

CS301 Project 1 Progress Report

Group 19

Group Members

Emre Koçer 26709

Ege Erdoğan 25331

Hüseyin Kaya 26460

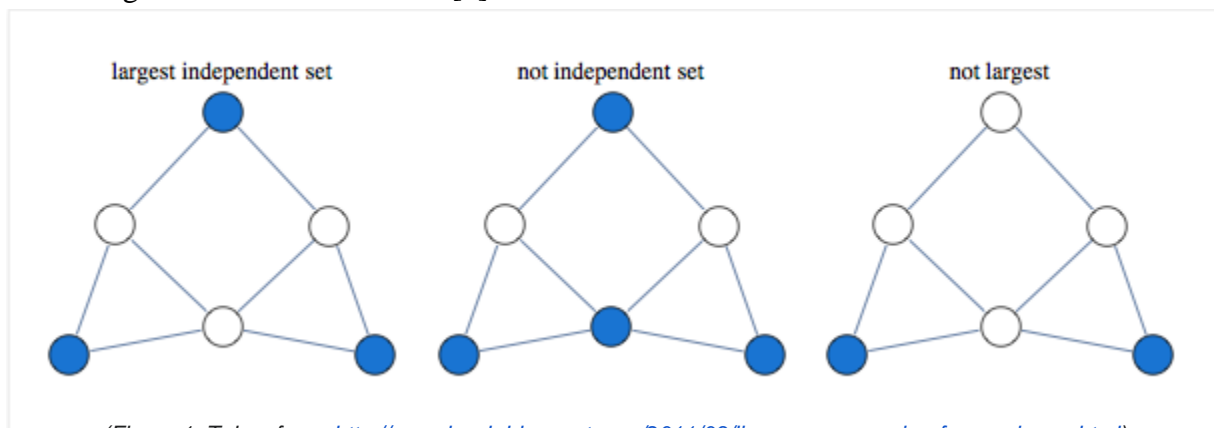
Jankat Yaşar 26635

1. Problem Description

Intuitively, the independent set corresponds to where there are vertices which are not neighbors of each other in the graph. In addition to this, the Maximum Independent set is the largest possible independent set for the given graph.

Formally, there is a graph called $G(V,E)$ where V represents vertices and E represents Edges (the connection between vertices). The maximal independent set is the largest cardinality subset W of V such that no pair of nodes in W is joined by an edge in E .

Maximal independent set can be used for complex scheduling problems such as scheduling Earth Observing Satellite Constellations.[1]



(Figure 1. Taken from: <http://yaroslavvb.blogspot.com/2011/03/linear-programming-for-maximum.html>)

This problem that is being considered has a hardness that is NP-Hard. NP-hard can be defined as following: Assume that there is problem K, it is NP-hard if there is an NP-Complete problem L such that L can be reduced into K in polynomial time therefore goal is to show that maximal independent set problem can be reduced to another problem in polynomial time. To be more precise maximal independent set problem is trivially solved in $2^n n^{O(1)}$ time will be shown that the problem can be transformed into some problem that has less complexity than this problem in polynomial time (the reduction process should take polynomial time).

Step 1 Show that problem is np-complete:

Assume that there is an undirected simple graph denoted as $G=(V,E)$, let a subset of $I \subseteq V$ in G such that now two vertices are connected together. Now we must check that there isn't any edge in G that connects at least another vertex in I then we reject it otherwise accept meaning that algorithm runs in polynomial time, so this problem is NP.[2]

Step 2 Show that it is NP-Hard:

It is sufficient to show that the np-complete problem which is the boolean satisfiability problem which asks for a given problem in boolean algebra what is the fastest algorithm that is denoted as 3SAT can be reducible to independent set problem in polynomial time. Let F be the boolean function in 3-CNF form, This function f with an input of the boolean formula F shall output a graph, G, and an integer k, That is, we are looking for a $f(F) = (G, k)$ with such that F is satisfiable if and only if G has an independent size of k.[2]

Correctness: We'll show that F is satisfiable if and only if G has an independent set of size k.

(\Rightarrow): If F is satisfiable, then each of the k clauses of F must have at least one true literal. Select such a literal from each clause. Let V_0 denote the corresponding vertices from each of the clause clusters (one from each cluster). We claim that V_0 is an independent set of size k . Since there are k clauses, clearly $|V_0| = k$. We only take one vertex from each clause cluster, and we cannot take two conflicting literals to be in V_0 . For each edge of G , both of its endpoints cannot be in V_0 . Therefore V_0 is an independent set of size k . [3]

(\Leftarrow): Suppose that G has an independent set V_0 of size k . We cannot select two vertices from a clause cluster, and since there are k clusters, V_0 has exactly one vertex from each clause cluster. Note that if a vertex labeled x is in V_0 then the adjacent vertex x cannot also be in V_0 . Therefore, there exists an assignment in which every literal corresponding to a vertex appearing in V_0 is set to 1. Such an assignment satisfies one literal in each clause, and therefore the entire formula is satisfied.[3]

2. Algorithm Description

A.

The algorithm is exponential time brute force where each vertex is visited with two different possibilities, vertex can be included in the maximum independent set or not. Each time, the algorithm picks a vertex from the graph and excludes this vertex from the graph. Then, the algorithm recursively checks the remaining graph to find the maximum independent set. Algorithm also includes this selected vertex in the maximum independent set and removes its neighbors from the graph. Then, it tries to maximum independent set with the remaining graph.

Since all vertices are checked and the algorithm runs recursively. The time complexity is exponential with 2^n which results with an inefficient algorithm in terms of run time.

Pseudocode:

Step 1: Select a vertex from our graph.

Step 2: Exclude this selected vertex from the maximum independent set by deleting the vertex from the graph and recursively call the function. Obtain set 1.

Step 3: Include this selected vertex in the maximum independent set by deleting the vertex and its neighbors from the graph and recursively call the function. At the end, add the selected graph to the result. Obtain set 2.

Step 4: Check which set is larger set 1 or set 2.

Step 5: Return the larger set which is the maximum independent set.

