Empirical Analysis Of Approximation Algorithms

(CS F376)

Under the supervision and guidance of :

Prof. Pritam Bhattacharya

Prepared by :

- ➢ **Karan Shetty** 2018A7PS0111G
  (f20180111@goa.bits-pilani.ac.in)
- ➢ **Shubh Shah** 2018A7PS0092G
  (f20180092@goa.bits-pilani.ac.in)
- ➢ **Rishit Patel** 2018A7PS0189G
  (f20180189@goa.bits-pilani.ac.in)

Acknowledgement

We would like to thank Prof. Pritam Bhattacharya for constant guidance and teaching support throughout the project which made it possible for us to implement and perform necessary analysis of the important algorithms for the Vertex Cover, Set Cover and Travelling Salesman Problem.

Table of contents

Minimum Vertex Cover

Minimum Set Cover

Travelling salesman problem

- Introduction
- Import
- Graph Generation
- Exact Algorithms
    - Brute Force
    - Integer Linear Programming
- Approximation Algorithms
    - MST Approximation
    - Christofides Approximation
- Functions for comparison
- Example Instance
- Comparison of Algorithms
    - Execution Time
    - Performance Ratios

Minimum Vertex Cover

Introduction

Vertex Cover (VC) Problem:

- **[Instance]** Graph $G(V, E)$.
- **[Feasible Solutions]** A subset $C \subseteq V$ such that for all $e = u, v \in$ E , $e \cap C \neq \phi$.
- **[Value]** The value of a solution is the size of the cover $|C|$, and the goal is to minimize it.

Weighted Vertex Cover Problem:

- **[Instance]** Graph $G(V, E)$ and a positive integer weight function $w : V \rightarrow Z$ on the vertices.
- **[Feasible Solutions]** A subset $C \subseteq V$ such that $\forall e = u, v \in E, e \cap C \neq \phi$.
- **[Value]** The value of a solution is the weight of the cover: $w(C)^*$, and the goal is to minimize it.

Implementation:
We use three metrics for the comparision of algorithms:

1. Performance Ratio: ratios of lengths of vertex covers (wrt Brute force or ILP).
2. Execution time of various algorithms.
3. Hamming distance of brute force vertex cover with other algorithms.

Hamming distance is a metric for comparing two binary data strings. While comparing two binary strings of equal length, Hamming distance is the number of bit positions in which the two bits are different.

For example:
For 6 vertices (numbered 0 to 5),

- Optimal (brute force) vertex cover: {1,2}
- Vertex cover according to G1 Algorithm: {1,3,5}

Characteristic Vector:

- Optimal: 011000
- G1 Algo: 010101

Hamming Distance: 3 (3 locations where bits are different - 2,3,5)

In order to calculate the Hamming distance between two strings a and b, we perform the XOR operation, (a⊕b), and then count the total number of 1s in the resultant string.

Import

In [ ]:
```
%%capture
import networkx as nx
import matplotlib.pyplot as plt
import pandas as pd
from tabulate import tabulate
import random
import numpy as np
import scipy
from scipy.optimize import linprog
from mpl_toolkits.mplot3d import Axes3D
from math import inf
from math import ceil
import time
import itertools
import warnings
warnings.filterwarnings("ignore")
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('pdf','svg')
```

In [ ]:
```
%%capture
!pip install mip
from mip import Model, xsum, minimize, BINARY
```

Exact Algorithms

Brute Force

Algorithm Brute Force

Input: Graph G (with vertex set V and edge set E). [Networkx graph object]
Output: Minimum vertex cover.

1. Iterate through all subsets of vertex in order of size.
2. Check if our subset is a vertex cover.
3. Calculate the cost function(size of vertex cover) and store it's minimum.

Implementation:

- Initial implementation was generation of all permutations of all lengths, which took more time.
- This was later pruned by generating permutations in increasing order of length, and breaking when we get a vertex cover (this will, by default be the min vertex cover).
- Graph is dict {1:[2,3,4], 2: [1,3,4,5]}
- Subset is a list [5,6,7,8]

Parameterization:

- *ALL* denotes whether to generate all possible minimum vertex covers. This will be used in finding minimum hamming distance wrt another vertex cover.

In [ ]:
```
def check_if_vertex_cover(graph, subset):
    for vertex in graph.keys():
        if vertex in subset:
            continue
        for adjacent_vertex in graph[vertex]:
            if adjacent_vertex not in subset:
                return False

    return True

class Brute:
    minSize=100000
    vertex_cover=[]
    all_vertex_covers = []

def brute_force(graph_object, ALL = False):
    brut = Brute()
    # Convert into an adjacency list
    graph = nx.to_dict_of_lists(graph_object)
    V = [x for x in graph.keys()]
    # Generate all Subsets and check
    found = False
    for i in range(1,len(V)+1):

        # Generate all subsets of a given length i
        subsets_i = list(itertools.combinations(V, i))
        for subset in subsets_i:
            if (check_if_vertex_cover(graph,subset) == True):
                found = True
                brut.vertex_cover = list(subset[:])
                if (ALL == True):
                    brut.all_vertex_covers.append(brut.vertex_cover)
        if found == True:
            break
    if ALL == False:
        return brut.vertex_cover
    else:
        return brut.all_vertex_covers
```

In [ ]:
```
# Plots the networkx graph
def plot_graph(example_graph_object):
    example_graph = nx.to_networkx_graph(example_graph_object)
    pos = nx.spiral_layout(example_graph, resolution=0.7)  # positions for all nodes
    nx.draw(example_graph, pos, with_labels=True)
    labels = nx.get_edge_attributes(example_graph,'weight')
    nx.draw_networkx_edge_labels(example_graph,pos,edge_labels=labels)
    plt.axis()
    plt.show()
```

Integer Linear Programming

Input: Graph G (with vertex set V and edge set E). [Networkx graph object]
Output: Minimum vertex cover.

$$\min \sum_{i=1}^{m} w_v x_v, \tag{1}$$
$$\text{subject to} \tag{2}$$
$$x_u + x_v \geq 1 \, , \forall e = (u, v) \in U \tag{3}$$
$$x_v \in \{0, 1\}, \forall v \in V \tag{4}$$

Notation

$$x_v = \begin{cases} 1 & \text{if the vertex } v \text{ is included in the set cover,} \\ 0 & \text{otherwise.} \end{cases}$$

w_v is the weight of the vertex v.

The first inequality implies that for each edge, atleast one vertex is included in the vertex cover.

Implementation

- If no weights are passed, we create an array with all weights = 1 to have a common implementation for weighted and unweighted case.

In [ ]:
```
def ilp(graph_object, weights):
    m = Model("ILP-Vertex Cover")
    E = list(graph_object.edges())
    # Convert into an adjacency list
    graph = nx.to_dict_of_lists(graph_object)
    n = len(graph.keys())

    # Xi's || vertex - whether to include or not
    x = [m.add_var(var_type=BINARY) for i in range(n)]

    # If no weights are passed, initialize the weights to 1
    if len(weights) == 0:
        for i in range(n):
            weights.append(1)

    # objective function: minimize the cost
    m.objective = minimize(xsum(x[i]*weights[i] for i in range(n)))

    # constraints: for each edge, atleast one vertex is included in the vertex cover
    for edge in E:
        m += (x[edge[0]] + x[edge[1]]) >= 1

    # optimizing
    m.optimize()

    # converting indices to vertex cover
    vertex_cover = [i for i in range(n) if x[i].x >= 0.99]

    return vertex_cover
```

Approximation Algorithms

Maximal Matching

Algorithm Maximal Matching

Input: Graph G (with vertex set V and edge set E). [Networkx graph object]
Output: Minimum vertex cover.

1. Pick any maximal matching M which is a subset of E in G.
2. Add all vertices matched in M to C.
3. Return C.

Performance Ratio: 2

- Since M is a maximal matching, all edges in $E - M$ are such that at least one of their end-points is incident to some $e \in M$ (otherwise, that edge could be added to M to provide a larger matching).
- Thus every edge in E has at least one end-point in C.
- To see that the ratio is 2, consider the edges in M. To cover these edges we need at least $|M|$ vertices, since no two of them share a vertex.
- This implies that the optimal vertex cover has size at least $|M|$.
- The cover C contains exactly $2 * |M|$ vertices.

Implementation:

- We find the maximal matching using networkx's max_weight_matching function.

In [ ]:
```
def mm_algo(graph_object):
    # Set of edges
    maximal_matching = nx.max_weight_matching(graph_object)
    C = []
    for edge in maximal_matching:
        C.append(edge[0])
        C.append(edge[1])

    return C
```

G1 Approximation

Algorithm G1

Input: Graph G (with vertex set V and edge set E). [Networkx graph object]
Output: Minimum vertex cover.

1. Initialize C to an empty set.
2. While E is not empty:
   A. Pick any edge e which belongs to E and choose an end-point v of e.
   B. Add this vertex v to C.
   C. Remove all the edges from E whose one end-point is v.
3. Return C.

