K-Best A* Parsing

Adam Pauls and Dan Klein
Overview

• Bottleneck of current K-Best Parsing algorithms is an exhaustive 1-best dynamic programming pass

• We propose an extension of the A* \cite{Klein2003} algorithm which permits extraction of the (true) K-Best parses without an exhaustive first pass
Lazy K-Best Parsing [Jimenez and Marzal (2000); Huang and Chiang (2005)]
Lazy K-Best Parsing

Jimenez and Marzal (2000); Huang and Chiang (2005)

run CKY inside

Time →
Lazy K-Best Parsing

run CKY inside

Time
Lazy K-Best Parsing

run CKY inside
while (#parses < k)
    lazily pop next parse
Lazy K-Best Parsing

run CKY inside
while (#parses < k)
  lazily pop next parse

Jimenez and Marzal (2000); Huang and Chiang (2005)
Lazy K-Best Parsing

run CKY inside while (#parses < k)
lazily pop next parse

very fast

Time
Lazy K-Best Parsing

Run CKY inside while (#parses < k) lazily pop next parse

bottleneck!

Very fast

Time
A Tale of Two Search Spaces

1-best Space

K-best Space
A Tale of Two Search Spaces

I-best Space

NP[0,2]

K-best Space
A Tale of Two Search Spaces

I-best Space

NP[0,2]

NN[1,2]  DT[0,1]

K-best Space
A Tale of Two Search Spaces

1-best Space

K-best Space

DT[0,1]  JJ[0,1]  NN[1,2]  NNP[1,2]

NP[0,2]
A Tale of Two Search Spaces

I-best Space

K-best Space

NP[0,2]

DT[0,1]  JJ[0,1]  NN[1,2]  NNP[1,2]

DT[0,1]  JJ[0,1]  NN[1,2]  NNP[1,2]
A Tale of Two Search Spaces

I-best Space

K-best Space

1-best Space

K-best Space
A Tale of Two Search Spaces

I-best Space

```
NP[0,2]
```

```
DT[0,1]  JJ[0,1]  NN[1,2]  NNP[1,2]
```

“edges”

K-best Space

```
NP[0,2]
```

```
DT[0,1]  NN[1,2]
```

```
JJ[0,1]  NNP[1,2]
```

```
DT[0,1]  JJ[0,1]  NN[1,2]  NNP[1,2]
```

A Tale of Two Search Spaces

I-best Space

```
NP[0,2]
```

```
DT[0,1]  JJ[0,1]  NN[1,2]  NNP[1,2]
```

“edges”

K-best Space

```
NP[0,2]
```

```
DT[0,1]  NN[1,2]
```
```
JJ[0,1]  NNP[1,2]
```

“trees”
A Tale of Two Search Spaces

I-best Space

K-best Space

$O(n^2)$
A Tale of Two Search Spaces

I-best Space

\[ O(n^2) \]

“edges”

K-best Space

“trees”

Exponential in \( n \)
Agenda-Based Search

- Maintain an *agenda* of items to be explored
- Put explored items in *chart* (or *closed list*)
Agenda-Based Search

- Maintain an *agenda* of items to be explored
- Put explored items in *chart* (or *closed list*)

```
repeat
```
Agenda-Based Search

- Maintain an *agenda* of items to be explored
- Put explored items in *chart* (or *closed list*)

```
repeat
  pop item from agenda
```
Agenda-Based Search

- Maintain an *agenda* of items to be explored
- Put explored items in *chart* (or *closed list*)

```
repeat
  pop item from agenda
  put item in chart
```
Agenda-Based Search

- Maintain an *agenda* of items to be explored
- Put explored items in *chart* (or *closed list*)

```plaintext
repeat
    pop item from agenda
    put item in chart
    build new items from item
```
Agenda-Based Search

- Maintain an *agenda* of items to be explored
- Put explored items in *chart* (or *closed list*)

```plaintext
repeat
  pop item from agenda
  put item in chart
  build new items from item
  enqueue new items
```
Agenda-Based Search

