

Earley Parsing with A* search

Yufei Liu

Pratyai Mazumder

Paul He

ETH Zürich

ETH Zürich

{yufliu, pmazumder}@inf.ethz.ch

paulhel@ethz.ch

Abstract

Earley’s algorithm can parse unrestricted context-free grammar in $\mathcal{O}(N^3|\mathcal{G}||\mathcal{P}|)$ runtime. In this paper, we reformulate Earley’s algorithm as a search problem and introduce the application of A* heuristics to enhance the efficiency of the parser. Our approach leverages the strength of A* search to prune the search space effectively, thus accelerating the parsing procedure while still being complete and finite. We provide a detailed description of our formulation, discuss the impact of different heuristics on the algorithm’s performance, and present empirical results to demonstrate the improvements achieved.¹

1 Introduction

Earley Parsing was the first algorithm to directly parse an input with length N under an *unrestricted* context-free grammar in $\mathcal{O}(N^3)$, and unambiguous grammars in $\mathcal{O}(N^2)$. Unlike other chart parsers such as CKY (Younger, 1967; Kasami, 1966), Earley’s algorithm parses the input from left to right, filling in the chart top-down (goal driven). Lets consider the objective of Earley’s algorithm as a search problem, where we want *one possible* parse of the input string, or declare that it is not recognized.

In Earley’s algorithm, an *item* represents a partially completed parse tree. We start the search from a *start item*, and explore new items iteratively until the *goal item* is found. As soon as we have found the goal item, we can immediately produce *one possible* parse tree. Unless we need other parse trees as well, we do not need to explore further. This begs the question whether we can find a path from a start item to a goal item quickly? Or even, can we find an optimal path from a start item to a goal item quickly? For a WCFG, our search

problem still remains the same, An example of optimality is the likeliest parse for a string in an ambiguous PCFG.


An $\mathcal{O}(N^3|\mathcal{G}||\mathcal{P}|)$ algorithm is limited as the size of the grammar increases. This is a downside since natural language are very large, for example, the Penn Treebank (PTB) (Marcus and et al., 1999) containing millions of productions, blowing up the $|\mathcal{P}|$ factor. Improvements towards the original Earley’s algorithm have been proposed already to shave off a factor of $\mathcal{O}(\mathcal{P})$. Opedal et al. (2023) presented a fast version of it as a modified deduction system, reducing the runtime to $\mathcal{O}(N^3|\mathcal{G}|)$ by iteratively applying a weighted fold transformation on the prediction and completion rules.

We aim to convert Earley parsing into a search problem, which allows us to speed up the process of recognition. This has already been performed on other chart parsing algorithms, notably Klein and Manning (2003) presented an extension of the classical A* search procedure to tabular PCFG parsing by precomputing grammar statistics as A* estimates. The main benefits of A* can substantially reduce the work required to parse a sentence, a structurally simpler parser, can be easily proven correct.

2 Background

2.1 Context-Free Grammars

A **context-free grammar (CFG)** \mathcal{G} is a tuple $\langle \mathcal{N}, \Sigma, \mathcal{P}, S \rangle$ where \mathcal{N} is a non-empty set of non-terminal symbols, Σ is the set of terminal symbols (alphabet) with $\mathcal{N} \cap \Sigma = \emptyset$, $S \in \mathcal{N}$ is a designated start non-terminal symbol, and \mathcal{P} is the set of production rules where each rule $p \in \mathcal{P}$ is a 2-tuple $\langle A, \alpha \rangle$, with $A \in \mathcal{N}$ and $\alpha \in (\mathcal{N} \cup \Sigma)^*$, we denote a production rule as $A \rightarrow \alpha$. We will introduce a few more notations we use in Appendix A.1.

¹ https://github.com/Hepaul7/Astar_earley

2.2 Earley's Algorithm

Earley's algorithm is a top-down dynamic program, unlike other chart parsing algorithms like CKY, it does not require the grammar to be in a normal form. There are three main operations of the Earley algorithm are **predict**, **scan**, and **complete**, which we will describe as a deduction system. The notation for the deduction system can be found in Appendix A.2.

