STUDY OF BOUNDS OF THE BIG CLIQUE BIG LINE CONJECTURE

A Project Report

Submitted by

ARNAB DAS
(18MA60R22)

Under the Supervision
of
Dr. Bodhayan Roy

in partial fulfillment for the award of the degree

of

MASTER OF TECHNOLOGY

in

COMPUTER SCIENCE AND DATA PROCESSING

at

DEPARTMENT OF MATHEMATICS
INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR
KHARAGPUR - 721302, INDIA
June 12, 2020

CERTIFICATE

This is to certify that M.Tech. project report entitled " Study of Bounds of the Big Clique Big Line Conjecture " is submitted by Arnab Das (18MA60R22) in partial fulfilment of the requirement for the award of Master of Technology in **Computer Science and Data Processing (CSDP)** during the academic session $2018 - 20$ is a record of bonafide work carried out by him at the Department of Mathematics, Indian Institute of Technology, Kharagpur, under my guidance and supervision. The report has fulfilled all the requirements as per regulations of this Institute and in my opinion, has reached the standard for submission.

Dr. Bodhayan Roy
Professor
Department of Mathematics
Indian Institute of Technology Kharagpur

DECLARATION

I hereby declare that the project work embodied in thesis entitled " Study of Bounds of the Big Clique Big Line Conjecture ", has been carried out under the supervision of Dr. Bodhayan Roy, Department of Mathematics, Indian Institute of Technology, Kharagpur, India and same report has not been submitted elsewhere for a degree. In keeping with the general practice of reporting scientific observations, due acknowledgements have been made. The author takes the sole responsibility for any error or ambiguity in facts or figures that may have crept in.

Arnab Das
Roll No : 18MA60R22
IIT Kharagpur, India
June 12, 2020

ACKNOWLEDGEMENTS

I sincerely express my deep sense of gratitude to Dr. Bodhayan Roy, Department of Mathematics, IIT Kharagpur, who has introduced me to the interesting field of " Study of Bounds of the Big Clique Big Line Conjecture " and has always been enthusiastic in enriching my knowledge in the same. I have enjoyed full freedom on working under his guidance which I will be remembering lifelong. I am grateful to Prof. Pawan Kumar, our faculty advisor Department of Mathematics, IIT Kharagpur, for providing me the opportunity to do my project under Dr. Bodhyan Roy.

Arnab Das
Roll No : 18MA60R22
IIT Kharagpur, India
June 12, 2020

ABSTRACT

The Big Clique Big Line conjecture states that for every l and k greater equals to 2, there exists a positive integer n such that every finite set of at least n points in a plane, Contains either l collinear points or k pairwise visible points. There been several works on this conjecture from time to time. There is an important constructive proof showing that the conjecture does not hold for infinite many point set. Since there is no derivation of this conjecture yet, so there are many studies in search of an algorithm that can determine such n from a given pair of l and k, or if not, maybe can provide us with some useful information about the bounds of n, there been works on it from both mathematics and computer science perspectives. This is a part of graph theory, where combinatorial geometry appears to be more helpful than co-ordinate geometry. There been several attempts based on these two approaches in past, trying to figure out simulations for higher values of n. now there is one thing to be noted, that for $l = 6$ and $k = 4$ finding an appropriate n is still an open problem. There is another result stating that for $k = 5$ and arbitrary l, $328l^2$ can be treated as one of the upper bound of n. but as we can see, that is too large to make any simulation for. But there is a recent proposition saying that there is a chance the bound maybe around some values of 13. So now there is a point trying to search through the simulations for those lower values of n, and to see, if it is possible to collect any new information about the bounds of n for these l , k values.

So for that purpose, in this project we concentrated on formulating an algorithm, which will verify whether a n satisfies the stated conjecture for a given pair of l and k or not. Which means this algorithm will try to generate every point visibility pattern of order n, and whenever there will be any indication of forming an invalid graph or a graph with more or equal collinearity or clique size, then the program will move to the next pattern, if every pattern gets visited this way, then the conjecture is justified once more and that n is proved to be one such n as mentioned, but if at least one pattern occurs which does not satisfy those, that means this n is not one such n, and hence we can try any other n above it. So this way, we can get some idea about the lower bound of n for a particular pair of k and l.

For this, we used two approaches, first one is co-ordinate geometry and the second one is combinatorial geometry. 1^{st} one went with brute force along with some necessary adjustments. But due to some of its serious drawbacks, we shifted to the second approach, also in which several obstacles were faced, some were known and

handled in textbook manner, but others were out of the blue, for which we had to study several pattern outputs to develop any suitable solution. We succeeded in most of those cases, which resulted us one such working algorithm, that we were hoping for. Needless to say, there still are various opportunities for improvement, besides some of those problems that we were not able to resolve are listed in future works, so there is that also.

But all over, it was quite a wholesome experience, working with the second approach and developing an algorithm on the way. So far, our final code did not surprise us with anymore odd results or unanticipated errors. All of it is running well as planned, producing the flow of output that it should be doing. Although this approach is more efficient than the previous one, but it is still not efficient enough, for some n values near 10, it runs for hours, or in some cases days, finding one single pattern. But in spite of that, we can still safely say that this algorithm gets the job done, which was the initial priority of us. So from that perspective, we think it's a win for this project.

Contents

1 INTRODUCTION

Big Clique Big Line Conjecture states that for every l and k (≥ 2) there exists a n such that every collection of n points consists of a clique of size k or a line of length l or in other words a collection l collinear points.

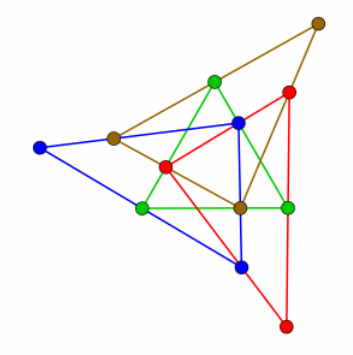

Figure 1: A (12, 3, 4) pattern.

As the figure shown here, this is a point visibility graph (definition is in the report) of order 12 with maximum clique size 4 and maximum collinearity also 3, so for any other l, k values above those, 12 is not an appropriate n for them. Similarly, another figure is shown below of order 12, maximum collinearity 3 and maximum clique size to be 4. We should mention that, these two figures are taken from a paper, whose reference is given at last.

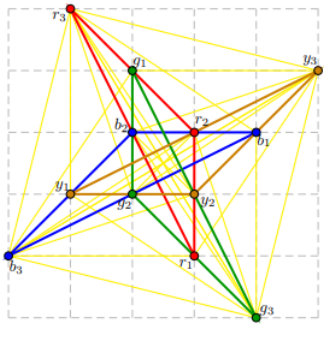

Figure 2: A (12, 3, 5) pattern.

It has been verified for several values of n, l, and k in some small range, but there is no generalized proof for it yet. Also there is another result stating that this

conjecture is not true for infinite point set, it has its constructive proof with it as well. But there is still a question that does this conjecture still hold for every finite point sets. But in this project we are not going that way.

Our main focus in this project is to find such n for a given pair of l and k, and try to figure out the lowest of all such n for a particular pair of l and k or any another relevant information related to it.

For the case of $k = 6$ and $l = \mathbf{4}$ it is still an open problem. For other lower values there are proofs or simulations already. There been some theoretical work over it in past, but determining an appropriate n was never possible, instead there were several studies over its bounds. The last closest result in past was a proof of the upper bound to be $\mathbf{328}^* l^2$, for this case $l = 4$ so the bound for this open problem is **5248**. But this bound value is too large to make any simulation in search of the patterns. But a recent study surprisingly proposed a result that claims the bound maybe around 13. So now there is a point to work on its simulation part which can check every possible pattern around this bound value and the objective of my project is to device an efficient algorithm and code for that.

Basically we have visited two approaches in my project, one is the basic classical approach, which pretty much follows the straight forward co-ordinate geometry concepts with some modification with the neighboring cases and some error correction along with appropriate scaling. Code on this approach works pretty well. But there are some massive dis-advantages of it as well. As in this approach there are some patterns which keep getting generated over and over, and for a sufficiently large n, it may hamper the runtime in a considerable amount. Other than that it still carries a huge runtime(exponential) along with it. To rectify this repeating pattern error we followed another approach known to be the combinatorial model. Most of our report and our final working code is based on this part.

As we told earlier, our goal is to develop an algorithm which can at least verify if a n is such n for a given l and k, as told in the conjecture. From there finding the least one won't be very hard. So keeping this in mind, we tried to generate every possible point visibility patterns of order n and check their collinearity and clique property until one bound violating pattern is intercepted. If so, then we can say that the n is not appropriate for this l and k values. Otherwise, if no such pattern is found, then conjecture wise that is a n for that l and k. So for that purpose, 1^{st} we tried to build every pattern with the help of combinatorial units of graph, known to be the " config ", a term that is shorted from the word " Configuration

". which is vastly described in our report, including the importance of config, the standard mathematical expression of it, the several conventional rules that it follows, or that we follow to use a config, along with its representation in code, and how the layered construction of it actually works, or why the construction is made to be layered, etc.basically what a config does is to store the adjacency properties of a point visibility pattern with respect to a particular vertex, so with the help of multiple such configs, one pattern can be formed easily, which is assured to be point visible, as the adjacency at every config is defined that way. Here is one more point to consider, which is the compatibility conditions between the configs, based on which the assembly of multiple configs to form a graphical pattern happens. Before this, there should come a discussion about the process to generate every possible configs for every possible roots and line numbers. In this part, we talked about a permutation based partition algorithm known to be the divider algorithm, in which we were curving a huge permuted data set to find some appropriate match from which, config can be formed. But also at the same time, all the rest of the patterns from which the valid ones are extracted are going to waste. This garbage patterns and the runtime that goes in generating them became a real problem after $n = 7$. So to make it work for the higher values of n, we 1^{st} tried to add some adjustments to reduce those invalid patterns, or to discard them completely if possible. But on the way, we were able to come up with a new algorithm involving time-memory tradeoff, which was a pretty good answer to our previous problem. Here we brought a different approach to generate the partitions from the one in the divider algorithm. There is a part in the discussion of the divider algorithm wishing for some convenient way to just generate those valid patterns with no other extra patterns along with them, and in this section, that wish came true. Here all the configs of all roots and possible lines are getting generated in the 1^{st} part of the program, and after that the process of gathering the configs based on their respective compatibility, form patterns from them, checking the bounds and all other inconveniences take place. This approach may seem efficient for now, and maybe for many average cases, but there are several cases in which this approach may not prove its worth in efficiency. But one should not get it the wrong way, this is still a valuable approach, maybe the best one, that we have yet. But the wrong thing is to isolate this process from the config gathering method, which are discussed in as much details as possible with some ideas of improvement in our report afterwards.

After this we came to the part of gathering the configs with their compatible

companions and making a point visibility graphical pattern out of a complete branch of compatible configs. There is nothing to modify in the compatibility testing, as it basically follows the normal concept as defined in the compatibility. So after covering this briefly with a few example. We went to the part that describes the gathering process, or the DFS traversal with the configs as its nodes. So far, this is the heart of our program, every single module of this program plays an important role here. Now there is also a small explanation about why the DFS is chosen for this job, why not any other Cartesian product based algorithm. This is a question which can surely appear, but it's not that important, so returning to the main topic 1st we implemented the DFS, traversing through every branch of configs up to its leaf nodes. But later on that felt unnecessary also, there is a reasonable justification for it too. So instead of letting every branch to fill themselves completely by traversing up to the end, we put a control in the growth of the branches, which is also referred as the intermediate checking in the report. That way we prevented the growth of the traversal in any irrelevant direction which is determined by the given bounds of clique size and collinearity and also another irrelevance that we discovered, named as the "Contradictory pattern" in our report. Details on it is there also, with the description of a method to detect those patterns when a branch of config is getting filled. We noticed only a few attributes in a branch of config that can cause such contradiction, and needless to say, we implemented only those in this method as well. But with further knowledge on the patterns, this method can be perfected by adding more attributes to it, which can only be possible after analyzing a decent amount of pattern output. So there is still room for improvement in this portion of our program. After this, another worthy topic to mention is the rectified concept about collinearity derived from a branch of configs. There are several scopes for overlapping and coupling between the unit lines in the configs, which we had to consider after having a few tampered results due to the absence of it. So after this modification, in the output we are being provided with the collinear components of the graphical pattern formed out of the branch, which is proved to be very much useful in drawing the pattern. In future we have a plan for implementing a graphics interface in our program that can draw each pattern using these results. But we don't need to worry about that right now, because it's merely a part of the representation of the output that we already possess, nothing more. There is one more thing to mention here, the contradictory pattern detecting algorithm that we discussed earlier, that algorithm also uses the collinear components of a graphical pattern as one of its argument. So

keeping this part of our program clean and sound is very essential because one wrong information in this part can affect the entire output in a geometric progression.

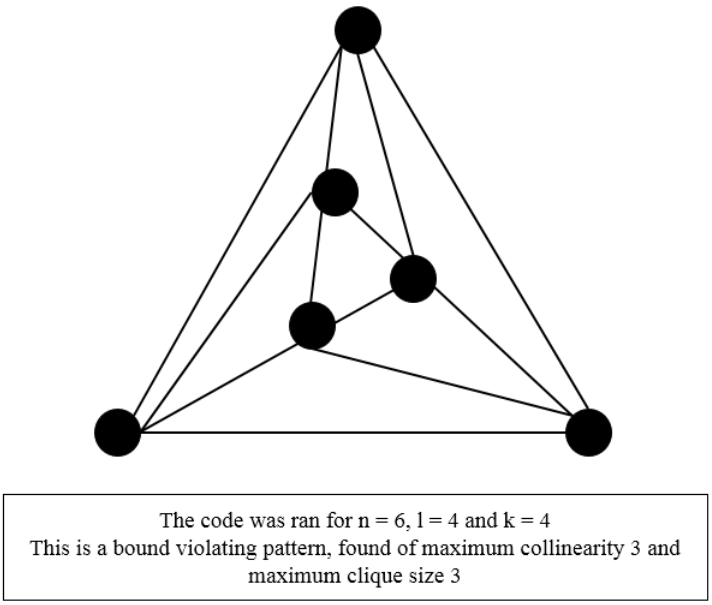

Figure 3: A (6, 3, 3) pattern output.

