Graph-based Dependency Parsing

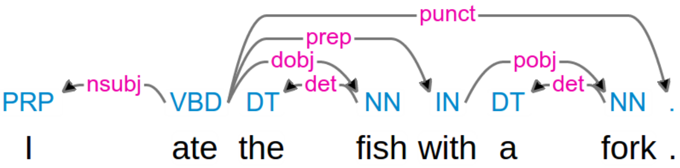
Chu-Liu-Edmonds and Camerini (k-best)

Swabha Swayamdipta Sam Thomson

Carnegie Mellon University

November 13, 2014

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

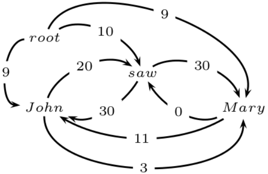
Arc-Factored Model

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

Arc-Factored Model

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

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



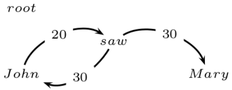
($O(n^2)$ edges)

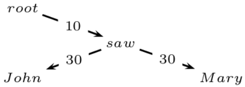
Example from *Non-projective Dependency Parsing using Spanning Tree Algorithms McDonald et al., EMNLP '05*

Arc-Factored Model

Best parse is:

$$A^{(1)} = \operatorname*{arg\,max}_{A \subseteq G} \sum_{e \in A} \texttt{score}(e)$$



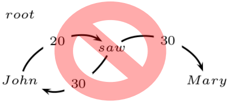


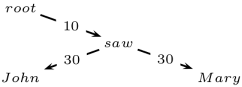
Example from *Non-projective Dependency Parsing using Spanning Tree Algorithms McDonald et al., EMNLP '05*

Arc-Factored Model

Best parse is:

$$A^{(1)} = \underset{\substack{A \subseteq G \\ \text{s.t. } A \text{ an arborescence}}}{\arg\max} \sum_{e \in A} \texttt{score}(e)$$



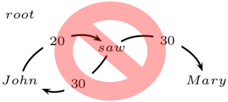


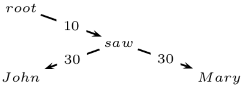
Example from *Non-projective Dependency Parsing using Spanning Tree Algorithms McDonald et al., EMNLP '05*

Arc-Factored Model

Best parse is:

$$A^{(1)} = \underset{\substack{A \subseteq G \\ \text{s.t. } A \text{ an arborescence}}}{\arg\max} \sum_{e \in A} \texttt{score}(e)$$





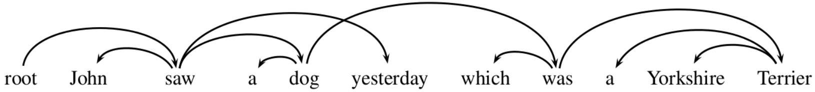
The Chu-Liu-Edmonds algorithm finds this argmax.

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

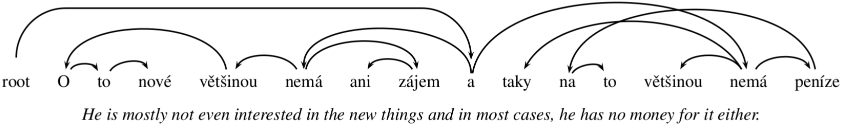
Projective / Non-projective

- 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:



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-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 needs exactly 1 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.

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!

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.

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.

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.

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

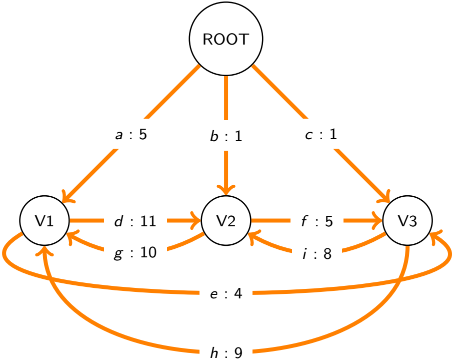
Consists of two stages:

- Contracting
- Expanding

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 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 `bestInEdge`[u] to `kicksOut`$[e]$, and
    - set the score of e to be `score`$[e]$ − `score`[`bestInEdge`$[u]$].
- Repeat until every non-`ROOT` node has an incoming edge and no cycles are formed