An item represents a partially completed parse tree and is denoted as dotted rules. More specifically, given a production rule $X \rightarrow \alpha\beta$, the notation $X \rightarrow \alpha \bullet \beta$ represents a condition in which α has already been parsed and β is expected. A finished item means there are no more symbols after the dot. An item can be processed and eventually finished by three actions: prediction, scan, and completion. Prediction acts on a non-terminal and generates new items by applying production rules. Scan matches the next symbol with a terminal, thus moving the dot forward by one position. Completion applies a finished item to the next symbol and moves the dot forward.

$$\begin{aligned} \text{PREDICTION: } & \frac{B \rightarrow \rho}{[j, j, B \rightarrow \bullet \rho]} [i, j, A \rightarrow \mu \bullet B \nu] \\ \text{SCAN: } & \frac{[i, j, A \rightarrow \mu \bullet a \nu] \quad [j, k, a]}{[i, k, A \rightarrow \mu a \bullet \nu]} \\ \text{COMP: } & \frac{[i, j, A \rightarrow \mu \bullet B \nu] \quad [j, k, a]}{[i, k, A \rightarrow \mu B \bullet \nu]} \end{aligned}$$

All items with the same dot position are arranged under a state. A chart is the list of all the states for the input. We start the parsing process by the *start item* $[0, 0, S' \rightarrow \bullet S]$, which is located at the first state in the chart. The parse is completed when reaching the *goal item* $[0, N, S' \rightarrow S \bullet]$ at the last state in the chart, which is equivalent to having the dot position at the end of the input string. Otherwise, the string is rejected. The runtime for Earley's Algorithm $\mathcal{O}(N^3|\mathcal{G}||\mathcal{P}|)$.

Opedal et al. (2023) proposed a sped up version of Earley's algorithm by introducing a new set of deduction rules to reduce a factor of $\mathcal{O}(\mathcal{P})$, which we will include in Appendix C. It iteratively applies a weighted fold transform (Burstall and Darlington, 1977), on the prediction and completion rule reducing their total runtime to $\mathcal{O}(N^3|\mathcal{G}|)$. A new constant symbol \star was introduced which is a wild card indicating any sequence ρ . Furthermore, they introduced a new set of items $[i, j, A \rightarrow \mu \bullet \nu]$,

$[j, k, a], A \rightarrow \rho, [i, j, A \rightarrow \star \bullet]$, with the axioms $A \rightarrow \rho \forall (A \rightarrow \rho) \in \mathcal{P}, [k-1, k, x_k] \forall k \in \{1, \dots, N\}, [0, 0, S \rightarrow \bullet \star]$, with the goal as $[0, N, S \rightarrow \star \bullet]$.

2.3 A* search

A* search is a widely used shortest path-finding (SPF) algorithm (Hart et al., 1968). Given a weighted graph, a start node, and a goal node, an SPF algorithm finds the shortest path from start to goal with respect to the weight. A typical SPF, like Dijkstra's, traverses the graph node-by-node, prioritising unvisited nodes known to be close to the start node, but uses no information about the distance from the goal node. Typical A* search implementation use a priority queue, which is often called as the *frontier*.

The key characteristic of A* search is the use of a heuristic function $h(n)$ that *estimates* the distance from a node n to the goal node, along with the function $g(n)$ that gives the known distance from n to the start node. Then, the function $f(n) := g(n) + h(n)$ estimates the shortest length of a path through the node n , which can be used to determine the traversal order of the nodes. A* search can be viewed as a generalisation of Dijkstra's, since setting $h(n) := 0$ gives us back the Dijkstra's algorithm.

The correctness guarantee and performance of an A* search depends on this heuristic function $h(n)$. If $h(n)$ *never overestimates* the true distance from n to the goal node, then A* search is guaranteed to return a shortest path after visiting each node at most once. Such an $h(n)$ is called an *admissible* heuristic.

We now state the A* search completeness theorem.

Theorem 2.1. *A* search will always find a solution if one exists as long as the branching factor is finite, every action has a finite cost $\varepsilon > 0$, and $h(n)$ is finite for every item n that can be extended to reach a goal item.*

Proof. Proof in Appendix D.1.1. \square

We now state the A* search optimality theorem.

Theorem 2.2. *A* search with an admissible heuristic always finds an optimal cost solution if it exists, as long as the branching factor is finite and every action has a finite cost $\varepsilon > 0$.*

Proof. Proof in Appendix D.2.1 \square

3 Algorithm

3.1 Fast-Track Earley Algorithm

We describe Earley’s algorithm as a search problem. The goal is to find one short path from the start item to the goal item by iteratively generating and processing items. The pseudocode for the algorithm can be found in Appendix E.

The original (or ‘naive’) Earley’s algorithm explores the search space naively by selecting an item from the set of unprocessed items, and processing it. We will call an item that requires such processing a ‘dirty’ item. Items corresponding to an earlier input position are processed earlier, and among the items that correspond to the same input position can be ordered arbitrarily. Importantly, this order ensures that the ‘state set’² for an input position is exhausted before the algorithm moves to a new input position. The search is finished when all reachable items are processed.

However, processing all items can be inefficient in various ways:

- If many items that are not on a path to a goal item are produced.
- If the input string can be recognized through many different paths.