Implementation [5]:

```
def find_max_set(G):
    vertices = list(G.keys())
    if len(vertices) == 0:
        return [], 0
    elif len(vertices) == 1:
        return [vertices[0]], 1
    else:
        temp = vertices[0]
        G2 = dict(G)
        del G2[temp]
        set1 = find_max_set(G2)[0]
        for vertex in G[temp]:
            if vertex in G2:
                del G2[vertex]
        set2 = [temp] + find_max_set(G2)[0]
        if len(set2) > len(set1):
            return set2, len(set2)
        else:
            return set1, len(set1)
```

→ **$O(2^V)$**

B.

The algorithm is a greedy algorithm where we need to find the vertex which has the smallest number of edges connected to it in polynomial time. Every vertex has three characteristics which are "Node id", "Connected vertices ([] as default)" and "Independency (0 as default)". For every iteration, we mark the node we are working as "Independent (1)" if the "Independency" of a vertex is not set. Then, we mark as "Not Independent (-1)" for every neighbor of that vertex and call a recursive algorithm for the neighbors of "Not Independent" marked vertices by beginning with the vertex which has the smallest number of neighbors.

Pseudocode:

Step 1: Sorting the neighbours of vertices with respect to their number of neighbours.

Step 2: Creating an empty array called max_independent

Step 3: Choose the vertex with the minimum number of neighbours and call it "min_id"

Step 4: If any of the neighbours of "min_id" did not get marked as independent, the chosen vertex is marked as independent, else marked as not independent.

Step 5: If the chosen vertex was marked as "Independent" its neighbours get marked as "Not Independent".

Step 6: Chosen vertex's neighbour's neighbours are evaluated and the one with the minimum number of neighbours gets chosen.


Step 7: The new chosen vertex gets recursively called in the function.

Step 8: Find the Independent nodes in the graph.

Implementation:

Find_min_element Algorithm:


```
1 def find_min_element(graph): # Finding the point which has smallest number of connected neighbours to start the program - O(N)
2   min_next_number = len(graph[0].next_vertices)
3   min_id = graph[0].id
4   for i in range(1, len(graph)):
5     if len(graph[i].next_vertices) < min_next_number and len(graph[i].next_vertices) > 0:
6       min_next_number = len(graph[i].next_vertices)
7       min_id = graph[i].id
8   return min_id
```



$O(V)$

Next_vertex_sort Algorithm:


```
1 def next_vertex_sort(graph): # For every element in graph, sort the nighbours of the element - O(N*N*N) = O(N^3)
2   for id in range(len(graph)): # O(N)
3     for vertex_id_prev in range(len(graph[id].next_vertices)): #O(N)
4       vertex_id = graph[id].next_vertices[vertex_id_prev]
5       min_elm = len(graph[vertex_id].next_vertices)
6       min_elm_id = vertex_id_prev
7       for vertex_id_check_prev in range(vertex_id_prev, len(graph[id].next_vertices)): #O(N)
8         vertex_id_check = graph[id].next_vertices[vertex_id_check_prev]
9         if min_elm > len(graph[vertex_id_check].next_vertices):
10           min_elm = len(graph[vertex_id_check].next_vertices)
11           min_elm_id = vertex_id_check_prev
12         #swap min with selected element
13         tmp_id = graph[id].next_vertices[vertex_id_prev]
14         graph[id].next_vertices[vertex_id_prev] = graph[id].next_vertices[min_elm_id]
15         graph[id].next_vertices[min_elm_id] = tmp_id
16   return graph
```




$O(V^3)$

Find_max_independent and max_cardinality Algorithms:

```
1 def find_max_independent(graph, id, count, counter):
2   check = True
3   #Check for whether the point has independent nighbour or not
4   for ix in range(len(graph[id].next_vertices)): # O(N) for checking every edges of a vertice
5     if graph[graph[id].next_vertices[ix]].independent == 1:
6       count = count + 1
7       check = False
8       break;
9   if check:
10    graph[id].independent = 1
11  else:
12    graph[id].independent = -1
13  #If point has marked as 1: mark it's neighbours as -1, dont mark anything otherwise
14  for i in range(len(graph[id].next_vertices)): # O(N) for iterating thorough every edges of a vertice
15    count = count + 1
16    if graph[graph[id].next_vertices[i]].independent == 0 and graph[id].independent == 1:
17      graph[graph[id].next_vertices[i]].independent = -1
18    for idx in range(len(graph[graph[id].next_vertices[i]].next_vertices)): # O(N) for iterating through every edges of the next independent points
19      #Call recursive Function for each nighbour of neighbour of the point
20      if graph[graph[graph[id].next_vertices[i]].next_vertices[idx]].independent == 0:
21        graph = find_max_independent(graph, graph[graph[graph[id].next_vertices[i]].next_vertices[idx]].id, count, counter) # O(N^2) recursion
22      else:
23        continue
24  return graph
25
26 def max_cardinality(graph):
27   max_cardinality = []
28   for elm in graph:
29     if elm.independent == 1: # If the id has marked "independent", add to max_cardinality
30       max_cardinality.append(elm)
31   return max_cardinality
```



$O(V^2 + V)$



$O(V)$

Ratio Bound:

Our algorithm can be seen as an example of maximization problem where the size of independent set is trying to be maximized with the below ratio bound formula:

$$C^*/C \leq \rho(n)$$

Where C^* corresponds to optimal solution and C corresponds to our algorithm's solution, and $\rho(n)$ corresponds to ratio bound with input size n .

