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](image)

$(O(n^2)$ 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:

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

s.t. A an arborescence

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

Best parse is:

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

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
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>
```

- **ROOT**
- **V1**: a:5, d:11, g:10
- **V2**: b:1, f:5, i:8
- **V3**: c:1, e:4, h:9

Diagram:
- **V1** to **ROOT**, **V2** to **ROOT**, **V3** to **ROOT**
- **V1** to **V2**, **V2** to **V3**, **V3** to **V1**
- **ROOT** to **V1**, **ROOT** to **V2**, **ROOT** to **V3**
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>g</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

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

<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>

Graph:
- Root
- V1: d: 11, g: 10
- V2: f: 5, i: 8 - 11
- V3
- V4: a: 5 - 10, b: 1 - 11, c: 1
An Example - Contracting Stage

![Diagram with nodes and edges labeled with values.]

<table>
<thead>
<tr>
<th>bestInEdge</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></td>
</tr>
<tr>
<td>V4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>kicksOut</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></td>
</tr>
<tr>
<td>i</td>
<td></td>
</tr>
</tbody>
</table>
An Example - Contracting Stage
An Example - Contracting Stage
An Example - Contracting Stage

- $a : -5 \rightarrow -1$
- $b : -10 \rightarrow -1$
- $c : 1 \rightarrow 5$
- $f : 5$
- $i : -3$
- $e : 4$
- $h : -1$

**bestInEdge**

<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>

**kicksOut**

<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></td>
</tr>
<tr>
<td>i</td>
<td></td>
</tr>
</tbody>
</table>

- $g, h$
- $d, h$
- $f$
An Example - Contracting Stage

![Diagram showing a tree structure with nodes and edges labeled with values.]

### bestInEdge

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

### kicksOut

<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></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

```
V5

b: -9

V1
V2
V3
V4
V5

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>
<tr>
<td>V5</td>
<td>a</td>
</tr>
</tbody>
</table>

