Shortest Beer Path Queries in Interval Graphs

Rathish Das
Department of Computer Science, University of Liverpool, UK

Meng He
Faculty of Computer Science, Dalhousie University, Halifax, Canada

Eitan Kondratovsky
Cheriton School of Computer Science, University of Waterloo, Canada

J. Ian Munro
Cheriton School of Computer Science, University of Waterloo, Canada

Anurag Murty Naredla
Cheriton School of Computer Science, University of Waterloo, Canada

Kaiyu Wu
Cheriton School of Computer Science, University of Waterloo, Canada

Abstract

Our interest is in paths between pairs of vertices that go through at least one of a subset of the vertices known as beer vertices. Such a path is called a beer path, and the beer distance between two vertices is the length of the shortest beer path.

We show that we can represent unweighted interval graphs using $2n \log n + O(n) + O(|B| \log n)$ bits where $|B|$ is the number of beer vertices. This data structure answers beer distance queries in $O(\log^{\varepsilon} n)$ time for any constant $\varepsilon > 0$ and shortest beer path queries in $O(\log^{\varepsilon} n + d)$ time, where d is the beer distance between the two nodes. We also show that proper interval graphs may be represented using $3n + o(n)$ bits to support beer distance queries in $O(f(n) \log n)$ time for any $f(n) \in \omega(1)$ and shortest beer path queries in $O(d)$ time. All of these results also have time-space trade-offs.

Lastly we show that the information theoretic lower bound for beer proper interval graphs is very close to the space of our structure, namely $\log(4 + 2\sqrt{3})n - o(n)$ (or about $2.9n$) bits.

2012 ACM Subject Classification Theory of computation → Data compression; Mathematics of computing → Paths and connectivity problems

Keywords and phrases Beer Path, Interval Graph

Digital Object Identifier 10.4230/LIPIcs.ISAAC.2022.59

Related Version *Full Version*: `https://arxiv.org/abs/2209.14401` [6]

1 Introduction

The concept of a beer path was recently introduced by Bacic et al. [2]. The premise is simple, suppose you wish to visit a friend, and wish to pick up some beer along the way because you don't want to show up empty handed, what is the fastest way to do so? More formally, for a graph, we specify a set of vertices, which will act as beer stores. A beer path is one which passes through at least one of these designated vertices. We will say a beer graph is one where we have designated a subset of the vertices to be beer stores. Though this premise may be somewhat silly, it can have many applications related to delivering objects. For example, suppose you are going on a road trip and to be efficient, want to drop something off at a post office on the way. Or perhaps on your trip, you realize that you currently don't have enough gas, so you must visit a gas station somewhere along the way. Another hypothetical situation would be if a package needs to be transported, but due to regulations, one of the stops must be equipped for an inspection.

33rd International Symposium on Algorithms and Computation (ISAAC 2022).
Editors: Sang Won Bae and Heejin Park; Article No. 59; pp. 59:1–59:17
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

It is easily seen that a shortest beer path may not be simple, but will always consist of two shortest paths: from the beer store to the source, and to the destination.

In this paper, we study the shortest beer path problem on unweighted interval graphs: intersection graphs of intervals on the real line. Interval graphs are a well-known class of graphs and have applications in operations research [3] and bioinformatics [18]. For a more indepth treatment of interval graphs and their applications, see the book of Golumbic [7].

Related Work. Bacic et al. [2] studied the problem on weighted outerplanar graphs. They showed that on an outerplanar graph of n vertices, a data structure of size $O(m)$ words for any $m \geq n$ can be constructed in $O(m)$ time to support shortest beer path and beer distance – the length of the shortest beer path in $O(\alpha(m,n))$ time, where α is the inverse Ackermann function.

On the more general problem of graph data structures, Acan et al. [1] showed that interval graphs may be represented succinctly using $n \log n + O(n)$ bits of space to answer basic navigational queries: `adjacent`, `degree`, `neighbourhood` and `shortest_path` in optimal time: $O(1)$ or $O(1)$ for each vertex in the output. Building on Acan et al.'s work, He et al. [9] added the `dist` query for interval graphs. Their data structure has the same space $n \log n + O(n)$[1] bits, the same run time of the old operations but also supports `dist` in $O(1)$ time.

1.1 Our Results and Paper Layout

We give data structures for beer interval graphs and beer proper interval graphs that have time-space trade offs. An interval graph is a graph where we may assign an interval on the real line to each vertex - $v \mapsto [l_v, r_v]$ such that two vertices u, v are adjacent exactly when the corresponding intervals intersect [8, 11]. A proper interval graph is an interval graph where the intervals must be chosen so that no two intervals nest. Furthermore, we prove a lower bound result on the space required for beer proper interval graphs.

The main obstacle in constructing the data structures for the beer path queries is that the set of paths between two vertices (which are normally the feasible solutions to the shortest path problem) are arbitrarily filtered by the beer nodes into a smaller set of feasible solutions to the beer shortest path problem - by whether a beer node exists on the path or not. In the case that a beer node exists on one of the shortest paths, then it is clearly optimal and we must be able to detect this. Thus we must be able conduct this filtering process as well, and we achieve this by using orthogonal range search on the previously established data structures which looks at all paths.

In section 3 we study the beer distance problem in proper interval graphs. We first outline the steps that we need to implement in our data structures. Then in subsection 3.2 we give a data structure occupying $3n + o(n) + O(|B| \log n)$ bits of space, supporting all regular operations in the same complexity as the previous works of Acan et al. and He et al., and the queries related to beer distance:

- `beer_dist` in $O(\log^{\varepsilon} n)$ time, for any constant $\varepsilon > 0$
- `beer_shortest_path` in $O(1)$ time per vertex on the path.

We may also utilize a trade-off provided by our auxiliary data structures, which decrease the query times by substituting $\log^{\varepsilon} n$ with $\log \log n$ at the cost of increasing one of the space cost terms from $O(|B| \log n)$ to $O(|B| \log n \log \log n)$ bits. We note that in this data structure,

[1] We will use log to denote $\log_2$.

the space is dependent on $|B|$ the number of beer vertices, and is therefore undesirable if $|B|$ is large. In the case that there are many beer vertices, the above data structure can use $O(n \log n)$ bits of space.

In subsection 3.3, we use the tree structure of the distance queries to eliminate the dependence on $|B|$ at the cost of slightly increasing the run time. For beer proper interval graphs, we have a data structure using $3n + o(n)$ bits of space, which supports all the regular queries in their original optimal complexities, and the beer queries:

- `beer_dist` in $O(f(n) \log n)$ time for any function $f(n) \in \omega(1)$. Different $f(n)$ will impact the lower order term $o(n)$ in the space complexity.
- `beer_shortest_path` in $O(1)$ per vertex on the path.

In section 4 we study the beer distance problem in interval graphs. For this, we give a data structure using $2n \log n + O(n) + O(|B| \log n)$ bits, where $|B|$ is the number of beer vertices in the graph, and supports all regular operations in the same time complexity as above, and

- `beer_dist` in $O(\log^{\varepsilon} n)$ time, for any constant $\varepsilon > 0$.
- `beer_shortest_path` in $O(\log^{\varepsilon} n + d)$ time, where d is the beer distance between the two vertices.

Again we may utilize the same trade off to replace the $\log^{\varepsilon} n$ term by $\log \log n$ at the cost of increase the space term $|B| \log n$ to $|B| \log n \log \log n$.