So this is pretty much everything that we have in our project report. There are several plans for future works and ideas for implementing them, mentioned in several parts of our report, whenever felt necessary, some of those are pretty ambitious, while others are somehow doable from where we are standing on our project right now. Nonetheless we enlisted every ideas and plans that we have, in the future work portion of the report.

Now about the output, we were able to run our program up to $n = 7$, during the time of completion of this project report. We ran it for several values of l, k, and as a result, several bound violating patterns were found, some of which are shown in this part of our project. In future we are hoping to find more of such interesting patterns for several values of n, l, k parameters.

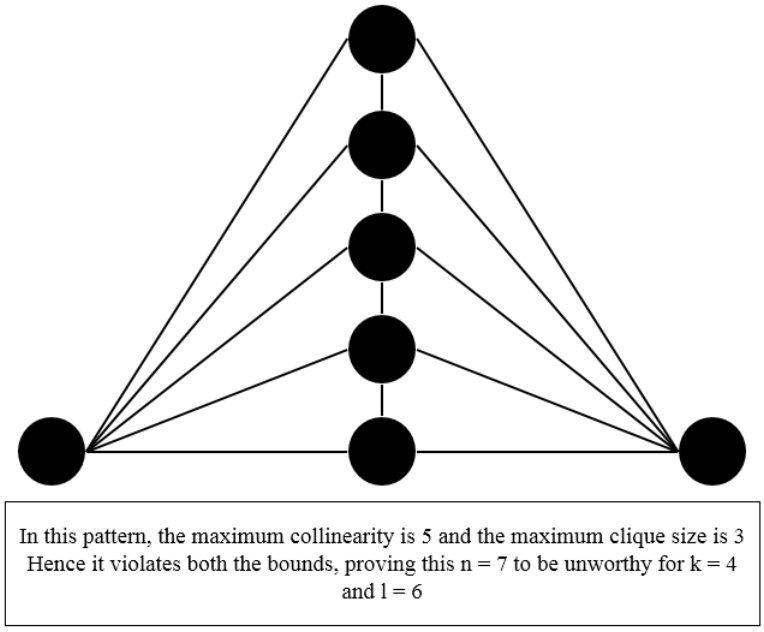

Figure 4: A (7, 5, 3) pattern output.

2 Detailed Description

2.1 Main Objective

As we described earlier, we are trying to find the lowest n for a certain pair of l and k, according to the conjecture whose existence is already assured, so if anything, it is a work having the conjecture as its base, so any result found through this project will support the conjecture more.

Definition : As a main keyword, we should define the concept of point visibility at first. Like the name suggests, it simply means the case, when in a graph, all of its vertices are visible with respect to the edges in that graph. So in formal sense, a graph is known to be point visible if no straight line connecting two vertices of that graph, having edge between them, passes through any other vertex of it. If so, then the points are said to be point visible and only then, the edges between them are granted. So edges are assigned here based on these visibility criteria.

2.2 Idea for Generalization

Now if a graph is given, it will not be tough to determine whether it is point visible or not. To apply brute force, we can take all the subsets of the set of vertices of the graph of order at least 3 and check for their collinearity. Since geometrical collinearity can be checked by equating the slopes of any arbitrarily chosen pair of points in the set, if all the calculated slopes for every possible pairs are equal then the whole subset in collinear. But a more convenient way is to calculate it in a chain, 1^{st} consider any 3 points in the subset and check for their collinearity, if it failed, then collinearity of the entire set is denied, but if not, then consider a new point, and make a triplet with this and any two points from the previous choice. If they failed collinearity then also return false, otherwise proceed further with any new points in the set unless all points are covered. Instead of $n!$ runtime exhaustive brute force, this approach even in its worst case, terminates in n-linear time, which is much efficient.

This approach can also be followed in finding the collinear components in a graph. In the previous description, after every step to include a new vertex , we should try every remained vertices for it, and if any one of then becomes collinear with the previous set then include it in the set, and if no more vertex can be found to satisfy such collinearity, then we should stop there and declare the set as one of the collinear component of the graph and discard them from the total set of vertices, and again restart the process with the remained vertices until there are no more vertices to restart the process with, and finally, all the resultant subsets of vertices we have by then are all the disjoint collinear components of the graphs. And eventually the maximum order among all those subsets is the maximum collinearity of the graph.

But even if this algorithm is theoretically sound, but there is one small drawback in its execution. Whether we calculate the slopes or put a point in the equation of a straight line joining two points, there will always be some error for float value points, and for small diagrams where some points are considerably close to each other can cause faulty outputs, so to overcome this, we can scale the whole diagram and set a small error measure which we can treat as zero for this case, later on this will also be implemented in the 1^{st} model.

One little point is important though, it can confuse someone that maybe the point visibility graphs are in some way those graphs whose every collinearity component is at most of order 2, but this is not correct at all. In a point visibility graph, collinearity components and their order can be anything, ofcourse lesser than or equals to the order of the graph itself, as long as the definition property of point visibility graph is

not broken. Such as, suppose we consider a straight line with n points on it, indexed from 1 to n. Now there is a valid question, is this line point visible or not. The answer is, it depends on how one defines this construction. Now if we just permit the edges only between any two consecutive vertices and no other, as in, if the edge set is: $\{(1,2),(2,3),(3,4),\ldots,(n-1,n)\}$. Then we see there is no breaking of the definition and it is fairly a point visible graph. But if we let any more edges except $(1,n)$ in this edge set, then there will always be some vertex lying on that edge except the terminal ones, and hence the definition is broken and this graph is failed to be point visible.

2.3 Challenge

So we just saw the preliminary part of our project, now since the problem is to find the existence of some pattern which satisfy some particular specifications, our goal will be to search through all the possible patterns of a given degree of vertex set to find the desired one. And that being said, the main challenge that arises is to generate all possible point visibility graphs of a given order.

Now firstly, there is one more thing before we get to this part, which is, if we are given with a set of co-ordinate points, what is the way to form a point visibility graph out of them. This process is also fairly straight forward. We need to check for every pair of vertices that if the straight line connecting them is passing through any other vertices or not, if does, then the edge between those two vertices are cancelled, and if not then permitted, in this way, we can come up with a unique point visibility graph pattern with a given set of co-ordinate vertices.

Now let us get to the main topic, which was to form every possible point visibility graphical pattern of a given order. For this we can take a general property of vertices in graph that a simple graph with n vertices, any vertex in that graph can be of degree atmost $(n-1)$. So for a point visibility graph pattern, there can be two extreme cases collinearity wise, between which any of the other cases will belong.

One is, where the vertex under consideration is of degree one, and every other vertex is on the same ray or straight line emitting from that vertex. This pattern we actually discussed before to clarify the definition of the point visibility graph. And the another pattern is, where the degree of the vertex of consideration is $(n-1)$, that means this vertex has $(n-1)$ emitting rays from it, and every other vertex are belonging to each one of them.

Now out of those vertex arrangement, we can form the point visibility graph easily

as we described previously, and we are extending this notion to generalize the pattern formation.Suppose one vertex is fixed. Which is our vertex of consideration or root, if we may. Since the order of the graph is n, so this vertex can have at most $(n-1)$ rays emitting from it, and also from the 1st pattern we see that each emitting ray can consist at most $(n-1)$ points other than the root vertex, so where fixing the root vertex, we got $(n-1) \times (n-1)$ variable point positions, among which any (n-1) positions can be chosen and form a point visibility pattern out of them. So the possible positions of the rest of the vertices will be all the subsets of cardinality $(n-1)$ of the $(n-1) \times (n-1)$ positions. As shown in figure.

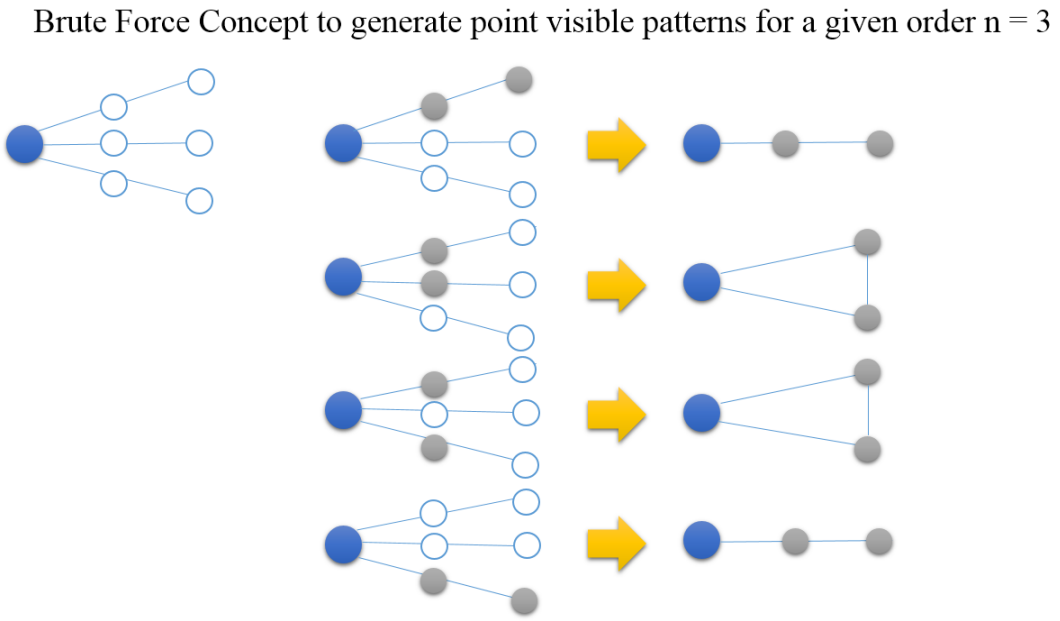

Figure 5: Concept of Brute Force Approach.

So to implement this approach, we formed a geometrical construction, which pretty much performs what this theory says.

Since the code (second code) is also attached with the project, so if one goes through it, one can see that, the same idea of the above has been followed and to avert the point closing error problem, the scaling concept is also tapped with the code

2.4 Process Description: First Approach

For the give order n of the set of vertices. First we set our fixed vertex or the root, as the point $(n-1, 0)$. Later on to generate all the possible positions of the rest of the vertices uniformly, we took all the Straight lines joining root and $(0, i)$ and $(0, -i)$ respectively for i upto the floor value of half of n. depending on whether n is even or odd, we include the line joining root and $(0, 0)$ which is basically the x-axis in our line sets. Now these are all the possible rays that can emit from the root. Now we need some construction to plot the vertex positions over those rays. For doing so, we took the help of all those parallel lines joining $\{(i, 0), (i, 1)\}$ respectively for i varying from 0 to $(n-2)$. The set of all the points of intersection is the required set of all the positions of the remained $(n-1)$ vertices of the graph. Now we just have to calculate every subset of size $(n-1)$ from those positions are appending the root with each of them, we have to put that set of vertices for further processing to make a point visibility graph out of them and later on check for the collinearity property and the cliques and the length of maximum clique and so on.

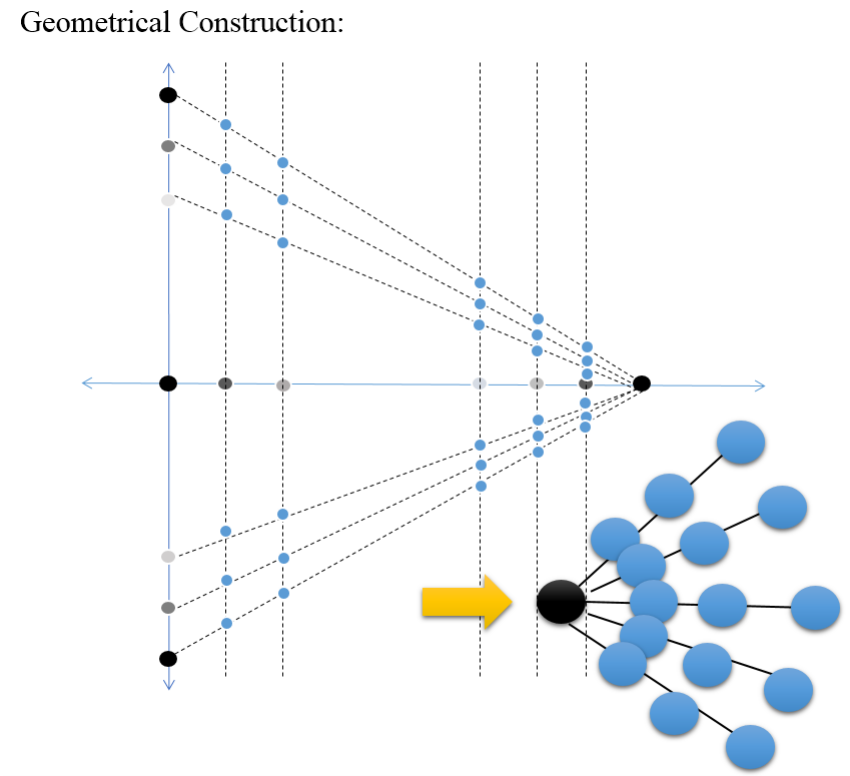

Figure 6: Ghraphical Construction to Impliment Brute Force.

