CS762: Graph-Theoretic Algorithms
Lecture 6: Recognizing Chordal Graphs
January 18, 2002

Scribe: Philip Yang and Therese Biedl

Abstract

Given a chordal graph, we have seen the algorithms LexBFS and MCS to find a perfect elimination order. In this lecture, we will show details of how to implement the LexBFS algorithm and a brief proof of the correctness. Also, we will see that checking whether the result of LexBFS is in fact a perfect elimination order can be done in $O(m+n)$ time.

1 Introduction

Every chordal graph has a perfect elimination order. We will show that finding such a order can be done in $O(m+n)$ running time with LexBFS. Recall that LexBFS works by assigning labels to vertices and removing repeatedly the vertex with lexicographically smallest label and updating its neighbours. Doing this in a straightforward way may require more than linear time. To get a linear time bound, we need to use special data structure which will give us $O(1)$ time to find or update a label.

To test whether a graph is chordal, we also need to check whether the order that is returned by the LexBFS algorithm is a perfect elimination order. A naive approach is to check for every vertex v whether its predecessors form indeed a clique, i.e., to check for every pair $v_i, v_j \in Pred(v)$ whether (v_i, v_j) is an edge. This takes $O(\sum_{v \in V}(deg(v)^2) = O(mn)$ running time. But in fact, one can show that with the right order of queries, every edge need not be checked more than twice in a perfect elimination order, and we can hence obtain an algorithm which achieves $O(n+m)$ time and space bound.

2 Complexity of LexBFS

We first analyze LexBFS. Recall the code for LexBFS that was given last time:

1. For all vertices v, set $L(v) = \emptyset$;
2. For $i = n \ldots 1$
3. among all vertices $\neq v_{i+1}, \ldots, v_n$
4. pick up v_i with the lexicographically largest label L(v_i);
5. for each unnumbered vertex w that is adjacent to v
6. Set $L(w) = L(w) \circ i$

We might want to point out that LexBFS algorithm is a breadth first searching algorithm. Take a look at the searching tree, the neighbors of v_n will be explored before those of v_{n-1} by lexicographical order. The only difference between LexBFS and BFS is that LexBFS imposes a specific order of neighbors of v_n.

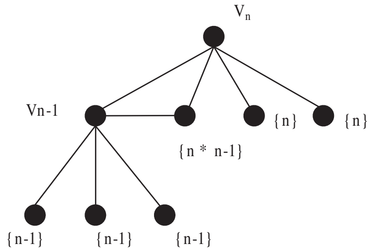

Figure 1: Breadth First Searching Tree

2.1 Data Structure

To implement LexBFS efficiently, we use a linked list data structure, which each node in this list Q is a pointer to another linked list, which here we call a Bucket. List S_l contains all vertices v with $L(v) = l$. Q will never contain an empty list. The buckets within Q are sorted by lexicographic order, with the largest label first. We will illustrate this on an example on the graph shown in Figure 2.

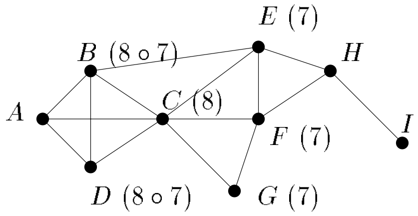

Figure 2: Sample Graph

Suppose we picked up vertex A first and labelled all its neighbors; at the second step, we picked up vertex C and also updated all its neighbors. We now see what is left in the list Q in Figure 3.

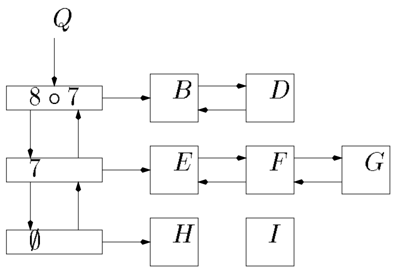

Figure 3: queue data structure

In fact, additionally every vertex knows which bucket contains it (i.e., vertex v has a reference to bucket $S_{L(v)}$) and where it is in this bucket. Furthermore, each bucket knows its place in Q. Note that all lists are doubly-linked for easier insertion and removal.

