Maximal Independent Set

Deniz Can Gezgin (27992) - Sueda Şeker (28325)

May 27, 2023

---

1 Problem Description

a. Definition and applications

Intuitively, independent set of a graph is a set of vertices of a graph in which no two vertices are adjacent to each other. Maximal independent set of a graph is the independent set with the largest number of vertices.

Formally, an independent set S of graph G = (V, E) is a subset S of V such that no two vertices in S are joined by an edge in E. An independent set that is not a proper subset of another independent set is called maximal or maximum. As a result the maximal independent set of a graph is the largest cardinality subset W of V such that no two vertices in in W is joined by an edge in E.

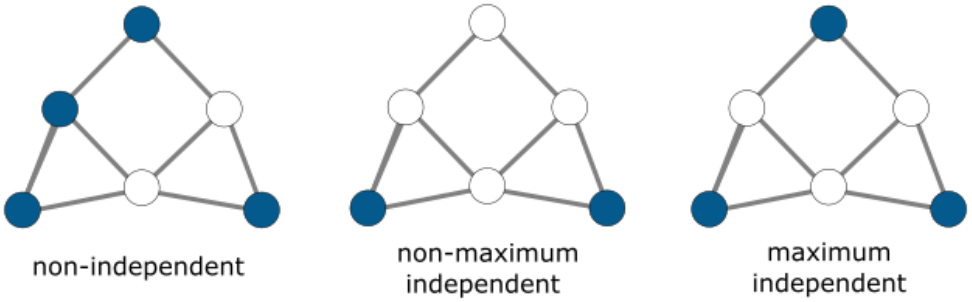


Figure 1: Maximal and non-maximal independent sets [5]

Maximal independent set problem has many practical applications in optimization problems in bio-informatics. For example it is used for finding maximally complementary sets of donor/acceptor pairs in macro-molecular dockings.[1]

b. Hardness

Decision version of maximal independent set problem is NP-complete. However maximal independent set problem is considered to be NP-Hard. To prove the hardness of problem we can find a polynomial-time reduction from any known NP-hard problem to our original problem.

We will use Max Clique in our reduction which is the optimization version of Clique problem. Clique is one of Karp's 21 problems shown in his paper "Reducibility Among Combinatorial Problems"[2]. Max clique is the highest cardinality clique of graph G and it is a well known NP-Hard problem [3]. Max clique problem tries to find the maximum number of pairwise adjacent nodes in a graph whereas the maximal independent set problem tries to find maximum number of pairwise non-adjacent nodes.

To reduce a Clique Problem to Independent Set problem for a given graph $G = (V, E)$ construct the complement $G' = (V, E')$ where E' is all the possible edges with V that are not present in E. Now let us prove two way implication for our reduction.

(=>) (a). If W is the maximum cardinality independent set in the graph G', it implies that no vertices in W share an edge in G', which in turn implies that all of these vertices share an edge with the others in G, forming a clique.(b). To see that this is also the a max clique, we can use a proof by contradiction. Lets say k is the cardinality of maximal independent set in the graph G'. Now assume the clique was not of maximum cardinality. So there must be a clique that has one more vertex, which implies that G' has an independent set with $k + 1$ vertices (using part a of the other proof). However, this creates a contradiction as k is the maximum cardinality independent set in the graph G'.
(<=) (a). If W is the max clique in the graph G, it implies that all vertices in W share an edge in G, which in turn implies that all of these vertices do not share an edge with the others in G', forming a independent set.(b). To see that this is also the maximum cardinality independent set, we can use a proof by contradiction. Lets say k is the number of vertices of max clique in the graph G. Now assume the independent set was not of maximum cardinality. So there must be a independent set that has one more vertex, which implies that G has clique with $k+1$ vertices (using part a of the other proof). However, this creates a contradiction as k is the number of vertices of max clique in the graph G.

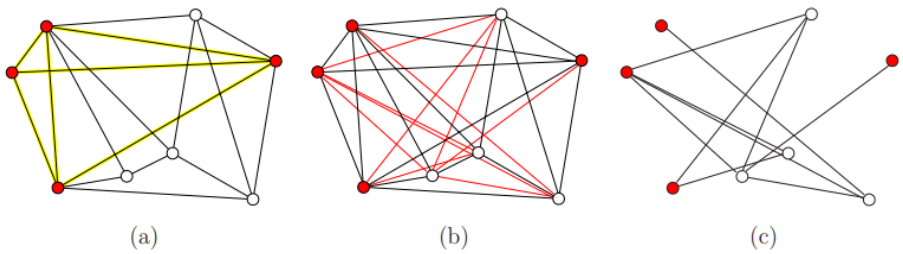


Figure 2: (a) A clique in a graph G, (b) the complement graph is formed by all the edges not appearing in G, and (c) the complement graph and the independent set corresponding to the clique in G [3].

We have shown max clique and maximal independent set can be reduced to each other using a complement graph. Now let us show that the complement can be constructed in polynomial time. To calculate $G' = (V, E')$ from $G = (V, E)$ we need to calculate E'. To calculate it we need to find all the possible edges that can be constructed using set V. All the possible edges is nothing but $\binom{v}{2}$ which can be calculated in polynomial time. After we can also subtract E from $\binom{v}{2}$ in polynomials time since size of $\binom{v}{2}$ is also polynomial. Therefore the maximal independent set problem is is NP-Hard.

2 Algorithm Description

a. Brute Force Algorithm

We will use a brute force recursive algorithm that we found from the internet [4]. Although this algorithm is capable of providing the correct solution each time, it has an exponential time complexity of $O(2^V)$ for graph G(V,E), making it quite inefficient in terms of runtime. Recursive calls make the algorithms very inefficient however as it will become apparent when we discuss the correctness proof this is necessary to ensure a guaranteed solution.

Note that this algorithm uses **divide and conquer** algorithmic paradigm. The problem is divided into smaller sub-problems using subgraphs in each recursive call until we get a small enough grapgh that is of size 0 or 1 which we can trivially solve. After recursive calls solutions are combined. The divide, conquer and combine steps of the paradigm will became apparent after full description of algorithm. In simple terms in order to find solution, we start with a vertex. We create two possible maximal sets - one including this vertex and the other not including it. We calculate the maximal independent sets of subgraphs in both cases recursively and return the higher result after combining.

The more detailed description of the function that solves this problem is as follows:

1. Start with a Graph G(V,E) and select a vertex $v \in V$ (We select this as the first vertex).
2. If the number of vertices in G is 0, return an empty array.
3. If the number of vertices in G is 1, return v.
4. Calculate the maximal independent set where we include v in our set. Call this set S_1.
   (a) This calculation can be made by creating a subgraph G' of G by deleting vertex v and all of its neighbors from G.
   (b) Recursively call the function to calculate the maximal independent set of G'. The set v + independent set of G' is the maximal independent set where we include v.
5. Calculate the maximal independent set where we don't include v in our set. Call this set S_2.
   (a) This calculation can be made by creating a sub-graph G' of G by deleting vertex v from G.
   (b) Recursively call the function to calculate the maximal independent set of G'. The independent set of G' is the maximal independent set of G where we don't include v.
6. Return the larger of S_1 and S_2.