Performance Ratio: O(log(|V|))

- Consider the following bipartite graph $B = (L, R, E)$.
- The vertex set L consists of r vertices. The vertex set R is further sub-divided into r sets called $R_1, \ldots, R_r$.
- Each vertex in R_i has an edge to i vertices in L and no two vertices in R_i have a common neighbour in L; thus, $|R_i| = \lfloor r/i \rfloor$.
- It follows that each vertex in L has degree at most r and each vertex in R_i has degree i.
- The total number of vertices $n = \theta(rlogr)$.
- Suppose that (out of sheer bad luck) the algorithm considers an edge out of R_r first, choosing the end-point in R as the vertex to be placed in the cover.
- Then it picks an edge out of R_{r-1}, again choosing its end-point in R for the cover C; and, so on.
- Therefore the vertex cover chosen is $C = R$.
- But L is itself a vertex cover since the graph is bipartite.
- It follows that the ratio achieved by this algorithm is no better than $|R|/|L| = \Omega(logn)$.

In [ ]:
```
def g1_algo(graph_object):
    vertex_cover = []
    # Convert into an adjacency list
    graph = nx.to_dict_of_lists(graph_object)
    # Iterate through the adjacency list
    for vertex,adj_vertices in graph.items():
        if len(adj_vertices) != 0:
            # Add to vertex cover
            vertex_cover.append(vertex)

            # Remove all edges incident on this vertex
            for adj_vertex in adj_vertices:
                if vertex in graph[adj_vertex]:
                    graph[adj_vertex].remove(vertex)

    return vertex_cover
```

G2 Approximation

Algorithm G2

Input: Graph G (with vertex set V and edge set E). [Networkx graph object]
Output: Minimum vertex cover.

1. Initialize C to an empty set
2. While E is not empty:
   A. Pick a vertex v of maximum degree that belongs to V in the current graph.
   B. Add this vertex v to C.
   C. Remove all the edges from E whose one end-point is v.
3. Return C

Performance Ratio: O(log(|V|))

- Let us consider the behaviour of this algorithm on the graph B (shown in G1).
- It should be easy to see that G2 could also output R as a vertex cover.
- It could choose vertices from R_r at the very first stage.
- After this, it could choose vertices from R_{r-1}. In general, it would choose the highest degree vertices from R at each stage.
- It is very surprising that a seemingly much more intelligent heuristic does no better than the rather simple-minded heuristic G1.

In [ ]:
```
def g2_algo(graph_object):
    vertex_cover = []
    # Convert into an adjacency list
    graph = nx.to_dict_of_lists(graph_object)

    # Sort graph according to size of item:
    sorted_keys = sorted(graph, key=lambda k: len(graph[k]), reverse=True)

    # Iterate through the sorted adjacency list
    for vertex in sorted_keys:
        if len(graph[vertex]) != 0:
            # Add to vertex cover
            vertex_cover.append(vertex)

            # Remove all edges incident on this vertex
            for adj_vertex in graph[vertex]:
                if vertex in graph[adj_vertex]:
                    graph[adj_vertex].remove(vertex)

    return vertex_cover
```

RA Approximation

Input: Graph G (with vertex set V and edge set E). [Networkx graph object]
Output: Minimum vertex cover.

1. Order the edges in E arbitrarily.
2. While E is not empty
   A. Pick the next edge e which belongs to E with end points u and v.
   B. Flip a fair coin to choose x uniformly from u and v.
   C. Add this vertex v to C.
   D. Remove all the edges from E whose one end-point is v.
3. Return C.

Performance Ratio: 2

- Let us fix an input graph $G(V, E)$, the order in which the edges are to be examined and some optimal cover $C^* \subseteq V$.
- Suppose that this algorithm outputs a cover C with t vertices in it.
- Clearly, this algorithm examines exactly t edges and flips as many coins in the course of its execution.
- Let us define the outcome of a coin flip as being *good* if it causes some vertex $v \in C^*$ to enter the cover C.
- Note that every edge has at least one end-point in C^* and so each coin flip is good with probability at least a half.
- But the number of good coin flips cannot exceed $c^* = C^*$, since by then all the vertices of C^* are in C and every edge in G must be covered by C.
- Thus, the total number of coin flips t is stochastically dominated by the number of unbiased coin flips needed to obtain c^* *good* coin flips.
- It follows that the expected number of coin flips needed is no more than $2 * c^*$.
- This implies the desired bound on the expected value of the performance ratio.

Implementation:

- Since RA is a random algorithm, we run it multiple times for the same graph to even it out.

In [ ]:
```
def ra_algo(graph_object, weights = [], WEIGHTED = False):
    vertex_cover = []

    # Edges gives the edgelist of the graph
    edges = list(graph_object.edges())

    while len(edges) > 0:
        # Pick u,v uniformly
        pick_prob = random.random()

        u = edges[0][0]
        v = edges[0][1]

        # If weighted, pick with a weighted prob
        if (WEIGHTED == True):
            PROB = weights[v] / (weights[u] + weights[v])
        # If unweighted, uniform prob
        else:
            PROB = 0.5

        if (pick_prob >= PROB):
            vertex = u
        else:
            vertex = v

        # Add to vertex cover
        vertex_cover.append(vertex)

        # Remove all edges incident on this vertex
        edges = [edge for edge in edges if vertex not in edge]
    return vertex_cover
```

Parametrization:

- *RUNS* denotes the number of runs for the same graph
- *PRINT* is used to print the standard deviation and average over all the runs.

In [ ]:
```
def multiple_ra_algo(graph_object,weights = [], WEIGHTED = False, RUNS = 30, PRINT = False):
    vertex_cover_lengths = 0
    info = []

    # Run RUNS times
    for i in range(RUNS):
        vertex_cover = ra_algo(graph_object)
        if WEIGHTED == True:
            cost = sum([weights[x] for x in vertex_cover])
        else:
            cost = len(vertex_cover)
        vertex_cover_lengths += cost
        info.append(cost)
    avg = vertex_cover_lengths/RUNS
    std_dev = np.std(np.array(info))

    if PRINT:
        plt.plot(info)
        plt.title("Length vs Runs")
        plt.xlabel("Number of Runs")
        plt.ylabel("Length of Vertex Cover")
        print("Standard Deviation:",std_dev)
        print("Average over",RUNS,"Runs:",avg,"\n")
    return avg,std_dev
```

Relaxed Linear Programming

Input: Graph G (with vertex set V and edge set E). [Networkx graph object]
Output: Minimum vertex cover.

$$\min \sum_{i=1}^{m} w_v x_v, \tag{5}$$
$$\text{subject to} \tag{6}$$
$$x_u + x_v \geq 1 \, , \forall e = (u, v) \in U \tag{7}$$
$$x_v \geq 0 \, , \forall v \in V \tag{8}$$

---

1. Solve LP to obtain an optimal fractional solution x^*.
2. If x_v^* is greater than 1/2 , then add vertex v in vertex cover C.

Performance Ratio: 2

Claim 1: C is a vertex cover.
Proof:

- Consider any edge, $e = (u, v)$. By feasibility of x^*, $x_u^* + x_v^* \geq 1$, and thus either $x_u^* \geq \frac{1}{2}$ or $x_v^* \geq \frac{1}{2}$.
- Therefore, at least one of u and v will be in C.

Claim 2: $w(S) \leq 2 * \mathrm{OPT}_{LP}(I)$.
Proof:

- $\mathrm{OPT}_{LP}(I) = \sum_v w_v x_v^* \geq \frac{1}{2} \sum_{v \in S} w_v = \frac{1}{2} w(S)$.
- Therefore, $\mathrm{OPT}_{LP}(I) \geq \frac{\mathrm{OPT}(I)}{2}$ for all instances I.

Implementation

- If no weights (*obj*) are passed, we create an array with all weights = 1 to have a common implementation for weighted and unweighted case.
- Since the default operation of the inequalities in scipy are $\leq$, we append our variables with a -ve sign to make it $\geq$.
- To avoid floating point precision errors, we subtract a small value EPSILON (10^{-6}) from 0.5 (to include vertex or not).

In [ ]:
```
EPSILON = 0.000001

def rlp(graph_object, obj):
    # Edges gives the edgelist of the graph
    E = list(graph_object.edges())

    # Convert into an adjacency list
    graph = nx.to_dict_of_lists(graph_object)

    # If there are no edges in the graph, return an empty vertex cover
    if (len(E) == 0):
        return []

    # Number of vertices
    n = len(graph.keys())

    #Objective function is the weights

    # If no weights are passed, initialize the weights to 1
    if len(obj)==0:
        obj = []
        for i in range(n):
            obj.append(1)

    # For constraints
    lhs_ineq = []
    rhs_ineq = []

    for edge in E:
        # Append a one-hot vector to the left part of the inequality, with only vertices in the edge as -1, res
        arr1 = [0]*n
        arr1[edge[0]] = -1
        arr1[edge[1]] = -1
        lhs_ineq.append(arr1)

        # Append -1 to the right part of the inequality
        rhs_ineq.append(-1)

    # Bound from 0 to Infinity (xi >= 0)
    bnd = []

    for i in range(n):
        bnd.append((0,inf))

    # Optimize using the linprog function
    opt = linprog(c=obj, A_ub=lhs_ineq, b_ub=rhs_ineq, bounds=bnd)

    # A characteristic vector which is 1 if the corresponding vertex is in the vertex cover, 0 otherwise.
    vertex_in_vertex_cover = []

    # If the value of the variable x is greater than 0.5, include the corresponding vertex in the vertex cover.
    for vertex in opt.x:
        if (vertex > 0.5 - EPSILON):
            vertex_in_vertex_cover.append(1)
        else:
            vertex_in_vertex_cover.append(0)

    # converting indices to set cover
    vertex_cover = []
    i = 0
    for yes in vertex_in_vertex_cover:
        if (yes == 1):
            vertex_cover.append(i)
        i += 1

    # Check if the generated vertex cover is actually correct
    sound = check_if_vertex_cover(graph,vertex_cover)

    # If not correct, print error
    if (sound == False):
        print("Graph",graph)
        print("No of edges:",len(E))
        print("Edges:",E)
        print("OPT.X:",opt.x)
        print("rlp Vertex Cover:",vertex_cover)
        print("Is Vertex Cover:",sound)
    return vertex_cover
```

