Graph-based Dependency Parsing
(Chu-Liu-Edmonds algorithm)

Sam Thomson (with thanks to Swabha Swayamdipta)

University of Washington, CSE 490u

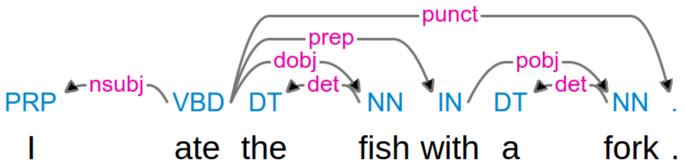
February 22, 2017

Outline

- Dependency trees
- Three main approaches to parsing
- Chu-Liu-Edmonds algorithm
- Arc scoring / Learning

Dependency Parsing - Output

Dependency Parsing



TurboParser output from
http://demo.ark.cs.cmu.edu/parse?sentence=I%20ate%20the%20fish%20with%20a%20fork.

Dependency Parsing - Output Structure

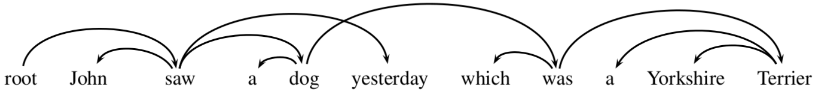
A parse is an arborescence (aka directed rooted tree):

- Directed [Labeled] Graph
- Acyclic
- Single Root
- Connected and Spanning: $\exists$ directed path from root to every other word

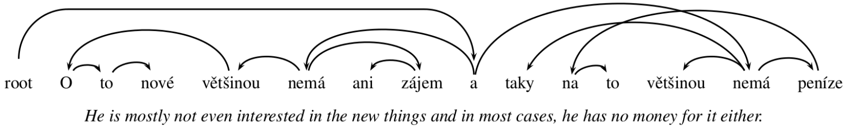
Projective / Non-projective

- 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:



and Czech:



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

Dependency Parsing - Approaches

Dependency Parsing Approaches

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

Dependency Parsing Approaches

- Chart (Eisner, CKY)
  - $O(n^3)$
  - *Only* produces projective parses
- Shift-reduce
  - $O(n)$ (*fast!*), but inexact
  - "Pseudo-projective" trick can capture some non-projectivity

Dependency Parsing Approaches

- Chart (Eisner, CKY)
  - $O(n^3)$
  - *Only* produces projective parses
- Shift-reduce
  - $O(n)$ (*fast!*), but inexact
  - "Pseudo-projective" trick can capture some non-projectivity
- Graph-based (MST)
  - $O(n^2)$ for arc-factored
  - Can produce projective *and* non-projective parses

Graph-based Dependency Parsing

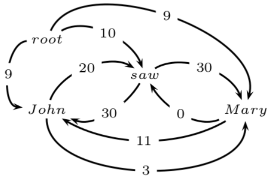
Arc-Factored Model

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

Arc-Factored Model

Every possible labeled directed edge e between every pair of nodes gets a score, `score`(e).

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



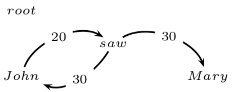
$(O(n^2)$ edges)

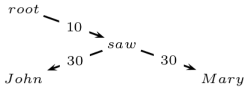
Example from *Non-projective Dependency Parsing using Spanning Tree Algorithms McDonald et al., EMNLP '05*

Arc-Factored Model

Best parse is:

$$A^* = \arg\max_{A \subseteq G} \sum_{e \in A} \texttt{score}(e)$$





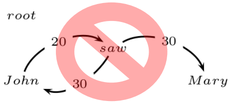
etc. . .

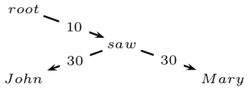
Example from *Non-projective Dependency Parsing using Spanning Tree Algorithms McDonald et al., EMNLP '05*

Arc-Factored Model

Best parse is:

$$A^* = \underset{\substack{A \subseteq G \\ \text{s.t. } A \text{ an arborescence}}}{\arg\max} \sum_{e \in A} \texttt{score}(e)$$





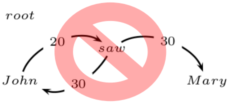
etc. . .

