An Introduction to Kolmogorov Complexity and Compressing Kolmogorov Random Graphs

Manav Bajaj

December 7, 2024

All code used throughout the paper is available here: Kolmogorov Covering.

1 Kolmogorov Complexity

The Kolmogorov Complexity or the K-Complexity of a sequence given a fixed language is the length of the shortest program that outputs the given sequence. The notation is as follows, we use K(x) to represent the K-complexity of a sequence 'x'. Mathematically, the K-Complexity is defined as:

$$K(x) = min(\{length(e) \mid \psi_e(x) = 0\})$$

Where:

- 'x' is a binary sequence (input)
- 'e' is the program that is to be evaluated, which is also a binary sequence.
- ψ is a Universal Turing Machine that runs the program 'e' and checks if it outputs 'x'. If $\psi_e(0) = x$, then 'e' outputs 'x'.

Simply put, the K-Complexity of 'x' is the length of the shortest program 'e' that outputs the given sequence 'x' given 'ψ'.

1.1 Computability

Intuitively, there are infinitely many programs capable of producing the output 'x'. To identify the shortest program, we would theoretically need to execute all possible programs, verify whether they produce 'x', and then select the one with the minimal length.

Alternatively, as the runtime of an algorithm designed to compute the complexity K approaches infinity, it converges to the shortest possible description of 'x'. This highlights why the K-complexity of a sequence is fundamentally incomputable: determining the shortest program requires exhaustive exploration

of all possibilities, which is not feasible.

Since the K-complexity is incomputable, we cannot determine a definitive lower bound. However, we do have an upper bound, which is given by the length of x itself. This is because the most straightforward description of 'x' is 'x' in its entirety, making its length an explicit upper limit on its complexity. This can be given by:

$$K(x) \leq length(x) + c_1$$

The additive constant c_1 comes from the invariance theorem. If you are interested, you can find the proof in [3 pp. 104-105]

Kolmogorov complexity is a measure of true randomness. In a pragmatic setting, we compute the resource-bounded Kolmogorov complexity, wherein we cannot run an algorithm for an infinite amount of time. The resource bound Kolmogorov complexity gives us a measure of pseudo-randomness.

1.2 Why Kolmogorov Complexity?

After reading the definition of K-Complexity, you might ask a question, "If we cannot compute the K-Complexity of a sequence, how is it of any use to us?".

The main idea is to use the fact that Kolmogorov Complexity is incomputable and relate that to other supposed 'incomputable', 'non-calculable' problems/results ((3) & (4) from the list below) and prove their results. In simpler words, show that an analogous problem is incomputable/undecidable using an already existing incomputable problem.

Here are a few uses of Kolmogorov complexity without the burden of detail:

- Kolmogorov Complexity is used to distinguish between random and non-random sequences, measuring the compressibility & entropy of algorithms/strings.
- Kolmogorov complexity makes an appearance in statistical physics as a measure of entropy.
- Kolmogorov complexity can be used to prove the undecidability of the Halting Problem.
- Kolmogorov complexity can also be used to prove the results of Godel's incompleteness theorem.

Throughout this paper, we will concern ourselves with sequences of $\mathcal{O}(1)$ K-Complexity and $\sim \mathcal{O}(1)$ K-Complexity that we can intuitively 'compute'. In case of graphs, we know that for a predefined type of graph, such as a cyclic graph, the description will be shorter than printing every bit out.

From [1], we will define basic and random graphs further down the paper, and

using a pre-defined set of $\mathcal{O}(1)$ graphs, we look to either compress non-random graphs or definitively say that given the predefined set of $\mathcal{O}(1)$ graphs, the graph is random.

1.3 Complex String 'Paradox'

In exchange for having you believe me on how important the K-Complexity of a sequence is, we can show an interesting paradox. The paradox is as follows:

- Most strings are complex, but there exists no general program that can output a complex string.
- The key term here is general, if we were looking for a specific program, we could feed the required string into the Universal Turing Machine.

Let us begin by showing that most strings are complex:

- Start by considering a binary sequence 'x' with length 'n' (integer), and let 'e' be a program (binary).
- We will show the paradox using the probability of existence of a set of programs with complexity less than the length of the sequence itself.
- For the set of programs with length shorter than x, we get: $\{e \mid K(x) \leq n-k\}$, where k is an integer that defines the offset.
- Since we consider $K(x) \leq n-k$, we need all programs with length ranging from $1, 2, ..., n-k$. Therefore, the cardinality of this set can be given by:

$$|\{e \mid K(x) \leq n-k\}| = \sum_{i=1}^{n-k} 2^i = 2^{n-k+1}$$

- The set of programs such that $K(x) = n$, has a cardinality of 2^n.
- Putting this together, we get the probability of strings with K-complexity $\leq n-k$:

$$P(e|K(x) \leq (n-k)) = \frac{|\{e \mid K(x) \leq n-k\}|}{|\{e \mid K(x) = n\}|}$$

$$\implies \frac{2^{n-k+1}}{2^n} \implies 2^{-k+1} \implies \frac{1}{2^{k-1}}$$

- As $k \to \infty$, the probability $\to 0$. Therefore, as we increase 'k' and look for strings with lesser complexity, the probability of such strings existing goes to 0. From this we can conclude that most strings are complex.

For the second part, we have to show that there exist no program that can output a string with complexity $\geq n-k$.

- Consider a general program that for n large enough, outputs a string 'x' such that $K(x) \geq n - k$, where $k \lneq n$.
- Since this program works for n, the length of the program is constant, that is, $\mathcal{O}(1)$. Additionally, we have to send in 'n' as an input, which has complexity $\mathcal{O}(log(n))$ (any integer 'n' can be expressed in binary using $log_2(n)$ bits).
- Now the K-complexity of the algorithm is $\mathcal{O}(1) + log(n)$ (description + input), which is $\lneq n - k$, for large enough 'n', as linear terms grow faster than logarithmic terms.
- Our goal was to generate a complex string such that $K(x) \geq n - k$ (the length of the program $\geq n - k$), and we 'proved' that if we had such a program, the length of it would be $\lneq n - k$, which is a contradiction.
- Therefore, there cannot exist a program that outputs complex strings.

By both cases, we can conclude that most strings are complex, but there exists no program that can output complex strings. Hopefully this gives some insight into how something that is incomputable can be useful to us.

1.4 K-Complexity of Graphs

The K-Complexity of graphs can be measured using the adjacency matrix of the graph. For an undirected graph on n vertices, we need $\binom{n}{2}$ bits to represent the graph. We use '1' when there is an edge between 2 vertices and '0' if there is not. For K-Complexity, we use bitstrings from the columns of matrices.

