NLP Programming Tutorial 13 - Beam and A* Search

Graham Neubig
Nara Institute of Science and Technology (NAIST)

Prediction Problems

- Given observable information X, find hidden Y

$$\underset{Y}{\operatorname{argmax}}\, P(Y|X)$$

- Used in POS tagging, word segmentation, parsing
- Solving this argmax is “search”
- Until now, we mainly used the Viterbi algorithm

Hidden Markov Models (HMMs) for POS Tagging

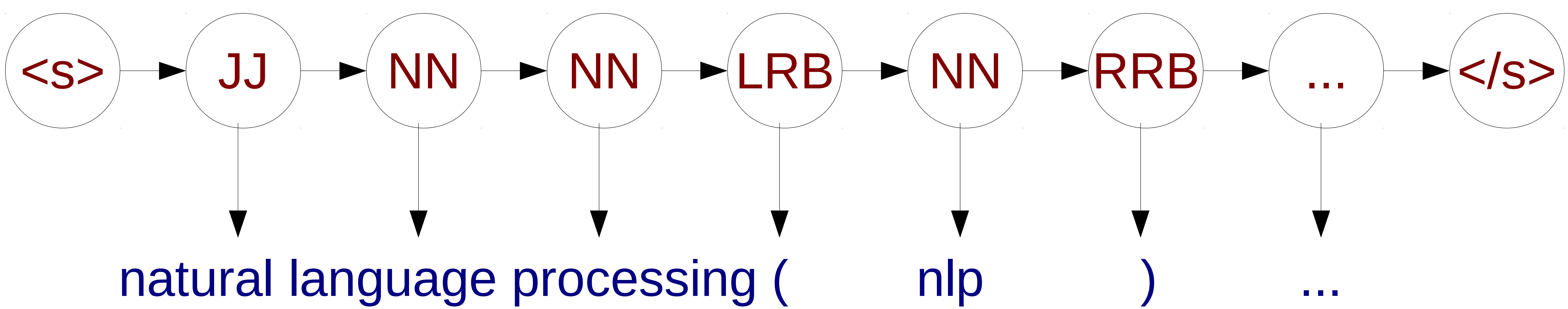
- POS→POS **transition** probabilities
  - Like a bigram model!

$$P(Y) \approx \prod_{i=1}^{I+1} P_T(y_i | y_{i-1})$$

- POS→Word **emission** probabilities

$$P(X|Y) \approx \prod_{1}^{I} P_E(x_i | y_i)$$

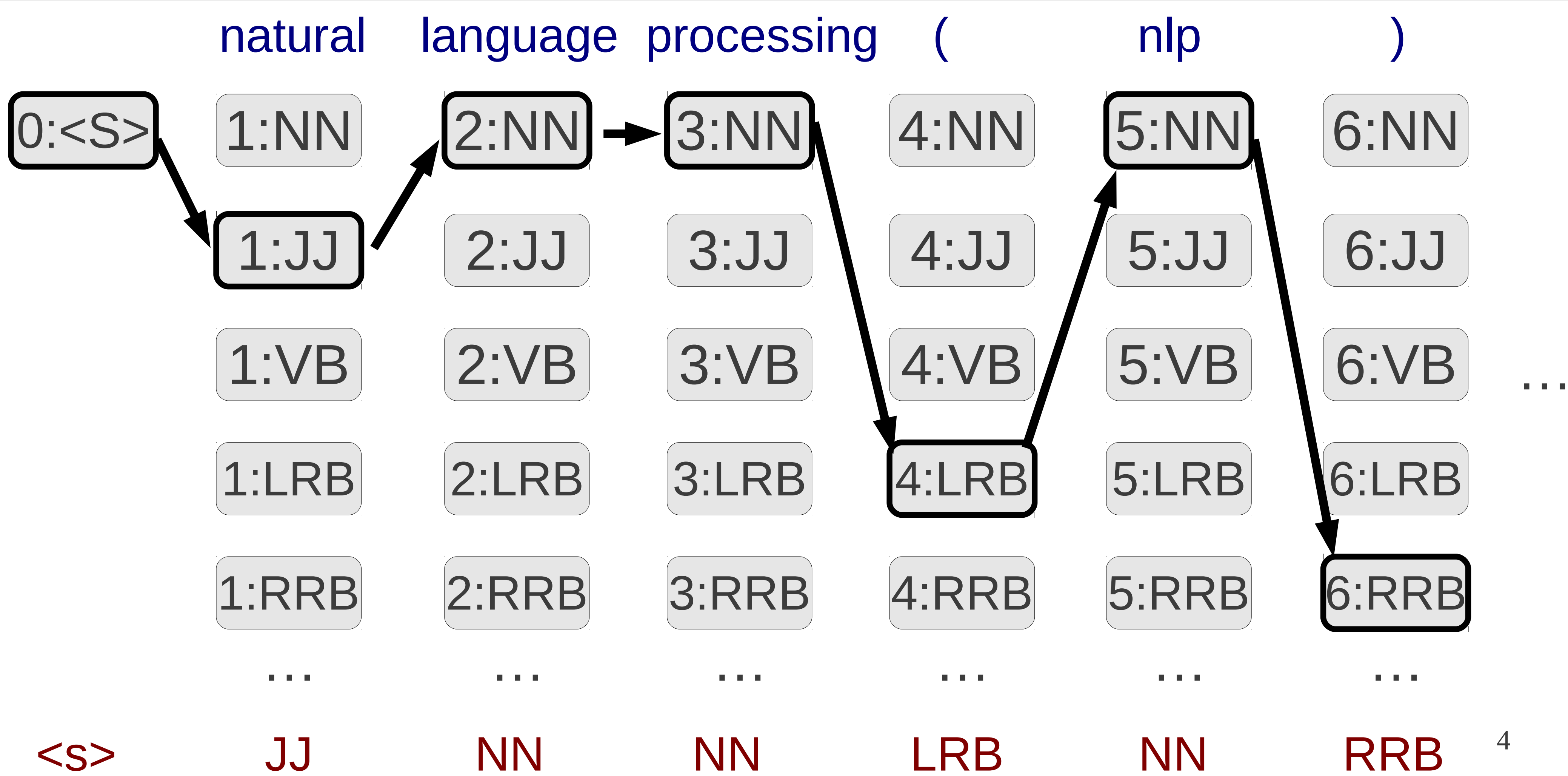
$P_T(JJ|<s>) * P_T(NN|JJ) * P_T(NN|NN)$ …