Steps 2 and 3 make up our base cases for recursion.

b. Heuristic Algorithm

We will use a polynomial time greedy algorithm that we implementing following the description highlighted in a paper by D. Gainanov et al.[6]. The algorithm is a **greedy algorithm** that can also provide an estimate value for the found solution. The algorithm is a greedy algorithm because it tries tries to locally minimize the estimate value while covering maximum number of vertices in each step. This is done by Min-Maxing m and k parameters in each step. The k parameter that we maximize is defined for a vertex v by counting the number of neighbors of v that are still in the set of applicants for inclusion in the independent set. In each step we also remove neighbors of v from V_0 since independent set can not contain any of them. Hence by maximising k we remove max number of vertices from V_0. On the other hand the m parameter that we try to minimize is defined as number of missing edges in the neighborhood of v. We minimize m since parameter m adds up to estimate for our solution. Closer this estimate is to zero the better our solution. We use m as an estimate because of the following property of m highlighted in the paper.
If the vertex v is a (k, m)-vertex in the graph $G = (V, E)$, then there exists a maximal independent set $S \subseteq V$ such that $v \in S$. The equation for the ratio bound can be expressed as:

$$|S| \geq \max S(G) - m \tag{1}$$

Based in this selecting the vertex with smallest m gets will provide the solution S such that cardinality of solution S will be closer to cardinality of maximum independent set of G. Moreover it can be seen if Est takes the value 0 for a solution S, the solution S is guaranteed to be the maximum independent set of G. The proof for this property will be detailed in ratio bound section as m is also used for ratio bound.
It is important to note that Min-Maxing is done by selecting vertex with the maximum k among vertices with the minimum. Hence minimizing m has a higher precedence.

1. Start with a Graph G(V,E) and initialize the current set of vertices V_0 with all vertices in the graph V. This set represents the applicants for inclusion in the independent set.
2. Initialize an empty set S to store the vertices selected for the independent set.
3. Initialize the maximum M value encountered to 0. Max m value is used for ratio bound.
4. Initialize the estimate Est to 0. This estimate represents the accumulated value of m[v] for the selected vertices. Est provides and estimate for deviation from the exact solution
5. Start a loop that continues until the set of applicants V_0 is not empty.
   (a) For each vertex v in the set of applicants V_0, perform the following steps:
      i. Calculate the k parameter for vertex v by counting the number of neighbors of v that are still in the set V_0.
      ii. Calculate the m parameter for vertex v by counting the number of missing edges in the neighborhood of v.
   (b) Select a vertex v_0 as the next candidate for inclusion in the independent set greedily by Min-Maxing m and k values. Among vertices with the minimum m the vertex with the maximum k is selected.
   (c) Add the selected vertex v_0 to the independent set S.
   (d) Update the estimate Est by adding the m value of the selected vertex v_0.
   (e) Update the maximum m value if needed by comparing to m value of the selected vertex v_0.
   (f) Update the set of applicants V_0 by removing the selected vertex v_0 and its neighborhood from V_0.
   (g) Repeat the loop until the set of applicants V_0 becomes empty.
6. Return the final value of independent set and estimate.

The algorithm is a maximization problem where the size of independent set is trying to be maximized. Thus, given $C*$ corresponding to optimal solution and C corresponding to approximate solution, and $\rho(n)$ corresponding to ratio bound with input size n bound can be expressed as follows:

$$\frac{C^*}{C} \leq \rho(n) \tag{2}$$

We will use the following proposition by Gainanov to prove a ratio bound for the algorithm.
If the vertex v is a (k, m)-vertex in the graph $G = (V, E)$, then there exists a maximal independent set $S \subseteq V$ such that $v \in S$. The equation for the ratio bound can be expressed as:

$$|S| \geq \max S(G) - m \tag{3}$$

So let us first demonstrate prove for this proposition as highlighted in Gainanov's paper. We will use the **Proposition 1**, **Proposition 2:** and **Corollary 2** from Gainanov's paper. They will be given below without proof. Please refer back to Gainanov's paper for their proof [6].
Corollary 2: Let $\{e_1, \ldots, e_k\}$ be a subfamily of k vertex pairs that are not edges of the graph $G = (V, E)$. And let the graph $G_0 = (V_0, E_0)$ be such that $V_0 = V$ and $E_0 = E \cup \{e_1, \ldots, e_k\}$. Then

$$\max S(G) \geq \max S(G_0) \geq \max S(G) - k.$$

Proposition 1: If the vertex v is a k-vertex in the undirected graph $G = (V, E)$, then there exists the maximum independent set of vertices $S \subseteq V$ such that $v \in S$.
Proposition 2: Let two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be such that $V_1 = V_2$ and $E_1 \subseteq E_2$. Then $S_{\max}(G_2) \subseteq S(G_1)$.
Using **Corollary 2:**, **Proposition 1** and **Proposition 2:** one can prove the propriety we introduced earlier as follows.
According to the definition, there are missed m edges in the neighborhood of the vertex v to be the complete induced subgraph. Suppose these edges are $\{e_1, \ldots, e_m\}$. Then the vertex v is a k-vertex in the graph G_0, which is obtained from the graph G by the addition of these m edges $\{e_1, \ldots, e_m\}$.
According to **Proposition 1**, there exists $S_0 : S_0 \in S_{\max}(G_0)$ such that $v \in S_0$.
According to **Corollary 2:**, we have:

$$|S_0| = \max S(G_0) \geq \max S(G) - m.$$

It follows from **Proposition 2:** that $S_0 \in S(G)$. By definition, there exists a maximal independent set S such that $|S| \geq |S_0|$ and, as a consequence:

$$|S| \geq |S_0| \geq \max S(G) - m,$$

as was to be proved.
From this theorem we know that max S(G) corresponds to C*, and $|S|$ corresponds to C that we introduced earlier. Then by substation we have,

$$\begin{aligned} C^* - m &\leq C \\ C^* &\leq C + m \\ \frac{C^*}{C} &\leq 1 + \frac{m}{C} \end{aligned} \tag{4}$$

Using this form, we can define the ratio bound $\rho(n)$ can be selected as $1 + \frac{m}{C}$. Where m denotes the maximum m value that we encounter during execution.

3 Algorithm Analysis

a. Brute Force Algorithm

Correctness Proof

To prove correctness of an algorithm we need to prove it always terminates, and it always returns correct answer for all the inputs. In our case we need the show algorithm returns a subset W of V where W is maximal independent set of G(V,E). Since our function uses vertices as input our prof must be valid for any number of vertices.

Theorem 1: The algorithm returns maximal independent set of vertices for all input values of v.

Base case: If the graph G has 0 vertices, the algorithm returns an empty array, which is the correct independent set of vertices. If the graph has 1 vertex the algorithm returns that vertex which is the the correct independent set of vertices.

Inductive hypothesis: Assume that the algorithm is correct for all input sets V with $|V| = k$, where k is some positive integer greater than 1.

Inductive step: Consider a graph G with k+1 vertices. Let v be a vertex in G. Then, by the definition of an independent set, either v is in the independent set or not in the independent set. The algorithm considers both cases:

Case 1: The algorithm includes v in the independent set. In this case, the algorithm recursively calculates the maximal independent set of the subgraph G' obtained by deleting v and its neighbors from G. By the inductive hypothesis, this returns the correct independent set S' of G'. The set $S = v \cup S'$ is an independent set of G, since v is not adjacent to any vertex in S' by construction.

Case 2: The algorithm does not include v in the independent set. In this case, the algorithm recursively calculates the maximal independent set of the subgraph G' obtained by deleting v from G. By the inductive hypothesis, this returns the correct independent set S' of G'. Since v is not included in S', S' is also an independent set of G.

Finally, the algorithm returns the larger of S and S', which is a valid independent set of G by definition.
Therefore, the algorithm always returns a valid independent set of vertices.

Theorem 2: The algorithm terminates after a finite number of steps.

We will prove this by showing that the size of the graph decreases in each recursive call until the base cases are reached.

Suppose that the algorithm is called with a graph G with k vertices. In each recursive call, the algorithm deletes at least one vertex from G, either v and its neighbors or just v. Since G has a finite number of vertices, the algorithm must terminate after a finite number of recursive calls, at which point one of the base cases is reached.
Therefore, the algorithm terminates after a finite number of steps.

Since the algorithm always returns a valid independent set of vertices and terminates after a finite number of steps, we have proved that the algorithm is correct.

Complexity Proof

Let $T(n)$ be the time complexity of the algorithm on a graph with n vertices. We will use induction to prove an upper bound for $T(n)$

Theorem: We will use induction to prove that $T(n) \leq c \cdot 2^n$ for all graphs with n vertices, where c is some constant.