Functions for comparision

In [ ]:
```
# Colours for lines in the graph
colors = {"brute":'k',"ilp":'#2f85ed',"rlp":'#ee82ee', "RLP/ILP":'#ee82ee',"mm":'#e36634',"g1":'#1fde1f',"g2":'
# Print the datapoints in a table format
def print_table(wrt, lst, ratios, title):
    ratios[wrt] = lst
    df = pd.DataFrame.from_dict(ratios)
    df.columns = df.columns.str.strip()
    cols = df.columns.tolist()
    cols = cols[-1:] + cols[:-1]
    df = df[cols]
    del ratios[wrt]
    df.columns = pd.MultiIndex.from_product([[title], df.columns])
    return df

# Plot all the ratios in one single line graph
def plot_ratios(wrt,ratios,lst,ax,title = "Ratio of Lengths",std_dev = {}):
    for key in ratios.keys():
        ax.errorbar([x for x in range(len(lst))], ratios[key], yerr=std_dev[key], fmt='-o', label=key, color=co
    plt.sca(ax)
    ax.set_xlabel(wrt)
    ax.set_ylabel(title)
    ax.set_title(title+" vs "+wrt)
    ax.legend()
    plt.xticks(list(range(len(lst))), [ceil(100*x)/100 for x in lst])

def plot_3d(title,ps,ns,Z,ax,z_max):
    X, Y = np.meshgrid(ps, ns)
    ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none')
    ax.set_title(title);
    ax.set_xlabel('P')
    ax.set_ylabel('N')
    ax.set_zlabel('Ratio')
    ax.set_zlim(0, z_max)
    ax.view_init(60, 20)
```

Performance Ratios

Parameterization:

- *PRINT* is used to specify whether to print or not.
- *lp* is a parameter which can be specified to be "ilp" or "none".
- *COMPARE* is used to specify whether we are comparing size or time.

In [ ]:
```
# This function compares both times and costs of g1, g2, ra, mm algorithms and RLP with brute or ILP.
def compare(graph_object, RUNS_FOR_RA = 30, PRINT = False, lp = "none", COMPARE = "size"):
    t_brute_start = time.time()
    if lp == "ilp":
        vertex_cover_brute = ilp(graph_object, [])
    else:
        vertex_cover_brute = brute_force(graph_object)
    t_brute_end = time.time()

    t_g1_start = time.time()
    vertex_cover_g1 = g1_algo(graph_object)
    t_g1_end = time.time()
    t_g2_start = time.time()
    vertex_cover_g2 = g2_algo(graph_object)
    t_g2_end = time.time()
    t_mm_start = time.time()
    vertex_cover_mm = mm_algo(graph_object)
    t_mm_end = time.time()
    t_ra_start = time.time()
    vertex_cover_ra = ra_algo(graph_object,[])
    t_ra_end = time.time()
    t_rlp_start = time.time()
    vertex_cover_rlp = rlp(graph_object,[])
    t_rlp_end = time.time()

    t_brute = t_brute_end - t_brute_start
    t_g1 = t_g1_end - t_g1_start
    t_g2 = t_g2_end - t_g2_start
    t_mm = t_mm_end - t_mm_start
    t_ra = t_ra_end - t_ra_start
    t_rlp = t_rlp_end - t_rlp_start

    if COMPARE == "time":
        t_ilp_start = time.time()
        vertex_cover_ilp = ilp(graph_object,[])
        t_ilp_end = time.time()
        t_ilp = t_ilp_end - t_ilp_start
        time_dict = {'brute':t_brute,'g1':t_g1,'g2':t_g2,'mm':t_mm,'ra':t_ra, 'ilp':t_ilp, 'rlp':t_rlp}
        return time_dict

    if len(vertex_cover_brute) == 0:
        return {'g1':0,'g2':0,'mm':0,'ra':0,'rlp':0}
    g1_ratio = len(vertex_cover_g1)/len(vertex_cover_brute)
    g2_ratio = len(vertex_cover_g2)/len(vertex_cover_brute)
    mm_ratio = len(vertex_cover_mm)/len(vertex_cover_brute)
    len_vertex_cover_ra,std_dev_ra = multiple_ra_algo(graph_object,[],WEIGHTED = False, RUNS = RUNS_FOR_RA, PRI
    ra_ratio = len_vertex_cover_ra/len(vertex_cover_brute)
    rlp_ratio = len(vertex_cover_rlp)/len(vertex_cover_brute)

    if (PRINT):
        info = {}
        info["Vertex Covers"] = [vertex_cover_brute,vertex_cover_g1,vertex_cover_g2,vertex_cover_mm,vertex_cove
        info["Length"] = [len(vertex_cover_brute),len(vertex_cover_g1),len(vertex_cover_g2),len(vertex_cover_mm
        info["Ratio wrt Brute Force"] = [1,g1_ratio,g2_ratio,mm_ratio,ra_ratio,rlp_ratio]
        df = pd.DataFrame.from_dict(info)

        if lp == "ilp":
            df.index = ["ILP","G1 Algo","G2 Algo","MM Algo","RA Algo","RLP"]
        else:
            df.index = ["Brute Force","G1 Algo","G2 Algo","MM Algo","RA Algo","RLP"]
        df.index.name = "Algorithms"
        print("---------------------------------Comparision of Algorithms---------------------------------")
        display(df)

    ratio_dict = {'g1':g1_ratio,'g2':g2_ratio,'mm':mm_ratio,'ra':ra_ratio,'rlp':rlp_ratio}
    return ratio_dict
```

Hamming Distance

Implementation:

- We take the mimum hamming distance from all the possible minimum vertex covers from brute force.

Parameterization:

- *PRINT* denotes whether to print or not.

In [ ]:
```
# Find Length of XOR of 2 sets (Gives all locations with different elements)
def find_hamming_distance(vertex_cover,vertex_cover_brute):
    return len(set(vertex_cover)^set(vertex_cover_brute))

# Out of all the brute vertex covers of min_length, find the one with minimum hamming distance with the specifi
def find_min_hamming_distance_brute(vertex_cover,all_brute_vertex_covers):
    min_dist = find_hamming_distance(vertex_cover, all_brute_vertex_covers[0])
    for vertex_cover_brute in all_brute_vertex_covers:
        dist = find_hamming_distance(vertex_cover,vertex_cover_brute)
        if dist < min_dist:
            min_dist = dist

    return min_dist

# Compare
def compare_hamming_distances(graph_object, PRINT = False):
    vertex_cover_g1 = g1_algo(graph_object)
    vertex_cover_g2 = g2_algo(graph_object)
    vertex_cover_mm = mm_algo(graph_object)
    vertex_cover_ra = ra_algo(graph_object,[])
    vertex_cover_rlp = rlp(graph_object,[])

    all_brute_vertex_covers = brute_force(graph_object, ALL = True)

    min_vertex_cover_brute_g1 = find_min_hamming_distance_brute(vertex_cover_g1,all_brute_vertex_covers)
    min_vertex_cover_brute_g2 = find_min_hamming_distance_brute(vertex_cover_g2,all_brute_vertex_covers)
    min_vertex_cover_brute_mm = find_min_hamming_distance_brute(vertex_cover_mm,all_brute_vertex_covers)
    min_vertex_cover_brute_ra = find_min_hamming_distance_brute(vertex_cover_ra,all_brute_vertex_covers)
    min_vertex_cover_brute_rlp = find_min_hamming_distance_brute(vertex_cover_rlp,all_brute_vertex_covers)

    if PRINT == True:
        info = {}
        info["Min Hamming Dist"] =  [min_vertex_cover_brute_g1,min_vertex_cover_brute_g2,min_vertex_cover_brute
        df = pd.DataFrame.from_dict(info)
        df.index = ["G1 Algo","G2 Algo","MM Algo","RA Algo","RLP Algo"]
        print("Comparision of Hamming Distances with Brute Force")
        display(df)

    hamming_ratios = {"g1":min_vertex_cover_brute_g1,"g2":min_vertex_cover_brute_g2,"mm":min_vertex_cover_brute
    return hamming_ratios
```

Implementation:

- Generating random undirected graphs using networkx's *gnp_random_graph* function, given number of vertices(n) and probability of an edge between 2 vertices(p).
- For the same number of vertices(n) and probability(p), we run the compare function with 10 different seeds to ensure random graphs.