Finally in section 5 we count the number of non-isomorphic beer proper interval graphs and use this to give an information theoretic lower bound on the space required for any data structure for beer proper interval graphs that can support `adjacent` and `beer_dist`. It may seem natural that to store which vertices are beer nodes will require an additional n bits but we show that the lower bound is actually asymptotically $\log(4 + 2\sqrt{3})n \approx 2.9n$ bits. The main insight into seeing why an additional n bits is not required is that for a clique, it suffices to only store $\log n$ bits for the count of how many beer nodes. For general interval graphs, the space required for the beer nodes is at most n and is a lower order term to the lower bound of $n \log n$ bits. Thus there is nothing study in this case.

A full version with all proofs can be found on arXiv [6].

2 Preliminaries

In this paper, we will use the standard graph theoretic notation. We will use $G = (V, E)$ to denote a graph with vertex set V and edge set E. We will use $n = |V|$ and $m = |E|$ to denote the number of vertices and edges. All of our graphs will be unweighted.

As we will be discussing both trees and graphs in general, we will use vertices to denote the vertices of a graph which may or may not be a tree, and nodes to denote the vertices of a tree.

In the paper, we assume the word-RAM model with $\Theta(\log n)$-size words. We use $\log(\cdot)$ to denote $\log_2(\cdot)$.

In a beer graph, we take any underlying graph G together with a set $B \subseteq V$ of *beer vertices*. This allows us to define the following queries:

- `beer_shortest_path`(u, v): return a shortest path between the vertices u and v such that at least one of the beer vertices appears on the path.
- `beer_dist`(u, v): return the length of the shortest path between vertices u and v such that at least one of the beer vertices appears on the path.

These are the restricted queries to the ordinary `shortest_path` and `dist` queries, which do not have the constraint that it must pass through a beer vertex.

For example, if $B = V$, then the two queries reduces to ordinary shortest path or distance in the graph. On the other extreme, if $B = \{b\}$ is a singleton, then the query reduces to two ordinary shortest path or distance queries in the graph.

2.1 Interval Graphs

An interval graph G is a graph where we may assign an interval on the real line to each vertex - $v \mapsto [l_v, r_v]$ such that two vertices u, v are adjacent exactly when the corresponding intervals intersect [8]. In particular, we may sort the endpoints so that the values are integers between 1 and $2n$. We sort the vertices based on their left endpoints, so that when we refer to vertex v, we are referring to a number (the rank of the vertex in the sorted order), and thus, a statement such as $u < v$ makes sense.

Acan et al. [1] showed that interval graphs can be represented using $n \log n + O(n)$ bits to support `adjacent`, `degree`, `neighbourhood`, `shortest_path` queries in optimal time. `adjacent` answers whether two given vertices are adjacent, `degree` answers the degree of the given vertex, `neighbourhood` gives a list of the neighbours of the given vertex, and `shortest_path` gives a shortest path between the two given vertices.

He et al. [9] showed that we can also answer the `dist` query in optimal time. Therefore we have the following theorem on interval graphs:

▶ **Lemma 1.** *An interval graph G on n vertices can be represented succinctly using $n \log n + O(n)$ bits to support `adjacent`, `degree` and `dist` queries in $O(1)$ time, `neighbourhood` in $O(1)$ time per neighbour and `shortest_path` in $O(1)$ time per vertex on the path.*

An interval graph G is *proper* (or a *proper interval graph*) if we can choose the intervals corresponding to vertices such that no two intervals are nested.

Acan et al. [1] also showed that proper interval graphs can be represented using $2n + o(n)$ bits to support `adjacent`, `degree`, `neighbourhood`, `shortest_path` queries in optimal time. He et al. [9] showed that we may also support the `dist` query in optimal time. Thus we have the following theorem on proper interval graphs:

▶ **Lemma 2.** *A proper interval graph G on n vertices can be represented succinctly using $2n + o(n)$ bits to support `adjacent`, `degree` and `dist` queries in $O(1)$ time, `neighbourhood` in $O(1)$ time and `shortest_path` in $O(1)$ time per vertex on the path.*

2.2 Dyck Paths

For our lower bound, we will be discussing Dyck paths. A Dyck path of length $2n$ is a path from $(0, 0)$ to $(2n, 0)$ using $2n$ steps, n of which are $(1, 1)$ steps which are referred to as up-steps, and n of which are $(1, -1)$ steps and are referred to as down-steps. Such a path must also satisfy the condition that it never reaches below the x-axis. It is well known that the number of Dyck paths of length $2n$ is $C_n = \frac{1}{n+1}\binom{2n}{n}$ the n-th Catalan number.

A Dyck path that never touches the x-axis except at the start and end, is referred to as an irreducible Dyck path. By removing the up-step at the beginning and the down-step at the end, the remainder of the path is simply a Dyck path of length $2(n - 1)$. Thus the number of irreducible Dyck paths of length $2n$ is simply C_{n-1}.

For any Dyck path, we may associate an up-step with an open parenthesis (and a down-step with a close parenthesis). The sequence we obtain from a Dyck path is a balanced parenthesis sequence (and vice versa) as the Dyck path condition is exactly the condition that the excess in the balanced parenthesis sequence is never negative. This well known bijection allows us to associate an forest to each Dyck path, using the well known bijection for forests and balanced parentheses (via a depth-first traversal). In particular, if the Dyck path were irreducible, then the forest is just a single tree.

2.3 Succinct Data Structures

The information theoretic lower bound to represent a family of objects with N elements is $\lceil \log N \rceil$ bits. Any fewer bits and we do not have enough bit strings to assign a unique one to each object, and thus cannot distinguish between them. A succinct data structure aims to use $\log N + o(\log N)$ bits to represent these objects while supporting the relevant queries.

A bit vector is a length n array of bits, that supports the queries `rank`(i): given an index, return the number of 1s up to index i, `select`(j): given a number j, return the index of the jth one in the array, and `access`(i): return the bit at index i.

▶ **Lemma 3** (Munro et al. [12]). *A bit vector of length n can be succinctly represented using $n + o(n)$ bits to support* `rank`*,* `select` *and* `access` *in $O(1)$ time.*

We will also require the compressed form of bit-vectors, where if the number of 1s is small, we are able to get away with using less space.

▶ **Lemma 4** (Patrascu [14]). *A bit vector of length n with m 1s can be represented using $\log \binom{n}{m} + O(\frac{n}{\log^c n} + m) \leq m \log \frac{n}{m} + O(\frac{n}{\log^c n} + m)$ bits for any constant c. The data structure supports* `rank`*,* `select`*,* `access` *in $O(1)$ time.*

We will be working with trees and thus will be discussing and using various tree operations. These will mainly be conversions between the nodes' numbers in the different traversals: pre-order, post-order, level-order. These operations can be done in $O(1)$.

▶ **Lemma 5** (He et al. [9]). *An ordinal tree on n nodes can be represented succinctly using $2n + o(n)$ bits and can support a variety of operations in $O(1)$ time. For the full list see table 1 in their paper.*

2.4 Orthogonal Range Queries

In these data structures, we store n d-dimensional points. The queries we wish to answer are: given a d-dimensional axis aligned rectangle $[p_1, p_2] \times [p_3, p_4] \ldots [p_{2d-1}, p_{2d}]$ (we use the closed intervals here in the definition, but when our points are integers it is easy to see how to support open or semi-open intervals as well), *emptiness*: does it contain a point? *count*: how many points does it contain? *reporting*: return each point. We say that the rectangle is k-sided if there is at most k coordinates that are finite (in general the coordinates p_i may be $\pm\infty$). When we do not mention how many sides, it is assumed that it is the maximum possible: $2d$.