For the actual implementation of this data structure, note that initially all vertices have label $\emptyset$. Thus, Q contains just one bucket ($S_\emptyset$) which contains all vertices. Clearly, this can be initialized in $O(n)$ time.

Now to obtain the next vertex, and to update the data structure, we proceed as follows:

1. Let v_i be the first vertex in the first bucket.
    (Since Q is sorted, v_i has the lexicographically largest label.)
2. Delete v_i from its bucket $S_{L(v_i)}$.
3. Delete $S_{L(v_i)}$ if it is now empty.
4. For all neighbors w of v_i
5. If w is still in Q
    (Note that w has not been chosen yet; we need to update its label and therefore its place in Q.)
6. Find $S_{L(w)}$ and its place in Q
7. Find the bucket that precedes $S_{L(w)}$ in Q
8. If there is no such bucket, or if this is not $S_{L(w)\circ i}$
9. Create bucket $S_{L(w)\circ i}$ at this place in Q.
10. Remove w from $S_{L(w)}$ and insert it into $S_{L(w)} \circ i$
11. Delete $S_{L(w)}$ if it is now empty
12. Update $L(w) = L(w) \circ i$

Now we analyze the running time of this algorithm. To get v_i takes $O(1)$ time, since we only have to find the first vertex in the first bucket of the linked list. Removing the vertex from the bucket takes also constant time since updating a doubly-linked list takes only $O(1)$ time.

To update Q, we need to update all unnumbered neighbors of v_i. This takes $O(1)$ time for each neighbor, since we have stored all the necessary reference with each vertex. Thus the total cost for updating the neighbors of v_i is $O(deg(v_i))$. Combining both, the total running time and space for LexBFS is $O(|V| + \sum_{v_i \in V} deg(v_i)) = O(m + n)$.

It is not entirely obvious why the space requirement is also linear. Note that the length of the label of each vertex might be $\Omega(n)$, for example if we have a complete graph. But note that we add to the label of a vertex w only if we choose a neighbour v. Hence, the length of the label of each vertex is proportional to its degree, and the total space for the labels is also $O(n + m)$.

We claim that this algorithm indeed works, i.e., it produces a perfect elimination order if the graph has one.

Theorem 1 *Let $G = (V, E)$ be a graph, and let $\{v_1, \ldots, v_n\}$ be the vertices chosen by lexBFS (i.e., v_n was chosen first). If G is chordal, then $\{v_n, \ldots, v_1\}$ is a perfect elimination order.*

Proof: All we have to do is to show that v_1 is simplicial, then we can show the rest by induction. We will only sketch this proof here, for details see [Gol80].

Suppose v_1 is not simplicial, thus there exist two neighbours v_i and v_j of v_1 that do not have an edge (v_i, v_j) between them. Without loss of generality, assume $j > i$, so v_j comes before v_i in the (supposed) perfect elimination order. See Figure 4 for an illustration.

Note that both v_i and v_j were therefore chosen during lexBFS before we chose v_1. When we chose v_j, we added j to the labels of all its neighbours that weren't chosen yet. In particular, we added j to $L(v_1)$, but we did *not* add j to $L(v_i)$, since v_i is not a neighbour of v_j. Thus we know:

$$
\begin{aligned}
L(v_1) &= \cdots j \cdots \\
L(v_i) &= \cdots \cancel{j} \cdots
\end{aligned}
$$

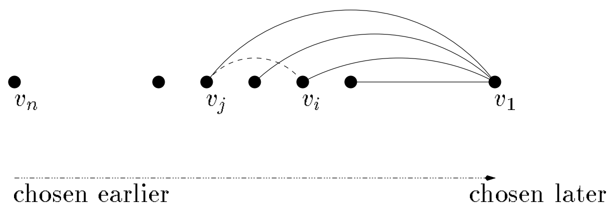

Figure 4: There are two neighbours of v_1 without an edge between them.

