Natural Language Processing

Homework 3

Ivan Christian
1003056

1. Language Model

Part 1

Prove that

$$p(D) = \frac{count(a, b)}{count(a)}$$

Given that

$$w1, w2, w3, w4, \ldots, wn$$

are sequences of words in the sentence, we want to have the probability to predict the next word given the current word. As such we are able to get this criteria that we want, which is

$$p(w_{n-1}|w_n)$$

However, every current and next words are dependent on previous words that results in the following:

$$p(w_{n-1}|w_1, w_2, w_3, \ldots, w_n)$$

The bigram model, however, approximates the probability of a word given bigram all the previous words,

$$P(w_n|w_1^{n-1})$$

Using the history approximation we can get the criteria

$$p(w_n|w_{n-1})$$

as the bigram model would assume such that it is only dependent on the previous word instead of the whole history (Markov assumptions).
The intuitive way to maximise a probability is to employ the **Maximum Likelihood Estimation** which includes counting the appearance of words in the sentence. This would get us

$$p(D) = \frac{count(a,b)}{count(a)}$$

as the probability calculation and hence proving that the

$$p(w_i = b|w_{i-1} = a) = p(D) = \frac{count(a,b)}{count(a)}$$

Part 2

Training Corpus:
〈START〉 John loves Mary 〈END〉
〈START〉 Mary likes NLP 〈END〉

All the count of each word in the training corpus

```
count(<START>) = 2
count(<END>) = 2
count(John) = 1
count(Mary) = 2
count(loves) = 1
count(likes) = 1
count(NLP) = 1
```

All the probability for the bigram

```
count(<START>,John)/count(<START>) = 1/2
count(John, loves)/count(John) = 1/1
count(loves, Mary)/count(loves) = 1/1
count(Mary, <END>)/count(Mary) = 1/2
count(<START>, Mary)/count(<START>) = 1/2
count(Mary, likes)/count(Mary) = 1/2
count(likes, NLP)/count(likes) = 1/1
count(NLP,<END>)/count(NLP) = 1/1
```

Part 3

To accomodate to the missing `[John, <END>]` case we would need to check for unknown cases and label them as `<unk>`. This is done so that the unknown cases are not automatically set to zero and this can be done by smoothing the probability. Smoothing the probability would mean adding 1 to both the denominator and the numerator to take into account unknown cases such as `[John, <END>]`. This can be done in trhough the following

$$p(D) = \frac{count(a, b) + 1}{count(a) + 1}$$

Through this, it would be highly unlikely to have a 0 probability for unknown cases.

If we are to take into account the unknowns using interpolation between bigrams and unigrams, the following can be done to accomodate the new unseen bigram. Assuming `a = John` and `b = <END>` in this case.

Under Markov assumption bigram can be approximated to be

$$bigram = \frac{count(a,b)}{count(a)}$$

(as seen in Part 1)

We also know that a unigram will be assumed to have the following probability

$$unigram = \frac{count(a)}{n}$$

where n is the total number of words in the corpus.

What we want is to get the bigram approximation for the new unseen bigram which would require us to count the number of appearance of (a,b) to get

$$count(a,b)$$

This is dependent on the number of appearance in the test sentence.

By rearranging the unigram expression we can get

$$count(a) = unigram(a) * n$$

This would mean that so long as the word `a` exists in the corpus it is possible to get count(a). This allows us to obtain the necessary elements for a preliminary calculation for the bigram without taking in

$$pre - bigram(a,b) = \frac{count(a,b)}{unigram(a) * n}$$

However, since count(a) and count(b) increases due to the unknown we need to rebalance to probability to take into acccount the increase in number of appearance of a and b. This can be done through the following

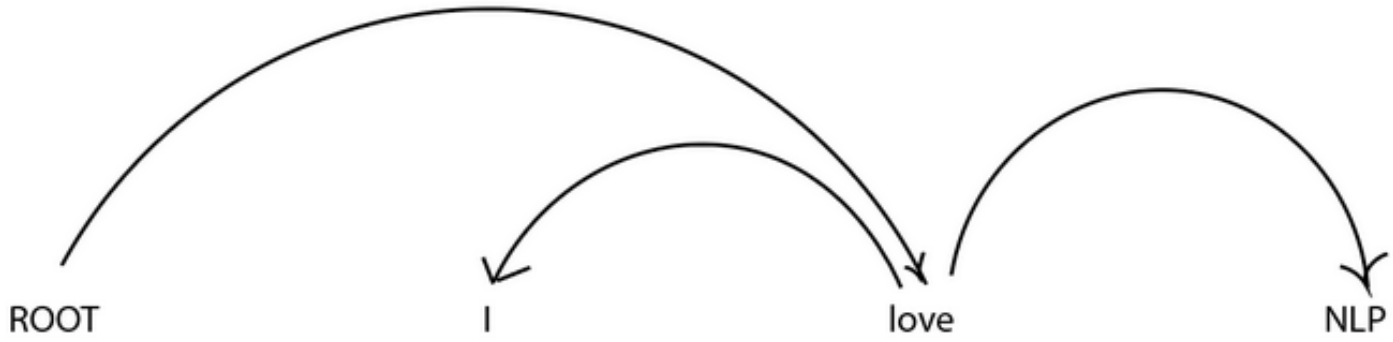
$$bigram(a,b) = \frac{pre - bigram(a,b)}{\sum_{i=1}^{k} bigram(a, w_i)}$$

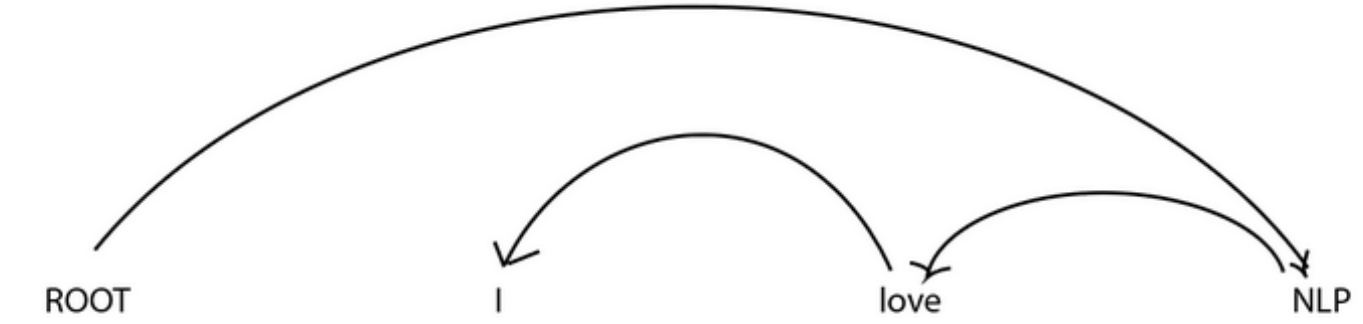
where k is the total number of unique words in the corpus.

