Interval Graphs

Structure vs Algorithmic Aspects:
Which Problems Become Easy and Which Remain Hard?

Côme Périn

December 22, 2024

1 Introduction

Definition 1.1. ***Intersection Graph***
An Intersection graph G is formed from a family of sets $(S_i)_i$. Each set S_i is represented by a vertex v_i in G. Two vertices v_i and v_j are adjacent in G if and only if $S_i \cap S_j \neq \emptyset$ [8].

Proposition 1.1. *Every graph is an intersection graph.*

Proof. Let G be a graph. G is an intersection graph with the family of sets $(S_i)_i$ where S_i is the set of neighbors of the vertex v_i in G. □

Definition 1.2. ***Interval Graph***
An interval graph is an intersection graph of a set of intervals on the real line $\mathcal{R}$ [9].

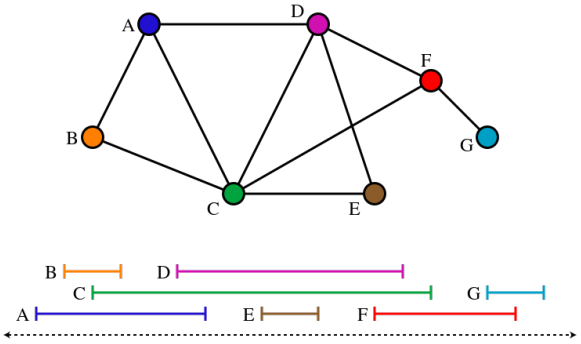


Figure 1: Some intervals and their corresponding interval graph [7].

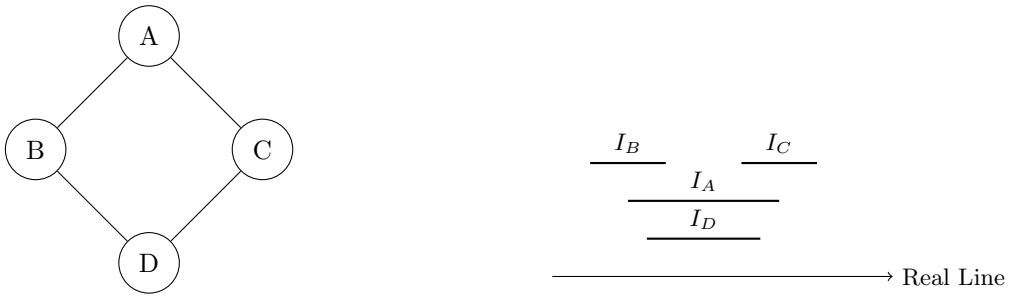
2 Chordality and Perfection

Definition 2.1. ***Chord***
A chord of a cycle C is an edge which is not in C but is adjacent to vertices of C [2].

Definition 2.2. ***Chordal Graph :***
A graph G is chordal if every cycle of length four or more in G has a chord in G [2].

Theorem 2.1. *Intervals graphs are chordal graphs.*

Proof. Let G be an interval graph.
Let C be a cycle of G of length ≥ 4 *without any chords.*
Without loss of generality we can consider that the cycle is of length 4.



(a) C_4. (b) Interval representation of C_4.

Figure 2: Interval representation of a cycle C_4.

According to the definition of a C_4 (see fig. 2a), we must have $I_B \cap I_D \neq \emptyset$ and $I_C \cap I_D \neq \emptyset$. It is clear that looking at fig. 2b, this implies that $I_A \cap I_D \neq \emptyset$ which is a contradiction.

□

Definition 2.3. ***Perfect elimination ordering***
A perfect elimination (PEO) of a graph G is an ordering of the vertices of G such that for each vertex v, v and the neighbors of v succeding v in the ordering, form a clique [14].

Example. *a perfect elimination ordering of the graph featured in fig. 1 is* `GFEDCAB`

Theorem 2.2. *Let G be a graph. The following are equivalent:*

1. *G is chordal.*
2. *G has a perfect elimination ordering.*

Proof. The proof is given by [14]. □

Proposition 2.1. *A PEO ordering of a graph is computable in linear time [4].*

Proof. Several processes can be used to compute a PEO, this proof is given by [4] and uses the following *maximum cardinality search* algorithm.

Let G be a graph with $|V(G)| = n$.