Let's prove the ratio bound for our approximation algorithm with the help of Gainanov[4]:

Important Proposition: There exists a maximal independent set $S \subseteq V$ such that a vertex v

$$|S| \geq \max S(G) - m \quad (1)$$

Proof of the proposition: Suppose there are 'm' number of missed edges for vertex v denotes as $\{e_1, \dots, e_m\}$. Add these missing edges to the graph G which denoted as G' and it is obtained k -vertex which means G has more than k vertices and if we remove a smaller number of k vertices it still stays connected. From Gainanov's first proposition, we know that there exists $S' \in S_{\max}(G')$ such that $v \in S'$. Then, the Corollary 2 of the Gainanov can be used to obtain $|S| = \max S(G') \geq \max S(G) - m$. At the end, if the second proposition of Gainanov's applied to this formula following result can be found to prove (1):

$$|S| \geq |S'| \geq \max S(G) - m$$

Now we can use the proposition (1), to **prove the ratio bound:**

$$\max S(G) - |S| \leq m$$

We know that $\max S(G)$ corresponds to C^* , and $|S|$ corresponds to C . Then we have,

$$C^* - C \leq m$$

Divide each term with C (if the graph is not empty to eliminate 0 division):

$$\frac{C^*}{C} - 1 \leq \frac{m}{C}$$

Then, define ratio bound $\rho(n)$ as $\frac{m}{C} + 1$ to obtain previously given ratio bound formula.

3. Algorithm Analysis

A.

The algorithm checks both possibilities for each vertex in the graph. There are only two possibilities: A vertex will be in the maximum independent set or not. Since both possibilities are checked recursively for each vertex, there are no other additional options to check. In short, the algorithm exhaustively finds the maximum independent set by considering all cases. Therefore, the algorithm is correct.

The running time of the algorithm can be seen as below recurrence:

$$T(n) = 2T(n-1) + O(n) = O(2^n)$$

The space complexity is $2O(n)$ where two graphs are used by the algorithm which is simply equal to $O(n)$.

B.

‘Find_min_element’ runs in $O(V)$ as we are visiting each vertex in the graph.

‘Find_max_independent’ function runs in $O(V^2 + V)$ time recursively. However, the

‘next_vertice_sort’ function runs in $O(V^3)$ time, so it dominates for a big number of inputs. At the last step, we find the independent nodes by ‘Max_Cardinality’ and it operates in $O(V)$ time.

Therefore, we can say that our time complexity is $O(V^3)$ as it is dominating one.

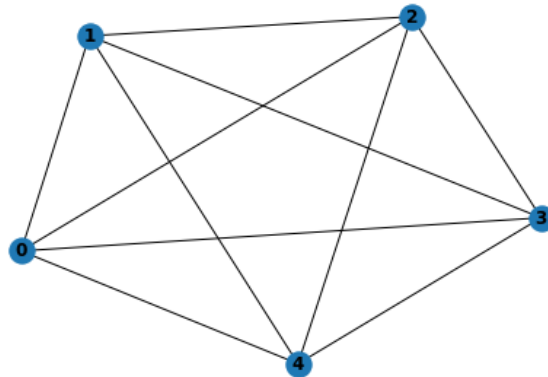
Space Complexity is $O(V^2)$ as we have only Vertices and Edges in our memory.

Correctness of our approximation algorithm:

The proposed algorithm always finds **an independent set**. However, this resulted independent set may not be the maximum independent set, i.e., optimal solution as the algorithm is **heuristic**.

There are two possible scenarios for each vertex in the graph:

1) A vertex may be in the maximum independent set:



Suppose selected vertex is the vertex 0 and its independency will be equal to 1, i.e, it is chosen as independent vertex and in the maximum independent set. Since the algorithm assigning neighbors of this selected vertex independency as -1, i.e., neighbors are chosen as dependent vertices, and they are not in the maximum independent set. Therefore, there is no possibility to have two connected vertices in the resulted set of the algorithm.

2) A vertex may not be in the maximum independent set:

Suppose selected vertex is the vertex 0 and its independency is equal to -1 because of one of its neighbors, i.e, it is chosen as dependent vertex, and it is not in the maximum independent set.

Then, it is obvious that one of its neighbors is in the maximum independent set as the vertex 0 has independency equal to -1. Therefore, there is no possibility to have two connected vertices in the resulted set of the algorithm.

In result, there is **no possible collision** in the result of the proposed approximation algorithm as it does not care about nodes which have independency equal to -1.

4. Sample Generation / An Initial Testing of Implementation of the Algorithm

Samples are generated by utilizing the networkx library of Python. Graphs are checked whether they are connected or not before testing as the maximum independent set is considered while having a connected graph.

Implementation:

```
import networkx as nx
from networkx.generators.random_graphs import erdos_renyi_graph
import matplotlib.pyplot as plt
from networkx.algorithms.approximation import clique_removal

[ ] def check_connected(G):
    if nx.is_connected(G):
        return False
    else:
        return True

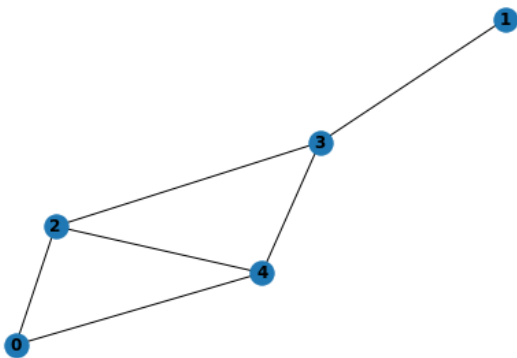
[ ] def create_graph(V,p):
    myflag = True
    while myflag:
        G = erdos_renyi_graph(V, p)
        myflag = check_connected(G)

    edges = list(G.edges)
    return G, V, edges

[ ] def draw_graph(G):
    nx.draw(G, with_labels=True, font_weight='bold')
```

Test Graphs and Results with Exact Brute Force Algorithm:

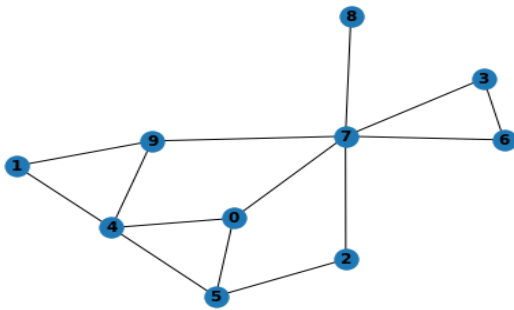
First Test with 5 vertices



Maximum Independent Set: {1, 4}

Corresponding Cardinality of the Maximum Independent Set: 2

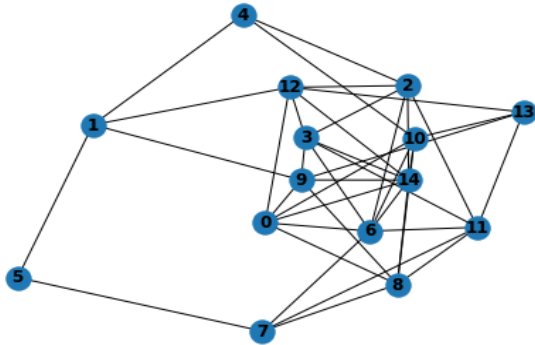
Second Test with 0 vertices



Maximum Independent Set: {0, 2, 6, 8, 9}

Corresponding Cardinality of the Maximum Independent Set: 5

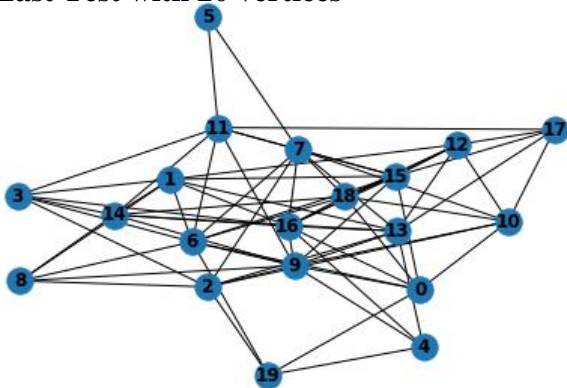
Third Test with 15 vertices



Maximum Independent Set: {4, 5, 9, 11, 12}

Corresponding Cardinality of the Maximum Independent Set: 5

Last Test with 20 vertices



Maximum Independent Set: {3, 5, 8, 10, 13, 15, 19}

Corresponding Cardinality of the Maximum Independent Set: 7

B.

Random Graph Generation Algorithm

```
def reorganize_graph(G):
    edges = list(G.edges)
    nodes = vertice_num
    random_vertices = []
    random_edges = []

    for elm in edges:
        elm1 = elm[0]
        elm2 = elm[1]
        random_edges.append([elm1,elm2])

    for i in range(nodes): # Generating vertices
        vex = vertice(i,[],0)
        random_vertices.append(vex)

    for i in range(len(random_edges)): # Adding neighbouts to points
        id_1 = random_edges[i][0]
        id_2 = random_edges[i][1]

        for j in range(len(random_vertices)):
            if random_vertices[j].id == id_1 and id_2 not in random_vertices[j].next_vertices:
                random_vertices[j].next_vertices.append(id_2)
            if random_vertices[j].id == id_2 and id_1 not in random_vertices[j].next_vertices:
                random_vertices[j].next_vertices.append(id_1)
        path = []
        #print(check_path_func(random_vertices, 0 , path))
        for i in range(len(random_vertices)):
            print(random_vertices[i].id, random_vertices[i].next_vertices , random_vertices[i].independent)
        graph = random_vertices
        return graph
graph = reorganize_graph(G)
```

Test Graphs and Results with approximation algorithm:

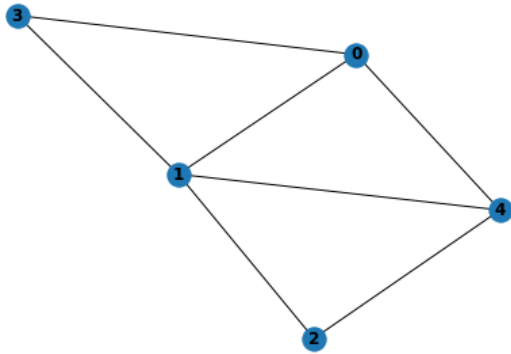
Notation for below examples:

First column: Vertex id

Second column: Neighbors

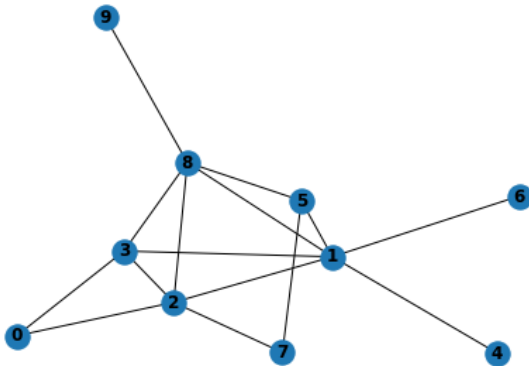
Third Column: Independency (It is 1 if the corresponding Vertex is in the independent set)

First Test with 5 Vertices:



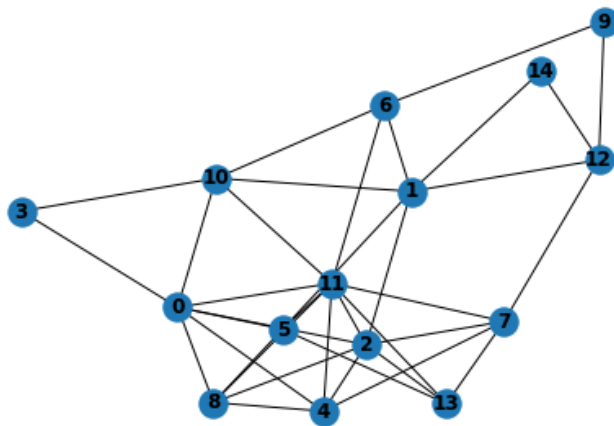
```
0 [3, 4, 1] 1
1 [2, 3, 0, 4] -1
2 [4, 1] 1
3 [0, 1] -1
4 [2, 0, 1] -1
Max Cardinality is: 2
```

Second Test with 10 Vertices:



```
0 [3, 2] -1
1 [4, 6, 5, 3, 2, 8] -1
2 [0, 7, 3, 8, 1] 1
3 [0, 2, 8, 1] -1
4 [1] 1
5 [7, 8, 1] 1
6 [1] 1
7 [5, 2] -1
8 [9, 5, 3, 2, 1] -1
9 [8] 1
Max Cardinality is: 5
```

Last Test With 15 Vertices:



```
0 [3, 4, 5, 8, 10, 2, 11] -1
1 [14, 6, 12, 10, 5, 2] -1
2 [13, 4, 7, 8, 1, 0, 11] 1
3 [10, 0] 1
4 [7, 8, 0, 2, 11] -1
5 [13, 8, 1, 0, 11] 1
6 [9, 10, 1, 11] 1
7 [12, 13, 4, 2, 11] -1
8 [4, 5, 0, 2, 11] -1
9 [6, 12] -1
10 [3, 6, 1, 0, 11] -1
11 [6, 13, 4, 5, 7, 8, 10, 0, 2] -1
12 [9, 14, 7, 1] 1
13 [5, 7, 2, 11] -1
14 [12, 1] -1
Max Cardinality is: 5
```

5. Experimental Analysis of the Performance (Performance Testing)

In the performance testing, the implication of the Central Limit Theorem is followed by calculating sample mean (m) and standard deviation (sd) from predetermined N measurements.