When $d = 2$, Chan et al. [5] showed that we solve the emptiness problem:

▶ **Lemma 6.** *We can solve the 2d range emptiness queries using $O(n \log n \log \log n)$ bits of space and $O(\log \log n)$ query time or $O(n \log n)$ bits of space and $O(\log^\epsilon n)$ query time for any constant $\epsilon > 0$.*

When $d = 3$, Nekrich [13] showed that we may solve the emptiness and reporting problems:

▶ **Lemma 7.** *For n 3D points and constant $\epsilon > 0$, we may support 5-sided orthogonal reporting queries using $O(n \log n)$ bits of space and $O(k \log^\epsilon n)$ time or $O(n \log n \log \log n)$ bits of space and $O(k \log \log n)$ time, where k is the size of the output.*

By setting $k = 0$, we may achieve the complexities to emptiness queries as well.

2.5 Predecessor Queries

Given a set of numbers S in a universe U, we wish to answer the following query: `pred`(i), given an element i of U, return the largest element j of S that is smaller than i.

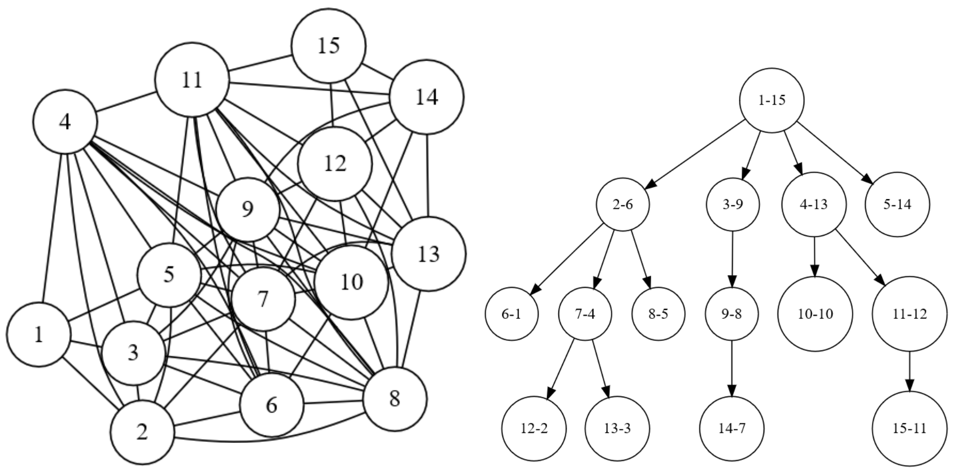
Though there have been a lot of work on this problem, we will only need one of the basic results by Willard [17] which gives a $O(N)$ word space solution to the problem with $O(\log \log U)$ query time.

▶ **Lemma 8.** *There is a data structure for the predecessor problem that uses $O(N)$ words of space and query time $O(\log \log U)$.*

3 Beer Paths in Proper Interval Graphs

In this section we investigate beer paths in proper interval graphs. We will be using the data structure of He et al. [9] as it supports the `dist` operation, which we will modify to account for the beer vertices. We begin with an example:

▶ **Example 9.** Consider the graph with the interval representation given by the bit string: 000001000101001110011011011111. This gives the vertex 1 a left endpoint at coordinate 1 and right index at coordinate 6 (the first 0 and first 1 in the sequence respectively). A graphical representation of the graph and the corresponding distance tree is given.



The first number in the node of the distance tree is the node's level-order number and the second its post-order traversal number.

Consider the shortest path between nodes 13 and 3. The shortest path algorithm given by He et al. [9] will give the path $13 \to 7 \to 3$. The problem would be easy if one of these nodes were a beer node, say node 7 then there would be far less to do. However consider the case that the only beer node were node 6, then a `beer_shortest_path` would be $13 \to 7 \to 6 \to 3$. On the other hand, if there were also a beer node at node 8, then we can take the path $13 \to 8 \to 3$.

3.1 Calculating Beer Distance

In the data structure of He et. al [9], we represent the proper interval graph G using a distance tree T. There is a bijection between the vertices of the graph G and the nodes of the tree - vertex v is mapped to the vth node in the tree in level-order. Thus by v we will simultaneously refer to the vertex and the node in the distance tree. As the conversion between level-order ranks, pre-order ranks and post-order ranks in a tree are all $O(1)$ (and typically, nodes in a tree are referred by their pre-order ranks), we will implicitly convert between them as the situation requires. All the queries are reduced to tree operations and can be done in $O(1)$ time.

Since we will be building upon the `dist` and `shortest_path` queries, we will explain it in detail. The distance tree's parent child relationship is the following: for a node v, the parent of v is the smallest (indexed) node that is adjacent to v.

Let $u < v$, be the nodes involved in the `dist`/`shortest_path` query, and suppose that $\texttt{depth}(u) = k_1 \leq k_2 = \texttt{depth}(v)$. We repeatedly take the parent of v to form the following chain: $v_{k_1}, v_{k_1+1}, \ldots, v_{k_2} = v$, where v_j is at depth j. If $v_{k_1} < u$ then a shortest path is $u, v_{k_1+1}, \ldots, v_{k_2} = v$. If $v_{k_1} = u$, then the shortest path is $u = v_{k_1}, v_{k_1+1}, \ldots, v_{k_2} = v$. Finally if $v_{k_1} > u$, then a shortest path is $u, v_{k_1}, v_{k_1+1}, \ldots, v_{k_2} = v$. To compute the length without getting every node, we simply subtract the depths of u and v, use level-ancestor to find v_{k_1} and find which case we fall into to adjust the distance.

We note that this is only one of many possible shortest paths. To accommodate the beer vertices, we will investigate what all the possible shortest paths might look like. To this end, we will consider the following question: let $u < w < v$, is w on a shortest path between u and v? Equivalently, is $\texttt{dist}(u, v) = \texttt{dist}(u, w) + \texttt{dist}(w, v)$? In this case we say that w *preserves the distance*. Let $\texttt{post}(v)$ denote the post-order rank of a node in the tree. It is clear that if $\texttt{post}(u) < \texttt{post}(v)$, then u is to the left of v_{k_1}, so that $u < v_{k_1}$, and similarly for the reversed inequality. Finally, we note that for nodes on the same level of the tree, their post-order numbers are sorted. That is if $u < v$ are on the same level, then $\texttt{post}(u) < \texttt{post}(v)$ (and vice versa).