Parameterization:

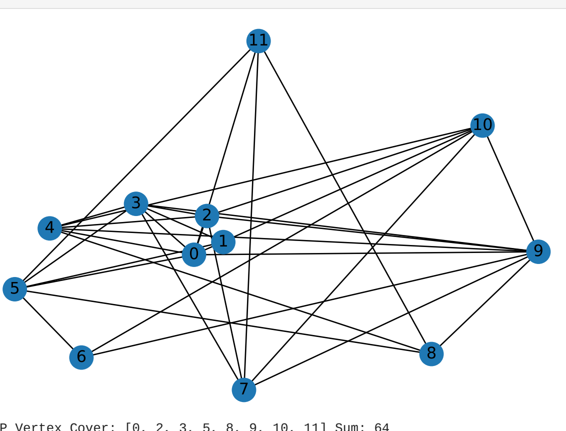
- *n_start* and *n_end* specify the range of number of vertices.
- *p_start, p_end* and *p_increment* determine the prob parameter for set generation, for each n, all algorithms are run 10*(p_start - p_end)/p_increment times.
- *lp* is a parameter which can be specified to be "ilp" or "none".
- *COMPARE* is used to specify whether we are comparing size or time.

--------------------------------Comparision of Algorithms--------------------------------

Algorithms	Vertex Covers	Length	Ratio wrt Brute Force
Brute Force	[2, 3, 4, 5, 9, 10, 11]	7	1.000000
G1 Algo	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]	10	1.428571
G2 Algo	[9, 0, 3, 2, 4, 5, 10, 7, 8]	9	1.285714
MM Algo	[11, 7, 9, 3, 10, 6, 1, 5, 0, 2, 8, 4]	12	1.714286
RA Algo	[0, 3, 1, 4, 2, 6, 5, 7, 9, 8]	10	1.376190
RLP	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]	12	1.714286

Weighted Vertex Cover

```
MIN_WEIGHT = 1
MAX_WEIGHT = 25
example_weights = [random.randint(MIN_WEIGHT,MAX_WEIGHT) for i in range(example_n)]
plot_graph(example_graph_object)
weighted_vertex_cover_rlp = rlp(example_graph_object,example_weights)
print("RLP Vertex Cover:",weighted_vertex_cover_rlp,"Sum:",sum([example_weights[x] for x in weighted_vertex_co
weighted_vertex_cover_ilp = ilp(example_graph_object,example_weights)
print("ILP Vertex Cover:",weighted_vertex_cover_ilp,"Sum:",sum([example_weights[x] for x in weighted_vertex_co
weighted_vertex_cover_ra = ra_algo(example_graph_object,example_weights,WEIGHTED = True)
print("RA Vertex Cover:",weighted_vertex_cover_ra,"Sum:",sum([example_weights[x] for x in weighted_vertex_cover
```

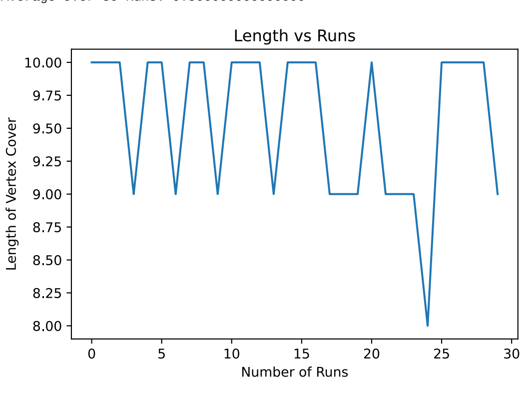


```
RLP Vertex Cover: [0, 2, 3, 5, 8, 9, 10, 11] Sum: 64
ILP Vertex Cover: [0, 2, 3, 5, 8, 9, 10, 11] Sum: 64
RA Vertex Cover: [2, 3, 4, 0, 1, 6, 5, 7, 9, 8] Sum: 117
```

Multiple RA Algorithm

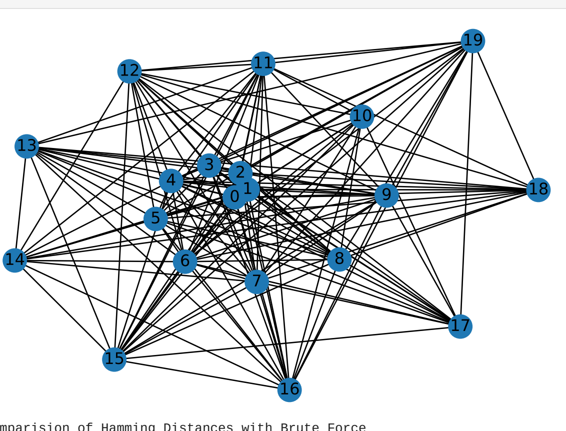
```
print("One Run of RA Algorithm:",ra_algo(example_graph_object,[]),"\n")
_,_ = multiple_ra_algo(example_graph_object,[], RUNS = 30, PRINT = True)
```

```
One Run of RA Algorithm: [2, 0, 1, 4, 5, 3, 6, 9, 7, 8]

Standard Deviation: 0.5587684871413404
Average over 30 Runs: 9.566666666666666
```



Hamming Distance

```
example_2_n = 20
example_2_p = 0.8
example_2_seed = 1203
example_2_directed = False
example_2_graph_object = nx.gnp_random_graph(example_2_n, example_2_p, example_2_seed, example_2_directed)
example_2_graph = nx.to_dict_of_lists(example_2_graph_object)
plot_graph(example_2_graph_object)
_ = compare_hamming_distances(example_2_graph_object, PRINT = True)
```



Comparision of Hamming Distances with Brute Force

	Min Hamming Dist
G1 Algo	2
G2 Algo	1
MM Algo	3
RA Algo	2
RLP Algo	3

Comparision of Algorithms

Execution Time

Implementation:

- Since brute force increases highly exponentially, it compresses the other graphs.
- We have displayed 2 other graphs for execution time without brute force.

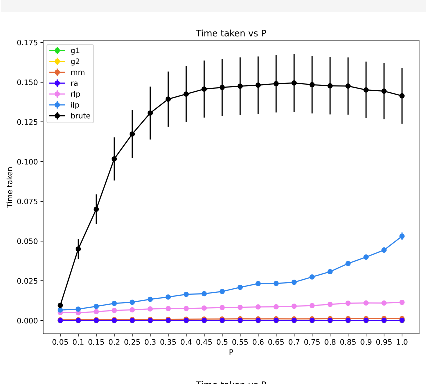
```
compare_algos(5,20,0.05,1,0.05,lp = "none",COMPARE = "time")
```

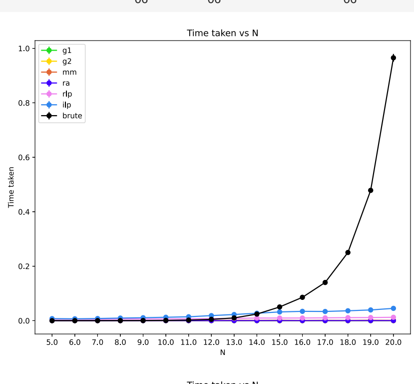
-------------------- Time taken vs N --------------------

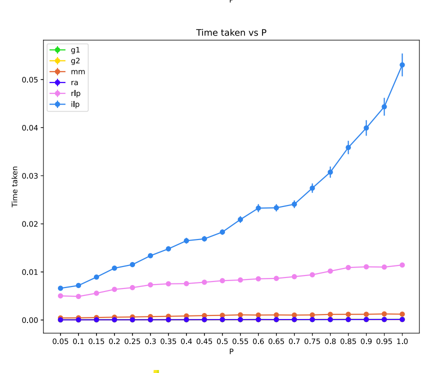
	Average													
	N	g1	g2	mm	ra	rlp	ilp	brute	N	g1	g2	mm	ra	rlp
0	5	0.000028	0.000034	0.000530	0.000040	0.005846	0.007517	0.000084	5	5.447538e-07	3.780376e-07	0.000011	3.940811e-07	0.000062
1	6	0.000027	0.000032	0.000420	0.000036	0.005733	0.006718	0.000095	6	3.270127e-07	3.032257e-07	0.000004	3.728743e-07	0.000054
2	7	0.000030	0.000036	0.000550	0.000038	0.005527	0.007824	0.000149	7	5.614046e-07	5.422901e-07	0.000012	5.752087e-07	0.000063
3	8	0.000039	0.000045	0.000512	0.000047	0.006114	0.009428	0.000288	8	6.780285e-07	7.468295e-07	0.000007	7.738116e-07	0.000065
4	9	0.000040	0.000045	0.000643	0.000046	0.006049	0.010759	0.000474	9	8.091951e-07	8.280930e-07	0.000013	8.111966e-07	0.000058
5	10	0.000051	0.000057	0.000601	0.000057	0.006365	0.012451	0.001002	10	1.090281e-06	1.189916e-06	0.000009	1.038614e-06	0.000067
6	11	0.000060	0.000066	0.000925	0.000067	0.006776	0.014280	0.002050	11	1.379949e-06	1.438037e-06	0.000022	1.393919e-06	0.000074
7	12	0.000098	0.000083	0.000788	0.000083	0.007654	0.018548	0.004789	12	4.062535e-06	1.585900e-06	0.000010	1.410011e-06	0.000078
8	13	0.000097	0.000091	0.001460	0.000093	0.008394	0.022589	0.010080	13	2.070389e-06	1.887082e-06	0.000038	1.876488e-06	0.000137
9	14	0.000114	0.000167	0.001035	0.000103	0.009524	0.027293	0.024189	14	2.206039e-06	9.221103e-06	0.000016	1.809191e-06	0.000153
10	15	0.000139	0.000102	0.001509	0.000105	0.009595	0.032043	0.050322	15	5.243690e-06	1.700128e-06	0.000029	1.679962e-06	0.000130
11	16	0.000101	0.000080	0.000807	0.000092	0.009746	0.034241	0.085679	16	1.220353e-06	1.121501e-06	0.000012	1.514053e-06	0.000154
12	17	0.000114	0.000087	0.001283	0.000093	0.010265	0.033771	0.140255	17	1.684882e-06	1.270538e-06	0.000021	1.463583e-06	0.000142
13	18	0.000116	0.000092	0.000885	0.000106	0.011082	0.036286	0.250355	18	1.348846e-06	1.440609e-06	0.000008	1.847663e-06	0.000171
14	19	0.000128	0.000102	0.001544	0.000114	0.011434	0.039326	0.478497	19	1.580677e-06	1.693444e-06	0.000027	2.032058e-06	0.000155
15	20	0.000134	0.000107	0.000969	0.000122	0.012371	0.045172	0.965385	20	1.678347e-06	1.869457e-06	0.000010	2.231093e-06	0.000188