Base case: When $n = 0$, the algorithm returns an empty array. This takes constant time, so $T(0) = O(1)$. Similarly, when $n = 1$, the algorithm returns a single vertex, which also takes constant time. Therefore, $T(1) = O(1)$.

Inductive hypothesis: Assume that the algorithm's time complexity is $T(k) \leq c \cdot 2^k$ for all graphs with k vertices, where c is some constant.

Inductive step: Now we will show that the algorithm's time complexity is $T(n) \leq c \cdot 2^n$ for all graphs with n vertices. We can split this into two cases:

Case 1: When we include vertex v in our maximal independent set. In this case, we create a subgraph G' by deleting vertex v and all of its neighbors, which takes $O(n)$ time. The subgraph G' has at most $n - 1$ vertices. By the inductive hypothesis, calculating the maximal independent set of G' takes at most $c \cdot 2^{n-1}$ time. Therefore, the time complexity of this case is $c \cdot 2^{n-1} + O(n)$.

Case 2: When we do not include vertex v in our maximal independent set. In this case, we create a subgraph G' by deleting vertex v, which takes $O(n)$ time (even though deletion is $O(1)$ we have to make copy of graph to have a subgraph). The subgraph G' has at most $n - 1$ vertices. By the inductive hypothesis, calculating the maximal independent set of G' takes at most $c \cdot 2^{n-1}$ time. Therefore, the time complexity of this case is $c \cdot 2^{n-1} + O(n)$.

After we take the larger of the two maximal independent sets which is $O(1)$. The overall time complexity is the maximum of the two cases, which is $c \cdot 2^{n-1} + O(n)$. We can simplify this as $c \cdot 2^{n-1}$. Therefore, $T(n) \leq c \cdot 2^n$, completing the inductive step.

We have shown that the algorithm's time complexity is $T(n) \leq c \cdot 2^n$ for all graphs with n vertices. Therefore, the algorithm runs in exponential time and complexity can be described as $O(2^n)$.

b. Heuristic Algorithm

Correctness

We will prove algorithm correct (or a feasible heuristic algorithm). We have already shown a ratio bound for our algorithm in the previous section. We will also show algorithm always terminates and always returns an independent set.

Theorem: Algorithm always returns an independent set and terminates.
The vertices added to set S in each iteration of the while loop are independent. The final set S obtained after the algorithm terminates is an independent set.

Property 1: Vertices added to S in the loop are independent Let v_0 be the vertex selected in iteration of the while loop. When v_0 is added to S, its neighbors are removed from V_0. Therefore, v_0 and its neighbors are not connected to each other in the independent set S. Hence, the vertices added to S in each iteration are independent.

Property 2: The final set S created in the loop is and independent set and loop always terminates. When the algorithm terminates, V_0 becomes empty, and all vertices have been removed from V_0 and their neighborhoods. We start with constant amount of vertices in v_0 and we remove at least one each iteration therefore we end up with 0 vertices eventually. And as Property suggest at each iteration we add independent vertices to S. Therefore, the final set S obtained after the algorithm terminates is an independent set.

Hence our initial claim is proven.

Complexity

The complexity of the given algorithm is $O(V^3)$. We have loop that runs for V times in step 4. Under this loop we there are four non-trivial steps. We calculate k value by counting the number of neighbors of current vector, which takes O(V) time. We calculate m value by counting the number of missing edges in the neighborhood of current vector. This count is done in a nested loop that loops on all neighbors of neighbors of initial vector. Therefore this step takes $O(V^2)$ time. Also we do min-max of parameters by searching all the vertices in v_0 which takes O(V) time. Lastly we remove current vertex and its neighborhood from v_0 which takes O(V) time, Highest order bound for the first loop is $O(V^2)$ and since first loop runs V times as stated algorithm runs in $O(V^3)$ time complexity.

4 Sample Generation (Random Instance Generator)

nx.Graph() creates an empty undirected graph using the ***NetworkX*** library. The two loops generate edges between the nodes of the graph. The outer loop iterates over all nodes, while the inner loop iterates over nodes that come after the current node. This ensures that each edge is only added once, and that the graph is undirected. Inside the nested loop, the ***if*** statement generates a random number between ***0 and 1***. If this number is less than ***edge probability***, an edge is added between the current node and the node being considered in the inner loop. Finally, the function returns the generated graph.

```python
def generate_random_graph(num_vertices, edge_probability):
    graph = nx.Graph()
    graph.add_nodes_from(range(num_vertices))
    for i in range(num_vertices):
        for j in range(i+1, num_vertices):
            if random.random() < edge_probability:
                graph.add_edge(i, j)
    return graph
```

Listing 1: Random Instance Generator

5 Algorithm Implementations

a. Brute Force Algorithm

We modified a brute force solution form internet to work with networkx library. The algorithm exactly follows description in part 2.

```python
def graphSets(graph):

    # Base Case - Given Graph has no nodes
    if(graph.number_of_nodes() == 0):
        return []

    # Base Case - Given Graph has 1 node
    if(graph.number_of_nodes() == 1):
        return [list(graph.nodes())[0]]

    vCurrent = list(graph.nodes())[0]

    # Case 1 - Proceed removing the selected vertex from the Maximal Set
    graph2 = graph.copy()
    graph2.remove_node(vCurrent)
    res1 = graphSets(graph2)

    # Case 2 - Proceed considering the selected vertex as part of the Maximal Set
    for v in graph.neighbors(vCurrent):

        if(graph2.has_node(v)):
            graph2.remove_node(v)

    res2 = [vCurrent] + graphSets(graph2)

    if(len(res1) > len(res2)):
        return res1
    return res2


```

Listing 2: Brute Force Algorithm

We have done 18 tests in total in 3 batches. Batches had edge probabilities of 0.45, 0.30 and 0.15 and vertex amounts of 5, 10 and 15. The algorithm worked in each case without any problems. Below you can find our test samples of the aforementioned function with random sample generation code. The results are visualised using mathplotlib and raw outputs are also given under each visualization. Moreover edges of each graph are listed after the figures.