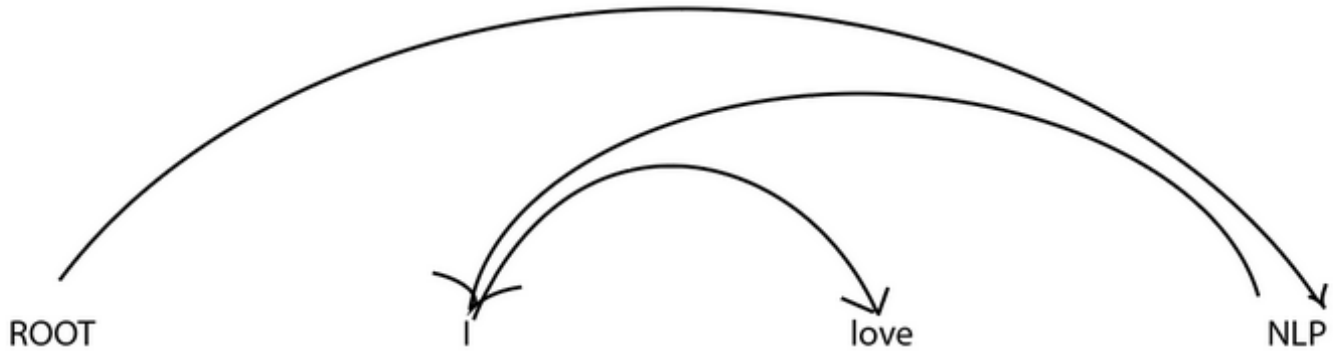
which would yield us the new badded bigram for the unknown.

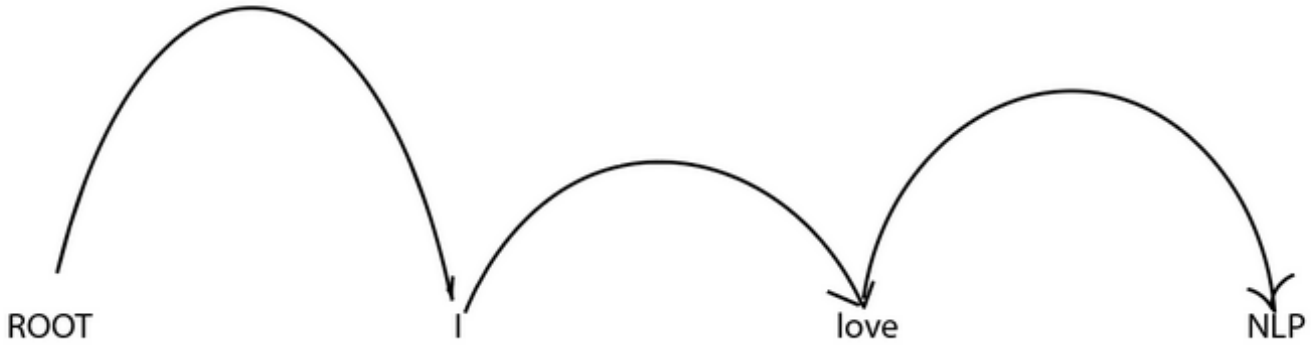
2. Dependency Parsing

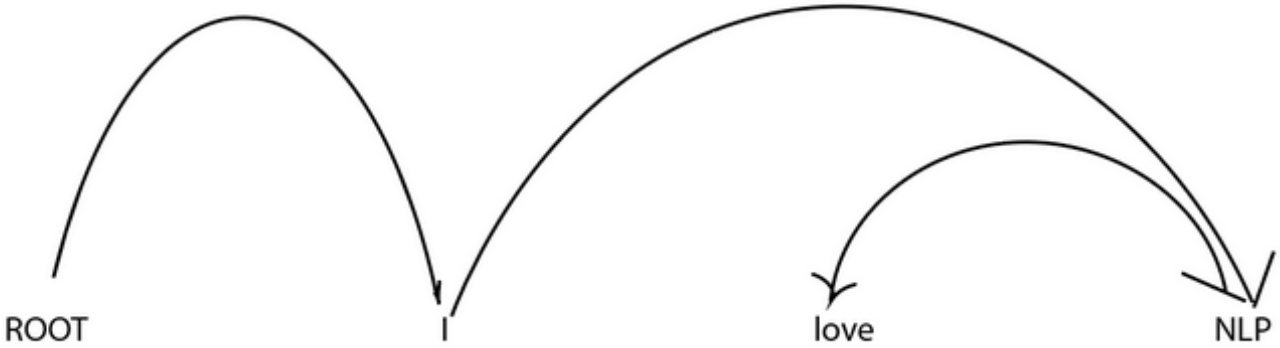
Part 1 (Projective)