Listing 1: Maximum Cardinality Search

```
W ← V(G)
∀v ∈ W, weight(v) ← 0
for i in [1, ..., n] do :
    u ← v ∈ W where weight(v) = max{weight(v') | v' ∈ W}
    v_{n−i+1} ← u
    for i in neighbors(u, W) do :
        weight(w) ← weight(w) + 1
    remove(u, W)
return v_1, v_2 ... v_n
```

□

Definition 2.4. ***Perfect graph***
A graph G is perfect if, for every induced subgraph H of G the chromatic number of H equals the size of the largest clique $\omega(G)$ in H [5].

Lemma 2.1. *Let G be a chordal graph, then $\mathcal{X}(G) = \omega(G)$.*

Proof. Let G be a chordal graph.
According to theorem 2.2, G has a *perfect elimination ordering* $(v_i)_{i \in [\![1, V(G)]\!]}$.

Let be $\{c_i\}_i = [\![1, \omega(G)]\!]$ a set of colors.
Let us use a greedy algorithm to color G :

Listing 2: Greedy algorithm for coloring a chordal graph

```
for k in [|V(G)|, ... , 2] do
    v_k ← smallest_available_color_among_neighbors_for(v_k, {c_i}_i)
```

This greedy algorithm gives a correct coloring of G providing that there are enough colors in $\{c_i\}_i$.
According to the definition of the PEO v_k as at most $\omega(G) - 1$ neighbors after itself in the PEO.
Thus, $\mathcal{X}(G) \leq \omega(G)$ which leads to $\mathcal{X}(G) = \omega(G)$. □

Lemma 2.2. *Let H be a subgraph of G. If G is chordal then H is chordal.*

Proof. Let suppose G is chordal and H is not chordal.
Then there exists a cycle c of length more than 4 and without chords in H.
This cycle also exists in G but it has at least one chord (G being chordal).
So this chord has been removed in H, but H is an **induced subgraph**, so an adjacent vertex to the chord have been removed.
So c is not a cycle in H, which is a contradiction.

□

Theorem 2.3. *Interval graphs are perfect graphs*

Proof. Let G be an interval graph and H a subgraph of G.
According to theorem 2.1, G is chordal, so, using lemma 2.2, H is chordal.
Moreover, using lemma 2.1, we have $\mathcal{X}(H) = \omega(H)$.

Finally, G is perfect. □

3 Recognition, Reconstruction and Optimization

Proposition 3.1. *Given $n \in \mathbb{N}$ intervals $\{I_i\}_{i\in[\![1,n]\!]}$ the related interval graph is constructible in $O(n\log(n))$.*

Proof. Each starting or ending point is sorted in an array ($O(n\log n)$, merge sort for instance) :
e.g. [Start_1, Start_2, End_1, Start_3, End_3, End_2]. While reading this array of event ($O(n)$), an *active set* is updated :

- If an interval starts, it is added to the *active set*. A vertex representing the interval is added to the graph and an edge is drawn between this vertex and all other vertices representing intervals of the *active set*.
- If an interval ends, it is removed from the *active set*.

If two interval overlap, it is clear that they will be together in the *active set* at a given moment and thus, they would be adjacent in the graph. This procedure operates $O(n\log(n))$.
NB. A counting sort might be used ensuring a better complexity of $O(n+k)$ where k is a constant (this is not detailed here). □

Theorem 3.1. *Interval graphs are recognizable in linear time.*

Proof. The proof is given by [3] and will not be detailed here. However, the characterization of interval graphs used is the following :
A graph G is an interval graph if and only if there exists an ordering of the cliques of G such that $\forall v \in V(G)$, elements of $C(v)$ are consecutive in this ordering.
$C(v)$ denotes the set of cliques containing v.

[13] summarizes the idea of the proof by giving the following recognition algorithm :

Listing 3: Interval graph recognition algorithm

```
if G is not chordal then return False #linear
Π ← all ordering of the cliques of G
for each v ∈ V(G) do
    remove from Π all ordering where
    elements of C(v) are not consecutive
if Π is not empty then return True
return False
```

The complexity of this algorithm mostly depends on the size of the set Π. Booth and Lueker have introduced **PQ-trees** to represent the set of all possible orderings of the cliques of a graph and thus operating the algorithm in linear time. See section 4.4 for more details.

