Graph-based Dependency Parsing
Chu-Liu-Edmonds and Camerini (k-best)

Swabha Swayamdipta    Sam Thomson

Carnegie Mellon University

November 13, 2014
I ate the fish with a fork.

TurboParser output from
http://demo.ark.cs.cmu.edu/parse?sentence=I%20ate%20the%20fish%20with%20a%20fork.
A parse is an arborescence (aka directed rooted tree):

- Directed [Labeled] Graph
- Acyclic
- Single Root
- Connected and Spanning: \( \exists \) directed path from root to every other word
Arc-Factored Model

Every possible labeled directed edge $e$ between every pair of nodes gets a score, $\text{score}(e)$. 
Arc-Factored Model

Every possible labeled directed edge $e$ between every pair of nodes gets a score, $\text{score}(e)$.

$$G = \langle V, E \rangle =$$

![Diagram of a graph with labels and scores]

$(O(n^2)$ edges)

Example from Non-projective Dependency Parsing using Spanning Tree Algorithms McDonald et al., EMNLP ’05
Arc-Factored Model

Best parse is:

\[ A^{(1)} = \arg \max_{A \subseteq G} \sum_{e \in A} \text{score}(e) \]

Example from *Non-projective Dependency Parsing using Spanning Tree Algorithms* McDonald et al., EMNLP '05
Arc-Factored Model

Best parse is:

\[ A^{(1)} = \arg \max_{A \subseteq G} \sum_{e \in A} \text{score}(e) \quad \text{s.t. } A \text{ an arborescence} \]

Example from *Non-projective Dependency Parsing using Spanning Tree Algorithms* McDonald et al., EMNLP '05
Arc-Factored Model

Best parse is:

\[ A^{(1)} = \arg \max_{A \subseteq G} \sum_{e \in A} \text{score}(e) \quad \text{s.t.} \quad A \text{ an arborescence} \]

The Chu-Liu-Edmonds algorithm finds this argmax.

Example from Non-projective Dependency Parsing using Spanning Tree Algorithms McDonald et al., EMNLP '05
Some parses are **projective**: edges don’t cross

Most English sentences are projective, but non-projectivity is common in other languages (e.g. Czech, Hindi)

Non-projective sentence in English:

```
root  John  saw  a  dog  yesterday  which  was  a  Yorkshire  Terrier
```

and Czech:

```
root  O  to  nové  většinou  nemá  ani  zájem  a  taky  na  to  většinou  nemá  peníze
```

*He is mostly not even interested in the new things and in most cases, he has no money for it either.*

Examples from *Non-projective Dependency Parsing using Spanning Tree Algorithms* McDonald et al., EMNLP ’05
Dependency Parsing Approaches

- Chart (Eisner, CKY)
  - *Only* produces projective parses
  - $O(n^3)$
Dependency Parsing Approaches

- Chart (Eisner, CKY)
  - Only produces projective parses
  - $O(n^3)$

- Shift-reduce
  - “Pseudo-projective” trick can capture some non-projectivity
  - $O(n)$ (fast!), but inexact
Dependency Parsing Approaches

- Chart (Eisner, CKY)
  - *Only* produces projective parses
  - $O(n^3)$
- Shift-reduce
  - “Pseudo-projective” trick can capture some non-projectivity
  - $O(n)$ (*fast!*), but inexact
- Graph-based (MST)
  - Can produce projective *and* non-projective parses
  - $O(n^2)$ for arc-factored
Chu and Liu ’65, On the Shortest Arborescence of a Directed Graph, Science Sinica

Edmonds ’67, Optimum Branchings, JRNBS
Chu-Liu-Edmonds - Intuition

Every non-ROOT node needs exactly 1 incoming edge
Every non-ROOT node needs exactly 1 incoming edge
In fact, every connected component needs exactly 1 incoming edge
Every non-ROOT node needs exactly 1 incoming edge.
In fact, every connected component needs exactly 1 incoming edge.

▶ Greedily pick an incoming edge for each node.
Every non-ROOT node needs exactly 1 incoming edge
In fact, every connected component needs exactly 1 incoming edge
  ▶ Greedily pick an incoming edge for each node.
  ▶ If this forms an arborescence, great!
Every non-ROOT node needs exactly 1 incoming edge
In fact, every connected component needs exactly 1 incoming edge

▶ Greedily pick an incoming edge for each node.
▶ If this forms an arborescence, great!
▶ Otherwise, it will contain a cycle $C$. 
Every non-ROOT node needs exactly 1 incoming edge
In fact, every connected component needs exactly 1 incoming edge

- Greedily pick an incoming edge for each node.
- If this forms an arborescence, great!
- Otherwise, it will contain a cycle $C$.
- Arborescences can’t have cycles, so we can’t keep every edge in $C$. One edge in $C$ must get kicked out.
Every non-ROOT node needs exactly 1 incoming edge
In fact, every connected component needs exactly 1 incoming edge
  ▶ Greedily pick an incoming edge for each node.
  ▶ If this forms an arborescence, great!
  ▶ Otherwise, it will contain a cycle $C$.
  ▶ Arborescences can’t have cycles, so we can’t keep every edge in $C$. One edge in $C$ must get kicked out.
  ▶ $C$ also needs an incoming edge.
Chu-Liu-Edmonds - Intuition