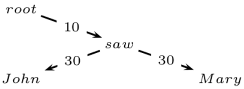
Example from *Non-projective Dependency Parsing using Spanning Tree Algorithms McDonald et al., EMNLP '05*

Arc-Factored Model

Best parse is:

$$A^* = \underset{\substack{A \subseteq G \\ \text{s.t. } A \text{ an arborescence}}}{\arg\max} \sum_{e \in A} \texttt{score}(e)$$





etc. . .

The Chu-Liu-Edmonds algorithm finds this argmax.

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

Chu-Liu-Edmonds

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

Chu-Liu-Edmonds - Intuition

Every non-`ROOT` node needs exactly 1 incoming edge
In fact, every connected component that doesn't contain `ROOT` needs exactly 1 incoming edge

Chu-Liu-Edmonds - Intuition

Every non-`ROOT` node needs exactly 1 incoming edge
In fact, every connected component that doesn't contain `ROOT` needs exactly 1 incoming edge

- Greedily pick an incoming edge for each node.

Chu-Liu-Edmonds - Intuition

Every non-ROOT node needs exactly 1 incoming edge
In fact, every connected component that doesn't contain ROOT needs exactly 1 incoming edge

- Greedily pick an incoming edge for each node.
- If this forms an arborescence, great!

Chu-Liu-Edmonds - Intuition

Every non-`ROOT` node needs exactly 1 incoming edge
In fact, every connected component that doesn't contain `ROOT` 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.

Chu-Liu-Edmonds - Intuition

Every non-`ROOT` node needs exactly 1 incoming edge
In fact, every connected component that doesn't contain `ROOT` 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.

Chu-Liu-Edmonds - Intuition

Every non-ROOT node needs exactly 1 incoming edge
In fact, every connected component that doesn't contain ROOT 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 that doesn't contain `ROOT` 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 - Recursive (Inefficient) Definition

```
def maxArborescence(V, E, ROOT):
    """ returns best arborescence as a map from each node to its parent """
    for v in V \ ROOT:
        bestInEdge[v] ← arg max_{e∈inEdges[v]} e.score

        if bestInEdge contains a cycle C:
            # build a new graph where C is contracted into a single node
            v_C ← new Node()
            V' ← V ∪ {v_C} \ C
            E' ← {adjust(e) for e ∈ E \ C}
            A ← maxArborescence(V', E', ROOT)
            return {e.original for e ∈ A} ∪ C \ {A[v_C].kicksOut}

    # each node got a parent without creating any cycles
    return bestInEdge

    def adjust(e):
        e' ← copy(e)
        e'.original ← e
        if e.dest ∈ C:
            e'.dest ← v_C
            e'.kicksOut ← bestInEdge[e.dest]
            e'.score ← e.score − e'.kicksOut.score
        elif e.src ∈ C:
            e'.src ← v_C
        return e'
```

Chu-Liu-Edmonds

Consists of two stages:

- **Contracting** (everything *before* the recursive call)
- **Expanding** (everything *after* the recursive call)

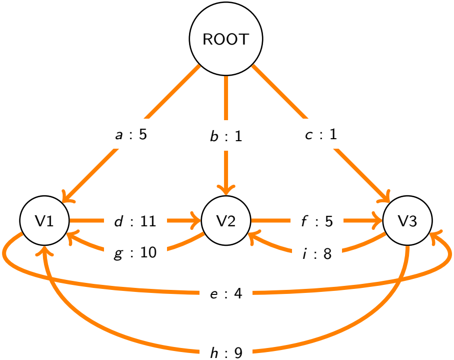
Chu-Liu-Edmonds - Preprocessing

- Remove every edge incoming to `ROOT`
  - This ensures that `ROOT` is in fact the root of any solution
- For every ordered pair of nodes, v_i, v_j, remove all but the highest-scoring edge from v_i to v_j