For example, consider a graph on 5 vertices:
$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 2 & - & 0 & 1 & 0 & 0 \\ 3 & - & - & 0 & 1 & 1 \\ 4 & - & - & - & 0 & 1 \\ 5 & - & - & - & - & 0 \end{bmatrix}$$

For an undirected graph we need $\binom{n}{2}$ bits because the graph repeats itself in the bottom triangle of the matrix. This is due to the fact that if there exists (x,y), then in an undirected graph, there also exists (y,x). Thus the matrix is symmetric and we can consider the upper triangle which gives us the all the information we need to construct the graph.

There are 2 ways to extract bitstrings from the matrix, we can traverse row-wise or column-wise. From either, we get a sequence for which we look to find the K-Complexity.

The column-wise bitstring of the above graph is: 010010101010110
The row-wise bitstring of the above graph is: 010110100011010

For the rest of this paper, we will consider the column wise bitstring.

A graph is said to have $\mathcal{O}(1)$ complexity when the bitstring of the graph can be described by a program that does not require the entire bitstring be printed out. For example, describing a complete graph on 'n' vertices is shorter than printing the bits out for every single edge.

To clarify, compressing a graph by its Kolmogorov complexity does not entail deleting edges/vertices, rather, it revolves around finding a shorter description of the bitstring of the graph.

Therefore, if the K-complexity of the graph is the length of the bitstring itself, we can say that the graph is random.

2 Shift Coefficient

From [1], the shift coefficient of a labeled graph is defined as the minimum number of circular rotations required to replicate the original node labeling. Formally:

Let ∃ a labeled graph G = (V,E).

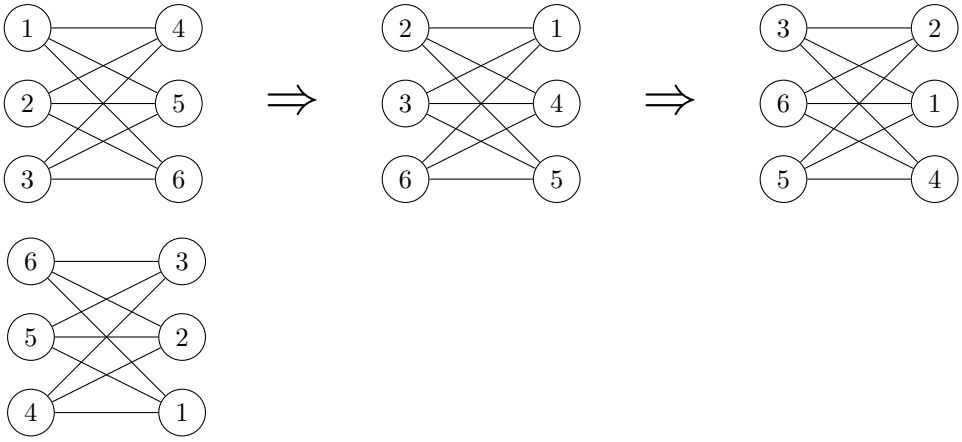
The Shift Coefficient, S(G) is the minimum number of rotations until we get the same adjacency matrix. Consider the number of iterations to be r, and $S^r(G)$ be the resulting graph after the r^{th} rotation, s.t.,

$S(G) = min\{r \mid S^r(G) = G\}$

From Lemma 1 in [1], the shift coefficient, S(G) always divides |V|.

2.1 Finding S(G)

Consider the maximal bipartite graph on 6 vertices:



After 3 clockwise shifts, we get the original graph back if we rearrange the vertices in each vertex class. This is how one would find the shift coefficient by 'holding' the edges. It might seem as though we are moving vertices around, but if you look at the edge set, with every iteration every edge (x,y) changes to (x+1, y+1).

When it comes to matrices, this is what the iteration looks like when we look at the adjacency matrix of a graph on 5 vertices:

Original: $\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 2 & - & 0 & 1 & 0 & 0 \\ 3 & - & - & 0 & 1 & 1 \\ 4 & - & - & - & 0 & 1 \\ 5 & - & - & - & - & 0 \end{bmatrix}$

After 1 rotation: $\begin{bmatrix} 0 & 5 & 1 & 2 & 3 & 4 \\ 5 & 0 & 1 & 0 & 1 & 1 \\ 1 & - & 0 & 1 & 0 & 0 \\ 2 & - & - & 0 & 1 & 1 \\ 3 & - & - & - & 0 & 1 \\ 4 & - & - & - & - & 0 \end{bmatrix}$

After 4 (n-1) rotations: $\begin{bmatrix} 0 & 2 & 3 & 4 & 5 & 1 \\ 2 & 0 & 1 & 0 & 1 & 1 \\ 3 & - & 0 & 1 & 0 & 0 \\ 4 & - & - & 0 & 1 & 1 \\ 5 & - & - & - & 0 & 1 \\ 1 & - & - & - & - & 0 \end{bmatrix}$

After 5 (n) rotations: $\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 2 & - & 0 & 1 & 0 & 0 \\ 3 & - & - & 0 & 1 & 1 \\ 4 & - & - & - & 0 & 1 \\ 5 & - & - & - & - & 0 \end{bmatrix}$

We rotated the vertex labels by 1 vertex, we can perform multiple rotations by adding the number of rotations at one and get the resulting matrix in one iteration. If we perform 5 (number of vertices) rotations, we get the original matrix back.

Our goal is to skip iterations, that is, directly shift by a factor of $|V|$ instead of shifting by 1 at a time. This gives us 2 cases:

If current_vertex + factor $\leq |V(G)|$*:*

vertex label = vertex label + factor

Else:

vertex label = (vertex label + factor) - $|V(G)|$

2.2 Algorithm

Input: Adjacency Matrix of the Graph

Output: Shift Coefficient of the Graph

Convert the matrix to a hash map, vertex_label:[edges] (1)

Calculates, sorts, and stores the factors in an array (2)

Iterate over the factors (3)

    Create a copy of the hash map (4)

    Iterate over the vertices (5)

        Shift the edges for each vertex based on the cases given above (6)

    If the shifted hash map = original dictionary (7)

        break

This algorithm depends only on vertices as we will see in the complexity analysis. This mitigates the need to depend on the edge set, making the algorithm faster as $|E| = |V|^2$ in the worst case.