Every non-ROOT node needs exactly 1 incoming edge
In fact, every connected component needs exactly 1 incoming edge

▶ Greedily pick an incoming edge for each node.
▶ If this forms an arborescence, great!
▶ Otherwise, it will contain a cycle $C$.
▶ Arborescences can’t have cycles, so we can’t keep every edge in $C$. One edge in $C$ must get kicked out.
▶ $C$ also needs an incoming edge.
▶ Choosing an incoming edge for $C$ determines which edge to kick out
Chu-Liu-Edmonds

Consists of two stages:

- Contracting
- Expanding
For each non-ROOT node \( v \), set \( \text{bestInEdge}[v] \) to be its highest scoring incoming edge.

If a cycle \( C \) is ever formed:
- contract the nodes in \( C \) into a new node \( v_C \)
- edges incoming to any node in \( C \) now get destination \( v_C \)
- edges outgoing from any node in \( C \) now get source \( v_C \)
- For each node \( u \) in \( C \), and for each edge \( e \) incoming to \( u \) from outside of \( C \):
  - add \( \text{bestInEdge}[u] \) to \( \text{kicksOut}[e] \), and
  - set the score of \( e \) to be \( \text{score}[e] - \text{score}[\text{bestInEdge}[u]] \).

Repeat until every non-ROOT node has an incoming edge and no cycles are formed.
An Example - Contracting Stage

<table>
<thead>
<tr>
<th>bestInEdge</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
</tr>
<tr>
<td>V2</td>
</tr>
<tr>
<td>V3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>kicksOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>g</td>
</tr>
<tr>
<td>h</td>
</tr>
<tr>
<td>i</td>
</tr>
</tbody>
</table>
An Example - Contracting Stage

**Best In Edge**

<table>
<thead>
<tr>
<th>V1</th>
<th>V2</th>
<th>V3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>g</td>
</tr>
</tbody>
</table>

**Kicks Out**

<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>g</td>
</tr>
<tr>
<td>h</td>
</tr>
<tr>
<td>i</td>
</tr>
</tbody>
</table>
An Example - Contracting Stage

<table>
<thead>
<tr>
<th></th>
<th>bestInEdge</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>g</td>
</tr>
<tr>
<td>V2</td>
<td>d</td>
</tr>
<tr>
<td>V3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>kicksOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
</tr>
</tbody>
</table>
An Example - Contracting Stage

Table 1: bestInEdge

<table>
<thead>
<tr>
<th>V1</th>
<th>V2</th>
<th>V3</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>d</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: kicksOut

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
An Example - Contracting Stage

<table>
<thead>
<tr>
<th></th>
<th>bestInEdge</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>g</td>
</tr>
<tr>
<td>V2</td>
<td>d</td>
</tr>
<tr>
<td>V3</td>
<td></td>
</tr>
<tr>
<td>V4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>kicksOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>g</td>
</tr>
<tr>
<td>b</td>
<td>d</td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
</tr>
</tbody>
</table>
An Example - Contracting Stage

```
<table>
<thead>
<tr>
<th></th>
<th>bestInEdge</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>g</td>
</tr>
<tr>
<td>V2</td>
<td>d</td>
</tr>
<tr>
<td>V3</td>
<td>f</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>kicksOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>g</td>
</tr>
<tr>
<td>b</td>
<td>d</td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td></td>
</tr>
</tbody>
</table>
```
An Example - Contracting Stage

**bestInEdge**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>g</td>
</tr>
<tr>
<td>V2</td>
<td>d</td>
</tr>
<tr>
<td>V3</td>
<td>f</td>
</tr>
<tr>
<td>V4</td>
<td>h</td>
</tr>
</tbody>
</table>

**kicksOut**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>g</td>
</tr>
<tr>
<td>b</td>
<td>d</td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>g</td>
</tr>
<tr>
<td>i</td>
<td>d</td>
</tr>
</tbody>
</table>
An Example - Contracting Stage

<table>
<thead>
<tr>
<th></th>
<th>bestInEdge</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>g</td>
</tr>
<tr>
<td>V2</td>
<td>d</td>
</tr>
<tr>
<td>V3</td>
<td>f</td>
</tr>
<tr>
<td>V4</td>
<td>h</td>
</tr>
<tr>
<td>V5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>kicksOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>g, h</td>
</tr>
<tr>
<td>b</td>
<td>d, h</td>
</tr>
<tr>
<td>c</td>
<td>f</td>
</tr>
<tr>
<td>d</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>g</td>
</tr>
<tr>
<td>i</td>
<td>d</td>
</tr>
</tbody>
</table>
An Example - Contracting Stage

<table>
<thead>
<tr>
<th>bestInEdge</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
</tr>
<tr>
<td>V2</td>
</tr>
<tr>
<td>V3</td>
</tr>
<tr>
<td>V4</td>
</tr>
<tr>
<td>V5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>kicksOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>g</td>
</tr>
<tr>
<td>h</td>
</tr>
<tr>
<td>i</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
An Example - Contracting Stage
After the contracting stage, every contracted node will have exactly one \texttt{bestInEdge}. This edge will kick out one edge inside the contracted node, breaking the cycle.