In practice, a single ‘recognition’ path often occupies only a small portion of the search space (which consists of the produced items). To mitigate this inefficiency, we present our optimized version of Earley’s algorithm called *fast-track Earley’s algorithm*. This algorithm employs an A* heuristic function to optimize the order of item processing, and prioritizes certain parts of the search space based on the given information, thereby efficiently identifying a shorter or optimal path.

The fast-track Earley’s algorithm has two key ideas:

- It can process items in completely arbitrary order, i.e., any dirty items in any input position. This is unlike naive Early parser, which must exhaust all items in all the earlier input positions before moving on to a later position.
- It can use an arbitrary heuristic to determine which dirty item to process next.

To ensure the correctness and performance, we introduce a few other secondary, but important ideas:

- Each *freshly predicted* (i.e. the dot at beginning) tracks all the input positions where it

²The set of all the produced items for a given input position.

has been completed so far (i.e. the dot at the end). This way, if this freshly predicted item is predicted again (which can happen as we are processing items in out-of-order, unlike naive Earley), we can avoid recomputing its completion paths.

- The algorithm terminates immediately when any goal item is found, since we need just one recognition path. This early termination is not particularly useful for naive Earley, since the goal items are always at the last input position. However, with fast-track Earley’s we expect to skip many items at earlier input positions.

A pseudocode of this fast-track Earley’s algorithm is presented in algorithm 2. The pseudocode does not show the implementation details such as constructing the parse tree for brevity. The choice of the heuristic $h()$ is also unspecified, whereas in the implementation we have experimented with several heuristics as discussed in section 3.2.

3.2 A* Heuristics

So far in appendix E, we have left the details of the heuristic function h that maps an Earley item to a scalar representing the priority the item in the queue of ‘dirty’ items. In this section, we discuss a few implemented heuristics. The results from the experiments with these heuristics is presented in 4.2.

We note that, due to how the implementation was done, we have explicitly implemented only $f := g + h$, instead of just h , since f is the function is used for the priority queue in A* search. Except for ‘naive’, the heuristics are not necessarily admissible and therefore requires recomputations when g changes for the items — however, we skip these recomputation with the ‘fast-tracking’ mechanism.

We recall that an Earley item encodes the following information:

- The production rule in the grammar.
- The dot’s position in that production rule.
- The input position where that production rule started.

Additionally, as we can derive from the item’s position on the Earley chart, the current input position (i.e., the part of the rule before the dot spans the input subsequence between starting and current position). The heuristics use these information along with any additional prior knowledge to determine the priority of an item in the queue.

3.2.1 ‘Naive’ Prioritizer

This assigns the priority based on solely the current position — the earlier the current position is, the more prioritized the item is. This is essentially the same processing order as the ‘naive’ Earley’s algorithm, with the overhead of the priority queue maintenance.

3.2.2 ‘Eager’ Prioritizer

This is the reverse of ‘naive’ prioritizer — the later the current position is, the more prioritized the item is. As a result, the algorithm can find a quicker path to the goal item, without having to evaluate virtually all possible items like ‘naive’ Earley’s algorithm.

3.2.3 ‘Terminal’ Prioritizer

This is similar to the ‘eager’ prioritizer, except that any item where the dot is right on a *terminal* symbol (i.e. the item will trigger a ‘scan’ operation) is assigned the highest priority (which is a fixed constant). This encourages the algorithm to progress on the input position even faster.

3.2.4 ‘Nearest Terminal’ Prioritizer

For this heuristic, we run a small analysis of the grammar first (using one depth-first traversal) and determine, for each item, the minimum number of predict/complete operations are needed to complete it or trigger a scan (i.e. the dot is on a terminal symbol).

4 Experiments

All our experiments follow the framework:

- Select an instance of a CFG.
- Select an input string that are recognized by that CFG.
- Select a variant of Earley’s algorithm.
- For fast-track Earley’s, select also a heuristic.
- Then run the algorithm and report the measurements along with additional statistics about the grammar and the input.

We will describe the various choices in this framework in the following sections.

4.1 PennTreeBank Experiment

To test our algorithms, we utilize the Wall Street Journal (WSJ) section of the Penn Treebank (PTB) (Marcus and et al., 1999) as the benchmark dataset for our experiments. The PTB contains a diverse set of annotated text from the WSJ. There are 95891

unique production rules. Since we tested correctness of our algorithms with a separate methodology, we only test the parsing speed of the algorithms. To do that, we reduced the grammar size to around 100 and selected inputs of variable length to test.

To ensure the quality of our sentences, we first performed preprocessing steps, namely, the sentences are tokenized into words, and filtered such that they only contain normal English words. Then, we extracted the grammar rules from the PTB. We tested all three algorithms `naiveEarley`, `FastEarley`, and `FastTrackEarley`. Each algorithm was selected to test on a subset of the sentences extracted from the parse trees of our dataset.