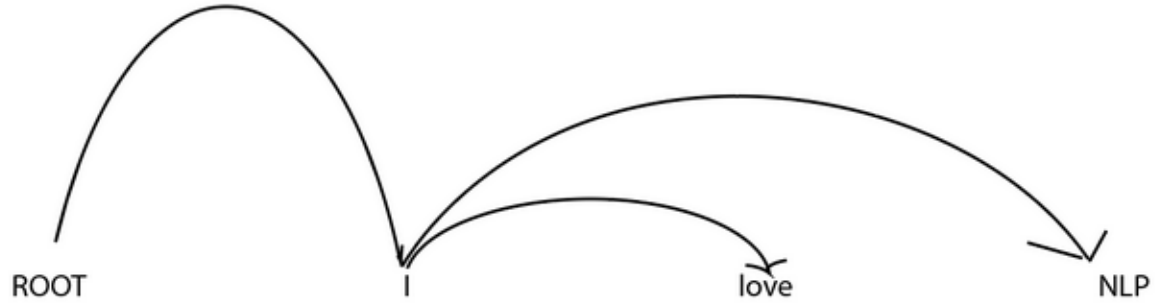


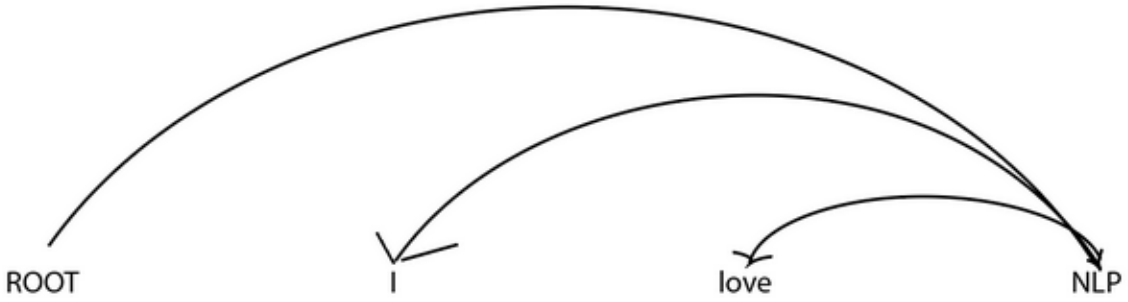

ROOT
I
love
NLP


ROOT
I
love
NLP

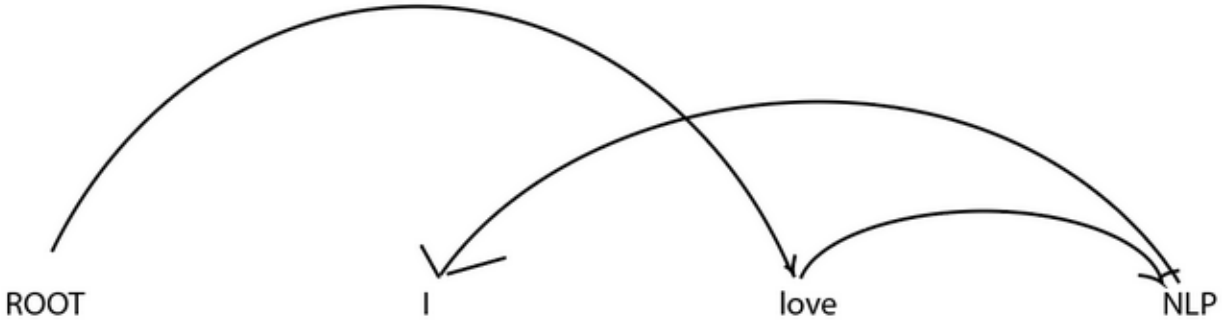