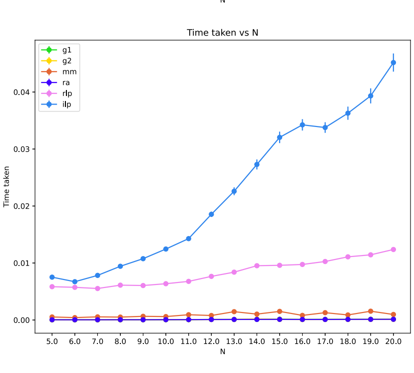
-------------------- Time taken vs P --------------------

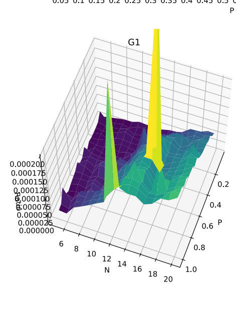
	Average													
	P	g1	g2	mm	ra	rlp	ilp	brute	P	g1	g2	mm	ra	rl
0	0.05	0.000035	0.000034	0.000428	0.000038	0.005019	0.006606	0.009571	0.05	9.833583e-07	3.277479e-07	0.000008	3.809837e-07	0.00006
1	0.10	0.000042	0.000037	0.000455	0.000037	0.004903	0.007179	0.045028	0.10	1.333760e-06	7.436847e-07	0.000006	5.200943e-07	0.00003
2	0.15	0.000047	0.000040	0.000516	0.000047	0.005572	0.008926	0.070037	0.15	1.546918e-06	6.914573e-07	0.000008	8.387448e-07	0.00005
3	0.20	0.000053	0.000046	0.000582	0.000048	0.006370	0.010784	0.101700	0.20	1.738638e-06	9.527779e-07	0.000011	8.303609e-07	0.00011
4	0.25	0.000057	0.000047	0.000633	0.000052	0.006736	0.011521	0.117351	0.25	2.023201e-06	1.065638e-06	0.000014	1.162432e-06	0.00010
5	0.30	0.000067	0.000054	0.000709	0.000058	0.007316	0.013373	0.130557	0.30	2.351652e-06	1.308272e-06	0.000018	1.305554e-06	0.00012
6	0.35	0.000068	0.000058	0.000764	0.000063	0.007528	0.014803	0.139319	0.35	2.383401e-06	1.492410e-06	0.000021	1.555656e-06	0.00015
7	0.40	0.000071	0.000062	0.000845	0.000067	0.007566	0.016488	0.142510	0.40	2.531392e-06	1.718083e-06	0.000027	1.914941e-06	0.00013
8	0.45	0.000079	0.000070	0.000923	0.000076	0.007853	0.016877	0.145653	0.45	2.805247e-06	1.954308e-06	0.000028	2.028192e-06	0.00014
9	0.50	0.000083	0.000075	0.000971	0.000086	0.008178	0.018291	0.146697	0.50	2.566828e-06	1.860113e-06	0.000027	2.426683e-06	0.00013
10	0.55	0.000088	0.000081	0.001074	0.000086	0.008324	0.020896	0.147482	0.55	2.647617e-06	1.878493e-06	0.000030	1.910732e-06	0.00011
11	0.60	0.000116	0.000082	0.001030	0.000088	0.008569	0.023264	0.148120	0.60	7.662603e-06	1.941054e-06	0.000025	2.088004e-06	0.00014
12	0.65	0.000090	0.000085	0.001057	0.000091	0.008651	0.023336	0.149066	0.65	2.854160e-06	2.070376e-06	0.000032	2.357252e-06	0.00014
13	0.70	0.000092	0.000086	0.001025	0.000094	0.009016	0.024075	0.149504	0.70	2.898569e-06	2.214865e-06	0.000028	2.289299e-06	0.00015
14	0.75	0.000095	0.000119	0.001054	0.000094	0.009408	0.027427	0.148400	0.75	2.943795e-06	7.259945e-06	0.000031	2.502098e-06	0.00018
15	0.80	0.000100	0.000123	0.001167	0.000098	0.010172	0.030748	0.147694	0.80	3.314700e-06	8.087332e-06	0.000044	2.876890e-06	0.00027
16	0.85	0.000110	0.000131	0.001178	0.000104	0.010908	0.035876	0.147558	0.85	3.475661e-06	8.677703e-06	0.000033	2.565973e-06	0.00024
17	0.90	0.000110	0.000103	0.001188	0.000107	0.011062	0.039935	0.145115	0.90	2.856441e-06	2.422142e-06	0.000035	2.713192e-06	0.00024
18	0.95	0.000110	0.000103	0.001264	0.000110	0.011008	0.044349	0.144363	0.95	3.147888e-06	2.486006e-06	0.000038	2.786392e-06	0.00020
19	1.00	0.000129	0.000097	0.001213	0.000112	0.011436	0.053051	0.141392	1.00	5.717060e-06	2.205903e-06	0.000038	3.001250e-06	0.00024

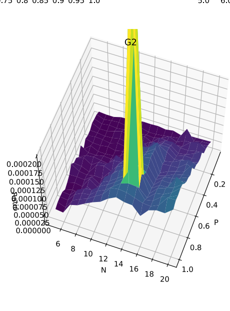


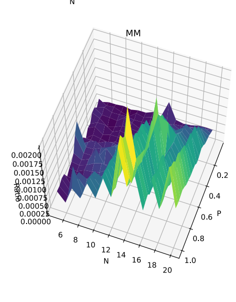


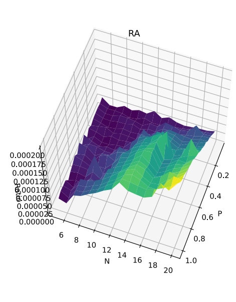


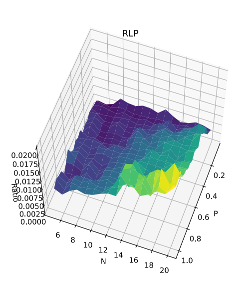


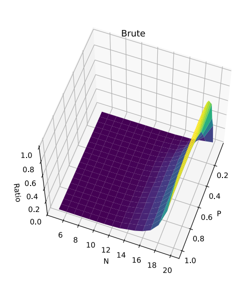


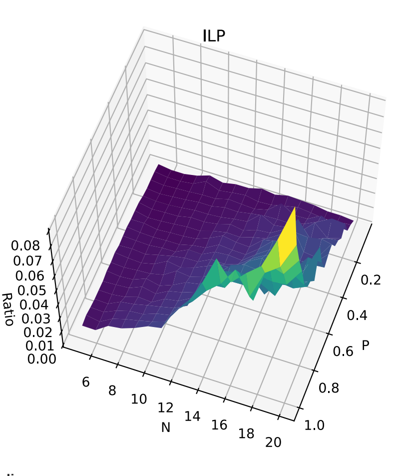












Findings:

- As expected, execution time for the brute force and ILP algorithm increases exponentially with n.
- For small p, Brute Force performs faster, because for sparse graphs, early stopping occurs faster as all the edges are covered by a smaller sized subset of vertices.
- Execution time for ILP algorithm grows at a smaller rate wrt both n and p than Brute Force.
- For small n, Execution time for ILP is higher than brute force but is overtaken by brute force due to the difference in growth rate.
- Other algorithms are much more efficient.
- RLP increases faster than other approximation algorithms but is better than ILP and Brute Force.