Observe that v_i was chosen by LexBFS before v_1 was chosen. Thus, the label of v_i must have been lexicographically not smaller than the label of v_1. But since $L(v_1)$ contains j and $L(v_i)$ does not, this is possible only if at some point earlier in the label, $L(v_i)$ is larger than $L(v_1)$. In other words, somewhere earlier there must be an index (say k) that is contained in $L(v_i)$ and not in $L(v_1)$. So we must have

$$\begin{aligned} L(v_1) &= \cdots \cancel{k} \cdots j \cdots \\ L(v_i) &= \cdots k \cdots \cancel{j} \cdots \end{aligned}$$

Therefore, there must have been some vertex v_k, with $k > j$, that was incident to v_i but not to v_1. See Figure 5.

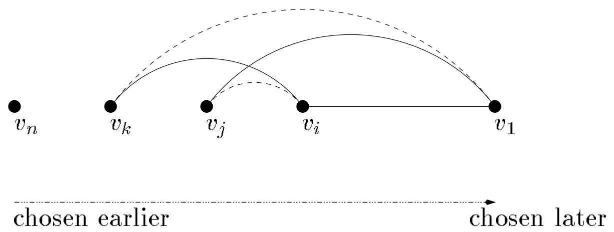

Figure 5: There must have been a vertex v_k before v_j that is incident to v_i but not to v_1.

Note that we cannot have edge (v_j, v_k), for if there were such an edge, then G would have a 4-cycle v_1, v_i, v_k, v_j without a chord, which contradicts that G is chordal. So there is no such edge.

Now we repeat this argument. The label of v_i contains k whereas the label of v_j does not contain k. So why was v_j chosen before v_i by the lexBFS? There must have been yet another vertex before v_k that is incident to v_j, but not to v_i. With a lot more arguing (here is where the details are omitted), we can show that this vertex also isn't incident to any of v_k and v_1.

And then we repeat the argument again. Why was v_k chosen before v_j? And we get another vertex before v_k. And we repeat the argument again. And again. And again. What we end up with is the construction shown in Figure 6.

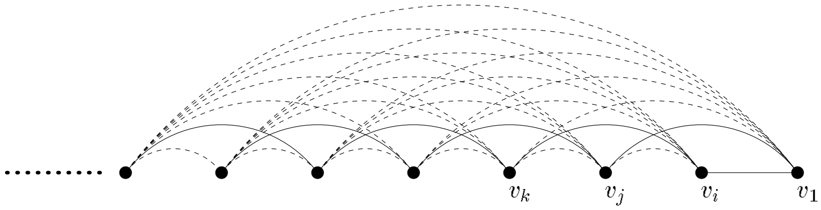

Figure 6: We can find more and more vertices, a contradiction.

This argument can be repeated ad infinitum, always adding another vertex that comes earlier in the ordering. But G is a finite graph, so this is a contradiction. $\square$

3 Testing a perfect elimination order

Theorem 1 proves that LexBFS returns a perfect elimination order if the given G is chordal. To recognize a chordal graph, we only need to test whether the LexBFS result is a perfect elimination order. As explained in the introduction, a naive implementation will give us $O(mn)$ running time. Here we present an algorithm which takes only $O(m+n)$ time.

The idea is as follows. Assume that vertex v has a number of predecessors. Let u be the last of those predecessors. If we have a perfect elimination order, then the predecessors of v are a clique, and so in particular u must be adjacent to all other predecessors of v. We will test that. But once we have tested that, we need not test for any other edges between the predecessors of v! For if these other predecessors of v are also predecessors of u (recall that u is the last of v's predecessors), then we will test whether they are all adjacent when we test whether all predecessors of u form a clique.

The pseudo-code of an efficient algorithm is therefore as follows:

Input: A graph $G=(V,E)$ and a vertex ordering $v_1,\dots,v_n$
Output: "TRUE" if and only if $v_1,\dots,v_n$ is a perfect elimination order

1. for $j=n$ down to 1 do
2.   if v_j has predecessors
3.   Let u be the last predecessor of v_j.
4.   Add $Pred(v_i)-\{u\}$ to $Test(u)$.
               ($Test(u)$ denotes the multi-set of vertices for which
               we want to test whether they are neighbours of u.)
5.   (Now test $Test(v_j)$.)
6.   Mark all vertices in Pred(v_j) as touched
7.   for every vertex w in $Test(v_j)$,
8.   if w is not touched, return FALSE.
9.   Mark all vertices in Pred(v_j) is untouched
10. return TRUE

We will illustrate this algorithm on the graph shown in Figure 7.

