Implementation of LexDFS on Chordal Graphs

Jakub Kowalczyk
Theoretical Computer Science Department of Jagiellonian University

Abstract

Lexicographic Depth First Search (LexDFS) is an algorithm from the group of lexicographic graph searches. In these algorithms, the order of visited vertices is determined by the lexicographic order of their labels. So far, there is no general linear time implementation of LexDFS. In 2014, Köhler et al. presented a linear time algorithm on cocomparability graphs, and in 2020, Beisegel et al. achieved the same on Chordal Graphs. This algorithm can be used to find minimum colorings of chordal graphs. It also has some use cases in data mining. This paper presents a detailed linear time implementation of LexDFS on chordal graphs in Python programming language based mainly on the work of Beisegel et al.

1 Introduction

Graph searches are some of the best-known and most-studied problems in computer science. In recent years, there have been some advancements in a lesser-known subcategory of these problems called lexicographic graph searches. These algorithms rely on vertex labels and always choose the vertex with the lexicographically largest label as the next one to visit. Some labels are then updated to direct the search towards a new vertex. *Lexicographic Breadth First Search* (LexBFS) introduced in [10] was the first algorithm from this subcategory. As the name suggests, the search is a variant of the traditional *Breadth First Search* (BFS). The label update strategy of the algorithm causes it to pick the next vertex in the same order as BFS. A similar *Lexicographic Depth First Search* (LexDFS) was introduced in [5]. Because finding the lexicographically largest label from a given set cannot be done in constant time, the most straightforward implementations of lexicographic searches are not linear. However, in some cases, it is not required to actually compute and compare the labels, which leads to linear time implementations. In [6], a general case linear LexBFS was introduced based on that approach. So far, the same has not been achieved for LexDFS in the general case. An algorithm computing the LexDFS order for any graph in $O(min\{n^2, n + m \log n\})$ is presented in [8], and an algorithm with running time in $O(m \log \log n)$ is given in [12], but the article is not yet published. In [7], authors proposed a linear algorithm on cocomparability graphs, and in [1], a linear algorithm on chordal graphs was presented. These lexicographic graph search algorithms are used as a preprocessing step in some graph problems, like computing perfect elimination orders [5] or finding minimum colorings on chordal graphs [14]. This paper provides a clear implementation of the linear LexDFS algorithm on chordal graphs and the general case linear LexBFS algorithm based on the work mentioned above. In Section 2, we present the key definitions that will be used throughout the paper. Sections 3 and 4 introduce the general case LexBFS and LexDFS algorithms along with the subprocedures used in the final LexDFS algorithm on chordal graphs, which is presented in section 5. Section 6 provides additional information about the testing of the algorithms implemented in this paper. Because of its popularity and pseudocode-like syntax, Python was chosen as the implementation language. Most significant code fragments are shown

directly in the paper, and some implementation details are left in the appendices. The entire source code, together with additional tests, can be found at https://github.com/jkowalczyk08/LexDFS.

2 Definitions

In this section, we list standard theoretical definitions as well as the definitions of data structures used in the algorithms in this paper. Some Python-specific concepts are also introduced.

2.1 Theoretical Definitions

A **graph** is a pair $G = (V,\ E)$, where V is a set of vertices and E is a set of edges. $n = |V|$ is the number of vertices and $m = |E|$ is the number of edges. Vertices are numbered from 0 to $n-1$. All graphs considered in the paper are finite, undirected, simple, and connected. Because the graphs are undirected, an edge between $u \in V$ and $v \in V$ is just uv. The *neighborhood* of a vertex $v \in V$ is a set $N(v) = \{u \in V \quad | \quad uv \in E\}$, and the *neighborhood* of a subset of vertices $S \subseteq V$ is a set $N(S) = \{v \in V \setminus S \quad | \quad \exists u \in S : uv \in E\}$.

A **chordal graph** is a graph in which all cycles of more than three vertices have an edge that connects two vertices of the cycle, but it is not a part of the cycle.

A **tree** is a finite, undirected, simple, connected, and acyclic graph. A **spanning tree** of a graph G is a tree with its vertex set equal to that of G. A **rooted tree** is a tree with a distinguished vertex s, called **root**. We say that such a tree, is **rooted** in s.

A **graph search** is a procedure that visits all vertices of a graph in some order. The procedure starts at some starting vertex $s \in V$. At each step, the procedure decides which vertex it should visit next. All graph searches in this paper are **connected**, meaning that the visited vertices form a connected graph at each step of the algorithm. A **search order** $\sigma = (v_1 = s,\ v_2,\ \ldots,\ v_n)$ of a graph search is a list of vertices in the order in which they were visited by the graph search procedure. If u precedes v in σ, we denote it as $u \prec_\sigma v$. We also say that u is to the left of v in σ and v is to the right of u in σ. For a graph search procedure $\mathcal{P}$, we say that σ is a $\mathcal{P} - order$ if there exists a run of the procedure $\mathcal{P}$ on G which returns the order σ. During a graph search, we say that a vertex is **unnumbered** if its position in the search order σ is not yet known, i.e., it has not yet been visited.

A **label** is a mutable sequence $(a_1,\ a_2,\ \ldots,\ a_n)$, where $a_i \in \mathbb{N}$. Labels can be compared in a traditional, lexicographic way. Given two labels $a = (a_1,\ \ldots,\ a_n)$ and $b = (b_1,\ \ldots,\ b_m)$, a is lexicographically smaller than b if for some $i \leq min(n, m)$, for each $1 \leq j \leq i$, $a_j = b_j$ and ($i = n < m$ or $a_{i+1} < b_{i+1}$).

2.2 Python Data Structures

In the implementation of the algorithms, we use some data structures that are available out of the box in the Python language.

A **List** is an ordered, mutable collection of elements. The elements are indexed from 0 to $n-1$, where n is the number of elements in a list (also called size or length). To get the size of a list in Python, we write `len(my_list)`. To initialize a list, we use the following:

- `my_list = []`
  creates an empty list in constant time.
- `my_list = [my_variable] * n`
  creates a list containing n elements, where each element equals `my_variable`. For example,

`my_list = [0] * n` creates a list containing n integers equal to 0. The complexity of this operation is $O(n)$.

We also use the following constant time List methods:

- `my_element = my_list[i]`
  access the *i-th* element in the list.
- `my_list[i] = my_element`
  set the *i-th* element in the list to `my_element`.
- `my_list.append(my_element)`
  appends `my_element` at the end of `my_list`.

A **Dictionary** (Dict) is a mutable set of *key*: *value* pairs called entries. The *keys* within a dictionary are unique, and *value* is the value associated with a given *key*. To initialize a dictionary, we use:

- `my_dict = {}`
  creates an empty dictionary in constant time.
- `my_dict = {(key_1: value_1), ..., (key_n: value_n)}`
  creates a dictionary with n entries in linear time.

We use the following constant time dictionary methods:

- `value = my_dict.get(key, default_value)`
  get the value associated with `key` or `default_value` if there is no entry with `key` in `my_dict`.
- `my_dict[key] = value`
  add the `(key: value)` entry to `my_dict` or update the value associated with `key` in `my_dict` if an entry for this key already exists.

2.3 Custom Data Structures

In addition to the built-in Python types, we also use some custom but rather basic data structures. Their implementation can be found in Appendix D.

A **Graph** corresponds to the theoretical data structure described in section 2.1. It has two attributes:

- `n`
  number of vertices in the graph. This number is set during the construction of a graph instance and cannot be changed later.
- `adj_list`
  a list of `List[int]` lists representing the adjacency list of the graph.

It has the following methods:

- `add_edge(self, u: int, v: int)`
  adds an undirected edge between vertices `u` and `v` in constant time.
- `reorder(self, order: List[int])`
  reorders the adjacency list of the graph according to the provided order. For vertices `u, v, w`, if `u, v` both appear in `adj_list[w]` and `v` is to the right of `u` in `order`, then `v` will appear before `u`

in `adj_list[w]` after graph reordering. For example, for a graph with vertices `[0, 1, 2, 3, 4]`, `adj_list[0] == [1, 4, 3]`, and `order == [4, 2, 1, 3]`, `adj_list[0]` will be equal to `[3, 1, 4]` after reordering. This operation is performed in linear time.

A **Tree** is represented by the `Graph` type described above. If an algorithm uses a rooted tree, we use an additional variable for the root vertex in the implementation.