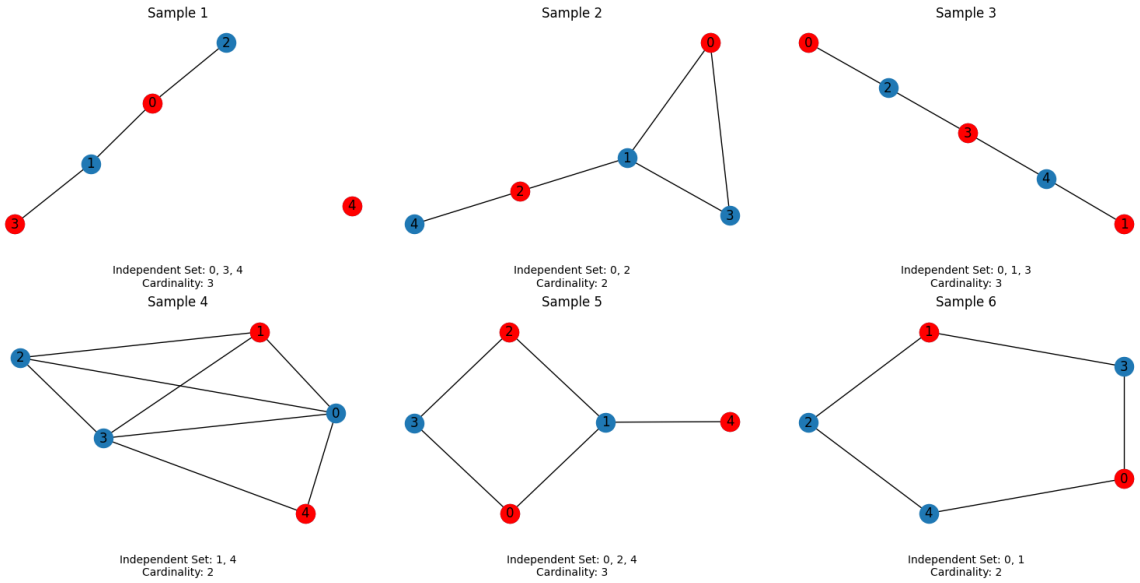


Figure 3: Test cases with vertex size = 5 and edge probability = 0.45

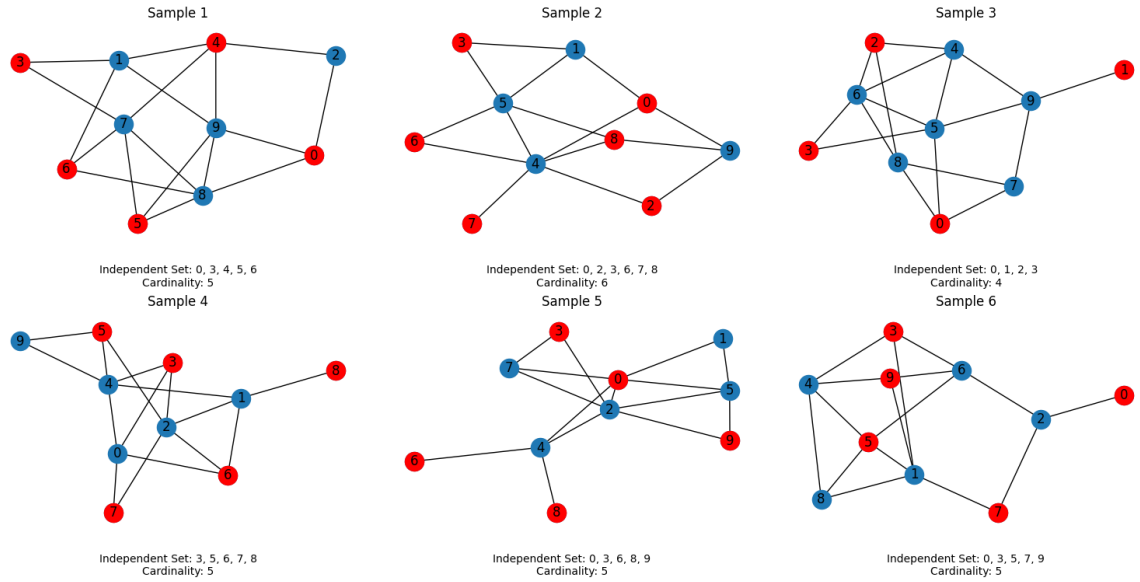


Figure 4: Test cases with vertex size = 10 and edge probability = 0.30

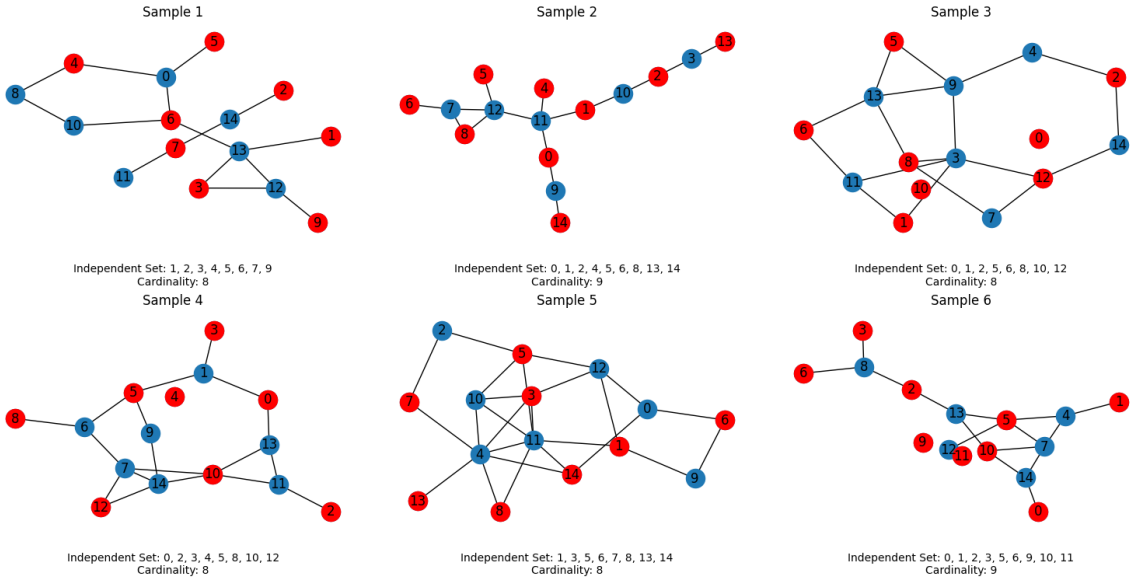


Figure 5: Test cases with vertex size = 15 and edge probability = 0.15

Batch 1	
1.1	(0, 1),(0, 2),(1, 3)
1.2	(0, 1),(0, 3),(1, 2),(1, 3),(2, 4)
1.3	(0, 2),(1, 4),(2, 3),(3, 4)
1.4	(0, 1),(0, 2),(0, 3),(0, 4),(1, 2),(1, 3),(2, 3),(3, 4)
1.5	(0, 1),(0, 3),(1, 2),(1, 4),(2, 3)
1.6	(0, 3),(0, 4),(1, 2),(1, 3),(2, 4)
Batch 2	
2.1	(0, 2),(0, 8),(0, 9),(1, 3),(1, 4),(1, 6),(1, 9),(2, 4),(3, 7),(4, 7),(4, 9),(5, 7),(5, 8),(5, 9),(6, 7),(6, 8),(7, 8),(8, 9)
2.2	(0, 1),(0, 4),(0, 9),(1, 3),(1, 5),(2, 4),(2, 9),(3, 5),(4, 5),(4, 6),(4, 7),(4, 8),(5, 6),(5, 8),(8, 9)
2.3	(0, 5),(0, 7),(0, 8),(1, 9),(2, 4),(2, 6),(2, 8),(3, 5),(3, 6),(4, 5),(4, 6),(4, 9),(5, 6),(5, 9),(6, 8),(7, 8),(7, 9)
2.4	(0, 3),(0, 4),(0, 6),(0, 7),(1, 2),(1, 4),(1, 6),(1, 8),(2, 3),(2, 5),(2, 6),(2, 7),(3, 4),(4, 5),(4, 9),(5, 9)
2.5	(0, 1),(0, 2),(0, 4),(0, 5),(0, 7),(1, 5),(2, 3),(2, 4),(2, 5),(2, 7),(2, 9),(3, 7),(4, 6),(4, 8),(5, 9)
2.6	(0, 2),(1, 3),(1, 5),(1, 7),(1, 8),(1, 9),(2, 6),(2, 7),(3, 4),(3, 6),(4, 5),(4, 8),(4, 9),(5, 6),(5, 8),(6, 9)
Batch 3	
3.1	(0, 4),(0, 5),(0, 6),(1, 13),(2, 14),(3, 12),(3, 13),(4, 8),(6, 10),(6, 13),(7, 11),(7, 14),(8, 10),(9, 12),(12, 13)
3.2	(0, 9),(0, 11),(1, 10),(1, 11),(2, 3),(2, 10),(3, 13),(4, 11),(5, 12),(6, 7),(7, 8),(7, 12),(8, 12),(9, 14),(11, 12)
3.3	(1, 3),(1, 11),(2, 4),(2, 14),(3, 8),(3, 9),(3, 11),(3, 12),(4, 9),(5, 9),(5, 13),(6, 11),(6, 13),(7, 8),(7, 12), (8, 13),(9, 13),(12, 14)
3.4	(0, 1),(0, 13),(1, 3),(1, 5),(2, 11),(5, 6),(5, 9),(6, 7),(6, 8),(7, 10),(7, 12),(7, 14), (9, 14),(10, 11),(10, 13),(10, 14),(11, 13),(12, 14)
3.5	(0, 6),(0, 12),(0, 14),(1, 9),(1, 11),(1, 12),(2, 5),(2, 7),(3, 4),(3, 10),(3, 11),(3, 12),(4, 7),(4, 8), (4, 10),(4, 11),(4, 13),(4, 14),(5, 10),(5, 11),(5, 12),(6, 9),(8, 11),(10, 11),(11, 14)
3.6	(0, 14),(1, 4),(2, 8),(2, 13),(3, 8),(4, 5),(4, 7),(5, 7),(5, 12),(5, 13),(6, 8),(7, 10),(7, 14),(10, 13),(10, 14)