ROOT
I
love
NLP

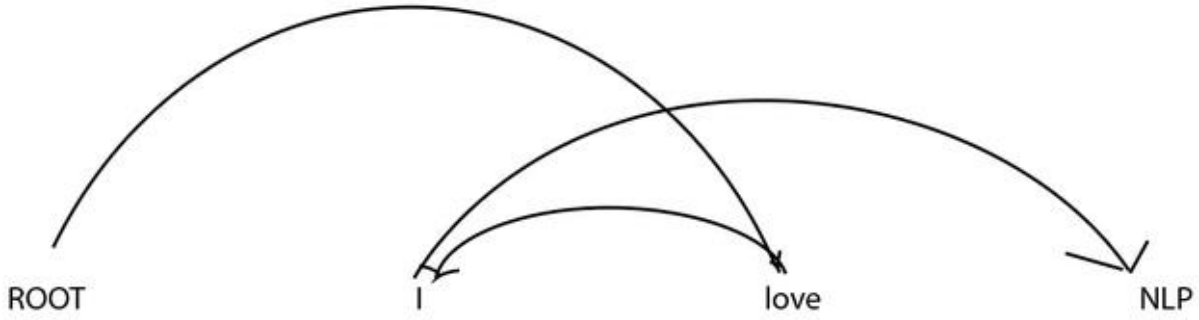

ROOT
I
love
NLP





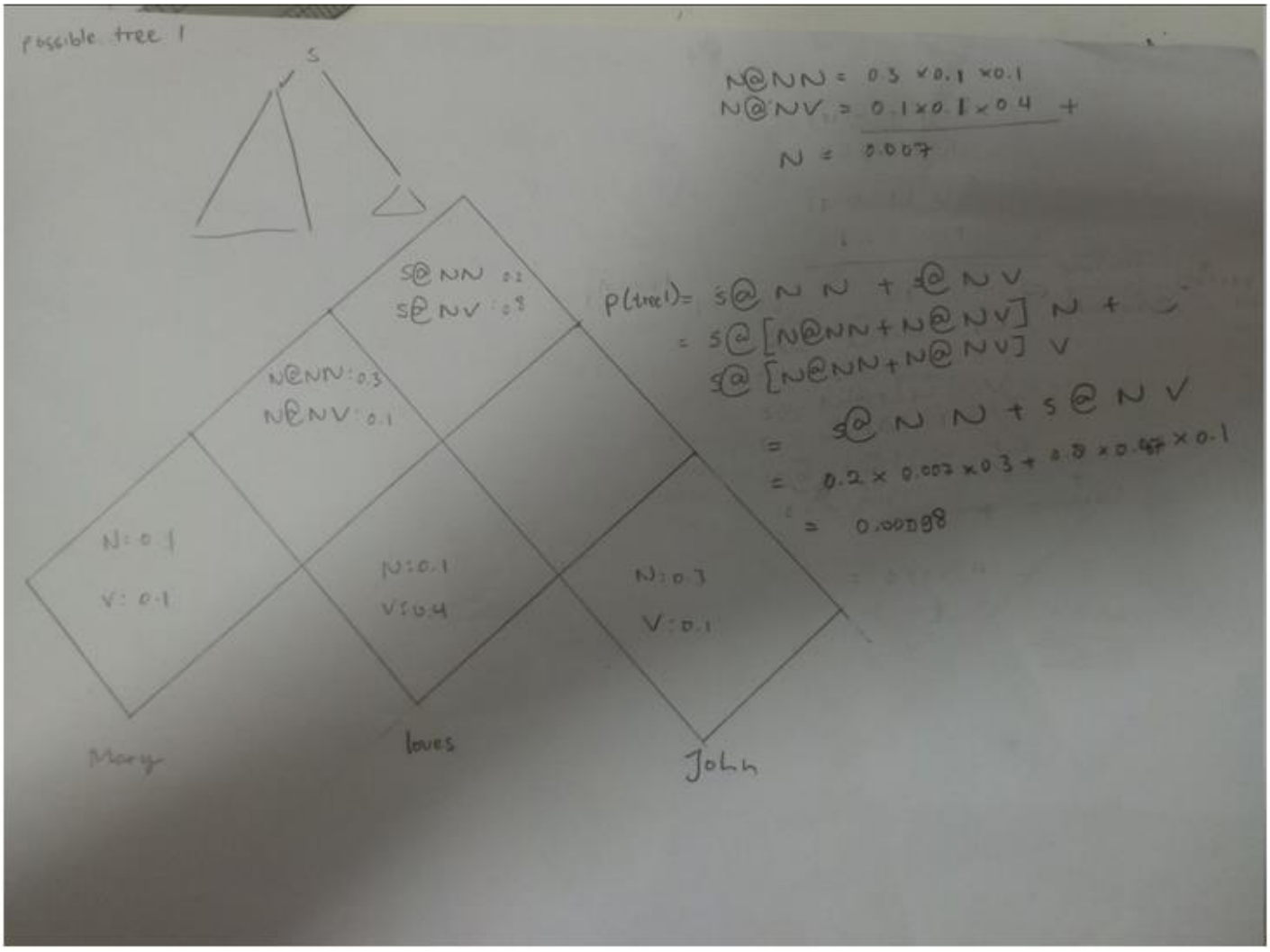
Part 2 (Non Projective)