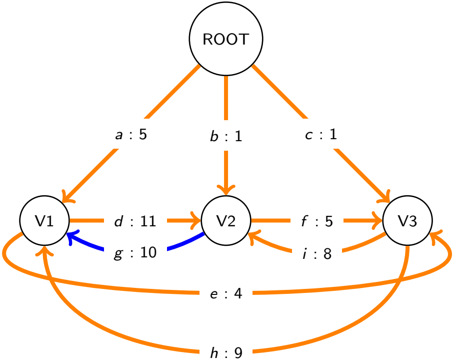
An Example - Contracting Stage



	`bestInEdge`
V1	
V2	
V3	

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

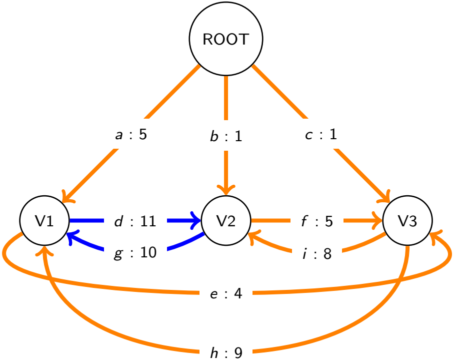
An Example - Contracting Stage



	`bestInEdge`
V1	g
V2	
V3	

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

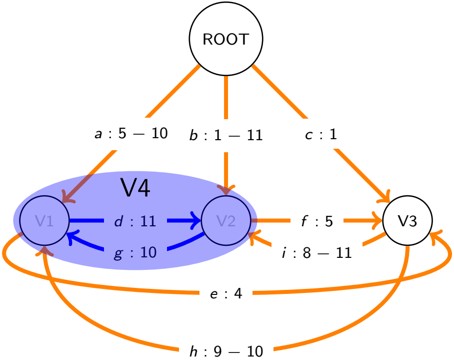
An Example - Contracting Stage



	`bestInEdge`
V1	g
V2	d
V3	

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

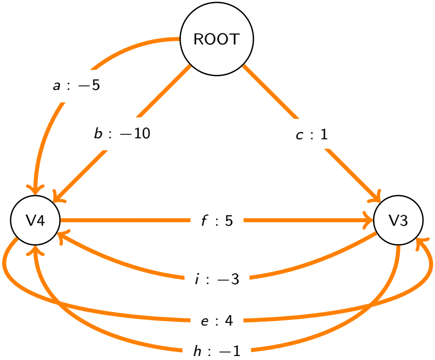
An Example - Contracting Stage



	`bestInEdge`
V1	g
V2	d
V3	

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

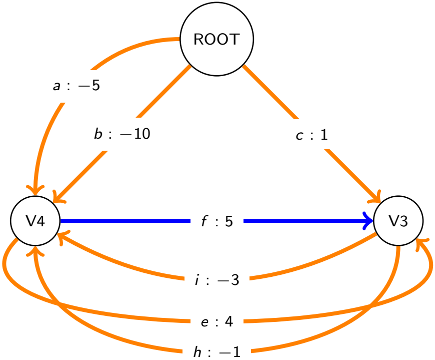
An Example - Contracting Stage



	`bestInEdge`
V1	g
V2	d
V3	
V4	

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

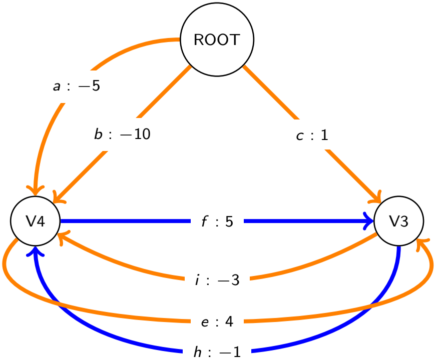
An Example - Contracting Stage



	bestInEdge
V1	g
V2	d
V3	f
V4	

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

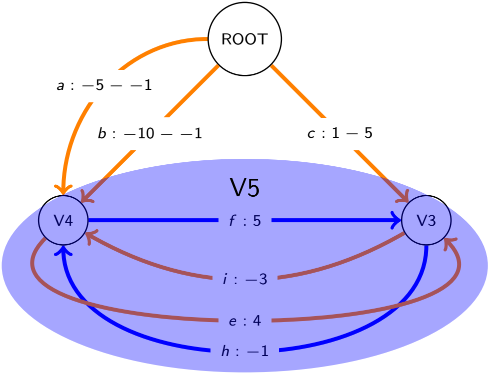
An Example - Contracting Stage



	bestInEdge
V1	g
V2	d
V3	f
V4	h

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

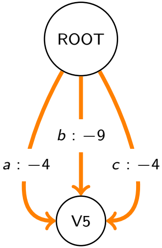
An Example - Contracting Stage



	`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

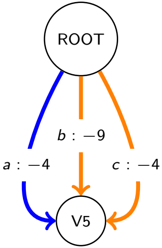
An Example - Contracting Stage



	`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

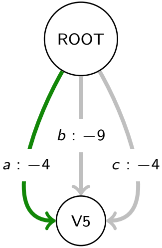
	`kicksOut`
a	g, h
b	d, h
c	f
d	
e	f
f	
g	
h	g
i	d

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`.