Performance Ratios

Unweighted Vertex Cover

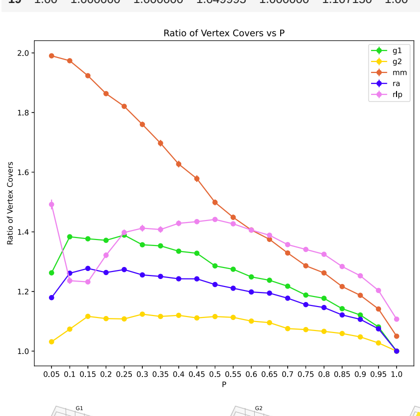
```
compare_algos(5,20,0.05,1,0.05,lp = "ilp", COMPARE = "size")
```

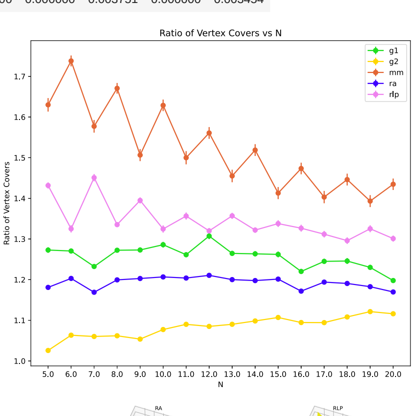
-------------------- Ratio of Vertex Covers vs N --------------------

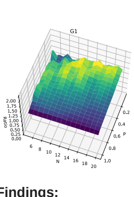
	Average						Standard Deviation				
	N	g1	g2	mm	ra	rlp	N	g1	g2	mm	ra	rlp
0	5	1.272778	1.025833	1.630000	1.180935	1.431667	5	0.006879	0.001677	0.016599	0.004431	0.007638
1	6	1.270476	1.063333	1.738000	1.203016	1.325024	6	0.006306	0.002431	0.013414	0.004073	0.009099
2	7	1.232306	1.060167	1.577167	1.168952	1.450861	7	0.004888	0.002275	0.015590	0.003658	0.009019
3	8	1.272324	1.061917	1.670643	1.199444	1.335328	8	0.005391	0.002374	0.013152	0.003485	0.005889
4	9	1.272972	1.053762	1.506389	1.202821	1.394875	9	0.006293	0.002031	0.014846	0.003997	0.007541
5	10	1.285937	1.077256	1.628714	1.206707	1.324594	10	0.007017	0.002612	0.014205	0.004541	0.008954
6	11	1.261323	1.090101	1.499885	1.203949	1.356442	11	0.005926	0.002510	0.016257	0.004041	0.008757
7	12	1.307246	1.084998	1.560807	1.210567	1.319962	12	0.007506	0.002371	0.014457	0.004165	0.007081
8	13	1.264572	1.090167	1.455006	1.200108	1.356973	13	0.006131	0.002074	0.015376	0.004102	0.007421
9	14	1.263483	1.098407	1.518722	1.197686	1.321924	14	0.006600	0.002319	0.014509	0.004215	0.006772
10	15	1.262064	1.106927	1.412931	1.201354	1.337873	15	0.007183	0.002370	0.014882	0.004392	0.008337
11	16	1.220113	1.094513	1.473433	1.171736	1.326536	16	0.006277	0.002143	0.014228	0.003865	0.008485
12	17	1.244930	1.094311	1.403107	1.193949	1.311836	17	0.006502	0.002184	0.015160	0.004458	0.008036
13	18	1.245828	1.108225	1.446143	1.190649	1.295946	18	0.007317	0.002725	0.014564	0.004664	0.008159
14	19	1.230115	1.121381	1.393220	1.182676	1.325211	19	0.006285	0.003098	0.014684	0.004105	0.008593
15	20	1.197816	1.116079	1.434621	1.169725	1.301003	20	0.004890	0.002439	0.014101	0.004108	0.007464

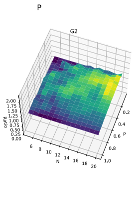
-------------------- Ratio of Vertex Covers vs P --------------------

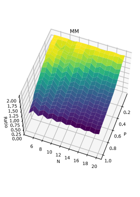
	Average						Standard Deviation				
	P	g1	g2	mm	ra	rlp	P	g1	g2	mm	ra	rlp
0	0.05	1.262593	1.031458	1.990278	1.179392	1.491790	0.05	0.008051	0.002684	0.001095	0.005525	0.016409
1	0.10	1.383418	1.073700	1.973755	1.261213	1.236173	0.10	0.007416	0.003324	0.002138	0.003672	0.010746
2	0.15	1.376670	1.116627	1.923475	1.277319	1.232179	0.15	0.007183	0.004352	0.003988	0.004103	0.007226
3	0.20	1.371604	1.109157	1.863467	1.263634	1.321621	0.20	0.006586	0.003780	0.005595	0.004040	0.009618
4	0.25	1.389052	1.108044	1.820809	1.273439	1.397720	0.25	0.003907	0.003937	0.006868	0.002287	0.009968
5	0.30	1.356610	1.123883	1.760204	1.255518	1.412235	0.30	0.003487	0.003380	0.008981	0.001822	0.011265
6	0.35	1.353042	1.116337	1.697222	1.250483	1.407792	0.35	0.003932	0.002670	0.010943	0.002046	0.010723
7	0.40	1.335045	1.119985	1.627235	1.242548	1.428635	0.40	0.004656	0.002819	0.011064	0.002748	0.009592
8	0.45	1.328192	1.111324	1.578342	1.242387	1.434181	0.45	0.004199	0.002362	0.011093	0.002483	0.008375
9	0.50	1.285784	1.115986	1.498825	1.223375	1.441470	0.50	0.003629	0.001906	0.009631	0.001869	0.006708
10	0.55	1.274544	1.113186	1.448656	1.210860	1.426878	0.55	0.003837	0.001909	0.008856	0.002121	0.006506
11	0.60	1.248892	1.100751	1.406247	1.198353	1.405875	0.60	0.003471	0.001890	0.008734	0.001983	0.006227
12	0.65	1.237603	1.095494	1.374866	1.194453	1.388569	0.65	0.003787	0.002367	0.008397	0.002146	0.006227
13	0.70	1.217654	1.075555	1.329120	1.177452	1.357178	0.70	0.004526	0.001837	0.007632	0.002830	0.005740
14	0.75	1.187966	1.072222	1.286157	1.155989	1.341022	0.75	0.003920	0.001867	0.006894	0.002493	0.006967
15	0.80	1.177140	1.066255	1.262303	1.146222	1.324800	0.80	0.003843	0.001242	0.006289	0.002241	0.007273
16	0.85	1.142329	1.059138	1.216560	1.121321	1.283750	0.85	0.001934	0.001151	0.005049	0.001342	0.005716
17	0.90	1.121116	1.047660	1.187161	1.106608	1.252630	0.90	0.002321	0.000808	0.005314	0.001890	0.005875
18	0.95	1.081099	1.027458	1.141309	1.074775	1.203421	0.95	0.000806	0.001077	0.004332	0.000790	0.003998
19	1.00	1.000000	1.000000	1.049995	1.000000	1.107150	1.00	0.000000	0.000000	0.003751	0.000000	0.003454

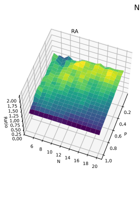


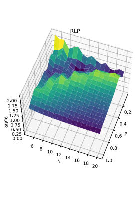












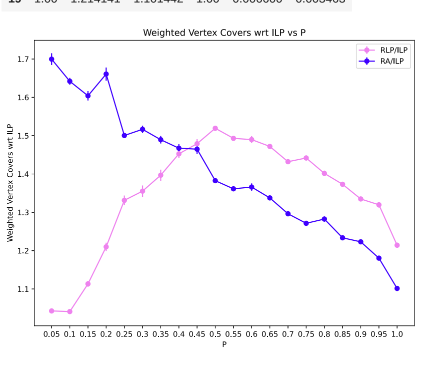
Findings:

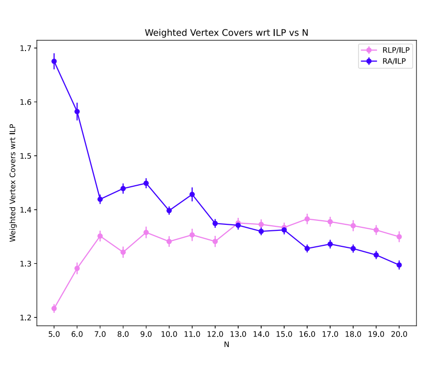
- Ratio for MM Algorithm is high at lower values of p, since the graph is sparse and most vertices belong to only 1 edge, hence MM picks two vertices, even though one is sufficient.
- MM Algorithm v/s n oscillates for odd and even number of vertices:
  - It spikes on even vertices as for odd vertices, we have a vertex which is never picked but for even vertices, all pairs are selected, hence for odd number of vertices, vertex cover is shorter.
  - This spiking behaviour decreases in amplitude as n increases, since the difference of including 1 vertex becomes negligible.
- G1, G2 and RA decrease slightly with an increase in p, as for denser graphs, there are more vertices with a higher degree, so we can cover multiple edges by including one vertex.
- For other algorithms, the ratio with n remains pretty much constant.
- G2 algorithm performs the best as it's ratio is closest to 1. This is because of sorted iteration on degree of vertices.
- RA algorithm lies between G1 and G2 algorithms even though it has a similar algorithmic structure since it incorporates a randomized approach which averages out the length.

