Performance Analysis of Maximum Independent Set Algorithms on Circle Graphs

Mohamed Bakr Difallah

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master in Computer Science(MCS)

Supervisor: David Gregg

April 2024

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

April 19, 2024

Performance Analysis of Maximum Independent Set Algorithms on Circle Graphs

Mohamed Bakr Difallah, Master in Computer Science

University of Dublin, Trinity College, 2024

Supervisor: David Gregg

Graph theory is a branch of mathematics with a great number of real world applications due to its strong ability to abstract many real world problems into a set of nodes and adjacent edges. One type of a graph is called a circle graph, which represents a circle's chords and the intersection between them. Finding the largest number of non intersecting nodes in this graph, called the maximum independent set is a problem that many algorithms have been developed to tackles, as it has applications in computational geometry, circuit development and compiler optimisation. However many of these algorithms lack experimental data to support their theoretical performance analysis. In this paper I perform experimental analysis on many of the latest algorithms for finding the maximum independent set of a circle graph, including an unpublished algorithm, and discuss the results.

Acknowledgments

I would like to thank David Gregg, my supervisor for his support and guidance. I would also like to thank everyone referenced in this paper for their scientific contribution.

Mohamed Bakr Difallah

University of Dublin, Trinity College

April 2024

Contents

List of Figures

Listings

Chapter 1

Introduction

Graph theory is a branch of mathematics that appears in a large variety of real word applications and has had many uses in the field of theoretical mathematics, computer science and engineering. Graphs are made up of nodes, which can represent any object, and edges between these nodes, which can represent connections between the objects. Graph theory appeared in the 18th century when Leonhard Euler solved the Seven Bridges of Königsberg problem by representing pieces of land as nodes and bridges between them as edges. Many problems as it turns out could be solved this way, by interpreting systems in the real world as a set of nodes and incident edges. Over time, many algorithms have emerged to solve a variety problems through this abstraction. A famous example is the shortest path problem where the nodes of a graph represent an intersection or destination and its edges represent paths between them.

Graphs have types and properties based on the relationship between their nodes and edges or based on what they represent. They also contain features that may be relevant to the problem being solved. One such featured is called an independent set, which is the set of nodes that have no edges connecting them. One type of graph is the circle graph. This is a graph that represents a chord diagram where the circle's chords are modeled as nodes and the intersection between the chords are represented with edges. In this paper we discuss algorithms for finding the cardinality of the largest independent set in a circle graph.

Many algorithms have been developed to solve this problem and theoretical analysis of these algorithms shows us that both the memory and the execution time performance of the state of the art has been constantly improving as I will discuss in section 1.2. While theoretical analysis is crucial to understanding the platform independent performance of the algorithm, empirical analysis can often reveal hidden flaws and inefficiencies, which may be unaccounted for. However there has been little to no empirical analysis of maximum independent set algorithms on circle graphs with the notable exception of Nash et al. (2009). In this paper I aim to build on the work of Nash et al. (2009) by providing empirical performance data on three algorithms presented by Nash and Gregg (2010). I will also provide data on the performance of an algorithm by Bonsma and Breuer (2009) and compare it to and unpublished improvement in Nash and Gregg (shed).

1.1 Relevant Definitions

I will now describe some important concepts that are crucial to understanding the problem of calculating the cardinality of the maximum independent set of a circle graph and computing the solution.

1.1.1 Circle Graph

A circle graph is formally defined as an undirected graph that is isomorphic to the intersection graph of a finite set of chords in a circle. In less formal terms: let's take a circle and a set of chords drawn in this circle where the chords may or may not cross each other, this is called a chord diagram, and the points where the chords touch the circle are called vertices. Each chord is represented by a node in the graph, and in the case where chords intersect, i.e. cross each other, an edge exists between the nodes that represent those chords. In the example in figure 1.1, chord C intersects with chords B, D and E in the chord diagram, so an edge exists between node C and each of the other 3 nodes. Meanwhile chord A does not intersect with any of the other chords in the circle and so

there are no edges between its corresponding node and the other nodes in the circle graph.

1.1.2 Independent Set

An independent set of a graph is a subset of vertices in a graph no two of which are adjacent, meaning that it is the list of nodes in a graph that have no edges between any of them. The maximum independent set is the largest possible independent set in the graph, i.e. the largest number of nodes that don't have edges between them. Therefore in the case of the circle graph it represents the largest possible set of chords in a circle that do not intersect with each other. Note that there can be many independent sets and even maximum independent sets formed from different combinations of chords. In figure 1.1 the chords A and C do not intersect with each other making them an independent set of size 2, however it is not the maximum independent set as there exists a larger set of chords namely A, B and D. The independence number of the circle graph is defined as the cardinality of the maximum independent set, i.e. its size. Therefore the independence number of the circle graph in figure 1.1 is 3.

1.1.3 Interval Representation

To solve the problem of finding the maximum independent set of a circle graph, most algorithms make use of an abstraction where the circle is represented as a line and each chord is represented as an interval on that line. This is known as an interval representation. The points on the line represent the vertices in the chord diagram ordered in a direction and the interval endpoints correspond to these vertices. This can be interpreted as cutting the circumference of the circle between two vertices and stretching it out flat. Many interval representations of the same circle graph can exist depending on where the circumference is cut. The density of the interval representation is the maximum number of intervals crossing any point on the real line. Chords are independent in this representation if their corresponding intervals do not overlap, or if one interval is completely inside

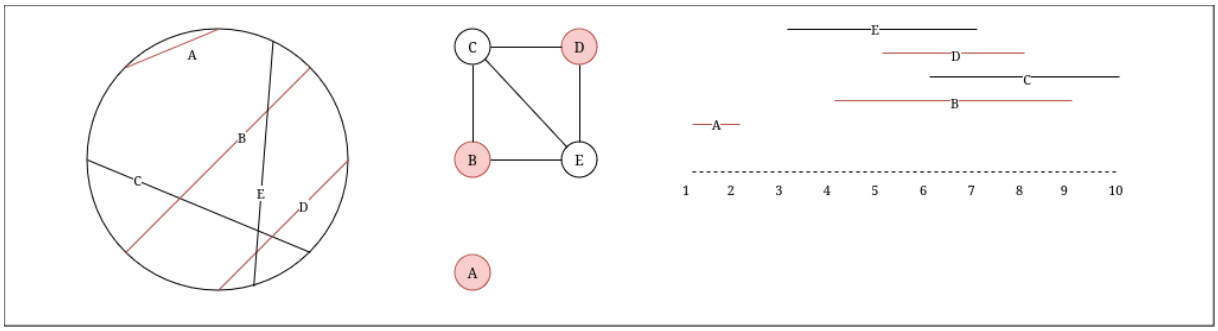


Figure 1.1: Chord diagram (left). A circle graph representing the chord diagram (middle) and their interval representation (right).

the other. The example in 1.1 shows that the independent pair of chords A and C do not overlap, while another independent pair C and D do overlap but D is completely inside C. The density of the interval representation in this case is 4 because intervals B, C, D and E overlap at a single point (6 or 7).

1.2 History and State of the Art

Algorithms for finding the maximum independent set in a circle graph began appearing in the late 20th century. Gavril (1973) presents an algorithm that solves this problem in $O(m^3)$ time (where m is the number of intervals in the interval representation). Supowit (1987) tackles the issue of routing two pin nets in a channel using two layers, a problem which often occurs in the field of microfluidic bio-chip technologies. This paper shows that the problem of finding the optimal set of nets to route is equivalent to finding the maximum independent set of a circle graph and presents a $O(m^2)$ time and space algorithm for solving this problem. Apostolico et al. (1992) present algorithms for finding cliques in a circle graph as well as its maximum independent set, the latter of which requires $O(dm)$ time and space, where d is the density of the circle graph, while Valiente (2003) presents an algorithm that solves this problem in $O(l)$ time and $O(m)$ space, where l is the total length of the chords in the interval representation. Significant improvements were made to this algorithm by Nash et al. (2009). In this paper redundant computations

were removed from both algorithms. This is especially the case with Valiente's algorithm where the optimised version was experimentally shown to perform 3 times faster than its original version. An output sensitive algorithm was developed by Nash and Gregg (2010) that runs in $O(\alpha m)$ time and $O(m)$ space where α is the independence number of the circle graph. The algorithm can be further improved by combining it with another algorithm and switching to it based on the density of the interval representation.

The algorithms given above all apply to the case where the chords in the graph do not share an endpoint, however there have been algorithms developed to solve the version of the problem where any number of chords may share an endpoint. Liu and Ntafos (1988) solve the problem of partitioning a polygon into smaller pieces. An algorithm for finding the maximum independent set of a circle graph that runs in $O(m^3)$ time is a step in that solution. Chang and Lee (1992) give an algorithm that runs in $O(nm)$ time where n is the number of endpoints in the interval representation. Another algorithm in the category is presented by Bonsma and Breuer (2009), where the problem of counting hexagonal patches in a planar graph is reduced to finding the maximum independent set of a circle graph running in $O(nm)$ time and $O(m^2)$ space.

It's interesting to note that many of these algorithms were developed as a direct result of attempting to solve another problem in computational geometry, either as a simplification or as a step in the problem, or while developing a solution to a real world application. However there is a lack of experimentation with little time and memory consumption data. A notable exception is Nash et al. (2009) as mentioned previously. This lack of experimentation may be because of the difficulty of comparing each algorithm as the format of their input may differ. This point will be elaborated on in 3.1.

Chapter 2

Algorithms

In this chapter I describe the operation of each of the algorithms that will be tested.

2.1 Endpoint Algorithms

I will begin by describing the algorithms that operate on interval representations with distinct endpoints.

2.1.1 Naive approach

One of the earlier algorithms is by Supowit (1987) which runs in $O(m^2)$ time and and uses $O(m^2)$ space, as was mentioned earlier. While the algorithm's time makes it impractical for real applications, the mathematical approach introduced by the paper forms the basis for many later algorithms. Specifically Nash and Gregg (2010) provide an algorithm similar to Supowit's which they refer to as a naive approach, which forms the basis for the output sensitive algorithm introduced in the same paper. I now describe this naive approach which will from now be referred to as the naive algorithm.

Let $i_{x,y}$ denote and interval with left endpoint x and right endpoint y. Given a set of intervals I, where $I_{x,y} \subset I$ represents the subset of intervals with left endpoint greater than or equal to x and right endpoint less than or equal to y, $MIS[I_{x,y}]$ or represent the

cardinality or size of the maximum independent set of intervals between endpoints x and y. Similarly $CMIS[I_{x,y}]$ or $CMIS[i_{x,y}]$ represents the cardinality or size of the contained maximum independent set of intervals between the endpoints x and y, not including the interval $i_{x,y}$ if it exists. This can be represented as $CMIS[I_{x,y}] = MIS[I_{x+1,y-1}]$. The algorithm makes use of the following properties. Given an interval $i_{x,y}$ on the real line in the interval representation:

If x is the right end point of an interval, then $MIS[I_{x,y}] = MIS[I_{x+1,y}]$. This follows since if x is the right endpoint of an interval then its left endpoint is less than x and hence outside the range $[x, y]$, meaning that we can be certain that its associated interval is not in the set of intervals $I_{x,y}$ allowing us to exclude the point x. If x is the left endpoint of an interval $j = i_{x,z}$, then: $MIS[I_{x,y}] = MIS[I_{x,y}]$ if $z > y$, otherwise $MIS[I_{x,y}] = max(MIS[I_{x+1,y}], 1 + CMIS[j] + MIS[I_{z+1,y}])$. Similarly to the first property, if z, which is the right endpoint to x, is greater than y, then it falls outside the range $[x, y]$, meaning that the interval j is not in the set of intervals in $I_{x,y}$, which again allows us to exclude the point x. However if z is less than y, we derive the MIS value from three components: the interval itself (1), the MIS contained in the interval ($CMIS[j]$) and the remaining intervals that come after ($MIS[I_{z+1,y]}$). A greater MIS value may have already been calculated so the $MIS[I_{x+1,y}]$ is selected instead if its value is larger.