On a short note, to find all the subsets of a set of given length, it is always a convenient way to apply to dynamic programming algorithm for it, given that it is way less expensive runtime wise. But here is a small problem applying that, not in this model, but in some later model, there will be some recursion call in the main algorithm and since the codes are considerably lengthy, so after a while it becomes a tough job to track where the function calls are going, if other recursions are being called in the main code recursion, then most of the times, we end up being trapped in a loop between two recursions. So to avoid that, we here used the one less efficient but more effective with respect to this matter, and also such an algorithm which can also be used as iterator directly.

This method is nothing but just tracing the binary pattern of every numbers whose binary representation lengths are at most the given size of the subset we intend to find. To describe it in a more elaborate way, let us suppose that we have a set of size n and we are trying to find all of its subsets or in other hand the power set of the set. Now we see that the number $N = 2n–1$ is the last positive integer whose length of the binary representation is n. so we just run a loop through every number from 1 to N, find every binary representations of those number and map the set with respect to those representations, such that in a representation where we read zero, we discard that element of the set, and if we find then we include that indexed element, and proceeding in such way, for every binary representation, we can come up with distinct subsets of the set. Now in case of a given length of subset, we run the same algorithm with a small modification which is, just letting those numbers of the loop in, whose binary representations having the parity equals to the given length of the subset.

Now getting back to the topic, this construction clearly generates every possible point visibility pattern of a given order, but there are two points to consider, one is, most of the possible positions of the vertices are float points so if we round up those points up to some finite length in decimal, there always going to be some round up error.So as we discussed earlier, we have to fix some error measure of the whole system and implement appropriate scaling with the construction.

That's why instead of fixing the root at a certain point, we made it variable and depending on the given value of order. And also as we saw that every other possible positions of the whole constructions are also n dependent, so larger the value of n, more the points will shift apart from each other, and there will be very less chance of them to be mistreated for overlapping due to their rounding error.

Another small modification could be done, which is making the error measure also dependent on n, of course in some inversely proportional way so that whenever the value of n will increase, along with it, the error measure will decrease accordingly so that there will be no miscalculations even if we enter a huge value for n, and that's how scaling is embedded with is construction.

Now another question arises, which is, is that really necessary for the positions of those points to be placed in an equidistance pattern on the rays emitting from the root. What if we place those points separately but randomly on the straight lines, will the results be changed? Actually it will not, if we see closely, we can see that even if the points are placed in that way, of course some collinearity will be changed, so in uniform construction one pattern that was point visible here can turn out not be that and also vice versa. But the whole set of the output of patterns will remain same up to isomorphism. So it does not really matter in which way or in which order we place those points, as long as the basic notion of the construction is followed, output will not be tampered. The random positioning of the points through coding is also possible but We followed the uniform construction because, from the coding perspective it is easier to implement and will work the same. More or less, this algorithm provides us with correct results and its advantage is, since it's always generating co-ordinate points and forming the point visible pattern out of those, so if we want we can plot out all those patterns, so there is a visual advantage of this approach. But after everything one cannot stick to this because of a huge drawback of it. We are now going to discuss that.

Suppose our general construction where n is the given order of the graph. So from the root there will be $(n-1)$ rays and $(n-1)$ vertices on each of those rays. Now since we are only concerned about the geometrical pattern, so once we find one pattern we don't need any of its copy or isomorphic duplicate in the output, which is exactly what wrong with this approach. Take this example, as we know one obvious pattern is the straight line having n points on it, so at a point of the power set calculation, one positions subset will be chosen, where the algorithm chooses only one ray from the root with every points over it, which gives us this construction. Since there are $(n-1)$ distinct rays, so finding other $(n-2)$ repetition of the same pattern will be completely unnecessary. And what we explained here is just the tip of the iceberg, as not only there are repetition of patterns but also a huge amount of repetitions of isomorphic copies as well. Some of which are shown in the below figure. Which is as we can see, a tremendous waste of both runtime and computational effort.

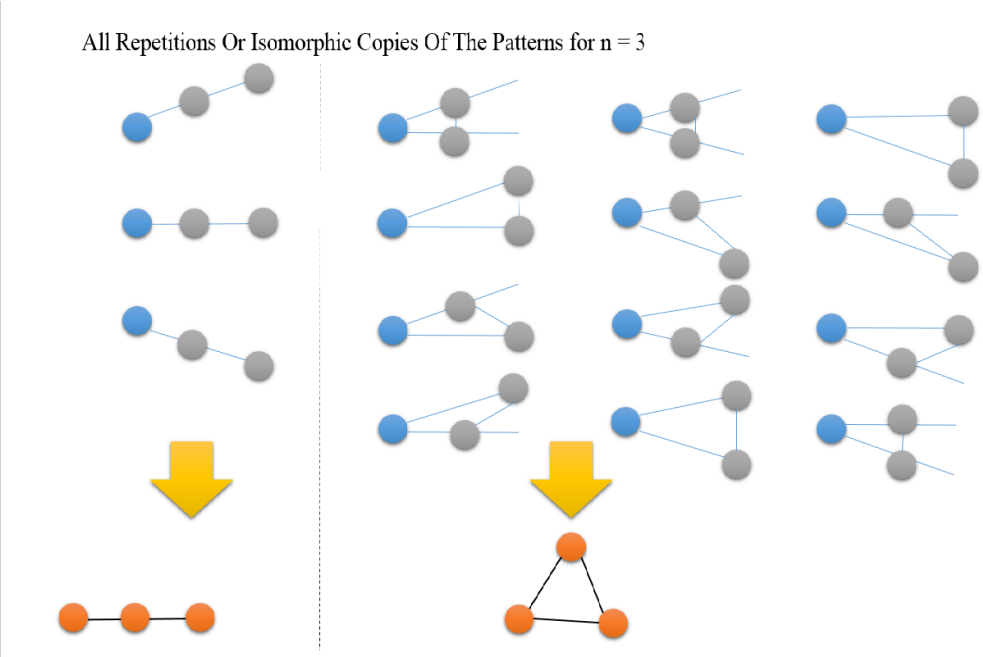

Figure 7: Repetitions in output for n = 3.

There was only one way to make it work, which was to passing this output data through another algorithm which will classify those data based on their isomorphic property. There are some classification algorithms which can serve this purpose with high accuracy, but once a heavy brute force algorithm and then another classification algorithm over a huge amount of data makes it very rich. We could implement it taking the degree property as the key attribute of the classification and maybe something more. But instead of exploring other approaches, it is not always helpful to improvise on the brute force one, since maybe we have found a way to precise the output, but after all the 1st step of it is still brute force, so there is no point dwelling on it further. If the rest of the approaches fail to produce some fruitful results, only then we can look into it further.

2.5 Second Approach: Computational Geometry

Our first approach was geometrical, but keeping in mind that we only need to study the patterns of those graphs, we don't actually need to visualize them or plot them in any way, as long as any approach provide us the interface to generate a valid adjacency list or adjacency matrix of the graph, it will be accepted. That's why instead of

straight forward geometrical approach, this time we tried the combinatorial approach to serve our purpose.

This concept is a bit complex to explain, since here in this algorithm neither we are generating the whole graph at once, nor we are trying to find its specific set of vertices like the previous one. Instead of doing all that, we are here actually just trying to figure out all the possible patterns of adjacency, but that also does not happen at once and this is where it gets interesting. Here we introduced a unit of graph or pattern. With the assembly of one or more of these will actually construct our required pattern, let's call this unit of adjacency as " Config ". The name is actually an abbreviation from the word " Configuration ", since the name will come so many times in our project so we needed a shorter name for it, which will also be uniquely recognizable from any other common words in English, hence the naming.

2.6 What is Config?

Config is actually a unit or an atomic part of a graph, which consists of the part of the adjacency property with respect to a specific vertex in a graph.

Maybe the definition is not all that sufficient to understand what this config really is. Perhaps it will get more clear once we get into the construction of it, and explain the importance of the construction.

2.7 Construction of a config

If the total number of vertices is given. Let's say its n, then every possible config of it is of order n. every config consists a root vertex and all the other vertices are incident on the several rays coming out of that root vertex. Now a config only holds the adjacency property of the root vertex with respect to the remaining vertices of it and by adjacency property, it means between which vertex and the root will be an edge and where will not. But one thing we have to keep in mind that we do not have any clue about the adjacency between any two vertices other than the root. So from this notion, it is obvious that if we need to construct any graph but need the adjacency properties with respect to every vertex for the rest of the vertices, or in other words, to construct a graph we need a config of each vertex as root, and taking the union of all of them will get us a required graphical pattern.

But there is a small correction in our concept, which is, if we have n vertices, we don't need n configs, one for every vertex all together, we need $(n-1)$ configs, one for

each vertices and the union of those $(n-1)$ configs will automatically cover the total adjacency of the graph, which makes the requirement of any more configs obsolete.

Now there are some conventions in the construction of a config. We will go through them along with the explanation one by one.

1. 1^{st} since the order of the graph will be given, we have to take that all the vertices are indexed from 0 to $(n-1)$, because this order will matter in many ways, later we will see.
2. Only for the configs, where root is zero. On the rays emitting from the root in the config, vertices that are situated will always be in increasing order, from root to the end of the rays
3. Only for the configs, where root is zero. The vertices will be in increasing order index wise, from top ray to the bottom one.
4. For any other configs, there are no rules for vertex placement of the rays from the root
5. No ray can be empty, if there is no vertex on a ray, that ray will not exist for the config.

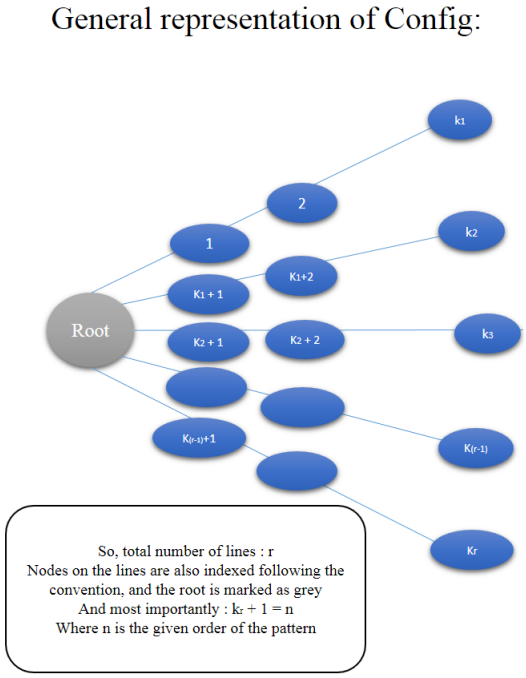

Figure 8: General Construction of a Config.

Now to explain those four convention rules, we have to keep in mind that we are only

interested in the pattern of the graphs, which does not include the indexing of the vertices. So if we don't follow any specific convention throughout our whole process. Then same pattern will generate more than one times just indexes altered. Which obviously we don't want, because to get rid of that repetition problem of the previous model, we came to this one in the 1^{st} place.

Now it is not all mandatory that the order has to be increasing or the vertices have to be indexed from 0 to $(n-1)$, and not in any other way. We have to understand that this is just a combinatorial pattern which can separate only one unique pattern and discard all the others among all the same but differently indexed patterns. So inclusion order can be decreasing instead of increasing or the vertices can be indexed in any other convenient ways, as long as the above explained purpose of the convention is served and the convention is followed strictly in all circumstances, we don't have any problem. In this theory however, we are going with the four conventions that we mentioned earlier.

2.8 Adjacency property that a Config provides

We told before that a config does not provide any information about the adjacency of two vertices belonging to two different rays from the root. Besides we also discussed about in which definition of a straight line having some points on it becomes point visible, here the adjacencies are defined mainly following that concept. Just as before, here are also some rules about the adjacency property interpreted from a config

1. Root of the config and the 1^{st} vertex on a ray, counting from the root to the end of the ray, are always adjacent.
2. Any two consecutive vertices on a ray from root are always adjacent.
3. There are no adjacencies other than those above tw0 in any ray.
4. There is no information about the adjacency between any two vertices belonging to two distinct rays. That are to be settled with the help of other configs.

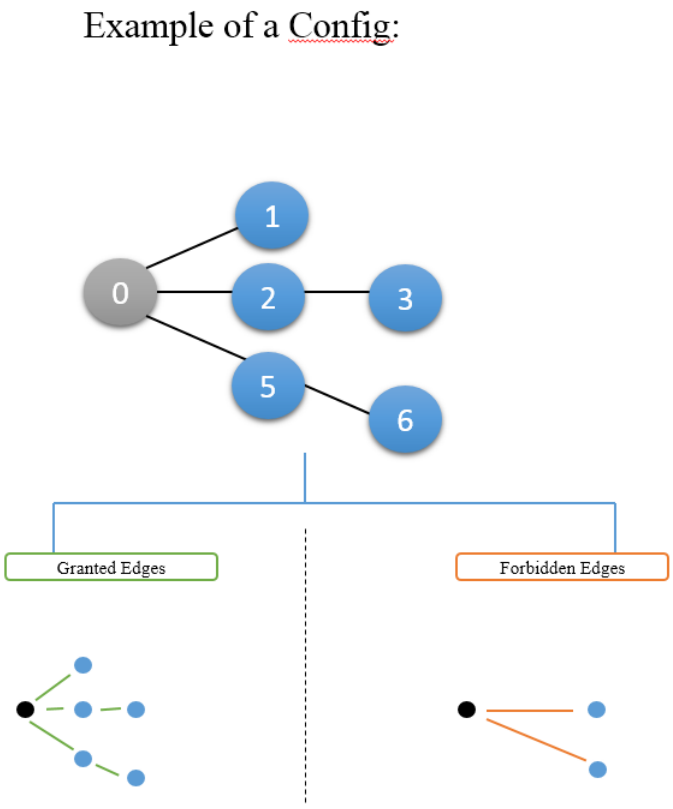

Figure 9: Interpretation of a Config.

