psi\)?
- How to learn \(\theta\)?
  - Locally normalized: predict each head and its label, softmax, log loss.
  - Hinge loss: Find the best scoring tree, and penalize edges not in the gold tree (with a margin).
**Graph-based dependency parsing**

- **Edge-factored** (or arc-factored) approaches: score of a tree decomposes as sum of edge scores.
  \[
  \psi(y, w; \theta) = \sum_{i \rightarrow j \in y} \psi(i \rightarrow j, w, \theta)
  \]
  First order

- How to parameterize \( \psi \) ?
- How to learn \( \theta \) ?
  - Locally normalized: predict each head and its label, softmax, log loss.
  - Hinge loss: Find the best scoring tree, and penalize edges not in the gold tree (with a margin).
  - Globally normalized CRF: can compute marginals/partition function using a variant of Kirchhoff's Matrix-Tree Theorem [Tutte, 1984; Koo et al. 2007].
Graph-based vs. transition-based parsing?
Graph-based vs. transition-based parsing?

- **Transition-based**
  - Fast
  - Greedy / local inference
  - Maybe closer to humans?

![Diagram of a basic transition-based parser](image)

In the standard approach to transition-based parsing, the operators used to produce new configurations are surprisingly simple and correspond to the intuitive actions one might take in creating a dependency tree by examining the words in a single pass over the input from left to right (Covington, 2001):

- Assign the current word as the head of some previously seen word,
- Assign some previously seen word as the head of the current word,
- Or postpone doing anything with the current word, adding it to a store for later processing.

To make these actions more precise, we'll create three transition operators that will operate on the top two elements of the stack:

- **LEFT ARC**: Assert a head-dependent relation between the word at the top of the stack and the word directly beneath it; remove the lower word from the stack.
- **RIGHT ARC**: Assert a head-dependent relation between the second word on the stack and the word at the top; remove the word at the top of the stack;
- **SHIFT**: Remove the word from the front of the input buffer and push it onto the stack.

This particular set of operators implements what is known as the arc standard approach to transition-based parsing (Covington 2001, Nivre 2003). There are two notable characteristics to this approach: the transition operators only assert relations between elements at the top of the stack, and once an element has been assigned its head it is removed from the stack and is not available for further processing.

As we'll see, there are alternative transition systems which demonstrate different parsing behaviors, but the arc standard approach is quite effective and is simple to implement.
Graph-based vs. transition-based parsing?

- **Transition-based**
  - Fast
  - Greedy / local inference
  - Maybe closer to humans?

- **Graph-based**
  - Slow
  - Exact inference
  - More accurate
Graph-based vs. transition-based parsing?

Dependency parsing: accuracy vs. speed

Accuracy (UAS)

Speed (sentences/sec)
Graph-based vs. transition-based parsing?

Dependency parsing: accuracy vs. speed

Accuracy (UAS)

Speed (sentences/sec)

Graph-based

TurboParser [Martins et al. 2010]

[Zhang & McDonald 2014]
Graph-based vs. transition-based parsing?

Dependency parsing: accuracy vs. speed

Accuracy (UAS)

Speed (sentences/sec)

Graph-based

Transition-based

MaltParser [Nivre et al. 2009]

TurboParser [Martins et al. 2010]

[Zhang & McDonald 2014]
Graph-based vs. transition-based parsing?

Dependency parsing: accuracy vs. speed

<table>
<thead>
<tr>
<th>Speed (sentences/sec)</th>
<th>Accuracy (UAS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph-based</td>
<td>93</td>
</tr>
<tr>
<td>Transition-based</td>
<td>91</td>
</tr>
</tbody>
</table>

[Structured prediction cascades]

[Rush & Petrov 2012; Weiss & Taskar 2010]
[MaltParser [Nivre et al. 2009]]
[TurboParser [Martins et al. 2010]]
[Zhang & McDonald 2014]
Graph-based vs. transition-based parsing?

<table>
<thead>
<tr>
<th>Speed (sentences/sec)</th>
<th>Accuracy (UAS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph-based</td>
<td></td>
</tr>
<tr>
<td>Transition-based</td>
<td></td>
</tr>
<tr>
<td>Structured prediction cascades</td>
<td></td>
</tr>
<tr>
<td>FF neural nets</td>
<td></td>
</tr>
<tr>
<td>[Chen &amp; Manning 2014]</td>
<td></td>
</tr>
<tr>
<td>[Rush &amp; Petrov 2012; Weiss &amp; Taskar 2010]</td>
<td></td>
</tr>
<tr>
<td>MaltParser [Nivre et al. 2009]</td>
<td></td>
</tr>
<tr>
<td>TurboParser [Martins et al. 2010]</td>
<td></td>
</tr>
<tr>
<td>[Zhang &amp; McDonald 2014]</td>
<td></td>
</tr>
</tbody>
</table>
Graph-based vs. transition-based parsing?

Dependency parsing: accuracy vs. speed

- Transition-based
  - FF neural nets
  - Structured prediction cascades
  - transition-based w/ dynamic feature selection
  - TurboParser [Martins et al. 2010]
  - Weiss & Taskar 2010

- Graph-based
  - [Strubell et al. 2015]
  - [Chen & Manning 2014]
  - [Rush & Petrov 2012]
  - MaltParser [Nivre et al. 2009]
  - [Zhang & McDonald 2014]
Graph-based vs. transition-based parsing?

Dependency parsing: accuracy vs. speed

- **Graph-based**
  - Accuracy (UAS): 94, 95, 96
  - Speed (sentences/sec)

- **Transition-based**
  - Accuracy (UAS): 93
  - Speed (sentences/sec)

- **Structured prediction cascades**
  - Accuracy (UAS): 89, 90, 91, 92
  - Speed (sentences/sec)

- **FF neural nets**
  - Accuracy (UAS): 92
  - Speed (sentences/sec)

- **transition-based w/ dynamic feature selection**
Graph-based vs. transition-based parsing?

Dependency parsing: accuracy vs. speed

- Transition-based
- FF neural nets
- Structured prediction cascades
- Graph-based

- Accuracy (UAS)
- Speed (sentences/sec)

- Transition-based w/ dynamic feature selection

Stacked LSTMs [Dozat and Manning 2017]*
Graph-based vs. transition-based parsing?

Dependency parsing: accuracy vs. speed

- Graph-based
- Transition-based
- FF neural nets
- Structured prediction cascades
- Transition-based w/ dynamic feature selection
- Stacked LSTMs [Dozat and Manning 2017]*

Speed (sentences/sec)
Accuracy (UAS)

*on GPU!!
Graph-based vs. transition-based parsing?

Dependency parsing: accuracy vs. speed

- Graph-based
- Transition-based
  - with dynamic feature selection
  - FF neural nets
  - Structured prediction cascades
  - Stacked LSTMs [Dozat and Manning 2017]*

*on GPU!!
Announcements

- No recitation on Friday (Tartan Community Day).