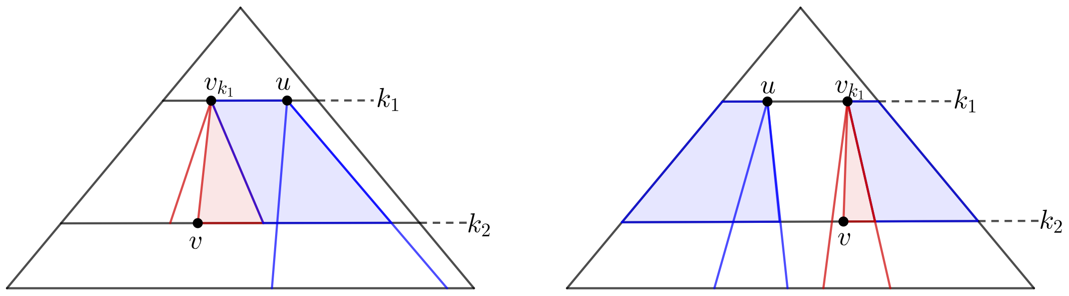


Figure 1 The union of the two shaded regions capture the nodes which preserve the distance in Lemma 10. The left is the case when $\texttt{post}(u) > \texttt{post}(v)$, and the right is the case when $\texttt{post}(u) < \texttt{post}(v)$. The blue region represents complete subtrees that are included, while the red represents the nodes to the right of the path to the root from v, used in subsection 3.3. Note the nodes on level k_1 to the left of u and on level k_2 to the right of v are not included.

▶ **Lemma 10.** *Let $u < w < v$ be 3 nodes in a proper interval graph. Suppose that $\texttt{post}(u) < \texttt{post}(v)$, then w is on a shortest path exactly when either $\texttt{post}(w) < \texttt{post}(u)$ or $\texttt{post}(w) > \texttt{post}(v)$ (that is w preserves the distance). If $\texttt{post}(u) > \texttt{post}(v)$, then w preserves the distance exactly when $\texttt{post}(v) < \texttt{post}(w) < \texttt{post}(u)$. Furthermore, if w does not preserve the distance, then the path passing through w increases the distance by 1. That is $\texttt{dist}(u, v) + 1 = \texttt{dist}(u, w) + \texttt{dist}(w, v)$.*

See figure 1 for a pictorial representation of the criteria; Now we can describe the process of determining the beer distance. The idea is to cover the nodes of the tree with 3 sets, and determine the best possible beer distance using beer vertices in each of the 3 sets. Finally we take the minimum of the 3. We will call the best vertex in each set a *candidate*.

First we note that if either $u, v \in B$, then we do not need to do anything and simply return $\texttt{dist}(u, v)$ (or $\texttt{shortest\_path}(u, v)$).

Candidate 1. The set of nodes is $\{w \in B; w > v\}$. We claim that the best beer node in this set is the smallest one. To show this, we will use the following lemma.

▶ **Lemma 11.** *Let $u < v < w$ be 3 nodes in a proper interval graph, then $\texttt{dist}(u, v) \leq \texttt{dist}(u, w)$. By symmetry, $\texttt{dist}(v, w) \leq \texttt{dist}(u, w)$.*

The node in $\{w \in B; w > v\}$ that minimizes the value of $\texttt{dist}(u, w) + \texttt{dist}(v, w)$ is of course the w of minimal index.

Candidate 2. The set of nodes $\{w \in B; w < u\}$. We take the largest node in the set as the candidate using Lemma 11.

Candidate 3. The set of nodes $\{w \in B; u < w < v\}$. By Lemma 10, we need to determine whether there exists a node such that either $\texttt{post}(u) < \texttt{post}(w) < \texttt{post}(v)$ or $(\texttt{post}(w) < \texttt{post}(u)$ or $\texttt{post}(w) > \texttt{post}(v))$, depending on $\texttt{post}(u) < \texttt{post}(v)$. If such a node exists, then it is the candidate, with distance $\texttt{dist}(u, v)$. If no such node exists, we may take any node as the candidate, with distance $\texttt{dist}(u, v) + 1$.

3.2 First Data Structure For Beer Distance

Here we discuss how to use the previous results to create a data structure for the queries. In this subsection, we discuss a relatively simple data structure which has decent run times. However, the space is dependent on $|B|$ the number of beer nodes and in the case that $|B| = \Theta(n)$ is large, the space bound is also unacceptably large. In the next subsection, we will show how to remove this dependence on $|B|$ at the cost of slightly worse run times.

To store the beer vertices, we store a bit vector B of length n so that $B[i] = 1$ if the ith vertex is a beer vertex. This uses $n + o(n)$ bits of space. As above we will assume that neither u nor v are beer vertices (and we can check by looking at $B[u], B[v]$).

Candidate 1. We use $\texttt{rank}(v)$ to find how many beer nodes are up to v. The smallest beer node that is larger than v can be found using $\texttt{select}(\texttt{rank}(v) + 1)$.

Candidate 2. Similarly to candidate 1, we find it by $\texttt{select}(\texttt{rank}(u))$.

Candidate 3. For every beer node w, we store it in a 2D range emptiness data structure using the coordinates $(w, \texttt{post}(w))$ (by our notation w is just the level-order number of the node w). In the case that $\texttt{post}(u) < \texttt{post}(v)$, we need the nodes such that $\texttt{post}(w) < \texttt{post}(u)$ or $\texttt{post}(w) > \texttt{post}(v)$ and $u < w < v$. This is translated to the rectangles $(u, v) \times (-\infty, \texttt{post}(u))$ and $(u, v) \times (\texttt{post}(v), \infty)$.

For the second case when $\texttt{post}(u) > \texttt{post}(v)$, we need the nodes that $\texttt{post}(v) < \texttt{post}(w) < \texttt{post}(u)$. This is the rectangle $(u, v) \times (\texttt{post}(v), \texttt{post}(u))$.

This suffices to determine the distance. To list out the path, we first determine which candidate to use. Candidates 1 and 2 can simply list out the path using two shortest_path queries. For candidate 3, we list out the path between u, v one step at a time, and, at each step, we consider the set V_k, which is an interval in level order. We note that the nodes preserving the distance is a prefix of this interval. Thus we find the first beer vertex in V_k, and check if it preserves the distance, if so add it to the path and list out the path from there. Otherwise, we continue to the next level. Furthermore, as listing out the path for Candidate 3 is at most $O(\mathtt{dist}(u,v))$ time, we may do this whenever $\mathtt{dist}(u,v) = O(\log^\epsilon n)$ rather than spending the time on the orthogonal range search in the distance query. Thus we have the following theorem:

▶ **Theorem 12.** *A beer proper interval graph G can be represented using $3n + o(n) + O(|B|\log n)$ bits to support the interval graph queries plus $\mathtt{beer\_shortest\_path}$ in $O(1)$ time per vertex on the path and $\mathtt{beer\_dist}(u,v)$ in $O(\min(\log^\epsilon n, \mathtt{dist}(u,v)))$ time. If we increase the extra space to $O(|B|\log n \log\log n)$ bits, we may support $\mathtt{beer\_dist}(u,v)$ in $O(\min(\log\log n, \mathtt{dist}(u,v)))$ time instead.*

Proof. The space required is the distance tree of [9], a single bit vector for the beer nodes, and a single 2D range emptiness data structure. ◀

We note that if there are many beer vertices, so that $|B| \in \Theta(n)$, then the space usage would be $\Theta(n\log n)$. However if $|B|$ were small (ex. $|B| = O(n/\log^2 n)$), then this data structure will suffice.

3.3 Improved Data Structure for Beer Distance

Here we show how to improve the space usage of our specialized ranged emptiness query, so that it no longer has any dependence on $|B|$. Since we have the tree structure, the range emptiness can be reduced to checking whether a certain set of tree nodes have any beer nodes in them. In particular, as seen from the previous subsection, we need to support the following rectangles using $o(n)$ bits: 1) $(u,v) \times (-\infty, \mathtt{post}(u))$, 2) $(u,v) \times (\mathtt{post}(v), \infty)$, and 3) $(u,v) \times (\mathtt{post}(v), \mathtt{post}(u))$. We will call these type 1,2 and 3 rectangles.