- Go through each \texttt{bestInEdge} $e$ in the reverse order that we added them
- lock down $e$, and remove every edge in \texttt{kicksOut}(e) from \texttt{bestInEdge}. 

\textbf{Chu-Liu-Edmonds - Expanding Stage}
An Example - Expanding Stage

- \( \text{ROOT} \):
  - \( b : -9 \)
  - \( a : -4 \)
  - \( c : -4 \)

- \( V5 \): kicksOut:
  - \( g, h \)
  - \( d, h \)
  - \( f \)
  - \( d \)

- \( \text{bestInEdge} \):
<table>
<thead>
<tr>
<th>V1</th>
<th>V2</th>
<th>V3</th>
<th>V4</th>
<th>V5</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>d</td>
<td>f</td>
<td>h</td>
<td>a</td>
</tr>
</tbody>
</table>
An Example - Expanding Stage

<table>
<thead>
<tr>
<th>V1</th>
<th>bestInEdge</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a, g</td>
</tr>
<tr>
<td>V2</td>
<td>d</td>
</tr>
<tr>
<td>V3</td>
<td>f</td>
</tr>
<tr>
<td>V4</td>
<td>a, h</td>
</tr>
<tr>
<td>V5</td>
<td>a</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>V1</th>
<th>kicksOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>g, h</td>
</tr>
<tr>
<td>b</td>
<td>d, h</td>
</tr>
<tr>
<td>c</td>
<td>f</td>
</tr>
<tr>
<td>d</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>g</td>
</tr>
<tr>
<td>i</td>
<td>d</td>
</tr>
</tbody>
</table>
An Example - Expanding Stage

**Table:**

<table>
<thead>
<tr>
<th>bestInEdge</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
</tr>
<tr>
<td>V2</td>
</tr>
<tr>
<td>V3</td>
</tr>
<tr>
<td>V4</td>
</tr>
<tr>
<td>V5</td>
</tr>
</tbody>
</table>

**List:**

<table>
<thead>
<tr>
<th>kicksOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>e</td>
</tr>
<tr>
<td>f</td>
</tr>
<tr>
<td>g</td>
</tr>
<tr>
<td>h</td>
</tr>
<tr>
<td>i</td>
</tr>
</tbody>
</table>
An Example - Expanding Stage

**Table: bestInEdge**

<table>
<thead>
<tr>
<th>Vertex</th>
<th>bestInEdge</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>a, g</td>
</tr>
<tr>
<td>V2</td>
<td>d</td>
</tr>
<tr>
<td>V3</td>
<td>f</td>
</tr>
<tr>
<td>V4</td>
<td>a, h</td>
</tr>
<tr>
<td>V5</td>
<td>a</td>
</tr>
</tbody>
</table>

**Table: kicksOut**

<table>
<thead>
<tr>
<th>Vertex</th>
<th>kicksOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>g, h</td>
</tr>
<tr>
<td>b</td>
<td>d, h</td>
</tr>
<tr>
<td>c</td>
<td>f</td>
</tr>
<tr>
<td>d</td>
<td>f</td>
</tr>
<tr>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td>f</td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>d</td>
</tr>
</tbody>
</table>
An Example - Expanding Stage

![Diagram of an expanding stage graph with vertices and edges labeled with weights.]

<table>
<thead>
<tr>
<th></th>
<th>bestInEdge</th>
<th></th>
<th>kicksOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>a, g</td>
<td>V2</td>
<td>d, h</td>
</tr>
<tr>
<td>V3</td>
<td>a, h</td>
<td>V4</td>
<td>f</td>
</tr>
<tr>
<td>V5</td>
<td>a</td>
<td></td>
<td>f</td>
</tr>
</tbody>
</table>
An Example - Expanding Stage

```
<table>
<thead>
<tr>
<th></th>
<th>bestInEdge</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>a, g, h</td>
</tr>
<tr>
<td>V2</td>
<td>d</td>
</tr>
<tr>
<td>V3</td>
<td>f</td>
</tr>
<tr>
<td>V4</td>
<td>a, h</td>
</tr>
<tr>
<td>V5</td>
<td>a</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th></th>
<th>kicksOut</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>g, h</td>
</tr>
<tr>
<td>b</td>
<td>d, h</td>
</tr>
<tr>
<td>c</td>
<td>f</td>
</tr>
<tr>
<td>d</td>
<td>f</td>
</tr>
<tr>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td>f</td>
<td>g</td>
</tr>
<tr>
<td>g</td>
<td>h</td>
</tr>
<tr>
<td>h</td>
<td>d</td>
</tr>
<tr>
<td>i</td>
<td></td>
</tr>
</tbody>
</table>
```
def Get1Best(⟨V, E⟩, ROOT):
    """ returns best arborescence as a map from each node to its parent """
    for v in V \ ROOT:
        bestInEdge[v] ← arg max_u∈V score[(u, v)]