2.3 Working of the Algorithm

The step-wise working of the algorithm is as follows:

1. Converts the matrix into a hash map, such that for a vertex v, the entry in the hash map is of the format:
   Vertex: Array of edges
2. For each entry/vertex in the hash map, we shift each entry of the edge by the factor of $|V|$ that we are currently iterating with.
   - Consider x: [0,1,0,1,0], where $x \in V(G),\ |V(G)| = 5$.
   - For a factor value of 1: we shift all entries by 1 index giving us x: [0,0,1,0,1].
   - Similarly, for a factor value 'f', we shift by 'f' indices.
3. After the reconstruction, we finally check if the reconstructed graph is the same as the original graph.

2.4 Complexity

Complexity breakdown: Let $|V| = n$

(1) $\leftarrow \mathcal{O}(n)$

(2) $\leftarrow \mathcal{O}\left(\sqrt{n} + \sqrt{n} \cdot log(\sqrt{n})\right)$

(3) $\leftarrow \mathcal{O}(\sqrt{n})$

- (4) $\leftarrow \mathcal{O}(n)$
- (5) $\leftarrow \mathcal{O}(n)$
  - (6) $\leftarrow \mathcal{O}(n)$
- (7) $\leftarrow \mathcal{O}(n^2)$

Final complexity:

$$\implies \mathcal{O}(n + \sqrt{n} + \sqrt{n} \cdot log(n) + \sqrt{n} \cdot (n + n^2 + n^2))$$

$$\implies \mathcal{O}(\sqrt{n} \cdot n^2)$$

$$\implies \mathcal{O}(n^{2.5}),\ where\ n = |V|$$

Now that we have covered the shift coefficient, we can move on to Basic Graphs.

3 Basic & Random Graphs

What is a Basic Graph?

- *From [1], a Basic Graph is a graph, that given the number of vertices 'n', has K-Complexity* $\mathcal{O}(1)$.
- *In other words, graphs that have a description shorter than the length of the bitstring of the adjacency matrix on a graph of 'n' vertices are Basic graphs.*

Theoretical Relation between S(G) & Basic Graphs:

- *Basic graphs have a shift coefficient with* $\mathcal{O}(1)$ *K-complexity. For a particular type of graph 'G', the S(G) is known. Consider a complete graph on 'n' vertices* K_n*, then* $S(K_n)\ \forall n \in \mathbb{N} = 1$.
- *The first S(G) columns of the adjacency matrix of G have* $\mathcal{O}(1)$ *K-Complexity. (Proof given in [1])*

Set of Basic Graphs

- *As stated above, for complete graphs,* $S(K_n) = 1\ \forall n \in \mathbb{N}$.
- *For Cyclic Graphs* $C_n,\ S(C_n) = 1\ \forall n \in \mathbb{N}$.
- *Star Graphs with density 2 (each vertex has degree 2), let* $S_n,\ S(S_n) = 1\ \forall n \in \mathbb{N}$.
- *Ladder Rung Graphs, let* $L_n,\ S(L_n) = 2\ \forall n \in \mathbb{N}$.
- *One-Edge Graphs, there is an edge between the first 2 nodes, let* $O_n,\ S(O_n) = n\ \forall n \in \mathbb{N}$.
- *Maximal Bipartite Graphs on even vertices,* $S(K_{\frac{n}{2},\frac{n}{2}}) = \frac{n}{2}\ \forall n \in \mathbb{N}\ s.t.\ n\%2 = 0$
- *Maximal Bipartite Graphs on odd vertices,* $S(K_{\frac{(n+1)}{2},\frac{(n-1)}{2}}) = n\ \forall n \in \mathbb{N}\ s.t.\ n\%2 = 1$

What is a Random Graph?

- *From [1], a random graph is a graph that requires* $\binom{n}{2}$ *bits to create the adjacency matrix of the graph. Random graphs are incompressible.*
- *A graph that needs less than* $\binom{n}{2}$ *bits to create the adjacency matrix is a non-random/compressible graph.*

Properties

- *Random graphs do not contain any basic subgraphs with more than* $2log_2(n)+4$ *vertices. [1] (This is a key result for the rest of the paper)*

4 Kolmogorov Graph Covering

4.1 Intuition

Keeping the properties of random graphs in mind, the intuition of the algorithm is as follows:

- let $|V| = n$
- Start with the complexity assuming the graph is random. This is given by:

$$complexity \leftarrow \binom{n}{2}$$

- We start an iterator 'i', which goes from $n \rightarrow 2log_2(n) + 5$ in decrements of 1.
- Our goal with 'i' is to create subgraphs of vertices from n, each of size 'i'. The number of subsets is:

$$|\{s \mid s \subseteq G, such\ that\ |s| = i\}| = \binom{n}{i}$$

- Then we check if these subgraphs are basic graphs, and look to replace their descriptions in the original bitstring with an $\mathcal{O}(1)$ description.
- After this, we delete the basic subgraph, update the complexity and move on to the next subgraph. We repeat this process until 'i' terminates or after deleting a subgraph, if $n \lneq 2log_2(n) + 5$.

How does the bitstring change:

Hopefully, this provides intuition on how the bitstring would change based on the execution of the algorithm:

- Consider a graph on 'n' vertices, such that $V = \{v_1, v_2, v_3, ..., v_n\}$.
- Now let the bitstring associated with a vertex v_i to be b_i $\forall i\ where\ 1 \leq i \leq n$. The set of all bitstrings, let B=$\{b_1, b_2, ..., b_n\}$.
- The bitstring of the graph 'G' itself is all $b_i's$ put together from $i = 1 \rightarrow n$.
- Pick a value of 'i' at random such that $1 \leq i \leq n$, and create $\binom{n}{i}$ subgraphs from $V(G)$, each of size 'i'.
- Pick a subgraph and assume that it is basic. Let that subgraph be 'C' where the bitstring of C is:

$$C = \{b_j, ..., b_k\}, where\ |C| = i,\ 1 \leq j,\ k \leq n\ \&\ C \subseteq B$$

- Since 'C' is basic, the bitstring can be represented with $\mathcal{O}(1)$ bits. Now, in the original set of bitstrings 'B', we can replace 'C' in 'B' with an $\mathcal{O}(1)$ description.
- WLOG we can apply this to all subgraphs of size 'i'. We follow the main idea of replacing 'C's that are basic with programs of $\mathcal{O}(1)$ length. If C is not basic, we move on to the next subgraph.
- Types of outputs:
  - **Fully Compressible**: The entire bitstring is of $\mathcal{O}(1)$ length. Put simply, the original graph itself is basic. K(x) = 0.
  - **Partially Compressible**: It is a combination of programs of $\mathcal{O}(1)$ length and $\sim \mathcal{O}(1)$ complexity. $0 \lneq K(x) \lneq length(x)$.
  - **Incompressible**: The graph is random. $K(x) = length(x)$.

4.2 The Covering Algorithm

This implementation based on the algorithm given in [1].