- Maintain an agenda of items to be explored
- Put explored items in chart (or closed list)

```
repeat
    pop item from agenda
    put item in chart
    build new items from item
    enqueue new items
until (GOAL item found)
```
Agenda-Based K-Best Search on Trees
repeat
repeat
  pop tree from queue
repeat
  pop tree from queue
repeat
  pop tree from queue
  if (tree is Complete-Parse) collect tree
repeat
    pop tree from queue
    if (tree is Complete-Parse) collect tree
    build bigger trees using tree
Agenda-Based K-Best Search on Trees

repeat
    pop tree from queue
    if (tree is Complete-Parse) collect tree
    build bigger trees using tree
repeat
  pop tree from queue
  if (tree is Complete-Parse) collect tree
  build bigger trees using tree
repeat
  pop tree from queue
  if (tree is Complete-Parse) collect tree
  build bigger trees using tree
repeat
  pop tree from queue
  if (tree is Complete-Parse) collect tree
  build bigger trees using tree

\[ \beta = \beta_L + \beta_R + w \]
repeat
  pop tree from queue
  if (tree is Complete-Parse) collect tree
  build bigger trees using tree
until (#parses < k)
Popping Trees
Popping Trees

DT

0 1
Popping Trees

DT

NN

0 1 1 2
Popping Trees
Popping Trees
Popping Trees
Popping Trees

DT

NN

JJ

NP

ROOT

S

NP

VP

DT NN VB NN

NNP

1 2
Popping Trees

DT

NN

JJ

NP

ROOT

DT

NN

VP

NP

S

0

1

2

0

2

4

0

1

2

0

2

NP

JJ

NNP

NNP

0

1

2
Popping Trees
Naive KA* Search on Trees

\[ \beta = \beta_L + \beta_R + w \]
Naive KA* Search on Trees

\[ \beta = \beta_L + \beta_R + w \]
Naive KA* Search on Trees

priority: $\beta + h_K(T)$

$\beta = \beta_L + \beta_R + w$
Naive KA* Search on Trees

intractable!

priority:

\[ \beta + h_K(T) \]

\[ \beta = \beta_L + \beta_R + w \]
Naive KA* Search on Trees

intractable! unless...

priority:

\[ \beta + h_K(T) \]

\[ \beta = \beta_L + \beta_R + w \]
Heuristics

- $h_K$ is a heuristic which lower bounds the Viterbi outside cost $\alpha$
Heuristics

- $h_K$ is a heuristic which lower bounds the Viterbi outside cost $\alpha$

$$h_K(T) \leq \alpha(T)$$
Heuristics

- $h_K$ is a heuristic which lower bounds the Viterbi outside cost $\alpha$

$$h_K(T) \leq \alpha(T)$$

k-best parsing is fun.
Heuristics

- $h_K$ is a heuristic which lower bounds the Viterbi outside cost $\alpha$

$$h_K(T) \leq \alpha(T)$$
Heuristics

- $h_K$ is a heuristic which lower bounds the Viterbi outside cost $\alpha$

$$h_K(T) \leq \alpha(T)$$

k-best parsing is fun
Heuristics

- $h_K$ is a heuristic which lower bounds the Viterbi outside cost $\alpha$

$$h_K(T) \leq \alpha(T)$$

- k-best parsing

- $T$ is fun
A Good Heuristic

• What if we had the exact heuristic, i.e.
A Good Heuristic

• What if we had the exact heuristic, i.e.

\[ h_K(T) = \alpha(T) \] equality!
A Good Heuristic

• What if we had the exact heuristic, i.e.

$$h_K(T) = \alpha(T)$$

equality!

• With this heuristic, it can be shown that this A* search will only pop trees that are actually part of true K-best parses
A Good Heuristic

• What if we had the exact heuristic, i.e.

\[ h_K(T) = \alpha(T) \]

equality!