kicksOut

<table>
<thead>
<tr>
<th></th>
<th>g, h</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>d, h</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>
```
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 \textit{reverse} order that we added them
- lock down $e$, and remove every edge in \texttt{kicksOut}(e) from \texttt{bestInEdge}. 
An Example - Expanding 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>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></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>d</td>
</tr>
</tbody>
</table>
An Example - Expanding Stage

\[
\begin{align*}
V1 & \quad a \quad g \\
V2 & \quad d \\
V3 & \quad f \\
V4 & \quad a \quad h \\
V5 & \quad a \\
\end{align*}
\]

\[
\begin{align*}
a & \quad g, h \\
b & \quad d, h \\
c & \quad f \\
d & \quad f \\
e & \quad f \\
f & \quad g \\
g & \quad d \\
h & \\
i & 
\end{align*}
\]
An Example - Expanding Stage

### Graph

- **ROOT**
  - $a: -5$
  - $b: -10$
  - $c: 1$

- **V4**
  - $f: 5$
  - $i: -3$
  - $e: 4$
  - $h: -1$

- **V3**

### Tables

#### bestInEdge

<table>
<thead>
<tr>
<th>Node</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>

#### kicksOut

<table>
<thead>
<tr>
<th>Node</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>f</td>
</tr>
<tr>
<td>g</td>
<td>g, h</td>
</tr>
<tr>
<td>h</td>
<td>g</td>
</tr>
<tr>
<td>i</td>
<td>d</td>
</tr>
</tbody>
</table>
An Example - Expanding Stage

**Graph Representation:**
- **Nodes:** ROOT, V1, V2, V3, V4, V5
- **Edges:**
  - ROOT to V4: $a = -5$
  - V4 to V3: $b = -10$
  - V3 to V4: $c = 1$
  - V4 to V2: $f = 5$
  - V3 to V2: $h = -1$
  - V2 to V1: $i = -3$
  - V1 to V2: $e = 4$
  - V2 to V4: $g$
  - V4 to V3: $h$

**Table:**

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

![Diagram of an example expanding stage with nodes labeled V1, V2, V3, ROOT, and edges labeled with weights such as a:5, b:1, c:1, d:11, g:10, e:4, h:9, f:5, i:8. There are tables labeled bestInEdge and kicksOut showing node V1 connected by a and g, V2 connected by b and d, V3 connected by c and f, and V4 and V5 with connections as indicated.]
An Example - Expanding Stage

- **V1**: a: 5, b: 1, c: 1
- **V2**: d: 11, e: 4, f: 5, i: 8
- **V3**: g: 10, h: 9

**Best In Edge**
- **V1**: a, g
- **V2**: d
- **V3**: f
- **V4**: a, h
- **V5**: a

**Kicks Out**
- a: g, h
- b: d, h
- c: f
- d: f
- e: f
- f: 
- g: 
- h: g
- i: d
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']]

    # remember the original
    E' ← E' ∪ {e'}
    A ← Get1Best(⟨V', E'⟩, ROOT)
    return {real[e'] | e' ∈ A} ∪ (CE \ {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:

*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.
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?
The Goal

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 $A_{(1)}$, 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 $e$.
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, $A_{(1)}$)
6. Partition the whole search space into two smaller subspaces.

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

Partitioning the solution space

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

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

\[
\begin{align*}
\text{reqd} &= \emptyset \\
\text{banned} &= \{e_0\}
\end{align*}
\]
Partitioning the solution space

\[ \text{ban } e_0 \]

- reqd = \( \emptyset \)
  - banned = \( \{ e_0 \} \)

- reqd = \( \emptyset \)
  - banned = \( \emptyset \)

- reqd = \( \{ e_0 \} \)
  - banned = \( \emptyset \)
Partitioning the solution space

reqd = ∅
banned = ∅

reqd = ∅
banned = {e₀}

reqd = ∅
banned = {e₀, e₁}

reqd = {e₁}
banned = {e₀}

reqd = {e₀}
banned = {e₀, e₂}

reqd = {e₀}
banned = {e₁}

reqd = {e₁}
banned = {e₀, e₁}

reqd = {e₀, e₂}
banned = ∅

ban e₀

req e₂

ban e₂
Partitioning the solution space

reqd = ∅
banned = ∅

reqd = ∅
banned = {e₀}

reqd = {e₁}
banned = {e₀}

reqd = {e₀}
banned = {e₁}

reqd = ∅
banned = {e₀, e₁}

reqd = {e₀}
banned = {e₂}

reqd = {e₀}
banned = {e₁, e₂}

reqd = {e₀, e₁}
banned = {e₂}
Partitioning the solution space

reqd = ∅
banned = ∅

reqd = ∅
banned = {e₀}

reqd = ∅
banned = {e₁, e₂}

reqd = {e₀}
banned = {e₁}

reqd = {e₁}
banned = {e₀}

reqd = {e₀}
banned = ∅

reqd = {e₁}
banned = ∅

reqd = {e₀}
banned = {e₁}

reqd = {e₀, e₁}
banned = {e₂}

reqd = {e₁}
banned = ∅

reqd = {e₀, e₁}
banned = ∅
Outline of the rest of the talk

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

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

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

Throw out edges before you feed the graph into \texttt{Get1Best}:

- Throw out every edge in \texttt{banned}
- Throw out every edge that \textit{competes} with any edge in \texttt{reqd}
- Run \texttt{Get1Best}

Runtime

$O(n^2)$
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 $\text{GetKBest}(G, \text{ROOT}, k)$
Algorithm \textbf{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 \text{eBan}, the edge \(e\) in \(A \setminus \text{reqd}\) with the highest scoring alternative
- Return \(\text{diff} = \text{score(eBan)} - \text{alt(eBan)}\)