Input: Adjacency Matrix of the Graph, G = (V,E)

Output: An Upper Bound on the Kolmogorov Complexity of the Graph

let $|V| = n$

Initialize the complexity as $\binom{n}{2}$ *← complexity (1)*

Iterate from $i = n, n-1, n-2, \ldots, \lceil 2\log(n) \rceil + 5$*: (2)*

Create $\binom{n}{i}$ *subsets, each of size 'i' (3)*

Iterate over the subgraphs: (4)

Let current subgraph ← C (5)

Create a copy of the graph ← matrix_copy (6)

Create the adjacency matrix of the C by deleting V(C) from matrix_copy (7)

Create the bitstring for C ← B (8)

Run the RNN on B ← output (9)

If output ≥ Threshold value (if the subgraph is basic):

Delete the V(C) from the adjacency matrix of G (10)

complexity ← complexity − $\binom{|V(C)|}{2}$

if $n \leq \lceil 2\log_2(n) \rceil + 5$*:*

break

Create the set of subgraphs again from the reconstructed graph, with the updated value of n (11)

4.3 Working of the Algorithm

The algorithm given above builds on the Kolmogorov Covering algorithm given in [1]. The fundamentals are based off of the original algorithm, and the following are what has been added to complete it:

- **Checking if a subgraph is basic or not**:
  - We use an RNN (Recurrent Neural Network) to perform sequence matching to predict if the bitstring of a subgraph is basic or not. The RNN is trained on the set of basic graphs (from Section 3) and random graphs.
  - The RNN avoids having to manually write and perform checks for each basic graph.
  - A perk of using an RNN is that the data set is scalable, that is, adding a basic graph to the set of existing basic graphs is quite simple.
  - Since the patterns for basic graphs are obvious, the accuracy of the model is on the higher side.
  - The provided code uses a probability threshold value of 0.95. The rationale behind this was to be extremely confident in classifying the graph as basic.
- **Dynamically reducing the size of the graph**:
  - The intuition behind this is as follows: After deleting the subgraph from the original graph, we reset our bounds w.r.t to the new graph. Then we treat the new graph as the 'input' graph and delete subgraphs using the same intuition.
  - This significantly decreases the number of computations as we delete subgraphs of size $\geq \lceil 2log(n) \rceil + 5$. This reduces the number of subsets from $\binom{n}{i}$ to $\binom{(n-k)}{i}$, where $k > \lceil 2log(n) \rceil + 5$. The values of 'i' & 'k' are updated with each deleted subgraph.
  - This has no impact on the runtime of the algorithm if the graph is random, but if we do delete subgraphs, the runtime decreases by a significant factor.

4.4 Complexity

This needs an additional step; we need to consider the asymptotic growth of $\binom{n}{\frac{n}{2}}$, where $n = |V|$.

Consider an iterator 'i' that goes from n to $\lceil 2log(n) \rceil + 5$, for large 'n', asymptotically $\mathcal{O}(n) \geq \mathcal{O}(\frac{n}{2}) \geq \mathcal{O}(\lceil 2log(n) \rceil + 5)$. Now, $max\{\binom{n}{i}\}$ occurs at $i = \frac{n}{2}$, and the asymptotic growth rate is as follows:

From [2], we know that:

$$\pi = \left(\frac{(c!)^4 \cdot 2^{4c}}{c \cdot [(2c)!]^2}\right)\left[1 - \frac{1}{8c} + \frac{1}{128c^2} + \frac{5}{1024c^3} + \mathcal{O}(c^{-4})\right]^2$$

$$Where,\ c \in \mathbb{Z}^+$$

$$Rearranging\ terms:$$

$$\frac{[(2c)!]^2}{(c)!^4} = \frac{2^{4c}}{\pi \cdot c}\left[1 - \frac{1}{8c} + \frac{1}{128c^2} + \frac{5}{1024c^3} + \mathcal{O}(c^{-4})\right]^2$$

$$Take\ the\ sqaure\ root\ on\ both\ sides:$$

$$\frac{(2c)!}{(c)!^2} = \sqrt{\left(\frac{2^{4c}}{\pi \cdot c}\right)}\left[1 - \frac{1}{8c} + \frac{1}{128c^2} + \frac{5}{1024c^3} + \mathcal{O}(c^{-4})\right]$$

$$Rewriting\ \frac{(2c)!}{(c)!^2}\ as\ \binom{2c}{c}$$

$$\binom{2c}{c} = \sqrt{\left(\frac{2^{4c}}{\pi \cdot c}\right)}\left[1 - \frac{1}{8c} + \frac{1}{128c^2} + \frac{5}{1024c^3} + \mathcal{O}(c^{-4})\right]$$

$$Simplifying,\ we\ get:$$

$$\binom{2c}{c} = \left(\frac{2^{2c}}{\sqrt{\pi \cdot c}}\right)\left[1 - \frac{1}{8c} + \frac{1}{128c^2} + \frac{5}{1024c^3} + \mathcal{O}(c^{-4})\right]$$

$$Using\ 2^{2c} = 4^c,\ we\ get:$$

$$\binom{2c}{c} = \left(\frac{4^c}{\sqrt{\pi \cdot c}}\right)\left[1 - \frac{1}{8c} + \frac{1}{128c^2} + \frac{5}{1024c^3} + \mathcal{O}(c^{-4})\right]$$

$$We\ can\ conclude\ that:$$

$$\binom{2c}{c} \sim \mathcal{O}\left(\frac{4^c}{\sqrt{\pi \cdot c}}\right)$$

$$Which\ is\ equivalent\ to:$$

$$\binom{2c}{c} \sim \mathcal{O}\left(4^c \cdot c^{-\frac{1}{2}}\right)$$

$$For\ the\ asymptotic\ growth\ of\ \binom{n}{\frac{n}{2}},\ let\ c = \frac{n}{2}$$

$$We\ get: \binom{n}{\frac{n}{2}} \sim \mathcal{O}\left(4^{\frac{n}{2}} \cdot \left(\frac{n}{2}\right)^{-\frac{1}{2}}\right)$$

$$This\ can\ be\ simplified\ to: \binom{n}{\frac{n}{2}} \sim \mathcal{O}\left(2^n \cdot n^{-\frac{1}{2}}\right)$$

Now we wrap each chunk of complexity and then combine them together once we have all pieces. Again, for readability, assume $|V| = n$.

