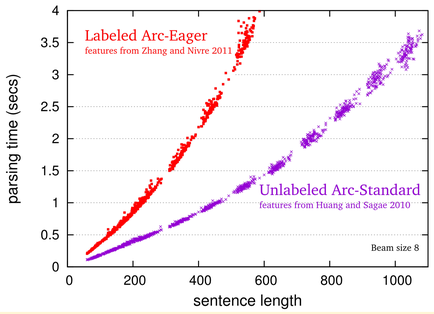
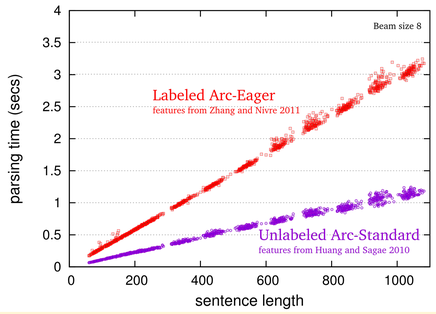
Efficient Implementation of Beam-Search Incremental Parsers

Yoav Goldberg — Bar Ilan University
Kai Zhao, Liang Huang — Graduate Center, CUNY



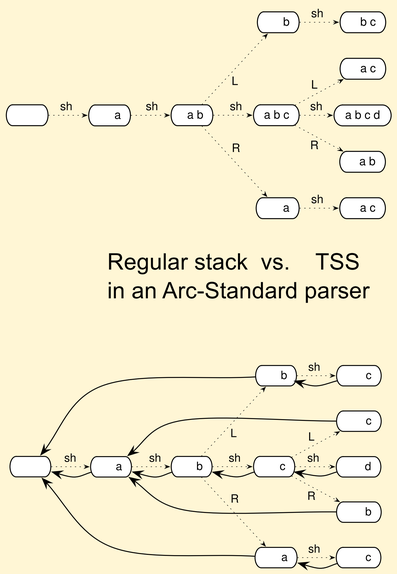
Incremental (transition) parsers advertise linear parsing time. However, implementations of beam-search transition parsers are in fact $O(n^2)$, as can be observed on long sentences.

We show how to implement them in real linear time.



Why are beam-search parsers quadratic?

Beam setup forces *copying* the state object at each transition. In a naive implementation, states are O(n) in size. Because each transition involves a state-copy, **each transition is O(n)**, and n transitions are $O(n^2)$.



Regular stack vs. TSS in an Arc-Standard parser

From O(n) to O(1) state transitions

(1) Naive State representation:

```
stack:
arcs:
last-action:Sh
buffer loc:36
prev-state:
```

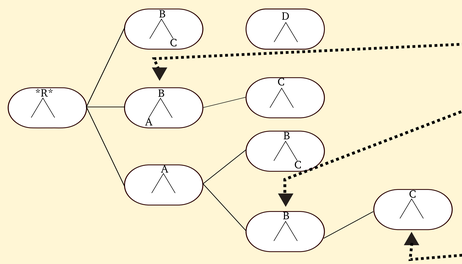
Size O(n) because of stack and arcs.

(2) Get rid of arcs:

```
stack:
last-action:Sh
buffer loc:36
prev-state:
```

Keep left- and right-most modifiers of stack-items on stack.
These are sufficient for features.
The tree is reconstructed by backtracking.

(3) Share the stack
with a persistent data structure (Tree Structured Stack)




```
stack:
last-action:Sh
buffer loc:18
prev-state:
```

```
stack:
last-action:Left
buffer loc:18
prev-state:
```

```
stack:
last-action:Sh
buffer loc:19
prev-state:
```

```
def Shift(s):
  new.stack.tail = s.stack
  new.stack.H = s.buf_loc
  new.stack.L = null
  new.stack.R = null
  new.action = Sh
  new.buf_loc = s.buf_loc + 1
  new.prev_state = s
```

```
def ReduceLeft(s):
  new.stack.tail = s.stack.tail.tail
  new.stack.H = s.stack.H
  new.stack.L = s.stack.tail.H
  new.stack.L = s.stack.R
  new.action = Left
  new.buf_loc = s.buf_loc
  new.prev_state = s
```

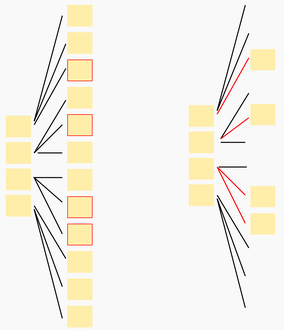
```
def ReduceRight(s):
  new.stack.tail = s.stack.tail.tail
  new.stack.H = s.stack.tail.H
  new.stack.L = s.stack.tail.L
  new.stack.L = s.stack.H
  new.action = Right
  new.buf_loc = s.buf_loc
  new.prev_state = s
```

Efficient State representation:
Size O(1) because of shard-stack and distribution of arcs.
Perform a state transition is O(1).

Further Improvements:

Lazy State Creation

Instead of creating all next states and keeping only the surviving ones, create only states that will survive.

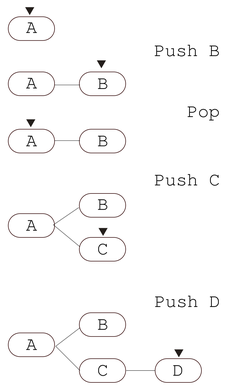


(Partial) Feature Sharing

Define state-signature as a subset of the "core features" in that state.
A signature determines either the complete feature vector, or a chunk of it.

If we have seen a particular signature on the beam already, re-use its outgoing transition scores instead of doing feature extraction and dot-products.

Main Data Structure:
Tree Structured Stack



Parsing speeds (sent/sec) for WSJ sentences:

All parsers implemented in python, and employ a beam of size 8.

system	plain	plain +TSS	plain +lazy	plain +TSS+lazy	+TSS+lazy +feat-sharing
ArcS-U	20.8	38.6	24.3	41.1	47.4
ArcE-U	25.4	48.3	38.2	58.2	72.3
ArcE-L	1.8	4.9	11.1	14.5	17.3