Two lists are maintained, M of size n which represents $MIS[I_{x,y}]$ and C of size m which represents $CMIS[I_{x,y}]$. The algorithm iterates over the interval line from left to right and assigns the value to y. If y is the right endpoint of an interval $i = [x, y]$. then $C[i] = M[x + 1]$, since it has already finished calculating the $CMIS$ for every interval in the set of intervals in $I_{x,y}$. For each iteration of y it iterates down from $y-1$ to 0, assigning this value to x. Applying the first property, $M[x] = M[x + 1]$. Then, if x is the left end point of an interval $j = [x, z]$, the second property is applied. $M[x]$ is overwritten with the larger value between $M[x + 1]\text{and}1 + C[j] + MIS[z + 1]$. At the end of the algorithm, $M[0]$ contains the cardinality of the maximum independent set. The pseudo-code can be seen in 2.1.1

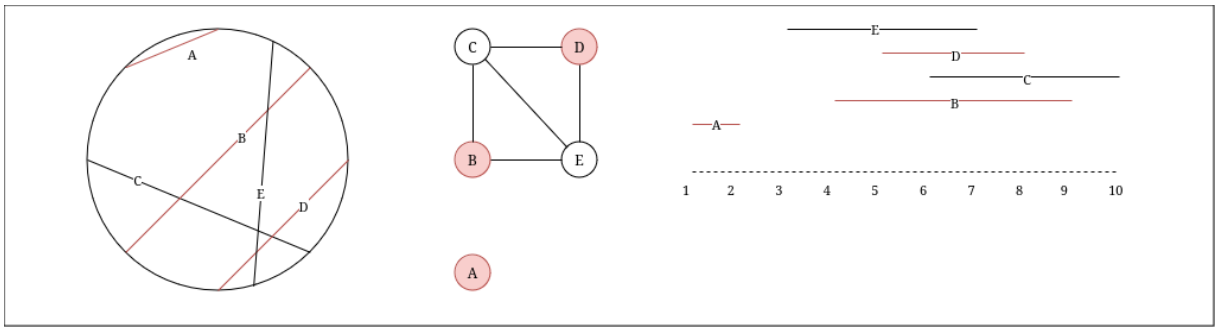


Figure 2.1: A copy of figure 1.1 placed here for convenience.

```
m = <number of intervals>
n = m * 2

M = [0] * n
C = [0] * m
for y from 0 upto n - 1:
    if y is the right endpoint of i = (z, y):
        C[i] = M[z + 1]
    for x from y - 1 downto 0:
        M[x] = M[x + 1]
        if x is the left endpoint of j = (x, z) and z <= y :
            M[x] = max(M[x + 1], 1 + C[j] + M[z + 1])

return M[0]
```

Listing 2.1: Pseudo-code for the naive algorithm

2.1.2 Valiente

Valiente (2003) introduces an algorithm for calculating the maximum independence set of a circle graph that requires $O(l)$ time and $O(n)$ space complexities, where l is the total length of all intervals in the interval representation. Though as we will show later

this is misleading as Nash et al. (2009) make significant improvements to this algorithm through time reducing optimisations by removing redundant calculations and skipping unnecessary iterations.

Valiente's algorithm makes use of a simple reduction where the problem of calculating the maximum independent set of a circle graph is reduced to calculating the maximum independent set of an interval graph. An interval graph is similar to a circle graph, with the exception being that two intervals also intersect if one interval is completely inside the other. Looking at figure 2.1, while chords B and D don't intersect in the circle graph they would intersect in an interval graph. The maximum independent set of an interval graph can be found in$O(n)$ time and space complexity, given the $CMIS$ values of all intervals, using two properties introduced by Gupta et al. (1982) that are similar to the properties described in the naive approach. Let $G[I_x]$ be the cardinality of the maximum independent set of intervals between endpoints x and n: $G[I_x] = MIS[I_{x,n}]$: If a is the right end-point of an interval, then $G[I_x] = G[I_{x+1}]$. This makes sense since if x is the right end-point of an interval $i = i_{z,x}$, it must be the case that $z < x$ and hence z is outside the range $[x, n]$. Therefore x can be excluded, since its corresponding interval is not in the range. In addition $G[I_n] = 0$, since there is only 1 end-point in the range $[n, n]$ and as such cannot it contain any intervals. If x is the left end-point on an interval $i = i_{x,y}$, then $G[I_x] = max(G[I_{x+1}], CMIS[i] + G[I_{y+1}])$. $CMIS[i]$ denotes the maximum independent set contained in $i_{x,y}$, and $G[I_{y+1}$ refers to the maximum independent set outside after $i_{x,y}$, since these ranges are disjoint they can be used to form a maximum independent set. $G[I_{a+1}]$ is in the same range but may have a larger value from a previous calculation, so that value is used instead in that case. $G[I_0]$ returns the weight of the maximum independent set.

The algorithm that uses the aforementioned properties allows us to obtain the maximum independent set of an interval graph given values for $CMIS[I_{x,y}]$ in $O(n)$ time as mentioned before. To obtain the maximum independent set Valiente's algorithm calculates the $CMIS$ values for every interval in the circle graph. The original version of

the algorithm ensures that $CMIS$ values are calculated by sorting the intervals in non-decreasing order of length. Since an interval can only contain shorter intervals this ensures that all $CMIS$ values are available when calculating the $CMIS$ value for any interval. The optimization instead achieves this by scanning left to right over the real line and at each right end-point calculating the $CMIS$ values via a right to left scan. During the right to left scan, the algorithm will inevitably encounter the right endpoint of the value of the interval it has just calculated, therefore a value is maintained that stores the left endpoint of that interval to avoid recalculating its $CMIS$ value.

The algorithm works as follows: Two lists are maintained M and C, with each of their cells set to 0, and a value $last$ which is set to 0. Then the algorithm iterates from 0 to $n-1$ using a variable y. On each iteration if y is the right endpoint of an interval $i_{z,y}$, then the algorithm iterates downwards from $last$ to $z+1$ using a variable x. On each iteration if x is the left end-point of an interval $j = i_{x,v}$ and $M[v+1] + C[j] > M[x+1]$ then the assignment $M[x] = M[z+1] + C[j]$ occurs, otherwise the assignment $M[q] = M[q+1]$ occurs. After the inner loop ends, the assignment $C[j] = M[x+1]+1$ occurs and variable last is set to x. When the outer loop ends, another downward loop is done from $n-2$ to 0 and the same steps as the previous downward loop are performed, with the exception of the assignment to $C[j]$ since we already have the $CMIS$ values. The cell $M[0]$ then contains the cardinality of the maximum independent set when the algorithm terminates. The pseudo-code for this algorithm can be seen in 2.1.2.

```
m = <number of intervals>
n = m * 2

M = [0] * n
C = [0] * m

last = 0
for y from 0 upto n - 1:
    if y is the right endpoint of i = (z, y):
        for x from last downto z + 1:
            M[x] = M[x + 1]
            if x is the left endpoint of j = (x, z):
                M[x] = max(M[x], C[j] + M[z + 1])
        C[i] = M[z + 1] + 1

        last = z

for x from n - 1 downto 0:
    M[x] = M[x + 1]
    if x is the left endpoint of i = (x, y):
        M[x] = max(M[x], C[i] + M[y + 1])
return M[0]
```

Listing 2.2: Pseudo code showing the operration of Valiente's algorithm. Note the second loop running n iterations using using the CMIS values to generate the cardinality of the maximum independent set

2.1.3 Nash-Gregg and its Combined Variant

The paper by Nash and Gregg (2010) builds on the naive approach discussed earlier to present an output sensitive algorithm that runs in $O(\alpha n)$ time where α is the independence number of the circle graph, or in other words, the output result of the algorithm. This is very useful in cases where the maximum independent set of a circle graph is expected to be very small relative to the total set of chords, and smaller than the density.

This algorithm uses the following definition of an update set S_y, $S_y = \{x | MIS[I_{x,y}] > MIS[I_{x,y-1}], 0 \leq x \leq y\}$. In addition, it is shown that $MIS[I_{x,y}] = 1 + MIS[I_{x,y-1}]$ for each x in the set S_y.

Sets M and C are maintained just like the naive approach, that represent MIS and $CMIS$ respectively. The algorithm iterates over the interval line from left to right and assigns the value to y. If y is the right end-point of an interval $j = i_{z,y}$ the assignment $C[j] = M[z + 1]$ occurs. Then instead of iterating down to 0, a function called update is called by passing in lists C and M and endpoint y.

The update algorithm creates a stack S that represents the update set S_y. If y is the right endpoint of an interval $j = i_{x,y}]$ the assignment $M[x] = 1 + C[j]$ occurs and x is pushed onto the stack. Then the following is performed until the stack is empty. The stack $Stack$ is popped into x and if $x > 0 and M[x] > M[x - 1]$ then the assignment $M[x-1] = M[x]$ occurs, then $x-1$ is pushed onto $Stack$. If $x-1$ is the right end-point of an interval $k = i_{p,x-1}$ and $1+C[k]+M[x] > M[p]$ then the assignment $M[p] = 1+C[k]+M[x]$ occurs and p is pushed onto the stack. When the stack is empty the update function returns and the algorithm continues its iteration.

When the update function returns $M[x] = MIS[I_{x,y}]$ if the list M has been modified. Note that the assignments in the update algorithm apply the two properties discussed in the naive approach. The time of the algorithm $O(\alpha n)$ is derived from the iterations performed by popping and pushing onto the stack. A stack push occurs whenever an assignment to M occurs and that only occurs whenever a cell of M is overwritten by a

greater number, which means that the size of the stack is directly related to the cardinality of the independent set. I have also noted that $M[I_{x,y}] = 1 + M[I_{x,y-1}]$ for each $x \in S_y$ showing that each assignment of M increases the value by 1, meaning the total size of the stack and independence number are very close. As the update function iterates over the size of the stack this means its iteration time is directly related to the independence number, which results in the $O(\alpha n)$ time complexity. The pseudo-code can be seen in 2.1.3

This algorithm can be further conditionally improved by combining it with the optimised version of Valiente's algorithm. This is done by constantly checking if the independence number of the circle graph surpasses the density and switching the calculation to Valiente's algorithm whenever this occurs. This is useful for applications where there may be a large difference between the density of the graph and the independence number but it is uncertain which is larger.