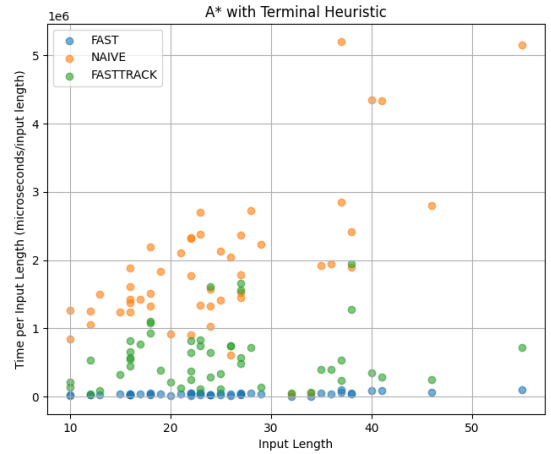


Figure 1: Time per Input Length for all 3 algorithms with terminal prioritizer

Figure 1 shows the results of our experiment with the ‘Terminal’ prioritizer. We notice that the `FastEarley` has a most consistent and best performance. We also see that our `FastTrackEarley` algorithm in general outperforms the `NaiveEarley` algorithm. The `NaiveEarley` has a clear cubic relationship in terms of the input length, while `FastEarley` remains low.

Interestingly, the performance of `FastTrackEarley` fluctuates. This fluctuation is likely due to the overhead of maintaining additional data structures for each item. The shorter runtimes observed for `FastTrackEarley`, which can sometimes be even faster than `FastEarley`, could be attributed to the savings in time from not predicting items that have already been completed. However, it is also noted that for shorter inputs, `FastTrackEarley`

sometimes exhibits performance similar to the `NaiveEarley` algorithm, most likely due to less common predictions.

4.2 Customized grammars and strings using Hypothesis

We design a pipeline to evaluate our algorithms against customized grammars using `Lark` ([Lark Team, 2018](#)) and `Hypothesis` ([Hypothesis Team, 2018](#)). This ensures that the algorithm can handle specific edge cases without requiring a large amount of data.

`Lark` is a parsing toolkit for Python capable of parsing all context-free languages. It is integrated with `Hypothesis` a Python library for creating unit tests.

We use the test files in the course assignments as a starting point. These test files contain 100 CFG grammars with the correct intermediate charts and output scores. We convert the CFG grammars into `Lark` syntax (EBNF syntax), and then generate strings for each grammar using `Hypothesis`. There are in total 6546 strings with an average string length 10.

Using the grammar rules and the strings, we have tested all three variants of Earley’s algorithm (‘naive’, ‘fast’ and ‘fast-track’). For the ‘fast-track’ variant, we have tested with four A* heuristics ‘naive’, ‘eager’, ‘terminal’ and ‘nearest terminal’. The results are presented in [fig. 2](#) and [fig. 3](#).

4.2.1 Observations

[fig. 2](#) shows that the runtime of fast-track Earley’s algorithm tends to be lower significantly lower than the other variants for almost all the test grammars and inputs. It also shows that, within the ‘fast-track’ variant, the choice of heuristic also has a significant impact on the runtime, as the ‘terminal’ heuristic were faster than its competitors for nearly all the test grammars and inputs.

Looking at [fig. 3](#) gives an explanation for this. For nearly all test cases, the ‘fast-track’ variant (with ‘terminal’ heuristic) has produced only a fraction of the items produced by the ‘naive’ variant, whereas the ‘fast’ variant in fact produces even more items. Similarly, within the ‘fast-track’ variant, the ‘terminal’ heuristic has produced the fewest items than the other heuristics. Since we expect the runtime of Earley’s algorithm to super-linearly scale with the number of items it produces, simply being able to skip unnecessary items greatly accelerates the parser *in practice* — even if the the-

oretical worst-case runtime complexity is slightly *higher* when the cost of priority queue operations are considered.

5 Conclusion

In this paper, we converted Earley Parsing into a search algorithm, enabling us to incorporate various A* heuristics to achieve a noticeable improvement to `NaiveEarley` and `FastEarley`. Additionally, our implementation is robust to cycles in the grammar, ensuring its versatility in more complex parsing scenarios.

However, we would like to further explore our algorithms with more types of context-free grammars, and different kinds of inputs. We also noticed that `FastEarley` excels when the branching factor is large in a grammar, while it may be slower than `FastTrackEarley` when there are few branching factors, and would like to show it through experiments in the future. We also believe it would be powerful to use treebanks and other real-world datasets to explore statistical heuristics, such that the heuristic can be guaranteed to be admissible and return some optimal solution. Lastly, as we mainly tried to improve `NaiveEarley`, we would like to extend `FastEarley` to use A* heuristics.