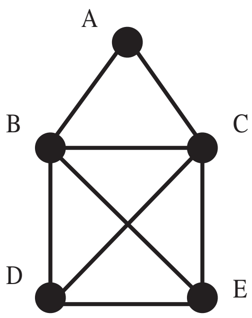

Figure 7: Another sample graph

Assume that we want to test the ordering $\{A, D, B, C, E\}$, which is not a perfect elimination order.[1] Running the algorithm will give the following result:

- 1st iteration: $Pred(E) = \{B, C, D\}$, $u = C$, $Test(C) = \{B, D\}$. Since $Test(A) = \emptyset$, we find no error and continue.
- 2nd iteration: $Pred(C) = \{A, B, D\}$, $u = B$, $Test(B) = \{A, D\}$ Since $Test(C) = \{B, D\}$, and C is adjacent to both, we find no error and continue.
- 3rd iteration: $Pred(B) = \{A, D\}$, $u = D$, $Test(D) = \{A\}$. Since $Test(B) = \{A, D\}$ and B is adjacent to both, we find no error and continue.
- 4th iteration: $Pred(D) = \emptyset$, so no changes to the test-sets. However, $Test(D) = \{A\}$, but A is not adjacent to D, so we return FALSE.

3.1 Correctness of the Algorithm

Theorem 2 *This algorithm returns TRUE if and only if $v_1, \ldots, v_n$ is a perfect elimination order.*

Proof: Suppose the algorithm returns FALSE. This only happens when there exists a vertex $w \in Test(v_i) - Pred(v_i)$. Now w was added to $Test(v_i)$ because both w and v_i were predecessors of some other vertex v_j. Since $w \notin Pred(v_i)$, therefore the predecessors of v_j are not a clique and $v_1, \ldots, v_n$ is not a perfect elimination order.

Now suppose $v_1, \ldots, v_n$ is not a perfect elimination order but the algorithm returns TRUE. Let i be minimal such that the predecessors of v_i, and let v_j and v_k with $j < k$ be two predecessors of v_i that are not adjacent to each other. Let u be the last predecessor of v_i. By choice of i, the predecessors of u are a clique, so in particular v_j and v_k cannot both be predecessors of u. But unless one of the is u, they would both have been added to $Test(u)$ (and the algorithm would have returned FALSE at u), so one of them must be u. Say $u = v_j$. Now we added v_k (among others) to $Test(u)$ while handling v_i. Therefore, when testing $Test(u)$ (while handling u), we will discover that $v_k \in Test(u)$, but $v_k \notin Pred(u)$, a contradiction to that the algorithm returns TRUE. □

One can observe that the entire algorithm can be performed in time and space proportional to

$$|V| + \sum_{v \in V} |Adj(v)| + \sum_{u \in V} |Test(u)|$$

Thus we must analyze how big $Test(u)$ can be. For each $v \in V$, the algorithm only adds $Pred(v)$ to one of the lists $Test(u)$. Thus for each vertex v, we increase $\sum_{u \in V} |Test(u)|$ by at most $\deg(v)$. Therefore, overall we have $\sum_{u \in V} |Test(u)| \in O(\sum_{v \in V} \deg(v)$, which proves that the running time and space requirements of this algorithm is $O(m + n)$.

Combining therefore lexBFS with this algorithm to test whether the resulting ordering is indeed a perfect elimination order, we obtain our main result:

Theorem 3 *Chordal graphs can be recognized in linear time.*

[1] This order would not have been obtained via lexBFS, because the sample graph actually is chordal. But we can apply the order-testing algorithm to any ordering we like, all it does is to test whether a given ordering is a perfect elimination order.

References

[Gol80] Martin Charles Golumbic. *Algorithmic graph theory and perfect graphs*. Academic Press, New York, 1980.