A **Doubly Linked List** is an ordered, mutable collection of elements of an arbitrary type `T`. Each element is stored in a **Node**, which holds the actual value of the element, as well as references to the next and previous nodes. Thanks to the linked nodes structure, `DoublyLinkedList` differs from the `List` provided by Python in the complexity of its operations. It has the following constant time methods:

- `append(self, node: Node[T])`
  appends the node to the end of the list.
- `prepend(self, node: Node[T])`
  prepends the node to the front of the list.
- `insert_before(self, node: Node[T], new_node: Node[T])`
  inserts the new node before the node in the list.
- `insert_behind(self, node: Node[T], new_node: Node[T])`
  inserts the new node after the node in the list.
- `delete(self, node: Node[T])`
  deletes a node from the list.
- `first(self) -> Node[T]`
  returns the first element in the list.
- `last(self) -> Node[T]`
  returns the last element in the list.
- `is_not_empty(self) -> bool`
  checks if the list is not empty.

An **Interval** is an interval of nodes in a `DoublyLinkedList` instance. It has the following constant time methods:

- `pop_start(self)`
  pops a node from the start of the interval.
- `pop_end(self)`
  pops a node from the end of the interval.
- `pop(self, node: Node[T])`
  pops the node from the interval.
- `is_singleton(self) -> bool`
  checks if the interval has only one node.

The methods do not change the structure of the nodes or lists to which the nodes in the interval belong. An interval instance is merely a way to describe the state of a part of a list if it is updated correctly during the algorithm.

2.4 Additional Python Concepts

Because of the pseudocode-like syntax of Python, most of the code in this paper should not require a deep understanding of the language. However some aspects require a short explanation.

- Range
  `range` type represents a sequence of numbers. `range(n)` creates a sequence `[0, 1, ..., (n-1)]`, which is usually used in a for loop `for i in range(n)`, allowing for an elegant iteration over increasing numbers.

- Enumerate
  `enumerate` is a built-in function that returns an `iterator` over `tuples` (pairs) containing the index and the value obtained from iterating over the provided sequence. For example, given a list `numbers = [3,7,5]`, `enumerate(numbers)` will give `[(0, 3), (1, 7), (2, 5)]`

- Len
  `len` is a built-in function returning the length of an object. For example, `len([3,5,7]` returns `3`.

- List Comprehensions
  List comprehensions are used for concise list creation. In this paper, it is used to initialize different lists. For example, to create a list of squares, we can use

  ```
  squares = []
  for i in range(n):
      squares.append(i*i)
  ```

  but we could also use a much more concise list comprehension

  ```
  squares = [i*i for i in range(n)]
  ```

  It is also possible to add specific conditions or to create nested lists.

- Dict Comprehensions
  Dict comprehensions are like list comprehensions but allow for the creation of dictionaries. To create a dictionary mapping each number from numbers list to its square, we can use:

  ```
  squares = {i: i*i for i in numbers}
  ```

- Negative List Indexing
  Python allows for negative list indexing, where -1 is the index of the last element, -2 is the index of the second to last element, and so on. Instead of writing `my_list[len(my_list) - 1]`, we can just use `my_list[-1]`.

- Typing
  Because Python is a dynamically typed language, it is sometimes hard to spot mistakes in code, like passing a value of a different type than expected to a function or assigning a value of a different type than expected to a variable. Because there are no type annotations in Python, it can also be harder to read, especially in more complex parts of code. To reduce errors during development and improve readability, this paper makes use of the `typing` module. Although the annotations provided by this module are not enforced by Python runtime, they are used by IDEs to display warnings and help to better understand and navigate the code base. Type annotations are used in the paper when declaring a variable whose type is not clear during creation, for example:

```
partition_refinement_pivot: List[int] = []
```

They are also used in function definitions to clearly define what is the type of arguments and the returned value:

```
def get_unvisited_neighbors(vertex: int, graph: Graph, visited: List[bool])
-> List[int]:
    partition_refinement_pivot: List[int] = []
    for neighbor in graph.adj_list[vertex]:
        if not visited[neighbor]:
            partition_refinement_pivot.append(neighbor)

    return partition_refinement_pivot
```

3 General Case BFS and DFS Variations

We now consider different BFS and DFS variations related to the LexDFS algorithm on chordal graphs. Algorithm 1 defines the general lexicographic graph search [9]. Every *LexSomething* algorithm chooses the next vertex by looking at their labels and selecting the unnumbered vertex with the lexicographically largest label. The only difference between the algorithms is how they update the labels in each iteration (line 8 of the algorithm).

Algorithm 1 General LexSomething

Input: graph $G = (V, E)$, $n = |V|$, starting vertex $s \in V$
Output: search order σ of vertices of G starting with s

1: $label(s) \leftarrow n$
2: **for each** $v \in V - s$ **do**
3: $\quad label(v) \leftarrow$ *empty label*
4: **for** $i \leftarrow 1$ to n **do**
5: $\quad$ select an unnumbered vertex v with lexicographically largest *label*
6: $\quad \sigma(v) \leftarrow i$
7: $\quad$ **for each** unnumbered vertex $u \in N(v)$ **do**
8: $\quad\quad$ update $label(u)$
9: **return** σ

The first such algorithm was LexBFS, introduced in [10] (Algorithm 2). It updates the labels by appending a number that decreases in each iteration. Note that the order produced by the LexBFS algorithm is, in fact, a BFS order. Because the number appended to the label decreases, the vertices labeled at an earlier stage of the algorithm are picked earlier, just like in the traditional BFS, where vertices added to the queue earlier are picked earlier. In the first iteration, starting vertex s updates the labels of its neighbors, i.e., the vertices at distance 1 from s. These vertices will then do the same for their unnumbered neighbors, but notice that vertices at distance 2 from s will not be picked until all vertices at distance 1 are numbered, and so on.

The analogous LexDFS algorithm was introduced in [5] (Algorithm 3). As in LexBFS, the only difference is how the algorithm updates the labels. Here, we update labels by prepending a number that increases with each iteration, so the vertices labeled later are more significant and, therefore, are picked earlier. Using a similar intuition as in LexBFS, we can see that every LexDFS order is, in fact, a DFS order.

Algorithm 2 LexBFS

Input: graph $G = (V, E)$, $n = |V|$, starting vertex $s \in V$
Output: search order σ of vertices of G starting with s

1: $label(s) \leftarrow n$
2: **for each** $v \in V - s$ **do**
3: $\quad label(v) \leftarrow$ *empty label*
4: **for** $i \leftarrow 1$ to n **do**
5: $\quad$ select an unnumbered vertex v with lexicographically largest *label*
6: $\quad \sigma(v) \leftarrow i$
7: $\quad$ **for each** unnumbered vertex $u \in N(v)$ **do**
8: $\quad\quad$ append $n - i$ to $label(u)$
9: **return** σ

Algorithm 3 LexDFS

Input: graph $G = (V, E)$, $n = |V|$, starting vertex $s \in V$
Output: search order σ of vertices of G starting with s

1: $label(s) \leftarrow n$
2: **for each** $v \in V - s$ **do**
3: $\quad label(v) \leftarrow$ *empty label*
4: **for** $i \leftarrow 1$ to n **do**
5: $\quad$ select an unnumbered vertex v with lexicographically largest *label*
6: $\quad \sigma(v) \leftarrow i$
7: $\quad$ **for each** unnumbered vertex $u \in N(v)$ **do**
8: $\quad\quad$ prepend i to $label(u)$
9: **return** σ

The general *LexSomething* algorithm, LexBFS, and LexDFS (Algorithms 1-3) do not specify what happens if multiple vertices have equal labels in line 5. of the algorithms. We call this situation a *tie*. A more specific algorithm was introduced in [11] to clarify this. The *LexSomething*$^+$ algorithms use an additional *tiebreaking order* ρ. If multiple vertices have equal, lexicographically largest labels, the algorithm chooses the rightmost vertex in ρ out of the tied vertices. The same idea can be used in traditional BFS and DFS algorithms to break the ties. We denote these algorithms as $BFS^+(\rho)$, $DFS^+(\rho)$, $LexBFS^+(\rho)$, and $LexDFS^+(\rho)$ respectively. Starting vertex s is not a parameter because choosing the starting vertex is considered the first *tie*. Therefore, ρ is used to break the *tie*.