□

Corollary 1. *Given an interval graph G, a set of intervals that generates G is constructible in linear time.*

Corollary 2. *Given an interval graph G, it is possible to find an interval reconstruction in linear time (corollary 1) and to operate dynamic programmation on these interval as it can be more efficient than operating on the graph (*e.g. `DOMINATION` *section 4.3).*

Theorem 3.2. *A graph G is an interval graph if and only if it does not contains a subgraph which is one the graphs* $\mathrm{I}, \mathrm{II}, \mathrm{III}_n, \mathrm{IV}_n, \mathrm{V}_n$.

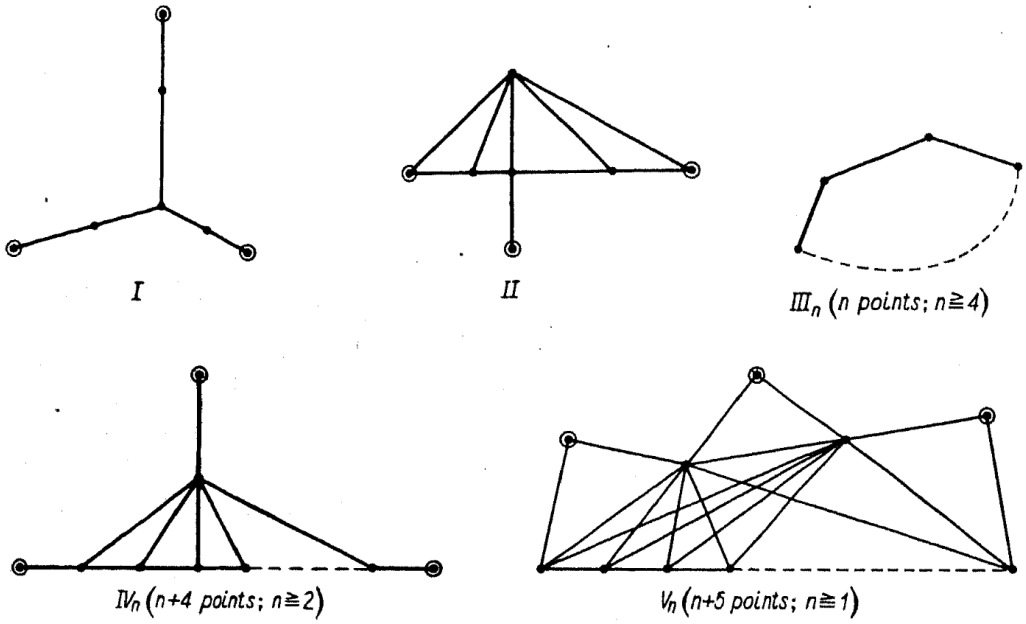


Figure 3: The five forbidden subgraphs [12].

Proof. It was one the first characterization of interval graphs, given in 1962 by Lekkerkerker & Boland [12]. □

4 Complexity : What is easier and what remains hard

Some famous problems are known to be NP-hard for general graphs. We will study the complexity of some of these problems when restricted to interval graphs.

4.1 CLIQUE

This section aims to study the CLIQUE problem on interval graphs.

Problem 4.1. `CLIQUE`
`INPUT :` *A graph G and a positive integer k.*
`OUTPUT :` *True if G has a clique of size k, False otherwise.*

`CLIQUE` is NP-complete for general graphs

Theorem 4.1. *`CLIQUE` restricted to interval graphs is linear.*

Proof. Let G be an interval graph.
According to theorem 2.1, G is chordal, so we can find a PEO of G (theorem 2.2) in linear time $O(|V(G)| + |E(G)|)$ (proposition 2.1).

Let us consider the following procedure :
For each vertex v in the PEO ($O(|V(G)|)$), compute neighbors of v succeeding v in the PEO ($O(|E(G)|)$ after all calculations).
If v has more than $k-1$ neighbors, return `True`.
If there is no vertex left to explore, return `False`.

So the complexity of the procedure is $O(|V(G)| + |E(G)|)$.

The procedure is correct according to the definition of PEO : if there is a clique in G, the clique will appear in the procedure as soon as one of the vertex of the clique is explored. □

NB. theorem 4.1 states a result for problem `CLIQUE` (problem 4.1) which is a decision problem, however, a polynomial reduction of the "instance searching" version of `CLIQUE` to `CLIQUE` (problem 4.1) is easily found. Idem with the famous `MAX_CLIQUE` problem.