$P_E(natural|JJ) * P_E(language|NN) * P_E(processing|NN)$ …

Finding POS Tags with Markov Models

- The best path is our POS sequence



Remember: Viterbi Algorithm Steps

- Forward step, calculate the best path to a node
  - Find the path to each node with the lowest negative log probability
- Backward step, reproduce the path
  - This is easy, almost the same as word segmentation

Forward Step: Part 1

- First, calculate transition from <S> and emission of the first word for every POS

natural

0:<S>

1:NN best_score[“1 NN”] = -log P_T(NN|<S>) + -log P_E(natural | NN)

1:JJ best_score[“1 JJ”] = -log P_T(JJ|<S>) + -log P_E(natural | JJ)

1:VB best_score[“1 VB”] = -log P_T(VB|<S>) + -log P_E(natural | VB)

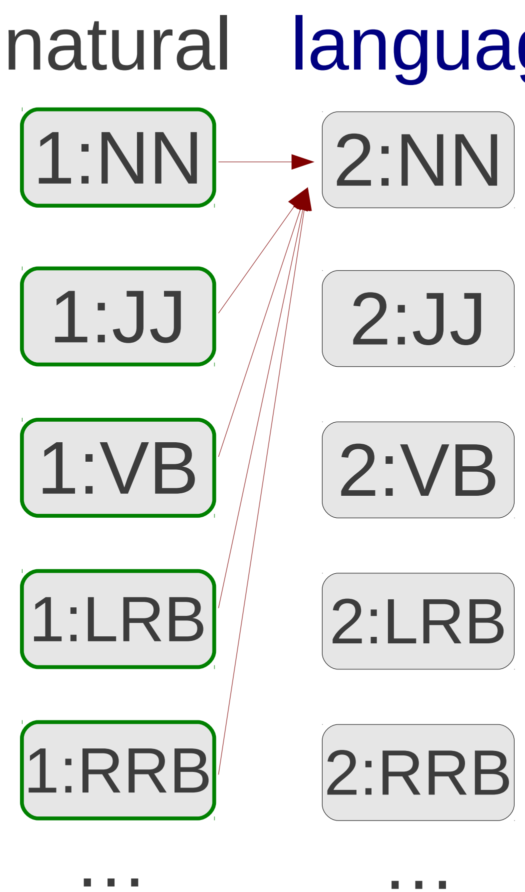
1:LRB best_score[“1 LRB”] = -log P_T(LRB|<S>) + -log P_E(natural | LRB)

1:RRB best_score[“1 RRB”] = -log P_T(RRB|<S>) + -log P_E(natural | RRB)

…

Forward Step: Middle Parts

- For middle words, calculate the minimum score for all possible previous POS tags



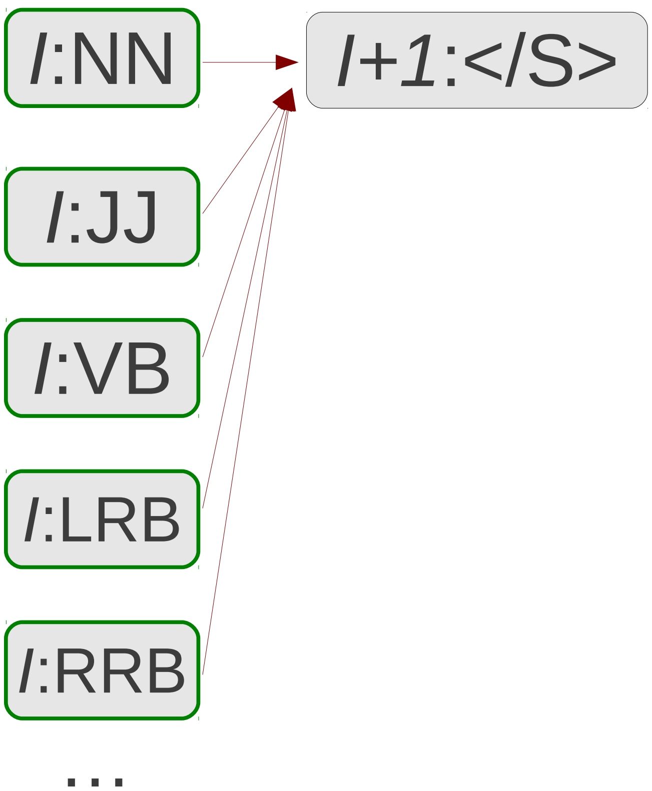
best_score["2 NN"] = min(
 best_score["1 NN"] + -log P_T(NN|NN) + -log P_E(language | NN),
 best_score["1 JJ"] + -log P_T(NN|JJ) + -log P_E(language | NN),
 best_score["1 VB"] + -log P_T(NN|VB) + -log P_E(language | NN),
 best_score["1 LRB"] + -log P_T(NN|LRB) + -log P_E(language | NN),
 best_score["1 RRB"] + -log P_T(NN|RRB) + -log P_E(language | NN),
 ...
)

best_score["2 JJ"] = min(
 best_score["1 NN"] + -log P_T(JJ|NN) + -log P_E(language | JJ),
 best_score["1 JJ"] + -log P_T(JJ|JJ) + -log P_E(language | JJ),
 best_score["1 VB"] + -log P_T(JJ|VB) + -log P_E(language | JJ),
 ...

Forward Step: Final Part

- Finish up the sentence with the sentence final symbol

science



best_score[“*I+1* </S>”] = min(
best_score[“*I* NN”] + -log P_T(</S>|NN),
best_score[“*I* JJ”] + -log P_T(</S>|JJ),
best_score[“*I* VB”] + -log P_T(</S>|VB),
best_score[“*I* LRB”] + -log P_T(</S>|LRB),
best_score[“*I* NN”] + -log P_T(</S>|RRB),
...
)

Viterbi Algorithm and Time

- The time of the Viterbi algorithm depends on:
  - type of problem: POS? Word Segmentation? Parsing?
  - length of sentence: Longer Sentence=More Time
  - number of tags: More Tags=More Time
- What is time complexity of HMM POS tagging?
  - T = Number of tags
  - N = length of sentence

Simple Viterbi Doesn't Scale

- Tagging:
  - Named Entity Recognition:
    T = types of named entities (100s to 1000s)
  - Supertagging:
    T = grammar rules (100s)
- Other difficult search problems:
  - Parsing: $T * N^3$
  - Speech Recognition: (frames)*(WFST states, millions)
  - Machine Translation: NP complete