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

\textbf{Runtime}
\(O(n^2)\)
Example run FindEdgeToBan

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

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

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

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

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

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

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

\begin{equation*}
\text{alt}(d) = b \\
\text{diff} = 10, \text{eBan} = d
\end{equation*}
Example run FindEdgeToBan

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

\[
\begin{align*}
\text{alt}(d) &= b \\
\text{diff} &= 10, \text{ eBan} = d
\end{align*}
\]
Example run \textbf{FindEdgeToBan}

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

\[
\begin{align*}
\text{alt}(d) &= b \\
\text{diff} &= 10, \text{eBan} = d
\end{align*}
\]
Example run **FindEdgeToBan**

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

\[ \text{alt}(d) = b \]

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

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

\[
\text{alt}(f) = e
\]

\[
\text{diff} = 1, \text{ eBan} = f
\]
Example run $\text{FindEdgeToBan}$

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

$\text{alt}(f) = e$

$\text{diff} = 1, \text{eBan} = f$
Example run FindEdgeToBan

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

alt\( (f) = e \)

diff = 1, eBan = f
Example run \texttt{FindEdgeToBan}

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

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

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

\[
\begin{align*}
\text{ROOT} & \quad b : -9 \\
V5 & \quad a : -4 \\
\text{alt}(a) & = c \\
\text{diff} & = 0, \ e\text{Ban} = a
\end{align*}
\]
Example run \textbf{FindEdgeToBan}

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

\begin{align*}
Alt(a) &= c \\
\text{diff} &= 0, \ e\text{Ban} = a
\end{align*}
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 $\text{GetKBest}(G, \text{ROOT}, k)$
Algorithm \text{GetKBest}(G, \text{ROOT}, k)

- For every partition, save the following tuple: \((wt, eBan, A, reqd, banned)\)
  - \(A = \text{GetConstrained1Best}(G, \text{ROOT}, reqd, banned)\) corresponds to the \textit{best} solution in the partition
  - \(\text{diff}, eBan = \text{FindEdgeToBan}(G, \text{ROOT}, A, reqd, banned)\)
  - \(wt = \text{score}(A) - \text{diff}\)
- 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

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

\[
\begin{array}{ccc}
\text{ROOT} & \quad & \\
\downarrow & \quad & \\
\text{V1} & \rightarrow & \text{V2} \\
\downarrow & \quad & \\
\text{V2} & \rightarrow & \text{V3} \\
\end{array}
\]

\[
\begin{array}{ccc}
a & \rightarrow & b \\
\quad & \quad & \\
5 & \rightarrow & 1 \\
\downarrow & \quad & \\
d & \rightarrow & c \\
\quad & \quad & \\
11 & \rightarrow & 1 \\
\downarrow & \quad & \\
g & \rightarrow & f \\
\quad & \quad & \\
10 & \rightarrow & 5 \\
\downarrow & \quad & \\
h & \rightarrow & i \\
\quad & \quad & \\
9 & \rightarrow & 8 \\
\downarrow & \quad & \\
e & \rightarrow & \\
\quad & \quad & \\
4 & \rightarrow & \\
\downarrow & \quad & \\
d & \rightarrow & \\
\quad & \quad & \\
11 & \rightarrow & \\
\downarrow & \quad & \\
g & \rightarrow & \\
\quad & \quad & \\
10 & \rightarrow & \\
\downarrow & \quad & \\
Q & \rightarrow & \\
\quad & \quad & \\
\end{array}
\]

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

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

\[
\begin{align*}
V1 & \quad d : 11 \\
V2 & \quad f : 5 \\
V3 & \quad e : 4 \\
\text{ROOT} & \quad a : 5 \\
\end{align*}
\]

\[
\begin{align*}
h : 9 \\
g : 10 \\
i : 8 \\
b : 1 \\
c : 1
\end{align*}
\]

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

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

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

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

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

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

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

\[
Q
\]

\[
\left(21, a, A^{(1)}, \emptyset, \emptyset\right) \quad \left(20, f, A^{(1)}, \{a\}, \emptyset\right)
\]

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

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

\[
\begin{aligned}
\text{root} & : 5 \\
\text{b} & : 1 \\
\text{c} & : 1 \\
\text{v1} & : 11 \\
\text{d} & : 10 \\
\text{g} & : 10 \\
\text{e} & : 4 \\
\text{h} & : 9 \\
\end{aligned}
\]

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

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

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

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

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

\[
(d_{\text{diff}} = 2, \ \text{eBan} = h) \leftarrow \text{FindEdgeToBan}(G, \text{ROOT}, A^{(1)}, \text{reqd} = \emptyset, \text{banned} = \{a\})
\]
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