    if bestInEdge contains a cycle C:
        # build a new graph in which C is contracted into a single node
        v_C ← new Node
        V' ← V ∪ {v_C} \ C
        E' ← ∅
        for e = (t, u) in E:
            if t ∉ C and u ∉ C:
                e' ← e
            elif t ∈ C and u ∉ C:
                e' ← new Edge (v_C, u)
                score[e'] ← score[e]
            elif u ∈ C and t ∉ C:
                e' ← new Edge (t, v_C)
                kicksOut[e'] ← bestInEdge[u]
                score[e'] ← score[e] − score[kicksOut[e']]
                real[e'] ← e
                # remember the original

            E' ← E' ∪ {e'}
        A ← Get1Best(⟨V', E'⟩, ROOT)
        return {real[e'] | e' ∈ A} ∪ (C_E \ {kicksOut[A[v_C]]})

    return bestInEdge
Efficient implementation:

Tarjan '77, Finding Optimum Branchings, Networks

Not recursive. Uses a union-find (a.k.a. disjoint-set) data structure to keep track of collapsed nodes.
Efficient (wrong) implementation:

- Tarjan '77, Finding Optimum Branchings*, Networks
*corrected in Camerini et al. '79, A note on finding optimum branchings, Networks

Not recursive. Uses a union-find (a.k.a. disjoint-set) data structure to keep track of collapsed nodes.
Efficient (wrong) implementation:

Tarjan '77, Finding Optimum Branchings*, Networks
*corrected in Camerini et al. ’79, A note on finding optimum branchings, Networks
Not recursive. Uses a union-find (a.k.a. disjoint-set) data structure to keep track of collapsed nodes.

Even more efficient:

Uses a Fibonacci heap to keep incoming edges sorted.
Describes how to constrain ROOT to have only one outgoing edge
Efficient (wrong) implementation:

Tarjan '77, Finding Optimum Branchings*, Networks
*corrected in Camerini et al. '79, A note on finding optimum branchings, Networks

Not recursive. Uses a union-find (a.k.a. disjoint-set) data structure to keep track of collapsed nodes.

Even more efficient:

Gabow et al. '86, Efficient Algorithms for Finding Minimum Spanning Trees in Undirected and Directed Graphs, Combinatorica

Uses a Fibonacci heap to keep incoming edges sorted.

Describes how to constrain ROOT to have only one outgoing edge

There is a version where you don’t have to specify ROOT
Camerini
The Goal

Find *exact* $k$-best parses of a sentence given the weights of the graph.
The Goal

Find *exact* $k$-*best* parses of a sentence given the weights of the graph

But why?
Find *exact $k$-best* parses of a sentence given the weights of the graph

But why?

- Model might not be correct, rerank $k$-best parses
- Constrained models (think global features)
State of the art

- MSTParser and MaltParser produce an *approximate* $k$-best list
- TurboParser has no $k$-best feature
Central Idea

1. We know how to get $A$, the 1-best arborescence.
2. There is at least one edge in $A$, which should not be in the 2nd best arborescence.
3. Let us call this maximum impact edge, say $e$.
4. We have an algorithm to find $e$.
5. Now consider two possibilities:
   - $e$ is banned (this includes the 2nd best solution)
   - $e$ is required (this includes the 1st best solution, $A$)
6. Partition the whole search space into two smaller subspaces.
   - Partition the solution space
     - $\text{reqd} = \text{set of edges that must be included}$
     - $\text{banned} = \text{set of edges that must be excluded}$
1. We know how to get $A^{(1)}$, the 1-best arborescence.
1. We know how to get $A^{(1)}$, the 1-best arborescence.
2. There is at least one edge in $A^{(1)}$, which should not be in the 2nd best arborescence.
Central Idea

1. We know how to get $A^{(1)}$, the 1-best arborescence.
2. There is at least one edge in $A^{(1)}$, which should not be in the 2nd best arborescence.
3. Let us call this maximum impact edge, say $e$. 

---

Partition the whole search space into two smaller subspaces.

Let $\text{reqd} = \text{set of edges that must be included}$ and $\text{banned} = \text{set of edges that must be excluded}$. 

▶ $e$ is banned (this includes the 2nd best solution)
▶ $e$ is required (this includes the 1st best solution, $A^{(1)}$)
Central Idea

1. We know how to get $A^{(1)}$, the 1-best arborescence.
2. There is at least one edge in $A^{(1)}$, which should not be in the 2nd best arborescence.
3. Let us call this maximum impact edge, say $e$. We have an algorithm to find $e$. 