The line-by-line Complexity breakdown is as follows:

(1) contributes $\mathcal{O}(1)$ to the algorithm and can be ignored

(2) contributes $\mathcal{O}(n)$ to the complexity and all other computations are wrapped in 1

All computations after (2):

(3) contributes $\mathcal{O}\left(2^{n-k} \cdot (n-k)^{-\frac{1}{2}}\right)$ from the upper bound given above. $k = 0$ to begin with since we consider the entire graph at first, we then update the value of 'k' in (11).

Similarly, (4) contributes $\mathcal{O}\left(2^{n-k} \cdot (n-k)^{-\frac{1}{2}}\right)$.

All computations wrapped in (4):

(5) Puts the vertices from the subgraph 'C' into an array, contributing $\mathcal{O}(n)$.

(6) Creates a copy of the matrix, contributing $\mathcal{O}(n^2)$.

(7) Creates the adjacency matrix of the subgraph, this contributes $\mathcal{O}(n^3)$

(8) Creates the bitstring by iterating over the subgraphs' matrix: $\mathcal{O}(n^2)$

(9) Runs the RNN on the bitstring of the subgraph, since the LSTM size is constant, this contributes $\mathcal{O}(n)$

If the probability from the RNN meets/exceeds the threshold value:

The code below only runs when the subgraph is basic. After deleting the vertices, the size of the set of subgraphs reduces significantly and the algorithm converges/terminates faster.

(10) Deletes the subgraph from the graph, this contributes $\mathcal{O}(n^3)$

(11) Creates the new set of subgraphs with the updated value of 'n', contributing $\mathcal{O}\left(2^{n-k} \cdot (n-k)^{-\frac{1}{2}}\right)$.

The initial value: $k = 0$, and with every subgraph deleted, we update 'k' as such: $k \leftarrow k + |C|$.

Since we delete basic subgraphs of size at least $\lceil 2log(n) \rceil + 5$, deleting more subgraphs significantly improves the runtime.

This also updates the complexity of (3) and (4).

Putting everything together: In the worst case, when the graph is random, i.e. k = 0, the complexity of the algorithm can be modeled by:

- The complexity wrapped within (4) is:

$$\mathcal{O}(n^3)$$

- The complexity with (4) is:

$$\mathcal{O}\left(2^n \cdot n^{-\frac{1}{2}} \cdot n^3\right)$$
$$\implies \mathcal{O}\left(2^n \cdot n^{2.5}\right)$$

- The complexity wrapped within (2) is:

$$\mathcal{O}\left(n \cdot \left(2^n \cdot n^{2.5}\right)\right)$$
$$\implies \mathcal{O}\left(2^n \cdot n^{3.5}\right)$$

- The final complexity when the graph is random is:

$$\mathcal{O}\left(2^n \cdot n^{3.5}\right)$$

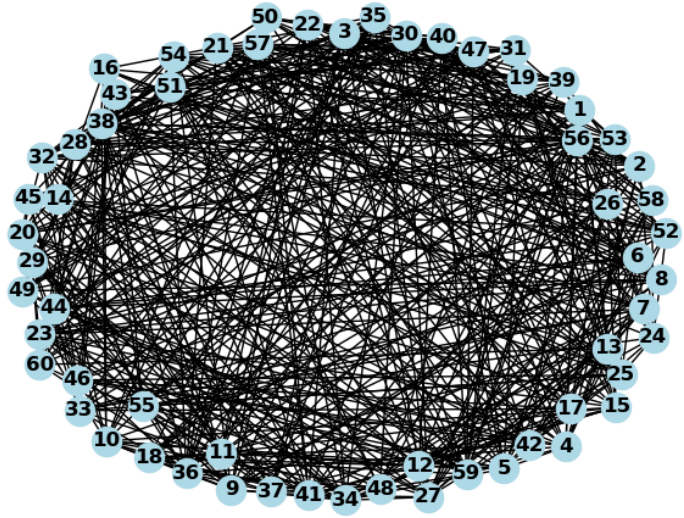
- We can derive a better/more precise asymptotic growth rate for the algorithm using 'k' as defined in the line-by-line breakdown. That would change: $\mathcal{O}\left(2^n \cdot n^{-\frac{1}{2}}\right) \rightarrow \mathcal{O}\left(2^{n-k} \cdot (n-k)^{-\frac{1}{2}}\right)$. We would update the value of 'k' as we delete subgraphs and as 'k' increases, the complexity decreases.
- Note: The part of the code that displays the graph can be made more efficient by using the labeled adjacency matrix of the graph. Using the labeled adjacency matrix should improve the time and space complexity but have no effect on the calculation of the complexity.

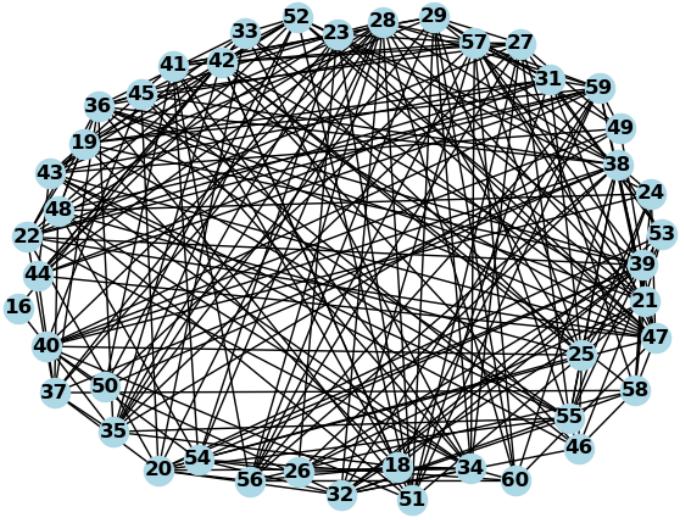
4.5 Workarounds/Alternatives

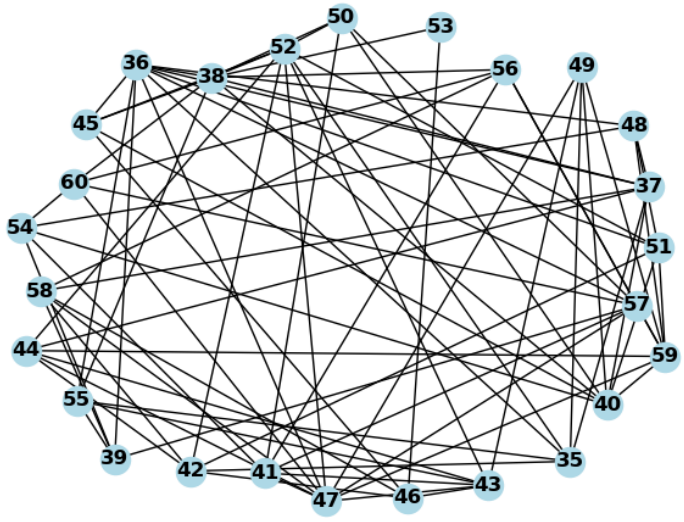
- In practice, the algorithm is painfully slow as its complexity is worse than exponential time.
- To avoid this, instead of creating and checking the subgraphs of size: $i = n, n-1, n-2, ..., \lceil 2log(n) \rceil + 5$, we can reconstruct the graph from $i = \lceil 2log(n) \rceil + 5, \lceil 2log(n) \rceil + 6, ..., n-1, n$.
- This gives us a worse upper bound on the Kolmogorov Complexity as opposed to the original algorithm, but the algorithm converges significantly faster.
- In the outputs section, we can see how the basic subgraphs are being deleted when we reconstruct the graph from $i = \lceil 2log(n) \rceil + 5$ up to n.