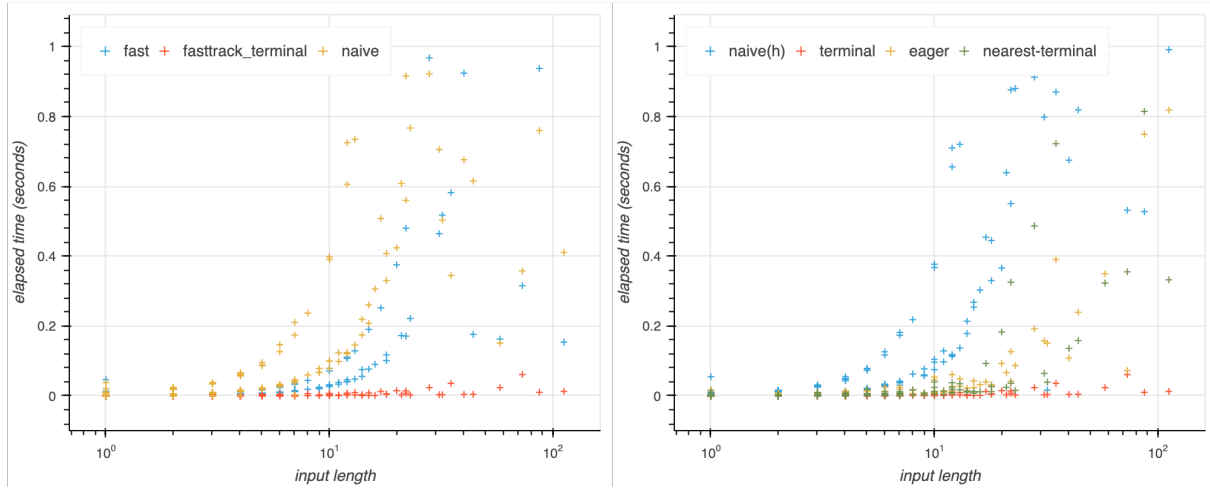


Figure 2: Left: The parsing time vs. the input length for the 3 variants of Earley's algorithm. Right: The parsing time vs. the input length for the 4 heuristics used in the fast-track Earley's algorithm. The data is aggregated from 10 test CFGs with varying sizes and many sample input strings for each of them.

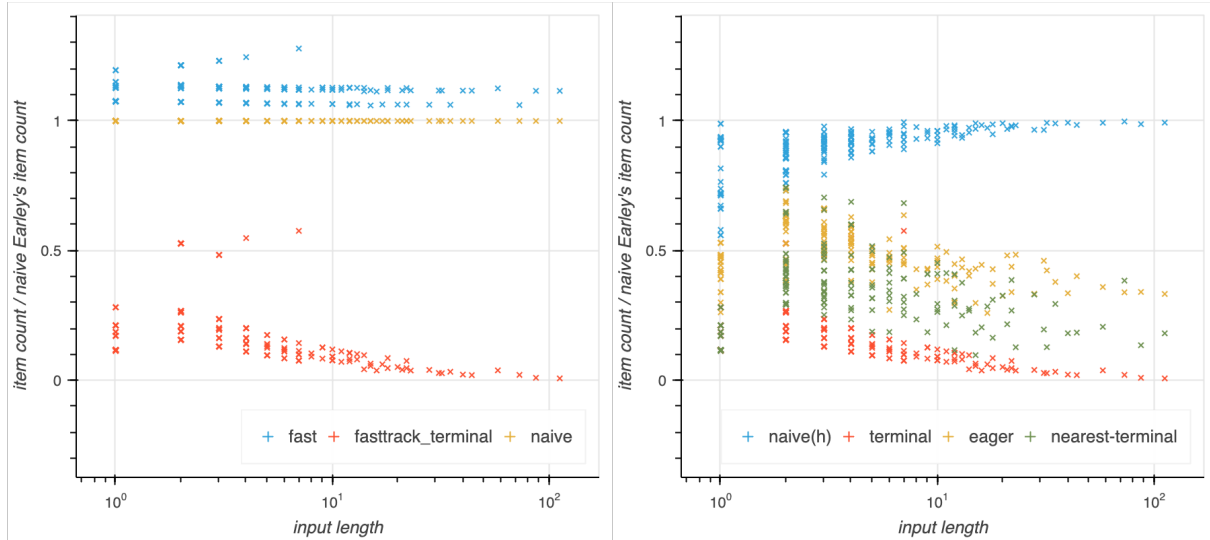


Figure 3: Left: The number of produced items for the 3 variants of Earley's algorithm, normalized by the same number for the naive variant. Right: The same metric for the 4 heuristics used in the fast-track Earley's algorithm. The data is aggregated from 10 test CFGs with varying sizes and many sample input strings for each of them.