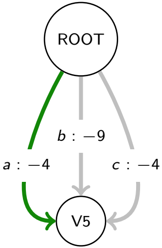
An Example - Expanding Stage



	`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

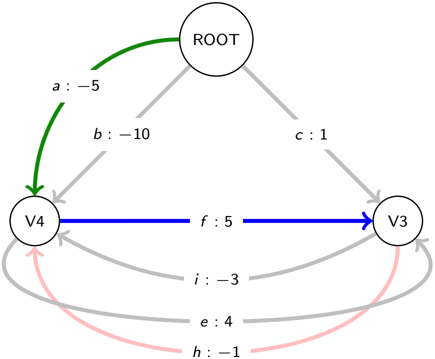
An Example - Expanding Stage



	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

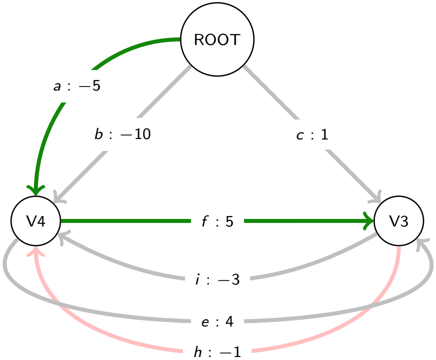
An Example - Expanding Stage



	`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

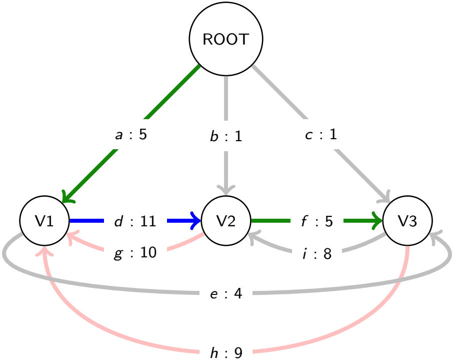
An Example - Expanding Stage



	`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

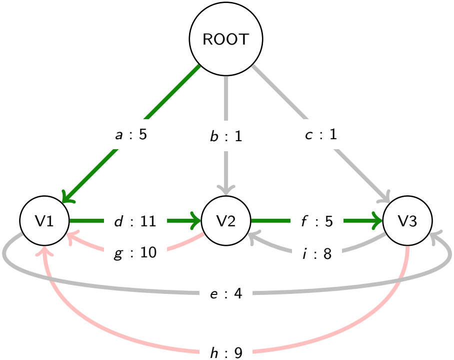
An Example - Expanding Stage



	`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 - Recursive Definition

```
def Get1Best($\langle V, E \rangle$, ROOT):
    """ returns best arborescence as a map from each node to its parent """
    for $v$ in $V \setminus ROOT$:
        bestInEdge[$v$] $\leftarrow \arg\max_{u \in V}$ score[$(u, v)$]

        if bestInEdge contains a cycle $C$:
            # build a new graph in which $C$ is contracted into a single node
            $v_C \leftarrow$ new Node
            $V' \leftarrow V \cup \{v_C\} \setminus C$
            $E' \leftarrow \emptyset$
            for $e = (t, u)$ in $E$:
                if $t \notin C$ and $u \notin C$:
                    $e' \leftarrow e$
                elif $t \in C$ and $u \notin C$:
                    $e' \leftarrow$ new Edge $(v_C, u)$
                    score[$e'$] $\leftarrow$ score[$e$]
                elif $u \in C$ and $t \notin C$:
                    $e' \leftarrow$ new Edge $(t, v_C)$
                    kicksOut[$e'$] $\leftarrow$ bestInEdge[$u$]
                    score[$e'$] $\leftarrow$ score[$e$] $-$ score[kicksOut[$e'$]]
                real[$e'$] $\leftarrow e$                                # remember the original

                $E' \leftarrow E' \cup \{e'\}$
            $A \leftarrow$ Get1Best($\langle V', E' \rangle$, ROOT)
            return $\{$real[$e'$] $\mid e' \in A\} \cup (C_E \setminus \{$kicksOut[$A[v_C]$]$\})$

    return bestInEdge
```

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.
  Describes how to constrain `ROOT` to have only one outgoing edge

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.
  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

Central Idea

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.

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.

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.
4. Now consider two possibilities:
   - e is banned (this includes the 2nd best solution)
   - e is required (this includes the 1st best solution, A)

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.
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` = set of edges that must be included
and `banned` = set of edges that must be excluded.