To make our notation cleaner, we will use the depth of a node in the first coordinate of a rectangle. This means to include all the nodes on that level. To accomplish this, we simply find the first node on that level (in $O(1)$ time) and substitute its level-order number as the value to be used in the rectangle; similarly use the last node of a level for the right end point of the rectangle.

Fix $\Delta = \omega(1)$, and choose an index $1 \le i \le \Delta$ such that the number of nodes on levels $k = i \mod \Delta$ is minimized. We will call these levels *selected levels*, and the nodes on them *selected nodes*. Thus the number of selected nodes is $O(n/\Delta)$ by the pigeonhole principle. For each of these nodes, consider the subtree rooted at them that extends down to the next select level. We will build the contracted tree T' with these subtree as nodes, and the appropriate edges. A node in T' is a beer node if at least one of the original nodes in the corresponding subtree *except the root* is a beer node.

We use a bit vector to store which nodes in level order are selected. As the number of nodes is $O(n/\Delta) = o(n)$, this compressed bit-vector uses $o(n)$ bits of space. We store the contracted tree T' succinctly, using $2n/\Delta + o(n) = o(n)$ bits. The bit vector to mark which nodes of T' are beer nodes is also $n/\Delta = o(n)$ bits. Thus the total space for our contracted tree is $o(n)$ bits.

To support these rectangles, we first reduce the general case to one where both u, v are on a selected level.

▶ **Lemma 13.** *We may assume that the inputs u, v are on selected levels at the cost of $O(\Delta)$ extra time.*

We will now assume that both u, v are on selected levels. To deal with these queries, we will build the 2D range emptiness query on our contracted tree T' and the nodes on the selected levels in the same way. We wish to convert as much of the query to the 2D range emptiness on the contracted tree as possible. We will denote the corresponding node in the contracted tree to u by u'.

Type 1 rectangles. We convert $(u, v) \times (-\infty, \texttt{post}(u))$ to $[\texttt{depth}(u'), \texttt{depth}(v')) \times (-\infty, \texttt{post}(u'))$ in the contracted tree and the same rectangle $(u, v) \times (-\infty, \texttt{post}(u))$ in the selected nodes. We note that in T', we exclude all nodes on $\texttt{depth}(v')$ since in T this includes only the nodes on that level, but in T' this would include the subtrees as well, which extend down. We also change the left endpoint so that we include the subtrees to the left of u, but as we exclude the roots from those subtrees (in our decision to mark them as beer nodes or not), we do not include more nodes in our search than required.

Type 2 rectangles. In the type 2 rectangles, we note that unfortunately, the rectangle does not contain all the nodes in the subtree $v_{\texttt{depth}(u)}$, only those to the right of the path to v. It does however include the entire subtrees of all the nodes to the right of $v_{\texttt{depth}(u)}$. To handle these complete subtrees (and exclude the subtree rooted at $v_{\texttt{depth}(u)}$) we use the rectangle $[\texttt{depth}(u'), \texttt{depth}(v')) \times (\texttt{post}(v_{\texttt{depth}(u')}), \infty)$. Of course we may find $v_{\texttt{depth}(u)}$ using level-ancestor. We again handle the selected nodes using the same rectangle on them $(u, v) \times (\texttt{post}(v), \infty)$.

Finally, we need to handle the the nodes in the subtree rooted at $v_{\texttt{depth}(u)}$, to the right of the path to v and above the level of v. We start with the entire subtree of $v_{\texttt{depth}(u)}$, whose nodes are an interval in post-order. Then nodes w with $\texttt{post}(w) > \texttt{post}(v)$ are exactly the ones we want, except that all the subtrees to the right of v extend down to the bottom of the tree, rather than being cut off at the depth of v. Thus we need to handle them as well. The way to do this in encoded in the lemma below, where $k_1 = \texttt{depth}(u)$.

▶ **Lemma 14.** *Let v be a node in a tree T at depth k_2 and v_{k_1} be the ancestor of v at depth k_1, where both k_1, k_2 are selected levels with a fixed $\Delta > 0$. Let T' be the contracted tree as defined above. Then we are able to answer the query: does the rectangle $(v_{k_1}, v) \times (\texttt{post}(v), \infty)$ in either:*

$O(n/\Delta \log n) + o(n)$ additional space and $O(\log n)$ time.

$O(n/\Delta \log n) + n + o(n)$ additional space and $O(\log \log n)$ time.

Here we do not count the space taken by the tree T.

Type 3 Rectangles. Type 3 rectangles are similar to type 2 rectangles. As above, we use the same rectangle $(u, v) \times (\texttt{post}(v), \texttt{post}(u))$ for the selected nodes. We again use the rectangle $[\texttt{depth}(u'), \texttt{depth}(v')) \times (\texttt{post}(v_{\texttt{depth}(u')}), \texttt{post}(u')]$ to capture the complete subtrees that we wish to use. The incomplete subtree is exactly the same as in type 2, so we are able to apply lemma 14.

Thus putting everything together we have the following theorem:

▶ **Theorem 15.** *Let G be a beer proper interval graph. Fix Δ, then G can be represented using $3n + o(n) + O((n/\Delta) \cdot \log n)$ bits and can support* ***adjacent***, ***degree***, ***neighbourhood***, ***dist*** *in $O(1)$ time,* ***shortest_path***, ***beer_shortest_path*** *in $O(1)$ time per vertex on the path and* ***beer_dist*** *in $O(\Delta + \log n)$ time.*

In particular, if we take $\Delta = \log n$, then the space is $O(n)$ with time $O(\log n)$ and if we take $\Delta = f(n) \log n$ for some $f(n) = \omega(1)$, then the space is $3n + o(n)$ and the time is $O(f(n) \log n)$.

If we wish to further our trade off and improve the time, we must explicitly store P as in the proof lemma 14, and use the space inefficient (but time efficient) range query data structures. Thus we obtain:

▶ **Theorem 16.** *Let G be a beer proper interval graph. Fix Δ, then G can be represented using $4n + o(n) + O((n/\Delta) \cdot \log n \log\log n)$ bits and can support* `adjacent`*,* `degree`*,* `neighbourhood`*,* `dist` *in $O(1)$ time,* `shortest_path`*,* `beer_shortest_path` *in $O(1)$ time per vertex on the path and* `beer_dist` *in $O(\Delta + \log\log n)$ time.*

In particular, if we take $\Delta = \log\log n$, then the space is $O(n \log n)$ with time $O(\log\log n)$.

4 Beer Paths in Interval Graphs

In this section, we study how to compute and construct data structures for beer paths and beer distances. We will use the the data structure of Acan et al. [1] as a black box, and work with the distance tree T of He et al. [9]. The major difference between interval graphs and proper interval graphs is how adjacency can be checked. In a proper interval graph, for a vertex v, and its parent p in the distance tree, v is adjacent to every vertex between v and p. However, in interval graphs, this is not the case, and depending on the graph structure, any of those vertices can be adjacent or not adjacent to v.