To respect the tiebreaking order, we will start some algorithms by reordering graphs' adjacency lists according to the provided order. The DFS$^+$ algorithm serves as a simple example but is also needed to implement the final algorithm, so we present it in Python.

```python
def dfs_plus_util(graph: Graph, v: int, visited: List[bool], result: List[int]):
    visited[v] = True
    result.append(v)
    for u in graph.adj_list[v]:
        if not visited[u]:
            dfs_plus_util(graph, u, visited, result)


def dfs_plus(graph: Graph, order: List[int]) -> List[int]:
```

```
10      graph.reorder(order)
11      result: List[int] = []
12      visited = [False] * graph.n
13      s = order[-1]
14      dfs_plus_util(graph, s, visited, result)
15
16      return result
```

Note that *ties* appear in DFS and BFS only if there are multiple unvisited vertices in the neighborhood of a currently considered vertex. If the neighborhood of a vertex is sorted by the tiebreaking order, then sequential iteration over this neighborhood in line 4 will adhere to the tiebreaking order. Because we can reorder the adjacency list in linear time and the rest of the algorithm is a standard DFS, the complexity of the `dfs_plus` algorithm is also linear. Implementation of graph's reorder method is done as follows:

```
1  def reorder(self, order: List[int]) -> None:
2      new_adj_list: List[List[int]] = [[] for v in range(self.n)]
3          for u in reversed(order):
4              neighbors = self.adj_list[u]
5              for v in neighbors:
6                  new_adj_list[v].append(u)
7
8          self.adj_list = new_adj_list
```

Because we iterate over the `order` in reverse (starting from the most important vertex), for tiebreaking order σ and two vertices $u \in N(w)$, $v \in N(w)$ where $u \prec_\sigma v$, v will be appended to `new_adj_list[w]` before u, therefore v will appear before u in the final state of `new_adj_list[w]`. `new_adj_list` is then substituted for the graph's `adj_list`.

We now move to a more complex LexBFS$^+$ algorithm, which was introduced in [6]. It uses a technique called *partition refinement* to simulate a run of LexBFS$^+$ (as defined in Algorithm 2 with additional tiebreaking) without actually calculating and comparing the labels. This will allow us to achieve the linear time complexity. A *partition* P of set S is an ordered collection of disjoint subsets $A_i \subset S$ called *partition classes* or *intervals* $P = (A_1, \ ..., \ A_m)$, where $\bigcup_{i=1}^{m} A_i = S$. A general partition refinement definition is given in Algorithm 4. We use a pivot Set Q to split every partition class A_i into $A_i \cap Q$ and $A_i \setminus Q$.

Algorithm 4 General Partition Refinement

Input: partition $P = (A_1, \ ..., \ A_l)$ of a set S, pivot set $Q \subset S$
Output: refined partition $P' = (B_1, \ ..., \ B_m)$

1: **for each** interval $A_i \in P$ **do**
2: $B \leftarrow$ elements of A_i that are in Q
3: **if** B is not empty and $B \neq Q$ **then**
4: remove B from A_i
5: insert B before A_i in P
6: **return** P

Let us call the classes we create to hold elements from a given class that are also in the pivot set *new classes* or *new intervals*. If, for each element v in pivot set Q, we can find the partition class A it

belongs to, find its corresponding *new class* B, or create it and place it before A if it does not exist, remove v from A, and insert it to B in constant time, then partition refinement can be performed in $O(|S|)$ time because we do not have to iterate over the entire partition. We just pick and insert elements from Q into desired places. This approach is defined in Algorithm 5.

Algorithm 5 General Partition Refinement in $O(|S|)$

Input: partition $P = (A_1, ..., A_l)$ of a set S, pivot set $Q \subset S$, dictionary $State$ where for element $v \in A_k$ $State[v] = A_k$

Output: refined partition $P' = (B_1, ..., B_m)$, updated dictionary $State'$ where for element $v \in B_k$ $State'[v] = B_k$

```
NewClasses ← empty dictionary
for each element v ∈ Q do
    A ← State[v]
    remove v from A
    if NewClasses does not contain entry for A then
        insert a new class B before A in P
        NewClasses[A] ← B
    else
        B ← NewClasses[A]
    append v to B
    State[v] ← B
return P, State
```

We will now use this approach to implement the LexBFS$^+$ algorithm based on Algorithm 2. from [6]. Due to the complexity of the implementation, some steps were extracted into helper functions with self-explanatory names. The complete implementation can be found in Appendices A and B. To achieve linear time complexity of partition refinement, a `Partition` instance is composed of three elements:

- `vertices: DoublyLinkedList[int]`
  a list holding elements of the partition ordered by the order of intervals and the order within each interval
- `intervals: DoublyLinkedList[Interval[int]]`
  a list holding ordered intervals of the partition
- `vertex_states: Dict[int, VertexState]`
  a dictionary holding the state of each element composed of references to the vertex's current interval and node in `vertices`. We use this information to remove any element from its current interval and append it to any other interval in constant time.

For example, given a set $S = \{1, 2, 3, 4, 5, 6\}$ and a partition $P = (3, 1)(5, 2, 6)(4)$, the Python representation of P would consist of `vertices = [3, 1, 5, 2, 6, 4]`, `intervals = [(3;1), (5;6), (4;4)]` and `vertex_states[1] = {interval_node = (3;1), vertex_node = 1}` and so on. `refine` method of `partition` class performs partition refinement based on Algorithm 5. The only difference is that we do not split a singleton interval, as this is an unnecessary step in the algorithm. If we encounter a singleton interval, we proceed to the next vertex in the pivot set using the `continue` keyword.

```python
def refine(self, pivot: List[int]):
    new_intervals: Dict[Node[Interval[int]], Node[Interval[int]]] = {}

    for vertex in pivot:
        vertex_interval_node = self.vertex_states[vertex].interval_node
        vertex_node = self.vertex_states[vertex].vertex_node

        new_interval_node = new_intervals.get(vertex_interval_node, None)

        if vertex_interval_node.data.is_singleton() and new_interval_node is None:
            continue

        self.__pop_from_interval(vertex_interval_node, vertex_node)

        if new_interval_node is None:
            new_interval_node = self.__append_to_new_interval(
                vertex_node,
                self.vertex_states[vertex],
                vertex_interval_node)

            new_intervals[vertex_interval_node] = new_interval_node

        else:
            self.__append_to_existing_interval(
                vertex_node,
                self.vertex_states[vertex],
                new_interval_node)
```

In LexBFS$^+$ implementation, we maintain a partition of unvisited vertices ordered by their labels. Every interval holds the vertices that currently have lexicographically equal labels. The initial partition consists only of a single interval containing all vertices of a graph ordered according to the tiebreaking order. A helper function `prepare_initial_algorithm_state` creates this configuration. In each iteration, we take the first element in the partition called `current_vertex`, remove it from the partition, mark it as visited, and add it to the result order (equivalent to lines 5 and 6 of Algorithm 2). We then create a partition pivot set containing the unvisited neighbors of `current_vertex` using the `get_unvisited_neighbors` helper function. This set contains all vertices for which we extend the labels in this iteration. We now perform the partition refinement. Therefore, we pull the vertices with updated labels into new intervals placed closer to the beginning of the partition (equivalent to lines 7 and 8 of Algorithm 2). Notice that we start the algorithm by reordering the graph adjacency list according to the tiebreaking order. Because the partition refinement algorithm iterates over the pivot set in the order imposed by the graph's adjacency list and, therefore, also the tiebreaking order, we eliminate ties according to the tiebreaking order. Given a pivot set Q, tiebreaking order σ, and two vertices $u,\ v \in Q$ where $u \prec_\sigma v$ if both vertices are in the same interval A, then v will be appended to a new interval B before u, so we keep the new interval B sorted according to the tiebreaking order. Because the initial partition state is also sorted that way, we maintain the order throughout the algorithm.

In each iteration of the LexBFS$^+$ algorithm, we build the partition pivot set in $O(|S|)$, where S is the

adjacency list of `current_vertex`. Partition refinement based on Algorithm 5 also has $O(|S|)$ running time. Thus, the LexBFS$^+$ algorithm is linear.

```
def lex_bfs_plus(graph: Graph, tie_breaking_order: List[int]) -> List[int]:
    partition, result, visited = prepare_initial_algorithm_state(
        graph,
        tie_breaking_order)

    while partition.is_not_empty():
        current_vertex = partition.pop_first()
        visited[current_vertex] = True
        result.append(current_vertex)

        pivot = get_unvisited_neighbors(current_vertex, graph, visited)
        partition.refine(pivot)

    return result
```

4 Search Orders and Trees

A *graph search tree* is a tree associated with some graph search procedure. The tree consists of the graph's vertices and some edges determined by the graph search order. This paper will consider *first-in* and *last-in trees* defined in [2].