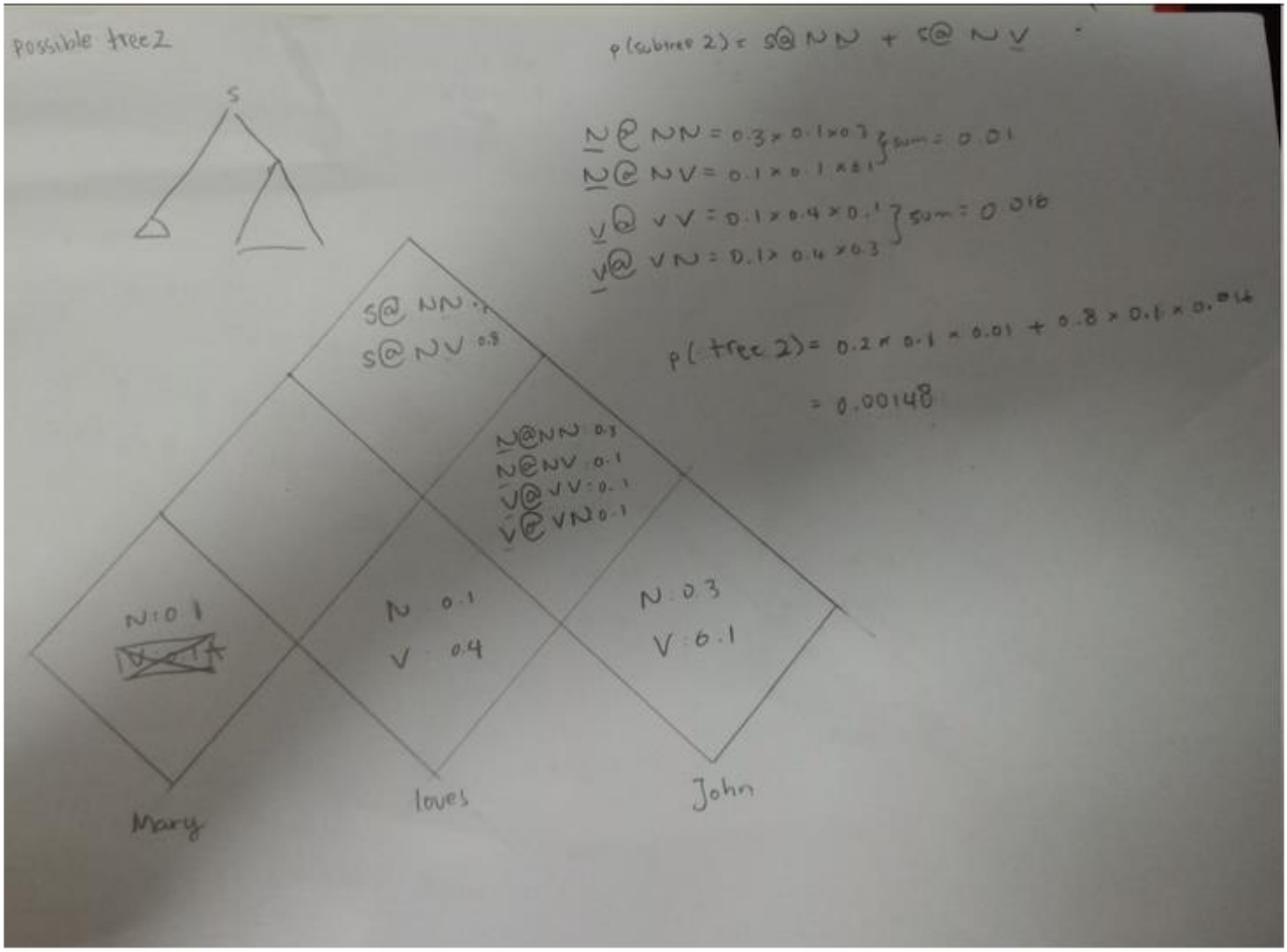




3. Context Free Grammar

Part 1





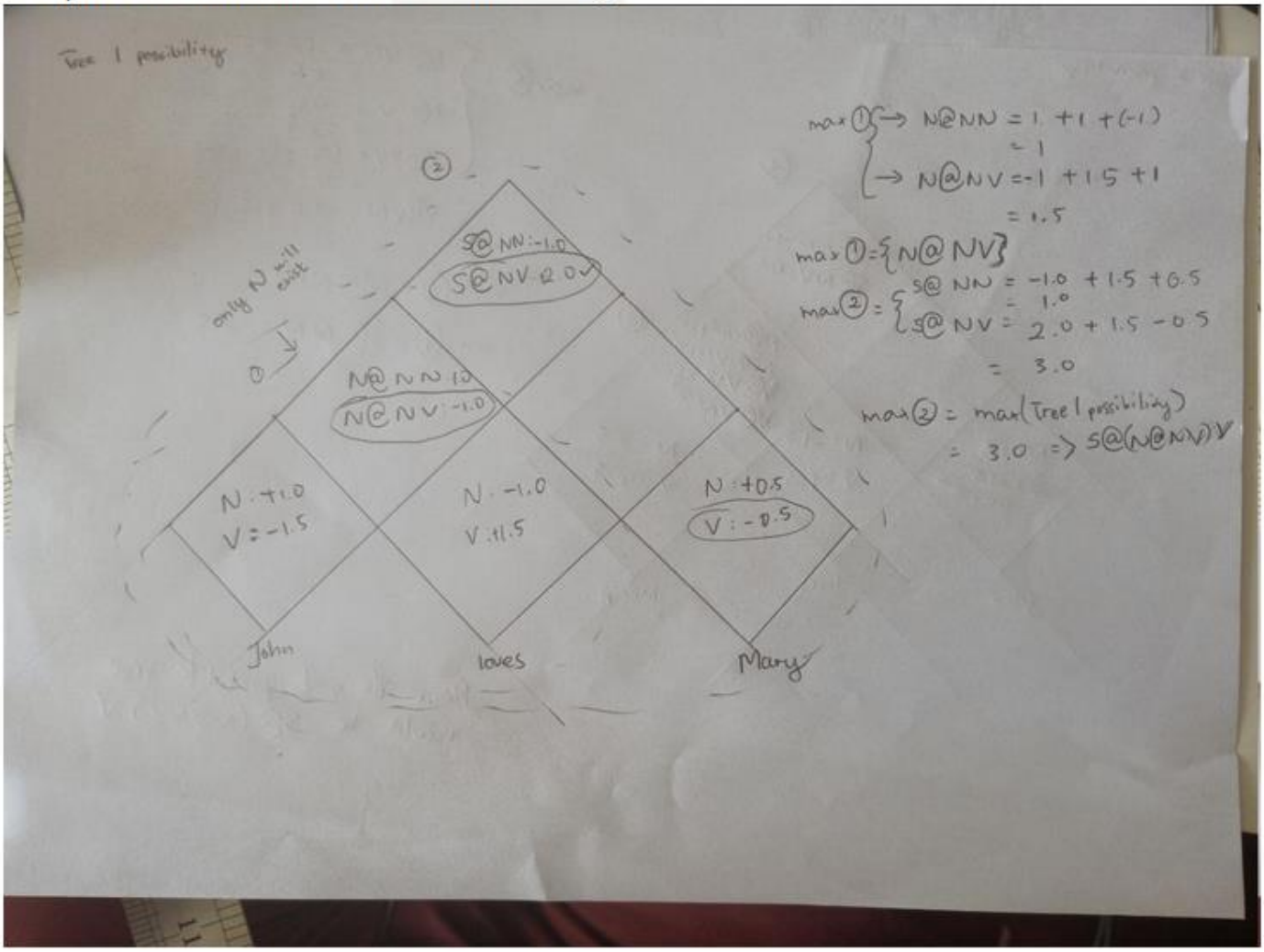