Estimated Standard Error is also calculated according to below formula:

$$\text{Estimated Standard Error (S}_m\text{)} = \frac{\text{Sample Standard Deviation (sd)}}{\sqrt{N}}$$

At the end, the mean running time of all possible runs (M) can be seen in the below confidence interval:

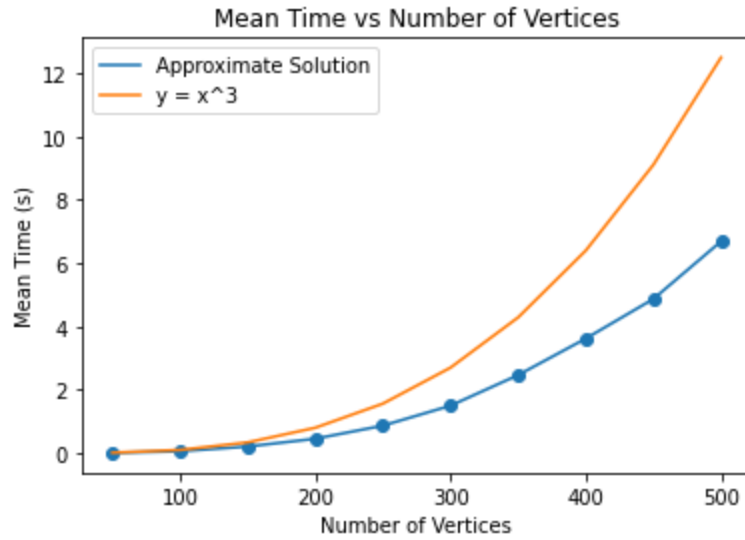
$$\text{Population Mean (M)} = [m - t \times S_m, m + t \times S_m] \text{ with probability CL\%}$$

where CL corresponds to the Confidence Level, t is a parameter from t-table.

Firstly, relation between **mean running time and the number of vertices** is investigated with 90% and 95% Confidence Level, and N = 20 runs of our program for a certain number of vertices while having constant Edge Density with the parameter p = 0.4. Corresponding t values for both confidence levels are 1.729 and 2.093 respectively.

Results can be seen below table and plot:

Vertex Number	Mean Time (s)	Std Deviation	Std Error	CL 90% Lower Bound	CL 90% Upper Bound	CL 95% Lower Bound	CL 95% Upper Bound
50	0.00861741	0.000843866	0.000188694	0.00829116	0.00894367	0.00822248	0.00901235
100	0.0580617	0.00189896	0.00042462	0.0573275	0.0587959	0.057173	0.0589504
150	0.20712	0.0473171	0.0105804	0.188827	0.225414	0.184976	0.229265
200	0.453413	0.0357836	0.00800146	0.439578	0.467247	0.436666	0.47016
250	0.867274	0.0463006	0.0103531	0.849374	0.885175	0.845605	0.888943
300	1.4987	0.13611	0.0304351	1.44607	1.55132	1.435	1.5624
350	2.46646	0.464046	0.103764	2.28706	2.64587	2.24929	2.68364
400	3.61949	0.423505	0.0946986	3.45576	3.78323	3.42129	3.8177
450	4.87133	0.0458208	0.0102458	4.85362	4.88905	4.84989	4.89278
500	6.68989	0.046165	0.0103228	6.67204	6.70773	6.66828	6.71149

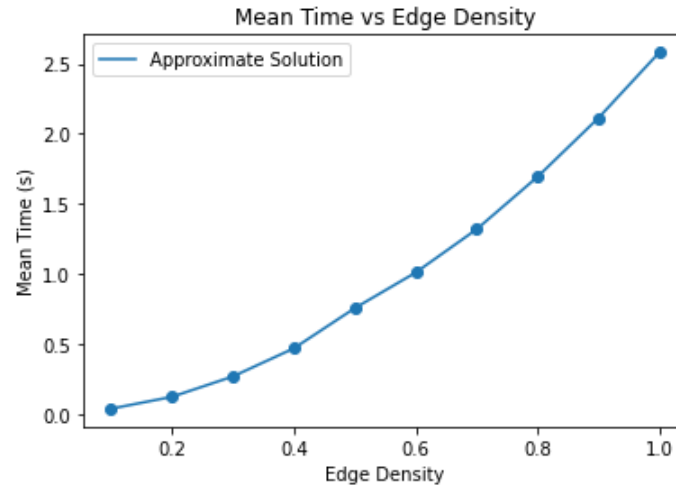


Non-surprisingly, mean running time of the algorithm increases as the number of vertices are increasing. It can be seen that our time complexity follows $O(V^3)$ as proposed.

Secondly, relation between **mean running time and the Edge Density** is investigated with 90% and 95% Confidence Level, and $N = 20$ runs of our program for a certain number of vertices while having constant number of vertices as 200. Corresponding t values for both confidence levels are 1.729 and 2.093 respectively.