4.2 COLORING

Problem 4.2. *`COLORING`*
`INPUT :` *A graph G*
`OUTPUT :` *A coloring function $\phi : V(G) \to [\![1, \mathcal{X}(G)]\!]$*

`COLORING` is NP-complete for general graphs [10].

Theorem 4.2. *`COLORING` restricted to interval graphs is linear.*

Proof. Let G be an interval graph.
G is chordal (theorem 2.1) so we can find a PEO (theorem 2.2) in $O(|V(G)| + |E(G)|)$ (proposition 2.1). Using listing 2 ($O(|V(G)| + |E(G)|)$) with enough colors, we get a coloring of G
This coloring uses $\omega(G)$ colors which is the minimal value (theorem 2.3). □

Exercise 1. *Show that `MAX INDEPENDENT SET` is linear for interval graphs.*
Hint : Consider a PEO of the graph.

4.3 DOMINATION

Problem 4.3. *`DOMINATION`*
`INPUT :` *A graph G and a positive integer k.*
`OUTPUT :` *True iff there exists a set D of vertices of G such that $|D| \leq k$ and each vertex of G is in D or adjacent to a vertex of D.*

`DOMINATION` is NP-complete for general graphs [10].

Theorem 4.3. *`DOMINATION` restricted to interval graphs is linear.*

Proof. The following proof is given by [15]. Algorithm 4 takes a set of intervals as an input. However, as is it seen in corollary 1, it is possible to construct an set of intervals matching an interval graph in linear time as seen in corollary 1. Moreover, the presented proof show the result for `MINIMUM WEIGHT DOMINATION`, but the proof is easily adaptable to `DOMINATION` by putting the same weight on all vertices.

Let us clarify the notations used in Algorithm 4:

- a_i and b_i: The left and right endpoints of interval i, respectively. The interval is represented as $I_i = [a_i, b_i]$.
- $IFB(a_i)$: The set of intervals that finish before the left endpoint a_i, i.e., $IFB(a_i) = \{I_j : b_j < a_i\}$.
- $ISB(b_i)$: The set of intervals that start before the right endpoint b_i, i.e., $ISB(b_i) = \{I_j : a_j < b_i\}$.
- $\max a(IFB(a_i))$: The largest left endpoint among the intervals in $IFB(a_i)$. If $IFB(a_i)$ is empty, this value is defined as 0.
- $W(i)$: The cumulative weight of the minimum partial dominating set $MPD(i)$ computed up to interval i.
- $pred(b_i)$: The predecessor of b_i in the list L of right endpoints.

The intervals are assumed to be sorted by their right endpoints b_i, which simplifies comparisons and enables efficient scans. This sorting is achievable in $O(n)$ time using a counting sort or a similar algorithm, as the endpoints can be constrained to integers between 1 and $2n$ using known interval graph constructions [15].

`INPUT :` A set I of sorted intervals.
`OUTPUT :` a $O(n)$ space date structure such that a $MPD(i)$ can be outpout in $O(|\text{structure}|)$.

Listing 4: Dominating set algorithm

```
1   Find max a(IFB(a_i))  ∀i ∈ I;
2
3   #Scan the endpoints of I to find left endpoints sets
4   {a_j : b_{i-1} < a_j < b_i}  ∀i ∈ I, using b_0 = 0;
5   #Each right endpoint set b_i is associated
6   #with the left endpoint set {a_j : b_{i-1} < a_j < b_i};
7   L ← {b_1, b_2, ..., b_n};
8   W(1) ← w(1); MPD(1) ← {1};
9   for i = 2 to n do
10      Find the set containing max a(IFB(a_i));
11      Let b_k be the right endpoint associated with this set;
12      MDP(i) ← MDP(k) ∪ {i}; #union is implemented
13      by setting a pointer from i to k.
14      W(i) ← w(i) + W(k);
15      while W(interval(pred(b_i))) > W(i) do
16          Unite the set associated with pred(b_i)
```

```
17                  to the set associated with b_i by union operation;
18                  Delete pred(b_i) from L;
19              end while
20          end for
```

[15] introduces the lemma above to prove correction of algorithm 4.
, However we will not prove it for the sake of brevity.