• With this heuristic, it can be shown that this A* search will only pop trees that are actually part of true K-best parses

• Can compute this with exhaustive 1-best dynamic program (Viterbi inside-outside)
Naive KA* w/ Exhaustive Exact Heuristic
Naive KA* w/ Exhaustive Exact Heuristic

run CKY inside
Naive KA* w/ Exhaustive Exact Heuristic

run CKY inside
Naive KA* w/ Exhaustive Exact Heuristic

run CKY inside
run CKY outside
Naive KA* w/ Exhaustive Exact Heuristic

- run CKY inside
- run CKY outside

inside  outside
Naive KA* w/ Exhaustive Exact Heuristic

run CKY inside
run CKY outside
repeat
  pop tree from queue
  ...
until (#parses < k)
Naive KA* w/ Exhaustive Exact Heuristic

run CKY inside
run CKY outside
repeat
  pop tree from queue
  ...
until (#parses < k)

inside   outside
Naive KA* w/ Exhaustive Exact Heuristic

run CKY inside
run CKY outside
repeat
  pop tree from queue
  ...
until (#parses < k)

same as Lazy K-Best
Naive KA* w/ Exhaustive Exact Heuristic

run CKY inside
run CKY outside
repeat
  pop tree from queue
...
until (#parses < k)

exactly what we want to avoid!

same as Lazy K-Best
Building Trees Smartly

\[ \beta = \beta_L + \beta_R + w \]
Building Trees Smartly

\[ \beta = \beta_L + \beta_R + w \]
Building Trees Smartly

\[ \beta = \beta_L + \beta_R + w \]
Building Trees Smartly

\[ \beta = \beta_L + \beta_R + w \]
Building Outside Edges

\[ \alpha \]

\[ \text{VP} \]

0 3 5 \( n \)
Building Outside Edges

\[ \alpha_0 \]

\[ S \]

\[ 0 \quad 1 \quad 5 \quad n \]

\[ \alpha \]

\[ VP \]

\[ 0 \quad 3 \quad 5 \quad n \]
Building Outside Edges
Building Outside Edges

\[ \alpha \]

\[ \alpha_0 \]

\[ S \]

\[ S \]

\[ NP \]

\[ \beta_L \]

\[ w \]

\[ VP \]

\[ VP \]

\[ \alpha \]

\[ \beta_L \]
Building Outside Edges

\[ \alpha = \alpha_O + w + \beta_L \]
Building Outside Edges

\[ \alpha = \alpha_O + w + \beta_L \]
Building Outside Edges

\[ \text{priority?} \]

\[ \alpha = \alpha_O + w + \beta_L \]

Diagram showing syntactic tree structures with nodes labeled 'S', 'NP', 'VP', '0', '1', '3', '5', 'n', and edges connecting them.
Building Outside Edges

\[ \alpha = \alpha_O + w + \beta_L \]

priority?
Building Outside Edges

$\alpha_O + w + \beta_L = \alpha_O + w + \beta_L$

priority?

$\alpha + \beta_R$
Building Outside Edges

\[ \alpha_O = \alpha + \beta_R + w + \beta_L \]

priority?

need inside edges
Building 1-best Inside Edges

\[ \beta = \beta_L + \beta_R + w \]
Building 1-best Inside Edges

priority:

\[ \beta = \beta_L + \beta_R + w \]
Building 1-best Inside Edges

priority:

\[ \beta + h_1(S[1,5]) \]

\[ \beta = \beta_L + \beta_R + w \]
Building 1-best Inside Edges

user-supplied

priority:

\[ \beta = \beta_L + \beta_R + w \]
KA*

repeat
    pop item from queue
repeat
  pop item from queue
  if (item is Inside-Edge)
    build bigger inside edges
    notify waiting outside edges
repeat
  pop item from queue
  if (item is Inside-Edge)
    build bigger inside edges
    notify waiting outside edges
repeat
  pop item from queue
  if (item is Inside-Edge)
    build bigger inside edges
    notify waiting outside edges
  if (item is Outside-Edge)
    build smaller outside edges
    notify waiting trees
repeat
  pop item from queue
  if (item is Inside-Edge)
    build bigger inside edges
    notify waiting outside edges
  if (item is Outside-Edge)
    build smaller outside edges
    notify waiting trees
KA*

repeat
  pop item from queue
  if (item is Inside-Edge)
    build bigger inside edges
    notify waiting outside edges
  if (item is Outside-Edge)
    build smaller outside edges
    notify waiting trees
  if (item is Tree)
    build bigger trees
KA*