Definition 1. Given a search order $\sigma = (v_1, ..., v_n)$ on a graph $G = (V,\ E)$:

- the *first-in tree* ($\mathcal{F}$*-tree*) is the tree consisting of vertices V and an edge from each vertex to its leftmost neighbor in σ.
- the *last-in tree* ($\mathcal{L}$*-tree*) is the tree consisting of vertices V and an edge from each vertex v_i to its rightmost neighbor v_j in σ with $j < i$.

Both trees are rooted in v_1.

To create a tree from a search order, we use a simple algorithm based directly on the tree's definition. For example, to build an $\mathcal{L}$*-tree*, we use Algorithm 6. For each vertex v in σ, we save its position in the *position* list and find its rightmost neighbor u that has already been visited. If u exists, we add the edge uv to T. The Algorithm is linear because, in each iteration, we look for the neighbor of v with the largest *position*, which can be done in $O(|N(v)|)$ time.

Algorithm 6 *$\mathcal{L}$-tree*

Input: graph $G = (V, E)$, order σ of V
Output: *$\mathcal{L}$-tree* T of order σ

$T \leftarrow$ *empty tree*
$p \leftarrow 1$
for $i \leftarrow 1$ to n **do**
    $v \leftarrow \sigma(i)$
    $position(v) \leftarrow p$
    $u \leftarrow$ *neighbor of v with largest position*
    if u exists **then**
        add vu to T
    $p \leftarrow p + 1$
return T

```python
def last_in_tree(graph: Graph, order: List[int]) -> Graph:
    tree = Graph(graph.n)
    positions = [-1] * graph.n

    for position, vertex in enumerate(order):
        positions[vertex] = position
        neighbors = graph.adj_list[vertex]
        rightmost_visited_neighbor = find_rightmost_visited_neighbor(
            neighbors, positions)
        if rightmost_visited_neighbor is not None:
            tree.add_edge(vertex, rightmost_visited_neighbor)

    return tree

def find_rightmost_visited_neighbor(neighbors: List[int], positions: List[int]) -> int:
    rightmost_visited_neighbor: int | None = None
    position = -1

    for vertex in neighbors:
        if positions[vertex] > position:
            position = positions[vertex]
            rightmost_visited_neighbor = vertex

    return rightmost_visited_neighbor
```

There is no straightforward algorithm to generate the order from which a given *$\mathcal{L}$-tree* was created. In the final LexDFS$^+$ algorithm on chordal graphs, we will have to compute the LexDFS order based on a given *$\mathcal{L}$-tree* of LexDFS. To achieve this, the Ordering algorithm from [1] is used. The algorithm takes the input in the form of a graph $G = (V, E)$, an *$\mathcal{L}$-tree* T of LexDFS order of G rooted in s, and an order ρ of V ending in s. Partition refinement is then used to create an order going from the leaves to the root of the *$\mathcal{L}$-tree*. The reverse of that order is finally used as the tiebreaking order in a regular DFS$^+$ algorithm on the tree, resulting in the LexDFS order σ of G.

Algorithm 7 Ordering

Input: graph $G = (V, E)$, an $\mathcal{L}$-*tree* T of LexDFS order of G rooted in s, order ρ of V ending with s
Output: LexDFS order σ of G starting at s

$\beta \leftarrow$ reverse of BFS on T starting in s
partition $\leftarrow (V)$
for each $i \leftarrow 1$ to n **do**
    $v \leftarrow \beta(i)$
    pivot $\leftarrow \{w \in N(v) : w \prec_\beta v\}$
    refine *partition* with *pivot*
order vertices within each interval of *partition* according to ρ^-
move $\{s\}$ to the beginning of *partition*
$\tau \leftarrow$ reverse order of vertices in *partition*
$\sigma \leftarrow DFS^+(\tau)$ on T
return σ

The following lemmas from [5], [13], and [1] are introduced to prove the correctness of Algorithm 7.

Lemma 2. *A vertex order σ is a DFS order of a graph $G = (V, E)$ if and only if for every triple $a \prec_\sigma b \prec_\sigma c$ where $ac \in E$ and $ab \notin E$, there is a vertex d with $a \prec_\sigma d \prec_\sigma b$ such that $db \in E$.*

Lemma 3. *Given a graph $G = (V, E)$ and a spanning tree T of G, T is an $\mathcal{L}$-tree of G constructed from DFS order if and only if for each edge $uv \in E$ either u is an ancestor of v in T or v is an ancestor of u in T.*

Lemma 4. *Given an $\mathcal{L}$-tree T of some DFS order on G rooted in s and a DFS order σ of T starting in s, σ is a DFS order of G with $\mathcal{L}$-tree T.*

Proof. We start by showing that σ is a DFS order of G. Let $a, b, c \in V(G)$ be vertices with $a \prec_\sigma b \prec_\sigma c$, $ac \in E(G)$, and $ab \notin E(G)$.

If $ac \in E(T)$, then due to Lemma 2, there exists a vertex $d \in V(T) = V(G)$, such that $a \prec_\sigma d \prec_\sigma b$ and $db \in E(T) \subseteq E(G)$, thus σ is a DFS order of G.

If $ac \notin E(T)$, then from Lemma 3, a is the ancestor of c in T. Let $P = (a = w_1, w_2, ..., w_k = c)$ be the unique path from a to c in T. Since σ is a DFS order on T, $a \prec_\sigma w2 \prec_\sigma ... \prec_\sigma c$. If there exists an $1 < i < k$ such that $w_i = b$, then $a \prec_\sigma ... \prec_\sigma w_{i-1} \prec_\sigma b \prec_\sigma ... \prec_\sigma c$ and $w_{i-1}b \in E(T) \subseteq E(G)$, as b is on the path P. Thus, from Lemma 2, σ is a DFS order of G. If there is no such i, then there exists an $1 \leq i < k$ such that $w_i \prec_\sigma b \prec_\sigma w_{i+1}$ with $w_i w_{i+1} \in E(T)$, so there is a vertex $d \in V(T) = V(G)$ with $a \preceq_\sigma w_i \prec_\sigma d \prec_\sigma b$ and $db \in E(T) \subseteq E(G)$, thus due to Lemma 2, σ is a DFS order of G.

Let T^* be the $\mathcal{L}$-*tree* of σ and suppose, for contradiction, that there exists an edge $uv \in E(T^*)$ such that $uv \notin E(T)$. From Lemma 3, either u is the ancestor of v in T, or v is the ancestor of u in T. Without loss of generality, we assume that u is the ancestor of v. Let p be the parent of v in T. Note that $u \neq p$, as $uv \notin E(T)$. We know that $u \prec_\sigma p \prec_\sigma v$ because σ is a DFS on T. However, this leads to a contradiction because u cannot be the parent of v in T^* due to the Definition 1. □

We now prove the correctness and linear running time of Algorithm 7.

Theorem 5. *Let T be an $\mathcal{L}$-tree of some DFS order on $G = (V, E)$ rooted in s, and let ρ be an arbitrary order of V ending in s. Let σ be the order produced by Algorithm 7 with input (G, T, s, ρ). Then T is an $\mathcal{L}$-tree of LexDFS rooted in s if and only if σ is a LexDFS order of G.*

Proof. Suppose σ is a LexDFS order of G. From Lemma 4, the $\mathcal{L}$-*tree* of σ is T. Thus, T is an $\mathcal{L}$-*tree* of a LexDFS order of G rooted in s.

For the reverse implication, let σ^* be the LexDFS order of G with the maximal common prefix of σ and σ^*, such that the $\mathcal{L}$-*tree* of σ^* T. Assume that $\sigma \neq \sigma^*$. Let i be the first index, where both orders do not match, i.e., $v = \sigma(i) \neq \sigma^*(i) = v^*$, and let σ_i be the prefix of the first $i-1$ elements that match in σ and σ^* on each position. We know from Lemma 4 that σ and σ^* are DFS orders of G, and both v and v^* have the same parent p in T. Suppose v and v^* have the same neighbors in σ_i. If this is the case, both vertices have equal labels during the selection of the next vertex in the *i-th* iteration of LexDFS, so the algorithm can select v instead of v^*. Due to Lemma 3, this change would not alter the $\mathcal{L}$-*tree* T. This contradicts the selection of σ^* as the LexDFS order of G with $\mathcal{L}$-*tree* T, which has the maximal common prefix with σ. Therefore, there exists a vertex in σ_i that is a neighbor of v^* in G but not a neighbor of v in G. Let w be the rightmost such vertex in σ_i. From Lemma 3, either v^* is the ancestor of w in T or w is the ancestor of v^* in T. Because w is in σ_i, v^* cannot be the ancestor of w due to Definition 1. Furthermore, w must also be an ancestor of v, as v and v^* have the same parent in T. Thus, w must be to the right of v and v^* in β.