▶ $e$ is banned (this includes the 2nd best solution)
▶ $e$ is required (this includes the 1st best solution, $A^{(1)}$)

Partition the whole search space into two smaller subspaces.

Let $\text{reqd} = \text{set of edges that must be included}$ and $\text{banned} = \text{set of edges that must be excluded}$. 
1. We know how to get $A^{(1)}$, the 1-best arborescence.

2. There is at least one edge in $A^{(1)}$, which should not be in the 2nd best arborescence.

3. Let us call this maximum impact edge, say $e$. We have an algorithm to find $e$.

4. Now consider two possibilities:
   - $e$ is banned (this includes the 2nd best solution)
   - $e$ is required (this includes the 1st best solution, $A$)
1. We know how to get $A^{(1)}$, the 1-best arborescence.
2. There is *at least one* edge in $A^{(1)}$, which should not be in the 2nd best arborescence.
3. Let us call this *maximum impact* edge, say $e$. We have an algorithm to find $e$.
4. Now consider two possibilities:
   - $e$ is banned (this includes the 2nd best solution)
   - $e$ is required (this includes the 1st best solution, $A$)
5. Partition the whole search space into two smaller subspaces.

**Partition the solution space**
Let $\textit{reqd} =$ set of edges that must be included and $\textit{banned} =$ set of edges that must be excluded.
Partitioning the solution space

\[
\begin{align*}
\text{reqd} & = \emptyset \\
\text{banned} & = \emptyset
\end{align*}
\]
Partitioning the solution space

\[
\text{reqd} = \emptyset \\
\text{banned} = \emptyset
\]
Partitioning the solution space

- reqd = ∅
- banned = ∅

- reqd = {e₀}
- banned = ∅
Partitioning the solution space
Partitioning the solution space

- reqd = ∅, banned = ∅
- reqd = ∅, banned = {e_0}
- reqd = {e_1}, banned = {e_0}
- reqd = {e_0}, banned = {e_0}
- reqd = {e_0}, banned = {e_2}
- reqd = {e_0, e_2}, banned = ∅
Partitioning the solution space

reqd = ∅
banned = ∅

reqd = ∅
banned = {e₀}

reqd = {e₁}
banned = {e₀}

reqd = {e₀}
banned = {e₀}

reqd = {e₀}
banned = {e₁}

reqd = {e₀, e₁}
banned = {e₀, e₁}

reqd = {e₀}
banned = {e₁}

reqd = {e₀}
banned = {e₀}

reqd = {e₀, e₁}
banned = {e₀, e₁}

reqd = {e₀, e₂}
banned = {e₀, e₂}

reqd = {e₀, e₂}
banned = {e₀, e₂}

reqd = {e₀, e₁}
banned = {e₀, e₁}

reqd = {e₀, e₂}
banned = {e₀, e₂}

reqd = ∅
banned = ∅
Partitioning the solution space

- reqd = ∅, banned = ∅
- reqd = {e₀}, banned = {e₀}
- reqd = {e₁}, banned = {e₀}
- reqd = {e₀}, banned = {e₀}
- reqd = {e₀, e₁}, banned = {e₀}
- reqd = {e₀, e₁}, banned = {e₀}
- reqd = {e₀, e₂}, banned = ∅
Find best arborescence $A$ s.t. $\text{reqd} \subseteq A \subseteq E \setminus \text{banned}$
Algorithm $\text{GetConstrained1Best}(G, \text{ROOT}, \text{reqd}, \text{banned})$

Find an edge $e \in A \setminus \text{reqd}$ that defines the next partition.
Algorithm $\text{FindEdgeToBan}(G, \text{ROOT}, A, \text{reqd}, \text{banned})$

Smart way to search the subspace of solutions
Algorithm $\text{GetKBest}(G, \text{ROOT}, k)$
Algorithm GetConstrained1Best(G, ROOT, reqd, banned)

Throw out edges before you feed the graph into Get1Best:
- Throw out every edge in banned
- Throw out every edge that competes with any edge in reqd
- Run Get1Best

Runtime
\(O(n^2)\)
Find best arborescence $A$ s.t. $\text{reqd} \subseteq A \subseteq E \setminus \text{banned}$
Algorithm $\text{GetConstrained1Best}(G, \text{ROOT}, \text{reqd}, \text{banned})$

Find an edge $e \in A \setminus \text{reqd}$ that defines the next partition.
Algorithm $\text{FindEdgeToBan}(G, \text{ROOT}, A, \text{reqd}, \text{banned})$

Smart way to search the subspace of solutions
Algorithm $\text{GetKBest}(G, \text{ROOT}, k)$
Algorithm \texttt{FindEdgeToBan}(G, \text{ROOT}, A, \text{reqd}, \text{banned})

