CKY Parse Time Optimization

The parsing algorithm itself leaves few opportunities for significant (basic) speed improvements, while guaranteeing to return the optimal parse. During the iteration over rows, columns and cells, the lookup in the parse chart is the most expensive operation. Since the chart is often represented as a list of lists, the index based access will be very fast. Also, the cells are implemented here as a dictionary from a symbol to information about its construction (backpointers) and probability. The values of the dictionary are accessed not more than two times per matched rule and the symbol information are represented as an object of the `ChartItem` class to guarantee efficiency while maintaining code readability. If one does not use techniques like pruning or heuristics, the algorithm itself is hard to optimize any further.

The implementation of the PCFG on the other hand can be improved to drastically speed up the parsing process. **In fact, I managed to optimize the time to parse the test set from ~31 hours to only 53 minutes.**

On the given testset, the parser archives an F1 score of **0.703.**

Build the project

This project requires **Python 3.6**. It can be built by running `make` in the folder.

Parser

The parser will be installed in the virtual environment as a script at `env/bin/parse` and can be used as followed:

```
-> env/bin/parse --help
usage: parse [-h] --grammar GRAMMAR [--threads THREADS]

optional arguments:
  -h, --help         show this help message and exit
  --grammar GRAMMAR  Path to grammar file. (default: None)
  --threads THREADS  Number of threads to run on. (default: 4)
```

Trees will be written to stdout, log messages to stderr.

CNF Conversion

To convert a treebank to CNF, run `env/bin/treebank_to_cnf`. It reads data from stdin and outputs to stdout.

Grammar Creation

To create a grammar out of the CNF treebank, run `env/bin/treebank_to_grammar`.

```
-> env/bin/treebank_to_grammar --help
usage: treebank_to_grammar [-h] --treebank TREEBANK [--grammar GRAMMAR]

optional arguments:
  -h, --help            show this help message and exit
  --treebank TREEBANK   Path to read treebank file from. (default: None)
  --grammar GRAMMAR     Path to write grammar file to. (default: gramma
r.txt)
```

Evaluation

To evaluate a parser output, execute `env/bin/evaluate`.

```
-> env/bin/evaluate --help
usage: evaluate [-h] --gold GOLD --test TEST

optional arguments:
  -h, --help   show this help message and exit
  --gold GOLD  The gold file to evaluate against. (default: None)
  --test TEST  The test file to evaluate. (default: None)
```

Models

Baseline

To define a baseline to evaluate all other improvements against, I use the PCFG implementation provided in the starter code. The only change is a reversed data structure for the rule lookup, so that the left hand side (lhs) of a rule can be retrieved by a tuple of the right hand side (rhs) symbols. The parser took 30.15h to parse the testset.

General Improvements

The first improvement is to use represent the strings of the grammar as integers. This not only decreases the used memory, but also improves performance as two integers can be compared in only clock cycle, while string comparison is more computationally intense.

Instead of using a dictionary that has to compute the hash value of a given rhs tuple, I decided to store the rules in an n*n sparse matrix, where n is the number of all terminal and non-terminal symbols. The first rhs symbol corresponds to the rows, while the second symbol stands for the

column. The column with index 0 is reserved to represent unary rules. The cells in the matrix contain the index for a list that holds the actual information about the rules, including the left hand side and the probability. This detour has to be taken because the used sparse matrix implementation only allowed primitive data types to be stored in the cell. This way, to find the lhs for one rule, the PCFG has to perform max. three index lookups which I believe to be faster than the previously needed hashing operation. Additionally the used libraries (NumPy and SciPy) are implemented very efficiently.

The third big improvement for the first model is the introduction of multithreading. Though this does not change the runtime of the parser itself, parallelizing the parsing of multiple sentences speeds up the whole process and was needed for a quicker evaluation.

While testing the improvements with example sentences, I noticed that the new parser performs much faster than the baseline implementation, but not fast enough to parse the testset in a reasonable amount of time.

Model 1: RHS Intersection

A bottleneck of the current parser is the unsuccessful lookup for many rhs symbol combinations. When the first cell holds for example the symbols “N” and “NP” and the second one the symbols “A”, “V” and “VP”, the algorithm tries to find a rule with the right hand side of any combinations of those, even if rules with an rhs like “N A” or “N V” do not exist in the grammar.