$$p(\text{`}MarylovesJohn\text{`}) = 0.00098 + 0.00148$$

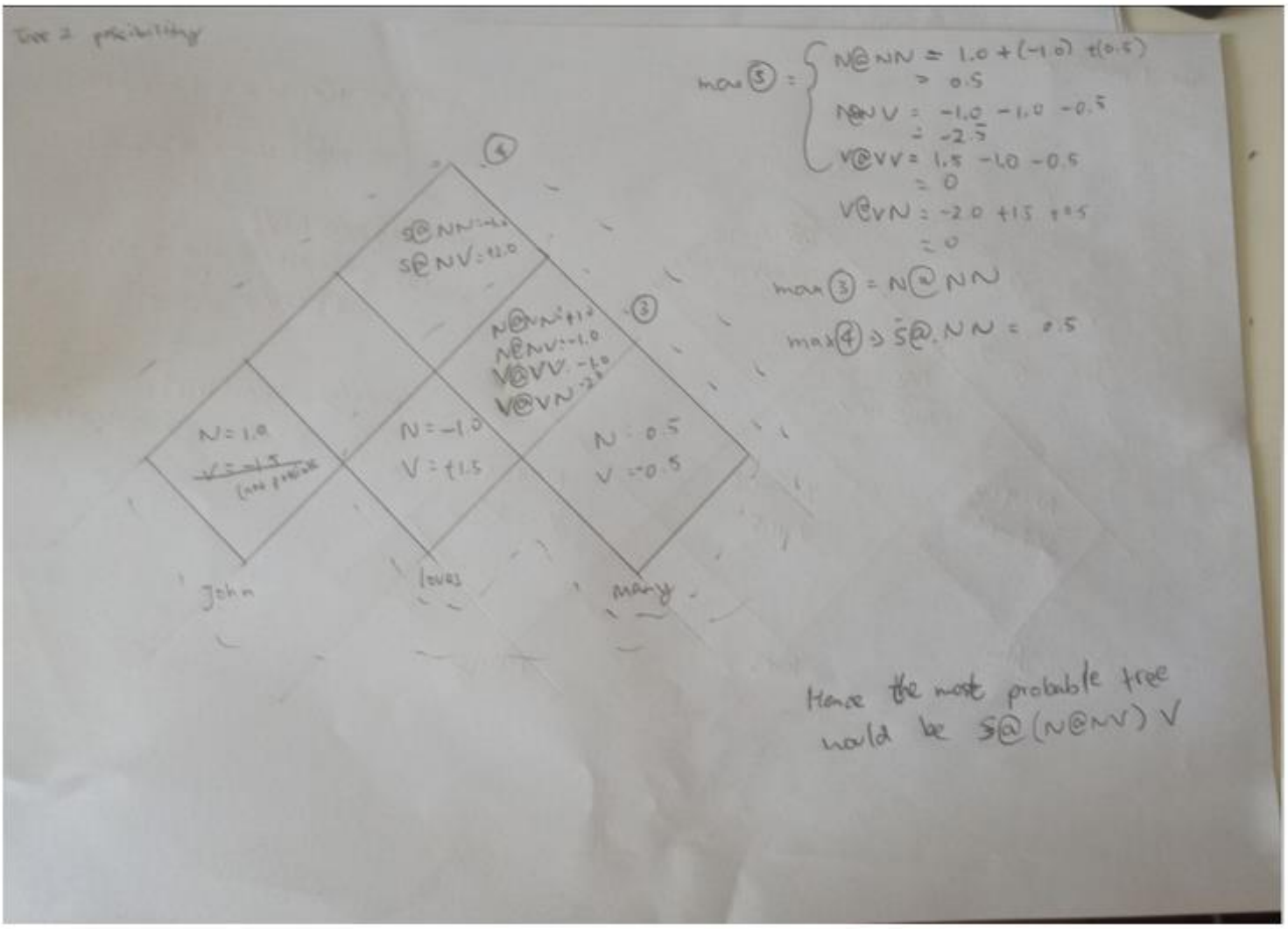
$$p(\text{" } MarylovesJohn \text{ "}) = 0.00246$$