- **Input** \((A, \text{reqd}, \text{banned})\),
- **For every edge** \(e\) in \(A \setminus \text{reqd}\), find the next best alternative edge, \(\text{alt}(e)\)
  - this alternative cannot be in \text{banned}
  - the source of this alternative must not be lower down in the tree \(A\)
- **Return** \texttt{eBan}, the edge \(e\) in \(A \setminus \text{reqd}\) with the highest scoring alternative
- **Return** \(\text{diff} = \text{score(}e\text{Ban}) - \text{alt(}e\text{Ban})\)

Return variables \texttt{eBan}, \texttt{diff}

**Runtime**

\(O(n^2)\)
Example run $\text{FindEdgeToBan}$

$\text{FindEdgeToBan}(G, \text{ROOT}, A^{(1)}, \text{reqd} = \emptyset, \text{banned} = \emptyset)$

diff $= +\infty$, eBan $= \emptyset$
Example run \textit{FindEdgeToBan}

\textbf{FindEdgeToBan}(G, \text{ROOT}, A^{(1)}, \text{reqd} = \emptyset, \text{banned} = \emptyset)

\textbf{diff} = +\infty, \text{eBan} = \emptyset
Example run FindEdgeToBan

FindEdgeToBan\((G, \text{ROOT}, A^{(1)}, \text{reqd} = \emptyset, \text{banned} = \emptyset)\)

diff = +\infty, e\text{Ban} = \emptyset
Example run FindEdgeToBan

FindEdgeToBan\((G, \text{ROOT}, A^{(1)}, \text{reqd} = \emptyset, \text{banned} = \emptyset)\)

\[\text{alt}(d) = b\]
\[\text{diff} = 10, \text{eBan} = d\]
Example run FindEdgeToBan

FindEdgeToBan\((G, \text{ROOT}, A^{(1)}, \text{reqd} = \emptyset, \text{banned} = \emptyset)\)

\[\text{alt}(d) = b\]
\[\text{diff} = 10, \ e\text{Ban} = d\]
Example run \textbf{FindEdgeToBan}

\textbf{FindEdgeToBan}(G, \text{ROOT}, A^{(1)}, \text{reqd} = \emptyset, \text{banned} = \emptyset)

\begin{itemize}
  \item \texttt{alt}(d) = b
  \item \texttt{diff} = 10, eBan = d
\end{itemize}
Example run $\text{FindEdgeToBan}$

$$\text{FindEdgeToBan}(G, \text{ROOT}, A^{(1)}, \text{reqd} = \emptyset, \text{banned} = \emptyset)$$

$$alt(d) = b$$

$$\text{diff} = 10, \text{eBan} = d$$
Example run \textbf{FindEdgeToBan}

\textbf{FindEdgeToBan}(G, \textsc{ROOT}, A^{(1)}, \text{reqd} = \emptyset, \text{banned} = \emptyset)

\[
\begin{align*}
\text{alt}(f) &= e \\
\text{diff} &= 1, \ e\text{Ban} = f
\end{align*}
\]
Example run FindEdgeToBan

FindEdgeToBan\((G, \text{ROOT}, A^{(1)}, \text{reqd} = \emptyset, \text{banned} = \emptyset)\)

\[\text{alt}(f) = e\]
\[\text{diff} = 1, \ e\text{Ban} = f\]
Example run **FindEdgeToBan**

\[
\text{FindEdgeToBan}(G, \text{ROOT}, A^{(1)}, \text{reqd} = \emptyset, \text{banned} = \emptyset)
\]

\[
\begin{align*}
V4 & \quad \text{ROOT} \\
V5 & \quad \text{V3}
\end{align*}
\]

\[
\begin{align*}
a & : -5 \\
b & : -10 \\
c & : 1
\end{align*}
\]

\[
\begin{align*}
a & : -5 \\
b & : -10 \\
c & : 1
\end{align*}
\]

\[
\begin{align*}
diff & = 1, \ e\text{Ban} = f
\end{align*}
\]
Example run \texttt{FindEdgeToBan}

\begin{align*}
\text{FindEdgeToBan}(G, \text{ROOT}, A^{(1)}, \text{reqd} = \emptyset, \text{banned} = \emptyset)
\end{align*}

\begin{align*}
\text{alt}(f) &= e \\
\text{diff} &= 1, \text{eBan} = f
\end{align*}
Example run \textbf{FindEdgeToBan}

\begin{align*}
\text{FindEdgeToBan} & \left( G, \text{ROOT}, A^{(1)}, \text{reqd} = \emptyset, \text{banned} = \emptyset \right) \\
\text{alt}(a) & = c \\
\text{diff} & = 0, \ \text{eBan} = a
\end{align*}
Example run \texttt{FindEdgeToBan}