To reduce the search space by intersecting the set of symbols in the cells with a set of symbols that actually exist in the grammar at that position. As the intersection algorithm can come with an overhead, I investigated multiple possibilities:

Model 1/Intersection First: The symbols in the first cell are intersected with a set of symbols that occur in the first position of the right hand side of the rule. We iterate over the resulting set and try all combinations with the symbols in the second cell. [Tag]

Model 1/Intersection Second: We iterate over all cells in the first cell and look up a precomputed set of all symbols that occur in the second position of the right hand side of a rule, given the current first symbol. [Tag]

Model 1/Intersection Both: The symbols for both the first and the second symbol in the rules are intersected. All rule lookups will be successful. [Tag]

To make use of the multithreading, the following experiments are performed on a server with 16 CPUs and 60GB RAM (Paperspace C8). This makes the runtime measurements not exactly comparable to the baseline (which is very unfortunate, but I did not want to pay for running the machine for 32 hours), but the CPU used for baseline has a similar performance. If not stated otherwise, the time scores are the sum of the parse time for all sentences, not the actual time the parser takes to finish with 16 CPUs.

Results

	Intersection First	Intersection Second	Intersection Both
Time	Not finished after >900 min	**401 min**	207 min
Logs	-	**Pastebin**	Pastebin

Further investigation with a profiler showed that the parser spends a large amount of time intersecting two sets.

Model 2: Matrix Operations

For the second model, I followed another idea to reduce the search space for finding the matching lhs symbols for the given rhs of a rule. I interpret the symbols in both cells as index vectors and use them to look up the correct rows and columns in the rule matrix. This way, the combination of symbols that lead to existing rules are retrieved by a few matrix operations. My hypothesis is that the internal optimizations in NumPy keep the overhead small and a speed up is archived.

Example
Consider the following rule matrix. The cells hold the index for a lhs lookup array that holds the actual rules. Empty cells are filled with zeros.

rhs[0] \ rhs[1]	**0 (S)**	**1 (NP)**	**2 (VP)**	**3 (Det)**	**4 (N)**	**5 (V)**
0 (S)						
1 (NP)			1			
2 (VP)						
3 (Det)					2	
4 (N)						
5 (V)		3			3	

Index	**Rules**
1	S → NP VP [1.0]
2	NP → Det N [1.0]
3	VP → V NP [0.5], VP → V N [0.5]

Assuming the first cell holds the symbols [1, 3, 4] (NP, Det, N) and the second one the symbols [0, 2, 4] (S, VP, N).

An index array lookup with the values of the first cell along the first axis gives us the following part of the matrix:

rhs[0] \ rhs[1]	0 (S)	1 (NP)	2 (VP)	3 (Det)	4 (N)	5 (V)
1 (NP)			1			
3 (Det)					2	
4 (N)						

After a second lookup using the values of the second cell, we end up with the following matrix:

rhs[0] \ rhs[1]	0 (S)	2 (VP)	4 (N)
1 (NP)		1	
3 (Det)			2
4 (N)			

After flattening it and removing zero values, we have a new index vector to look up the actual rules: [1, 2]. This results in the following set of valid rules for the two given cells: [[S → NP VP [1.0]], [NP → Det N [1.0]]].

Here is an excerpt of the actual code.

```
first_symbols = np.array(list(first_nts.keys()))
second_symbols = np.array(list(second_nts.keys()))

valid_rows = np.take(pcfg.rhs_to_lhs_id, first_symbols, axis=0)
valid_ids = np.take(valid_rows, second_symbols, axis=1).flatten()

non_zero = np.nonzero(valid_ids)
lhs_ids = valid_ids[non_zero].flatten()

lhs_items = np.take(pcfg.id_to_lhs, lhs_ids, axis=0)

for rule in itertools.chain(*lhs_items):
  # ...
```

To make these approach possible, certain changes have to be made to the PCFG data structures:

- The ids of the terminal and non terminal symbols have to be ordered so that the non-terminal symbols start at 0 and are continuous.
- The rule matrix must be a NumPy array and not a SciPy sparse matrix.
- To reduce memory consumption, the rule matrix only holds binary rules. Terminal rules are stored in a separate array.

Results
While the new approach performs good on certain very long sentences, the introduced overhead rather slows the parser down than improving its speed.

The test set was parsed in 802 min. The detailed output can be found here, the code here.