References

- R. M. Burstall and John Darlington. 1977. [A transformation system for developing recursive programs](#). *J. ACM*, 24(1):44–67.
- Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. [A formal basis for the heuristic determination of minimum cost paths](#). *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107.
- Hypothesis Team. 2018. [Hypothesis package](#). Github.
- Tadao Kasami. 1966. An efficient recognition and syntax-analysis algorithm for context-free languages. *Coordinated Science Laboratory Report no. R-257*.
- Dan Klein and Christopher D. Manning. 2003. [A* parsing: Fast exact Viterbi parse selection](#). In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, pages 119–126.
- Lark Team. 2018. [Lark package](#). Github.
- Mitchell P. Marcus and et al. 1999. Treebank-3 ldc99t42. Web Download.
- Andreas Opedal, Ran Zmigrod, Tim Vieira, Ryan Cotterell, and Jason Eisner. 2023. [Efficient semiring-weighted Earley parsing](#). In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 3687–3713, Toronto, Canada. Association for Computational Linguistics.
- Daniel H. Younger. 1967. [Recognition and parsing of context-free languages in time \$n^3\$](#) . *Information and Control*, 10(2):189–208.

A Background and Notations

A.1 Context-Free Grammars

A **context-free grammar (CFG)** \mathcal{G} is a tuple $\langle \mathcal{N}, \Sigma, \mathcal{P}, S \rangle$ where \mathcal{N} is a non-empty set of non-terminal symbols, Σ is the set of terminal symbols (alphabet) with $\mathcal{N} \cap \Sigma = \emptyset$, $S \in \mathcal{N}$ is a designated start non-terminal symbol, and \mathcal{P} is the set of production rules where each rule $p \in \mathcal{P}$ is a 2-tuple $\langle A, \alpha \rangle$, with $A \in \mathcal{N}$ and $\alpha \in (\mathcal{N} \cup \Sigma)^*$, we denote a production rule as $A \rightarrow \alpha$. We will refer to $|\alpha| > 0$ as the **arity** of the production rule, $|A \rightarrow \alpha| = 1 + |\alpha|$ as the **size** of the production. We define **grammar size** as $|\mathcal{G}| = \sum_{A \rightarrow \alpha \in \mathcal{P}} |A \rightarrow \alpha|$. We define productions of size 0 as **nullary** productions and size 1 as **unary** productions. CFGs can also be weighted, however, throughout this paper, we will consider the unweighted case, this is akin to a Weighted-CFG under the boolean semiring.

Furthermore, we say that a rule $B \rightarrow \beta$, $\beta \in (\mathcal{N} \cup \Sigma)^*$ is **applicable** to B in a rule p if the rule takes the form $A \rightarrow \alpha B \gamma$.

A.2 Deduction System

$$\text{EXAMPLE: } \frac{U_1 \quad U_2 \quad \dots}{V}$$

We will use the following example to illustrate a deduction rule. The left label indicates the **name** of the rule. A deduction rule may have side-conditions that must hold true when applying the rule. U_1, U_2, \dots are called the premises, which are the initial statements or hypotheses from which the conclusion is derived. We call V the conclusion of the deduction. **Axioms** are rules without a premise. In the unweighted case, we want to show whether certain **goal item** can be proved using the set of deduction rules from axioms that encode \mathcal{G} and \mathbf{y} . There are axioms for each rule $A \rightarrow \alpha$ in \mathcal{G} . For terminals, the axioms are $[j, j + 1, a]$ which is true if the position j in the input is equal to the non-terminal symbol a . The other items are in the form of $[i, j, A \rightarrow \alpha \bullet \beta]$, where $\alpha, \beta \in (\mathcal{N} \cup \Sigma)^*$.

B Naive Earley Pseudocode

Algorithm 1 Naive Earley

```

1: class NAIVE EARLEY algorithm 1 is our im-
   implementation for the Naive Earley Algorithm.
2:   Attribute: chart
3:   Constructor(input_str, g)
4:     Initialize empty chart of length
       len(input_str)+ 1
5:     Add items to the initial state of the
       chart
6:   EndConstructor

7:   Method process_one_dirty_item(pos,
       dirty_items)
8:     Pop one item from dirty_items
9:     if no symbol after dot_pos :
10:      Complete
11:    else if the symbol after dot is in-
       put_str[pos] :
12:      Scan
13:    else if the symbol after dot is non-
       terminal :
14:      Predict
15:    EndMethod

16:   Method run
17:     for pos = 0 to len(chart) :
18:       Initialize dirty_items as an empty
       set
19:       for it in chart[pos] :
20:         Add it to dirty_items
21:       while dirty_items is not empty :
22:         process_one_dirty_item(pos,
           dirty_items)
23:     EndMethod