\texttt{FindEdgeToBan(G, ROOT, A^{(1)}, reqd = \emptyset, banned = \emptyset)}

\begin{center}
\begin{tikzpicture}
  \node (root) at (0,0) {ROOT};
  \node (v5) at (0,-5) {v5};
  \draw[->] (root) -- node[above] {$b: -9$} (v5);
  \draw[->] (v5) -- node[below] {$a: -4$} (root);
  \draw[->, color=orange] (v5) -- node[below] {$c: -4$} (root);
\end{tikzpicture}
\end{center}

\begin{center}
alt(a) = c
\end{center}

\begin{center}
diff = 0, eBan = a
\end{center}
Outline of the rest of the talk

- Find best arborescence $A$ s.t. $\text{reqd} \subseteq A \subseteq E \setminus \text{banned}$
  
  Algorithm $\text{GetConstrained1Best}(G, \text{ROOT}, \text{reqd}, \text{banned})$

- Find an edge $e \in A \setminus \text{reqd}$ that defines the next partition.
  Algorithm $\text{FindEdgeToBan}(G, \text{ROOT}, A, \text{reqd}, \text{banned})$

- Smart way to search the subspace of solutions
  Algorithm $\text{GetKBest}(G, \text{ROOT}, k)$
Revisit partitioning

- reqd = ∅, banned = ∅
- reqd = ∅, banned = {e₀}
- reqd = {e₀}, banned = ∅
- reqd = {e₀}, banned = {e₀, e₁}
- reqd = {e₁}, banned = {e₀}
- reqd = {e₀, e₂}, banned = ∅
- reqd = {e₀, e₁}, banned = {e₀, e₁
Algorithm $\text{GetKBest}(G, \text{ROOT}, k)$

- For every partition, save the following tuple:
  $(wt, e\text{Ban}, A, \text{reqd}, \text{banned})$
- $A = \text{GetConstrained1Best}(G, \text{ROOT}, \text{reqd}, \text{banned})$
corresponds to the best solution in the partition
- $\text{diff}, e\text{Ban} = \text{FindEdgeToBan}(G, \text{ROOT}, A, \text{reqd}, \text{banned})$
- $wt = \text{score}(A) - \text{diff}$
- Maintain a priority queue, $Q$ containing all tuples sorted by $wt$
- $Q$ determines which path to traverse in the search space
def GetKBest(G, ROOT, k):
    """ returns k-best arborescences ""
    reqd ← ∅ banned ← ∅
    A^{(1)} ← Get1Best(⟨G.V, G.E⟩, ROOT)
    diff, eBan ← FindEdgeToBan(G, ROOT, A^{(1)}, reqd, banned)
    Q.push((score(A^{(1)}) − diff, eBan, A^{(1)}, reqd, banned))
    for j in 2 . . . k:
        (wt, eBan, A, reqd, banned) ← Q.pop()
        if wt==−∞:
            return A^{(1)}, . . . , A^{(j−1)}
        reqd ← reqd ∪ {eBan}
        banned ← banned ∪ {eBan}
        A^{(j)} ← GetConstrained1Best(G, ROOT, reqd, banned)
        diff, eBan ← FindEdgeToBan(G, ROOT, A, reqd, banned)
        Q.push((score(A) − diff, eBan, A, reqd, banned))
        diff, eBan ← FindEdgeToBan(G, ROOT, A, reqd, banned)
        Q.push((wt − diff, eBan, A, reqd, banned))
    return A^{(1)}, . . . , A^{(k)}

Runtime
O(kn^2)
GetKBest example : 1-best

$A^{(1)} \leftarrow \text{GetConstrained1Best}(G, \text{ROOT}, \text{reqd} = \emptyset, \text{banned} = \emptyset)$
GetKBest example : 1-best

\[ A^{(1)} \leftarrow \text{GetConstrained1Best}(G, \text{ROOT}, \text{reqd} = \emptyset, \text{banned} = \emptyset) \]

\[
\begin{align*}
A^{(1)} & \leftarrow \text{GetConstrained1Best}(G, \text{ROOT}, \text{reqd} = \emptyset, \text{banned} = \emptyset) \\
& \leftarrow \text{FindEdgeToBan}(G, \text{ROOT}, A^{(1)}, \text{reqd} = \emptyset, \text{banned} = \emptyset)
\end{align*}
\]
GetKBest example : 1-best

\[ A^{(1)} \leftarrow \text{GetConstrained1Best}(G, \text{ROOT, reqd} = \emptyset, \text{banned} = \emptyset) \]

\[
\begin{array}{c}
\text{V1} \\
\downarrow \\
\text{V2} \\
\downarrow \\
\text{V3}
\end{array}
\]

\[
\begin{array}{c}
a : 5 \\
b : 1 \\
c : 1 \\
d : 11 \\
g : 10 \\
f : 5 \\
i : 8 \\
e : 4 \\
h : 9
\end{array}
\]

\[
Q = (21, a, A^{(1)}, \emptyset, \emptyset)
\]

\[
(\text{diff} = 0, \text{eBan} = a) \leftarrow \text{FindEdgeToBan}(G, \text{ROOT, } A^{(1)}, \text{reqd} = \emptyset, \text{banned} = \emptyset)
\]
GetKBest example: 2-best