Table 1: Edges for batches 1, 2, and 3

b. Heuristic Algorithm

```python
def calculate_k_parameter(v, V0, graph):
    count = 0
    for neighbor in graph.neighbors(v):
        if neighbor in V0:
            count += 1
    return count

def calculate_m_parameter(v, graph):
    count = 0
    for neighbor1 in graph.neighbors(v):
        for neighbor2 in graph.neighbors(v):
            if neighbor1 != neighbor2 and not graph.has_edge(neighbor1, neighbor2):
                count += 1
    return count


def heuristic_max_independent_set(graph):
    V0 = set(graph.nodes())      # Set of all vertices in the graph
    S = set()                    # Independent set
    Est = 0                      # Estimate for deviation from the exact solution
    maxM = 0                     # Maximum m value encountred

    while V0:   # Iterate while there are vertices remaining in V0
        k = {}  # Dictionary to store the values of parameter k for vertices
        m = {}  # Dictionary to store the values of parameter m for vertices

        for v in V0:
            k[v] = calculate_k_parameter(v, V0, graph)  # k[v] is the number of neighbors of v
    ↪ that are still in V0
            m[v] = calculate_m_parameter(v, graph)      # m[v] is the number of edges that need
    ↪ to be added to v's neighborhood to make it a complete induced subgraph, it is missing edges
    ↪  amoung all possible edges with neighborhood.

        v0 = min(V0, key=lambda v: (m[v], -k[v]))   # MinMax(m,k) Among vertices with the minimum
    ↪  m the vertex with the maximum k is selected
        S.add(v0)                                   # Add v0 to the independent set S
        Est += m[v0]                                # Update the estimation by adding m[v0]
        if(m[v0] > maxM):
            maxM = m[v0]

        V0.remove(v0)                   # Remove v0 from V0
        V0 -= set(graph.neighbors(v0))  # Remove the neighborhood of v0 from V0

    return S, Est, maxM

```

Listing 3: Heuristic Algorithm

We have done 18 tests in total in 3 batches. Batches had edge probabilities of 0.45, 0.30 and 0.15 and vertex amounts of 5, 10 and 15. Below you can find our test samples of the aforementioned function with random sample generation code. The results are visualised using mathplotlib and raw outputs are also given under each visualization. Moreover edges of each graph are listed after the figures. Out of 18 test 16 had the correct cardinality for max set and 2 sets had cardinality lower than 1 compared to real set.