2.9 Coding Representation of a Config

For the coding purpose of our project, first we used C language, but more the code grows in length, harder it becomes to handle all the separate 0bjects of the code, as config or graph etc. so we thought it will become much easier, if we use some language, through which we can treat each of those as separate objects with the customized properties and functions of their own. And hence we used C++ instead of C, as for its object orientation advantages. besides it has its own template libraries, which provides some much convenient data types, which are so much easier to use, than implement all of those out of array in C.

So that being said, clearly in our code, we have represented our config as an object. In it, we put five separate attributes, among which two are primary, which means, those two the 1st to initialize either through the constructor, or the classical setter function. In other words, these two properties define the shell of a config class.

These two properties are the root and the number of lines emitting from the root. Of course both are integer valued, we could use exception handling to make sure that the constructor or the setter function will never accept any negative integer or the number of lines from the root to be zero as an argument, but since our algorithm is bound in such way that it will never generate any argument of such values, so it was not felt necessary to implement this part.

The secondary attribute is the set of lines, which is represented by an array of vector (one of the STL data type in C++), where each vector represents each line, here we took vector instead of an array, because insertion or deletion in or from some intermediate index is much easier in vector than array. By the way, one little thing is that, when we initialize the primary attributes, there is only the reference of the object of the lines is present, but as soon as the value of the number of lines is fixed, by using it, the reference is initialized with an instance of order, which is the given number of lines. So we can say that the line set container is formed in the 1^{st} initialization but the set is filled in the second step. Filling this line set is easy as well, we just insert or push the index of the vertices to the corresponding indexed lines, which is here the vectors.

Now at first, we just kept our config class up to this, but later we had to introduce two more attributes. Because the previous construction was sufficient too, but in many places of our project, we need the adjacency information of a config, which we discussed earlier. So instead of generating those information by a member function over and over, we decided to generate it once and store it in some attributes of the config instead of exhausting the algorithm. And that's why, we set two more attributes granted and forbidden, which are basically the sets of the granted and the forbidden edges of a config. Following the theory, we here also defined them as the set of pair of integer values in the code (set and pair both are the STLs of C++), we choose here set instead of any other STL data types because, even if there are repetitions in the edge generation process, then it will be rectified, because set only keeps one copy of each distinct element and does not allow any repetition.

But there is a valid query, which is, why the granted and the forbidden sets of edges are filled at the 3^{rd} stage, why not with the introduction of every new vertex in any line, we store the corresponding granted and forbidden edges in the sets, but there is a small problem doing so, each time we try to determine the adjacency properties with respect to a vertex of a config, 1^{st} we have to determine if it's the 1^{st} vertex of that line or not, and later on, starting from the root, we have to traverse the whole

line and have to generate the adjacencies following the conventional rules we discussed earlier. But this way, every time a vertex in inserted, the line is being traversed all over again, so by the A.P series, we can see its runtime becomes n square. But if we determine all the adjacencies at once after all the lines are filled, then every line will be traversed only once, so the runtime becomes linear. So for this reason, we store all the adjacencies with a member function of the class after filling up all the lines, hence the 3^{rd} step becomes relevant. After the 3^{rd} step the config becomes complete. But there is a catch, since only filling up the lines does not complete the config, so if somehow one forgets to call the 3^{rd} step initialization at the end, then it's going to show null pointer error at the time of compatibility check, which is something we will be introduced to the upcoming chapters.

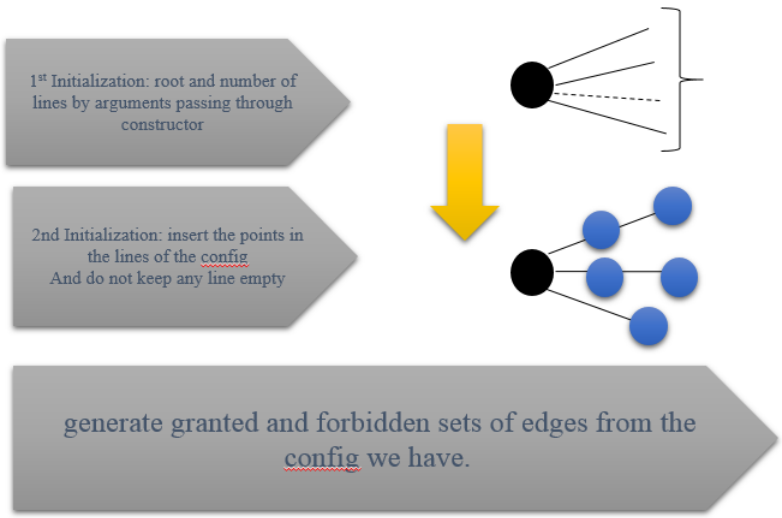

Figure 10: Formation of Config Class in Coding.

2.10 Generating config by coding: Discussion of the Corresponding Algorithm

We covered everything about the definition, properties and the representation of a config in coding. But now, it comes to the question, that how to generate every possible config of a given order and a given root. So except the root, with the rest of the vertices, sorted in increasing order or not, based on the root, we have to partition them from one partition up to (n-1) partition in such a way, that no partition is

left empty. So it goes without saying that each partition represents each line, and to follow the general convention we have to partition the remained set of vertices in sorted order.

So before any other customization, 1st we have t0 implement the separation algorithm, and later on we will come to the several customizations that we appended with this algorithm to generate our required outcomes.

2.11 Concept of divider algorithm

Nowlet's start with an easy example, which is to partition a given set in every possible two partition. So initially what we do is to take another number or element which is not in the set, and can be recognized distinctly from the other elements of that set. We push it into the set, and then start permutation of the whole set, and when we have every permutation. Then we just traverse each permutation in search of that divider element, and whenever we see it we break the set from there, and hence due to the exhaustive result of permutations, we got every possible two partition of that set. This is known as the divider algorithm of partitioning problem. So following this notion, similarly, we can implement k partitions as well, where without pushing one divider, we will push k dividers. And just as before we will find the partitions, and trace out the divider elements in every possible permutation of that whole set.

But now, there are some drawbacks of this algorithm, both in small and large scale partitioning. So 1st we start with the problem in two partitioning process. Here since we are only partitioning a set, so re-arranging the elements in a specific partition will not be necessary. Even if the set is ordered, then also we need the output, where the same order is also maintained in the partitions as well. but when we using permutation, then even the elements inside the partitions are also getting permuted in the process, which we don't want, and that's why we are in need of some filters to avoid those permuted copies of each partitions.

Now another problem is in using a unique divider for more than two partitioning. In that case, the set of dividers get permuted as well, which ends up generating same partition multiple times, so we have to use distinct dividers for those type of cases, implement some filters over the set of dividers as well to prevent the unnecessary permutation between them. And as we discussed before, we know that defining some sort of conventional order can serve as a standard filter, but here we have to define order separately on both, the set of elements and the set of dividers simultaneously.

So 1^{st} of all, since all the index of our vertices are non-negative, so we introduced

every divider starting from -1,-2,-3... and so on, as per the need of the given degree of partition. Now as another convention, we took the increasing order for the element set, and decreasing for the set of elements. Now keep in mind, the choice of orders is not mandatory, they both can be of same type or in any other way, but here is also a small reason why we chose our divider set order to be decreasing, since later, we have to implement another filter which will check if there is no empty partition in our output, for that we need to keep one end of the divider set in hand, and we chose that to be -1. Because, size of the divider set of the degree of the partition is dynamic, so if we chose to keep the other end, every time, we had to keep track of the last element of the divider set, so we made this adjustment. So to be specific, for our case, we will set two order filter, one will check if the positive elements in the permutation are in increasing order and other will check if the negative elements or the dividers are in decreasing order. And also to make sure that the dividers are well distributed in the permutation or in other words, there are no two consecutive duplicates in the pattern, which may cause any empty partition. That's why we implement another function which will just check for the consecutive existence of dividers inside the permutation and also will make sure that the dividers are well synced with the vertex elements, by which we mean checking the 1st divider or -1 at the beginning of the permutation and checking for non-existence of any divider at the end of the permutation, because if the config generating algorithm reads any divider during the traversal of the permutation, then even at end, it will take another new line with no vertex to fill in it, which makes a violation of our convention.

So this was pretty much the procedure to generate every possible config of given order and given root. But now one question can arise, which we don't have any answer in this project. Generating all the permutation of a finite set in any algorithm is never an efficient attribute of any algorithm. Unfortunately, which is being followed here in many cases. And even after that checking a huge amount of patterns through several filters, only a handful amount of patterns is being selected, which is constant times the same exhaustion all over again. So surely one can ask, instead of curving out a huge dataset to find a few matches, what if there was an algorithm which can generate only those patterns instead of a large amount of garbage data along with it. Maybe using dynamic programming, it can be done in a much efficient way, but we are not sure yet, neither we tried over it. Moreover, there is a small concern why we don't want to use dynamic programming for any part of our project. That's because dynamic programming is basically a time memory trade-off approach, and we do not

want to test our algorithm in any limited bound, so for any higher order test set, there is a chance for the algorithm to make use of a huge data-type resources and it may even cause STL data-type overload as well, so that's why we tried to rule out any dynamic programming approach initially, but later of course if some further work on it provides any way to make these approach work with in an affordable data-type resources, then it will be much accepted, until then we have to go with this inconvenience of permutation filtering process.

Now there is another thing to discuss, which is actually the base of the whole config generating algorithm, which is of course the permutation generating algorithm. Now every well-known algorithm about thisis either in recursion or in dynamic programming, and every one of them are light, easy to implement and less time consuming also. But as we mentioned earlier, we cannot take the chances of unhandled recursion call which may get stuck in infinite loop, and neither can we use dynamic programming because of the data-type resource management criteria, which left us in a tough spot, besides we don't want to store all the permutations of a given set into any STL data types for the overflow concern. So considering everything, we went with the implementation of permutation iterator, which not only avoids recursion but also does not store any sort of output. What is basically does, it to take a permutation pattern as input and return its next permutation as output, here as argument or as output permutations can be presented using array or any other suitable STL's. This method is helpful to us because we can just run a single loop through this implemented iterator initiated with one permutation (here the 1st one), and every time the loop executes we will continue to receive every permutation one by one, until the initial permutation is reversed, which is the loop termination condition for that iterator. We could discuss this algorithm in details here, but that would have made this report unnecessarily lengthy, so we mentioned the gist of it, but skipped the discussion, if anyone is interested, it is just an internet search away.

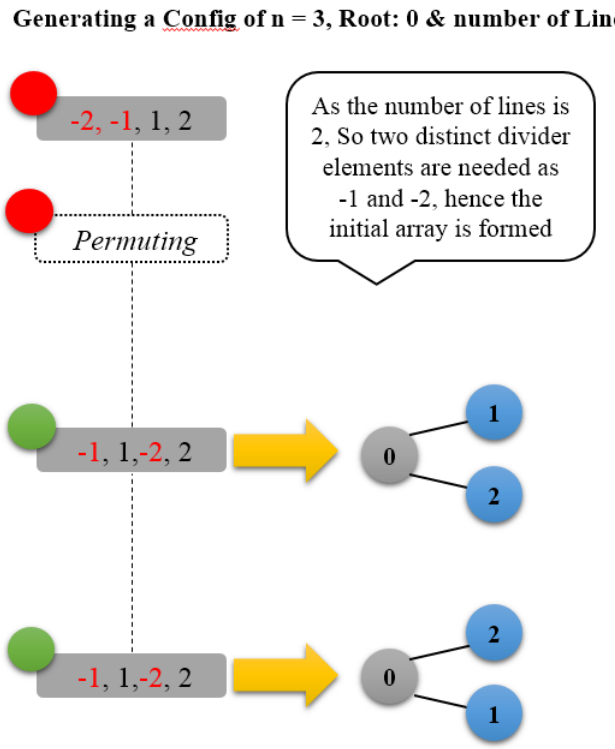

Figure 11: Generating Config With The Divider Algorithm.

2.12 Compatibility between configs

Now to reach up to the all possible pattern generating process, as discussed before, we need to gather the configs in all possible ways, and combining their adjacencies, all the required patterns will be generated. So now, there is one thing, on which we have to clear our concept. Not every config goes with every other configs. So if one assumes that we can randomly choose a config for every single vertex as root, and gathering them will get us a point visible pattern, then he cannot be more wrong, which also means not the normal Cartesian product of every set of configs of a given root will work in this case, as we have presumed. There should be some customizations.

So now, the question arises, why two configs may not go with each other. The answer is the compatibility between the configs. What a config represents is the binary information of some edges between some vertices of a graph, whether they are valid or not, the valid ones are stored in the granted set of edges and non-valid ones are in the forbidden set. So now the contradiction occurs when two chosen configs differs in their adjacency properties, that means some edges that are granted in one

config becomes forbidden in other and vice versa, so using them together, it's not possible to form a pattern.

So to get rid of those contradictions. We need to implement some process which will only gather the sets of all possible configs of the root of each vertexwhich are strictly compatible with each other. This process has no mentionable tricks in it, at least not in this project, each time we push a new config in a set of configs(only if a complete set of configs is in making), we check its compatibility with every other configs that are already present in the set. And by checking compatibility between two configs, we simply traversed the granted and forbidden of each config, to look for any mismatch, which may cause such contradiction.