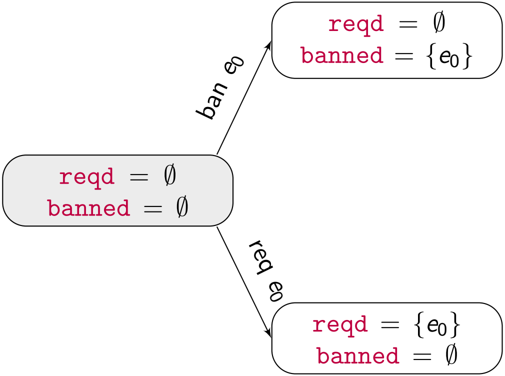
Partitioning the solution space

$$\texttt{reqd} = \emptyset$$
$$\texttt{banned} = \emptyset$$

Partitioning the solution space

`reqd` $= \emptyset$
`banned` $= \emptyset$

Partitioning the solution space



`reqd` $= \emptyset$
`banned` $= \emptyset$

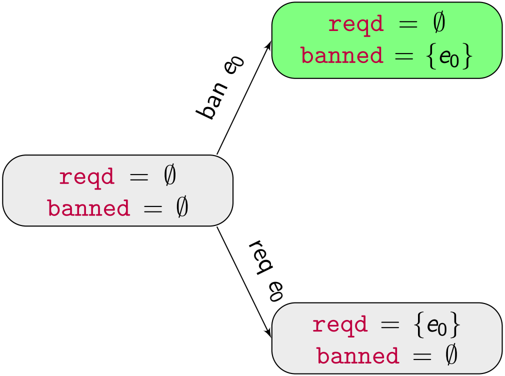
ban e_0

`reqd` $= \emptyset$
`banned` $= \{e_0\}$

req e_0

`reqd` $= \{e_0\}$
`banned` $= \emptyset$

Partitioning the solution space



`reqd` $= \emptyset$
`banned` $= \emptyset$

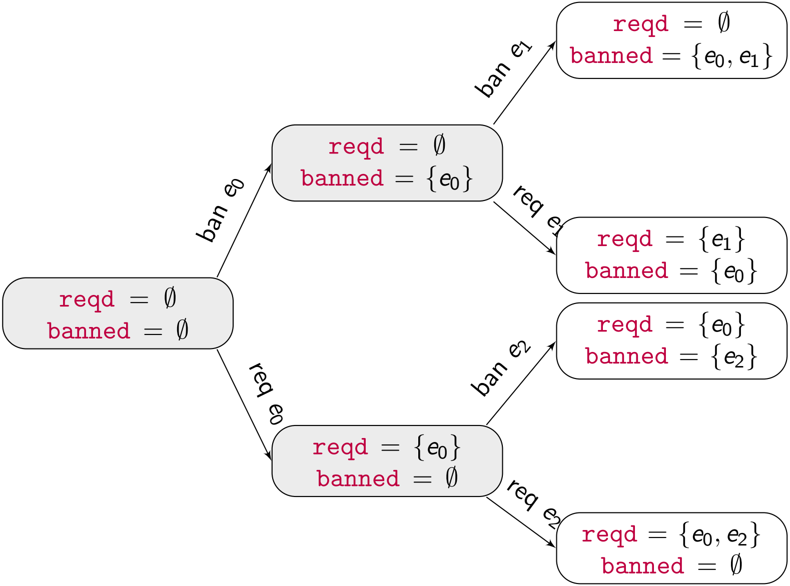
ban e_0

`reqd` $= \emptyset$
`banned` $= \{e_0\}$

req e_0

`reqd` $= \{e_0\}$
`banned` $= \emptyset$

Partitioning the solution space



`reqd` $= \emptyset$
`banned` $= \emptyset$

ban e_0

`reqd` $= \emptyset$
`banned` $= \{e_0\}$

ban e_1

`reqd` $= \emptyset$
`banned` $= \{e_0, e_1\}$

req e_1

`reqd` $= \{e_1\}$
`banned` $= \{e_0\}$

req e_0

`reqd` $= \{e_0\}$
`banned` $= \emptyset$

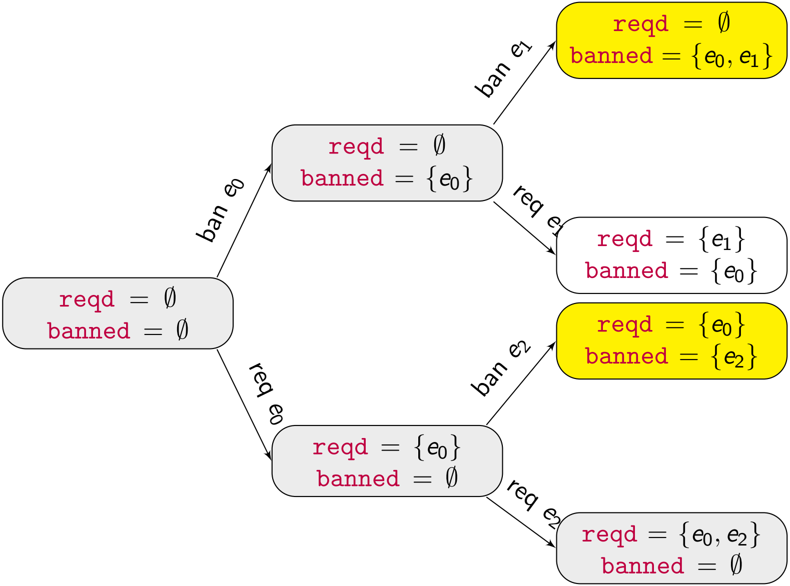
ban e_2

`reqd` $= \{e_0\}$
`banned` $= \{e_2\}$

req e_2

`reqd` $= \{e_0, e_2\}$
`banned` $= \emptyset$

Partitioning the solution space



reqd $= \emptyset$
banned $= \emptyset$

ban e_0

reqd $= \emptyset$
banned $= \{e_0\}$

ban e_1

reqd $= \emptyset$
banned $= \{e_0, e_1\}$

req e_1

reqd $= \{e_1\}$
banned $= \{e_0\}$

req e_0

reqd $= \{e_0\}$
banned $= \emptyset$

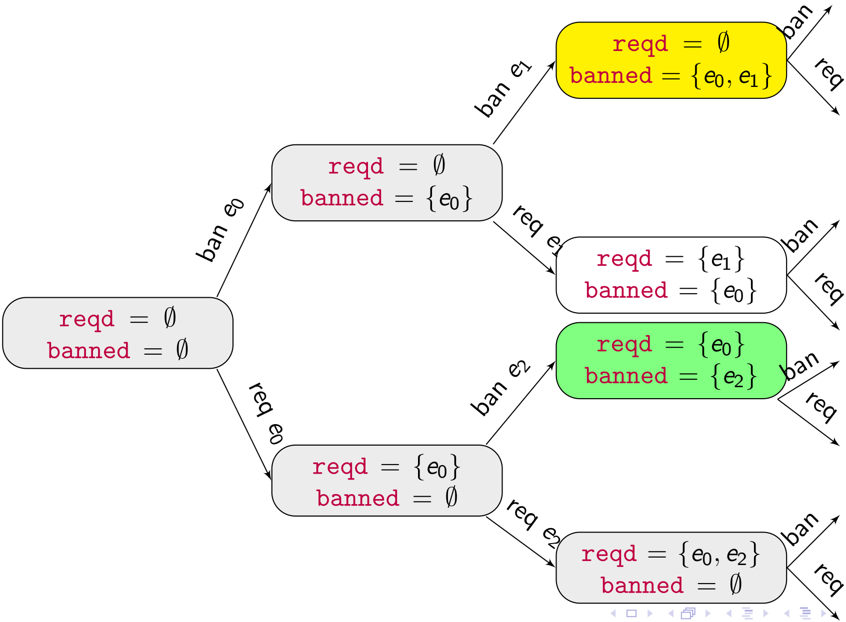
ban e_2

reqd $= \{e_0\}$
banned $= \{e_2\}$

req e_2

reqd $= \{e_0, e_2\}$
banned $= \emptyset$

Partitioning the solution space



$\texttt{reqd} = \emptyset$
$\texttt{banned} = \emptyset$

ban e_0

$\texttt{reqd} = \emptyset$
$\texttt{banned} = \{e_0\}$

ban e_1

$\texttt{reqd} = \emptyset$
$\texttt{banned} = \{e_0, e_1\}$

ban
req

req e_1

$\texttt{reqd} = \{e_1\}$
$\texttt{banned} = \{e_0\}$

ban
req

req e_0

$\texttt{reqd} = \{e_0\}$
$\texttt{banned} = \emptyset$

ban e_2

$\texttt{reqd} = \{e_0\}$
$\texttt{banned} = \{e_2\}$

ban
req

req e_2

$\texttt{reqd} = \{e_0, e_2\}$
$\texttt{banned} = \emptyset$

ban
req

Outline of the rest of the talk

- Find best arborescence A s.t. $\texttt{reqd} \subseteq A \subseteq E \setminus \texttt{banned}$
  Algorithm $\texttt{GetConstrained1Best}(\texttt{G}, \texttt{ROOT}, \texttt{reqd}, \texttt{banned})$
- Find an edge $e \in A \setminus \texttt{reqd}$ that defines the next partition.
  Algorithm $\texttt{FindEdgeToBan}(\texttt{G}, \texttt{ROOT}, A, \texttt{reqd}, \texttt{banned})$
- Smart way to search the subspace of solutions
  Algorithm $\texttt{GetKBest}(\texttt{G}, \texttt{ROOT}, \texttt{k})$

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
$O(n^2)$

Outline of the rest of the talk

- Find best arborescence A s.t. $\texttt{reqd} \subseteq A \subseteq E \setminus \texttt{banned}$
  ~~Algorithm `GetConstrained1Best`$(\texttt{G}, \texttt{ROOT}, \texttt{reqd}, \texttt{banned})$~~
- Find an edge $e \in A \setminus \texttt{reqd}$ that defines the next partition.
  Algorithm `FindEdgeToBan`$(\texttt{G}, \texttt{ROOT}, A, \texttt{reqd}, \texttt{banned})$
- Smart way to search the subspace of solutions
  Algorithm `GetKBest`$(\texttt{G}, \texttt{ROOT}, \texttt{k})$

Algorithm `FindEdgeToBan`($\mathtt{G}, \mathtt{ROOT}, A, \mathtt{reqd}, \mathtt{banned}$)