We now analyze how v and v^* are positioned during the partition refinement. Because w is the rightmost vertex in σ, such that $wv^* \in E(G)$ and $wv \notin E(G)$, v and v^* stay in the same partition class until the iteration of the for loop in line 3 of Algorithm 7, when w is the currently considered vertex. In that iteration, v^* is in the partition pivot set $P = \{x \in N(w) \mid x \prec_\beta w\}$ and v is not, therefore v^* is moved the the partition class to the left of v. Later in Algorithm 7, a tiebreaking order τ is created from the reverse order of vertices in partition, so $v \prec_\tau v^*$. Because v and v^* have the same parent in T, v^* is then selected before v in $\mathrm{DFS}^+(\tau)$ on T, therefore $v^* \prec_\sigma v$. This is a contradiction to $v \prec_\sigma v^*$. □

In [1], there is a small mistake in the pseudocode of the Ordering algorithm because DFS^+ is performed on G instead of T. However, the authors correctly write about DFS^+ on T in the proof.

Lemma 6. *Algorithm 7 is linear.*

Proof. BFS algorithm is linear, and from previous sections, we know that DFS^+ algorithm is also linear. To efficiently build the partition pivot sets, we create a helper list holding the position of each vertex in β. To construct the pivot set for vertex v, a simple iteration over $N(v)$ is enough because the helper list can be used to determine in constant time if a neighbor $w \in N(v)$ is to the left of v in β. These steps are omitted from the pseudocode of Algorithm 7 to maintain clarity but are available in the Python implementation. Partition refinement defined in Algorithm 5 also takes $O(|N(v)|)$, so the total running time of the for loop in lines 3-6 is linear.

To reorder vertices within each interval of the partition according to some order ρ of V in linear time, we iterate over ρ, and for each vertex v, v is removed from its current interval I and then appended again to I. This operation is done in constant time for a single vertex due to partition internal representation based on doubly linked lists and a dictionary holding the state of each vertex. □

The Implementation uses DFS^+ and partition refinement algorithms, which were presented in previous sections. Additional `reversed_bfs_order_positions` helper list, `get_pivot_set` helper function, and `order_within_intervals` method of `Partition` class are based on the concepts presented in Lemma 6. Some functions are omitted and available in appendices A and C.

```
1 def ordering(graph: Graph, last_in_tree: Graph, s: int, order: List[int])
2 -> List[int]:
3     reversed_bfs_order = reverse(bfs(last_in_tree, s))
4     reversed_bfs_order_positions = get_positions(reversed_bfs_order)
5     partition, s_node = initialize_partition(graph.n, s)
```

```
6
7       for vertex in reversed_bfs_order:
8           pivot = get_pivot_set(graph, reversed_bfs_order_positions, vertex)
9           partition.refine(pivot)
10
11      partition.order_within_intervals(reverse(order))
12      partition.move_to_front(s_node)
13      reversed_vertices = reverse(partition.get_vertices())
14      result = dfs_plus(last_in_tree, reversed_vertices)
15
16      return result
17
18  def get_pivot_set(graph: Graph, reversed_bfs_order_positions: List[int], vertex: int)
19  -> List[int]:
20      pivot: List[int] = []
21      vertex_position = reversed_bfs_order_positions[vertex]
22
23      for neighbor in graph.adj_list[vertex]:
24          if reversed_bfs_order_positions[neighbor] < vertex_position:
25              pivot.append(neighbor)
26
27      return pivot
```

```
1   def order_within_intervals(self, order: List[int]):
2       for vertex in order:
3           vertex_state = self.vertex_states[vertex]
4           vertex_node = vertex_state.vertex_node
5           vertex_interval_node = vertex_state.interval_node
6           vertex_interval = vertex_interval_node.data
7
8           if vertex_interval.is_singleton():
9               continue
10
11          self.__pop_from_interval(vertex_interval_node, vertex_node)
12          self.__append_to_existing_interval(
13              vertex_node, vertex_state, vertex_interval_node)
```

The following lemma from [1] regarding Algorithm 7 is also presented and is used in the next section.

Lemma 7. *Let T be an $\mathcal{L}$-tree of a DFS of G rooted in s, let ρ be an arbitrary order of V ending in s, and let σ be the order produced by Algorithm 7 with input (G, T, s, ρ). Furthermore, let v and w be two vertices in G with $v \prec_\sigma w$, which have the same parent in T and the same neighborhood in the set $Y = \{x \mid x \prec_\sigma v\}$. Then $w \prec_\rho v$.*

Proof. If $v \prec_\sigma w$, then from the definition of σ, DFS$^+(\tau)$ algorithm on T must have taken v before w. Because both vertices have the same parent in T, v must be to the right of w in the tiebreaking order τ.

We now analyze the positions of v and w in the partition throughout the for loop in lines 3-6 of Algorithm 7. If v is pulled from its partition class by a vertex x, then $vx \in E(G)$, and x is closer to the root s of T than v. From Lemma 3, x must be the *ancestor* of v in T, therefore $x \prec_\sigma v$, and so $x \in Y$. Vertices v and w have the same neighbors in Y from assumption. Thus, $wx \in E(G)$ and w is also pulled from its partition class by x. All vertices start in one partition class, so v and w are in the same class after the analyzed for loop.

τ is the reverse order of vertices in the partition, where vertices in every class are ordered according to ρ. Thus, if $w \prec_\tau v$, v must be to the right of w in ρ. □

5 LexDFS on Chordal Graphs

To construct the final LexDFS$^+$ algorithm on chordal graphs, we will use algorithms from previous sections together with some relevant facts and results about chordal graphs from cited articles. In [3], *CompLexSomething* algorithms were introduced. These algorithms replace line 5 of general Algorithm 1 with "Choose a connected component C of a graph induced by the unnumbered vertices and take the vertex in C with lexicographically largest label". The authors also present the following lemma, which we will use as a "bridge" between the LexBFS and LexDFS orders on chordal graphs in the final algorithm.

Lemma 8. *For any chordal graph G, a linear vertex order is a CompLexDFS order if and only if it is a CompLexBFS order.*

Furthermore, we show, based on [1], that for any graph G, LexDFS order of G has the same $\mathcal{L}$-*tree* as CompLexDFS order of G, and LexBFS order of G has the same $\mathcal{L}$-*tree* as CompLexBFS order of G.

Lemma 9. *A spanning tree T of a graph $G = (V, E)$ rooted in s is an $\mathcal{L}$-tree of LexDFS order on G if and only if T is an $\mathcal{L}$-tree of CompLexDFS order on G.*

Proof. We know from the definitions of LexDFS and CompLexDFS that every LexDFS order of G is a CompLexDFS order of G. Thus, every $\mathcal{L}$-*tree* of LexDFS order of G is also an $\mathcal{L}$-*tree* of CompLexDFS order of G.

For the reverse implication, we start by introducing the following definition. Given an order $\tau = (w_1, ..., w_n)$ of $V(G)$, $C_\tau(w_i)$ is the connected component of induced graph $G - \{w_1, ..., w_{i-1}\}$, that contains w_i. Let σ be the CompLexDFS order of G with $\mathcal{L}$-*tree* T, and let σ^* be the LexDFS$^+(\sigma^-)$ order of G. We now show that T is also the $\mathcal{L}$-*tree* of σ^*.

For every vertex $v \in V$, $C_\sigma(v) = C_{\sigma^*}(v)$, and for every $w \in C_\sigma(v) = C_{\sigma^*}(v)$, the label of w at the point when v is picked as the next vertex in σ equals the label of w at the point when v is picked in σ^*. To prove this observation, suppose that it is not true. Let v be the leftmost vertex in σ for which these properties do not hold. Let w be the rightmost vertex in σ, such that $w \prec_\sigma v$ and w has a neighbor in $C_\sigma(v)$. Because $w \prec_\sigma v$, we know from the selection of v that $C_\sigma(w) = C_{\sigma^*}(w)$, and for every $u \in C_\sigma(w) = C_{\sigma^*}(w)$, the label of u at the point when w is picked as the next vertex in σ equals the label of u at the point when w is picked in σ^*. Note that for any unnumbered vertex y and the connected component C containing y, the label of y can only be updated by vertices in C. Therefore, there must exist a vertex $x \in C_\sigma(v)$, with $w \prec_{\sigma^*} x \prec_{\sigma^*} v$. Let x be the leftmost such vertex in σ^*. Because x is the leftmost vertex in σ^* with this property, when x is selected as the next vertex in σ^*, the label of x is the same as the label of x when v is selected in σ. Similarly, when x is selected in σ^*, the label of v is the same as the label of v when v is selected in σ. If both x and v are in $C_\sigma(v)$ and v is selected in σ, then the label of v must be lexicographically greater or equal than the label of x. On the other hand, if x is selected instead of v in σ^*, then the label of x must be

lexicographically greater or equal than the label of v. Thus, both labels must be equal when x is selected in σ^*. We know that $v \prec_\sigma x$, so $x \prec_{\sigma^-} v$. This means, that if x and v have equal labels in LexDFS$^+(\sigma^-)$, then x cannot be selected before v; therefore, we reach a contradiction.