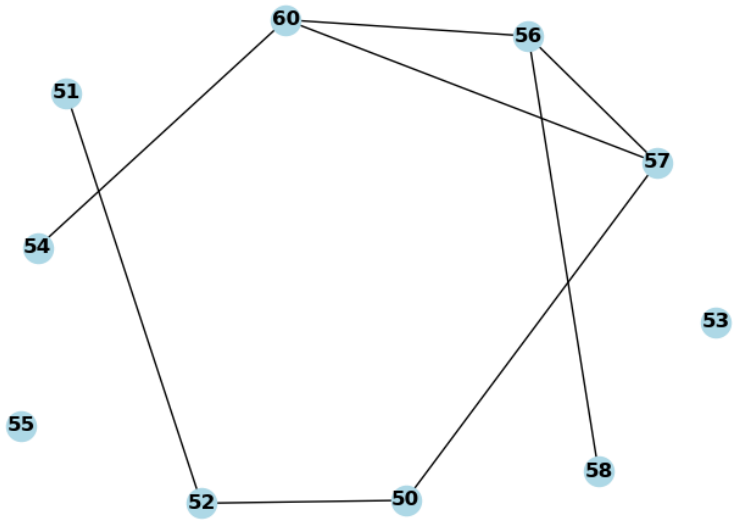
4.6 Visualizing the Execution of the Algorithm

Consider a graph $G \in \mathcal{G}(60, 0.3)$, which is quite a dense graph. We iterate from $2log(n) + 5 \rightarrow n$ as the algorithm converges in seconds. For reference, when iterating from $60 \rightarrow 2log(60) + 5$, my computer ran out of memory after 2 days of execution.