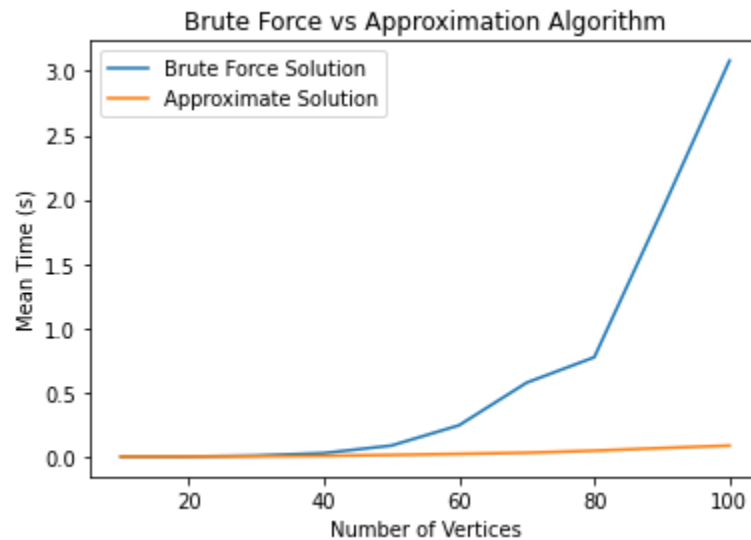
Results can be seen below table and plot:

Edge Density	Mean Time (s)	Std Deviation	Std Error	CL 90% Lower Bound	CL 90% Upper Bound	CL 95% Lower Bound	CL 95% Upper Bound
0.1	0.0407324	0.0135249	0.00302425	0.0355034	0.0459613	0.0344026	0.0470621
0.2	0.125333	0.022642	0.00506291	0.116579	0.134087	0.114736	0.13593
0.3	0.270726	0.042873	0.00958668	0.254151	0.287302	0.250661	0.290791
0.4	0.470811	0.0644964	0.0144218	0.445876	0.495746	0.440626	0.500996
0.5	0.75745	0.132896	0.0297163	0.706071	0.80883	0.695254	0.819647
0.6	1.01129	0.118959	0.0266	0.965301	1.05728	0.955619	1.06697
0.7	1.3196	0.140825	0.0314893	1.26515	1.37404	1.25369	1.38551
0.8	1.69372	0.0680565	0.0152179	1.66741	1.72004	1.66187	1.72557
0.9	2.11339	0.0217157	0.00485578	2.10499	2.12179	2.10323	2.12355
1	2.58131	0.0145831	0.00326089	2.57567	2.58695	2.57448	2.58813



It can be observed that mean running time of the algorithm increases as the Edge Density is increasing. It confirms our proposal that sorting ('next_vertice_sort') is the most time-consuming part in our algorithm.

Lastly, we also compared both exact and approximation algorithms' running times with different number of vertices. Corresponding result can be seen below plot:



It can be seen that brute force solution dominates the approximation solutions as its time complexity is exponential time compared to polynomial time.

To analyze ratio bound experimentally, quality measure can be used with the help Brute Force algorithm. Quality of the Approximation algorithm will be calculated as following way with 50 tries each time for same number of vertices:

$$Quality = \frac{\text{Cardinality of Approximation Algorithm}}{\text{Cardinality of Brute Force Algorithm}}$$

Number of Vertices	Quality	Min Difference of Cardinalities	Max Difference of Cardinalities
5	1	0	0
6	0.999	0	1
7	0.9935	0	2
8	0.982667	0	1
9	0.97475	0	1
10	0.955667	0	1

As the number of vertices is increasing, the quality of the heuristic algorithm is decreasing.

We can also observe that Heuristic algorithm usually finds an independent set with the size equals to maximum independent set size – 1.

6. Experimental Analysis of the Correctness (Correctness Testing)

Proposed approximation algorithm may find an independent set which is not a maximum independent set for a given graph. However, its result is always an independent set which is explained in the algorithm analysis part. Therefore, we can say that approximation algorithm is correct. To prove this argument, approximation algorithm is executed for 200 times with 100 number of vertices which is quite big graph. Then, each time result of the algorithm is checked whether it is an independent set or not.

Showing Independency of Result Implementation:

In the implementation, networkx's hasedge() function is utilized to check if there is an edge between a vertex and the other vertex in the result of approximation algorithm.

```
result = []
vertex_num = 100
for _ in range(200):
    independent_vertices = []
    G,v, edges = create_graph(vertex_num, 0.5)
    graph = reorganize_graph(G)
    min_id= find_min_element(graph)
    graph = next_vertex_sort(graph)
    graph_edited = find_max_independent(graph,min_id,0,0)
    for i in range(len(graph_edited)):
        if graph[i].independent == 1:
            independent_vertices.append(i)
    for i in range(len(independent_vertices)):
        for k in range(i+1, len(independent_vertices)):
            if G.has_edge(independent_vertices[i], independent_vertices[k]):
                result.append(1)
            else:
                result.append(0)
if sum(result) == 0:
    print("All Results are Independent SET")
else:
    print("WHOOOPS")
```

All Results are Independent SET

It can be seen that our algorithm is correct!

Then, we performed Black Box testing to check correctness of our algorithm with the help of Exhaustive Search algorithm.

We tried extreme cases to see whether approximation algorithm is correct or not as following examples:

Test 1: Empty Graph

```
Solution Set: []  
Max Cardinality by Approximation Algorithm: 0  
Cardinality of Optimal Solution: 0
```

It is obvious that the result is correct.

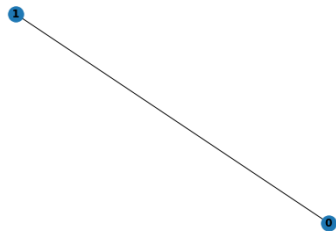
Test 2: 1 Vertex Only Graph



```
0 [] 1  
Solution Set: [0]  
Max Cardinality by Approximation Algorithm: 1  
Cardinality of Optimal Solution: 1
```

It is obvious that the result is correct.