Two Popular Solutions

- Beam Search:
  - Remove low probability partial hypotheses
  - + Simple, search time is stable
  - - Might not find the best answer
- A* Search:
  - Depth-first search, create a heuristic function of cost to process the remaining hypotheses
  - + Faster than Viterbi, exact
  - - Must be able to create heuristic, search time is not stable

Beam Search

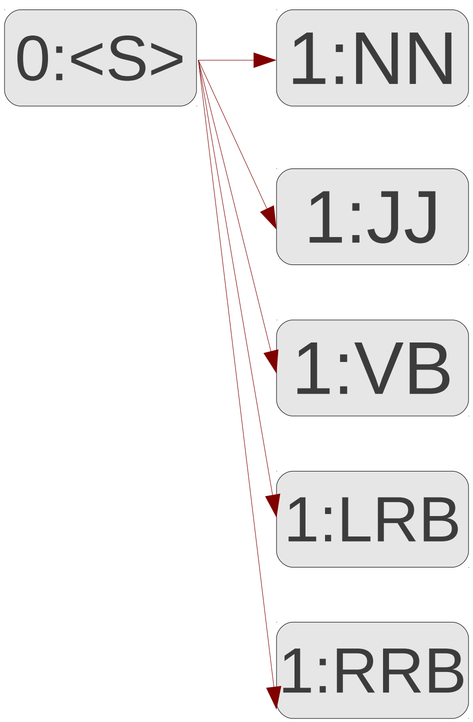
Beam Search

- Choose beam of *B* hypotheses
- Do Viterbi algorithm, but keep only best *B* hypotheses at each step
- Definition of “step” depends on task:
    - Tagging: Same number of words tagged
    - Machine Translation: Same number of words translated
    - Speech Recognition: Same number of frames processed

Calculate Best Scores (First Word)

- Calculate best scores for first word

natural



best_score[“1 NN”] = -3.1

best_score[“1 JJ”] = -4.2

best_score[“1 VB”] = -5.4

best_score[“1 LRB”] = -8.2

best_score[“1 RRB”] = -8.1

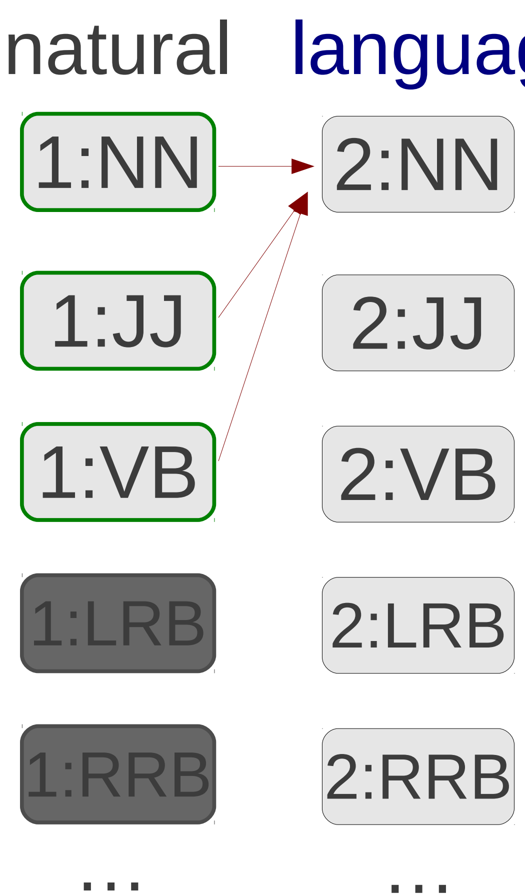
…

Keep Best B Hypotheses (w_1)

- Remove hypotheses with low scores
- For example, B=3

Calculate Probabilities (w_2)

- Calculate score, but ignore removed hypotheses



best_score[“2 NN”] = min(
best_score[“1 NN”] + -log P_T(NN|NN) + -log P_E(language | NN),
best_score[“1 JJ”] + -log P_T(NN|JJ) + -log P_E(language | NN),
best_score[“1 VB”] + -log P_T(NN|VB) + -log P_E(language | NN),
~~best_score[“1 LRB”] + -log P_T(NN|LRB) + -log P_E(language | NN),~~
~~best_score[“1 RRB”] + -log P_T(NN|RRB) + -log P_E(language | NN),~~
...
)

best_score[“2 JJ”] = min(
best_score[“1 NN”] + -log P_T(JJ|NN) + -log P_E(language | JJ),
best_score[“1 JJ”] + -log P_T(JJ|JJ) + -log P_E(language | JJ),
best_score[“1 VB”] + -log P_T(JJ|VB) + -log P_E(language | JJ),
...

Beam Search is Faster

- Remove some candidates from consideration → faster speed!
- What is the time complexity?
  - T = Number of tags
  - N = length of sentence
  - B = beam width

Implementation: Forward Step

best_score[“0 <s>”] = 0 # Start with <s>
best_edge[“0 <s>”] = NULL
active_tags[0] = [“<s>”]
for *i* **in** 0 … *I*-1:
    make map *my_best*
    for each *prev* **in** keys of *active_tags*[*i*]
        for each *next* **in** keys of *possible_tags*
            if best_score[“*i prev*”] **and** transition[“prev next”] **exist**
                score = best_score[“i prev”] +
                    -log P_T(next|prev) + -log P_E(word[i]|next)
                if *best_score*[“*i+1 next*”] **is** *new* **or** > *score*
                    best_score[“*i+1 next*”] = *score*
                    best_edge[“*i+1 next*”] = “*i prev*”
                    my_best[*next*] = *score*
    active_tags[*i+1*] = best *B* elements of *my_best*
Finally, do the same for </s>

A* Search

Depth-First Search

- Always expand the state with the highest score
- Use a heap (priority queue) to keep track of states
  - heap: a data structure that can add elements in O(1) and find the highest scoring element in time O(log n)
  - Start with only the initial state on the heap
  - Expand the best state on the heap until search finishes
- Compare with breadth-first search, which expands states at the same step (Viterbi, beam search)

Depth-First Search

- Initial state:

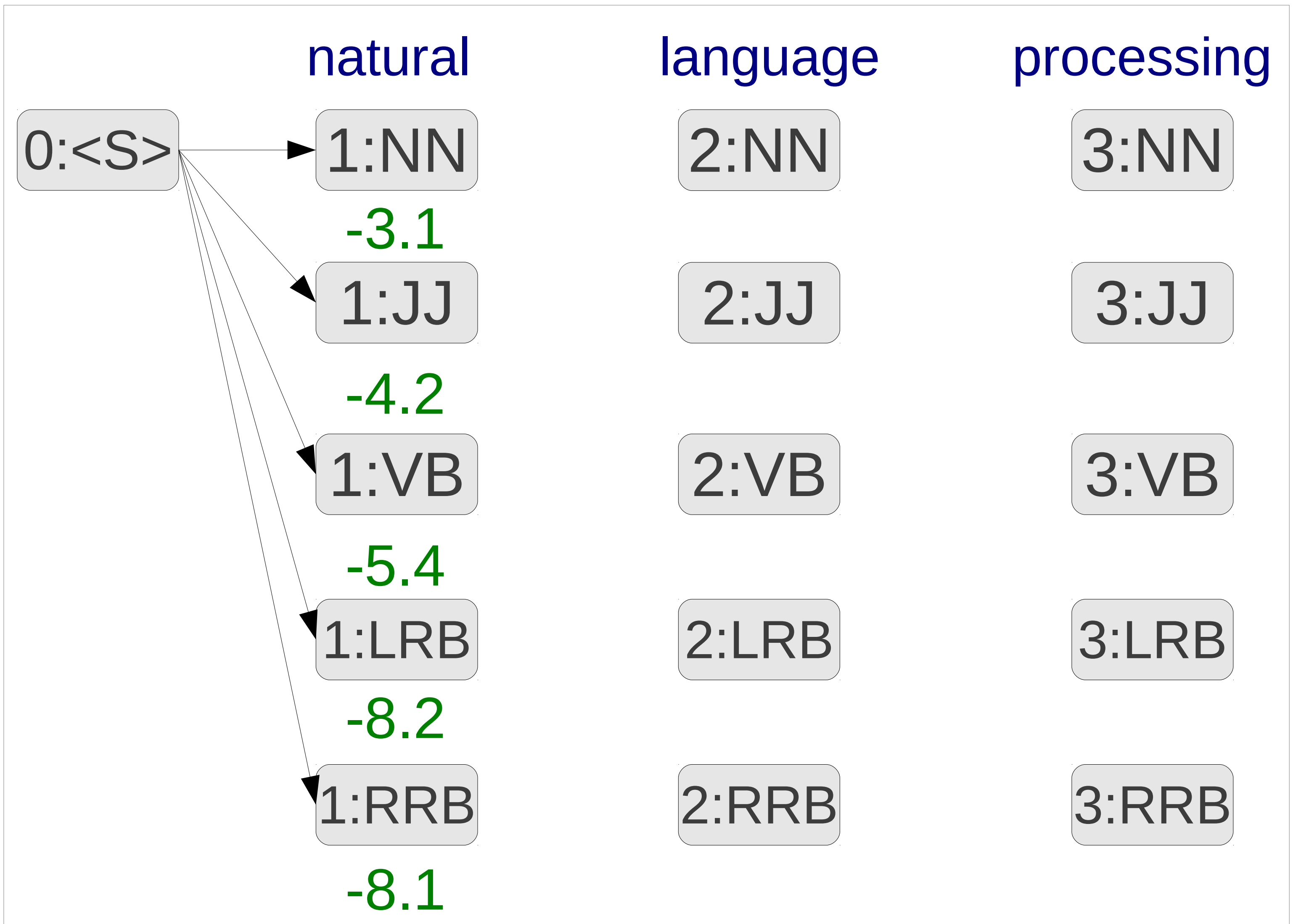
	natural	language	processing
0:<S>	1:NN	2:NN	3:NN
	1:JJ	2:JJ	3:JJ
	1:VB	2:VB	3:VB
	1:LRB	2:LRB	3:LRB
	1:RRB	2:RRB	3:RRB