```

C FastEarley Deduction Rules

We present the FastEarley Deduction Rules for reference. The items are in the form of $[i, j, A \rightarrow \mu \bullet \nu]$, $[j, k, a]$, $A \rightarrow \rho$, $[i, j, A \rightarrow \star \bullet]$, with the axioms $A \rightarrow \rho \forall (A \rightarrow \rho) \in \mathcal{P}$, $[k-1, k, x_k] \forall k \in \{1, \dots, N\}$, $[0, 0, S \rightarrow \bullet \star]$, with the goal as $[0, N, S \rightarrow \star \bullet]$.

PREDICTION1: $\frac{}{[j, j, B \rightarrow \bullet \star]} [i, j, A \rightarrow \mu \bullet B \nu]$

PREDICTION2: $\frac{B \rightarrow \rho}{[j, j, B \rightarrow \bullet \rho]} [j, j, B \rightarrow \bullet \star]$

SCAN: $\frac{[i, j, A \rightarrow \mu \bullet a \nu] \quad [j, k, a]}{[i, k, A \rightarrow \mu a \bullet \nu]}$

COMP1: $\frac{[j, k, B \rightarrow \rho \bullet] \quad [j, k, B \rightarrow \star \bullet]}{[i, k, A \rightarrow \mu B \bullet \nu]}$

COMP2: $\frac{[i, j, A \rightarrow \mu \bullet B \nu] \quad [j, k, B \rightarrow \star \bullet]}{[i, k, A \rightarrow \mu B \bullet \nu]}$

D A* Proofs

D.1 Proof A* completeness

D.1.1 Proof for Theorem 2.1

Proof. Assume that a solution node n exists, then we will have two possible cases:

1. n has been expanded by A*
2. An ancestor of n is on the Frontier

Assume that the second case holds, denote the ancestor of the Frontier be n_i , then n_i must have a finite f -value. As A* runs, the f -value of the nodes on the Frontier eventually increases, hence either A* terminates due to the existence of a solution, or, n_i becomes the node on the Frontier with the lowest f -value.

If n_i is expanded, then either $n_i = n$ and A* returns n as the solution, or, n_i is replaced by its successors, one of it is n_{i+1} which is a closer ancestor of n .

Applying the same argument to n_{i+1} , we see that if A* continues to run without finding a solution, it will eventually expand every ancestor of n , including n itself, therefore finds and returns a solution. \square

D.2 Proof for A* optimality

To prove the optimality of A* search, we first define a proposition

Proposition D.1. *A* with an admissible heuristic never expands a node with f -value greater than the cost of an optimal solution.*

Proof. Let C^* be the cost of an optimal solution $p : (s_0, s_1, \dots, s_k)$, $\text{cost}(p) = \text{cost}(s_0, s_1, \dots, s_k) = C^*$. We show that for each node in the search space that is reachable from the initial node, at every iteration we have an ancestor of the path is on the Frontier.

Let n be a node reachable from the initial state and n_0, n_1, \dots, n_i be ancestors of n , so at least one of them is always on the Frontier. We can show

that with an admissible heuristic, for every prefix n_i of n , we have $f(n_i) \leq C^*$:

$$C^* = \text{cost}(s_0, s_1, \dots, s_k) \quad (1)$$

$$= \text{cost}(s_0, s_1, \dots, s_i) = \text{cost}(s_i \dots, s_k) \quad (2)$$

$$= g(n_i) + h^*(n_i) \quad (3)$$

$$\geq g(n_i) + h(n_i) = f(n_i) \quad (4)$$

where we used the fact that $g(n_i) = \text{cost}(n_i) = \text{cost}(s_0, s_1, \dots, s_k)$, where $h^*(n_i)$ is the cost of an optimal path from s_i to any goal state, which must be equal to $\text{cost}(s_i \dots, s_k)$, since (s_0, s_1, \dots, s_k) is optimal. \square

D.2.1 Proof for Theorem 2.2

Proof. Let C^* be the cost of an optimal solution, assume that a solution exists by Theorem 2.1, then we know that A* will terminate by expanding some solution node n . Then, can use the previous proposition to know $f(n) \leq C^*$, since n is a goal node, $h(n) = 0$ by definition and therefore $f(n) = g(n) = \text{cost}(n)$ and therefore $\text{cost}(n) \leq C^*$. Furthermore, we also have that $C^* \leq \text{cost}(n) = f(n)$ due to the fact we cannot find any solution that has a lower cost than the optimal. Hence, $\text{cost}(n) = C^*$, therefore A* returns a cost-optimal solution. \square

E The FastTrack Earley Algorithm

Please refer to Algorithm 2 for the pseudocode.