Lemma 4.1. *The following statements are true :*

1. $MDP(1) = 1$
2. $\forall i$ *such that* $2 \leq i \leq n$, $MDP(i) = \{i\} \cup Min\{MDP(j) : max\ a(IFB(a_i)) < b_j < b_i\}$

We prove the correction of the algorithm 4 by induction.

The correctness of Algorithm 4 can be proved by induction. Algorithm MPD processes the intervals one by one in increasing order of their right endpoints b_i.

Invariants maintained by the algorithm: Let L' denote the set of right endpoints e in L such that $e < b_i$. The algorithm ensures the following invariants:

1. For any two endpoints $f, h \in L'$, if $f < h$, then $W(interval(f)) \leq W(interval(h))$.
2. If the set containing a left endpoint e_1 is associated with the right endpoint e_2, then e_2 is the successor of e_1 in L.
3. For every endpoint e removed from L, there exists an endpoint $e' \in L$ such that $e < e'$ and $W(interval(e)) > W(interval(e'))$.

Initially, all three invariants hold because L is initialized in increasing order, no intervals are merged, and no endpoints are removed. These invariants are maintained throughout the algorithm, especially in the while-loop on lines 15 to 17, where predecessors are merged if their weights exceed the weight of the current interval.

Base case: For $i = 1$, the algorithm sets $MPD(1) = \{1\}$ and $W(1) = w(1)$. This is correct as interval I_1 must dominate itself.

Inductive step: Assume the algorithm computes $MPD(k)$ and $W(k)$ correctly for all $k < i$. For i, the algorithm:

1. Finds the interval k such that $\max\ a(IFB(a_i)) < b_k < b_i$. By invariants 1 and 2, $MPD(k) = \min\{MPD(j) : \max\ a(IFB(a_i)) < b_j < b_i\}$.
2. Sets $MPD(i) = MPD(k) \cup \{i\}$, ensuring that I_i is included in the dominating set and extends the domination to I_k.
3. Updates $W(i) = w(i) + W(k)$ to reflect the cumulative weight.

By invariant 3, merging predecessors maintains the minimality of the dominating set. The correctness of the algorithm follows by induction.

Final domination problem: To solve the domination problem on interval graphs, the interval set I is extended to $I_d = I \cup \{I_0, I_{n+1}\}$, where I_0 dominates the leftmost part of the real line and I_{n+1} dominates the rightmost part. A subset $S \subseteq I$ is a dominating set of $G(I)$ if and only if $S \cup \{I_0, I_{n+1}\}$ is a dominating set of $G(I_d)$.

The minimum-weight dominating set of $G(I)$ is computed as $MPD(n+1)\backslash\{I_0, I_{n+1}\}$. By Lemma 2.6, $MPD(n+1)$ can be computed in $O(n)$ time and space, leveraging Algorithm P to calculate $\max a(IFB(a_i))$ in $O(n)$ time (this algorithm is only detailed in [15]). The overall complexity of the algorithm remains $O(n)$.

□

4.4 ISOMORPHISM

Problem 4.4. *ISOMORPHISM*
INPUT : *Two graphs G_1 and G_2.*
OUTPUT : *True iff there exists a bijection $f : V(G_1) \to V(G_2)$ such that $\forall u, v \in V(G_1),\ (u, v) \in E(G_1) \Leftrightarrow (f(u), f(v)) \in E(G_2)$.*

ISOMORPHISM is NP-complete for general graphs [10].

Theorem 4.4. *ISOMORPHISM restricted to interval graphs is linear.*

Proof. The proof is given by [13]. The whole proof will not be detailed since it is quite long, however, mains ideas will be drawn here.

Definition 4.1. ***PQ-trees :*** *"A PQ-tree is an ordered tree whose non-terminal (*i.e *all nodes except leaves) are either of class Q or P. Two trees T, T' are said to be equivalent written $T = T'$ if one may be obtained from the other by applying any combination (possibly none) of the following two classes of transformations, called* equivalence transformations*: " [13]*

- *arbitrary permutation of the children of a P-node.*
- *reversing the ordering of the children of a Q-node.*

*"A **PQ-tree** is proper if each P-node has at least two children, and each Q-node has at least three children. The frontier of a PQ-tree is the ordering of its leaves obtained by reading them from left to right. The **frontier** of a **PQ-tree** is the ordering of its leaves obtained by reading them from left to right. An ordering of the leaves of T is **consistent** with T if it is the frontier of a tree equivalent to T. The set of all orderings consistent with T is called the **consistent set** of T, and is denoted **CONSISTENT(T)**. " [13]*

The following lemma is an extension of theorem 3.1.