Chu-Liu-Edmonds - Contracting Stage

- For each non-ROOT node v, set `bestInEdge`[v] to be its highest scoring incoming edge.
- If a cycle C is formed:
  - contract the nodes in C into a new node v_C
  - edges outgoing from any node in C now get source v_C
  - edges incoming to any node in C now get destination v_C
  - For each node u in C, and for each edge e incoming to u from outside of C:
    - set e.`kicksOut` to `bestInEdge`[u], and
    - set e.`score` to be e.`score` − e.`kicksOut`.`score`.
- Repeat until every non-ROOT node has an incoming edge and no cycles are formed

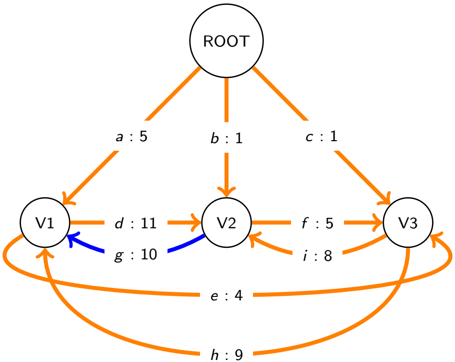
An Example - Contracting Stage



	bestInEdge
V1	
V2	
V3	

	kicksOut
a	
b	
c	
d	
e	
f	
g	
h	
i	

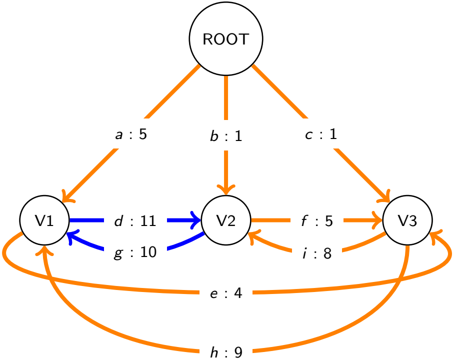
An Example - Contracting Stage



	bestInEdge
V1	g
V2	
V3	

	kicksOut
a	
b	
c	
d	
e	
f	
g	
h	
i	

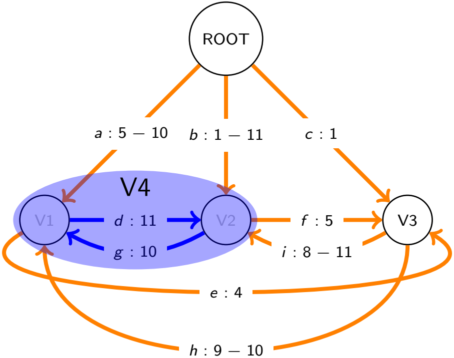
An Example - Contracting Stage



	bestInEdge
V1	g
V2	d
V3	

	kicksOut
a	
b	
c	
d	
e	
f	
g	
h	
i	

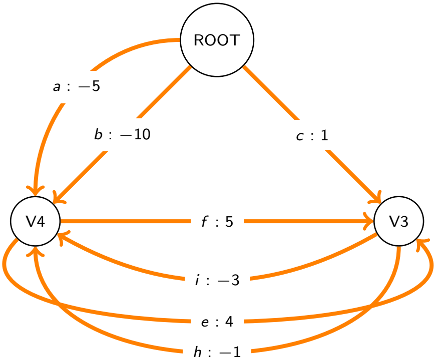
An Example - Contracting Stage



	bestInEdge
V1	g
V2	d
V3	

	kicksOut
a	g
b	d
c	
d	
e	
f	
g	
h	g
i	d

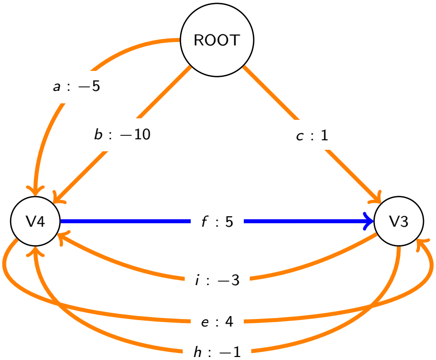
An Example - Contracting Stage