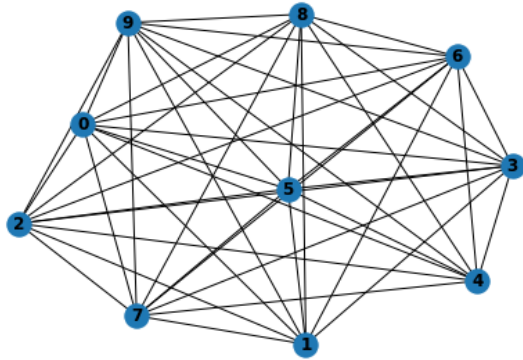
Test 3: 2 Vertices Graph



```
0 [1] 1  
1 [0] -1  
Solution Set: [0]  
Max Cardinality by Approximation Algorithm: 1  
Cardinality of Optimal Solution: 1
```

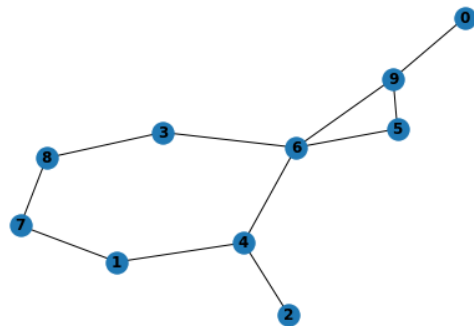
It is obvious that the result is in all possible maximum independent sets. Thus, it is correct.

Test 4: 10 Vertices with high Dense Edges Graph



```
0 [1, 4, 3, 2, 5, 6, 7, 8, 9] -1
1 [0, 2, 3, 5, 6, 7, 8, 9] 1
2 [1, 4, 3, 0, 5, 6, 7, 8, 9] -1
3 [1, 4, 2, 0, 5, 6, 7, 8, 9] -1
4 [0, 2, 3, 5, 6, 7, 8, 9] 1
5 [1, 4, 2, 3, 0, 6, 7, 8, 9] -1
6 [1, 4, 2, 3, 0, 5, 7, 8, 9] -1
7 [1, 4, 2, 3, 0, 5, 6, 8, 9] -1
8 [1, 4, 2, 3, 0, 5, 6, 7, 9] -1
9 [1, 4, 2, 3, 0, 5, 6, 7, 8] -1
Solution Set: [1, 4]
Max Cardinality by Approximation Algorithm: 2
Cardinality of Optimal Solution: 2
```

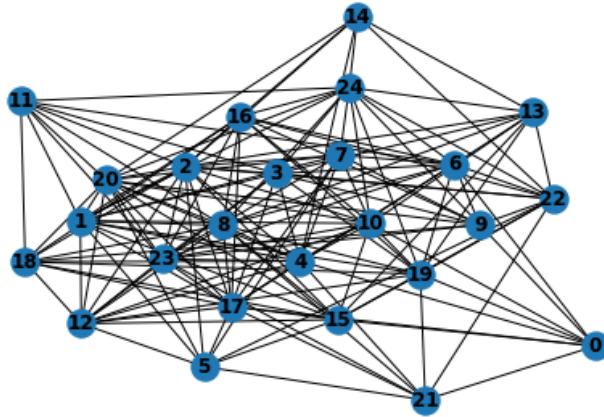
Test 5: 10 Vertices with low Dense Edges Graph



```
0 [9] 1
1 [7, 4] 1
2 [4] 1
3 [8, 6] -1
4 [2, 1, 6] -1
5 [9, 6] 1
6 [3, 5, 4, 9] -1
7 [1, 8] -1
8 [3, 7] 1
9 [0, 5, 6] -1
Solution Set: [0, 1, 2, 5, 8]
Max Cardinality by Approximation Algorithm: 5
Cardinality of Optimal Solution: 5
```

It can be seen that algorithm works fine with same number vertices and different density of edges by looking both test 4 and test 5.

Test 6: 25 Vertices Graph



```

0 [21, 9, 10, 15, 6, 19, 4, 17] -1
1 [5, 11, 7, 10, 3, 12, 15, 16, 8, 19, 24, 4, 20, 17, 2] -1
2 [14, 5, 11, 13, 7, 18, 22, 12, 3, 15, 16, 10, 23, 1, 4, 20, 17] -1
3 [11, 9, 13, 22, 12, 23, 19, 6, 24, 4, 1, 17, 2] -1
4 [14, 0, 21, 7, 18, 3, 10, 12, 23, 6, 8, 1, 20, 17, 2] -1
5 [21, 7, 22, 12, 15, 1, 20, 17, 2] -1
6 [0, 11, 9, 13, 7, 22, 3, 16, 23, 19, 8, 24, 4, 17] -1
7 [5, 9, 10, 23, 6, 24, 4, 1, 20, 17, 2] -1
8 [21, 11, 13, 18, 12, 15, 16, 6, 19, 24, 1, 4, 17, 20] -1
9 [0, 7, 18, 22, 16, 3, 23, 6, 24, 20] 1
10 [0, 13, 7, 12, 15, 16, 23, 19, 24, 1, 4, 20, 2] 1
11 [18, 3, 23, 6, 8, 24, 1, 20, 2] 1
12 [5, 18, 3, 10, 15, 23, 19, 8, 4, 1, 17, 20, 2] -1
13 [14, 22, 10, 15, 3, 8, 19, 6, 24, 2] -1
14 [13, 22, 16, 24, 4, 20, 2] 1
15 [0, 5, 13, 18, 22, 10, 12, 19, 8, 1, 20, 17, 2] -1
16 [14, 9, 18, 22, 10, 23, 6, 19, 8, 24, 1, 17, 2] -1
17 [0, 21, 5, 7, 18, 3, 12, 15, 16, 23, 8, 6, 4, 1, 20, 2] -1
18 [11, 9, 12, 15, 16, 8, 19, 4, 17, 20, 2] -1
19 [0, 21, 13, 18, 22, 10, 12, 3, 15, 16, 6, 8, 24, 1] -1
20 [14, 5, 11, 9, 7, 18, 10, 12, 15, 23, 8, 24, 1, 4, 17, 2] -1
21 [0, 5, 22, 23, 19, 8, 4, 17] 1
22 [14, 21, 5, 9, 13, 15, 16, 3, 19, 6, 24, 2] -1
23 [21, 11, 9, 7, 10, 3, 12, 16, 6, 4, 17, 20, 2] -1
24 [14, 11, 9, 13, 7, 22, 10, 3, 16, 8, 19, 6, 1, 20] -1
Solution Set: [9, 10, 11, 14, 21]
Max Cardinality by Approximation Algorithm: 5
Cardinality of Optimal Solution: 6

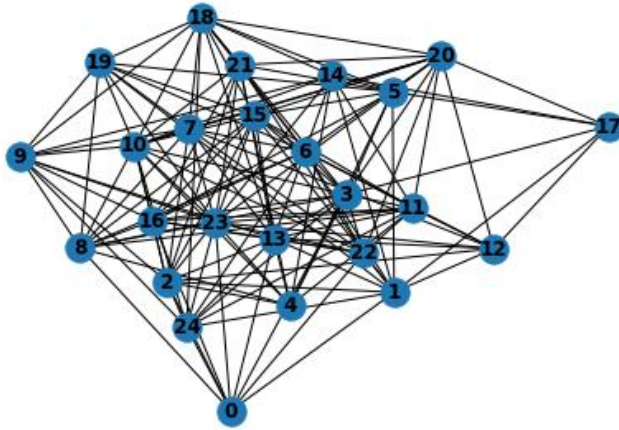
```