Let T^* be the $\mathcal{L}$-*tree* of σ^* and assume for contradiction, that $T^* \neq T$. Under this assumption, there exists a vertex $y \in V(G)$, such that p is the parent of y in T, p^* is the parent of y in T^*, and $p \neq p^*$. From the definition of $\mathcal{L}$-*tree*, $p \prec_\sigma y$ and $p^* \prec_{\sigma^*} y$. It follows from the observation from the previous paragraph that p and p^* must be to the left of y in both σ and σ^*. Therefore, $p \prec_{\sigma^*} p^*$, as p^* is the parent of y in T^*, and similarly, $p^* \prec_\sigma p$. Thus $p \in C_\sigma(p^*)$ and $p \notin C_{\sigma^*}(p^*)$, so we reach a contradiction. □

Lemma 10. *A spanning tree T of a graph $G = (V,\ E)$ rooted in s is an $\mathcal{L}$-tree of LexBFS order on G if and only if T is an $\mathcal{L}$-tree of CompLexBFS order on G.*

Proof. Proof of Lemma 9 used only properties of general LexSomething and CompLexSomething algorithms, so the same result is true for LexBFS and CompLexBFS algorithms. □

By combining results from Lemmas 8, 9, and 10, we get the following corollary that is essential for the final algorithm.

Corollary 11. *Given a chordal graph $G = (V,\ E)$ and a spanning tree T of G rooted in $s \in V$, T is an $\mathcal{L}$-tree of LexDFS order of G if and only if T is an $\mathcal{L}$-tree of LexBFS order of G.*

For any chordal graph G, we can get the $\mathcal{L}$-*tree* of LexDFS order by calculating the $\mathcal{L}$-*tree* T of LexBFS order. From Lemma 10, we know that T is also an $\mathcal{L}$-*tree* of CompLexBFS order on G. From Lemma 8, we know that T is an $\mathcal{L}$-*tree* of CompLexDFS order on G. Finally, from Lemma 9, we know that T is an $\mathcal{L}$-*tree* of LexDFS order on G. To get the LexDFS order from which T was constructed, we use the Algorithm 7 from the previous section. This approach is shown in Algorithm 8. The order calculated by this algorithm is actually the LexDFS$^+$ order, which we prove in Theorem 13. For the proof, we also give the following

Algorithm 8 *LexDFS*$^+$ on chordal graphs

Input: graph $G = (V, E)$, starting vertex $s \in V$, order ρ of V ending with s
Output: *LexDFS*$^+(\rho)$ order σ of G starting with s
1: $\pi \leftarrow$ *LexBFS*$^+(\rho)$ on G
2: $T \leftarrow \mathcal{L}$-*tree* of π
3: $\sigma \leftarrow$ *ordering*$(G,\ T,\ s,\ \rho)$
4: **return** σ

four point property of every LexDFS order introduced in [5].

Lemma 12. *A vertex order σ is a LexDFS order of a graph $G = (V,\ E)$ if and only if for every triple $a \prec_\sigma b \prec_\sigma c$ where $ac \in E$ and $ab \notin E$, there is a vertex d with $a \prec_\sigma d \prec_\sigma b$ such that $db \in E$ and $dc \notin E$.*

Theorem 13. *Given a chordal graph $G = (V,\ E)$, a vertex $s \in V$, and an order ρ of V ending in s as input, Algorithm 8 returns a LexDFS$^+(\rho)$ order of G starting at s.*

Proof. We start by showing that the order σ returned by the algorithm is a LexDFS order of G starting in s. From Corollary 11, we know that T is an $\mathcal{L}$-*tree* of LexDFS order on G. Furthermore, s is the rightmost vertex in ρ, so it is selected as the first vertex in π, therefore due to Definition 1, s is the root of T. Thus, from Theorem 5, we know that σ is a LexDFS order of G starting at s.

We now prove that σ is, in fact, the LexDFS$^+(\rho)$ order of G. Let σ^* denote the LexDFS$^+(\rho)$ order of G, and assume that $\sigma \neq \sigma^*$. Let i be the first index, where both orders do not match, i.e., $v = \sigma(i) \neq \sigma^*(i) = v^*$,

and let σ_i be the prefix of the first $i-1$ elements that match in σ and σ^* on each position. Many facts about v and v^* are now shown, which will eventually lead to a contradiction, meaning that σ is, in fact, the LexDFS$^+(\rho)$ order of G.

LexDFS and LexDFS$^+(\rho)$ selected the same vertices in the first $i-1$ iterations of the algorithm. The difference at the *i-th* position means that v and v^* had equal labels at this point and were tied to be the vertex visited in the next iteration. Their equal labels mean that both labels were updated during the same iterations and, therefore, by the same vertices in σ_i, as the number prepended to the label changes in each iteration, so v and v^* have the same neighbors in σ_i. It follows that $v \prec_\rho v^*$ because both vertices have equal labels when v^* is selected instead of v as the next vertex in LexDFS$^+(\rho)$.

Assume $v \prec_\pi v^*$. We know that $v \prec_\rho v^*$, so v must have had a lexicographically larger label than v^* when it was picked as the next vertex in π. Therefore, there exists a vertex $w \in V$, such that $w \prec_\pi v \prec_\pi v^*$ with $wv \in E(G)$ and $wv^* \notin E(G)$. From Lemma 3, w is an ancestor of v in T, so it must be in σ_i. This is a contradiction because v and v^* have the same neighbors in σ_i. Therefore, $v^* \prec_\pi v$. Presume that v and v^* have the same parent in T. We know that $v \prec_\sigma v^*$ and both vertices have the same neighbors in σ_i, so from Lemma 7, $v^* \prec_\rho v$, which is a contradiction.

Let $p \in V$ be the parent of v in T, which is not a parent of v^* in T. p is in σ_i because it is a parent of v in T, and due to Definition 1, it has to be to the left of v in σ. We know that v and v^* have the same neighbors in σ_i, so from Lemma 3, v^* is a descendant of p in T. Let x be the unique child of p, that is the ancestor of v^* in T. $x \prec_\pi v^*$ because x is the ancestor of v^* in T. x cannot be equal to v, because then $v = x \prec_\pi v^*$, which is a contradiction. It follows that $x \prec_\pi v^* \prec_\pi v$. Assume $x \prec_\sigma v$. σ is a DFS order on T starting at s, so when p is the currently visited vertex, the algorithm chooses x before v. This means that the algorithm will reach v^* before v, which is a contradiction with $v \prec_\sigma v^*$. Therefore, $v \prec_\sigma x$. Note that every neighbor q of x that is to the left of x in π is in σ_i. From Lemma 3, we know that q must be the ancestor of x in T. Because x and v have the same parent p in T, q must also be an ancestor of v. σ is a DFS order on T, so q must be to the left of v in σ. Assume for contradiction, that x has the same neighborhood in σ_i as v and v^*. This means that when x is selected as the next vertex in π, vertices x, v and v^* have equal labels. Therefore, x must be to the right of v^* in ρ, as $x \prec_\pi v^*$. We also know that $v \prec_\rho v^*$, so $v \prec_\rho x$. On the other hand, $v \prec_\sigma x$, so due to Lemma 7, $x \prec_\rho v$; therefore, we reach a contradiction.

Let y be the rightmost neighbor of x in σ_i, which is not a neighbor of v or v^*. We know from selection of y that $y \prec_\sigma v \prec_\sigma x$ and $yx \in E(G)$ and $yv \notin E(G)$. From Lemma 12, there exists a vertex $z \in V$, such that $y \prec_\sigma z \prec_\sigma v$ with $vz \in E(G)$ and $xz \notin E(G)$. Because v and v^* have the same neighbors in σ_i, $zv^* \in E(G)$. By applying Lemma 12 again for $z \prec_\sigma x \prec_\sigma v^*$, we know that there is a vertex $u \in E(G)$ such that $z \prec_\sigma u \prec_\sigma x$ with $ux \in E(G)$ and $uv^* \notin E(G)$.