	bestInEdge
V1	g
V2	d
V3	
V4	

	kicksOut
a	g
b	d
c	
d	
e	
f	
g	
h	g
i	d

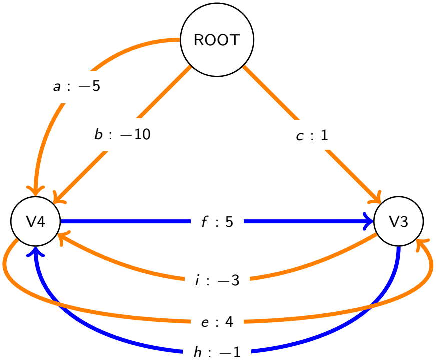
An Example - Contracting Stage



	bestInEdge
V1	g
V2	d
V3	f
V4	

	kicksOut
a	g
b	d
c	
d	
e	
f	
g	
h	g
i	d

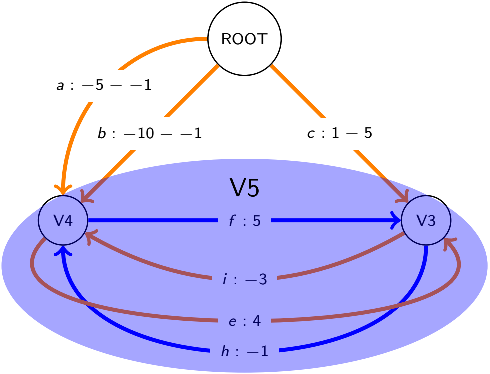
An Example - Contracting Stage



	bestInEdge
V1	g
V2	d
V3	f
V4	h

	kicksOut
a	g
b	d
c	
d	
e	
f	
g	
h	g
i	d

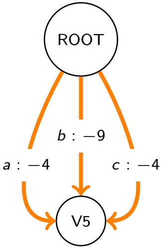
An Example - Contracting Stage



	bestInEdge
V1	g
V2	d
V3	f
V4	h
V5	

	kicksOut
a	g, h
b	d, h
c	f
d	
e	
f	
g	
h	g
i	d

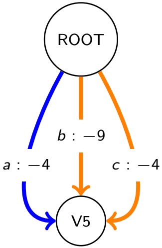
An Example - Contracting Stage



	bestInEdge
V1	g
V2	d
V3	f
V4	h
V5	

	kicksOut
a	g, h
b	d, h
c	f
d	
e	f
f	
g	
h	g
i	d

An Example - Contracting Stage



	bestInEdge
V1	g
V2	d
V3	f
V4	h
V5	a

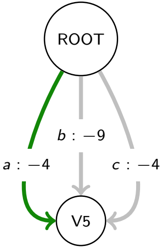
	kicksOut
a	g, h
b	d, h
c	f
d	
e	f
f	
g	
h	g
i	d

Chu-Liu-Edmonds - Expanding Stage

After the contracting stage, every contracted node will have exactly one `bestInEdge`. This edge will kick out one edge inside the contracted node, breaking the cycle.

- Go through each `bestInEdge` e in the *reverse* order that we added them
- lock down e, and remove every edge in `kicksOut`(e) from `bestInEdge`.