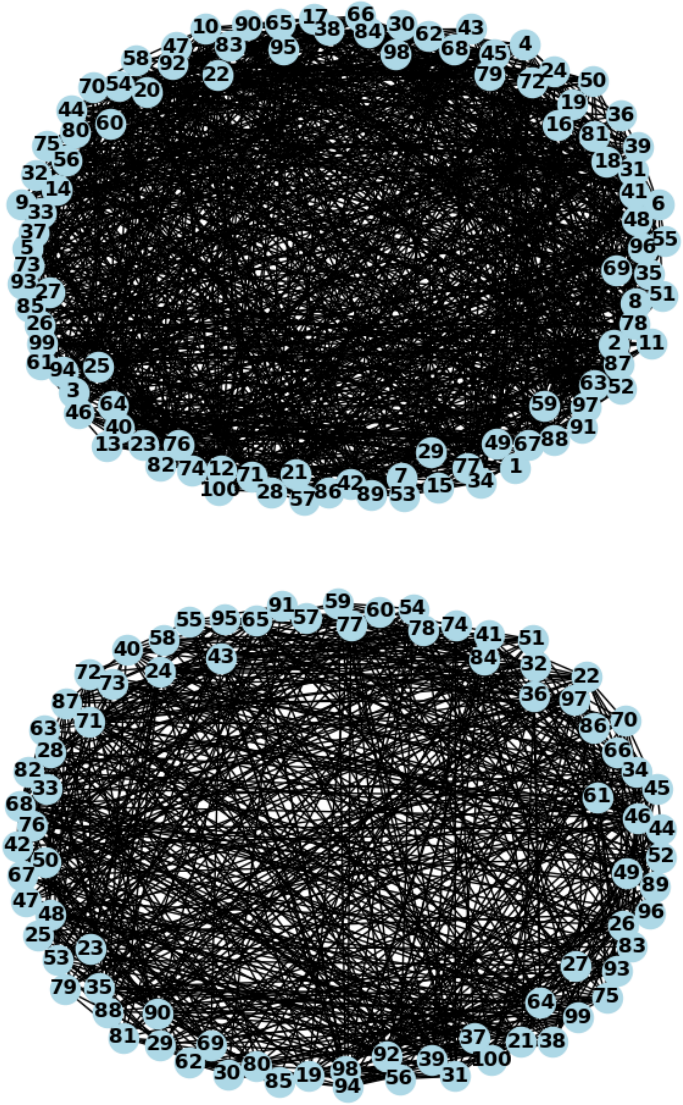


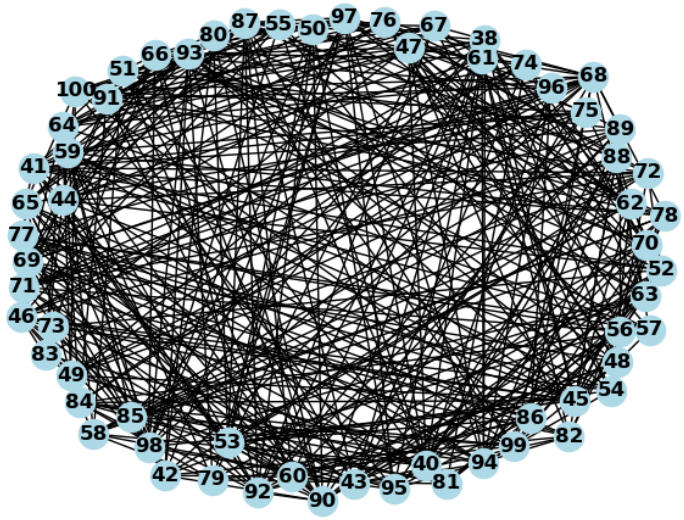


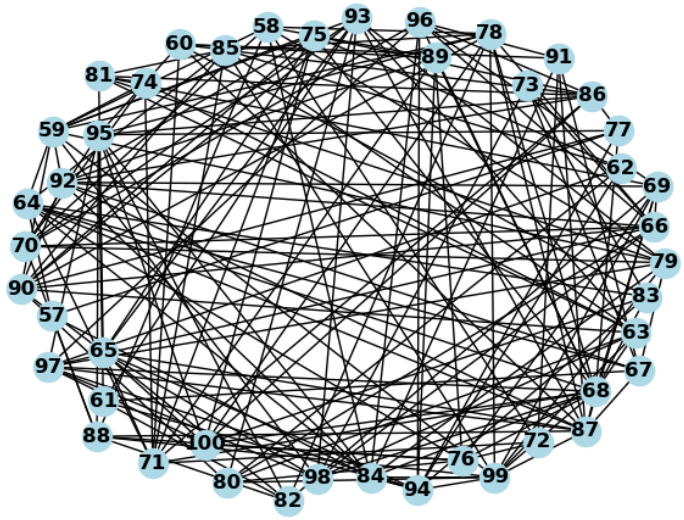

50
53
52
36
38
56
49
45
48
60
37
54
51
58
57
44
59
55
40
39
42
41
35
43
47
46

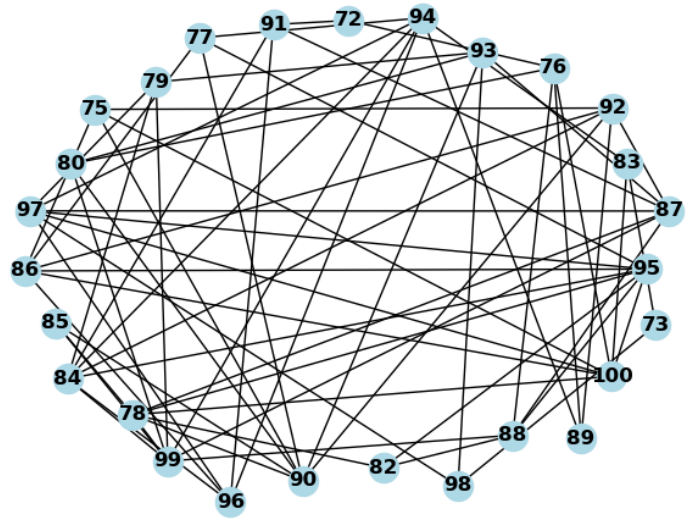

60
56
51
57
54
53
55
58
52
50

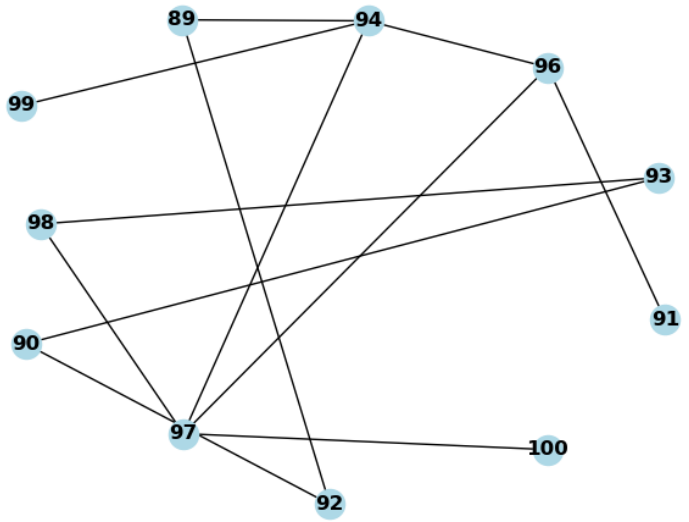
For another example, consider a graph $G \in \mathcal{G}(100, 0.2)$, which, again, is quite a dense graph. Despite the number of vertices, the algorithm converged in seconds.




97 76 67 38 87 55 50 80 47 61 74 66 93 51 68 100 91 96 75 64 89 59 88 41 72 65 44 62 78 77 70 69 52 71 63 46 73 56 57 83 48 49 54 84 45 85 86 58 98 53 82 99 42 79 40 94 60 43 81 92 95 90


93 96 58 75 78 60 85 89 91 81 74 73 86 59 95 77 92 62 69 64 66 70 79 90 83 57 63 65 67 97 61 68 88 87 72 100 71 76 80 98 84 99 94 82





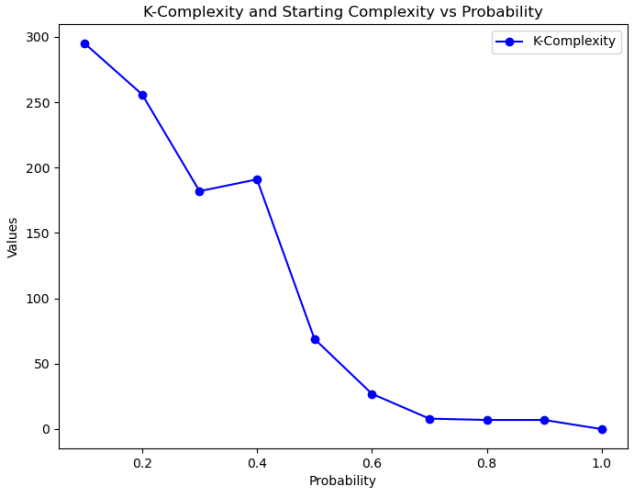
Note: I could not find the maximum number of vertices that my computer could handle. I went up to graphs on 1000 vertices and it converged in a couple of minutes. I am confident to conclude that even though iterating from $\lceil 2log(n) \rceil + 5 \rightarrow n$ gives us a worse upper bound, the faster runtime makes the trade off worth it.

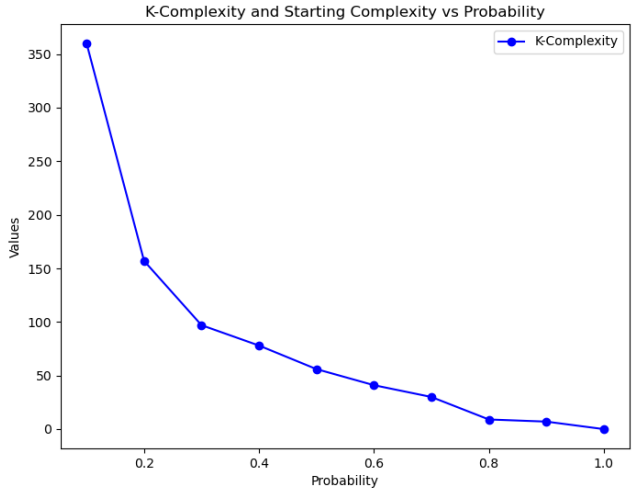
4.7 K-Complexity Plots from $n \to \lceil 2log(n) \rceil + 5$