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

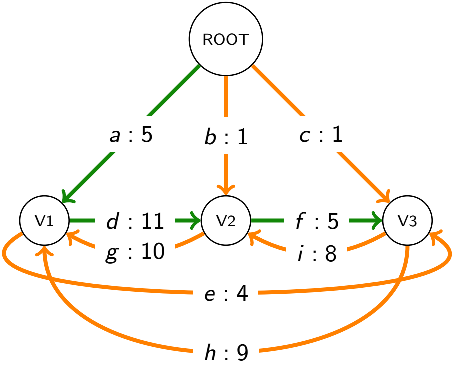
Return variables `eBan`, `diff`

Runtime

$O(n^2)$

Example run `FindEdgeToBan`

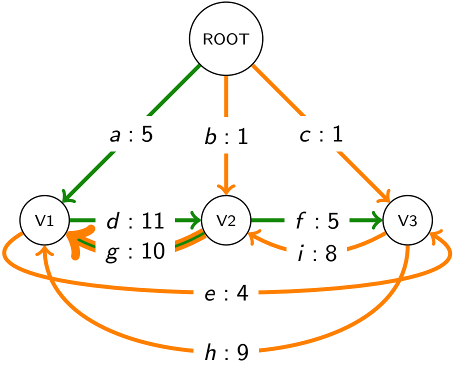
$\texttt{FindEdgeToBan}(\texttt{G}, \texttt{ROOT}, A^{(1)}, \texttt{reqd} = \emptyset, \texttt{banned} = \emptyset)$



$\texttt{diff} = +\infty, \ \texttt{eBan} = \emptyset$

Example run FindEdgeToBan

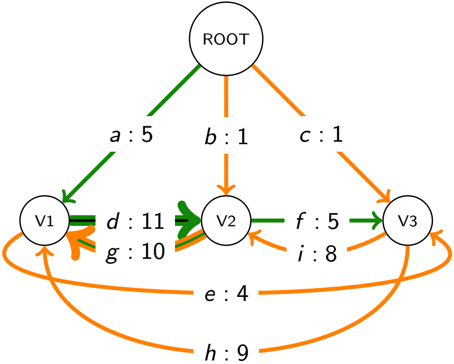
$\texttt{FindEdgeToBan}(\texttt{G}, \texttt{ROOT}, A^{(1)}, \texttt{reqd} = \emptyset, \texttt{banned} = \emptyset)$



$\texttt{diff} = +\infty, \ \texttt{eBan} = \emptyset$

Example run FindEdgeToBan

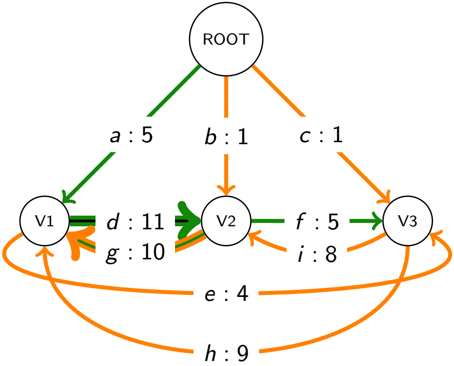
$\texttt{FindEdgeToBan}(\texttt{G}, \texttt{ROOT}, A^{(1)}, \texttt{reqd} = \emptyset, \texttt{banned} = \emptyset)$