Part 2

Instead of multiplying in the usual CKY algorithm to find the max tree probability, you instead sum it. Doing so would make it similar to sum the log probabilities of probability in a vanilla CKY algorithm. This would achieve the same result fo maximising the probability to get the best possible weight for the algorithm. This would maximise the score using the weights corresponding to the nodes.

Best possible combination for `John loves Mary` :





Based on the weights given and the calculation we know that hte maximum possible weights that can be gotten is from tree 1 which makes the score of 3.0.

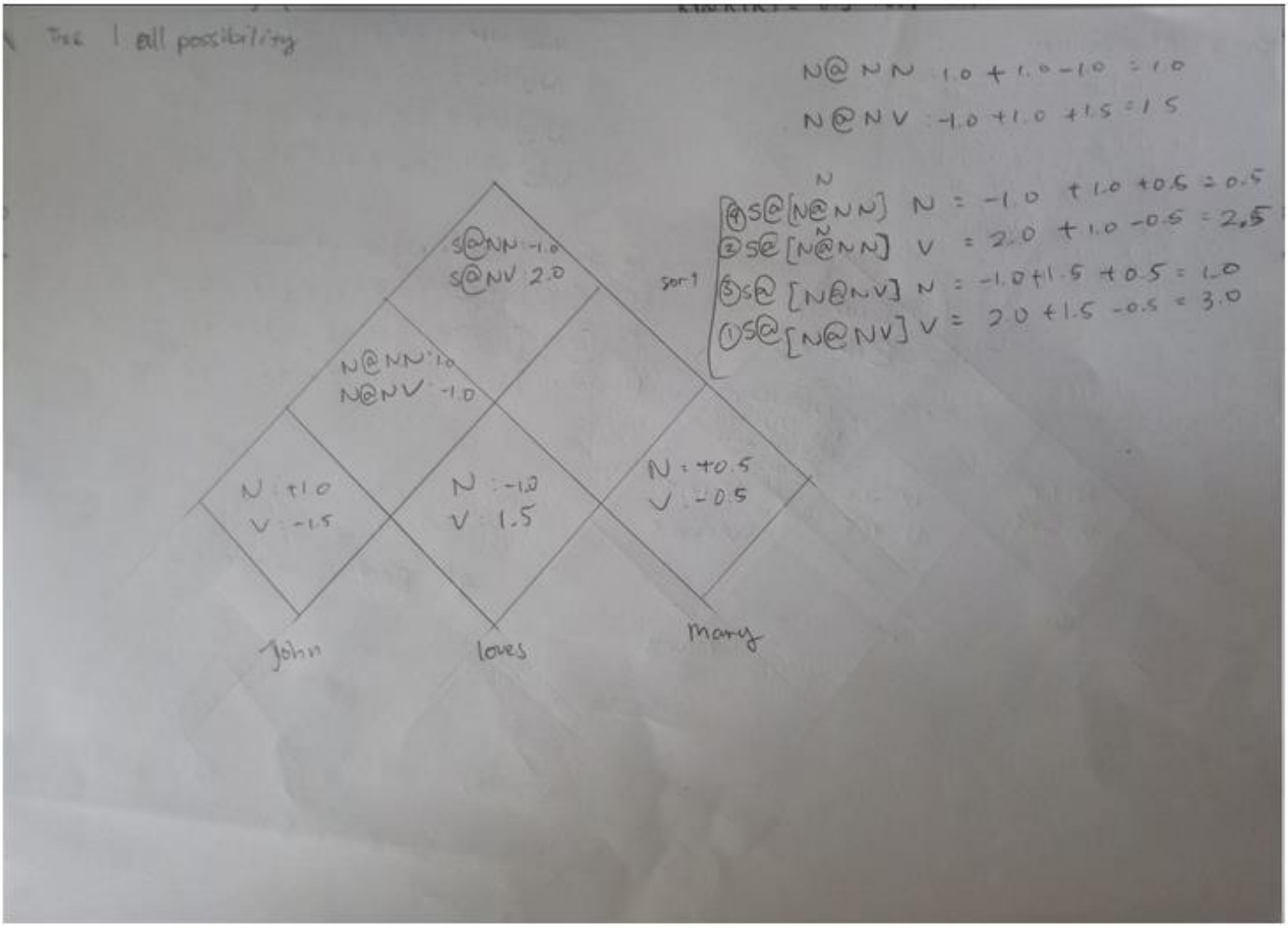
$$S@[N@NV]V$$

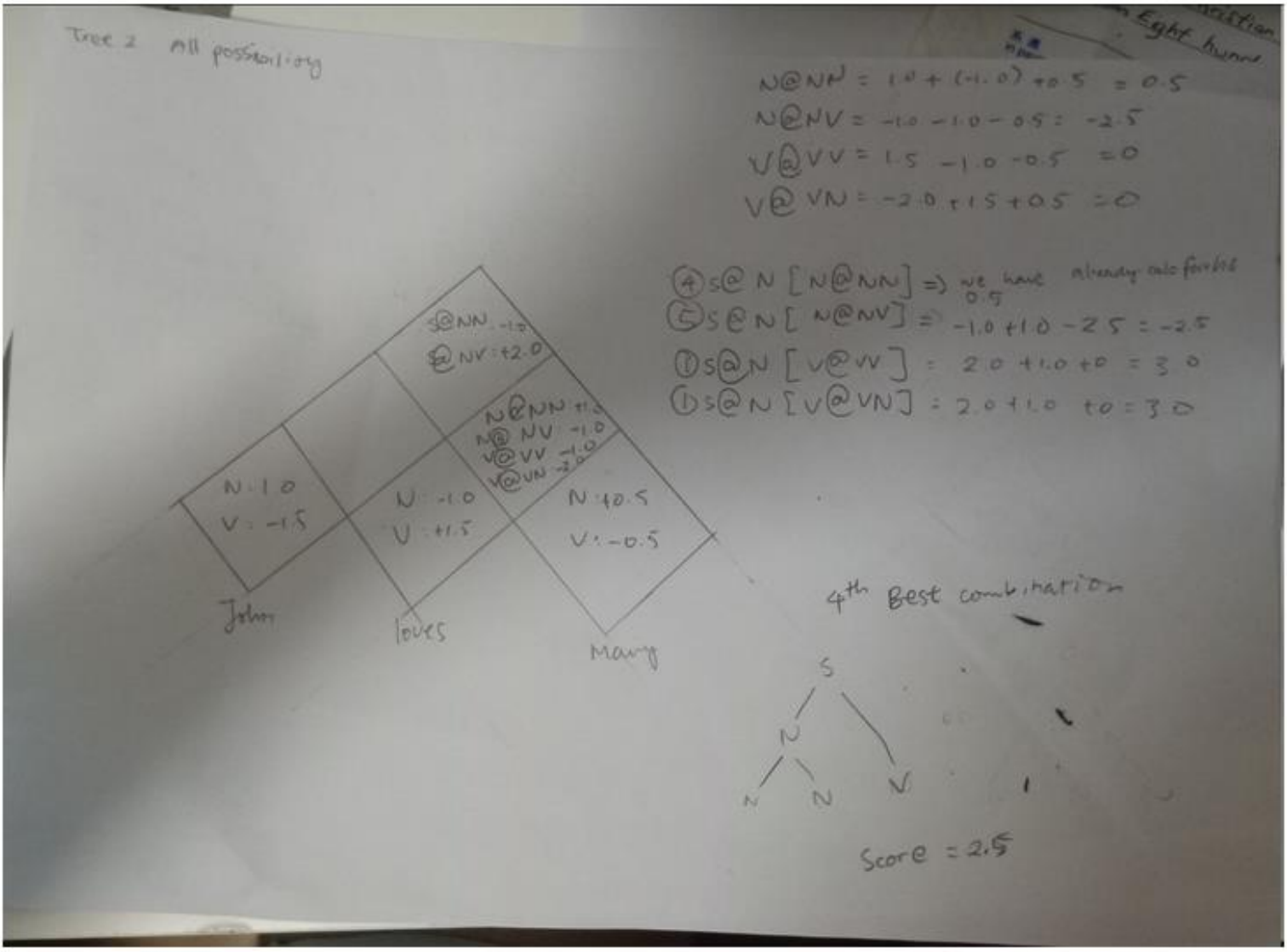
where `John` would have the tag `N`, `loves` would have the tag `V`, and `Mary` would have the tag `V`.

Part 3

To find the 4th best score for `John loves Mary`, we need to modify the CKY algorithm such that it records all the possible values that the tree can make and then save it. filter it out such that there is only n best possible values left to consider and then take the nth best possible combination. This would mean that at each step reacord all the possible combinations of score, sort them out, then take only the n best scores that would be needed for the criteria.

4th Best score for `John loves Mary` (n = 4):