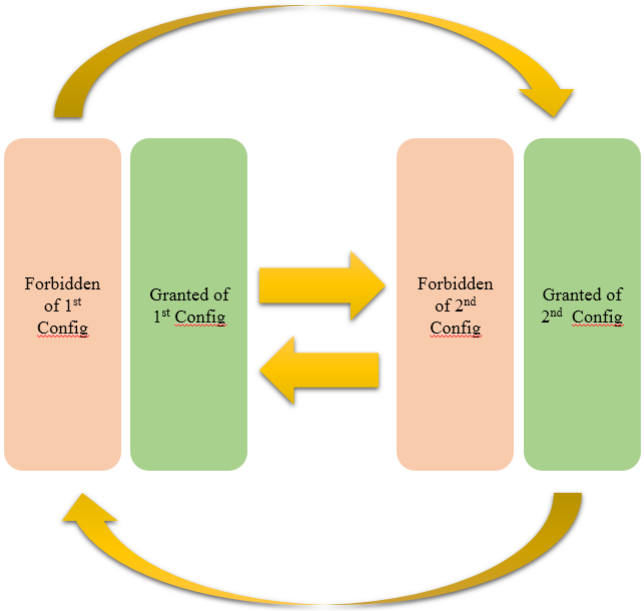

Figure 12: Concept Of What Causes InCompatibility.

Now there is an important query, why not at first generate all sets of distinct but exhaustive rooted configs, and then apply the compatibility test on each of them, which one gets through, stays, and which one fails, gets discarded. The answer of this question tied with the reason, why we used DFS traversal instead of generating customized Cartesian products.Suppose any two configs are not compatible with each other, so of course they both cannot belong to the same set, so when we come across with one such set and it gets discarded, we can understand that there is no point generating another sets which consists of these two configs. But as are we already

set to generating the Cartesian product, so every other such sets will come later and every time for the compatibility test, a huge runtime will be wasted, unless we keep the record of the subset in those already tested sets which are responsible for the non-compatibility, and whenever a new set is generated, we should 1st search for those subset in it, this searching algorithm will cost less time than compatibility test, so as the best case, if at least one of such subset is found, then that set will be discarded and the extra time of compatibility test, will be saved, but if not found, then we have to run the compatibility test over it, if compatible, then the set is passed, and if not, a new pair or subset of incompatible configs will be pushed into the record. These two are the worst case scenarios which takes even more time than the previous process, since both compatibility and subset matching algorithms are executed simultaneously. so we have some reasons not to use even this modified approach, one is the excessive runtime of the worst case, two is that, we don't know how big will be the record, in which we are storing the incompatible configs, also the upper bound of its size is in exponential time, so it won't be a wise choice for any worst case. Besides, maybe for the best case, it's a time memory trade-off, but for the worst one it's the wastage of memory and even more draining of time.

So that's when, the DFS approach came to light. Here we followed the order of indexing of the root of configs, and create them in branches, whenever an incomplete branch is proved to be incompatible, we cancel that branch, or in other words we stop its growth in the algorithm, and for that no further branched grow from this one, that means no subsets are being generated consisting this set of incompatible set. And so the branch never reaches to the full length (which is here $(n-1)$). And unless a branch is fully grown, we will not use it to form the point visible graphical pattern. So we can see, it is not just ordinary DFS, it's a DFS with few more extra controls, which can manipulate the traversal or the growth of each branches.

2.13 DFS: The Most Important part of the program

This DFS actually starts with all the configs with root set to zero, (here also. For the sake of order). This is the 1st level of the DFS, like every other DFS, it doesn't start from a single root node, although, if we separately assign some empty config, which will play no role in the adjacency, that can work as a unique root, from which the zero rooted config level will start, but that seemed rather pointless.

Now after the 1^{st} level, following the ordering, the second level is of all the configs of root indexed to be 1, and it goes on like this. Now every node of the 1^{st} level is

tested with every node of the 2^{nd} level, if found to be compatible, then they will form a branch otherwise not. Now after that for the 3^{rd} level, all the node, that means the configs of root indexed to be 2, will be tested with each branch that are already formed, if found to be compatible, then that node will be appended with that branch, otherwise not, and hence the tree starts growing with only the branches that only consists of the compatible configs.

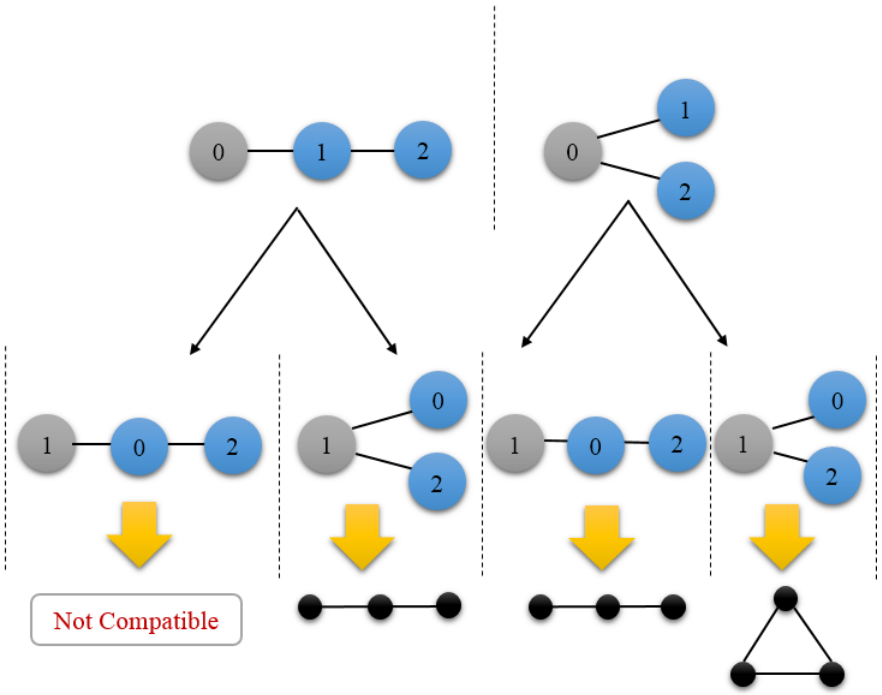

Figure 13: To Understand The Notion, DFS For n = 3.

So we see, a whole lot of extra compatibility testing, extra space consumption and most of all unnecessary repetitions are averted in this approach.

Now to implement DFS in coding, there is a small technical issue that we had to face and overcome. What happens in normal DFS is that, an empty array if maximum possible size of the data-type of the nodes of the tree is created and it is pushed into the DFS algorithm, and meanwhile from one recursion to another, the array is updated at its several indexes, and another variable keeps the track of the length of the array to which it is updated from the previous, when that length equals to the height of the tree, then we can say that one branch traversal is complete, and we traversed a branch up to a leaf node, and then if we want we can go printing out

every branch separately or we can print the original DFS traversal algorithm. Now the text book DFS traversal can also be coded without the help of any array, but as here we are more concerned about each branch, so we need an array or something of that kind to store that, otherwise we don't intend to generate them from the total traversal output. But we face a big problem doing so,we see for an array of primitive data types, updating it at some index is not much of a work, either we use pointer if its passed through any function to do the job, otherwise simply assigning the value was more than sufficient. But here in our case, the matter is somewhat different. 1^{st} we attempted the previous approach, where 1^{st} we fill an array of maximum size with empty config.(object created by the call of empty constructor), and then used this array in the DFS function, and we faced type mismatch and unrecognized operator error. Later on, after some trial and error, we discovered that, since in config the 1st two attributes which were being initialized during the non-empty constructor call, so they are necessary when we create an instance out of a config reference. And when it's not, the object created by the empty constructor just remains as a decoy with no room for it to be updated. Since we don't have any getter setter function in our class, and neither did we overridden the assignment operator for this class. And even after that the same problem comes to the next level of attributes in the config which are not also primary data types, so for them, it is also required to override some of the operator, since this approach was becoming more and more complicated in its way, so instead of previously filled array, we user the vector of config, where we can get pass the updating part, with the implementation of it in terms of push_back, push_front and same for the pop.

And doing so, we faced a few problems as well. maybe out of some misconception 1stwe wrote the code, where after filling up one complete compatible branch in the vector, obviously the stored configs in the vector will be processed to form a point visible pattern and generate some of its properties. But afterwards we were emptying the vector fully to make the room for the new branch, which is a huge mistake that we anticipated after analyzing the output, because this mistake won't show any kind of compile time or runtime error. for this mistake for the next branch all the previous configs except the last one was getting erased, so it was not even showing us incompatibility, so it took us a while to track it down. So after that rectifying it, we introduced another special push function which will keep the whole vector intact, and will push the config at a specified index in the vector. Keeping in mind the resemblance with the array approach in DFS, we did it, because this method

is nothing but updating the vector at a certain index, and we also have the index of corresponding level in hand, but we did mistake again, we forgot that there was also an integer attribute in the array approach which was keeping track of the length of the array up to which it was updated for the next branch. So for the lack of it, our branch was getting updated alright, there was also some of the configs of the previous branch in it, that we don't need. This happens when all the child configs for a particular parent node is traversed and the traversal pointer jumps back to its previous level, so for this case the next node of the previous level will be introduced, and that corresponding index of the vector will be updated while the next level node from the previously generated branch is still in the vector, and hence the problem. And to rectify this, we brought inplace push method, where not only the branch of vector is updated at some index, but also if the index is less than the last index of the vector, that means the traversal pointer is jumped to some previous level of the tree, and wherever a config is updated to some index of the vector, we don't need the configs in the part of the vector after that, so those configs are popped out in this function, so that, after this when the traversal pointer will again jump down to the lower levels, the newly traversed nodes will be pushed to this empty space in the vector, what we freed by popping. This method is working perfectly, so far we didn't come across with any inconvenience in the output.

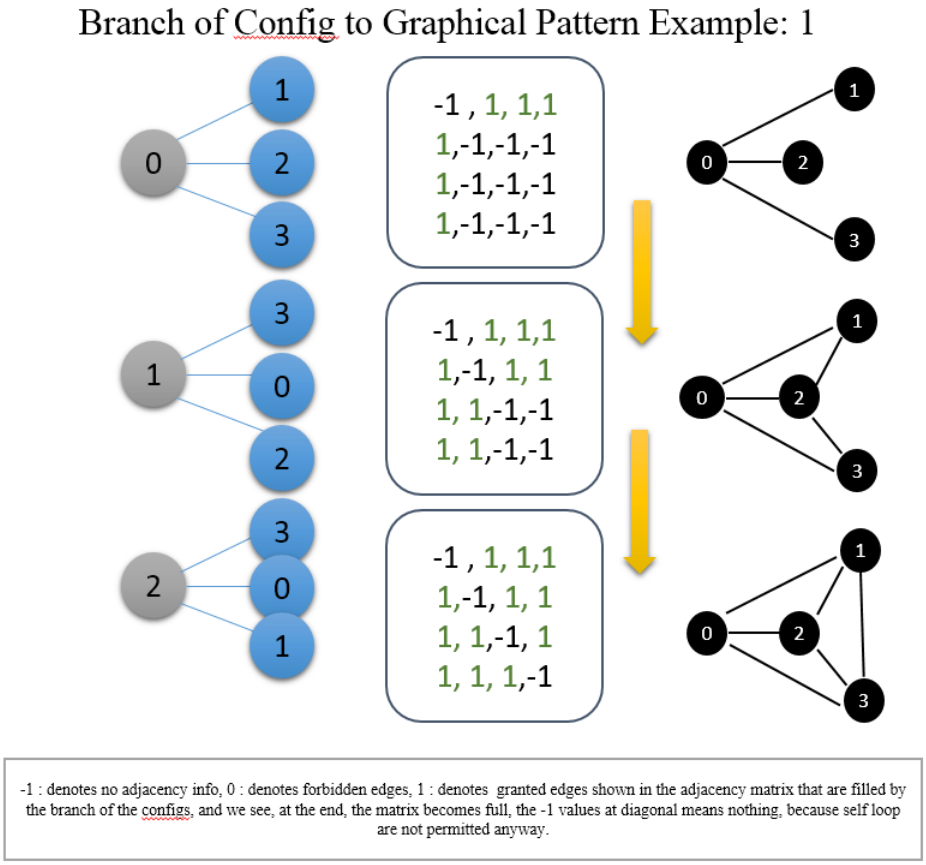

Figure 14: Graph Filling With The Growth Of A Branch Of Configs.

So from this designed process, we get every pattern that we need. But there are some repetitions and garbage patterns as well. but they are considerably less in amount, so that we don't need to worry about it, that much for now. Still forming a mechanism to eliminate those would have been nice, implementation of which will come later in details. Now the 1st kind is the isomorphic copies of already visited patterns, which we don't have solution in this project. So we have to bear with them, with no way around.

But the second kind is much trickier, which we found after some attempts to manually draw every pattern found for low order input. There are some patterns in the output, which satisfy every conditions of our algorithm, and after everything, it even produces a graphical pattern also, along with the collinearity and max clique properties, so as usual one has no confusion about its existence, and that's where we have been proved to be wrong, these graphical pattern may look real, but there

lies some contradictory adjacency properties in its adjacency list, which makes it impossible to represent in a point visible way, or represent at all. And yet, we don't have any counter measure to detect them either.