The algorithm works by modifying the update function to return a boolean flag. Whenever the function modifies the list M, it checks if the value that has just been entered is larger than the density. If that is the case the function terminates and returns True. In the main function, if the flag returned by update is set, then the algorithm is aborted and Valiente's algorithm is called instead. Otherwise it continues to run the output sensitive algorithm as normal. Since Valiente's algorithm has a time complexity of $O(l)$ where $l \leq dn$ and the standard version of this algorithm has a time complexity of $O(\alpha n)$, this combined algorithm has a time complexity of $O(nmind, \alpha)$.

```
m = <number of intervals>
n = m * 2

M = [0] * n
C = [0] * m
for y from 0 upto n - 1:
    if y is the right endpoint of i = (z, y):
        C[i] = M[z + 1]
    update(y)

return M[0]

update (y):
Stack = []
if y is the right endpoint of an interval i = (x, y):
    M[x] = 1 + C[i]
    Stack.push(x)
    while Stack is not empty:
        z = Stack.pop()
        if M[z] > M[z - 1]:
            M[z - 1] = M[z]
            Stack.push(z - 1)
        if z - 1 is the right endpoint of an interval
        j = (zp, z - 1):
            if 1 + C[j] + M[z] > M[zp]:
                M[zp] = 1 + C[j] + M[z]
                Stack.push(z)
```

Listing 2.3: The basic variant of the Nash-Gregg output sensitive algorithm

2.2 Exclusive Endpoint Algorithms

In this section we will describe the operation of two algorithms that can operate on a circle graph where the chords may share an endpoint. These algorithms can also operate on exclusive endpoint circle graphs, though they are significantly slower.

2.2.1 Bonsma-Breuer

So far we have only described algorithms that operate on circle graphs where the chords do not share an endpoint. These algorithms cannot operate on circle graphs where the chords may have common endpoints as accessing a chord via its endpoint may no longer return a single value. This problem can be solved by converting this circle graph to a form where the chords do not share endpoints as shown in figure 2.2. This is done by creating new endpoints for each shared endpoint and ensuring that the newly created chords intersect. However as noted by Bonsma and Breuer (2009), this can take $O(n^4)$ in the worst case, making it an inefficient and slow solution.

Bonsma and Breuer (2009) have developed an algorithm that solves this problem in $O(nm)$ time and $O(max(m,n))$ space complexity algorithm that is described now:

$L_{x,y}$ denotes the set of intervals with left endpoint at x and right endpoint less than y.

$L_{x,y} = \{i_{x,z} | i \in I, z \leq y\}$

$MIS[S]$ denotes the cardinality of the maximum independent set contained in the set of intervals S. $U[S]$ denotes a value formed from a set of intervals S according to the following definition: $U[S] = max\{(1 + CMIS[i] + MIS[I_{x+1,y}])\forall i_{x,y} \in S\}$. Similarly to the previous algorithms discussed, a recurrence is used for computing the maximum independent set. This recurrence is makes use of the aforementioned definitions as follows: $M[I_{x,y}] = max(MIS[I_{x,y}], U[L_{x,y}])$

Two arrays, M and C, are maintained that represent MIS and $CMIS$ respectively. The algorithm operates by iterating from 0 to $n-1$ and assigning that value to y, at

each iteration, it iterates downwards from $y-1$ to 0 assigning that value to x. In each iteration the assignment $M[x] = M[x+1]$ occurs. Then, for each interval $i_{x,z}$ where $z \leq y$, if $1+C[i]+M[z+1] > M[x]$ then the assignment $M[x] = 1+C[i]+M[z+1]$ occurs. This loop over intervals $i_{x,z}$ is the application of the definition U[S]. To finish the downward loop, if there is an interval $j = i_{x,y}$, then the assignment $C[j] = M[x+1]$ occurs. The pseudo code for this algorithm is in 2.2.1.

```
m = <number of intervals>
n = <number of endpoints>


M = [0] * n
C = [0] * m
for y from 0 upto n - 1:
    for x from y - 1 downto 0:
        M[x] = M[x + 1]
        foreach interval j = (x, z) where x <= z:
            M[x] = max(M[x + 1], 1 + C[j] + M[z + 1])
        if x is the left endpoint of i = (x, y):
            C[i] = M[x + 1]
return M[0]
```

Listing 2.4: The basic implementation of the Bonsma-Breuer algorithm

2.2.2 Improved Bonsma-Beuer

The approach taken by Bonsma and Breuer's algorithm is notably similar to the naive approach discussed earlier. The algorithm computes $MIS[I_{x,y}]$ for every possible pair of x and y in the iteration which is unnecessary as many of these values are not used in the computation. In an unpublished paper Nash and Gregg (shed) suggest an improvement to this algorithm that is inspired by Valiente's algorithm.

The problem is reduced to calculating the values of $CMIS$ for each interval the using these value to calculate the maximum independent set of an interval graph. This maximum independent set can be computed given the $CMIS$ values. If there are intervals $j = i_{x,y}$ and $k = i_{z,y}$, then if $x > z$, i.e. j is shorter than k, $CMIS[j]$ will have already been computed when evaluating $CMIS[k]$. This occurs because during the downward loop, the $CMIS$ will be calculated for all intervals that share a right endpoint, with the

$CMIS$ for intervals with higher left endpoints being computed first.

Two arrays, M and C, are maintained that represent MIS and $CMIS$ respectively. The algorithm iterates from 0 to $n-1$ and assigns the value to y. In each iteration, if there is an interval j $= i_{v,y}$, the algorithm iterates downwards from $y-1$ to v assigning the value to x, otherwise the iteration is skipped. In each iteration the assignment $M[x] = M[x+1]$ occurs, then for each interval $k = i_{x,z}$ where $z < y$, if $1 + C[k] + M[z+1] > M[x]$ then the assignment $M[x] = 1 + C[k] + M[z+1]$ occurs. Let c_y be the number of intervals contained in the longest interval with right end-point y. C denotes the sum of all $c_y \in \{0, 1, \ldots, n-1\}$. Similarly let l_y be the length of the longest interval with right endpoint y, and let L be the sum of all $l_y \in \{10, 1, ..., n-1\}$. This improved algorithm has a time complexity of $O(L + C)$. 2.2.2 shows the pseudo-code for this algorithm.

```
m = <number of intervals>
n = <number of endpoints>

M = [0] * n
C = [0] * m

for y from 0 upto n - 1:
    if y is the right endpoint of i = (z, y) get the longest:
        for x from y - 1 downto z + 1:
            M[x] = M[x + 1]
            foreach interval j = (x, z) where x <= z:
                M[x] = max(M[x + 1], 1 + C[j] + M[z + 1])
            if x is the left endpoint of i = (x, y):
                C[i] = M[x + 1]

for x from n - 1 downto 0:
    M[x] = M[x + 1]
    foreach interval j = (x, z) where x <= z:
        M[x] = max(M[x + 1], 1 + C[j] + M[z + 1])
return M[0]
```

Listing 2.5: The improved variant of the Bonsa-Breuer algorithm. Note the similarity to Valiente's algorithm in the final loop (for x from n - 1 downto 0)

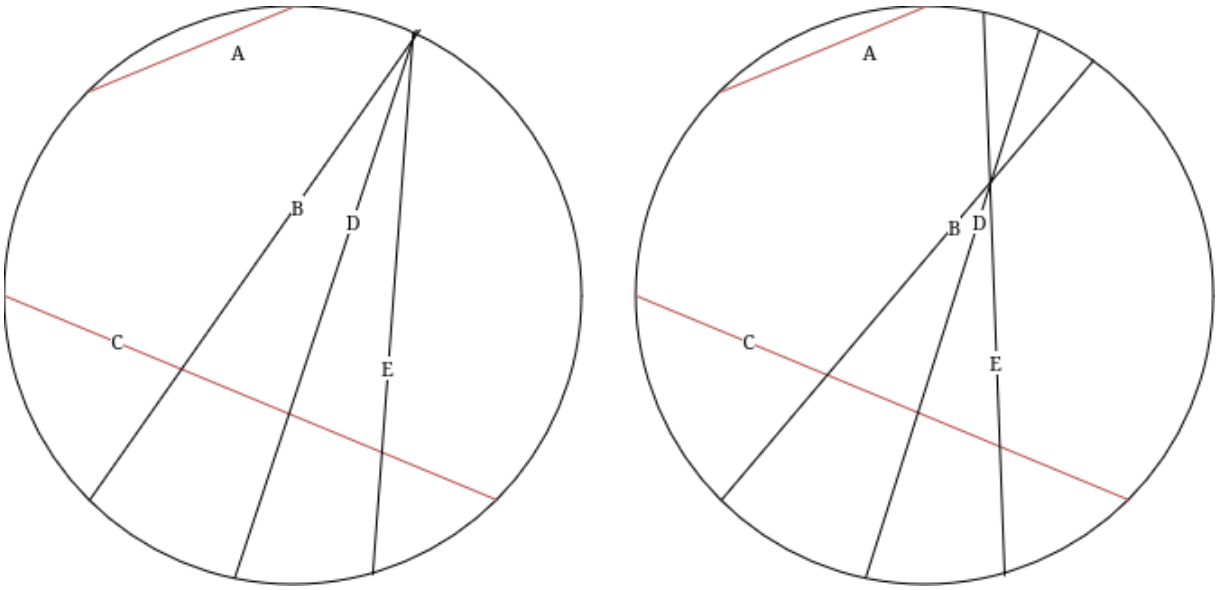


Figure 2.2: Chord diagram with shared endpoints (left), is converted to a chord diagram with exclusive endpoints by splitting the endpoint so that the chords still intersect.

Chapter 3

Evaluation

In this chapter I describe the methodology used to test the algorithms described in Chapter 2, then discuss the results. The following will be used to test execution time, memory usage at peak and iteration count of each algorithm. The code was written in C++ and the standard library function $chrono :: high\_resolution\_clock :: now()$ is used to obtain the time measurements. The measurements were all collected on a machine running a AMD Ryzen 5600x. Table 3.1 clarifies how the algorithms are referred to from here on out and in the figures.

Name	Description
Naive	The naive approach similar to Supowit's algorithm described in section 2.1.1
Valiente	The **improved** variant of Valiente's algorithm described in section 2.1.2
Nash-Gregg	The standard variant of the output sensitive algorithm described in 2.1.3
Combined	The combined variant of Nash-Gregg described in the same section 2.1.3
Bonsma-Breuer	The aglorithm for common endpoints described in section 2.2.1
Improved	The improved variant of Bonsma-Breuer described in section 2.2.2

Table 3.1: This table clarifies which algorithm is being referred by which name

3.1 Experiments

I now describe the experimental setup, how the sample inputs were generated and how these inputs were used to obtain the data.

3.1.1 Lookup Table

When describing the execution time and memory complexity of the algorithms, the time it takes to obtain certain information about the interval representation such as whether a given endpoint is a right or left endpoint, or how many intervals it is incident to is often not addressed. The speed of the first operation for example would vary depending on how the input is processed, but could take $O(m)$ time in the worst case, where m is the number of intervals. To avoid this variation a lookup table is generated using the interval representation as input, by storing relevant information in arrays and using endpoints as indexes. This lookup table allows the algorithms to obtain the relevant information in $O(1)$ for the distinct endpoint case. It operates by storing several variables and lists, namely a list of intervals and an list of endpoints that use endpoints as keys. The variation of the lookup table for the distinct endpoint case is created in $O(m)$ time and uses $O(n)$ space, exposing the following functions:

getDensity: This function retrieves the density of the graph, which is obtained in $O(l)$ time where l is the total length of all intervals.

getNumOfIntervals: This retrieves the number of intervals. The number of endpoints is derived by doubling this value.

getCorrespondingInterval: This retrieves the interval incident to a given endpoint. Intervals are returned as key to and array that stores the interval.

getCorrespondingEndpoint: Given an endpoint, this retrieves the other endpoint that forms the same interval.