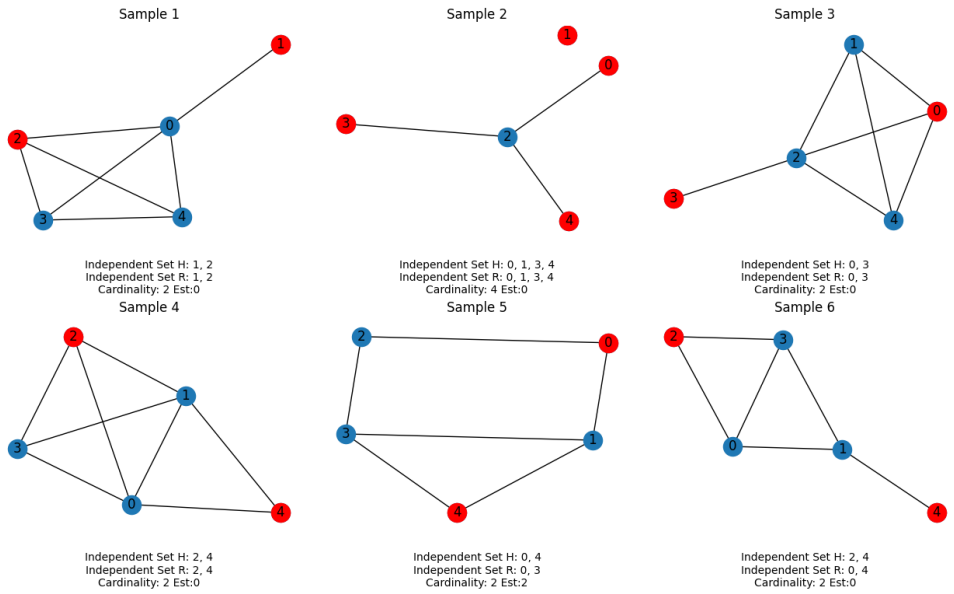


Figure 6: Test cases with vertex size = 5 and edge probability = 0.45

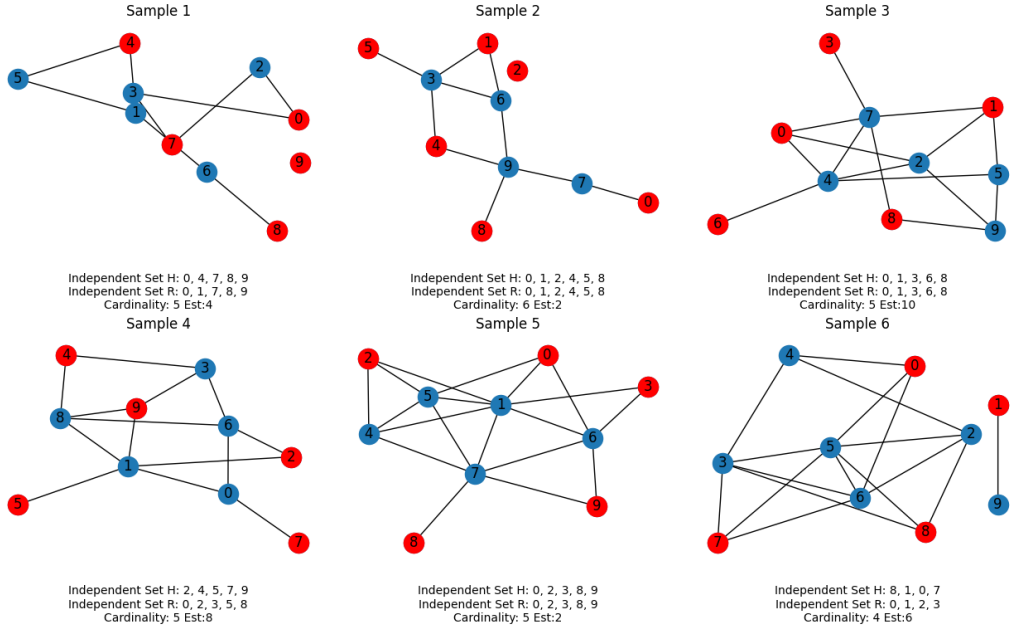


Figure 7: Test cases with vertex size = 10 and edge probability = 0.30

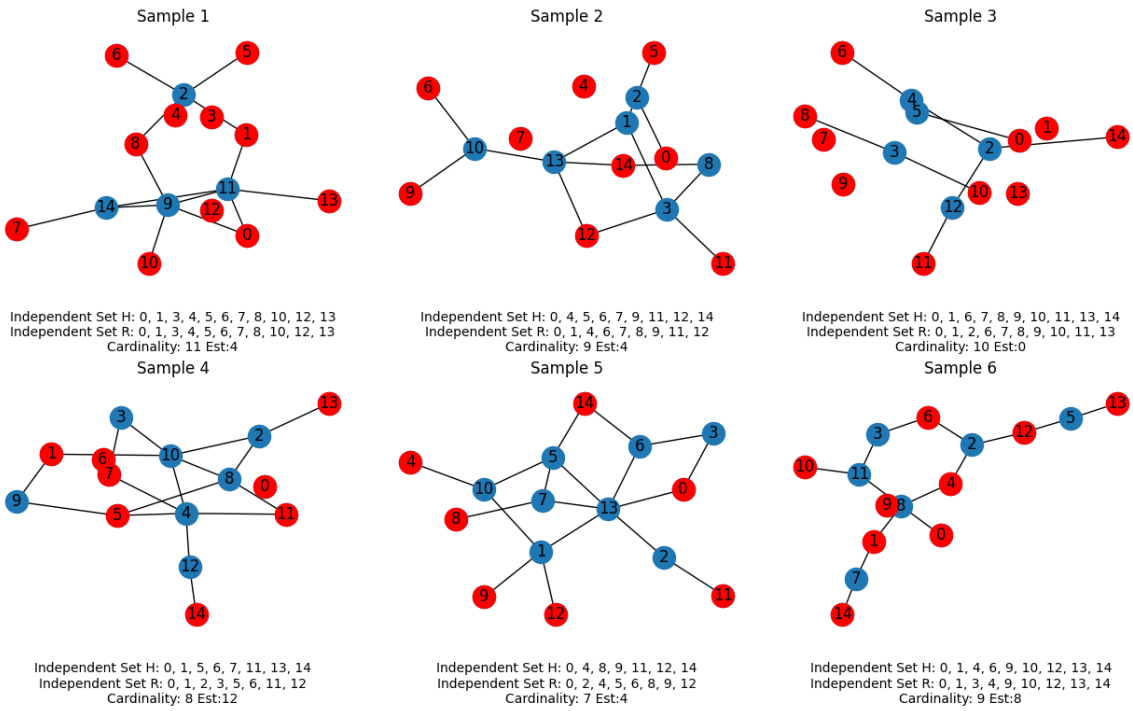


Figure 8: Test cases with vertex size = 15 and edge probability = 0.15

Batch 1	
1.1	(0, 1),(0, 2),(0, 3),(0, 4),(2, 3),(2, 4),(3, 4))
1.2	(0, 2),(2, 3),(2, 4)
1.3	(0, 1),(0, 2),(0, 4),(1, 2),(1, 4),(2, 3),(2, 4)
1.4	(0, 1),(0, 2),(0, 3),(0, 4),(1, 2),(1, 3),(1, 4),(2, 3)
1.5	(0, 1),(0, 2),(1, 3),(1, 4),(2, 3),(3, 4)
1.6	(0, 1),(0, 2),(0, 3),(1, 3),(1, 4),(2, 3)
Batch 2	
2.1	(0, 2),(0, 3),(1, 4),(1, 5),(1, 6),(2, 7),(3, 7),(4, 5),(6, 8)
2.2	(0, 7),(1, 3),(1, 6),(3, 4),(3, 5),(3, 6),(4, 9),(6, 9),(7, 9),(8, 9)
2.3	(0, 2),(0, 4),(0, 7),(1, 2),(1, 5),(1, 7),(2, 4),(2, 9),(3, 7),(4, 5),(4, 6),(4, 7),(5, 9),(7, 8),(8, 9)
2.4	(0, 1),(0, 6),(0, 7),(1, 2),(1, 5),(1, 8),(1, 9),(2, 6),(3, 4),(3, 6),(3, 9),(4, 8),(6, 8),(8, 9)
2.5	(0, 1),(0, 5),(0, 6),(1, 2),(1, 3),(1, 4),(1, 5),(1, 6),(1, 7),(2, 4),(2, 5),(3, 6),(4, 5),(4, 7),(5, 7),(6, 7),(6, 9), (7, 8),(7, 9)
2.6	(0, 4),(0, 5),(0, 6),(1, 9),(2, 4),(2, 5),(2, 6),(2, 8),(3, 4),(3, 5),(3, 6),(3, 7),(3, 8),(5, 6),(5, 7),(5, 8),(6, 7)
Batch 3	
3.1	(0, 9),(0, 11),(1, 2),(1, 11),(2, 5),(2, 6),(2, 8),(7, 14),(8, 9),(9, 10),(9, 11),(9, 14),(11, 13),(11, 14)
3.2	(0, 2),(1, 3),(1, 5),(1, 13),(3, 8),(3, 11),(3, 12),(6, 10),(8, 14),(9, 10),(10, 13),(12, 13),(13, 14)
3.3	(0, 5),(2, 4),(2, 12),(2, 14),(3, 8),(3, 10),(4, 6),(11, 12)
3.4	(1, 9),(1, 10),(2, 8),(2, 10),(2, 13),(3, 7),(3, 10),(4, 5),(4, 7),(4, 10),(4, 11),(4, 12),(5, 8),(5, 9),(8, 10), (8, 11),(12, 14)
3.5	(0, 3),(0, 13),(1, 9),(1, 10),(1, 12),(1, 13),(2, 11),(2, 13),(3, 6),(4, 10),(5, 7),(5, 10),(5, 13),(5, 14),(6, 13), (6, 14),(7, 8),(7, 13)
3.6	(0, 8),(1, 7),(1, 8),(2, 4),(2, 6),(2, 12),(3, 6),(3, 11),(4, 8),(5, 12),(5, 13),(7, 14),(8, 11),(10, 11)

Table 2: Edges for batches 1, 2, and 3

6 Experimental Analysis of The Performance (Performance Testing)

In performance testing we will first look at our solution's performance in practice. We will be conducting tests different vertex sized and we will calculate confidence interval for running time for the mean of tests of a given vertex size. First we will be calculating sample mean (m) and standard deviation (sd) from predetermined N measurements. After we will be calculating estimated error S_m as follows:

$$\text{Estimated Standard Error} S_m = \frac{\text{Sample Standard Deviation (sd)}}{\sqrt{N}} \tag{5}$$

Than a confidence interval for mean of time of all possible runs (M) of a given input size can be calculated with following formula:

$$M = [m - t \times S_m, m + t \times S_m] \quad \text{with probability CL\%}$$

For our test we will pick **CL = 95** and **N = 30** while having a constant edge density with edge density parameter set to p=0.4. Test results will tell us that mean of all possible graph with given edge density and vertices number is within the given confidence interval with probability of 95
The test results are given in the below table.

Table 3: Running Time Statistics

Vertex Number	Mean Running Time (s)	Std. Dev.	Std Error	CI Lower Bound	CI Upper Bound
50	0.014	0.001	0.000	0.014	0.015
100	0.127	0.010	0.002	0.123	0.130
150	0.427	0.020	0.004	0.419	0.434
200	0.972	0.048	0.009	0.954	0.990
250	1.902	0.079	0.014	1.872	1.931
300	3.388	0.222	0.041	3.305	3.471
350	4.990	0.238	0.043	4.901	5.078
400	7.615	0.381	0.070	7.473	7.757
450	10.437	0.565	0.103	10.226	10.648
500	12.864	0.349	0.064	12.733	12.994

Using the above results we plotted log of mean values and test values. We used a log plot since our running time is polynomial with degree higher than one. Than we have fitted a line this log plot. Below plot shows the fitted line. The fitted line had a **slope = 2.84** This result tells us our actual running time function polynomial has complexity degree close to 3 with respect to vertex size. This observation follows our complexity analysis as we argued our graph runs in $O(n^3)$ time.