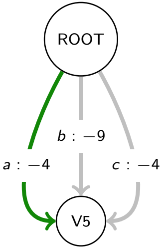
An Example - Expanding Stage



	bestInEdge
V1	g
V2	d
V3	f
V4	h
V5	a

	kicksOut
a	g, h
b	d, h
c	f
d	
e	f
f	
g	
h	g
i	d

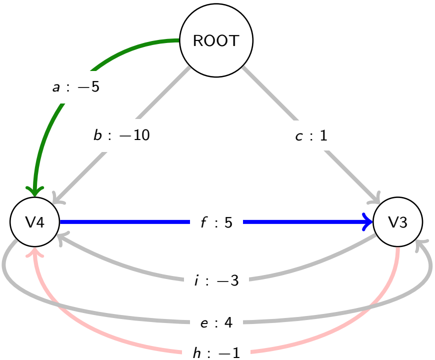
An Example - Expanding Stage



	bestInEdge
V1	a ~~g~~
V2	d
V3	f
V4	a ~~h~~
V5	a

	kicksOut
a	g, h
b	d, h
c	f
d	
e	f
f	
g	
h	g
i	d

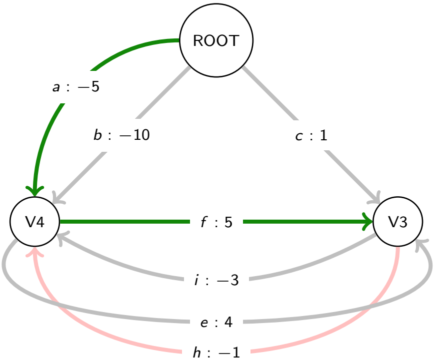
An Example - Expanding Stage



	bestInEdge
V1	a ~~g~~
V2	d
V3	f
V4	a ~~h~~
V5	a

	kicksOut
a	g, h
b	d, h
c	f
d	
e	f
f	
g	
h	g
i	d

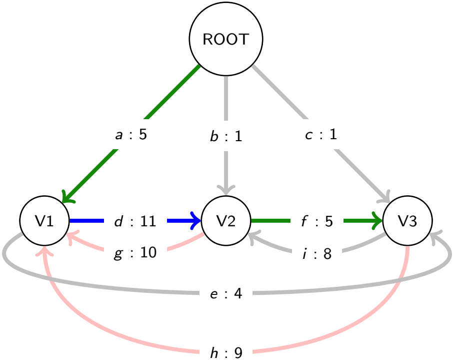
An Example - Expanding Stage



	bestInEdge
V1	a ~~g~~
V2	d
V3	f
V4	a ~~h~~
V5	a

	kicksOut
a	g, h
b	d, h
c	f
d	
e	f
f	
g	
h	g
i	d

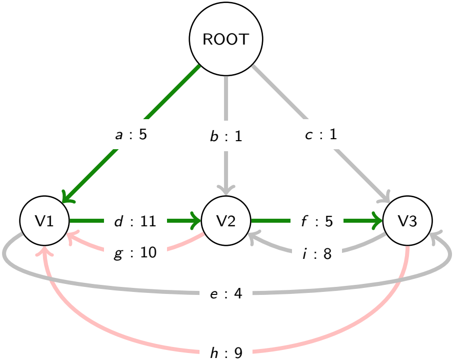
An Example - Expanding Stage



	bestInEdge
V1	a ~~g~~
V2	d
V3	f
V4	a ~~h~~
V5	a

	kicksOut
a	g, h
b	d, h
c	f
d	
e	f
f	
g	
h	g
i	d

An Example - Expanding Stage



	`bestInEdge`
V1	a ~~g~~
V2	d
V3	f
V4	a ~~h~~
V5	a

	`kicksOut`
a	g, h
b	d, h
c	f
d	
e	f
f	
g	
h	g
i	d

Chu-Liu-Edmonds - Notes

- This is a greedy algorithm with a clever form of delayed back-tracking to recover from inconsistent decisions (cycles).
- CLE is exact: it always recovers the optimal arborescence.

Chu-Liu-Edmonds - Notes

- 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.

Chu-Liu-Edmonds - Notes

- 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.

Chu-Liu-Edmonds - Notes

- 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.
  Finds cycles by following `bestInEdge` instead of randomly visiting nodes.
  Describes how to constrain `ROOT` to have only one outgoing edge

Arc Scoring / Learning

Arc Scoring

Features

can look at source (head), destination (child), and arc label.
For example:

- number of words between head and child,
- sequence of POS tags between head and child,
- is head to the left or right of child?
- vector state of a recurrent neural net at head and child,
- vector embedding of label,
- etc.

Learning

Recall that when we have a parameterized model, and we have a decoder that can make predictions given that model. . .

Learning

Recall that when we have a parameterized model, and we have a decoder that can make predictions given that model...
we can use structured perceptron, or structured hinge loss:

$$\mathcal{L}_\theta(x_i, y_i) = \max_{y \in \mathcal{Y}} \{\texttt{score}_\theta(y) + \texttt{cost}(y, y_i)\} - \texttt{score}_\theta(y_i)$$