Heap

0:<S> 0

Depth-First Search

- Process 0:<S>

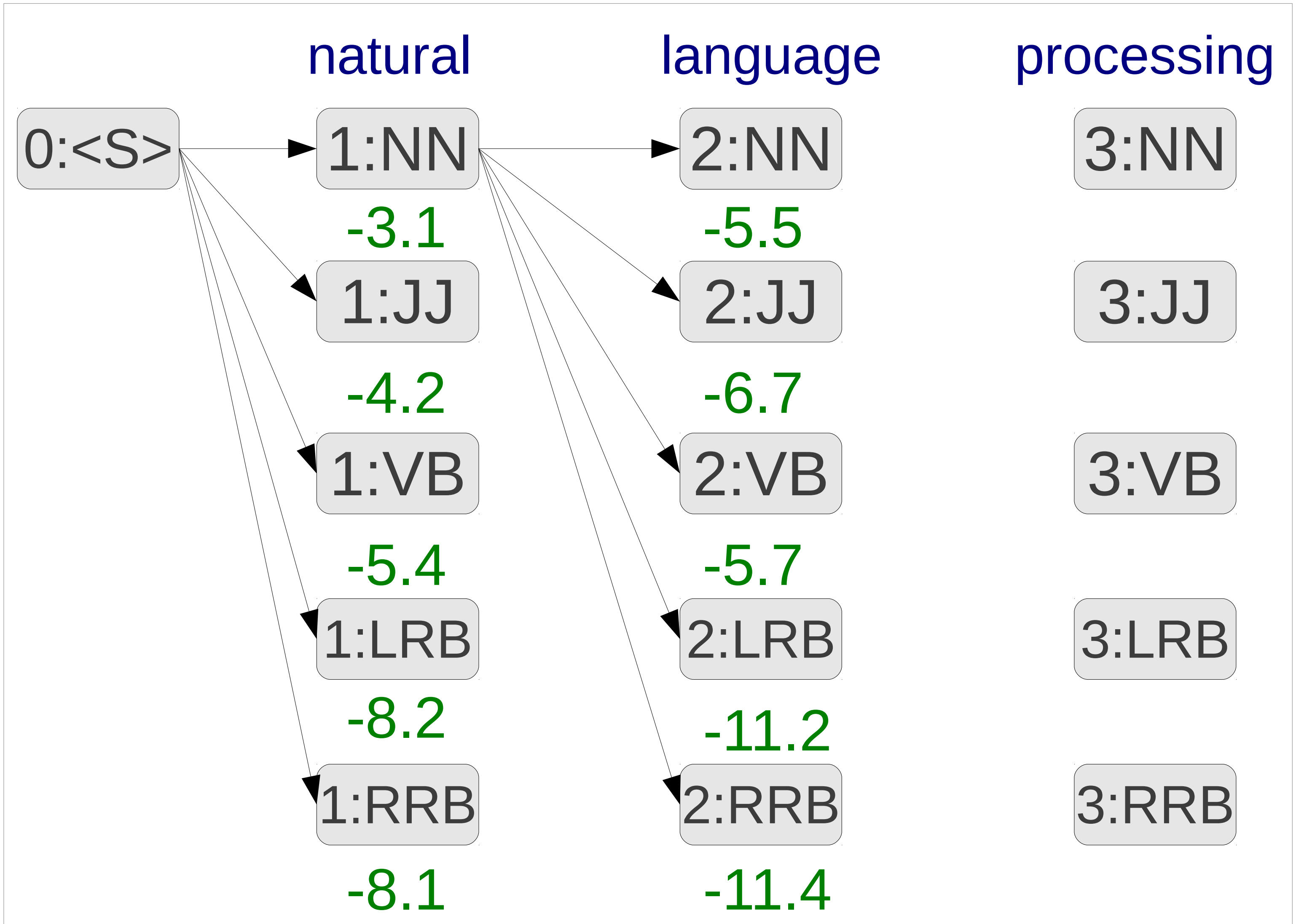


<u>Heap</u>

1:NN -3.1
1:JJ -4.2
1:VB -5.4
1:RRB -8.1
1:LRB -8.2

Depth-First Search

- Process 1:NN

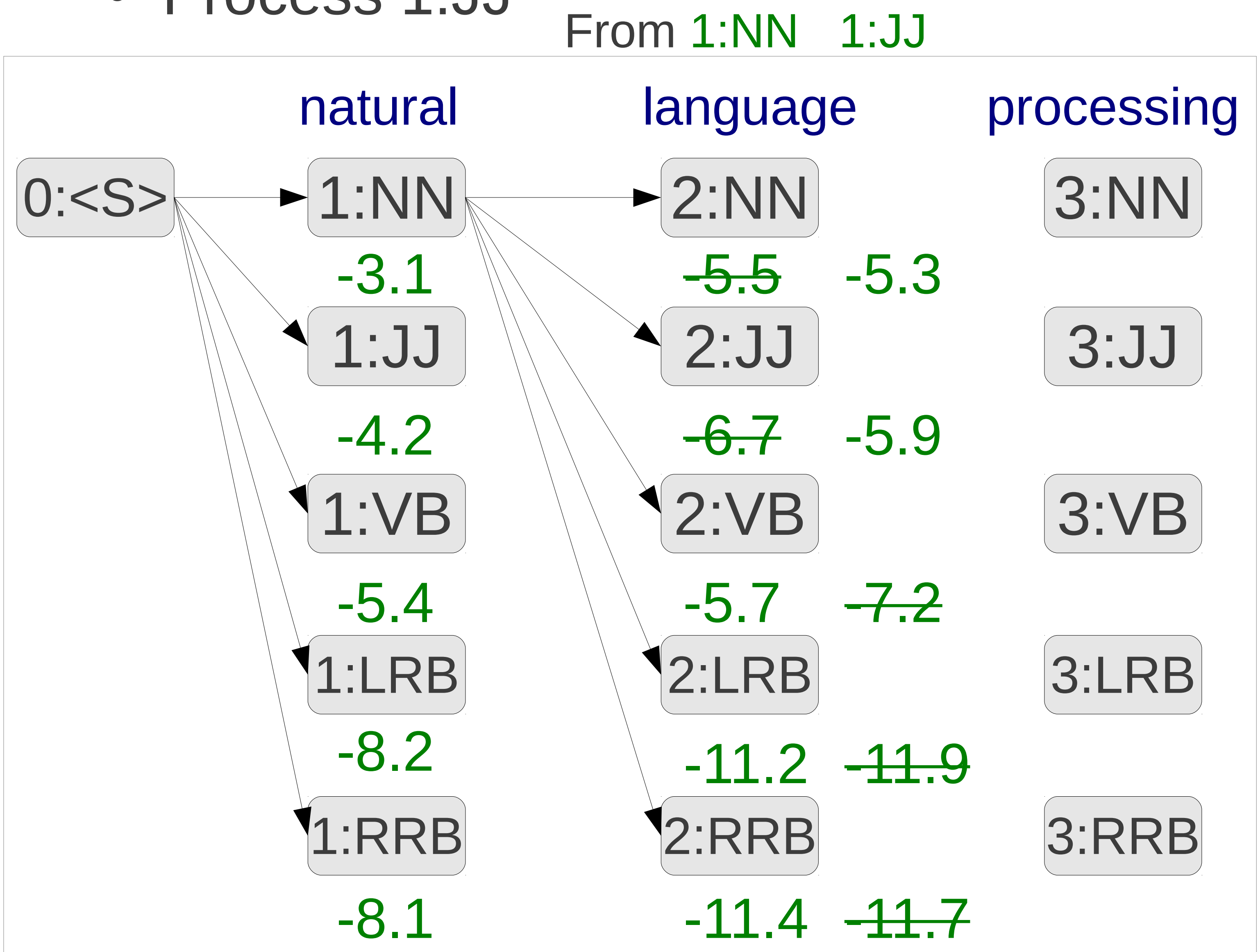


<u>Heap</u>

1:JJ -4.2
1:VB -5.4
2:NN -5.5
2:VB -5.7
2:JJ -6.7
1:RRB -8.1