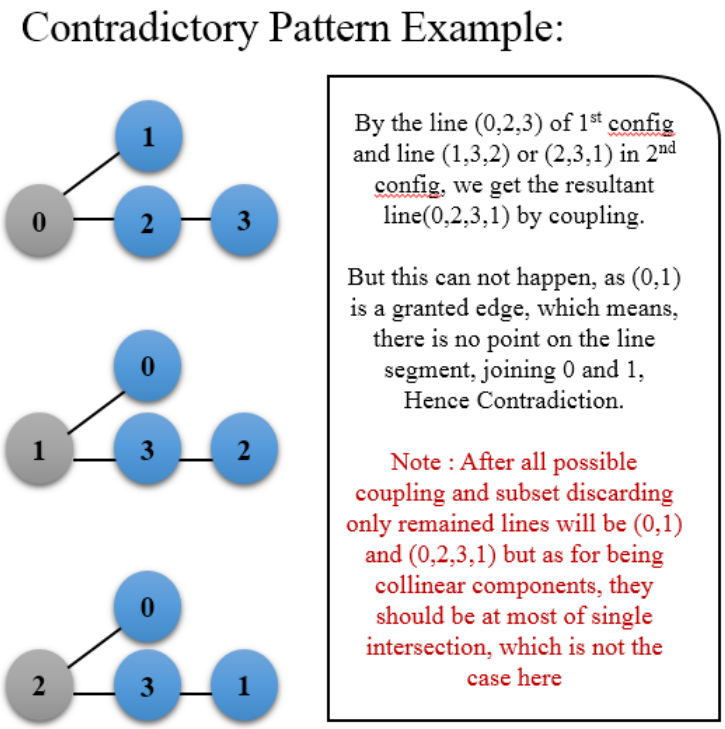

Figure 15: One Example Of Contradictory Pattern.

Now we have to understand what our code does, and even with the permutation exhaustion, why are we still pursuing with this approach. As the l and k are given so for a given n, our code will generate every possible point visibility pattern of order n and check for the specified degree of clique and collinearity in those patterns, if found then that's just what the conjecture states so we let that pass, but if not then that implies not every point visibility pattern of order n possess such clique collinearity attributes, hence this is not a suitable n for this pair of l, k.

So we see, finding a valid pattern out of our clique and collinearity bound is enough, so we don't have to wait our entire DFS traversal to complete, as soon as our program will intercept one such pattern, the program will terminate then and there after printing that in terminal or writing that in file stream, basically storing

it in some form. So as for this reason unless we go for the worst case analysis, the program saves a great deal of runtime. So for average cases, it is a pretty acceptable approach.

So after everything is up and running as it should be then we can just write a function which will take l, k as input and a big upper bound as n, if that n is found to be valid, then from there it will start decreasing until such n is reached for which a point visibility pattern out of the given l, k bound is generated. We stop there and declare the previous value of the n as the lowest of all such n, that we were looking for.

There is nothing to this function so for now we are checking each n manually, since for some upper value of n, it takes a lot of time to execute so running it for several values of n at once may not be a good idea for now, or maybe for some time in future as well. One observation along the way, lesser the values of l, k compared to the value n, the program runs for less time whether n is valid or not. Such an example we can mention for the value of n to be 5 For (l, k) = (4, 4) , n is not valid because it has point visibility pattern of clique 3 or collinearity 3, and there are so many isomorphic copies of such pattern, but as we are done by only having one such pattern in our output, so that is not our problem unless we are in search of every point visibility pattern of such attributes. But same thing for (l, k) = (3, 3), takes much less time compared to the previous, also it is noteworthy that 5 is a valid n for this l, k pair.

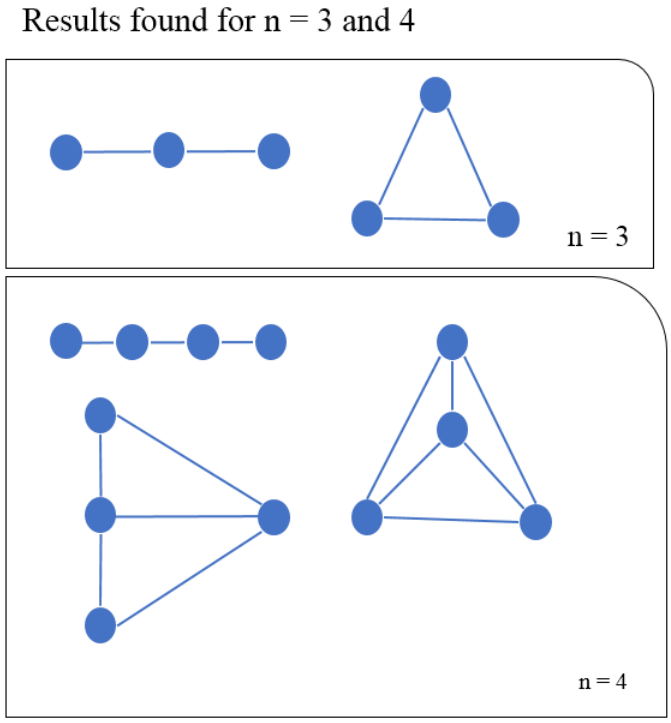

Figure 16: Known Outputs For n Value 3 And 4.

2.14 About the optimization

Since we are still going with the algorithm, one which includes generating every possible permutations of a given set of values, so we can't say that much optimization happened there, but if one goes through this whole project report, one can see that, without tampering the working flow of our algorithm, whenever there was a scope for some optimizations, we took it, whether that's in the coding of some parts of the main algorithm or changing some approaches at some parts. Although one can ask about the many searching and sorting algorithms that have been used in this project. Now where it was our turn to code, we mostly used some linear time algorithm for searching and n square algorithm sorting, to make things simpler, because at that point of time, our main concern was to first make this algorithm work properly. Once that goal is achieved, one can easily replace those n and n square algorithms with some log(n) and nlog(n) ones respectively. And where the searching or sorting member functions were called of some vector or list, we can safely say, they are already optimized, because in almost every STL data types in C++, the sorting member functions that is used

in merge sort and the searching algorithm is the binary search, so our concerns for those parts are already covered.

2.15 A revised Concept: Generating Collinear Components

Now there is nothing to mention about the clique check method, that we implemented here. It's pretty much a brute force attempt with every possible subset of the vertices of the pattern with more than 2 elements, and checking if it is clique or not. We can keep storing them or keeping space-time in mind, we can directly calculate the maximum clique on the process.

But there are a few things to specify about the collinearity check.1st is to settle our concept about it. Now at the time when we started coding, we thought as the lines in the configs are actually the straight lines in the graphical patterns, constructed by them. So maybe the lines of the configs are actually the set of all lines in that graph, so one can think that finding the line of maximum length among the configs in a branch can give us the maximum collinearity in the graph, and that's where we stood corrected. We can give an example to make things clear, suppose there is a line 0,1,2 in some config of a branch, so undoubtedly that 0,1,2 is a line in the graph, but suppose there is another line 1,2,3 in some other config of the same branch, so like before, it's also a line, now suppose, considering the whole branch, these are the highest length lines found so far, so by our previous notion, our maximum collinearity output should be 3. By the way, as we specified the lines, of course those were including the root vertices in them. So now getting back to the topic, here we see 3 is not the maximum collinearity, since (0,1,2) and (1,2,3) have an intersection (1,2), since this is more than a single point, so both our lines are being coupled by that portion, giving us the resultant line (0,1,2,3). Hence the maximum collinearity will be 4 here not 3. This happens a lot for the linear representation of the n vertices. We can show some of them in the example as follows.

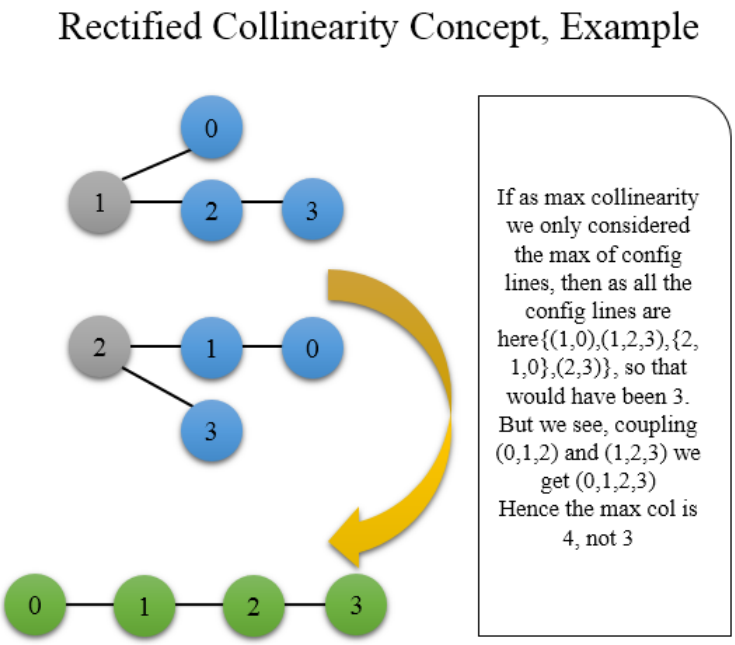

Figure 17: Concept Of Coupling Between the Unit Lines of Config.

The notion is clear now, but while implementing it in code, we had some setbacks, that we are going to discuss now in short. 1st of all we implemented the method, which will detect between any two lines in a branch of config, whether there exists intersection between them for coupling, or not. Now see, if the intersection is a single point, then coupling the two lines, may or may not produce another straight line, so by convention we did not take the lines having a single point in intersection for coupling. Most probably the overall output won't be disturbed by it. Now while implementing this, we forgot about the commutativity of the edges, such as, let (2,1,0) is a line, and another is (1,0,2). So practically then cannot be coupled because of the repetition of the point 2, out of the intersection area, in fact these two lines can cause a contradictory pattern, to which we will come later, but for now, we see this sort of coupling should not be permitted, so we implemented another filter method, which prevents all those coupling that can cause repetition of values in the resultant line.

Now someone can think that, while two lines of two different configs are getting coupled, then why keeping those two lines, we can just keep the line that we coupled

and delete these two. Actually the thinking is not wrong, because, they have no place at the final set of collinear components of the pattern forged from the branch of configs, but we cannot delete them in some intermediate stage just yet, because afterwards there may be some other lines in the branch, which can be coupled with one of those but not with their previously coupled lines.so considering this possibility, after generating every possible coupling, then finally we apply another filter method, known as the subsetFilter, which checks if any line is a proper subset of some other line, present in the set, if so, then that child line can be deleted, and hence finally we have only the collinear components of the graph.

Another thing to specify, which is, whenever we are checking for intersection for coupling between two lines, we should always check them from both sides and from reverse as well, as example, line (0,1,2) and line (1,2,3) has intersection (1,2) for coupling which will give us (0,1,2,3) as a result, but if we checked line (1,2,3) with line (0,1,2) we see there is no intersection for coupling, hence our point.

Now after we find the complete set of collinear components of a pattern, then we can take the longest of them, whose length will be the maximum collinearity for that pattern.

2.16 Next Plan: Some necessary modifications

Now what we implemented so far, is to traverse every permitted branch up to the end, that means when it grows to be of length (n-1). We stop further traversal depth wise and calculate the maximum clique and maximum collinearity of the pattern formed by the configs in that branch. One of them is supposed to be (n-1), if so, we move on to the next pattern generation procedure, but if not, then 1st to check two things, one is if the convention for the line coupling really applicable here, if not, then we have to couple two lines even for single point intersection, and apart from that checking for contradictory adjacency in the pattern is also necessary, because that can affect the collinearity as well, but we will get back to it later.

Now keeping main concern in mind, our goal is to find both out of bound collinearity and out of bound clique property in a pattern. But there is a question, do we really need a completed branch for that? The answer is NO. such as, if in a branch for given n=3, there should be two configs in every complete branch, but if the 1st config consists a line 0,1,2, then at that point of time the collinearity becomes full, hence we can move on without visiting the next config of that branch. This notion saves us a great deal of extra traversal. So instead of applying the clique and collinearity check

at the end, we just need to apply every time a branch gets a new config pushed in it. For best case or for average case, it is a very efficient approach for no doubt. But in worst case scenario, all those intermediate checking appears to be for nothing, which is a big waste. But for the sake of efficiency in average case scenario, this approach can be implemented. The second thing, which is very much necessary is to implement some filter, which can recognize the contradictory patterns from the output, because, we don't know much about those patterns so far, since they are non-existent collinearity wise, so most of the time, miscalculation will happen when we try to determine the maximum collinearity of those patterns, and due to this, it can produce less than complete collinearity, which may seem to be a counter example, that we are looking for, so there will be several confusions in the output because of this kind of patterns.

But the problem is, the output over order 5 becomes huge in amount, so checking each of them manually to look for the attributes which cause a contradictory pattern is a tough work. And until we have a considerable amount of knowledge about it, it will not be possible to implement this method in general.

But studying the patterns for n = 3,4 and partly 5. We noticed a collinearity attribute which causes contradiction in many places. If two collinear components appear to be intersecting in a portion with greater than one point, then obviously there is no place for this sort of construction in a point visibility pattern, hence a contradiction is created, until any other such attributes are discovered, we can just use this one in our contradictory pattern filter. So these were the two things that can but yet to be implemented in our code. Besides any other approach apart from what we have now, any other optimization approaches on our existing algorithm are also the field to keep an eye for.