We now analyze the order of the introduced vertices in σ. By gathering the facts from this proof, we know that $y \prec_\sigma z \prec_\sigma v \prec_\sigma x \prec_\sigma v^*$ and $z \prec_\sigma u \prec_\sigma x$. Additionally, $ux \in E(G)$ and $uv^* \notin E(G)$. If $u \prec_\sigma v$, then $uv \notin E(G)$, since v and v^* have the same neighbors in σ_i. In that case, u would be chosen instead of y as the rightmost neighbor of x in σ_i, which is not a neighbor of v or v^*, so u has to be to the right of v in σ. Thus, the order of vertices is $y \prec_\sigma z \prec_\sigma v \prec_\sigma u \prec_\sigma x \prec_\sigma v^*$. Because p is the parent of v in T, p has to be to the left of v in σ, so $p \prec_\sigma u$. However, $ux \in E(G)$, so p cannot be chosen as the parent of x in T instead of u; therefore, we reach a contradiction to the selection of p. □

Lemma 14. *Algorithm 8 has a linear running time.*

Proof. As described in the previous sections, all three algorithms used in Algorithm 8 are linear, so the final algorithm is also linear. □

The implementation is rather straightforward and based directly on Algorithm 8. The algorithms used in `lex_dfs_plus_on_chordal` algorithm were described in previous sections.

```
def lex_dfs_plus_on_chordal(graph: Graph, tie_breaking_order: List[int]) -> List[int]:
    s = tie_breaking_order[-1]
    lex_bfs_plus_order = lex_bfs_plus(graph, tie_breaking_order)
    tree = last_in_tree(graph, lex_bfs_plus_order)
    lex_dfs_order = ordering(graph, tree, s, tie_breaking_order)

    return lex_dfs_order
```

6 Testing

Algorithms are tested using the Unittest Python package. Most of the tests are dedicated to the two main algorithms, i.e., LexBFS$^+$ and LexDFS$^+$ on chordal graphs. Tests of these algorithms share the same structure. A graph is generated based on some predefined text input. Two versions of the same algorithm are then performed on the graph. One is a linear version of LexBFS$^+$ (or LexDFS$^+$ on Chordal graphs) described in this paper, and the other is a nonlinear simple version based directly on Algorithm 2 (or 3). The results of these two versions are then compared and the test passes if both orders are equal. Because the algorithms reorder graphs' adjacency lists, the tests also shuffle the lists and verify the equality of search orders multiple times to ensure the result is not a false positive due to a lucky test input. An example of this test structure is given below.

```
def test_lex_dfs_plus_on_chordal(self, n: int):
    graph = build_chordal_graph(n)
    tie_breaking_order = list(range(n))
    for i in range(10):
        shuffle_adj_lists(graph)
        order = lex_dfs_plus_on_chordal(graph, tie_breaking_order)
        correct_order = simple_lex_dfs_plus(graph, tie_breaking_order)
        self.assertEqual(correct_order, order)
```

`simple_lex_bfs_plus` and `simple_lex_dfs_plus` both have a clear implementation, where only the label update strategy is passed to the general `simple_lex_something_plus` algorithm.

```
def lex_bfs_update_strategy(label: List[int], n: int, i: int) -> List[int]:
    return label + [n - i - 1]

def simple_lex_bfs_plus(graph: Graph, tie_breaking_order: List[int]) -> List[int]:
    return simple_lex_something_plus(graph, tie_breaking_order, lex_bfs_update_strategy)
```

```
def lex_dfs_update_strategy(label: List[int], n: int, i: int) -> List[int]:
    return [i] + label
```

```python

def simple_lex_dfs_plus(graph: Graph, tie_breaking_order: List[int]) -> List[int]:
    return simple_lex_something_plus(graph, tie_breaking_order, lex_dfs_update_strategy)
```

```python
def simple_lex_something_plus(
        graph: Graph,
        tie_breaking_order: List[int],
        update_strategy: Callable[[List[int], int, int], List[int]]
) -> List[int]:
    n = graph.n
    tie_breaking_order_map = {
        vertex: position for position, vertex in enumerate(tie_breaking_order)
    }
    label: List[List[int]] = [[] for i in range(n)]
    order = [-1] * n
    is_numbered = [False] * n
    s: int = largest_unnumbered_label(n, label, is_numbered, tie_breaking_order_map)
    label[s] = [n]

    for i in range(n):
        v = largest_unnumbered_label(n, label, is_numbered, tie_breaking_order_map)
        order[i] = v
        is_numbered[v] = True
        for w in graph.adj_list[v]:
            if not is_numbered[w]:
                label[w] = update_strategy(label[w], n, i)
    return order
```

The NetworkX Python package is used to create graphs for test input. The graphs are saved in text format, to not regenerate them during each test run. The size ranges from 5 to 250 vertices. There are no direct methods in the package to create the required inputs. However, the generation is not complicated. A connected graph for LexBFS input is created using the Erdős-Rényi model, where for each pair of vertices u and v, an edge uv is added to the graph with probability p. This method does not guarantee that a graph is connected, but for $p > \frac{\ln(n)}{n}$, the graph is almost definitely connected. For each graph, we verify that it is indeed connected before saving it to text format. Chordal graphs for LexDFS input are created by generating a cycle graph and then extending its adjacency list using the NetworkX `complete_to_chordal_graph` method. This method uses the MCS-M algorithm introduced in [4] to transform the graph into a chordal graph. Other algorithms used in LexBFS and LexDFS implementations are tested using simpler graphs generated manually for each test.

References

[1] Jesse Beisegel et al. "Linear Time LexDFS on Chordal Graphs". en. In: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. DOI: 10.4230/LIPICS.ESA.2020.13. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ESA.2020.13.

[2] Jesse Beisegel et al. "Recognizing Graph Search Trees". In: *Electronic Notes in Theoretical Computer Science* 346 (2019). The proceedings of Lagos 2019, the tenth Latin and American Algorithms, Graphs and Optimization Symposium (LAGOS 2019), pp. 99–110. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2019.08.010. URL: https://www.sciencedirect.com/science/article/pii/S157106611930060X.

[3] A. Berry, R. Krueger, and G. Simonet. "Maximal Label Search Algorithms to Compute Perfect and Minimal Elimination Orderings". In: *SIAM Journal on Discrete Mathematics* 23.1 (2009), pp. 428–446. DOI: 10.1137/070684355. eprint: https://doi.org/10.1137/070684355. URL: https://doi.org/10.1137/070684355.

[4] Anne Berry et al. "Maximum Cardinality Search for Computing Minimal Triangulations of Graphs". In: *Algorithmica* 39 (Aug. 2004), pp. 287–298. DOI: 10.1007/s00453-004-1084-3.

[5] Derek G. Corneil and Richard M. Krueger. "A Unified View of Graph Searching". In: *SIAM Journal on Discrete Mathematics* 22.4 (2008), pp. 1259–1276. DOI: 10.1137/050623498. URL: https://doi.org/10.1137/050623498.

[6] Michel Habib et al. "Lex-BFS and partition refinement, with applications to transitive orientation, interval graph recognition and consecutive ones testing". In: *Theoretical Computer Science* 234.1 (2000), pp. 59–84. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(97)00241-7. URL: https://www.sciencedirect.com/science/article/pii/S0304397597002417.

[7] Ekkehard Köhler and Lalla Mouatadid. "Linear Time LexDFS on Cocomparability Graphs". In: *CoRR* abs/1404.5996 (2014). arXiv: 1404.5996. URL: http://arxiv.org/abs/1404.5996.

[8] Richard M. Krueger. "Graph Searching". PhD thesis. University of Toronto, 2005. URL: http://www.cs.toronto.edu/~krueger/papers/thesis.ps.

[9] Arthur Milchior. *(Quasi-)linear time algorithm to compute LexDFS, LexUP and LexDown orderings.* 2017. arXiv: 1701.00305 [cs.DS]. URL: https://arxiv.org/abs/1701.00305.

[10] Donald J. Rose, R. Endre Tarjan, and George S. Lueker. "Algorithmic Aspects of Vertex Elimination on Graphs". In: *SIAM Journal on Computing* 5.2 (1976), pp. 266–283. DOI: 10.1137/0205021. URL: https://doi.org/10.1137/0205021.

[11] Klaus Simon. "A new simple linear algorithm to recognize interval graphs". In: *Computational Geometry-Methods, Algorithms and Applications*. Ed. by H. Bieri and H. Noltemeier. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 289–308. ISBN: 978-3-540-46459-4.