$\texttt{diff} = +\infty, \ \texttt{eBan} = \emptyset$

Example run `FindEdgeToBan`

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

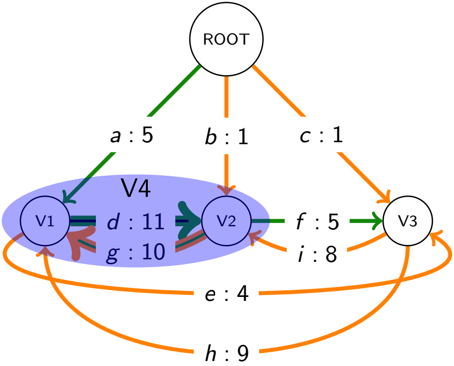


$\texttt{alt}(d) = b$

$\texttt{diff} = 10, \ \texttt{eBan} = d$

Example run `FindEdgeToBan`

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

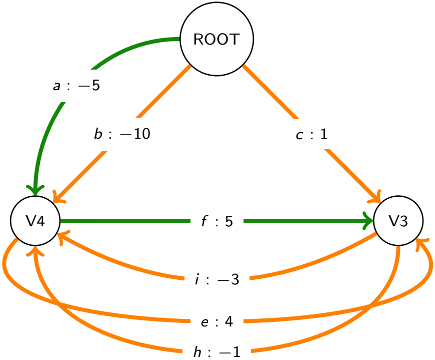


$\texttt{alt}(d) = b$

$\texttt{diff} = 10, \ \texttt{eBan} = d$

Example run `FindEdgeToBan`

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

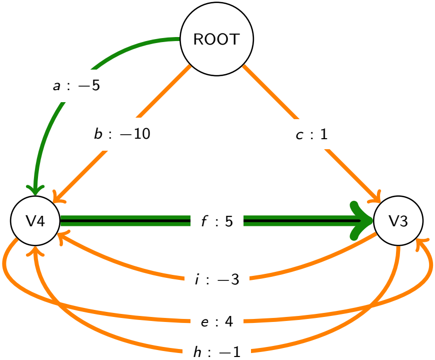


$$\texttt{alt}(d) = b$$

$$\texttt{diff} = 10, \ \texttt{eBan} = d$$

Example run `FindEdgeToBan`

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

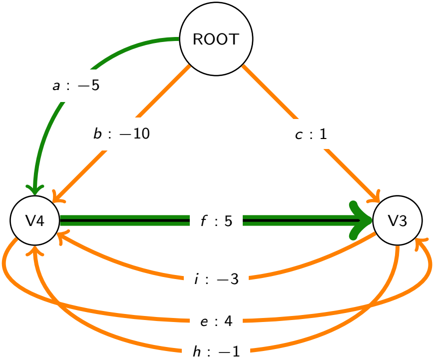


$$\texttt{alt}(d) = b$$

$$\texttt{diff} = 10, \ \texttt{eBan} = d$$

Example run `FindEdgeToBan`

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

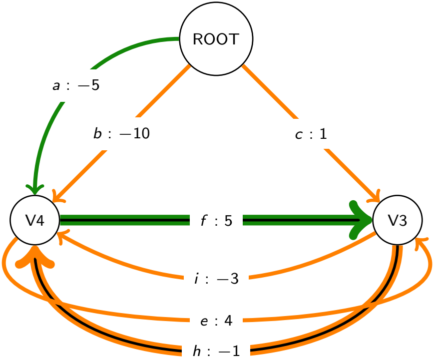


$$\texttt{alt}(f) = e$$

$$\texttt{diff} = 1, \ \texttt{eBan} = f$$

Example run `FindEdgeToBan`

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

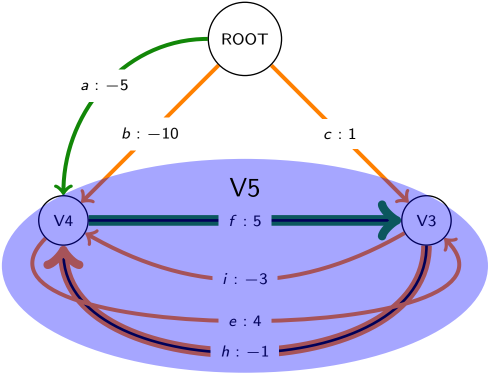


$$\texttt{alt}(f) = e$$

$$\texttt{diff} = 1, \ \texttt{eBan} = f$$

Example run FindEdgeToBan

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

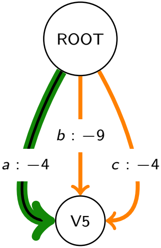


$$\texttt{alt}(f) = e$$

$$\texttt{diff} = 1, \ \texttt{eBan} = f$$

Example run `FindEdgeToBan`

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

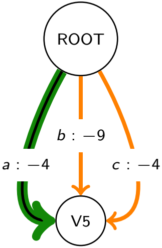


$$\texttt{alt}(f) = e$$

$$\texttt{diff} = 1,\ \texttt{eBan} = f$$

Example run `FindEdgeToBan`

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

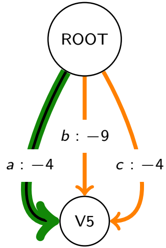


$$\texttt{alt}(a) = c$$

$$\texttt{diff} = 0, \ \texttt{eBan} = a$$

Example run FindEdgeToBan

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



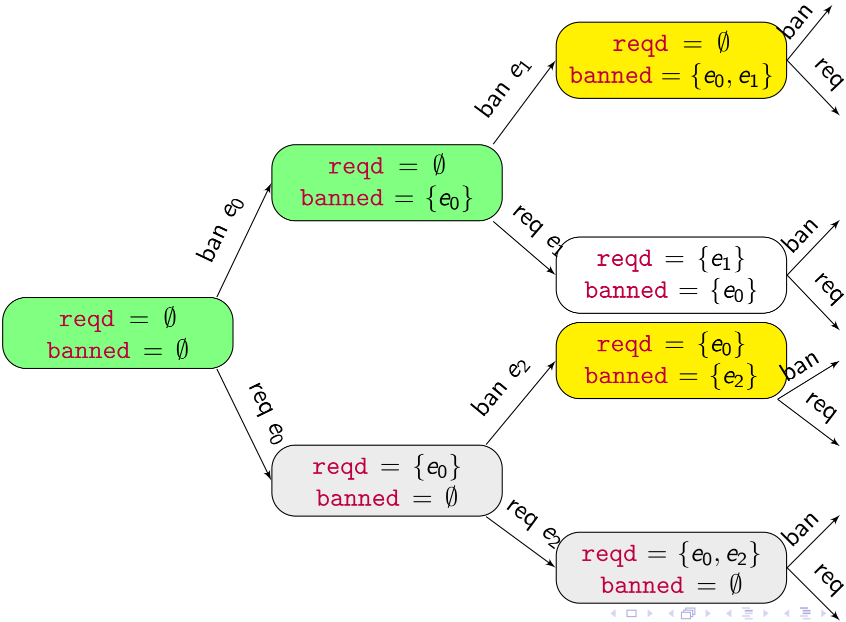
$\texttt{alt}(a) = c$

$\texttt{diff} = 0, \ \texttt{eBan} = a$

Outline of the rest of the talk

- Find best arborescence A s.t. `reqd` $\subseteq A \subseteq E \setminus$ `banned`
  ~~Algorithm `GetConstrained1Best`(G, ROOT, reqd, banned)~~
- Find an edge $e \in A \setminus$ `reqd` that defines the next partition.
  ~~Algorithm `FindEdgeToBan`(G, ROOT, A, reqd, banned)~~
- Smart way to search the subspace of solutions
  Algorithm `GetKBest`(G, ROOT, k)

Revisit partitioning



Algorithm `GetKBest`$(\mathtt{G}, \mathtt{ROOT}, \mathtt{k})$

- For every partition, save the following tuple: $(\mathtt{wt}, \mathtt{eBan}, A, \mathtt{reqd}, \mathtt{banned})$
- $A = \mathtt{GetConstrained1Best}(\mathtt{G}, \mathtt{ROOT}, \mathtt{reqd}, \mathtt{banned})$ corresponds to the *best* solution in the partition
- $\mathtt{diff}, \mathtt{eBan} = \mathtt{FindEdgeToBan}(\mathtt{G}, \mathtt{ROOT}, A, \mathtt{reqd}, \mathtt{banned})$
- $\mathtt{wt} = \mathtt{score}(A) - \mathtt{diff}$
- Maintain a priority queue, `Q` containing all tuples sorted by `wt`
- `Q` determines which path to traverse in the search space