1:LRB -8.2
2:LRB -11.2
2:RRB -11.4

Depth-First Search

- Process 1:JJ

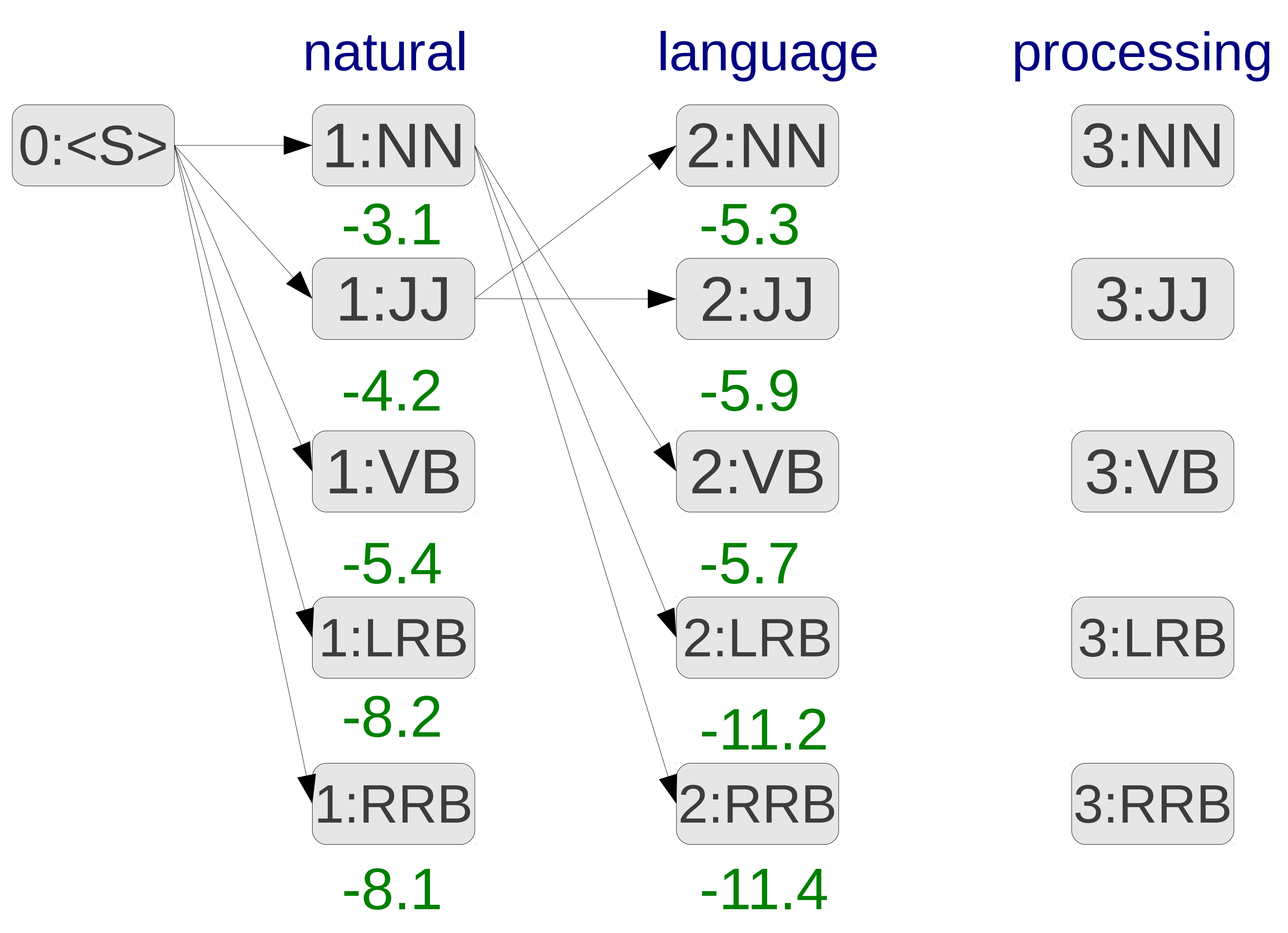


Heap

2:NN -5.3
1:VB -5.4
2:NN -5.5
2:VB -5.7
2:JJ -5.9
2:JJ -6.7
1:RRB -8.1
1:LRB -8.2
2:LRB -11.2
2:RRB -11.4

Depth-First Search

- Process 1:JJ

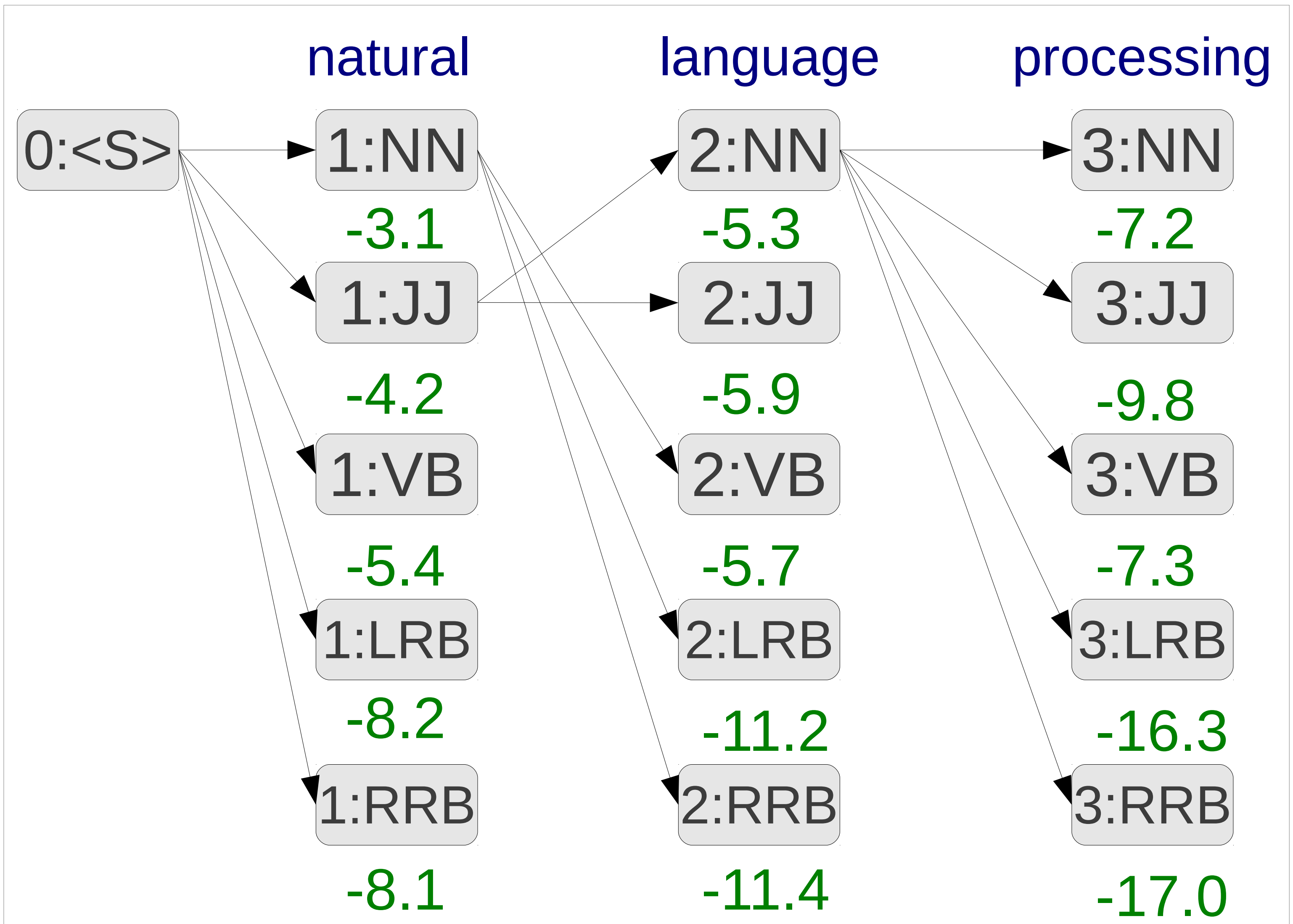


Heap