Due to the method of storing information in arrays with endpoints as indexes, this same lookup table cannot be used for the common endpoint case because the indexes would then conflict as they may have multiple corresponding intervals and endpoints. A different lookup table is used instead that is adapted for the this case. The performance of the lookup table is slightly worse as it needs to iterate over multiple intervals and endpoints to retrieve the correct information. To facilitate this a significantly larger amount is memory is used by making the array of intervals 2-dimensional and making the array of endpoints store a dynamic list of its corresponding endpoints. This table is still created in $O(m)$ time but uses $O(n^2)$ space. Its functions are the following:

getNumOfIntervals and getNumOfEndpoints: Retrieves the number of intervals and the number of endpoints respectively. Unlike the distinct endpoint lookup table, the number of intervals doesn't correspond to the number of endpoints so they cannot be derived from each other. This takes $O(1)$ time.

getIntervalsBefore: Given inputs x and y, this retrieves the list of intervals with left endpoint x and right endpoint less than or equal to y. This takes $O(m')$ time in the worst case where m' is the frequency of the most common endpoint.

getInterval: Given two input x and y this retrieves the interval with x and y as endpoints.

getLongestInterval :Given and endpoint y this retrieves the longest interval with right endpoint at y. This takes $O(1)$ time as the intervals in each list were sorted in the initialisation step.

3.1.2 Interval Generation

To test the distinct algorithms I followed a methodology similar to Nash et al. (2009) to generate the intervals two types of intervals. The first was generated by creating a list of values from 1 to n where n is the desired number of endpoints. This ordered list was then

shuffled to create a random permutation of said values. The intervals are then obtained by taking each pair of values as the two endpoints of an interval. Mathematically, the intervals can be represented by a permutation of the set $S = \{0, 1, \ldots, n-1\}$, where an interval i is obtained from $i = [\, S_{2i}, S_{2i+1}]$. This type of interval will henceforth be called a *Permutation* interval.

The second type of interval was generated using a method described by Scheinerman (1990). This method allows for the generation of intervals of various lengths granting control over the density of intervals. These intervals are generated by selecting a set of random values in the range [0, 1] to be a centre C_i in the set $C = \{C_1, C_2, \ldots, C_n\}$. Then another set of random values in the range $[0, RMax]$ is selected to be radius R_i in the set $R = \{R_1, ..., R_n\}$. Then for each $C_i \in C$ and $R_i \in R$, interval $i = [C_i - R_i, C_i + R_i]$. The values can then be substituted for their rank in ascending order to determine the interval endpoints. $RMax$ is the maximum possible length of the radius which will vary depending on the density of intervals we want to generate. This type of interval will be called *Radial* intervals. Nash et al. (2009) show that the mean density of a large number of Permutation intervals is $n/2$, while the mean density of radial intervals increases greatly from at values of $RMax$ from 0 to 3 and begins to plateau at 4.0, as can be seen from figure 3.1a. We can also see from figure 3.1b that changing the value of $RMax$ initially decreases the independence number of the generated intervals, but increases again beginning at around $RMax$ 1.0. Recall from 1.1.3 that two chords form an independent set if its corresponding intervals don't intersect or one is completely inside the other. Initially, the low value of $RMax$ generates intervals that are too short to intersect with one another, but as it increases that longer intervals begin intersecting with each other. Increasing the range further however creates opportunities for intervals to be inside one another increasing the independence number. For the test cases, permutation intervals will be generated with sizes from 1000 to 20000 at a step of 1000. 15 of each size will be generated and the mean of results are used to obtain the data. Radial intervals will also be generated at a fixed size of 5000, with $RMax$ increasing from 0.2 to 10 at a step of 0.2.

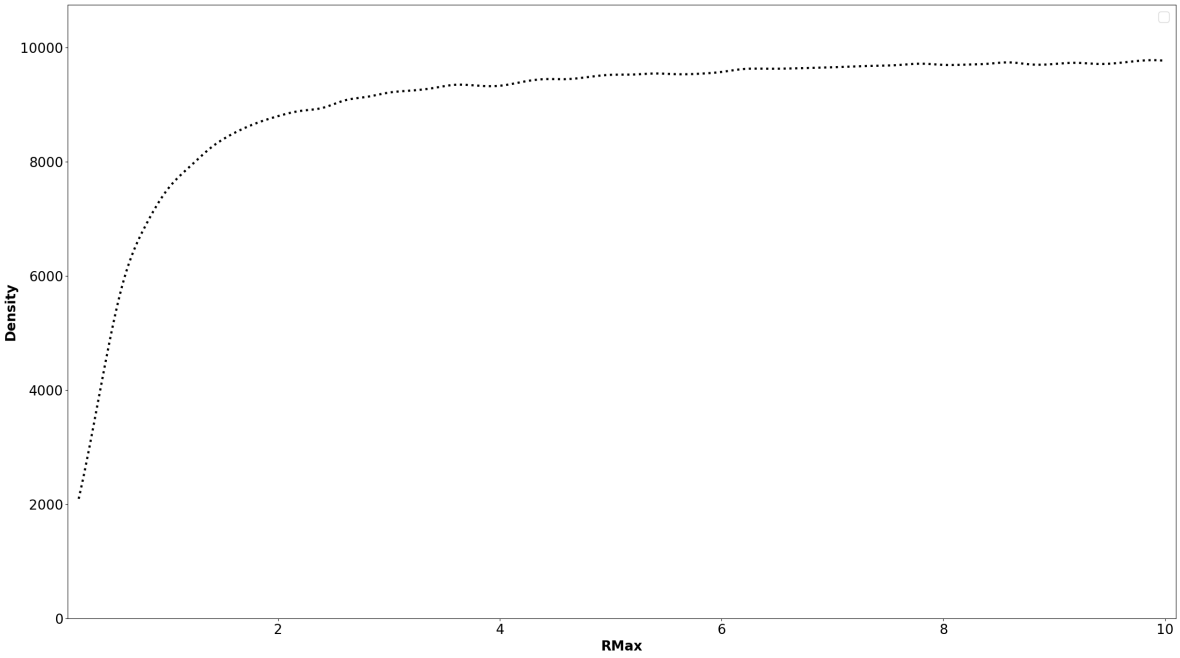
For the common endpoint case I now present a method for generating intervals that may share common endpoints: Given the set of endpoints $S = \{S_1, S_2, ..., S_n\}$ and an empty set X, a random index i is generated in the range $[1, n]$. While $S \neq \emptyset$ An element S_i is removed from S and added to X. There is a probability $PKeep$ however that S_i will not be removed from S allowing it can be reused as an endpoint. Increasing $PKeep$ increases the frequency that and endpoint is reused, which in turn increases the number of intervals generated. This is done until the S is empty. Every pair must be checked to ensure that no two endpoints of the same interval are the same. When $S = \emptyset$, X may have an odd number of endpoints insufficient to generate the intervals. In this case a random value from X can be chosen as the final endpoint. Figure 3.2 shows how the number of intervals generated increases as $PKeep$ is increased. The number initially increases slowly then begins to increase rapidly as $PKeep \rightarrow 100$. This exponential increase is reflected in the increase of the density shown in figure 3.3a. Figure 3.3b shows a similar albeit slower rate of increase, due to new intersections being formed as the number of intervals increases.

3.2 Results for Distinct Endpoint Algorithms

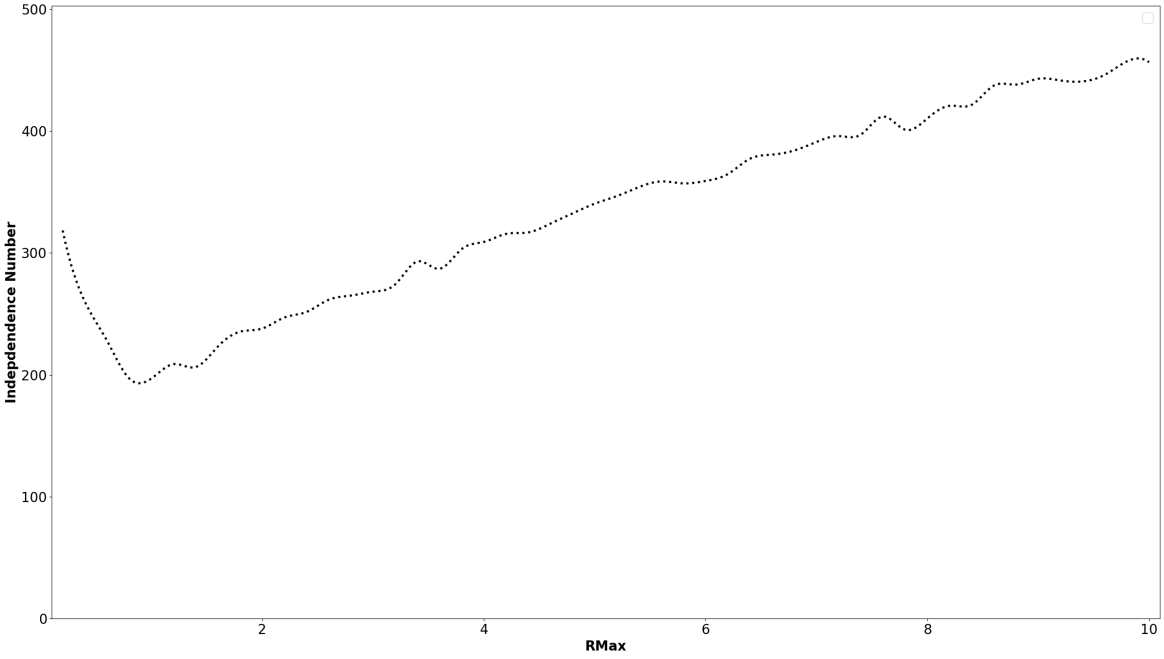
The following are the results of testing the distinct endpoint algorithms and comparing them.

3.2.1 Execution time

I now discuss the results of measuring the execution time of the algorithms. 3.4a shows the time of each algorithm of the distinct endpoint algorithms for permutation type intervals as they increase in amount. As expected, the naive algorithm by far had the worst execution time out of the four algorithms tested. Valiente's algorithm can be seen to perform drastically better, running 20 times faster at an interval count of 20000. The output sensitive algorithm and its combined variant performed the best out of all algo-



(a)



(b)

Figure 3.1: The change in density of the interval representation *(a)* and its independence number *(b)* as $RMax$ is increased