Algorithm could not find optimal solution, but it found an independent set within the ratio

bound. Ratio bound equals to $\frac{14}{5} + 1 \approx 4$ and $\frac{C^*}{C} = \frac{6}{5} = 1.2$. It is clear that $1.2 < 4$. Therefore,

approximation algorithm correctness holds.

Test 7: Repeat Test 6 to see it can find an optimal solution for 25 vertices Graph



```

0 [24, 8, 22, 1, 16, 2, 23, 3, 13] -1
1 [17, 12, 0, 14, 19, 24, 5, 22, 2, 6, 15, 13] -1
2 [0, 12, 4, 14, 9, 21, 24, 7, 1, 16, 15, 10, 13] -1
3 [17, 0, 4, 24, 8, 5, 20, 22, 16, 18, 11, 15, 6, 13, 10] -1
4 [14, 21, 5, 8, 11, 2, 23, 3, 10, 13] 1
5 [17, 4, 14, 19, 21, 1, 20, 18, 16, 15, 23, 3] -1
6 [12, 19, 21, 24, 1, 20, 7, 22, 8, 11, 16, 18, 15, 3] -1
7 [12, 19, 9, 8, 20, 22, 18, 2, 6, 23, 13, 10] 1
8 [0, 4, 19, 9, 7, 11, 16, 6, 15, 23, 3, 10] -1
9 [14, 19, 24, 7, 8, 2, 16, 18, 23, 10, 13] -1
10 [4, 14, 19, 9, 21, 24, 20, 8, 22, 7, 2, 16, 18, 23, 15, 3] -1
11 [12, 4, 14, 24, 20, 22, 8, 16, 15, 23, 6, 3, 13] -1
12 [17, 7, 1, 20, 11, 2, 6, 23, 13] -1
13 [0, 12, 4, 14, 9, 21, 24, 1, 7, 22, 16, 18, 11, 2, 15, 3] -1
14 [17, 4, 9, 5, 1, 11, 2, 18, 10, 13] -1
15 [19, 5, 8, 1, 20, 11, 16, 18, 2, 6, 23, 3, 10, 13] -1
16 [0, 9, 21, 5, 8, 22, 2, 11, 15, 6, 3, 10, 13] 1
17 [12, 14, 5, 1, 20, 3] 1
18 [14, 19, 9, 21, 24, 5, 20, 7, 15, 6, 3, 13, 10] -1
19 [9, 5, 7, 8, 1, 18, 15, 6, 23, 10] -1
20 [17, 12, 21, 7, 5, 22, 18, 11, 6, 15, 3, 10] -1
21 [4, 5, 20, 22, 16, 18, 2, 6, 23, 13, 10] -1
22 [0, 21, 7, 20, 1, 11, 16, 6, 23, 3, 13, 10] -1
23 [0, 12, 4, 19, 9, 21, 24, 5, 7, 8, 22, 11, 15, 10] -1
24 [0, 9, 1, 2, 11, 18, 6, 23, 3, 13, 10] 1
Solution Set: [4, 7, 16, 17, 24]
Max Cardinality by Approximation Algorithm: 5
Cardinality of Optimal Solution: 5

```

We can observe that approximation algorithm may find the optimal solution or not. However, all results are independent sets.

7. DISCUSSION

The approximation algorithm is designed and implemented for the **Maximum Independent Set** problem which is an NP-Hard problem in polynomial time $O(V^3)$ compared to brute force exponential time $O(2^V)$. Since proposed algorithm is a **Heuristic algorithm**, it may not find the optimal solution for given graph as there is a trade-off between the runtime and finding optimal solution. The correctness and ratio bound of the algorithm is shown in the algorithm description and analysis parts. Sample generation is done by utilizing Networkx library of Python. Then, the approximation algorithm is analyzed experimentally. Firstly, it gone through **the performance testing** to demonstrate its running time for different scenarios (changing vertex number & edge density) and comparison between brute force algorithm by using the implication of **Central Limit Theorem**. It is found that algorithm has slightly better upper bound for time complexity than theoretical $O(V^3)$. Also, algorithm's quality for different number of vertices shown in the performance testing part where the quality decreases as the number of vertices increasing in the graph. Secondly, the algorithm gone through **the functionality test** to demonstrate its correctness. In the functionality test, proposition of the result of the algorithm's is an independent set is proved by experimentally. Then, **Black Box** testing is used with number of different extreme cases to show algorithm is working correctly.

REFERENCES

[1]Eddy, D., & Kochenderfer, M. J. (2021). A Maximum Independent Set Method for Scheduling Earth-Observing Satellite Constellations. *Journal of Spacecraft and Rockets*, 1-14.

[2]Yan, Chengxiao & Berube, Rachel & Damodara, Srikant. (2020). The Independent Set and NP- Completeness.

[3]<http://www.cs.umd.edu/class/fall2017/cmsc451-0101/Lects/lect20-np-3sat.pdf>

[4]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).

[5] <https://www.geeksforgeeks.org/maximal-independent-set-in-an-undirected-graph/>