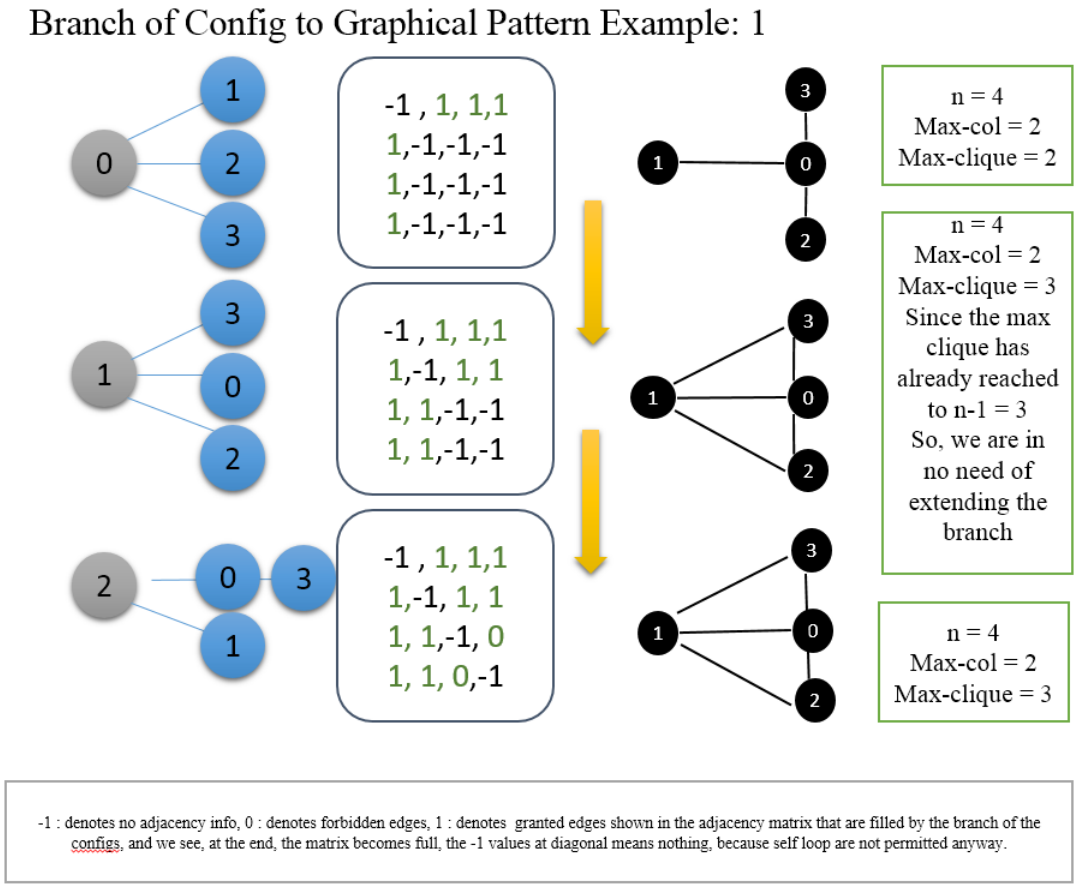

Figure 18: Why Intermediate Checking Is Necessary.

2.17 Implemented Modifications, Results and Setting Future Works

As we planned before our next step, we implemented those two modifications in coding, 1^{st} we had a few problems implementing the collinearity filter because, if at the end we include all the corresponding lines of all the configs of a branch in a set, and then try to couple them and sort out the components, then we had to deal with a whole lot of inconveniences at once, so we implemented the intermediate line modifications in this algorithm as well, instead of gathering all the lines at the end and matching each of them in search of positive coupling, it starts with the 1^{st} two configs and by coupling filter, it generates the possible coupled lines out of these two,

and includes them in the set, this set is now an initial set of lines in which we are keeping both the coupled and the unit lines, and every time we get a new config in our branch, we iteratively modify this set by applying coupling filter with respect to the newly introduced config and the lines that are already present in the set. Now there is a question, why do we keep both coupled lines and the lines by which the couples are formed both in the set, why not applying subset filter in those intermediate stages to discard them and make the set lighter. But there is a problem in doing that, if later on in some stage of iteration some other lines appear, which can be coupled with some unit lines from other previous configs but not with those coupled lines, then for that our unit lines won't be available since they are cancelled by subset filter long ago. Only for that reason, we are keeping the units of the coupled lines as well, applying the commutativity and subset filter at last, when all couplings are done. And after this, the set that we will be left with is the collinearity components of that graph.

But we have a small doubt in this matter, that we have to investigate more in future. After each iteration, the set is updated with the newly formed coupled lines and the unit lines of the new config. But is it possible that there is still scope for coupling between the members of this set, ifso, then after this iteration we have to pass this set once more through the coupling filter, customized for the members of its own. But we cannot say anything after we study a fair amount of output for several values of parameters.

Another similar query is, if it is possible to apply the commutativity filter in the intermediate stages. Since at the time of overlap checking, each lines are checked both in normal and reverse way another line. So We think it will be alright if we don't keep the reverse copies of the lines in the set. But still, we need to be sure about that before we make any move.

After all these, the 1st thing to implement was intermediate check, to do this, we needed a allFilter function, which includes every conditions that matters for a branch termination in the middle of its making. So needless to say the contradictory pattern checking, the clique checking and the collinearity checking, all those filters will be there, if any one of them is satisfied, the entire function will stop checking further, and will return false, reading which the branch will stop growing, and the leaf node of this current branch will be replaced by the next pattern of its kind.

Now one thing we have to keep in mind, that the contradiction or the collinearity or the clique property, they are invariant under the extension of a branch, or in other words the further decoration of a pattern with edges. Which means, suppose a set

of vertices is found to be generating a contradiction or a collinear set or a clique of a pattern in making, but that does not mean the set will lose its attribute when the pattern is done making. And that's the reason we can use these properties to quit a branch formation in some intermediate point. If it was other way around, then we did not have any other choice but to traverse up to the end of every branch. Now since we discussed the techniques about the implementation of the filters in the previous section, so here we are trying to focus on what happened after we implemented it.

First of all, what our main concern was, the extensive runtime, that became handy a lot. Previously for n = 5, what was taking more than 5 hours to run completely,now it's getting done in about one and a half hours and also generating very less amount of text output compared to before(when generating not one but every bound violating patterns, for one, it only takes about 27 seconds for (l, k) = (3, 3) and about a minute for (4, 4)). that's a progress for sure. Besides the patterns that are getting printed now are much relevant to our search. So for this advantage we were able to run it up to n = 8 for several l and k values. The whole thing is working perfectly, but we noticed as the n increases, the isometric copies in the output is becoming a serious problem. Most of the times for values above n = 7, these generating copies takes up several extra hours with no new information what so ever.

So now, I think our main focus on future should be specifically on two things, one is to develop at least some basic property based classification filter to walk pass the isomorphic pattern repetition problem for which formation of many similar intermediate branch will be prevented, which will come as an additional help later. And another comparatively less important thing is to modify the contradictory filter, the one that we already have. Apart from these two, I don't see any more major issue that needs to be worked on currently. By the way, another surprising thing we observed, is that, once we implemented the collinearity filter, most of the time contradiction filter is not even needed, before coming to that it hits to some collinearity check and returns false as we formulated. But that does not mean we can rule out the contradiction checking, it's still very much necessary. We just stated an observation while studying the flow of the program in text output, nothing more than that.

2.18 An Efficient Time Memory Trade Off: Replacement of Divider Algorithm and Additional Future Works

We ran our program for $l = 6$ and $k = 4$ and after almost two days, it came up with a bound violating pattern of maximum collinearity 5 and maximum clique size 3, showing that 7 is not appropriate for this pair of l, k conjecture wise. And after this little achievement, we went off to run it for $n = 8$, for the same l, k as before, and while the debugging output was being printed on the terminal, we noticed one thing. That its taking too long (almost an hour or two) from going to one valid array pattern to another, as the divider algorithm is doing its job. So we all know this gap means nothing but all the garbage array patterns in middle of it, which are totally unnecessary. Besides if we thing about it, suppose our described exhaustion is taking place the config generation of root 3, which separately runs for every single compatible leaf node of its previous level, causing the same exhaustion, only just multiplied with a huge number this times. So as we keep staring to the terminal, most of the time, its remaining static because of those intermediate invalid array patterns.

So two sorts of modification are needed based on the current situation. One is to develop some new algorithm which will not be generating those intermediate unnecessary pattern, or if does, will do in a very low scale, which won't keep our program hanging. And another is to implement some techniques which won't repeat the same config generating algorithm for every cases all over from the beginning. Solution for this second dilemma came in quick, we just generated every possible configs at the beginning of the program and stored them in proper data structure, and whenever they are required we accessed them from the storage instead of wasting the runtime in regeneration process. So needless to say, after storing of all the configs of all the roots are complete, only then the DFS traversal is initiated, and for this modification DFS can run smoothly without any interruption of config generation by the mining process of divider algorithm. We are describing it first because we implemented it before dealing with the 1st issue, because that time, we did not have any idea of alternative algorithm which can replace the divider algorithm. So at this point, we ran our algorithm, it went fine up to $n = 7$, but at $n = 8$, still the same problem, only this time the advantage was, we were being able to read the time of all configs generation separately, because it runs first and then the traversal starts. So we saw it went almost 12 hours on the 5 line config generation alone with no sign to yield, so we had to force stop it, otherwise who knows how much time it would

have taken to terminate on its own. So after this disappointing experience one thing became clear to us, that we need to do something about those never ending invalid array patterns in the divider algorithm, so after a whole lot of trial and error, we came up with another efficient algorithm that can reduce the runtime of divider algorithm by approximately $(2n)!/(n)!$ times.

We should mention first that the queue approach to implement BFS and the normal DFS were used in combine to device this new algorithm. And surprisingly beyond our expectation it did not produce one single invalid array pattern, for which we were troubled mainly. We can elongate the report blabbering about the mathematical description of this algorithm in details, but to be honest there is not much to it that can come as a surprise to anyone who have gone through any normal course of DSA, we just put a few things from several standard algorithms together, with only the plan of assembly of our own. So we are directly skipping to the example part, which probably won't cause anyone any problem to understand the basic algorithm. Suppose we are dealing with 6 points apart from the root, that are to be situated over the lines of the generated configs and let 3 is the given count of lines. So here we 1st determined all the distinct 3 partitions of 6.

For doing so, we 1^{st} initialized one vector $(4, 1, 1)\ldots$ idea is to give single points to every other partitions but the 1^{st}, and put the remaining amount in the 1^{st} partition. We store those partition patterns in a map STL, why map? We will see that later. Now we see the map is empty so, we push $(4, 1, 1)$ in it. As we followed DFS so, it will call to the next recursion from here. So the leaf node will execute 1^{st} for the remaining part of the function. The notion is that we insert every permutation of this pattern into the map. This is done, only when a leaf node is intercepted. and to create children from a partition node, we reduce one from the 0th position and add that to the ith, while i is varying from 1 to 2 positions. So from $(4, 1, 1)$ we get $(3, 2, 1)$ and $(3, 1, 2)$ respectively, since when $(3, 2, 1)$ is processed, all of its permutation copies are getting included into the map, among which $(3, 1, 2)$ is also present, so when it comes to this element, we don't process it to avoid any unnecessary repetitions. Now similarly from $(3, 2, 1)$ we are getting $(2, 3, 1)$ and $(2, 2, 2)$. As $(2, 3, 1)$ is already present in the map, so we again we don't process it, but for $(2, 2, 2)$ we include every permutation of it, which is pretty much itself. After that from $(2, 2, 2)$ we get $(1, 3, 2)$ and $(1, 2, 3)$ which are also of no use. Now here is a thing, what if any of those last two patterns was absent in the map, then also we would have stopped, because there is 1 at the 0th position of both of the patterns, here what we set as the termination

condition of the DFS traversal branch wise.

Finally, as we chart the map at several stages
1^{st} $(4,1,1)$ from which $(3,2,1)$ and $(3,1,2)$ Now from $(3,2,1)$ we get $(2,3,1)$ and $(2,2,2)$
From $(2,3,1)$ we have $(1,3,2)$ and $(1,4,1)$ both are terminating nodes, since 1 is present in their 1st index. So now by permuting $(1,3,2)$ map becomes
$\{(1,2,3),(1,3,2),(2,3,1),(2,1,3),(3,2,1),(3,1,2)\}$.
Then by permuting (1,4,1) it becomes:
$\{(1,2,3),(1,3,2),(2,3,1),(2,1,3),(3,2,1),(3,1,2),(1,4,1),(1,1,4),(4,1,1)\}$.
Now tracking back, we come to $(2,3,1)$ which is already present, so we don't process it, then comes $(2,2,2)$ it is not present, so we permute it, and add to the map. And the map becomes now:
$\{(1,2,3),(1,3,2),(2,3,1),(2,1,3),(3,2,1),(3,1,2),(1,4,1),(1,1,4),(4,1,1),(2,2,2)\}$.
After $(2,2,2)$ we get $(1,3,2)$ and $(1,2,3)$ which are already present so we don't process any of them. Backtracking 2 steps from there we come to $(3,2,1)$, after that from its right sibling $(3,1,2)$ we don't proceed further because this node is already present in the map. So at last we have $(4,1,1)$ the root node, which is also present, so nothing to process further, hence we get the map as our required $6,3$ partition. We see, in this process no unnecessary patterns are getting generated. So from that point of view it is very much efficient, moreover it follows the DFS approach in map filling, so runtime is optimized as well. now we are not going to the details about how configs can be formed out of those partitions, anyone who have come across this report from the top won't be needing that discussion, besides it is quite straight forward also. So now we come to answer the question, why map? Why not any other data structures of the STL family. The reason is pretty simple too, without the hashing feature of map, we had to traverse the whole content of data structure to determine the presence of a pattern in it, which would also be a waste of runtime.