Model 3: Hybrid Model

Assuming that the intersection model performs well for cells with few values and the matrix model to perform well for cells with a huge amount for entries compensating the overhead, I created a third model that combines both approaches.

Here I use a threshold value to decide which approach to choose. When the number of elements in the first and second cell are larger than the threshold, the matrix lookup is performed, otherwise the intersection.

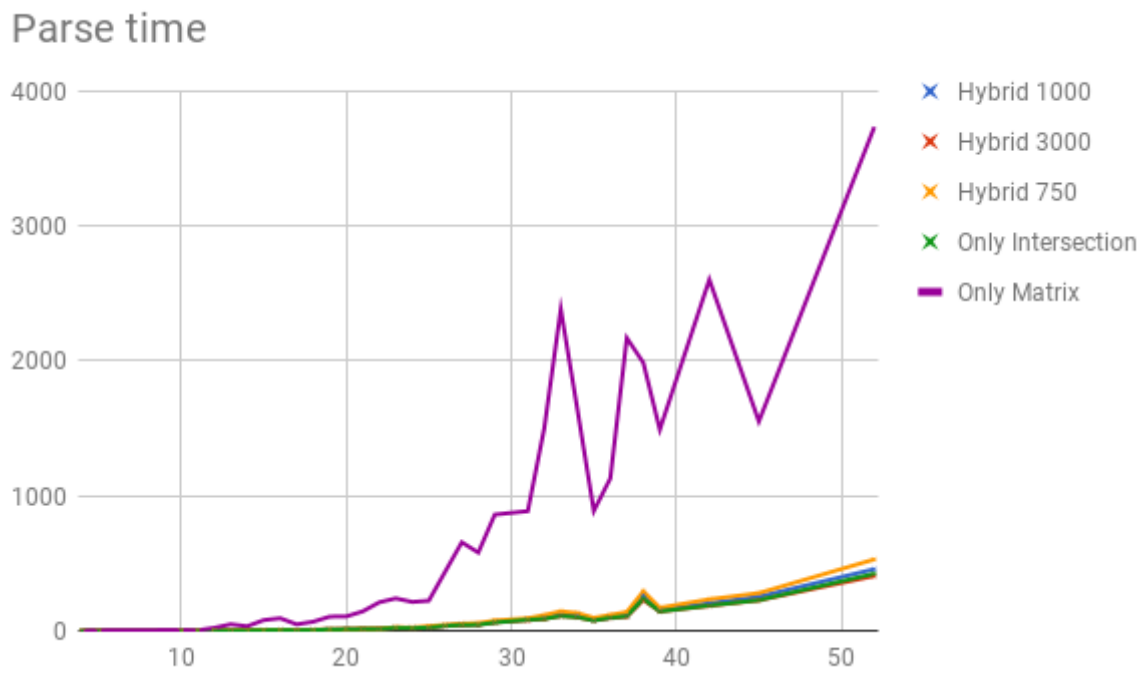
Results
Intersecting both symbols:

Threshold	Time	Results
1000	168 min	Pastebin
3000	**53 min**	**Pastebin**

Intersecting second symbol:

Threshold	Time	Results
Only Matrix	802 min	Pastebin
750	79 min	Pastebin
1000	69 min	Pastebin
1500	65 min	Pastebin
3000	**64 min**	**Pastebin**
Only Intersection	65 min	Pastebin

The following graph shows the parse time per sentence length for the experiment using intersection only for the second symbol. The time for sentences with the same length has been averaged. Link to interactive graph.



The code at the time of the experiments is in the tag model3/hybrid.

Discussion

Overall, the best performing approach is the hybrid model, where the matrix operation based lookup is used for only a few and very large cells. However, the, in the experiment where only the second symbol is intersected, the performance gain compared to the only intersection based run is neglectable.

As there is a dramatic difference between said performance and the intersection experiments (401 min), I argue that the data structures used in model 2 are to blame for the differences. A lookup in a sparse matrix seems to come with a huge overhead, while a smaller NumPy array can be accessed very quickly.

Nevertheless, I find the idea behind the matrix lookup intriguing and wonder if it can be possibly extended in a way to precompute a matrix for multiple cells at a time.

However, in the end of my experiments I started to doubt the times measured. Apparently, the CPU performance of the used server varies depending on the time and date, as it might be shared with other users. Despite of it, a general trend is recognizable, making the hybrid model with a threshold of 3000 the best performing configuration.

The GitHub repository for the project can be found here.