4.1 Calculating Beer Distance

As in proper interval graphs, we begin by investigating the conditions in which nodes *preserve* the distance. For a node w, we say that w is $+k$ distance if $\texttt{dist}(u, v) + k = \texttt{dist}(u, w) + \texttt{dist}(v, w)$ (and thus preserving the distance is equivalent to being $+0$ the distance). So, using w as a beer node will add k to the optimal (non-beer path) distance. To do this, we will add one more condition to that of Lemma 10.

Let u be a node in T. As shown in the proof of lemma 13, the node on the next level that splits it in the same way as u is the largest neighbour of u. We denote this by $\texttt{last}(u)$. For example, in example 9, the largest neighbour of the node 8, is the node 13, as 8 is adjacent to node 13, but not node 14. Thus $\texttt{last}(8) = 13$.

▶ **Lemma 17.** *Let $u < v$ be vertices in a beer interval graph G with depths $\texttt{depth}(u) = k_1 \leq k_2 = \texttt{depth}(v)$. Consider the nodes $u < w < v$. Then we have the following two criteria:*

- *If $\texttt{post}(\texttt{last}(u)) < \texttt{post}(v)$, then either $\texttt{post}(w) > \texttt{post}(v)$ or $\texttt{post}(w) < \texttt{post}(\texttt{last}(u))$. If $\texttt{post}(\texttt{last}(u)) > \texttt{post}(v)$, then $\texttt{post}(v) < \texttt{post}(w) < \texttt{post}(\texttt{last}(u))$.*
- *If $\texttt{post}(w) < \texttt{post}(v)$, then $\texttt{post}(\texttt{last}(w)) > \texttt{post}(v)$ or $\texttt{post}(\texttt{last}(w)) < \texttt{post}(w)$. If $\texttt{post}(w) > \texttt{post}(v)$, then $\texttt{post}(v) < \texttt{post}(\texttt{last}(w)) < \texttt{post}(w)$.*

If w satisfies both criteria, then w preserves the distance. If w satisfies one criteria, then w is +1 distance and if w satisfies neither criteria, then w is +2 distance.

Again, we will find several sets that cover the beer nodes, and argue about the optimal beer node in these sets. We then take the minimum distance of these candidates. As before, we will assume that neither u nor v are beer nodes.

Candidate 1. This is the same as the proper interval graphs: $\{w \in B; w > v\} = \{w \in B; l_w > l_v\}$. We again claim that the best beer node is the smallest one. It turns out the exact same proof of Lemma 11 will work here.

Candidate 2. We wish to use the symmetric set of Candidate 1. Unfortunately, in interval graphs, this is not as simple. The set is $\{w \in B; r_w < r_u\}$. We note that this condition and the proper interval graph non-nesting condition gives $l_w < l_u$ so that $w < u$, and thus this is the right analogous set to consider. We claim that the best node is the node with the largest r_w in this set. This can be seen by reflecting the intervals of the vertices - equivalent to sorting them by the right endpoints instead. In this reflected graph, apply Lemma 11, and the result follows.

Candidate 3. The nodes $\{w \in B; u < w < v\}$. By Lemma 17, we can obtain the distance of the best beer node using the criteria in the lemma.

Candidate 4. The left over nodes. The nodes that do not belong to the previous candidate sets are w with: $l_w < l_v$ and $r_w > r_u$ and $l_w < l_u$. Thus these are the nodes with $l_w < l_u < r_w$, so they are adjacent to u. Formally, this is the set: $\{w \in B; w < u, r_w > r_u\}$. As these nodes are adjacent to u, all we need to check is their distance to v.

First consider the path from u to v. By the distance algorithm, we have the path to the root from v: $v_1, \ldots, v_{k_2} = v$, and furthermore suppose that k is the index such that $v_k \leq u < v_{k+1}$. The end of the path could look like either $u, v_{k+1}, \ldots$ or $u, v_k, v_{k+1}, \ldots$, depending on whether v_{k+1} is adjacent to u or not.

First assume that v_{k+1} is not adjacent to u, then for any possible candidate w, if w were adjacent to v_{k+1} (that is $\texttt{last}(w) \geq v_{k+1}$), then w preserves the distance (as $\texttt{dist}(w, v) = \texttt{dist}(u, v) - 1$). Otherwise, as $r_w > r_u > l_{v_k}$, w is adjacent to v_k and hence $\texttt{dist}(w, v) = \texttt{dist}(u, v)$ and w is $+1$ distance.

Next assume that v_{k+1} is adjacent to u. Then again for any candidate w, $l_w < l_u < l_{v_k+1} < r_u < r_w$, so w is adjacent to v_{k+1} and w is $+1$ distance. We note that w cannot be adjacent to v_{k+2} as in this case we would contradict that fact that v_{k+1} is the smallest node adjacent to v_{k+2}, by the parent relationship in T.

Finally we note that the only property of w that we used is that w is adjacent to u, and that if $x > u$ is adjacent to u, then w is also adjacent to x and hence we may relax the set to $\{w \in B; w < u, \texttt{last}(w) \geq \texttt{last}(u)\}$.

4.2 Data Structure for Beer Distance

We discuss how to use the previous results to create a data structure for the queries. We use the data structure of He et al. [9] which supports the interval graph queries in optimal time. This uses $n \log n + O(n)$ bits of space. We note that this data structure itself builds upon the data structure of Acan et al. [1]. We store a bit vector B as before, which stores which nodes are beer nodes in level-order. This take $n + o(n)$ bits.

Candidate 1. We handle this in exactly the same way as in proper interval graphs.

Candidate 2. We need to be able to find nodes in the mirrored graph.

▶ **Lemma 18.** *We can find the desired node in the mirrored graph using $n \log n + O(n)$ bits in $O(1)$ time.*

Candidate 3. We are able to handle this using 3D 5-sided orthogonal range emptiness data structures.

▶ **Lemma 19.** *We can check if a beer node satisfies the criteria of lemma 17 using a constant number of 3D 5-sided orthogonal range emptiness data structures. Thus the space/time requirements are either $O(n \log n)$ space and $\log^{\varepsilon} n$ time or $O(n \log n \log\log n)$ space and $\log\log n$ time.*

Candidate 4. We will use the following lemma:

▶ **Lemma 20.** *We can convert the criteria of candidate 4 into a constant number of 5-sided rectangles.*

Finally, to handle shortest paths, we use the reporting query rather than the emptiness query. When the reporting query returns the first point, we stop. After we find the best beer node, we list out the path using two `shortest_path` queries.

▶ **Theorem 21.** *Let G be a beer interval graph, with beer nodes B. The there exists a data structures using $2n \log n + O(n) + O(|B| \log n)$ bits that supports `degree`, `adjacent`, `dist` in $O(1)$ time, `neighbourhood`, `shortest_path` in $O(1)$ time per vertex in the path/neighbourhood, `beer_dist` in $O(\log^{\epsilon} n)$ time and `beer_shortest_path` in $O(\log^{\epsilon} n + d)$ time where d is the distance between the two vertices.*

Alternatively, we may increase the space from $O(|B| \log n)$ to $O(|B| \log n \log\log n)$ and replace the $\log^{\epsilon} n$ in `beer_shortest_path`, `beer_dist` with $\log\log n$.

5 Lower Bounds for Beer Interval Graphs

In this section, we will derive lower bounds for beer interval graphs and beer proper interval graphs. The lower bounds we will derive will be information theoretic, so that for a set of objects X, we will need at least $\log |X|$ bits in the worst case to represent any specific object.