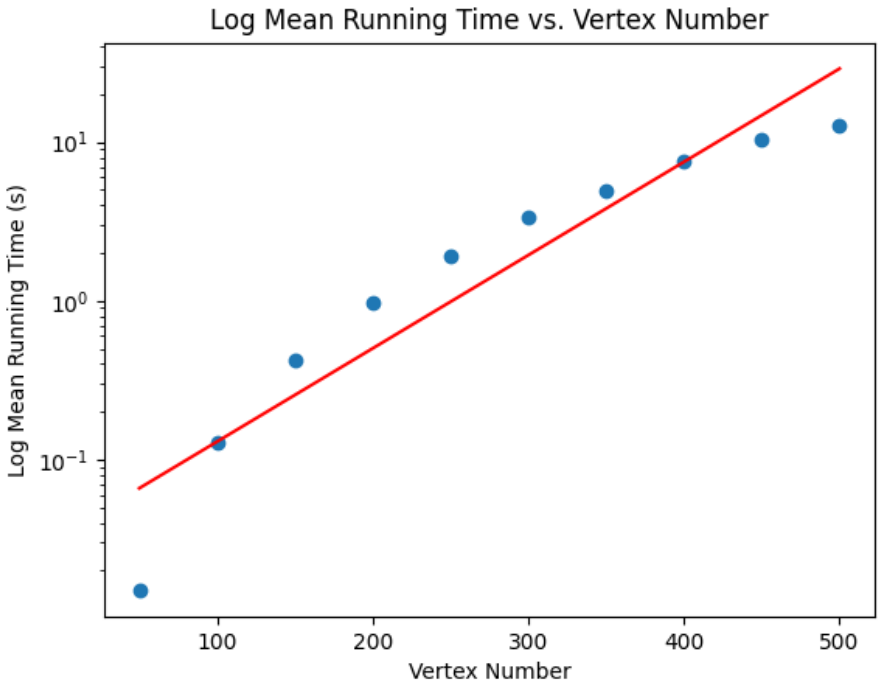


Figure 9: Mean Running time vs Vertex Number

Next we have done solution quality tests. To experimentally analyze the ratio bound from the previous chapter we will use the following ratio definition with help of Brute Force algorithm.

$$Ratio = \frac{Cardinality of Brute Force Algorithm}{Cardinality of Approximation Algorithm} \tag{6}$$

Ratio of the Approximation algorithm will be calculated as following way with 50 for the below vertex number. The bound for given ratio $\rho(v) = maxM/C$ is also displayed in table. Notice that max m is the maximum m value we have encountered in algorithm. We have also asserted the ratio $\leq \rho(v)$. Out of 500 tests conducted this assertion always held correct.

Table 4: Test Results

Vertex Number	Ratio	Max Cardinality Difference	Ratio Bound
5	1.06	1	1.266666667
10	1.078333333	1	2.638666667
15	1.082666667	2	4.414
20	1.164	2	7.490571429
25	1.152	2	11.60133333
30	1.139333333	2	15.06809524
35	1.179333333	2	20.19690476
40	1.183047619	3	25.4197619
45	1.22352381	3	30.12777778
50	1.235333333	3	37.18730159

Notice that quality of the algorithm solution decreases as vertex size increase. However this decrease is always below the ratio bound we have provided. Based test results we can conform that quality of our algorithm is bounded by ratio bound. Also below you can find graph of the average ratio bounds and ratios plotted. This graph also shows the rate of increase for our ratio is lower than ratio bound function.

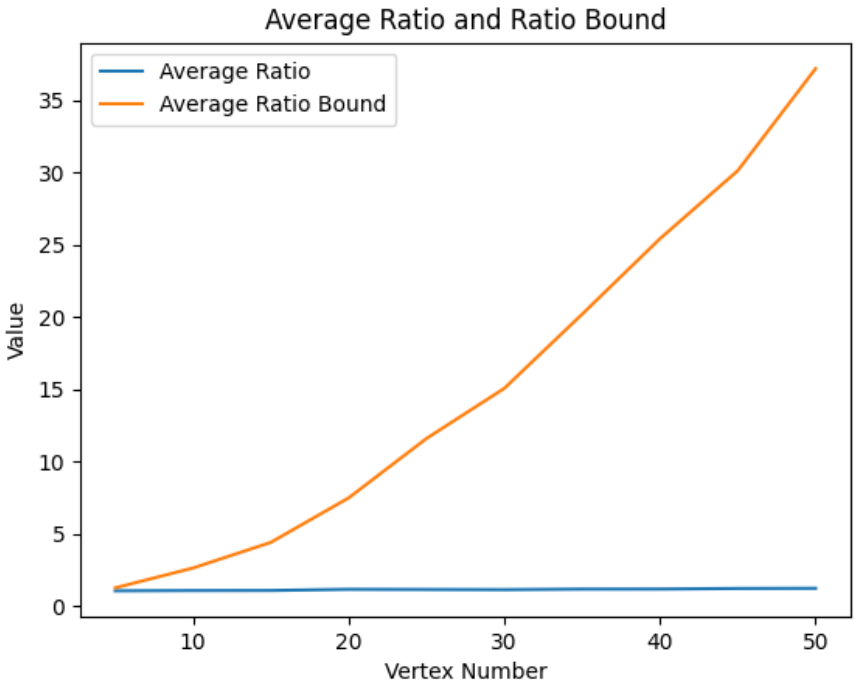


Figure 10: Ratio Bound vs Ratio comparison

7 Experimental Analysis of the Correctness (Functional Testing)

We have done black box testing with equivalence classes to prove correctness of the algorithm. We have identified four valid input graph equivalence classes which are:

1-Empty graphs
2-Sparse graphs
3-Dense graphs
4-Complete graphs

Along with testing items from each equivalence class we will test with inputs around boundary cases.