2:NN -5.3
1:VB -5.4
2:NN -5.5
2:VB -5.7
2:JJ -5.9
2:JJ -6.7
1:RRB -8.1
1:LRB -8.2
2:LRB -11.2
2:RRB -11.4

Depth-First Search

- Process 2:NN

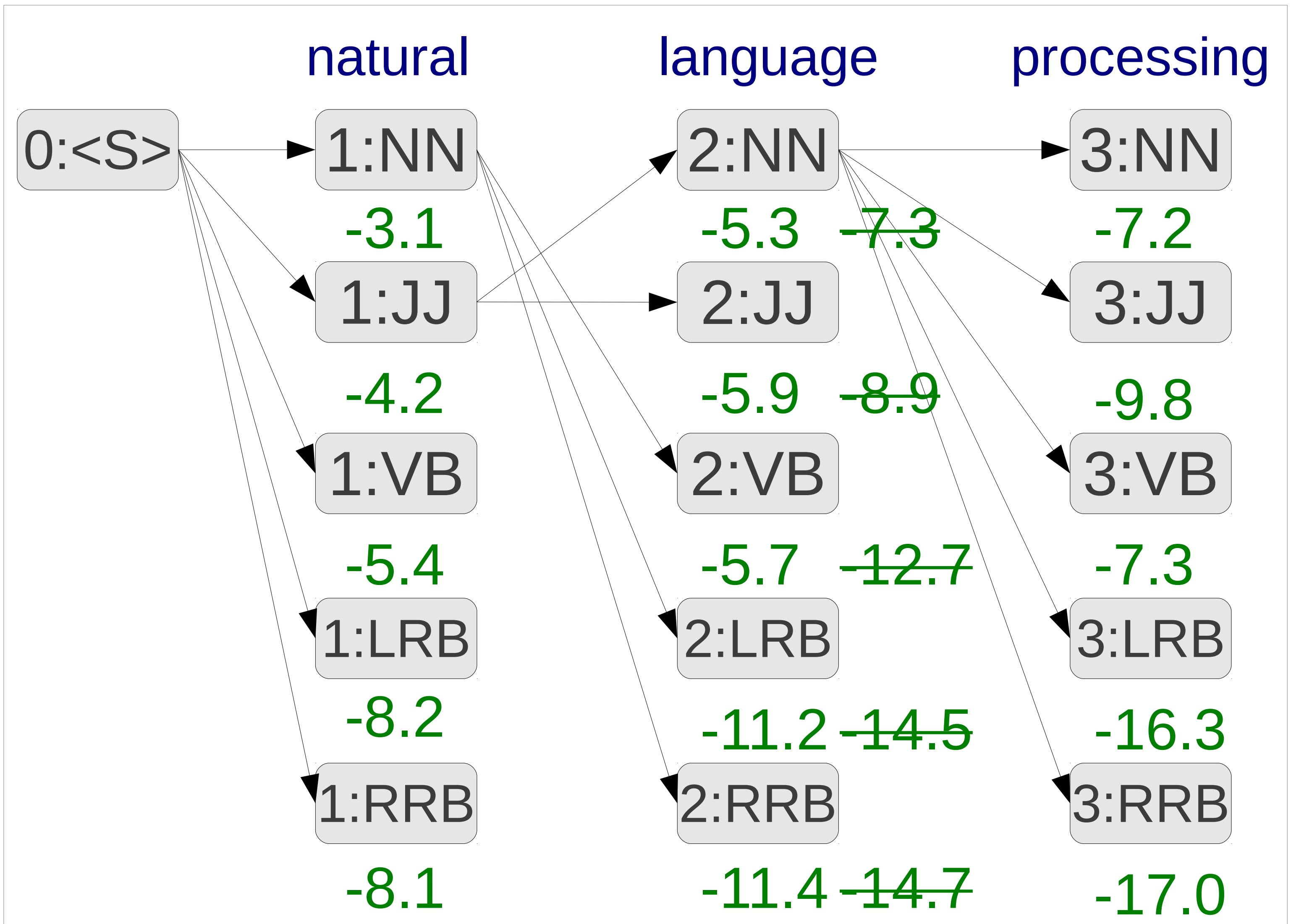


<u>Heap</u>

1:VB -5.4
2:NN -5.5
2:VB -5.7
2:JJ -5.9
2:JJ -6.7
3:NN -7.2
3:VB -7.3
1:RRB -8.1
1:LRB -8.2
3:JJ -9.8
2:LRB -11.2
...