F The Grid Parsing Algorithm

We also explored Earley's algorithm as a grid search approach. The algorithm works as follows. Namely, we keep track of two axis, the horizontal axis with length of the input length + 1, and the vertical is the level of prediction. However, this algorithm does not work where grammar contain cycles, as it could potentially have an infinite level of prediction.

Algorithm 3 GrammarPoint Class

```

1: Class GrammarPoint:
2:   Attributes: sym, rule, dot
3:   Method proceed():
4:     Return GrammarPoint(sym, rule, dot +
      1)
5:   Method reverse():
6:     Return GrammarPoint(sym, rule, dot - 1)
```

Algorithm 4 Item Class

```

1: Class Item:
2:   Attributes: point, beg
3:   Method proceed():
4:     Return Item(point.proceed(), beg)
```

Algorithm 5 GridNode Class

```

1: Class GridNode:
2:   Attributes: position, state, point_pos,
      scanned_symbol, leftover, to_be_completed
```

Algorithm 6 GridParser Class

```

1: Class GridParser:
2:   Constructor(input_str, g):
3:     Initialize grid with GridNode instances
4:     Set initial state in grid[0][0]
5:   Method collect_productions(state, node):
6:     Return Set of productions from grammar
      rules based on state and node
7:   Method process(next_sym):
8:     For each item in state:
9:       Complete:
10:        Update state based on rules and po-
          sitions
11:      Predict:
12:        Expand productions in current node
13:      Scan:
14:        Move to next state if terminals match
15:      Check leftovers:
16:        Update grid if leftovers exist
17:   Method run_parse():
18:     While parsing:
19:       Process each input symbol
20:       Update grid state and position
```

Algorithm 2 Fast-Track Earley

```
1:  $S[1 \dots n]$  := the input sequence of length  $n$ 
2:  $C[0 \dots n]$  := Earley chart to be constructed
3:  $G$  := Grammar
4:  $h$  := heuristic mapping items to priorities
5:  $T$  := a table mapping a freshly predicted item to
   a list of completion positions
6: method RUN( $G, S$ )
7:   Initialize  $C$  with all empty sets
8:   Initialize  $C[0]$  with starting items from  $G$ 
9:    $Q \leftarrow$  priority queue with  $C[0]$ 
10:   $done \leftarrow False$ 
11:  while  $Q$  is not empty and  $done! = True$  :
12:     $t \leftarrow$  Pop item from  $Q$ 
13:    if  $t$  is a goal item :
14:       $done \leftarrow True$ 
15:    else if  $t$ 's dot is at end :
16:      COMPLETE( $t, Q$ )
17:    else if  $t$ 's dot is on a terminal :
18:      SCAN( $t, Q$ )
19:    else  $\triangleright t$ 's dot is on a non-terminal
20:      PREDICT( $t, Q$ )
21:  end
22: method COMPLETE( $t, Q$ )
23:   $b, p \leftarrow t$ 's start and current position
24:   $s \leftarrow$  the head symbol for  $t$ 
25:   $Z \leftarrow$  {items in  $C[b]$  whose dot is on  $s$ }
26:  for  $z \in Z$  :
27:     $z' \leftarrow z$  with dot one step forward
28:    if  $z' \notin C[p]$  :
29:      Add  $z'$  to  $Q$  with weight  $h(z')$ 
30:      Add  $z'$  to  $C[p]$ 
31:   $t_0 \leftarrow t$  with dot at start
32:  Add  $p$  to  $T[t_0]$ 
33: end
34: method SCAN( $t, Q$ )
35:   $b, p \leftarrow t$ 's start and current position
36:   $t' \leftarrow t$  with dot one step forward
37:  if  $t' \notin C[p + 1]$  :
38:    Add  $t'$  to  $Q$  with  $h(t')$ 
39:    Add  $t'$  to  $C[p + 1]$ 
40: end
```

```
1: method PREDICT( $t, Q$ )
2:   $b, p \leftarrow t$ 's start and current position
3:   $U \leftarrow$  set of all predictions from  $t$ 
4:  for  $u \in U$  :
5:    if  $u \notin C[p]$  :
6:      Add  $u$  to  $Q$  with  $h(u)$ 
7:      Add  $u$  to  $C[p]$ 
8:    else  $\triangleright$  Fast-track already seen items
9:       $u' \leftarrow u$  with dot one step forward
10:     for  $p' \in T[u]$  :
11:       if  $u' \notin C[p']$  :
12:         Add  $u'$  to  $Q$  with  $h(u')$ 
13:         Add  $u'$  to  $C[p']$ 
14:  end
```