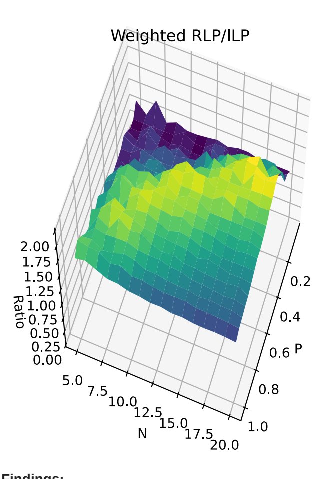
Weighted Vertex Cover

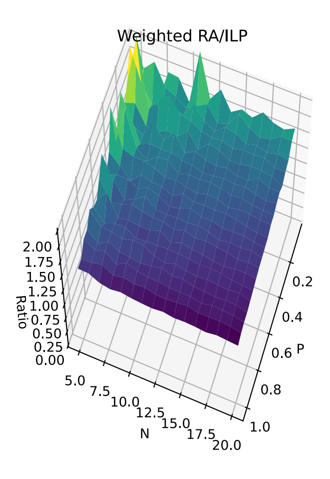
```
compare_weighted(5,20,0.05,1,0.05,1,25)
```

		Average			Standard Deviation	
	P	RLP/ILP	RA/ILP	P	RLP/ILP	RA/ILP
0	0.05	1.042717	1.699637	0.05	0.004800	0.015351
1	0.10	1.041068	1.641886	0.10	0.003228	0.008800
2	0.15	1.113337	1.603987	0.15	0.007499	0.012266
3	0.20	1.209976	1.660737	0.20	0.010810	0.017103
4	0.25	1.331220	1.500408	0.25	0.012601	0.006487
5	0.30	1.355280	1.516577	0.30	0.014774	0.009593
6	0.35	1.396870	1.489450	0.35	0.014455	0.009915
7	0.40	1.452593	1.467310	0.40	0.011214	0.010422
8	0.45	1.479018	1.464770	0.45	0.011895	0.012535
9	0.50	1.519331	1.382568	0.50	0.006695	0.005409
10	0.55	1.493003	1.361333	0.55	0.005658	0.004400
11	0.60	1.489747	1.365933	0.60	0.009151	0.009521
12	0.65	1.471901	1.337599	0.65	0.006158	0.006026
13	0.70	1.432035	1.296222	0.70	0.006192	0.006655
14	0.75	1.441762	1.271347	0.75	0.006589	0.004439
15	0.80	1.401629	1.282452	0.80	0.007234	0.006761
16	0.85	1.373179	1.233525	0.85	0.005828	0.004538
17	0.90	1.334844	1.223037	0.90	0.005387	0.004961
18	0.95	1.319616	1.180460	0.95	0.007261	0.002973
19	1.00	1.214141	1.101442	1.00	0.006600	0.003463









Findings:

- RA decreases with increase in p, as for denser graphs, there are more vertices with a higher degree, so we can cover multiple edges by including one vertex.
- RA is high for lower values of n since picking a wrong vertex has a higher consequence, and since RA is randomized, there is always a probability that a wrong vertex will be picked in the beginning, making the ratio higher.
- For higher n, since there are more options available, this balances out.

Hamming Distance

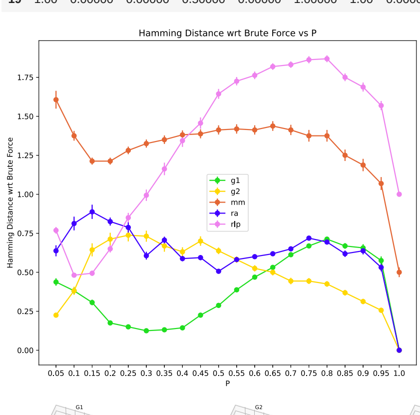
```
In [ ]: compare_algos(5,20,0.05,1,0.05,COMPARE = "hamming")
```

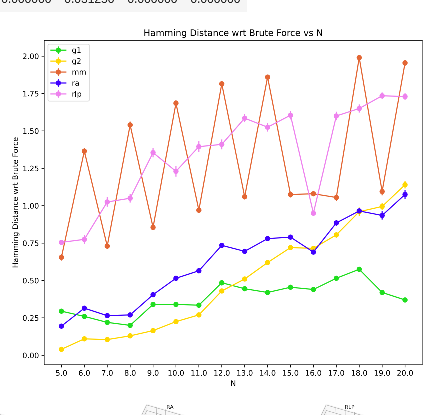
```
-------------------- Hamming Distance wrt Brute Force vs N --------------------
```

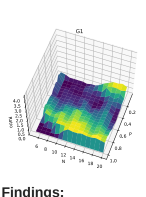
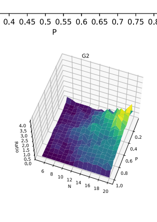
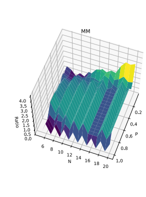
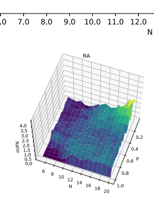
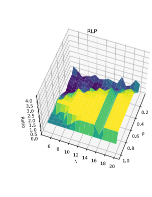
		Average					Standard Deviation					
	N	g1	g2	mm	ra	rlp	N	g1	g2	mm	ra	rlp
0	5	0.295	0.040	0.655	0.195	0.755	5	0.010894	0.004583	0.021993	0.008288	0.014096
1	6	0.260	0.110	1.365	0.315	0.775	6	0.016155	0.004975	0.022487	0.012774	0.029106
2	7	0.220	0.105	0.730	0.265	1.025	7	0.011358	0.004867	0.018378	0.012173	0.031459
3	8	0.200	0.130	1.540	0.270	1.050	8	0.011068	0.004213	0.022439	0.008958	0.028918
4	9	0.340	0.165	0.855	0.405	1.355	9	0.015922	0.007949	0.015849	0.012891	0.031602
5	10	0.340	0.225	1.685	0.515	1.230	10	0.016462	0.008496	0.020017	0.013989	0.035711
6	11	0.335	0.270	0.970	0.565	1.395	11	0.015023	0.009887	0.012659	0.012070	0.035794
7	12	0.485	0.430	1.815	0.735	1.410	12	0.017977	0.011948	0.014771	0.011099	0.033162
8	13	0.445	0.510	1.060	0.695	1.585	13	0.015849	0.013029	0.018262	0.011562	0.027077
9	14	0.420	0.620	1.860	0.780	1.525	14	0.017364	0.014457	0.011874	0.014195	0.029363
10	15	0.455	0.720	1.075	0.790	1.605	15	0.016315	0.016016	0.019992	0.015322	0.028305
11	16	0.440	0.715	1.080	0.690	0.950	16	0.015764	0.016376	0.009695	0.012639	0.010062
12	17	0.515	0.805	1.055	0.885	1.600	17	0.011322	0.018471	0.021993	0.018593	0.028810
13	18	0.575	0.960	1.990	0.965	1.650	18	0.013953	0.028213	0.016726	0.018926	0.027318
14	19	0.420	0.995	1.095	0.935	1.735	19	0.016553	0.024823	0.025665	0.027716	0.022151
15	20	0.370	1.140	1.955	1.075	1.730	20	0.015419	0.024413	0.019331	0.031420	0.021512

```
-------------------- Hamming Distance wrt Brute Force vs P --------------------
```

		Average					Standard Deviation					
	P	g1	g2	mm	ra	rlp	P	g1	g2	mm	ra	rlp
0	0.05	0.43750	0.22500	1.60625	0.63750	0.76875	0.05	0.023991	0.014741	0.057089	0.035416	0.021392
1	0.10	0.38125	0.38125	1.37500	0.81250	0.48125	0.10	0.021562	0.026166	0.031367	0.044077	0.013298
2	0.15	0.30625	0.64375	1.21250	0.88750	0.49375	0.15	0.017742	0.041924	0.022521	0.045762	0.016897
3	0.20	0.17500	0.71250	1.21250	0.82500	0.65000	0.20	0.011158	0.040797	0.022194	0.025911	0.023385
4	0.25	0.15000	0.73750	1.28125	0.78750	0.85000	0.25	0.009632	0.039644	0.024329	0.034508	0.030619
5	0.30	0.12500	0.73125	1.32500	0.60625	0.99375	0.30	0.006811	0.035474	0.026099	0.025046	0.036475
6	0.35	0.13125	0.66875	1.35000	0.70625	1.16250	0.35	0.007564	0.037155	0.027333	0.021927	0.040195
7	0.40	0.14375	0.63125	1.38125	0.58750	1.34375	0.40	0.007318	0.029041	0.028403	0.016810	0.040018
8	0.45	0.22500	0.70000	1.38750	0.59375	1.45625	0.45	0.008119	0.030298	0.028885	0.016308	0.036374
9	0.50	0.28750	0.63750	1.41250	0.50625	1.64375	0.50	0.008522	0.021295	0.029554	0.009496	0.031790
10	0.55	0.38750	0.58125	1.41875	0.58125	1.72500	0.55	0.008231	0.020755	0.029992	0.013838	0.026837
11	0.60	0.46875	0.52500	1.41250	0.60000	1.76250	0.60	0.010532	0.019949	0.029800	0.012885	0.024395
12	0.65	0.53125	0.50000	1.43750	0.61875	1.81875	0.65	0.010762	0.020010	0.031005	0.013298	0.020989
13	0.70	0.61250	0.44375	1.41250	0.65000	1.83125	0.70	0.008804	0.020843	0.032915	0.011049	0.020696
14	0.75	0.66875	0.44375	1.37500	0.71875	1.86250	0.75	0.013020	0.018876	0.037467	0.014358	0.022844
15	0.80	0.71250	0.42500	1.37500	0.69375	1.86875	0.80	0.013235	0.017747	0.037467	0.015224	0.021732
16	0.85	0.66875	0.36875	1.25000	0.61875	1.75000	0.85	0.018049	0.018318	0.036644	0.020637	0.024705
17	0.90	0.65625	0.31250	1.18750	0.63750	1.68750	0.90	0.023898	0.017098	0.039644	0.023372	0.028970
18	0.95	0.57500	0.25625	1.06875	0.53125	1.56875	0.95	0.026746	0.015304	0.041500	0.025457	0.029458
19	1.00	0.00000	0.00000	0.50000	0.00000	1.00000	1.00	0.000000	0.000000	0.031250	0.000000	0.000000