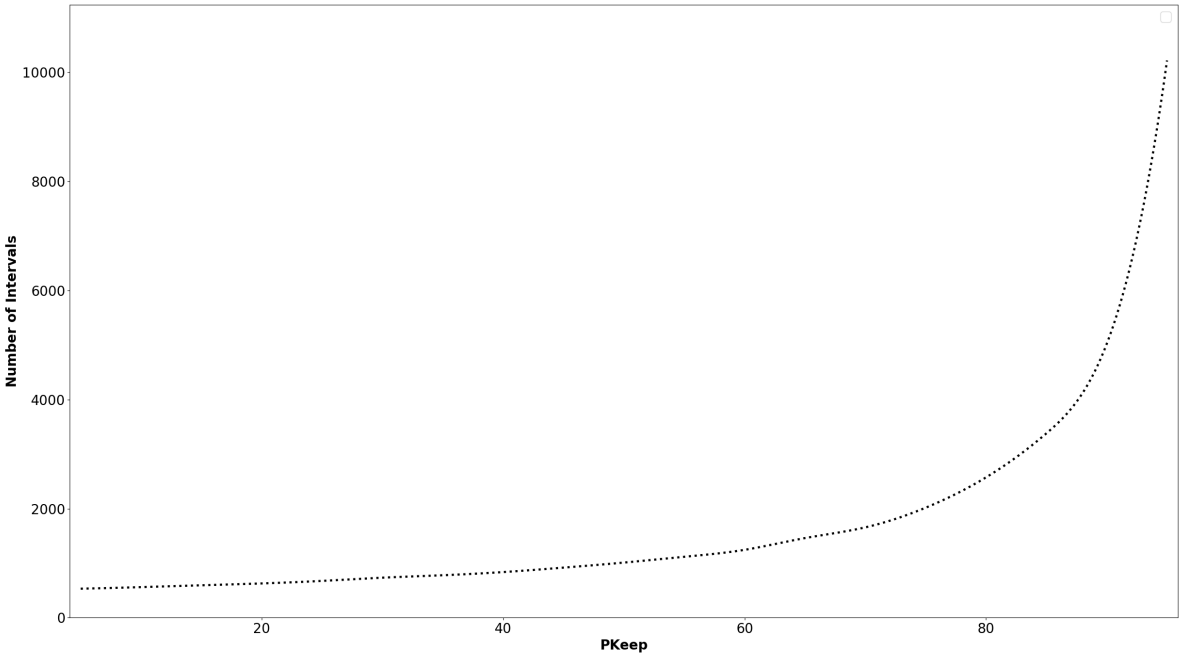


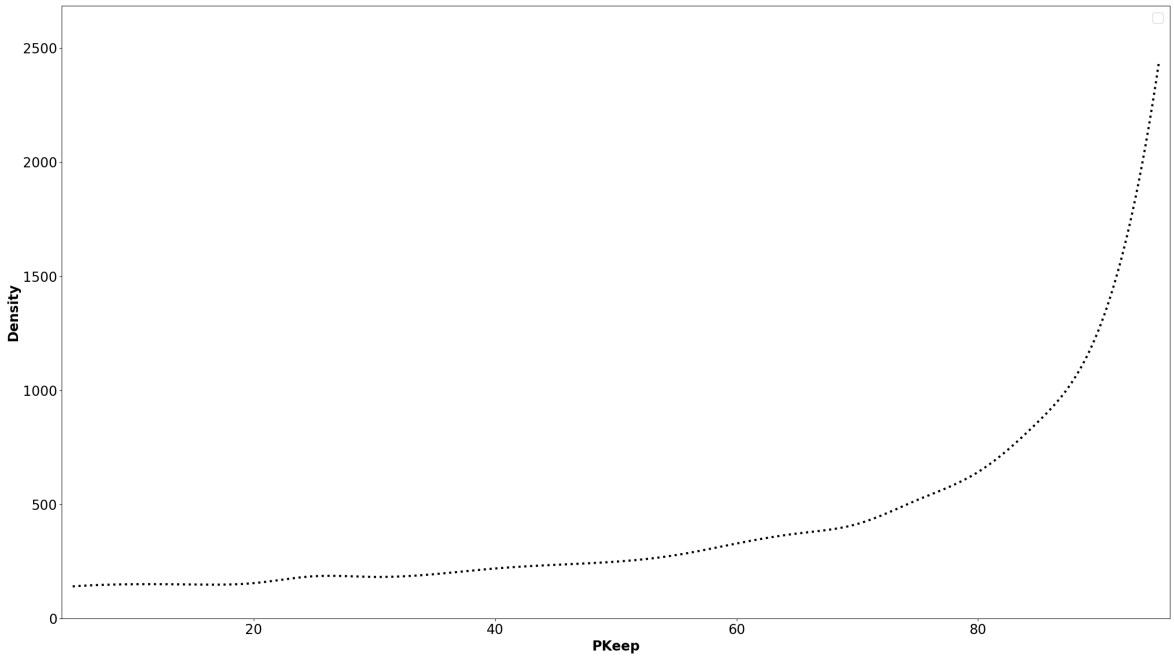
Figure 3.2: The change in the number of intervals in the interval representation as $PKeep$ is increased

rithms with the combined variant running very slightly faster. These two algorithms ran 3 times faster than Valiente's algorithm at 20000 intervals.

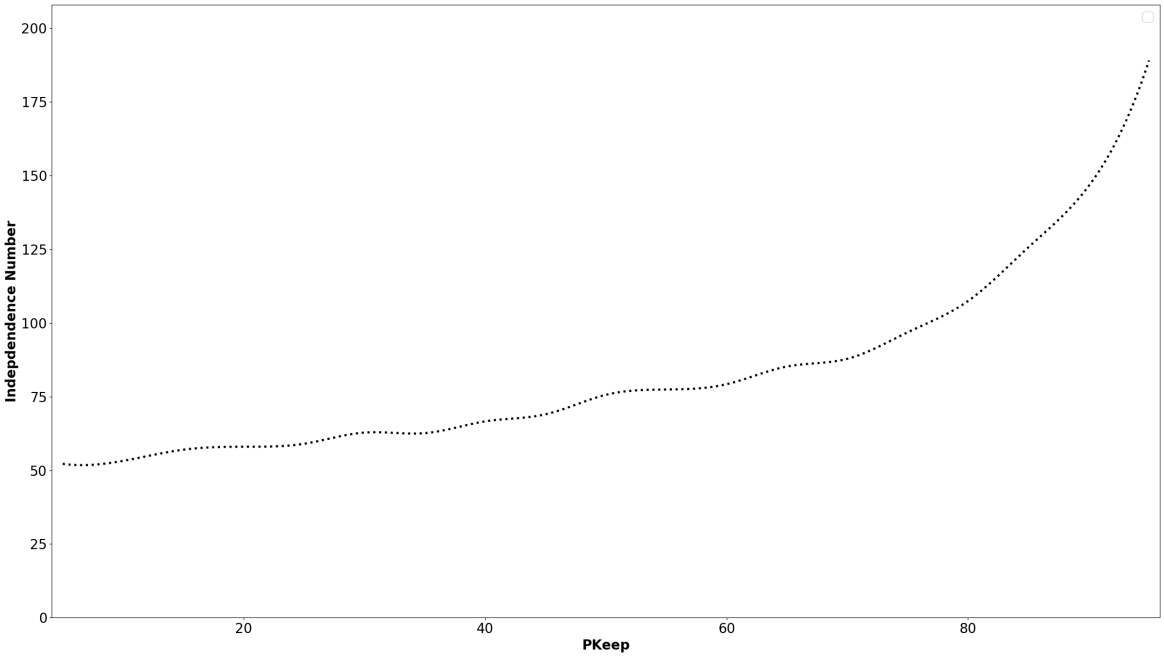
Figure 3.4b shows the execution time of each algorithm with radial intervals as $RMax$ is increased. The naive approach performed the worst as expected, with Valiente's algorithm initially getting slower and then speeding up again as the density increases, which highlight its deceptive $O(l)$ execution time. The Nash-Gregg algorithm and its combined variant perform identically, initially getting faster as the independence number is reduced as shown in figure 3.1b, then remaining stable. The combined algorithm only shows improvements at really low density values, when $RMax < 0.05$. It cannot switch to Valiente's algorithm at higher densities as the independence number tends to be much lower than the density, causing it to never trigger the switch condition.

3.2.2 Memory Consumption

I now discuss the memory performance of the algorithms. Figure 3.5a shows the memory consumption of each algorithm for permutation type intervals. The memory consumption

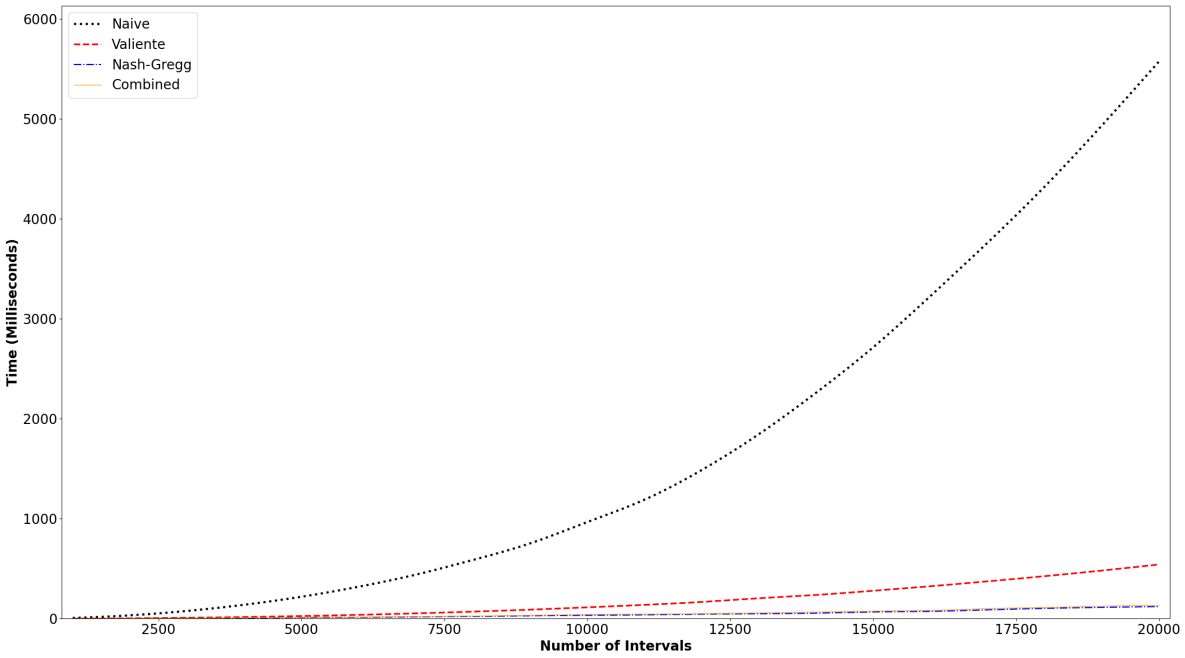


(a)

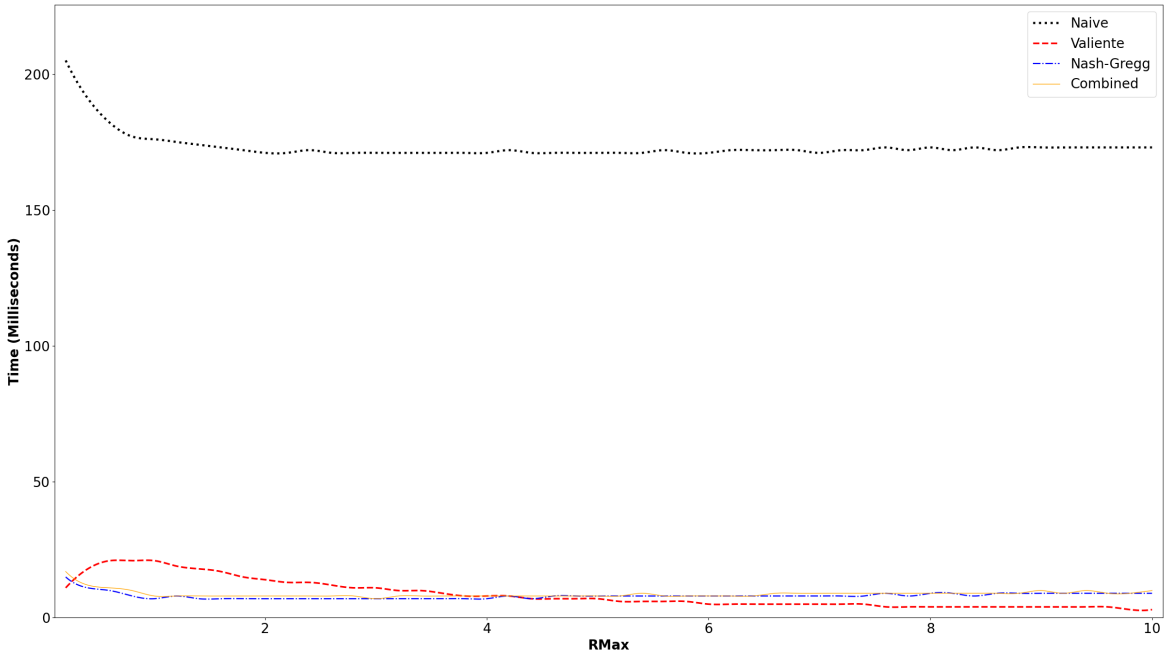


(b)

Figure 3.3: The change in the density in the interval representation *(a)* and its independence number *(b)* as $PKeep$ is increased.