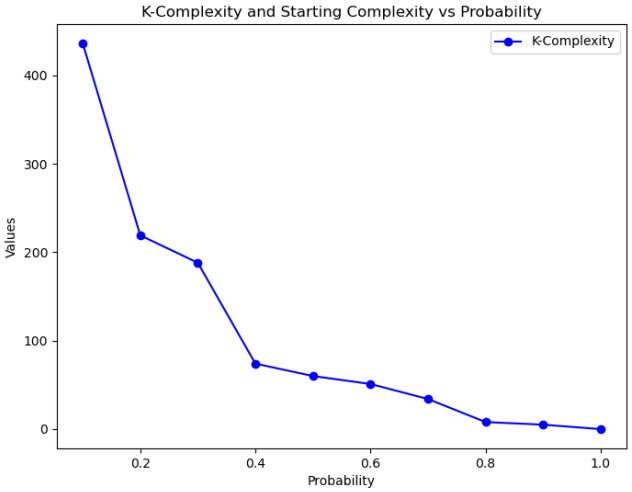
- y-axis: Mean of the K-Complexity calculated over 30 graphs at a set probability.
- x-axis: The probability at which the K-Complexity has been calculated, in increments of 0.1.

The graphs given below are as follows:

- Plot of K-Complexity vs Probability for graphs on 15 vertices.
- Plot of K-Complexity vs Probability for graphs on 20 vertices.
- Plot of K-Complexity vs Probability for graphs on 30 vertices.







Inferences:

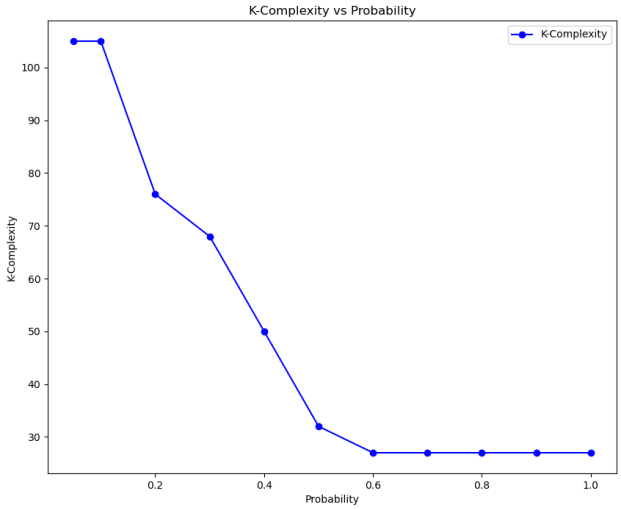
- Given a graph $G \in \mathcal{G}(n,p)$ as p increases, the K-Complexity of the graph decreases. Intuitively, this makes sense as $p \to 1,\ G \to K_n$.
- After running tests, a common result is that smaller graphs (n $\approx$ 20) are generally basic.
- The computing power and memory available to me restricted me to graphs on around 30 vertices. My machine ran out of memory to store the subgraphs for $n \geq 40$. This is why reconstructing the graph, from $\binom{n}{2log(n)+5}$ subgraphs to n makes sense as smaller graphs are generally compressible.

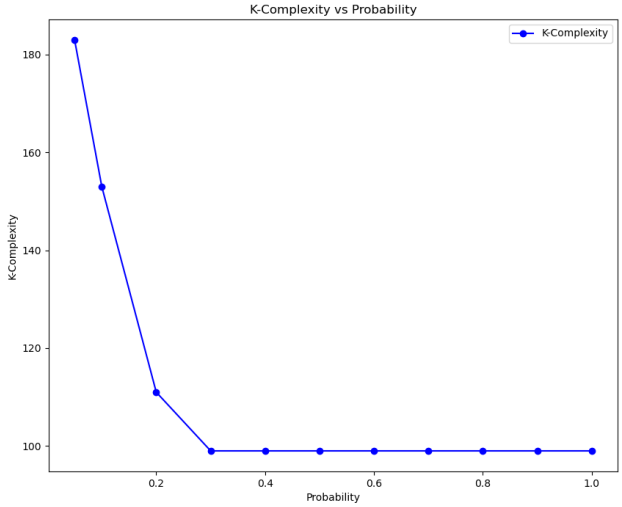
4.8 K-Complexity Plots from $\lceil 2log(n) \rceil + 5 \rightarrow n$

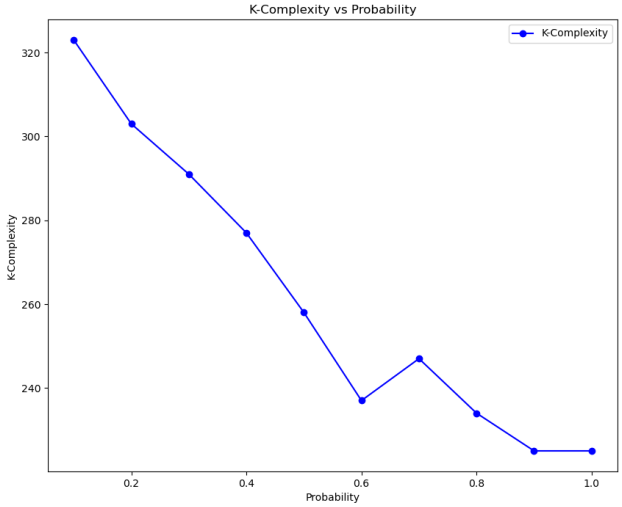
- y-axis: Mean of the K-Complexity calculated over 30 graphs at a set probability.
- x-axis: The probability at which the K-Complexity has been calculated, in increments of 0.1.

The graphs given below are as follows:

- Plot of K-Complexity vs Probability for graphs on 15 vertices.
- Plot of K-Complexity vs Probability for graphs on 20 vertices.
- Plot of K-Complexity vs Probability for graphs on 30 vertices.







Inferences:

- In comparison with the previous graphs, iterating up to 'n' vertices does give us a worse upper bound, but the convergence is significantly faster.
- I would have liked to include plots to compare the number of the subgraphs iterated over between both approaches and the K-complexity (have both plots on the same graph). This would have allowed for better comparison, but given the deadline for this project, I was not able to compute the required materials to create these plots.

5 References

[1] Farzaneh A, Coon JP, Badiu M-A. Kolmogorov Basic Graphs and Their Application in Network Complexity Analysis. Entropy. 2021; 23(12):1604. https://doi.org/10.3390/e23121604

[2] Luke, Yudell L. (1969). The Special Functions and their Approximations, Vol. 1. New York, NY, USA: Academic Press, Inc. p. 35.

[3] Li, Ming & Vitányi, Paul. (2019). An Introduction to Kolmogorov Complexity and Its Applications. 10.1007/978-3-030-11298-1.