repeat
  pop item from queue
  if (item is Inside-Edge)
    build bigger inside edges
    notify waiting outside edges
  if (item is Outside-Edge)
    build smaller outside edges
    notify waiting trees
  if (item is Tree)
    build bigger trees
KA*

repeat
  pop item from queue
  if (item is Inside-Edge)
    build bigger inside edges
    notify waiting outside edges
  if (item is Outside-Edge)
    build smaller outside edges
    notify waiting trees
  if (item is Tree)
    build bigger trees
until (#parses < k)
repeat
  pop item from queue
  if (item is Inside-Edge)
    build bigger inside edges
    notify waiting outside edges
  if (item is Outside-Edge)
    build smaller outside edges
    notify waiting trees
  if (item is Tree)
    build bigger trees
until (#parses < k)
KA*

repeat
    pop item from queue
    if (item is Inside-Edge)
        build bigger inside edges
        notify waiting outside edges
    if (item is Outside-Edge)
        build smaller outside edges
        notify waiting trees
    if (item is Tree)
        build bigger trees
until (#parses < k)

Speedup depends on $h_1$
repeat
  pop item from queue
  if (item is Inside-Edge)
    build bigger inside edges
    notify waiting outside edges
  if (item is Outside-Edge)
    build smaller outside edges
    notify waiting trees
  if (item is Tree)
    build bigger trees
until (#parses < k)
repeat
  pop item from queue
  if (item is Inside-Edge)
    build bigger inside edges
    notify waiting outside edges
  if (item is Outside-Edge)
    build smaller outside edges
    notify waiting trees
  if (item is Tree)
    build bigger trees
until (#parses < k)
repeat
  pop item from queue
  if (item is Inside-Edge)
    build bigger inside edges
    notify waiting outside edges
  if (item is Outside-Edge)
    build smaller outside edges
    notify waiting trees
  if (item is Tree)
    build bigger trees
until (#parses < k)
repeat
  pop item from queue
  if (item is Inside-Edge)
    build bigger inside edges
    notify waiting outside edges
  if (item is Outside-Edge)
    build smaller outside edges
    notify waiting trees
  if (item is Tree)
    build bigger trees
until (#parses < k)
repeat
  pop item from queue
  if (item is Inside-Edge)
    build bigger inside edges
    notify waiting outside edges
  if (item is Outside-Edge)
    build smaller outside edges
    notify waiting trees
  if (item is Tree)
    build bigger trees
until (#parses < k)
repeat
  pop item from queue
  if (item is Inside-Edge)
    build bigger inside edges
    notify waiting outside edges
  if (item is Outside-Edge)
    build smaller outside edges
    notify waiting trees
  if (item is Tree)
    build bigger trees
until (#parses < k)
repeat
  pop item from queue
  if (item is Inside-Edge)
    build bigger inside edges
    notify waiting outside edges
  if (item is Outside-Edge)
    build smaller outside edges
    notify waiting trees
  if (item is Tree)
    build bigger trees
until (#parses < k)
Interleaving

repeat
  pop item from queue
  if (item is Inside-Edge)
    build bigger inside edges
    notify waiting outside edges
  if (item is Outside-Edge)
    build smaller outside edges
    notify waiting trees
  if (item is Tree)
    build bigger trees
until (#parses < k)
repeat
  pop item from queue
  if (item is Inside-Edge)
    build bigger inside edges
    notify waiting outside edges
  if (item is Outside-Edge)
    build smaller outside edges
    notify waiting trees
  if (item is Tree)
    build bigger trees
until (#parses < k)
repeat
  pop item from queue
  if (item is Inside-Edge)
    build bigger inside edges
    notify waiting outside edges
  if (item is Outside-Edge)
    build smaller outside edges
    notify waiting trees
  if (item is Tree)
    build bigger trees
until (#pares < k)
repeat
  pop item from queue
  if (item is Inside-Edge)
    build bigger inside edges
    notify waiting outside edges
  if (item is Outside-Edge)
    build smaller outside edges
    notify waiting trees
  if (item is Tree)
    build bigger trees
until (#parses < k)
Correctness