(a)



(b)

Figure 3.4: The execution time of the distinct endpoint algorithms. In *(a)* permutation intervals are used the interval size is increased, and in *(b)* the interval size is kept static at 5000 and $RMax$ is increased

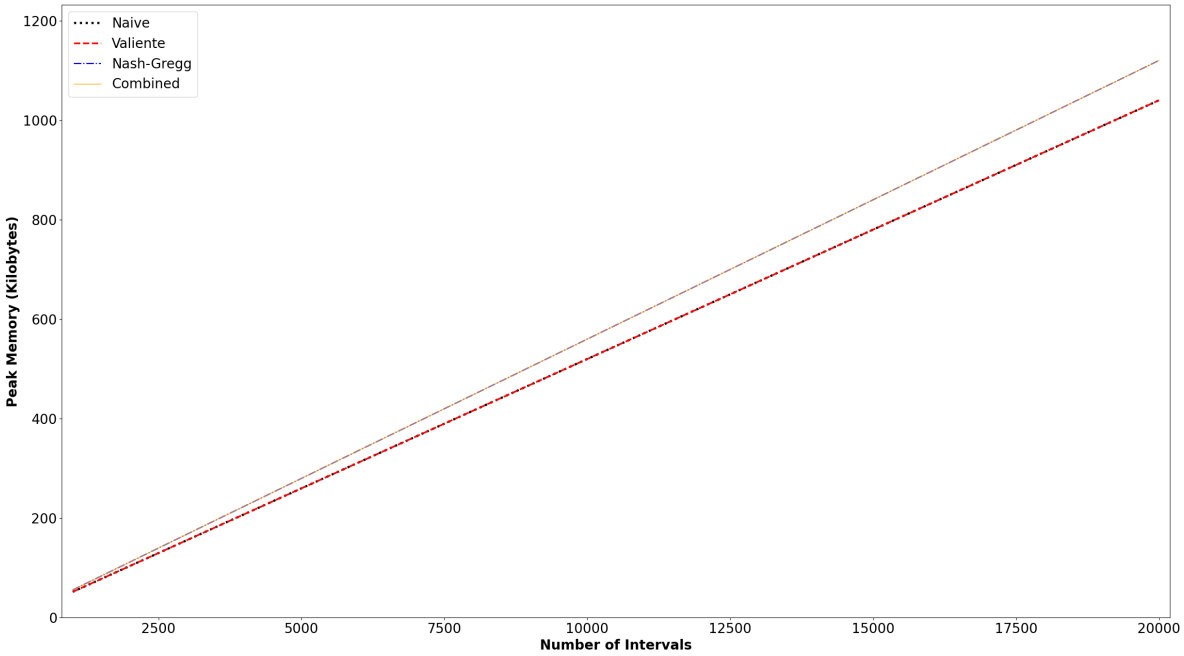
of the Naive approach and Valiente's algorithm is the same, as both only store the $CMIS$ and MIS arrays when computing the the independence number. The Nash-Gregg algorithm and the combined algorithms share the same memory consumption, which is higher than the former two algorithms. The stack data structure used in these two algorithms is created by initialising an array of size m where m is the number of intervals, and changing a variable size to modify the correct cell when popping and pushing. While this consumes more memory, the alternative of using a dynamic stack was much slower.

Figure 3.5b shows the memory consumption of each algorithm as the radius is increased. Since the memory complexity $O(n)$ depends entirely on the number of endpoints, there was no change in memory consumption, which was expected. Once again the consumption of the Nash-Gregg algorithm and its combined variant is slightly larger due to the statically sized stack.

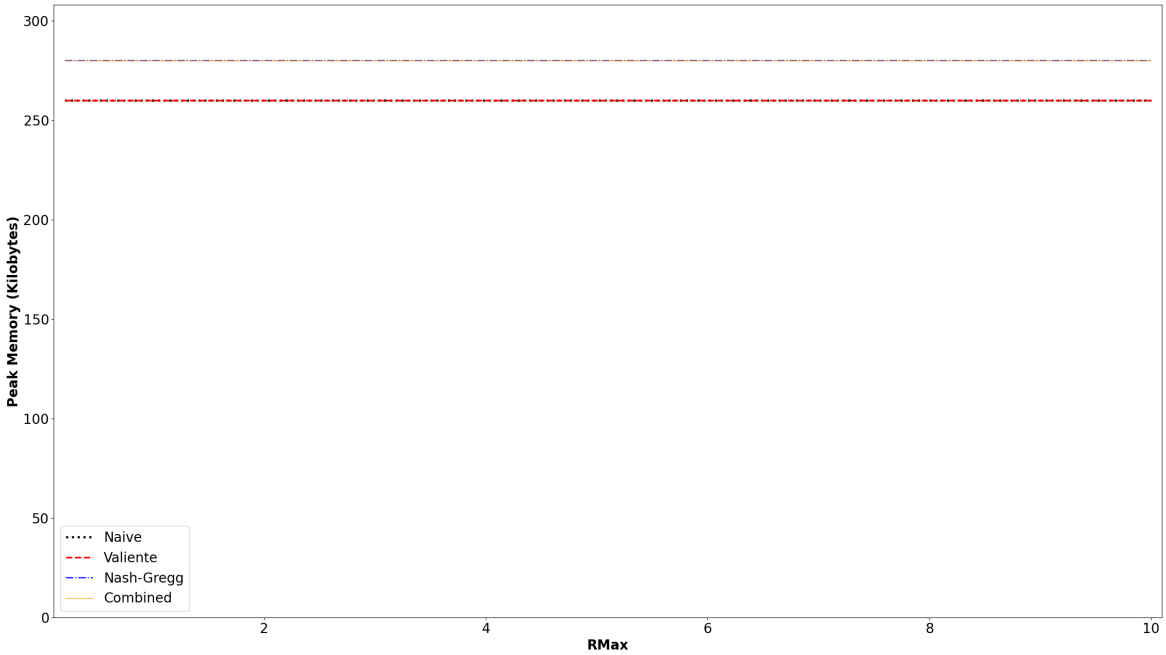
3.2.3 Iteration Count

I now discuss the inner loop iterations count of the algorithms. For the permutation intervals in figure 3.6a, we can see that the Naive approach performs the worst case as expected since it iterates over every possible pair of endpoints. Valiente's algorithm is a significant improvement to this since it reduces the number of intervals it needs to compute for by skipping intervals with already computed $CMIS$ values. Even though their execution time complexity suggested they would perform better than the former two algorithms, the Nash-Gregg algorithm and the combined algorithm performed even better than expected. The iteration count appears negligible compared to the naive approach and is 100 times less compared to Valiente's algorithm. The reason the time performance is not as drastically better is due to the time spent allocating and freeing the stack, which is done by creating it at the start of each call of update and freed when it returns to avoid a memory leak.

The number of iterations shown in figure 3.6b also reflects the time measurements I



(a)



(b)

Figure 3.5: The peak memory conumption of the distinct endpoint algorithms. In *(a)* permutation intervals are used the interval size is increased, and in *(b) the interval size is kept static at 5000 and RMax is increased*

discussed previously. Once again the Naive algorithm performs the worst, while Valiente's algorithm initially gets slower then becomes faster as $RMax$ is increased. This time it takes a much higher density for the Valiente's algorithm to perform better than Nash-Gregg and the combined algorithm since the expensive assignments are not accounted for in this measurement.

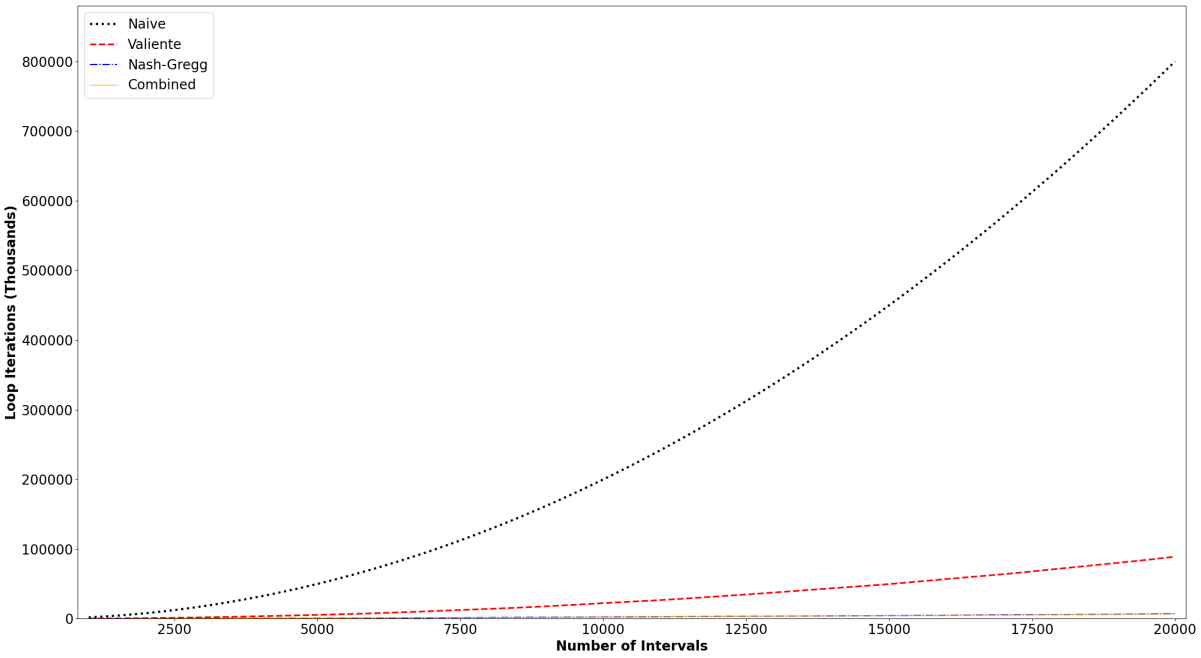
3.3 Results for Common Endpoint Algorithms

The following are the results of testing the common endpoint algorithms.