GetKBest

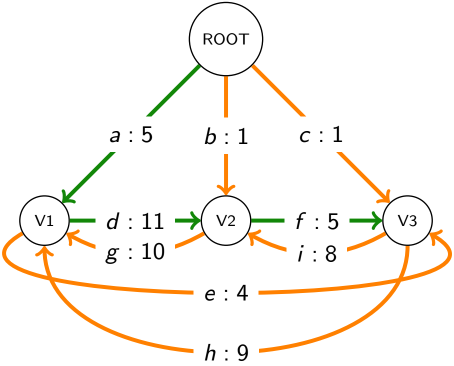
```
def GetKBest($G$, $ROOT$, $k$):
    """ returns k-best arborescences                                   """
    reqd $\leftarrow \emptyset$ banned $\leftarrow \emptyset$
    $A^{(1)} \leftarrow$ Get1Best($\langle G.V, G.E \rangle$, ROOT)
    diff, eBan $\leftarrow$ FindEdgeToBan(G, ROOT, $A^{(1)}$, reqd, banned)
    Q.push((score($A^{(1)}$) $-$ diff, eBan, $A^{(1)}$, reqd, banned))
    for $j$ in $2 \ldots k$:
        (wt, eBan, $\bar{A}$, reqd, banned) $\leftarrow$ Q.pop()
        if $wt == -\infty$:
            return $A^{(1)}, \ldots, A^{(j-1)}$
        $\overline{\text{reqd}}$ $\leftarrow$ reqd $\cup$ {eBan}
        $\overline{\text{banned}}$ $\leftarrow$ banned $\cup$ {eBan}
        $A^{(j)} \leftarrow$ GetConstrained1Best(G, ROOT, reqd, banned$'$)
        diff, eBan $\leftarrow$ FindEdgeToBan(G, ROOT, $\bar{A}$, $\overline{\text{reqd}}$, banned)
        Q.push((score($\bar{A}$) $-$ diff, eBan, $\bar{A}$, $\overline{\text{reqd}}$, banned))
        diff, eBan $\leftarrow$ FindEdgeToBan(G, ROOT, $\bar{A}$, reqd, $\overline{\text{banned}}$)
        Q.push((wt $-$ diff, eBan, $\bar{A}$, reqd, $\overline{\text{banned}}$))
    return $A^{(1)}, \ldots, A^{(k)}$
```

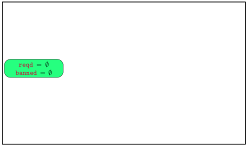
Runtime

$O(kn^2)$

GetKBest example : 1-best

$$A^{(1)} \leftarrow \texttt{GetConstrained1Best}(\texttt{G}, \texttt{ROOT}, \texttt{reqd} = \emptyset, \texttt{banned} = \emptyset)$$



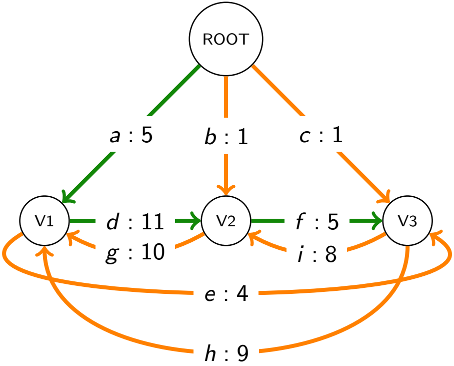


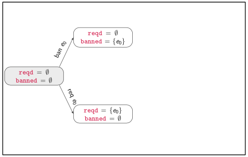
reqd = ∅
banned = ∅

Q

`GetKBest` example : 1-best

$$A^{(1)} \leftarrow \texttt{GetConstrained1Best}(\texttt{G}, \texttt{ROOT}, \texttt{reqd} = \emptyset, \texttt{banned} = \emptyset)$$



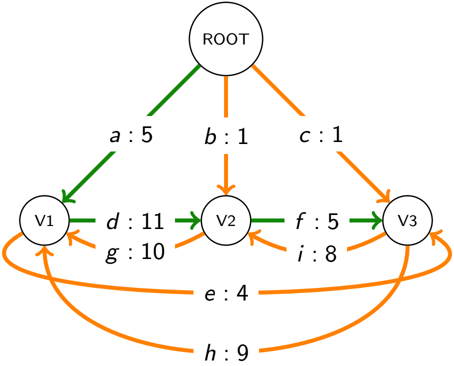


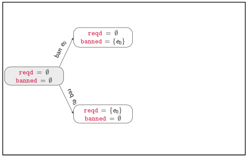
Q

$$(\texttt{diff} = 0, \texttt{eBan} = a) \leftarrow \texttt{FindEdgeToBan}(\texttt{G}, \texttt{ROOT}, A^{(1)}, \texttt{reqd} = \emptyset, \texttt{banned} = \emptyset)$$

GetKBest example : 1-best

$$A^{(1)} \leftarrow \texttt{GetConstrained1Best}(\texttt{G}, \texttt{ROOT}, \texttt{reqd} = \emptyset, \texttt{banned} = \emptyset)$$



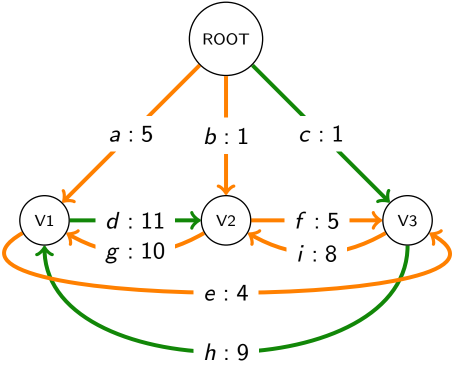


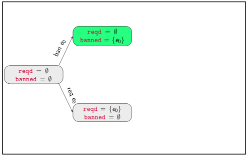
Q
$(21, a, A^{(1)}, \emptyset, \emptyset)$

$$(\texttt{diff} = 0, \texttt{eBan} = a) \leftarrow \texttt{FindEdgeToBan}(\texttt{G}, \texttt{ROOT}, A^{(1)}, \texttt{reqd} = \emptyset, \texttt{banned} = \emptyset)$$

`GetKBest` example: 2-best

$$A^{(2)} \leftarrow \texttt{GetConstrained1Best}(\texttt{G}, \texttt{ROOT}, \texttt{reqd} = \emptyset, \texttt{banned} = \{a\})$$

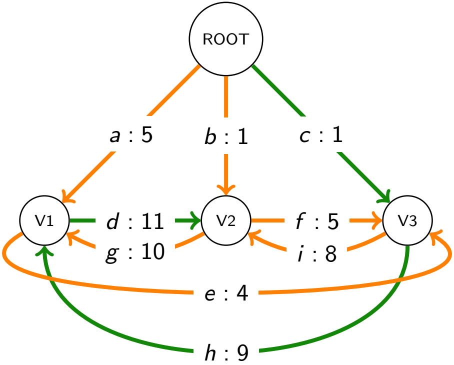


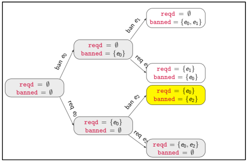


Q
~~$(21, a, A^{(1)}, \emptyset, \emptyset)$~~

`GetKBest` example: 2-best

$$A^{(2)} \leftarrow \texttt{GetConstrained1Best}(\texttt{G}, \texttt{ROOT}, \texttt{reqd} = \emptyset, \texttt{banned} = \{a\})$$