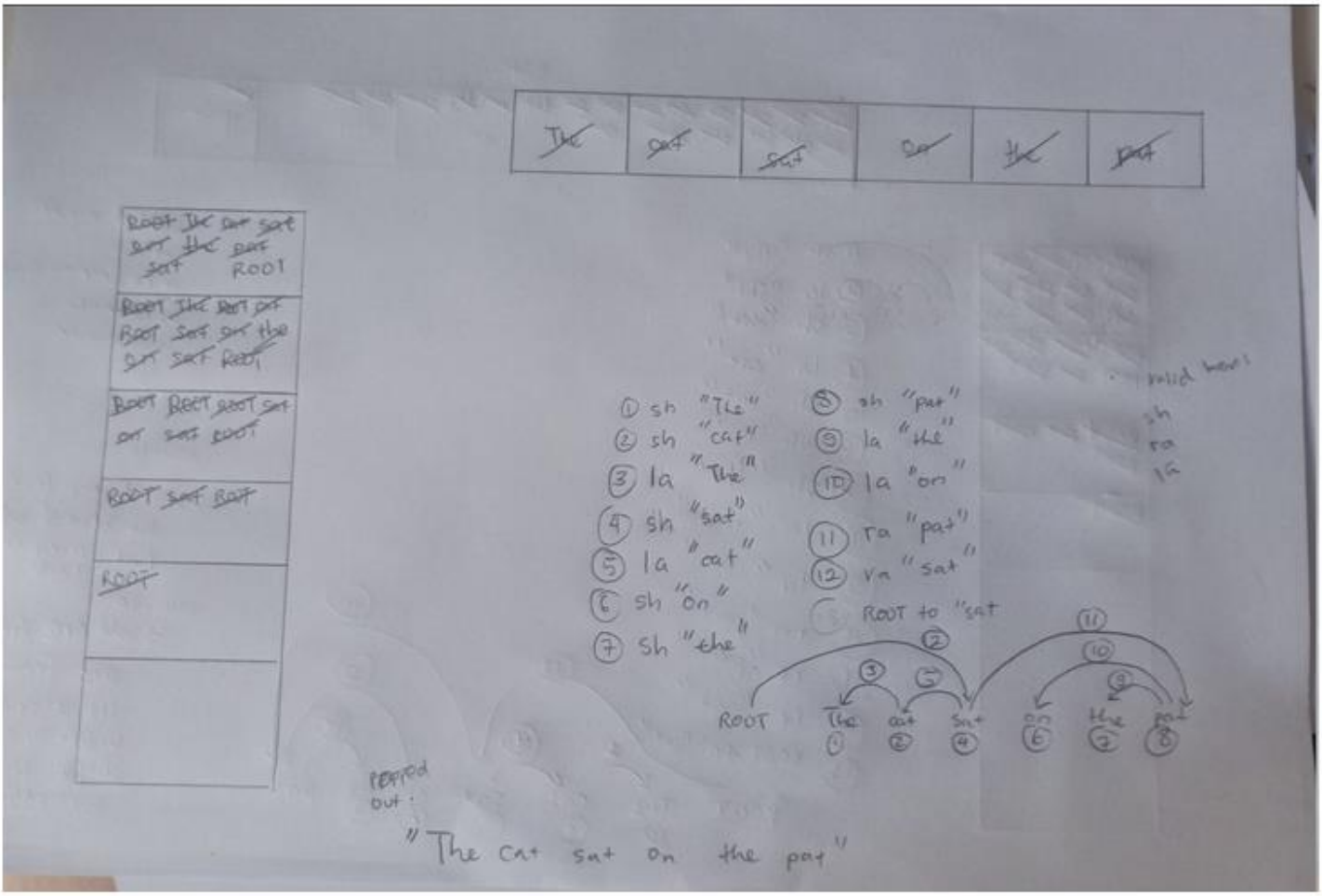


$$S@[N@NN]V$$

4th best score = 2.5

4. Transition-Based Parsing

Part 1



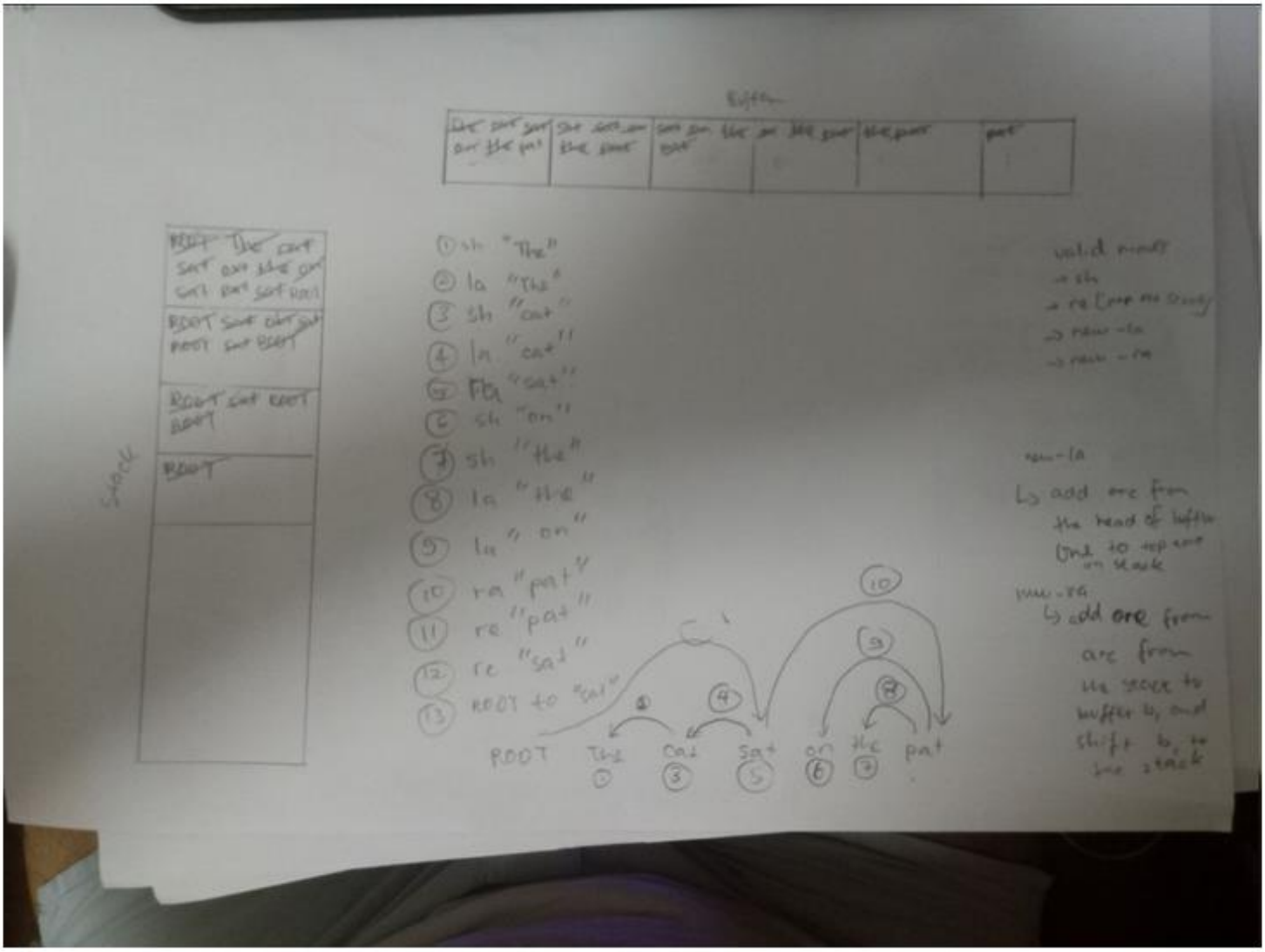
Part 2

The worst time-complexity is

$$O(n)$$

This is because the transition based parsing at worst case only needs to go through the whole sentence once to move it from buffer through stack [O(n)] and removing the whole sentence from the stack, at worst case would take O(n) time complexity.

Part 3



Part 4

The worst time complexity is

$$O(n)$$

Similar to the standard approach it takes in O(n) complexity to move the whole sentence from buffer and popping requires O(n) to remove eveerything from the stack. Nothing seems to indicate that there will be a loop to keep checking for the whole sentence to parse the sentence and hence it is O(n) time complexity at worst.