[12] Jeremy P. Spinrad. "Efficient implementation of lexicographic depth first search".

[13] Robert Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010. URL: https://doi.org/10.1137/0201010.

[14] Shou-Jun Xu, Xianyue Li, and Ronghua Liang. "Moplex orderings generated by the LexDFS algorithm". In: *Discrete Applied Mathematics* 161.13 (2013), pp. 2189–2195. ISSN: 0166-218X. DOI: https://doi.org/10.1016/j.dam.2013.02.028. URL: https://www.sciencedirect.com/science/article/pii/S0166218X13001157.

A Partition Class Implementation Details

```
class Partition:
    def __init__(self, vertices: DoublyLinkedList[int]):
        self.vertices = vertices

        self.intervals: DoublyLinkedList[Interval[int]] = DoublyLinkedList()
        initial_interval = Node(Interval(vertices.first(), vertices.last()))
        self.intervals.prepend(initial_interval)
        self.vertex_states = {
            node.data: VertexState(initial_interval, node) for node in vertices
        }

    def is_not_empty(self) -> bool:
        return self.intervals.is_not_empty()

    def get_vertices(self) -> List[int]:
        return [vertex.data for vertex in self.vertices]

    def pop_first(self) -> int:
        interval_node = self.intervals.first()
        interval = interval_node.data

        vartex_node = interval.start
        vertex = vartex_node.data

        if interval.is_singleton():
            self.intervals.delete(interval_node)
        else:
            interval.pop_start()

        self.vertices.delete(vartex_node)
        self.vertex_states[vertex].interval_node = None

        return vertex

    def move_to_front(self, vertex_node: Node[int]):
        vertex_state = self.vertex_states[vertex_node.data]
        vertex_interval_node = vertex_state.interval_node

        self.__pop_from_interval(vertex_interval_node, vertex_node)
        self.__prepend_to_existing_interval(
            vertex_node, vertex_state, self.intervals.first())

    def __pop_from_interval(
```

```
        self,
        interval_node: Node[Interval[int]],
        vertex_node: Node[int]):

    interval = interval_node.data

    if interval.is_singleton():
        self.intervals.delete(interval_node)
    else:
        interval.pop(vertex_node)

    self.vertices.delete(vertex_node)

def __append_to_new_interval(
        self,
        vertex_node: Node[int],
        vertex_state: VertexState,
        original_interval_node: Node[Interval[int]]
) -> Node[Interval[int]]:

    new_interval_node = Node(Interval(vertex_node, vertex_node))
    self.intervals.insert_before(original_interval_node, new_interval_node)
    self.vertices.insert_before(original_interval_node.data.start, vertex_node)
    vertex_state.interval_node = new_interval_node

    return new_interval_node

def __append_to_existing_interval(
        self,
        vertex_node: Node[int],
        vertex_state: VertexState,
        existing_interval_node: Node[Interval[int]]
):
    existing_interval = existing_interval_node.data
    existing_interval_end = existing_interval.end
    self.vertices.insert_behind(existing_interval_end, vertex_node)
    existing_interval.end = vertex_node
    vertex_state.interval_node = existing_interval_node

def __prepend_to_existing_interval(
        self,
        vertex_node: Node[int],
        vertex_state: VertexState,
        existing_interval_node: Node[Interval[int]]
):
```

```
        existing_interval = existing_interval_node.data
        existing_interval_start = existing_interval.start
        self.vertices.insert_before(existing_interval_start, vertex_node)
        existing_interval.start = vertex_node
        vertex_state.interval_node = existing_interval_node

class VertexState:
    def __init__(self, interval_node: Node[Interval[int]], vertex_node: Node[int]):
        self.interval_node: Node[Interval[int]] = interval_node
        self.vertex_node: Node[int] = vertex_node
```

B $LexBFS^+$ Implementation Details

```
def prepare_initial_algorithm_state(
        graph: Graph,
        tie_breaking_order: List[int]
) -> Tuple[Partition, List[int], List[bool]]:

    vertices: DoublyLinkedList[int] = DoublyLinkedList()
    graph.reorder(tie_breaking_order)
    for vertex in tie_breaking_order:
        vertices.prepend(Node(vertex))

    partition = Partition(vertices)
    result: List[int] = []
    visited = [False] * graph.n

    return partition, result, visited


def get_unvisited_neighbors(vertex: int, graph: Graph, visited: List[bool]) -> List[int]:
    partition_refinement_pivot: List[int] = []
    for neighbor in graph.adj_list[vertex]:
        if not visited[neighbor]:
            partition_refinement_pivot.append(neighbor)

    return partition_refinement_pivot
```

C *Ordering* Implementation Details

```
def initialize_partition(n: int, s: int) -> Tuple[Partition, Node[int]]:
    vertices: DoublyLinkedList[int] = DoublyLinkedList()
    s_node: Node[int]

    for i in range(n):
```

```
        node = Node(i)
        vertices.append(node)
        if i == s:
            s_node = node

    return Partition(vertices), s_node


def get_positions(order: List[int]) -> List[int]:
    positions = [-1] * len(order)

    for position, vertex in enumerate(order):
        positions[vertex] = position

    return positions
```

D Data Structures Implementation Details

```
class Graph:
    def __init__(self, n: int) -> None:
        self.n: int = n
        self.adj_list: List[List[int]] = [[] for v in range(n)]

    def add_edge(self, u: int, v: int) -> None:
        self.adj_list[u].append(v)
        self.adj_list[v].append(u)

    def reorder(self, order: List[int]) -> None:
        new_adj_list: List[List[int]] = [[] for v in range(self.n)]
        for u in reversed(order):
            neighbors = self.adj_list[u]
            for v in neighbors:
                new_adj_list[v].append(u)

        self.adj_list = new_adj_list

    def __eq__(self, other) -> bool:
        if not isinstance(other, Graph):
            return NotImplemented

        if self.n != other.n:
            return False

        for u in range(self.n):
            self_neighbors = self.adj_list[u]
```

```
            other_neighbors = other.adj_list[u]
            if set(self_neighbors) != set(other_neighbors):
                return False

        return True

class Node(Generic[T]):
    def __init__(self, data: T):
        self.data: T = data
        self.prev: Optional[Node[T]] = None
        self.next: Optional[Node[T]] = None

class DoublyLinkedList(Generic[T]):
    def __init__(self):
        self.head: Optional[Node[T]] = None
        self.tail: Optional[Node[T]] = None

    def append(self, node: Node[T]):
        if self.head is None:
            self.head = node
            self.tail = node
        else:
            self.tail.next = node
            node.prev = self.tail
            self.tail = node

    def prepend(self, node: Node[T]):
        if self.head is None:
            self.head = node
            self.tail = node
        else:
            self.head.prev = node
            node.next = self.head
            self.head = node

    def insert_before(self, node: Node[T], new_node: Node[T]):
        new_node.next = node
        new_node.prev = node.prev
        if node.prev is not None:
            node.prev.next = new_node
        node.prev = new_node

        if self.head is node:
            self.head = new_node

    def insert_behind(self, node: Node[T], new_node: Node[T]):
```

```
        new_node.next = node.next
        new_node.prev = node
        if node.next is not None:
            node.next.prev = new_node
        node.next = new_node

        if self.tail is node:
            self.tail = new_node

    def delete(self, node: Node[T]):
        if self.head is None:
            return

        if self.head is node:
            self.head = node.next

        if self.tail is node:
            self.tail = node.prev

        if node.next is not None:
            node.next.prev = node.prev

        if node.prev is not None:
            node.prev.next = node.next

        node.next = None
        node.prev = None

    def first(self) -> Node[T]:
        if self.head is None:
            raise Exception('Empty List')
        return self.head

    def last(self) -> Node[T]:
        if self.tail is None:
            raise Exception('Empty List')
        return self.tail

    def is_not_empty(self) -> bool:
        return self.head is not None

    def __iter__(self):
        self._current_node = self.head
        return self
```

```
    def __next__(self) -> Node[T]:
        if self._current_node is None:
            raise StopIteration
        next_result = self._current_node
        self._current_node = self._current_node.next
        return next_result

class Interval(Generic[T]):
    def __init__(self, start: Node[T], end: Node[T]):
        self.start: Node[T] = start
        self.end: Node[T] = end

    def pop_start(self):
        self.start = self.start.next

    def pop_end(self):
        self.end = self.end.prev

    def pop(self, node: Node[T]):
        if node is self.start:
            self.start = self.start.next
        elif node is self.end:
            self.end = self.end.prev

    def is_singleton(self) -> bool:
        return self.start == self.end
```