Depth-First Search

- Process 1:VB

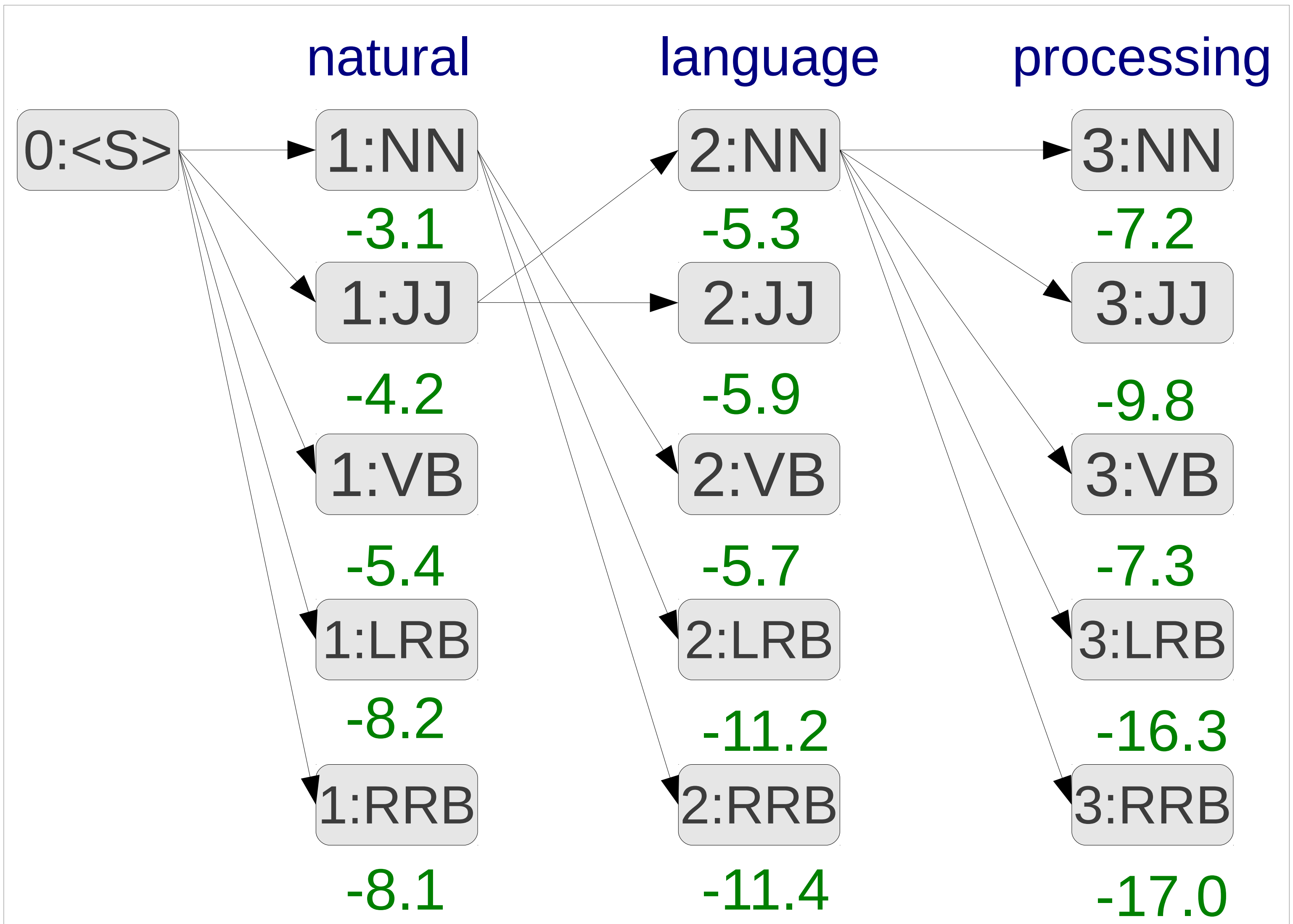


<u>Heap</u>

2:NN -5.5
2:VB -5.7
2:JJ -5.9
2:JJ -6.7
3:NN -7.2
3:VB -7.3
1:RRB -8.1
1:LRB -8.2
3:JJ -9.8
2:LRB -11.2
...

Depth-First Search

- <u>Do not</u> process 2:NN (has already been processed)



<u>Heap</u>

2:VB -5.7
2:JJ -5.9
2:JJ -6.7
3:NN -7.2
3:VB -7.3
1:RRB -8.1
1:LRB -8.2
3:JJ -9.8
2:LRB -11.2
...

Problem: Still Inefficient

- Depth-first search does not work well for long sentences
- Why?
  - Hint: Think of 1:VB in previous example

A* Search: Add Optimistic Heuristic

- Consider the words remaining
- Use Optimistic Heuristic: BEST score possible
- Optimistic heuristic for tagging: Best Emission Prob

natural	language	processing
log(P(natural|NN)) = -2.4	<u>log(P(lang.|NN)) = -2.4</u>	log(P(proc.|NN)) = -2.5
<u>log(P(natural|JJ)) = -2.0</u>	log(P(lang.|JJ)) = -3.0	log(P(proc.|JJ)) = -3.4
log(P(natural|VB)) = -3.1	log(P(lang.|VB)) = -3.2	<u>log(P(proc.|VB)) = -1.5</u>
log(P(natural|LRB)) = -7.0	log(P(lang.|LRB)) = -7.9	log(P(proc.|LRB)) = -6.9
log(P(natural|RRB)) = -7.0	log(P(lang.|RRB)) = -7.9	log(P(proc.|RRB)) = -6.9

H(1+) = -5.9 H(2+) = -3.9 H(3+) = -1.5 H(4+) = 0.0

A* Search: Add Optimistic Heuristic

- Use Forward Score + Optimistic Heuristic

Regular Heap

2:VB	F(2:VB)=-5.7	H(3+)=-1.5
2:JJ	F(2:JJ)=-5.9	H(3+)=-1.5
2:JJ	F(2:JJ)=-6.7	H(3+)=-1.5
3:NN	F(3:NN)=-7.2	H(4+)=-0.0
3:VB	F(3:VB)=-7.3	H(4+)=-0.0
1:RRB	F(1:RRB)=-8.1	H(2+)=-3.9
1:LRB	F(1:LRB)=-8.2	H(2+)=-3.9
3:JJ	F(3:JJ)=-9.8	H(4+)=-0.0
2:LRB	F(2:LRB)=-11.2	H(3+)=-1.5

A* Heap

3:NN	-7.2
2:VB	-7.2
3:VB	-7.3
2:JJ	-7.4
2:JJ	-8.2
3:JJ	-9.8
1:RRB	-12.0
1:LRB	-12.1
2:LRB	-12.7

Exercise

Exercise

- Write test-hmm-beam
- Test the program
    - Input: `test/05-{train,test}-input.txt`
    - Answer: `test/05-{train,test}-answer.txt`
- Train an HMM model on `data/wiki-en-train.norm_pos` and run the program on `data/wiki-en-test.norm`
- Measure the accuracy of your tagging with `script/gradepos.pl data/wiki-en-test.pos my_answer.pos`
- Report the accuracy for different beam sizes
- Challenge: implement A* search

Thank You!