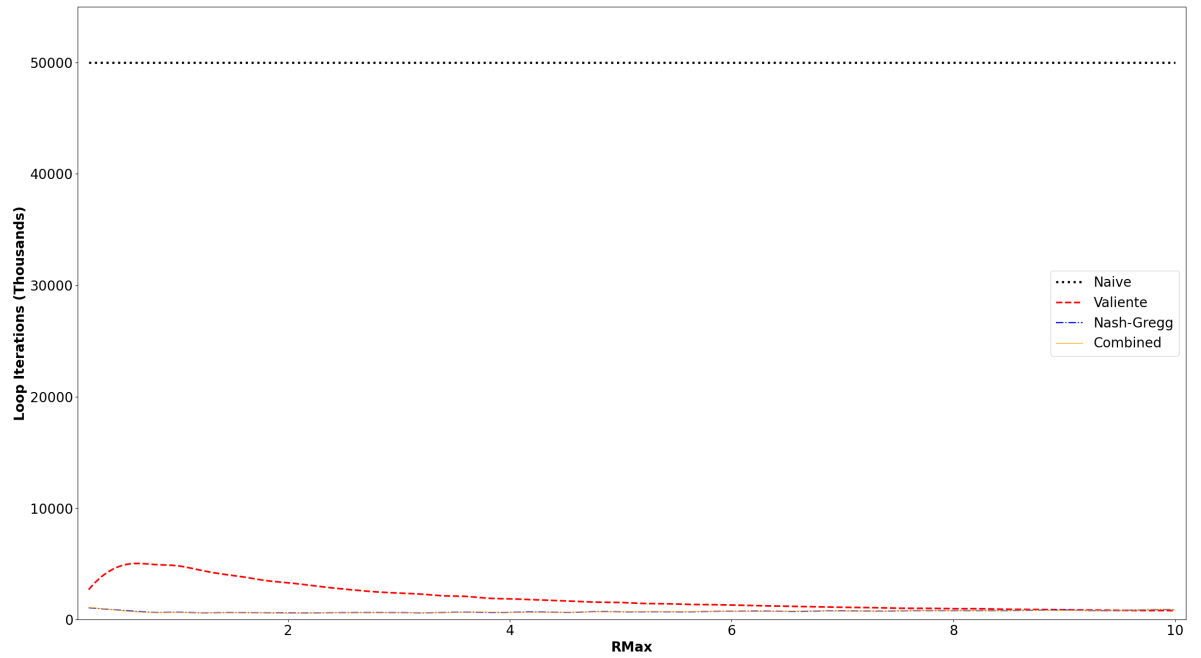
Figure 3.7 shows the execution time of the two common endpoint algorithms. As can be seen in figure 3.7a, the improved variant of the Bonsma-Breuer algorithm runs up to 1.5 times faster than the standard variant algorithm. We can also see from figure 3.7b that as the execution time increases as $PKeep$ increases, with the speed of the increase getting larger with $PKeep$. Since we've shown earlier that the increase in $PKeep$ results in an increase in the number of chords, this result shows that the algorithms' execution times are dependent on the number of intervals as described with its $O(mn)$ time complexity.

Figure 3.8 shows the peak memory consumption of the two common endpoint algorithms. Both algorithms have the same memory consumption. Unlike the execution time of the algorithm the increase in memory consumption is linear, however the increase is still faster the higher the value of $PKeep$. When increasing $PKeep$ the memory consumption does not change until $PKeep$ exceeds 96%. This occurs because the memory complexity is $O(max(n, m))$ and at higher values of $PKeep$ the number of intervals begins to exceed the number of endpoints.

Figure 3.9 shows how the number of loop iterations reflects the speed of the algorithm. The difference in performance between the Bonsma-Breuer algorithm and the improved variant is identical to the time performance. Figure 3.9a shows how the improved variant increases the number of loop iterations more slowly than the standard variant and figure 3.9b shows how increasing $PKeep$ increases the iteration count slowly at first then quickly



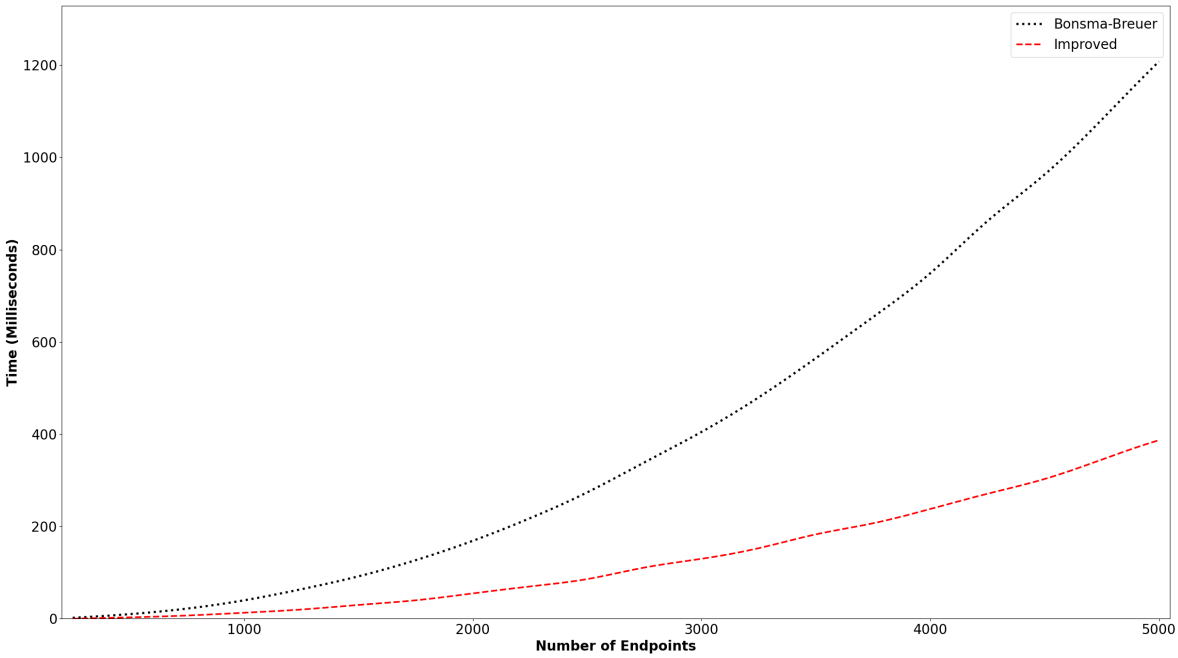
(a)



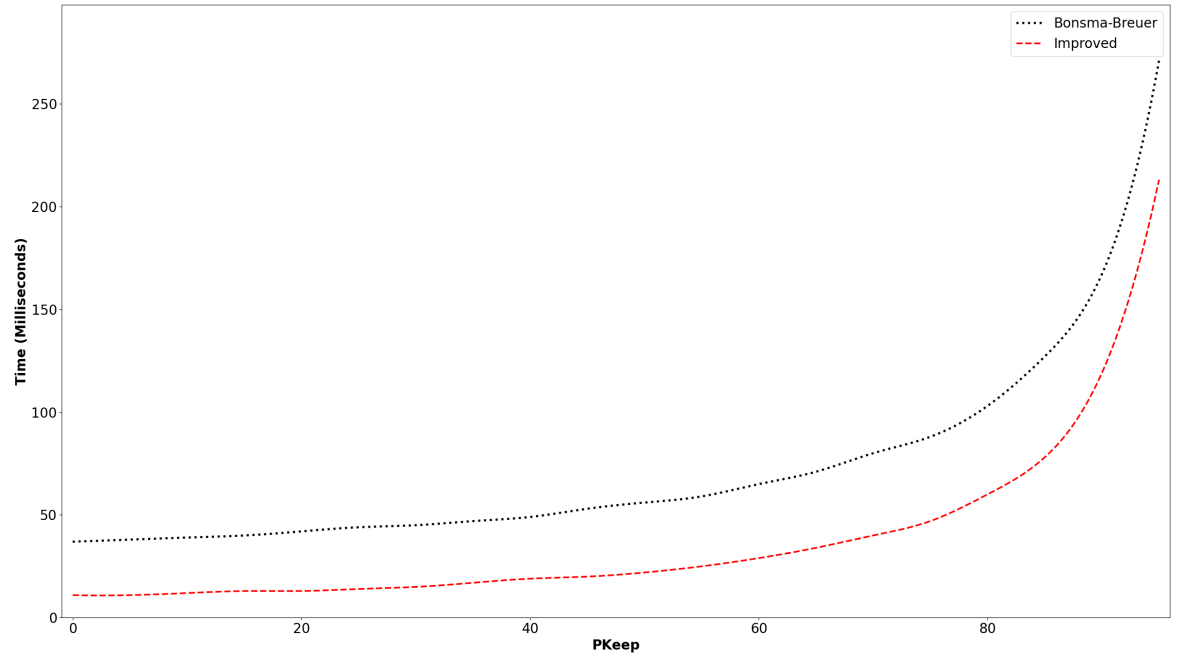
(b)

Figure 3.6: The number of iteration in the distinct endpoint algorithms. In *(a)* permutation intervals are used the interval size is increased, and in *(b) the interval size is kept static at 5000 and RMax is increased*

beginning at around $PKeep = 60\%$. The time complexity $O(mn)$ for the standard variant of Bonsma-Breuer is derived from this iteration count, as the increasing value of $PKeep$ increases the number of intervals, therefore increasing m. Meanwhile since the density and total length of intervals are also increasing with the increase in $PKeep$, L and C are also increasing which increases the iteration count of the improved variant due to its time complexity $O(L + C)$.

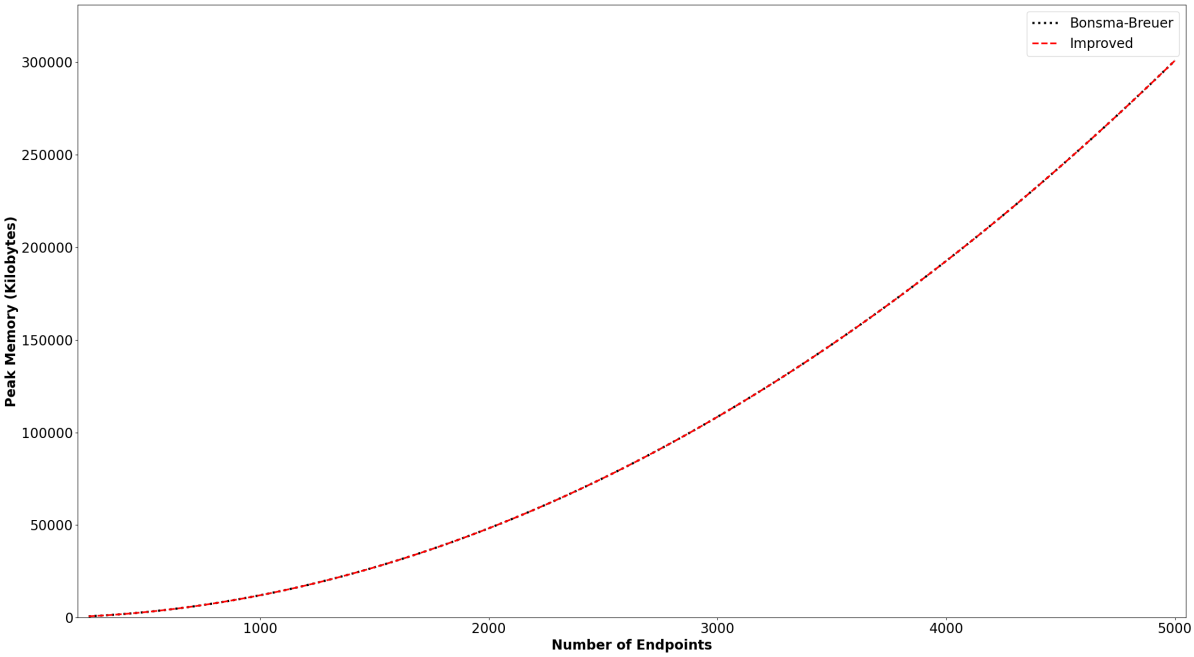


(a)