We were amazed by the result that this algorithm provided, what was taking at least 12 hours (" at least " because we terminated the program without giving it a chance to complete) before, is now getting executed in only about 20 minutes for $n = 8$. After which the flow is transferred to the main DFS traversal of our program. Not only this time memory tradeoff discarded every unnecessary patterns of divider algorithm, but also it resulted beyond our imagination in runtime. Here we have optimized in one more aspect. Now we are only generating all the permutations of the rest of the vertices for a given root once, and storing it. And using that storage

multiple times with several sets of partitions with respect to different line numbers, from our previously described algorithm, we are generating every possible configs of a given root. Which was also not the case in the divider algorithm as we recall.

Not that this approach is not a good modification in our program, but it has a few consequences as well. if we are in a best case scenario and since it is a DFS approach, so even if the config generation process was embedded with the main traversal then also after generating a few configs of each stages, the flow of our program would have taken us to the bound violating pattern that we are looking for. So we see, for this case, generating every configs of every stages are not only unnecessary but also pointless. Needless to say, an awful waste of memory and runtime as well.

So taking this under consideration, an approach might have been helpful, but because of its complicacy we have not put our hands on it yet. In this approach, we have to maintain pointers in each config generation stages. Of course the stages are distinguished root wise. Then up to which the stages are filled by one branch traversal, that will be marked by those pointers individually, and for the next branch traversal, 1^{st} the traversal will use the existing resources and after that if needed, new configs will be generated, according to the which the pointers will be displaced. This way the storage is getting filled according to our need, and not more which can get wasted for any best or average case scenario. But from our point of view, keeping and shifting those pointers won't be hard. But what we really be concerned about is the technique by which we can sync those pointers with the multi-functional config generating process. Because we noticed, pointers may be different for different stages, but the config generation function is the same for each of them, only the root altered through parameter, so by that also, if we can manipulate those respective pointers somehow, that can be helpful.

Another thing is the concern about the memory consumption due to this approach. We all agree that this time- memory tradeoff is a great solution from many aspects, but still one undeniable thing is the size of the memory that are being used in the process. So it's not a general approach to follow. For the n values near 10 and above, we have to be cautious about the space-time, so that any RAM overload does not occur, which may collapse our system.

So in our opinion the best solution is to keep those calculated configs into a database system, so that they won't cause any problem for the primary memory of our computer. But in that case, data insertion and retrieving, to and from the data base will cost us some extra times, but for the sake of a compact data management

system and not worrying about the program running for any higher values of n, we think that's a reasonable price to pay for. But in that case, we have to transfer the whole code in java from which we can implement this approach with the help of standard SQL and JDBC. So we are leaving these last two ideas in the hand of future work for now. If in future we can implement this, there is another modification that can be done with the program, that is from the 1st level of the DFS, means the level consisting of all the configs of root zero, we can divide our traversal there by those configs in multiple thread, since the data that we need for the traversal is already in our hand because of pre-generating the configs, so it won't be a problem for the threads to access that simultaneously. In that case, not only the runtime will be divided by the number of configs of root zero, but also bound violating pattern will be discovered in a bounded time, for all best, average and worst case scenarios, if there is such pattern at all for those given set of inputs.

2.19 Results

Last we ran our code for several l, k values and for $n = 7$ and $n = 8$ only for the values $l = 4$ and $k = 5$. So Initially we found several patterns for $n = 5$, but in our knowledge, that's fairly common, and available in many books, so we didn't include those in our project report.

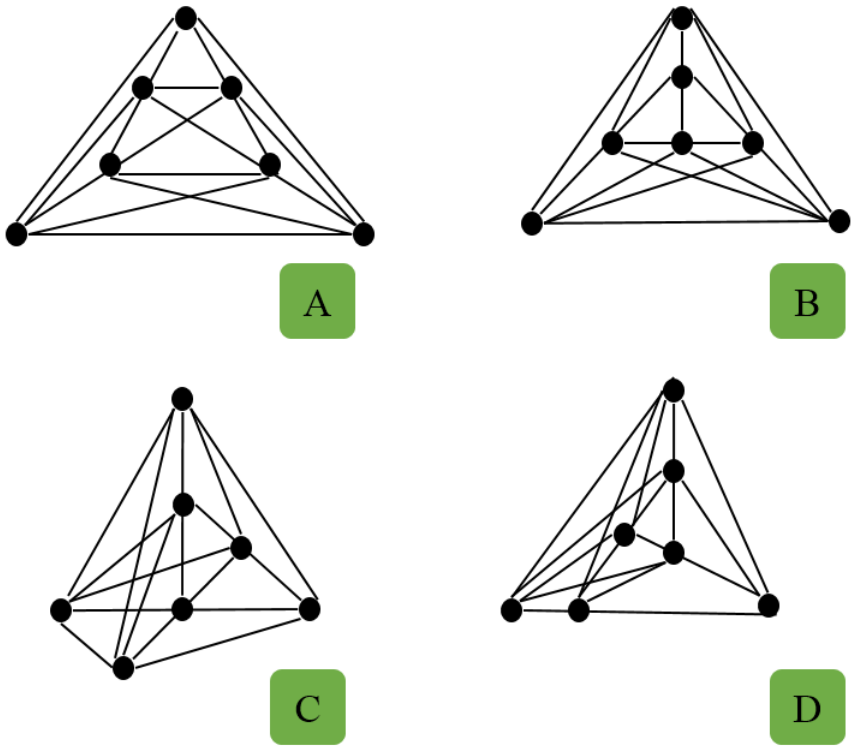

Figure 19: A Few Drawable Patterns Of n = 7 with Maximum Clique Size = 4 And Maximum Collinerity 3.

We showed two patterns one of $n = 6$ and another of $n = 7$, in the beginning. They both were formed by the output of this, program, but for those, we did not show anything about its formation from the output, which we are going to do now. In the last attempt, before our report, submission, we ran it for $n = 7$, $l = 4$ and $k = 5$. So generated some filtered bound violating pattern output in text file, some of which were found to be contradictory, but we were not able to extract the property which was causing it, so that we can include it in our contradictory pattern filter. Some others were sort of confusing, we are not yet sure, that they are valid or not, just because we are not being able to form an image out of it, does not mean that the pattern is contradictory, it can also be possible that we are lacking some point of view from which, the drawing can be made. So We kept those two kind of output for our future study. But fortunately it was not all, we found some valid patterns too. Among them, we are showing one, and also the process of its formation. One valid

pattern of order $n = 7$, maximum clique size = 4, and maximum collinearity = 3 was found, along with this branch of configs.

1. 1^{st} config: Root 0, Lines (0,1,2), (0,3,4), (0,5,6)
2. 2^{nd} config Root 1, Lines (1,0), (1,2), (1,3,5), (1,4,6)
3. 3^{rd} config Root 2, Lines (2,1,0), (2,3), (2,4,5), (2,6)
4. 4^{th} config Root 3 Lines (3,0), (3,1), (3,2), (3,4), (3,5), (3,6)

Here we see, branch is not of length 6, but we are still getting a pattern, which has no edge to fill further. This happens often, before completion of the branch, the graph becomes full in some intermediate position, so clearly letting that branch grow after this is unnecessary, so we just traverse up to this, and then move on to the next branch, this control in implemented in the intermediate checking process, which we have probably forgotten to mention before. Getting back to our description, this branch produced a full graph, with the set of collinear components given below:

$$\{(0,1,2),(0,3,4),(0,5,6),(1,3,5),(1,4,6),(2,4,5),(2,6),(2,3),(3,6)\}$$

. With the help of these component we were able to draw this graph.

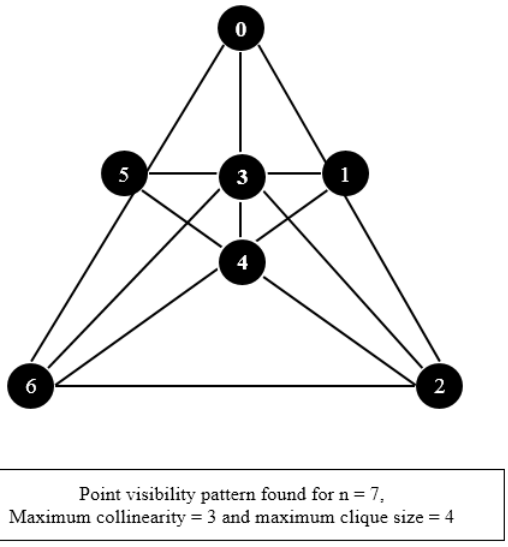

Figure 20: Drawn Pattern From The Described Output.

The graphs shown in the figure 19 are also of such kind, which we were able to draw successfully. There are four patterns in this picture, where first two patterns are symmetric and the next two are non-symmetric. The first two may seem familiar,

and there is a possibility of finding some of them in several papers or books. What we observed is that the symmetric patterns are comparatively easy to think, and even in some cases, we can create them without the help of the output of our program. But the non-symmetric patterns are quite hard to think of our own. So specially for them our program becomes very much helpful.

We would like to mention one more pattern found for $n = 8$, with maximum clique size 4 and maximum collinearity 3. There is a reason why we are only mentioning this one, not any other. That's because if we see closely, we can see that, this pattern is formed from another lower order pattern, for $n = 6$, which we have already shown in introduction. And this two patterns can imply probably imply a bigger, which is that, we can make any point visibility pattern of bigger orders if we have some relevant patterns in smaller order. In them, if we can push the new points without disturbing the collinearity, then joining those point individually with every other points of the previous pattern will only increase the clique size, but following our construction, the collinearity will remain same.

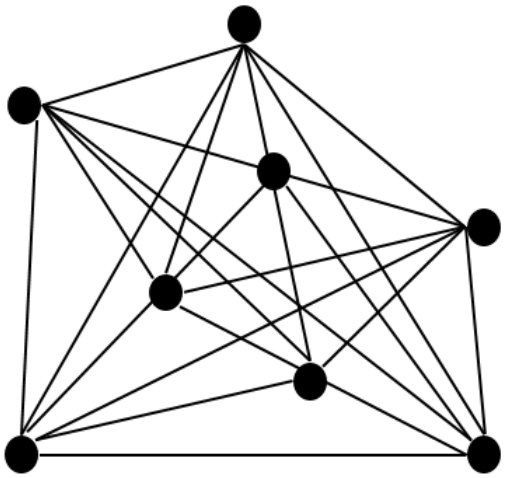

Figure 21: A (8,4,3) Pattern Output, Concept Of Forming Higher Order Pattern From Some Lower Ones.

As in the above picture, we just added one line with two extra points on its terminal, with the middle point being one of the point in the inner triangle of the $6, 3, 3$

pattern in the introduction, and after that we just connected those two terminals with all the previously present vertices, and as a result, it gives us one $8, 3, 4$ pattern.

3 Conclusion

As we expected in the beginning, we now have an algorithm that can check the validity of a n corresponding to a given pair of l, k. first this program takes the value of n as input, and then it asks for the upper bounds of the cliques and collinearity, which bound we enter, the program will discard the formation of every pattern equal and above those bounds, and the patterns which are staying below it, the program releases them as output in some text file.

We can use this program for two purposes, one is to find the existence of a bound violating point visibility pattern which is set to be our main objective, and second is generating every such violating patterns, which in general may seem unnecessary but if anyone is interested to study those filtered patterns, for that it will be a big help.

We came a long way following this objective. Our 1^{st} brute force approach was proved to be correct, but we abandoned that in search of something new and more efficient, so we followed this combinatorial approach, which is pretty much the rest of the report. Now our 1^{st} concern was to create a working algorithm of this sort, when that was done, we went for perfecting it, some changes were made, some are still due, and some are yet to be found. We ran our algorithm for the value of n up to 8, for $n = 8$ it ran for several hours, after creating almost 35mb of text output, we had to make it stop. When the further modifications will be appended, maybe this output amount will reduce considerably, or maybe some other drawbacks will arise, in need of further correction. So to sum up, no it's not a complete success, maybe not even close to that, but it's a first step towards that, which worked. Besides the second approach that we used here is a pretty unique way to deal with this problem, which actually resulted a successful program with way more efficiency than the basic geometrical approach, so there is chance that this combinatorial approach may majorly benefit not only this problem but also this kind of other problems as well in future, and can become popular in practice.

Ps: There is one very important suggestion on how to use this program, which can save us a lot of time, that is for a given n, we should start with the smallest l, k bounds possible and run the program for that, because if it finds any bound violating pattern for those small bounds, those patterns will also hold for the bigger bounds as

well, and if no such pattern is found, then we have no other choice but to move on the higher bounds. We learned it the hard way after running it for $n = 8$, $l = 6$ and $k = 4$.

References

[1] Improved Bound in the proof for k = 5 and arbitrary l by Abel et al.

[2] Pentagons on point sets with Collinearities by Janos Barat, Vida Dujmovic, Gwenael Joret, Michael S. Payne, Ludmila Scharf, Daria Schymura, Pavel Valtar and David R. Wood

[3] Big Clique Big Line Conjecture is false for infinitely many points. A proof made by constructing a countably infinite point set with no line of size 4 or clique of size 3 made by Attila por, David R.Wood.

[4] On Visibility and Blockers: Journal of computational Geometry by Attila Por and David. R. Wood.

[5] Geometry and Combinatorics. A collection of research papers by Zachary Abel, Brad Ballinger, Prosenjit Bose, Sébastien Collette, Vida Dujmović, Ferran Hurtado, Scott D. Kominers, Stefan Langerman, Attila Pór, David R. Wood

[6] Blocking Coloured Point Sets by Greg Aloupis, Brad Ballinger, Sebastien Collette, Stefan Langerman, Attila Por and David R. Wood. (The two starting pictures at the introduction are taken from here)