First, we note that it is not interesting for beer interval graphs, since adding beer vertices can increase the lower bound by at most n bits. Since the lower bound for interval graphs is already $n \log n - o(n \log n)$ bits, the increase in space to account for the beer vertices is a lower order term and does not impact our data structures. For proper interval graphs however, the lower bound is $2n$ and thus it is natural to ask whether adding the beer vertices requires the full n bits to store them. That is, is the lower bound for beer proper interval graphs $3n$? In the main result of this section, we will show that it is not necessarily the case, and that if X were the set of beer proper interval graphs, then $\log |X| = n \log(4 + 2\sqrt{3}) - o(n) \approx 2.9n$.

In our case, we are interested in beer graphs, that is a graph G together with a set $B \subseteq V$ of beer vertices. We will refer to B as a beer vertex pattern. We will say that two beer graphs (G_1, B_1) and (G_2, B_2) are isomorphic (and thus are the same object) if there exists a bijection $f : V(G_1) \mapsto V(G_2)$ such that $(u, v) \in E(G_1) \Leftrightarrow (f(u), f(v)) \in E(G_2)$ and $u \in B_1 \Leftrightarrow f(u) \in B_2$. The first condition is the standard condition for two graphs to be isomorphic and the second condition says that this isomorphism also preserves beer vertices. Thus for two beer graphs to be isomorphic, the underlying graphs must also be isomorphic as well.

▶ **Example 22.** Suppose our graph class are cliques, then how many beer cliques are there? On n vertices, there is exactly one underlying graph $G = K_n$ on n vertices that is a clique. Thus it remains to see how many different beer vertex patterns we can have. By definition, if (K_n, B_1) were isomorphic to (K_n, B_2), then there exists an automorphism σ of K_n mapping vertices $u \in B_1$ to $\sigma(u) \in B_2$ bijectively, and thus $|B_1| = |B_2|$. Conversely, if $|B_1| = |B_2|$ then there exists a bijection σ that maps the elements of B_1 to B_2 and fixes every other vertex. As the underlying graph is the complete graph K_n, this σ is also an automorphism of the underlying graph as well. Thus (K_n, B_1) is isomorphic to (K_n, B_2) exactly when $|B_1| = |B_2|$. The number of different ways to add beer nodes to a clique on n vertices is thus $n + 1$. ◀

As $(G_1, B_1) \cong (G_2, B_2)$ happens only when $G_1 \cong G_2$, it remains to develop the theory to compute the number of beer vertex patterns that are different when given a specific underlying graph G. Let $Aut(G)$ denote the automorphism group of a graph G. We will view $B \subseteq V$ as a vector $B \subseteq 2^n$ on the hypercube (where the i-th bit denotes whether the i-th vertex belong to the set or not), and $Aut(G)$ as a group that acts on 2^n. In this lens, two beer vertex patterns B_1, B_2 are the same if there exists a group element σ mapping B_1 to B_2, and thus B_1 and B_2 belong to the same orbit of this group action. The number of different beer vertex patterns is thus the number of orbits $|2^n/Aut(G)|$. To count the number of orbits, we will use the Polya enumeration theorem [15], which in its most basic form, states that if we denote $c(\sigma)$ as the number of cycles in σ when viewed as a permutation of $V(G)$, $|2^n/Aut(G)| = \frac{1}{|Aut(G)|} \sum_{\sigma \in Aut(G)} 2^{c(\sigma)}$.

5.1 Automorphism Groups of Proper Interval Graphs

Klavic and Zeman [10] showed that Aut(connected PROPER INT) = Aut(CATERPILLAR). A caterpillar graph/tree is a path together with a set of leaves that are adjacent some vertex on the path. In particular, the automorphism group of any particular connected proper interval graph is generated by 2 types of automorphisms. First are automorphisms that swap twin vertices - which corresponds to those that swap the leaves adjacent to the same vertex on the path of a caterpillar graph. In a proper interval graph, twin vertices are those that have the same set of maximal cliques. The second is an automorphism that reverses the proper interval graph, which corresponds to reversing the path of a caterpillar graph. This reversal corresponds to a reversal of the maximal cliques. Of course, for any particular graph, there may not be any twin vertices, and thus there are no automorphisms of the first type. As for the second type, it can only exist when the number of vertices in the maximal cliques are symmetrical - as the vertices in the first maximal clique are mapped to those in the last maximal clique etc.

We will assume that the maximal cliques are not symmetrical and thus no automorphisms of the second type exists. To see this, we may always desymmetrize the sequence of maximal cliques by adding one vertex to only the first maximal clique if necessary.

Now suppose that G is a connected proper interval graph. As being twin vertices are an equivalence relation, let $S_1, \ldots, S_h$ be the equivalence classes of twin vertices, that is $u, v \in S_i$ implies that u, v are twins. Let $k_i = |S_i|$ and we will say that vertices which have no twins are in a class of size 1, so that $\sum_i k_i = n$.

▶ **Lemma 23.** *Let G be a proper interval graph with twin vertex classes of sizes $|S_1| = k_1, \ldots, |S_h| = k_h$. Then $|2^n/Aut(G)| \leq (k_1 + 1)(k_2 + 1) \cdots (k_h + 1)$. This is an equality in the case that the graph is connected and the maximal cliques are non-symmetrical.*

For a proper interval graph, we will also refer to the above quantity as its weight, as the number of beer proper interval graphs would be the weighted sum of proper interval graphs.

5.2 Lower Bound

In the section, we will prove a tight lower bound on the number of beer proper interval graph. The main idea is to decompose a Dyck path (which is in more or less a one-to-one correspondence to proper interval graphs) in such a way that it preserves the weights.

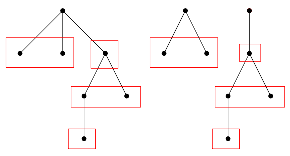


Figure 2 The twin vertex classes in the distance tree, and a way to decompose the tree.

To see the one-to-one correspondence, we see that by the interval graph recognition algorithm of Booth and Leuker [4], the PQ-tree of the maximal cliques is a single Q node, so that there are at most two ordering of maximal cliques representing each graph (one order and the reverse of that order). As each Dyck path gives a different sequence of maximal cliques, each graph can have at most two Dyck paths representing it.

For a particular proper interval graph, with twin vertex classes of sizes $k_1, \ldots, k_l$, the weight assigned to it is $\Pi_i(k_i + 1)$. Now consider the following blocking scheme for the distance tree associated with the proper interval graph: start at the root and continue in level-order, add the vertices to the block until either: the vertex has a different parent, or the vertex is not a leaf. In this manner, we consider the root as a sibling of its left child.

▶ **Lemma 24.** *In the above blocking scheme, two vertices u, v are in the same block if and only if they are twins.*

We may look at this in the same way by replacing the root with a dummy root and dropping the original root as the first child of the dummy root. This blocking scheme is illustrated in Figure 2.

Note that we do not consider the dummy root as part of our blocks and we can also view this as deleting the dummy root and consider the roots of this new forest as siblings. The second is our proposed way to decompose the tree into two trees while preserving all the blocks. If we consider the balanced parenthesis view of the tree (without the dummy root), we see that the sequences are the same $()()|((())())$, but we cut it into two at the $|$. Precisely, $|$ is at the first spot in the sequence such that the excess is 0 and the next two parentheses are ((. In the language of Dyck paths, this is the first time the path touches the x-axis and the next two steps are both up-steps.