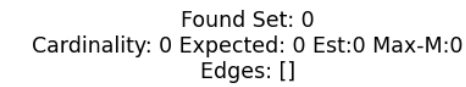


Figure 11: Test 1 Empty graph

Figure 12: Test 2 One vertex graph

Figure 13: Test 3 Two vertex sparse graph

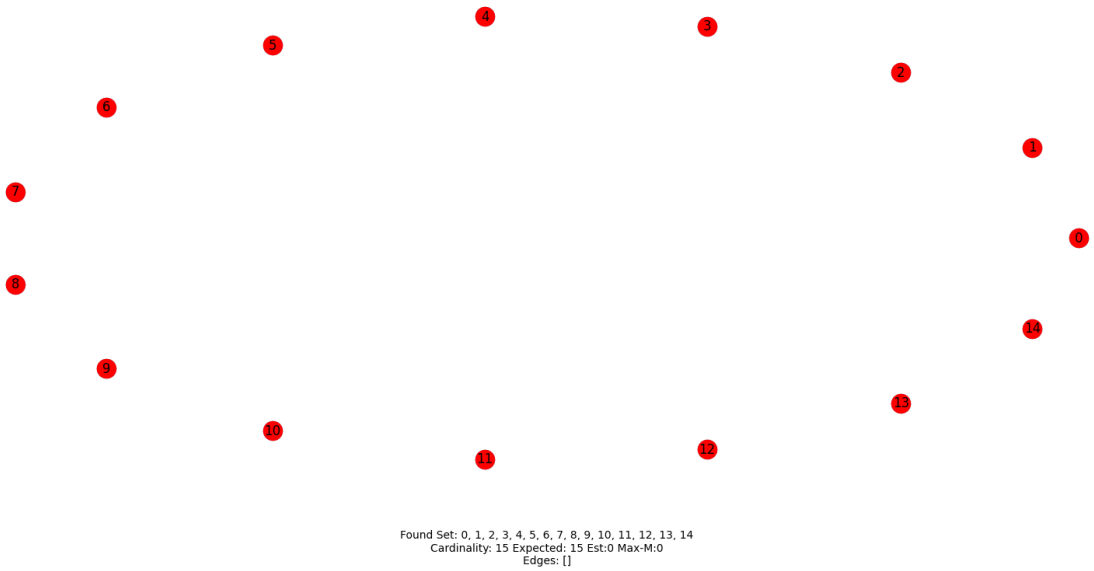


Figure 14: Test 4 No edge graph

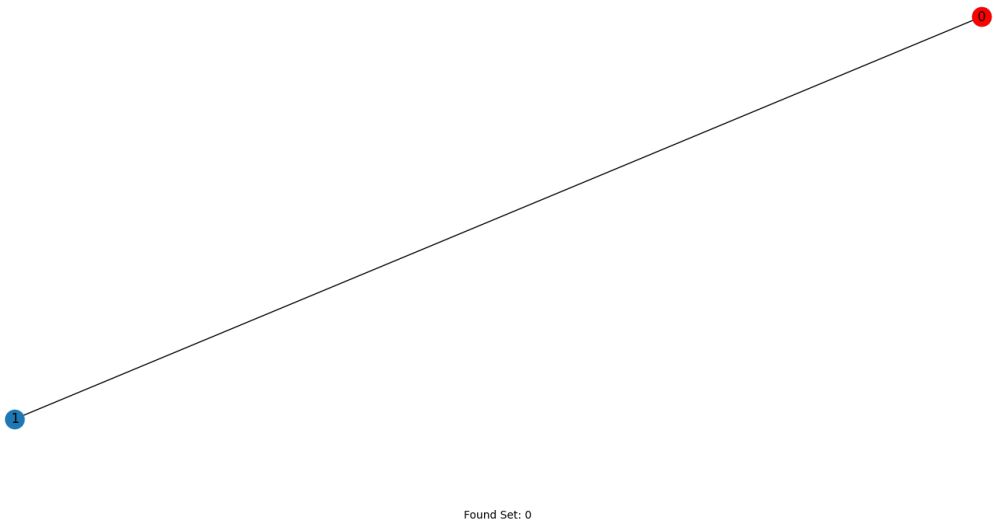


Figure 15: Test 5 Two vertex dense graph

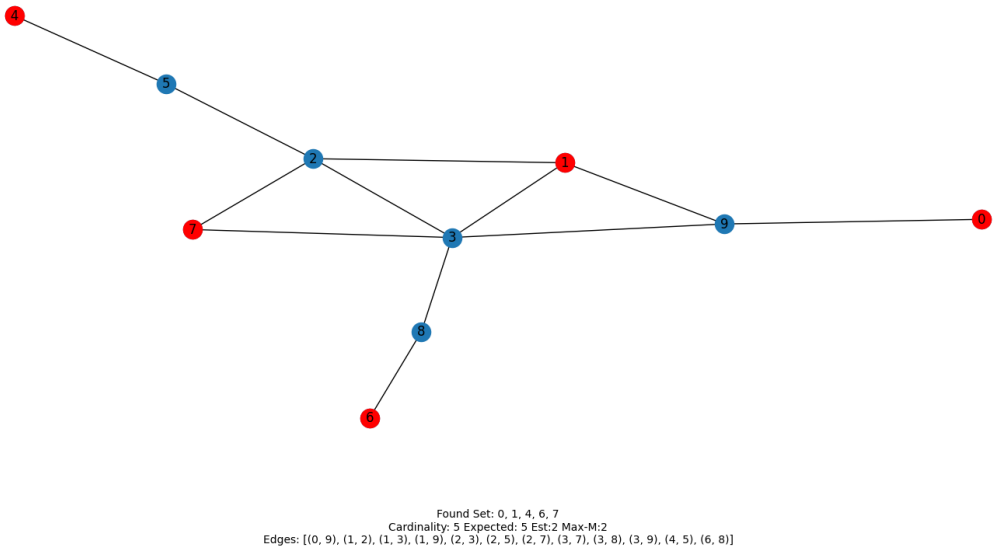
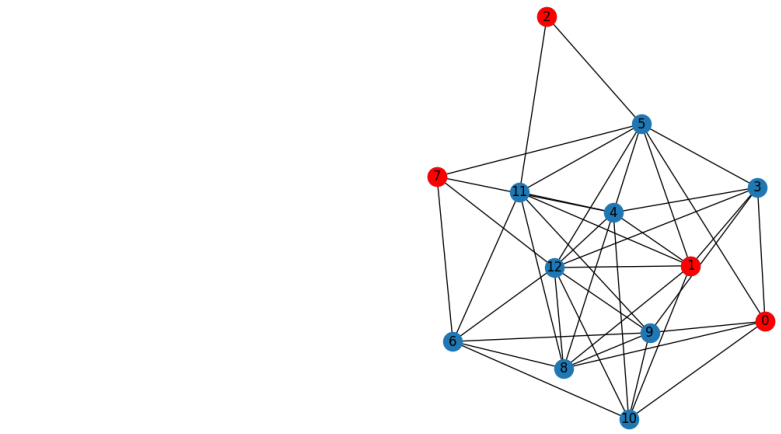
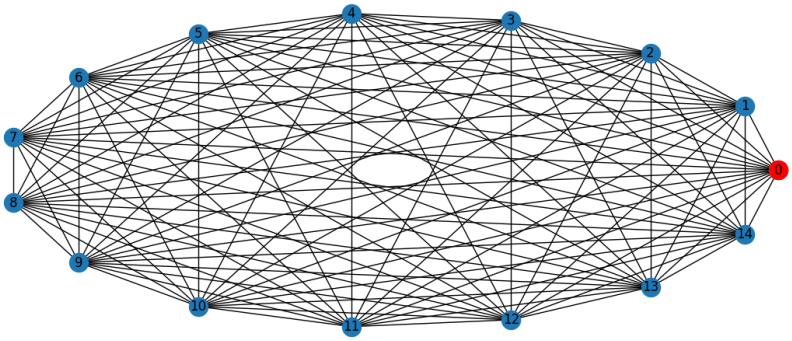


Figure 16: Test 6 Sparse graph



Found Set: 0, 1, 2, 7
Cardinality: 4 Expected: 5 Est:32 Max-M:16
1, 3), (1, 4), (1, 5), (1, 8), (1, 10), (1, 11), (1, 12), (2, 5), (2, 11), (3, 4), (3, 5), (3, 9), (3, 12), (4, 5), (4, 7), (4, 8), (4, 10), (4, 11), (4, 12), (5, 7), (5, 11), (5, 12), (6, 7), (6, 8), (6, 9), (6, 10), (6, 11), (6, 12), (7, 12), (8, 9), (8, 1

Figure 17: Test 7 Dense graph - Error lower than ratio bound = 5



Found Set: 0
Cardinality: 1 Expected: 1 Est:0 Max-M:0

Figure 18: Test 8 Complete graph

8 Discussion

We have analyzed the maximal independent set problem which is a NP-Hard problem. Also we analyzed and tested a brute force algorithm and a heuristic algorithm to solve this problem. Our experiments have shown that brute force algorithm performs very slowly for large input sizes. Even going above 25 vertex in some test caused algorithm to solve down significantly. This showed us brute force algorithm is not very reliable in practice. We have also look at a heuristic algorithm that solves the problem by staying under a certain ratio bound. Our tests shown that ratio bound we proposed for our algorithm grows larger than actual ratio. Therefore we believe a better ratio bound for the algorithm is possible. Another comment is about the slope of the fitted line. Slope of the fitted suggests that algorithm a slightly lower complexity than the initially proposed complexity of $O(V^3)$. However results did not changed significantly. Moreover we have calculated a %95. We believe %95 is a good industry standard and reasonable enough bound for our results. And finally the black-box tests we have done show algorithm correctness using equivalence classes. Algorithm was tested for only a limited number of members from each equivalence class. However this values include generic members as well as boundary values. We expect all the other member in the class to function the same way. However it would not be possible to do a a test for all members of class.

References

[1] E.Gardiner, P.Willett, and P.Artymiuk. Graph-theoretic techniques for macromolecular docking. J. Chem. Inf. Comput., 40:273–279, 2000.

[2] Richard M. Karp, Reducibility among Combinatorial Problems, In Proceedings of a symposium on the Complexity of Computer Computations, 1972, pp.85–103.

[3] S. Har-Peled, NP-Completeness II (CS 573 Lecture 2), University of Illinois at Urbana-Champaign, 2012.

[4] GeeksforGeeks, "Maximal Independent Set in an Undirected Graph," [Online]. Available: https://www.geeksforgeeks.org/maximal-independent-set-in-an-undirected-graph/ (accessed Apr 25, 2023).

[5] "Independence Number of a Graph," [Online]. Available: https://symbio6.nl/en/blog/analysis/independence-number-graph (accessed Apr 25, 2023).

[6] Gainanov, D. N., Mladenovic, N., Rasskazova, V., Urosevic, D. (2018, January). Heuristic algorithm for finding the maximum independent set with absolute estimate of the accuracy. In Proc. CEUR Workshop (pp. 141-149).