Lemma 4.2. *Let G be an interval graph. There is a linear time algorithm to construct a proper **PQ-tree** T such that **CONSISTENT(T)** is precisely the set of ordering of the cliques in which $\forall v \in V(G)$, elements of $C(v)$ appear consecutively.*

[13] shows that isomorphic graphs will have equivalent **PQ-trees**, however, the reciprocal is not true. *Labelled* **PQ-trees** and *L_identity* are then introduced.

The following theorem in conjugation with **linear** algorithms of transformations from a **PQ-tree** to a *labelled* **PQ-tree** leads to the wanted result.

Theorem 4.5. *Two graphs G_1 and G_2 are isomorphic if and only if their labelled **PQ-trees** are equivalent.*

□

4.5 MAXIMUM CUT

Problem 4.5. *MAXIMUM CUT*
INPUT : A graph G and an integer k OUTPUT : True iff the vertices of G can be partitioned into two sets A, B such that there are at least k edges in G with one endpoint in A and the other endpoint in B.

MAXIMUM CUT is NP-complete for general graphs [10].

Theorem 4.6. *MAXIMUM CUT restricted to interval graphs is NP-complete.*

Proof. The proof is given by [1] operating a reduction from MAXIMUM CUT on cubic graphs to MAXIMUM CUT on interval graphs.

□

Remark : In 2024, no polynomial time algorithm is known for MAXIMUM CUT on interval graphs [6].

5 Interval Scheduling Problem

This section aims to bring an application of interval graphs in scheduling problems in order to link the theoretical results to potential practical applications.

5.1 Basic Interval Scheduling Problem

Basic interval scheduling problem is a well-known problem resource allocation problem.

Let be n jobs that each need to be scheduled on a single machine during some time (positive) intervals $\{I_i\}_{i\in[\![1,n]\!]}$. Interrupting and resuming jobs is not allowed. The basic interval scheduling problem is to process these jobs on a minimal number of machines [11].

It is clear that this problem can be modeled as an interval graph problem.
Actually, it corresponds to finding a minimal coloring of the associated interval graph [9].

References

[1] Ranendu Adhikary, Kaustav Bose, Satwik Mukherjee, and Bodhayan Roy. Complexity of maximum cut on interval graphs. 5 2020.

[2] Geir Agnarsson. On chordal graphs and their chromatic polynomials, 2003.

[3] Kellogg S Booth and George S Lueker. Linear algorithms to recognize interval graphs and test for the consecutive ones property.

[4] Mario Carneiro. Maximum cardinality search and chordal graphs, 2018.

[5] Maria Chudnovsky, Neil Robertson, P D Seymour, and Robin Thomas. Progress on perfect graphs, 2003.

[6] Graph Classes. Gc 234 - chordal graphs, 2024. Accessed: 2024-12-22.

[7] Wikipedia contributors. Interval graph — wikipedia, the free encyclopedia, 2024. Accessed: 2024-12-22.

[8] Erdos, Goodman, and Posa. The representation of a graph by set intersections.

[9] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs (Volume 57) (Annals of Discrete Mathematics (Volume 57))*. 2004.

[10] Richard M. Karp. *Reducibility among Combinatorial Problems*. 1972.

[11] Antoon W.J. Kolen, Jan Karel Lenstra, Christos H. Papadimitriou, and Frits C.R. Spieksma. Interval scheduling: A survey, 8 2007.

[12] Lekkerkerker and Boland. Representation of a finite graph by a set of intervals on the real line. 1962.

[13] George S Lueker and Kellogg S Booth. A linear time algorithm for deciding interval graph isomorphism, 1979.

[14] Donald J Rose. Triangulated graphs and the elimination process, 1970.

[15] Maw shang Chang and Siam J Comput. Efficient algorithms for the domination problems on interval and circular-arc graphs *, 1998.