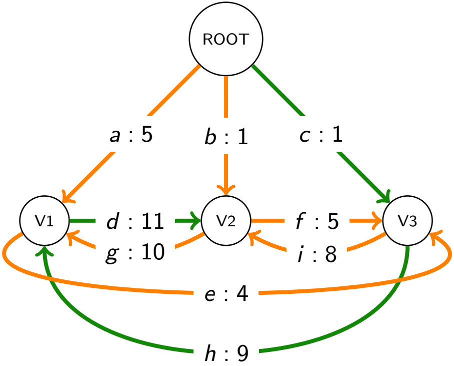


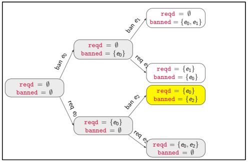
Q
~~$(21, a, A^{(1)}, \emptyset, \emptyset)$~~

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

`GetKBest` example: 2-best

$$A^{(2)} \leftarrow \texttt{GetConstrained1Best}(\texttt{G}, \texttt{ROOT}, \texttt{reqd} = \emptyset, \texttt{banned} = \{a\})$$



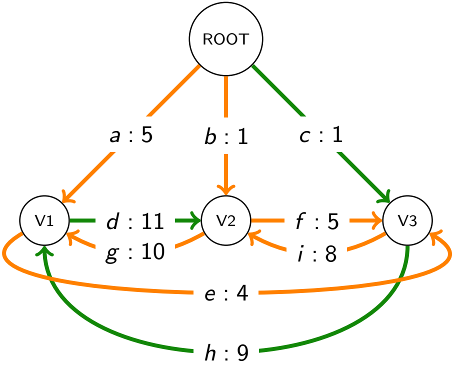


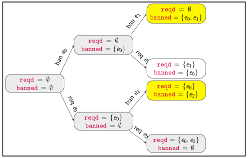
`Q`
~~$(21, a, A^{(1)}, \emptyset, \emptyset)$~~
$(20, f, A^{(1)}, \{a\}, \emptyset)$

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

GetKBest example: 2-best

$$A^{(2)} \leftarrow \texttt{GetConstrained1Best}(\texttt{G}, \texttt{ROOT}, \texttt{reqd} = \emptyset, \texttt{banned} = \{a\})$$





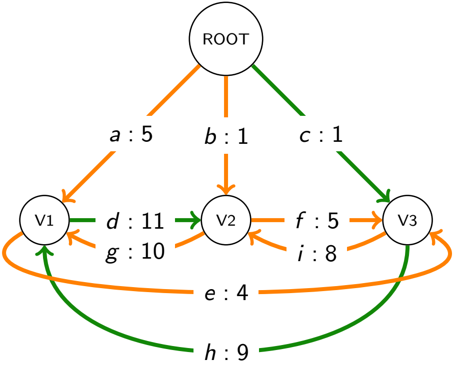
Q
~~$(21, a, A^{(1)}, \emptyset, \emptyset)$~~
$(20, f, A^{(1)}, \{a\}, \emptyset)$

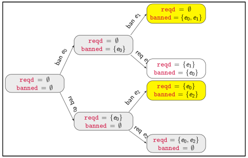
$$(\texttt{diff} = 1,\ \texttt{eBan} = f) \leftarrow \texttt{FindEdgeToBan}(\texttt{G}, \texttt{ROOT}, A^{(1)}, \texttt{reqd} = \{a\}, \texttt{banned} = \emptyset)$$

$$(\texttt{diff} = 2,\ \texttt{eBan} = h) \leftarrow \texttt{FindEdgeToBan}(\texttt{G}, \texttt{ROOT}, A^{(1)}, \texttt{reqd} = \emptyset, \texttt{banned} = \{a\})$$

GetKBest example: 2-best

$$A^{(2)} \leftarrow \texttt{GetConstrained1Best}(\texttt{G}, \texttt{ROOT}, \texttt{reqd} = \emptyset, \texttt{banned} = \{a\})$$





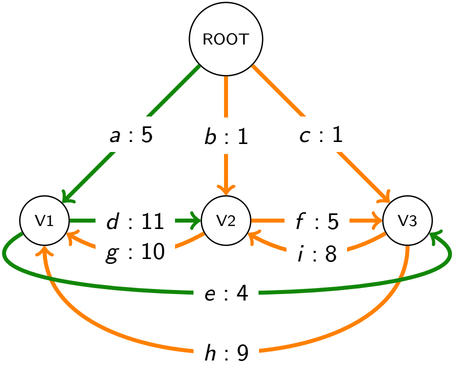
Q
~~$(21, a, A^{(1)}, \emptyset, \emptyset)$~~
$(20, f, A^{(1)}, \{a\}, \emptyset)$
$(19, h, A^{(2)}, \emptyset, \{a\})$

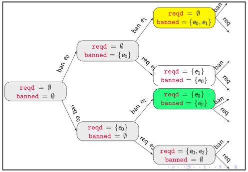
$$(\texttt{diff} = 1,\ \texttt{eBan} = f) \leftarrow \texttt{FindEdgeToBan}(\texttt{G}, \texttt{ROOT}, A^{(1)}, \texttt{reqd} = \{a\}, \texttt{banned} = \emptyset)$$

$$(\texttt{diff} = 2,\ \texttt{eBan} = h) \leftarrow \texttt{FindEdgeToBan}(\texttt{G}, \texttt{ROOT}, A^{(1)}, \texttt{reqd} = \emptyset, \texttt{banned} = \{a\})$$

`GetKBest` example : 3-best

$$A^{(3)} \leftarrow \texttt{GetConstrained1Best}(\texttt{G}, \texttt{ROOT}, \texttt{reqd} = \{a\}, \texttt{banned} = \{f\})$$





Q
~~$(21, a, A^{(1)}, \emptyset, \emptyset)$~~
~~$(20, f, A^{(1)}, \{a\}, \emptyset)$~~
$(19, h, A^{(2)}, \emptyset, \{a\})$

Conclusion

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