- Guarantees:
  - The first $k$ complete parses popped are the optimal $k$-best
  - Will pop only trees that are subtrees of the optimal $K$-best parses
Experimental Setup #1

• Use the state-split grammars of Petrov et al. 2006
Experimental Setup #1

- Use the state-split grammars of Petrov et al. 2006

```
S  
 \  
 NP VP

coarse
```
Experimental Setup #1

- Use the state-split grammars of Petrov et al. 2006

\[ S \rightarrow \text{refine} \]

\[ S \rightarrow \text{NP} \rightarrow \text{VP} \]

course
Experimental Setup #1

- Use the state-split grammars of Petrov et al. 2006

\[
\begin{align*}
S & \quad \text{refine} \quad S-2 \\
NP & \quad \text{coarse} \quad NP-1 \\
VP & \quad \text{fine} \quad VP-2
\end{align*}
\]
Experimental Setup #1

• Use the state-split grammars of Petrov et al. 2006

```
S
NP    VP
```

coarse

```
S-2
NP-1  VP-2
```

fine

• Parse with fine grammar (~1 million rules)
Experimental Setup #1

- Use the state-split grammars of Petrov et al. 2006

```
S               S-2
  NP         VP-2
      NP-1
```

- Parse with fine grammar (~1 million rules)

- Compute $h_1$ with coarse grammar (~100k rules)
Experimental Results #1

Inside Edges

Outside+Trees

Items pushed (millions)

k

0 1 5 10 50 100 500 1000 10000

0 2000 4000 6000 8000
Experimental Results #1

The k-best time is tiny!
Experimental Results #1

Heuristic h1

Inside Edges

Outside+Trees

Heuristic h1

Items pushed (millions)

1,000

2,000

3,000

4,000

5,000

6,000

7,000

8,000

k

1

5

10

50

100

500

1,000

10,000

Outside+Trees

Inside Edges

Heuristic h1
Experimental Results #1

Lazy K-Best

Items pushed (millions)

Outside+Trees
Inside Edges
Heuristic h1

k
1 5 10 50 100 500 1000 10000

0 5000 10000 15000
Experimental Setup #2

- We use the factored lexicalized grammar of Klein and Manning (2003)

- They construct a lexicalized grammar as the cross-product of a dependency grammar and PCFG
Experimental Setup #2

• We use the factored lexicalized grammar of Klein and Manning (2003)

• They construct a lexicalized grammar as the cross-product of a dependency grammar and PCFG

\[
\begin{array}{c}
S \\
\downarrow \\
W_P \\
\downarrow \\
NP \quad VP \\
\downarrow \\
PCFG
\end{array}
\]
Experimental Setup #2

• We use the factored lexicalized grammar of Klein and Manning (2003)

• They construct a lexicalized grammar as the cross-product of a dependency grammar and PCFG

```
S
  \( \times \)
NP \( W_p \) VP

is-VB
  \( \times \)
parsing-NN \( W_d \) is-VB

PCFG
Dependency Grammar
```
Experimental Setup #2

- We use the factored lexicalized grammar of Klein and Manning (2003)

- They construct a lexicalized grammar as the cross-product of a dependency grammar and PCFG

\[
S \times \begin{array}{c} \text{is-VB} \\ \text{parsing-NN} \\ \text{is-VB} \end{array} = \begin{array}{c} \text{S-is-VB} \\ \text{NP-parsing-NN} \\ \text{VP-is-VB} \end{array}
\]

PCFG  \quad \text{Dependency Grammar}  \quad \text{Lexicalized Grammar}
Experimental Results #2

Edges Pushed (millions)

- **Outside+Derivations**
- **Inside Edges**
- **Heuristic**

$k$ values: 1, 5, 10, 50, 100, 500, 1000, 10000
Experimental Results #2

Lazy K-Best runs out of memory on 32 GB machine
Conclusions

• We have provided an optimal extension of A* to K-Best parsing

• Given appropriate (problem-specific) heuristics, we can save significant time in 1-best phase compared to exhaustive dynamic programming

• In worst case, we do as much work as Lazy K-Best, better for general heuristics
Thank you