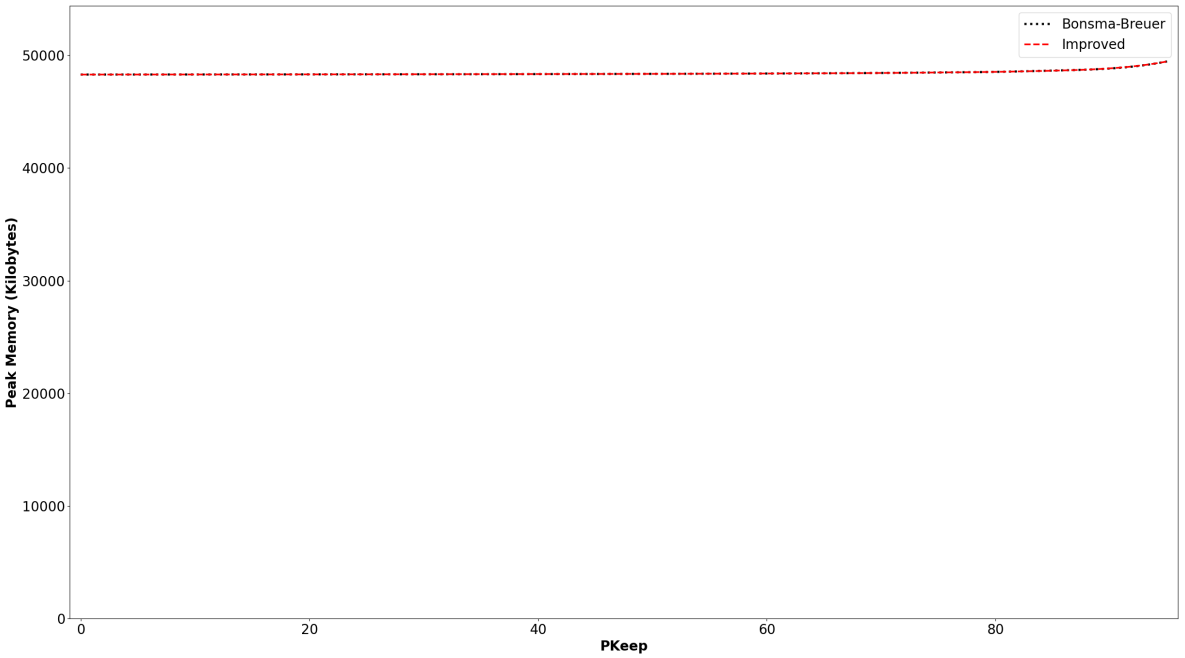


(b)

Figure 3.7: The execution time of the common endpoint algorithms. In *(a) PKeep* was kept static at 15 and the interval size was increased, and in *(b) the interval size is kept static at 1000 and Pkeep is increased*

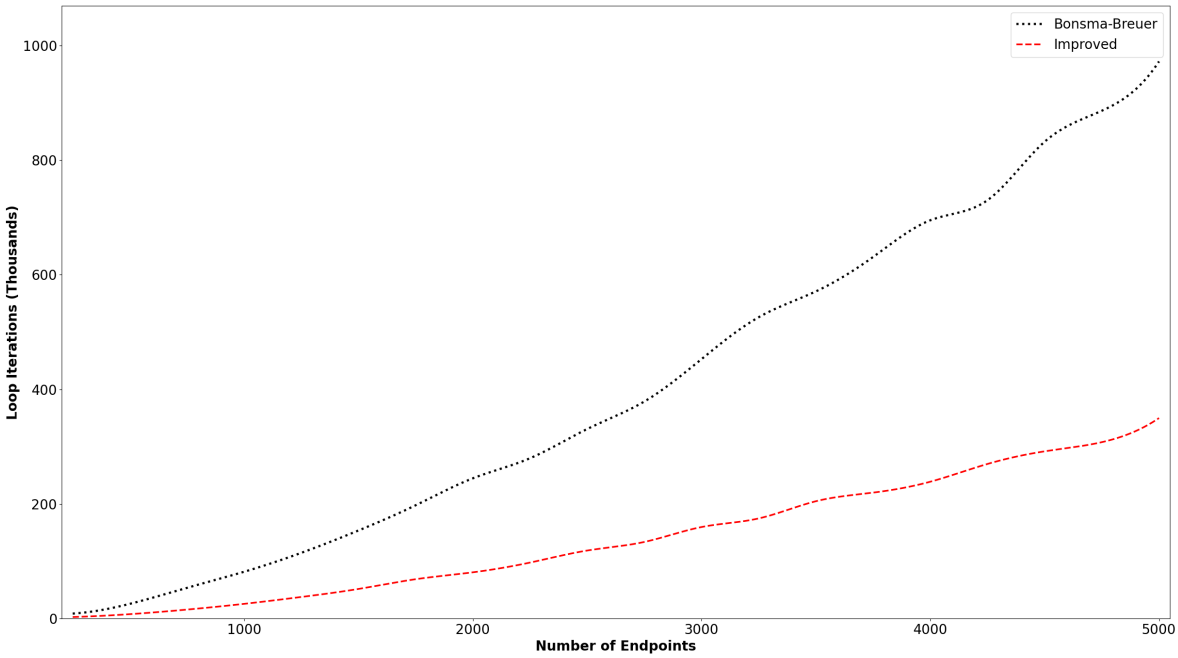


(a)

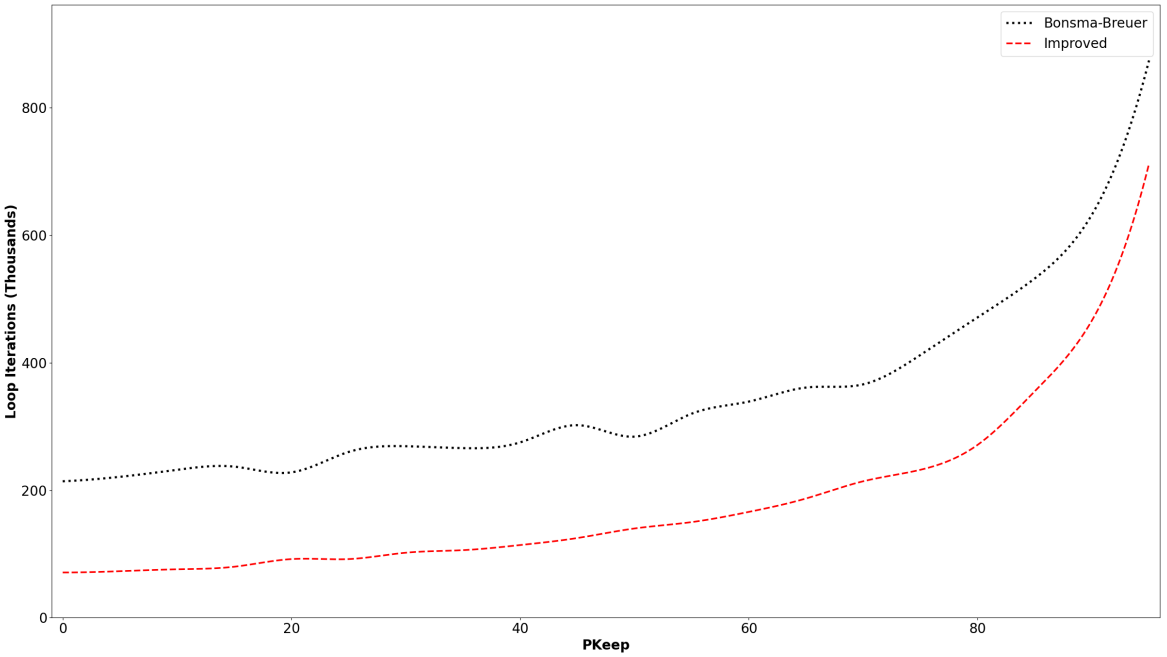


(b)

Figure 3.8: The peak memory consumption of the common endpoint algorithms. In *(a) PKeep* was kept static at 15 and the interval size was increased, and in *(b) the interval size is kept static at 1000 and Pkeep is increased*



(a)



(b)

Figure 3.9: The number of iteration in the common endpoint algorithms. In *(a) PKeep* was kept static at 15 and the interval size was increased, and in *(b) the interval size is kept static at 1000 and Pkeep is increased*

Chapter 4

Conclusions & Future Work

This paper has provided results for previously unimplemented state of the art algorithms for finding the cardinality of the maximum independent set of a circle graph. I first described some of the latest state of the art algorithms that exist both for the case where the chords in a circle have distinct vertices and in the case where these chords may share a vertex. I provided experimental results for Valiente's algorithm, the latest previously implemented algorithm, and for the Nash-Gregg algorithm which I have shown to be more efficient than Valiente's algorithm at low density values for Radial intervals. The combined variant of this algorithm showed little improvement on the base version since the cardinality is significantly lower than the density above a low value of $RMax$ for radial intervals, giving it no opportunity to adapt. I also presented a technique for generating intervals that may share endpoints and have shown the effect varying its primary parameter $PKeep$ has on the independence number, density and the number of intervals. I used this technique to evaluate the performance of the Bonsma-Breuer algorithm and compared it to an unpublished improved variant. This improved version was shown to be significantly better at all values of $PKeep$, with the improvement increasing as $PKeep$ increases. Although the execution times of both algorithms was significantly worse than the distinct endpoint algorithms. In the case of that set of algorithms, the results showed Valiente's algorithm with the improvements made by Nash et al. (2009) was shown to

be the best at high densities, while at lower densities the Nash-Gregg algorithm and its combined variant performed noticeably better.

Since radial intervals were not sufficient to account for the optimisation of the combined Nash-Gregg algorithm, future work could attempt to generate interval type that could parameterise the cardinality and density separately to allow for the algorithm's switch to occur at known conditions. Future work could also explore the performance of parallel algorithms such as the near maximum independent set computed in Takefuji et al. (1990). An analysis was done on the performance of such an algorithm on hyper-graphs by **?** for example, and a version adapted to circle graphs could produce better results than what was shown in this paper.

Bibliography

Apostolico, A., Atallah, M. J., and Hambrusch, S. E. (1992). New clique and independent set algorithms for circle graphs. *Discrete Applied Mathematics*, 36(1):1–24.

Bonsma, P. and Breuer, F. (2009). Counting hexagonal patches and independent sets in circle graphs.

Chang, R. C. and Lee, H. S. (1992). Finding a maximum set of independent chords in a circle. *Inf. Process. Lett.*, 41:99–102.

Gavril, F. (1973). Algorithms for a maximum clique and a maximum independent set of a circle graph. *Networks*, 3(3):261–273.

Gupta, U. I., Lee, D. T., and Leung, J. Y. (1982). Efficient algorithms for interval graphs and circular-arc graphs. *Networks*, 12(4):459–467.

Liu, R. and Ntafos, S. (1988). On decomposing polygons into uniformly monotone parts. *Information Processing Letters*, 27(2):85–89.

Nash, N. and Gregg, D. (2010). An output sensitive algorithm for computing a maximum independent set of a circle graph. *Inf. Process. Lett.*, 110(16):630–634.

Nash, N. and Gregg, D. (unpublished). New algorithms for maximum independent sets of circle graphs. unpublished.

Nash, N., Lelait, S., and Gregg, D. (2009). Efficiently implementing maximum independent set algorithms on circle graphs. *ACM J. Exp. Algorithmics*, 13.

Scheinerman, E. R. (1990). An evolution of interval graphs. *Discrete Mathematics*, 82(3):287–302.

Supowit, K. (1987). Finding a maximum planar subset of a set of nets in a channel. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 6(1):93–94.

Takefuji, Y., Chen, L.-L., Lee, K.-C., and Huffman, J. (1990). Parallel algorithms for finding a near-maximum independent set of a circle graph. *IEEE Transactions on Neural Networks*, 1(3):263–267.

Valiente, G. (2003). A new simple algorithm for the maximum-weight independent set problem on circle graphs. In *International Symposium on Algorithms and Computation*.