Let $C(n)$ be the n-th Catalan number and the number of Dyck paths of length $2n$. The above decomposes the path into two subpaths. Let $L(n)$ be the number of paths of length n for the subpath to the left of $|$. and $R(n)$ be the number of paths of length n of the subpath to the right. We will also abuse notation and use $L(n), R(n)$ as the set of Dyck paths of the respective forms. Thus we have the recurrence $C(n) = \sum_{k=0}^{n} L(k)R(n-k)$.

Now we consider the weighted versions. Let $\bar{C}(n)$ be the sum of all Dyck paths with our weighting system. Similarly for $\bar{L}(n)$ and $\bar{R}(n)$. Because we preserve all the blocks with our split, we have the same recurrence $\bar{C}(n) = \sum_{k=0}^{n} \bar{L}(k)\bar{R}(n-k)$, which holds for $n \geq 1$. For $n = 0$, we see that $\bar{L}(0) = 0, \bar{R}(0) = 1$ and $\bar{C}(0) = 1$. Now it remains to compute $\bar{L}(n)$ and $\bar{R}(n)$.

▶ **Lemma 25.** $\bar{L}(n) = \sum_{l=0}^{n} \bar{C}(n-l-1)(l+2)$ *and* $\bar{R}(n) = \bar{C}(n) - \sum_{l=1}^{n}(l+1)\bar{R}(n-l)$.

Now consider the following generating functions. Let $f(x) = \sum_{n\geq 0} \bar{C}(n)x^n$, $g(x) = \sum_{n\geq 0} \bar{L}(n)x^n$ and $h(x) = \sum_{n\geq 0} \bar{R}(n)x^n$. The above recurrences says that these generating functions are linked and that we have a very nice closed form for f.

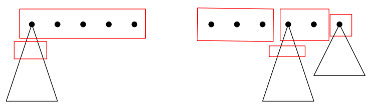


Figure 3 Decomposition of Dyck paths (as viewed as trees) of the forms L and R.

▶ **Lemma 26.** *Let $b(x) = (1-x)^{-2}$. Then we have $f = gh + 1$, $g = f \cdot (b-1)$ and $h = f/b$. Finally $f = \left(1 - \sqrt{1 - 8x + 4x^2}\right) / \left(4x - 2x^2\right)$.*

We note that the sequence A108524 of OEIS [16] has the same generating function and thus $\bar{C}(n)$ is exactly A108524. Thus we can finally prove our desired lower bound for the number of beer proper interval graphs.

▶ **Theorem 27.** *The number of beer proper interval graphs on n vertices is asymptotically $(4 + 2\sqrt{3})^n \cdot poly(n, 1/n)$. Therefore to represent a beer proper interval graph G which is able to support `adjacent` and `beer_dist` will require at least $\log(4 + 2\sqrt{3})n - o(n) \approx 2.9n$ bits in the worst case.*

References

1 Hüseyin Acan, Sankardeep Chakraborty, Seungbum Jo, and Srinivasa Rao Satti. Succinct encodings for families of interval graphs. *Algorithmica*, 83(3):776–794, 2021. doi:10.1007/s00453-020-00710-w.

2 Joyce Bacic, Saeed Mehrabi, and Michiel Smid. Shortest beer path queries in outerplanar graphs. In Hee-Kap Ahn and Kunihiko Sadakane, editors, *32nd International Symposium on Algorithms and Computation, ISAAC 2021, December 6-8, 2021, Fukuoka, Japan*, volume 212 of *LIPIcs*, pages 62:1–62:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ISAAC.2021.62.

3 Amotz Bar-Noy, Reuven Bar-Yehuda, Ari Freund, Joseph Naor, and Baruch Schieber. A unified approach to approximating resource allocation and scheduling. *J. ACM*, 48(5):1069–1090, 2001. doi:10.1145/502102.502107.

4 Kellogg S. Booth and George S. Lueker. Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms. *J. Comput. Syst. Sci.*, 13(3):335–379, 1976. doi:10.1016/S0022-0000(76)80045-1.

5 Timothy M. Chan, Kasper Green Larsen, and Mihai Patrascu. Orthogonal range searching on the ram, revisited. In Ferran Hurtado and Marc J. van Kreveld, editors, *Proceedings of the 27th ACM Symposium on Computational Geometry, Paris, France, June 13-15, 2011*, pages 1–10. ACM, 2011. doi:10.1145/1998196.1998198.

6 Rathish Das, Meng He, Eitan Kondratovsky, J. Ian Munro, Anurag Murty Naredla, and Kaiyu Wu. Shortest beer path queries in interval graphs, 2022. arXiv:2209.14401.

7 Martin Charles Golumbic. *Algorithmic graph theory and perfect graphs.* Elsevier, 2004.

8 G Hajós. Über eine art von graphen. int. *Math. Nachr*, 11:1607–1620, 1957.

9 Meng He, J. Ian Munro, Yakov Nekrich, Sebastian Wild, and Kaiyu Wu. Distance oracles for interval graphs via breadth-first rank/select in succinct trees. In Yixin Cao, Siu-Wing Cheng, and Minming Li, editors, *31st International Symposium on Algorithms and Computation, ISAAC 2020, December 14-18, 2020, Hong Kong, China (Virtual Conference)*, volume 181 of *LIPIcs*, pages 25:1–25:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.ISAAC.2020.25.

10 Pavel Klavík and Peter Zeman. Automorphism Groups of Geometrically Represented Graphs. In Ernst W. Mayr and Nicolas Ollinger, editors, *32nd International Symposium on Theoretical Aspects of Computer Science (STACS 2015)*, volume 30 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 540–553, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.STACS.2015.540.

11 C Lekkeikerker and Johan Boland. Representation of a finite graph by a set of intervals on the real line. *Fundamenta Mathematicae*, 51:45–64, 1962.

12 J. Ian Munro, Venkatesh Raman, and S. Srinivasa Rao. Space efficient suffix trees. *J. Algorithms*, 39(2):205–222, 2001. doi:10.1006/jagm.2000.1151.

13 Yakov Nekrich. New data structures for orthogonal range reporting and range minima queries. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 1191–1205. SIAM, 2021. doi:10.1137/1.9781611976465.73.

14 Mihai Patrascu. Succincter. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 305–313. IEEE Computer Society, 2008. doi:10.1109/FOCS.2008.83.

15 G. Pólya. Kombinatorische Anzahlbestimmungen für Gruppen, Graphen und chemische Verbindungen. *Acta Mathematica*, 68(none):145–254, 1937. doi:10.1007/BF02546665.

16 N. Sloane. The on-line encyclopedia of integer sequences. *Notices Amer. Math. Soc.*, 50:912–915, September 2003.

17 Dan E. Willard. Log-logarithmic worst-case range queries are possible in space theta(n). *Inf. Process. Lett.*, 17(2):81–84, 1983. doi:10.1016/0020-0190(83)90075-3.

18 Peisen Zhang, Eric A. Schon, Stuart G. Fischer, Eftihia Cayanis, Janie Weiss, Susan Kistler, and Philip E. Bourne. An algorithm based on graph theory for the assembly of contigs in physical mapping of DNA. *Comput. Appl. Biosci.*, 10(3):309–317, 1994. doi:10.1093/bioinformatics/10.3.309.