\[ A^{(2)} \leftarrow \text{GetConstrained1Best}(G, \text{ROOT}, \text{reqd} = \emptyset, \text{banned} = \{a\}) \]
GetKBest example: 2-best

\[ A^{(2)} \leftarrow \text{GetConstrained1Best}(G, \text{ROOT}, \text{reqd} = \emptyset, \text{banned} = \{a\}) \]

\[
\begin{array}{c}
\text{ROOT} \\
\text{V1} \quad d: 11 \quad \text{g: 10} \\
\text{V2} \quad f: 5 \\
\text{V3} \quad i: 8 \quad \text{e: 4} \\
\text{h: 9}
\end{array}
\]

\[Q(21, a, A^{(1)}, \emptyset, \emptyset)\]

\[(\text{diff} = 1, \text{eBan} = f) \leftarrow \text{FindEdgeToBan}(G, \text{ROOT}, A^{(1)}, \text{reqd} = \{a\}, \text{banned} = \emptyset)\]
GetKBest example: 2-best

\[ A^{(2)} \leftarrow \text{GetConstrained1Best} (G, \text{ROOT}, \text{reqd} = \emptyset, \text{banned} = \{a\}) \]

\[
\begin{array}{c}
\begin{array}{c}
\text{V1} \\
\text{V2} \\
\text{V3}
\end{array} \\
\begin{array}{c}
d : 11 \\
g : 10 \\
f : 5 \\
i : 8
\end{array} \\
\begin{array}{c}
a : 5 \\
b : 1 \\
c : 1 \\
e : 4 \\
h : 9
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\text{Q} \\
\text{(21, a, } A^{(1)} \text{, a, } \emptyset \text{)} \\
\text{(20, f, } A^{(1)} \text{, a, } \emptyset \text{)}
\end{array}
\end{array}
\]

\[(\text{diff} = 1, \text{ eBan} = f) \leftarrow \text{FindEdgeToBan} (G, \text{ROOT}, A^{(1)}, \text{reqd} = \{a\}, \text{banned} = \emptyset)\]
GetKBest example: 2-best

\[ A^{(2)} \leftarrow \text{GetConstrained1Best}(G, \text{ROOT}, \text{reqd} = \emptyset, \text{banned} = \{a\}) \]

\[
\begin{array}{c}
V1 \\
\downarrow d: 11 \quad \downarrow g: 10 \\
V2 \\
\downarrow f: 5 \\
V3 \\
\downarrow e: 4 \\
\downarrow h: 9 \\
\end{array}
\]

\[
\begin{array}{c}
\text{ROOT} \\
a: 5 \\
b: 1 \\
c: 1 \\
\end{array}
\]

\[
Q = \begin{cases} 
(21, a, A^{(1)}, \emptyset, \emptyset) \\
(20, f, A^{(1)}, \{a\}, \emptyset) 
\end{cases}
\]

(diff = 1, eBan = f) \leftarrow \text{FindEdgeToBan}(G, \text{ROOT}, A^{(1)}, \text{reqd} = \{a\}, \text{banned} = \emptyset)

(diff = 2, eBan = h) \leftarrow \text{FindEdgeToBan}(G, \text{ROOT}, A^{(1)}, \text{reqd} = \emptyset, \text{banned} = \{a\})
GetKBest example: 2-best

\[ A^{(2)} \leftarrow \text{GetConstrained1Best}(G, \text{ROOT}, \text{reqd} = \emptyset, \text{banned} = \{a\}) \]

\[
\begin{array}{c|c|c|c|c}
\text{V1} & \text{V2} & \text{V3} \\
(21, a, A^{(1)}, \emptyset, \emptyset) & (20, f, A^{(1)}, \{a\}, \emptyset) & (19, h, A^{(2)}, \emptyset, \{a\}) \\
\end{array}
\]

\[
\begin{align*}
\text{d} & : 11 \quad \text{f} & : 5 \\
\text{g} & : 10 \quad \text{i} & : 8 \\
\text{e} & : 4 \quad \text{h} & : 9 \\
\end{align*}
\]

\[
\begin{align*}
\text{diff} = 1, \ e\text{Ban} = f & \leftarrow \text{FindEdgeToBan}(G, \text{ROOT}, A^{(1)}, \text{reqd} = \{a\}, \text{banned} = \emptyset) \\
\text{diff} = 2, \ e\text{Ban} = h & \leftarrow \text{FindEdgeToBan}(G, \text{ROOT}, A^{(1)}, \text{reqd} = \emptyset, \text{banned} = \{a\})
\end{align*}
\]
GetKBest example: 3-best

\[ A^{(3)} \leftarrow \text{GetConstrained1Best}(G, \text{ROOT}, \text{reqd} = \{a\}, \text{banned} = \{f\}) \]
Conclusion

- Graph-based formulation for dependency parsing
- 1-best algorithm by Chu-Liu-Edmonds
- $k$-best algorithm by Camerini