Findings:

- MM Algorithm v/s n oscillates for odd and even number of vertices - it spikes on even vertices as for odd vertices, we have a vertex which is never picked but for even vertices, all pairs are selected, hence for odd number of vertices, vertex cover is shorter.
- Other algorithms increase with an increase in n, as expected.
- Hamming distance for MM Algorithm is high at lower values of p, since the graph is sparse and most vertices belong to only 1 edge, hence MM picks two vertices, even though one is sufficient.
- At lower and higher values of p, hamming distances of other algorithms are less, since the ratios of 1s and 0s in the characteristic vector (of the vertex cover) is very skewed, so the XOR of these vectors cannot be that high.

Minimum Set Cover

Introduction

Set Cover (SC) Problem:

- **[Instance]** Set Universe U = {$u_1, u_2, ... , u_n$} and a family S = {$s_1, s_2, ... , s_n$} of subsets of U.
- **[Feasible Solutions]** A subfamily C ⊆ S such that union of C is U.
- **[Value]** The value of a solution is the size of the cover |C|, and the goal is to minimize it.

Weighted Set Cover Problem:

- **[Instance]** Set Universe U = {$u_1, u_2, ... , u_n$} and a family S = {$s_1, s_2, ... , s_n$} of subsets of U with a weights array W.
- **[Feasible Solutions]** A subfamily C ⊆ S such that union of C is U.
- **[Value]** The value of a solution is the weighted sum of the cover: $\sum w_i$ for all i such that $S_i \in$ C, and the goal is to minimize it.

Implementation:

We use two metrics for the comparision of algorithms:

1. Performance Ratio: ratios of lengths of vertex covers (wrt Brute force or ILP).
2. Execution time of various algorithms.

Import

In [ ]:
```
%%capture
import networkx as nx
import matplotlib.pyplot as plt
import pandas as pd
from tabulate import tabulate
import random
import numpy as np
import scipy
from scipy.optimize import linprog
from mpl_toolkits.mplot3d import Axes3D
from math import inf
from math import ceil
import warnings
warnings.filterwarnings("ignore")
import time
import itertools
import copy
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('pdf','svg')
```

In [ ]:
```
%%capture
!pip install mip
from mip import Model, xsum, minimize, BINARY
```

Problem Set Generation

The generate_sets function generates two sets, that is a universal set (U) of size n and a family of subsets (S) of size m.

Weights are generated randomly from MIN_WEIGHT to MAX_WEIGHT.

Implementation:

- Universal set is given by list [0...n], while we generate the family of subsets by randomly deciding if we include an element in a subset i or not.
- This does not guarantee that a set cover is possible, so we check if the union gives universal set, if not then we add missed elements to randomly chosen subsets.
- Our earlier implementation allowed some of the generated subsets in the family of subsets to be same - this reduces the value of m and hence can affect the graphs.
- The family of subsets is now a set, and hence ensuring uniqueness.

In [ ]:
```
def generate_sets(n,m,PROB = 0.5):
    universal_set = [x for x in range(n)]
    family_of_subsets = set()

    while (len(family_of_subsets) < m): # Ensure size is m
        # Temporary list
        temp = []
        for vertex in universal_set:    # 0...n
            # Higher the prob, more number of elements in a subset
            if random.random() >= PROB:
                temp.append(vertex)

        # Typecasted to tuple, since set cannot have lists
        family_of_subsets.add(tuple(temp))

    # Check if union is universal
    union = []
    for subset in family_of_subsets:
        for vertex in subset:
            if vertex not in union:
                union.append(vertex)

    # Typecasted to set, to check if both lists are the same
    if set(union) == set(universal_set):
        return universal_set,list(family_of_subsets)

    # Add missed vertices to a randomly chosen subset
    diff = list(set(universal_set) - set(union))
    for vertex in diff:
        # Pick any random subset from the family
        random_i = random.randint(0,m-1)
        i = 0
        for subset in family_of_subsets:
            if random_i == i:
                # Make a copy of this subset
                subset_copy = list(subset)
                # Delete this subset
                family_of_subsets.remove(subset)
                # Add the same subset appended with the missed vertex
                subset_copy.append(vertex)
                # Typecasted to tuple, since set cannot have lists
                family_of_subsets.add(tuple(subset_copy))
            i+=1
    return universal_set,list(family_of_subsets)

def generate_weights(family_of_subsets, MIN_WEIGHT = 1, MAX_WEIGHT = 25):
    weights = {}
    for set_i in family_of_subsets:
        # As a list is not hashable, it is coverted to a tuple
        weights[tuple(set_i)] = random.randint(MIN_WEIGHT,MAX_WEIGHT)

    return weights
```

Exact Algorithms

Brute Force

Algorithm Brute Force

Input: Universal Set (U), Family of subsets (S) and a weights array.
Output: A set cover.

1. Iterate through all subsets of Family in order of size.
2. Check if our subset is a set cover.
3. Calculate the cost function and store it's minimum

Implementation

- If no weights are passed, we create an array with all weights = 1 to have a common implementation for weighted and unweighted case.
- Initial implementation was generation of all permutations of all lengths, which took more time.
- This was later pruned by generating permutations in increasing order of length, and breaking when we get a set cover (this will be smallest for the unweighted case).
- For weighted, we continue to generate all permutations (no early stopping).

In [ ]:
```
[illegible]
```

Integer Linear Programming

Input: Universal Set (U), Family of subsets (S) and a weights array.
Output: A set cover.

$$\min \sum_{i=1}^{m} w_i x_i, \quad (1)$$

subject to $\quad (2)$

$$\sum_{i: v \in S_i} x_i \geq 1, \quad \forall v \in U \quad (3)$$

$$x_i \in \{0,1\} \quad i = 1,2,\dots,n \quad (4)$$

$x_i = \begin{cases} 1 & \text{if the set } S_i \text{ is included in the set cover,} \\ 0 & \text{otherwise.} \end{cases}$

w_i is the weight of the i^{th} set

Implementation

- If no weights are passed, we create an array with all weights = 1 to have a common implementation for weighted and unweighted case.

In [ ]:
```
def ilp(universal_set, family_of_subsets, weights = {}):

    mod = Model("ILP-Set Cover")
    n = len(family_of_subsets)

    # Xi's || set of sets => subsets
    x = [mod.add_var(var_type=BINARY) for i in range(n)]

    if len(weights) == 0:
        for i in range(n):
            # As a list is not hashable, it is coverted to a tuple
            weights[tuple(family_of_subsets[i])]=1

    # objective function: minimize the cost
    mod.objective = minimize(xsum(x[i]*weights[tuple(family_of_subsets[i])] for i in range(n)))

    # constraints: for every element in the universal set include at least one subset which contains the elemen
    for element in universal_set:
        mod += xsum(x[i] for i in range(len(family_of_subsets)) if element in family_of_subsets[i]) >= 1

    # optimizing
    mod.optimize()

    # If the value of the variable x is greater than 0.99, include the corresponding subset in the set cover.
    set_cover = [i for i in range(n) if x[i].x >= 0.99]

    # converting indices to set cover
    set_cover = [family_of_subsets[i] for i in set_cover]

    return set_cover
```

Approximation Algorithms

Greedy Algorithm

Algorithm G2:

Input: Universal Set (U), Family of subsets (S) and a weights array.
Output: A set cover.

1. Initialize C to an empty set
2. While U is not empty
   A. Pick a subset with minimum weight/size ratio which belongs to S.
   B. Add this subset to C.
   C. Remove elements of this subset from the universal set and other subsets in the family of subsets.
3. Return C

Performance Ratio: $O(\log(|\text{Family of subsets}|))$

- It is easy to see that this is a generalization to the weighted case of the heuristic G2 discussed in the minimum Vertex Cover problem.
- As such, it cannot be expected to have a performance ratio better than that of G2, i.e. $O(\log n)$.

Implementation:

- A copy (copy.deepcopy) is made of both the universal set and the family of subsets. All the changes (removal of selected elements) are done on these copies.
- If no weights are passed, we create an array with all weights = 1 to have a common implementation for weighted and unweighted case.

In [ ]:
```
def g2_algo(universal_set, family_of_subsets, weights = {}):
    set_cover = []
    # Create a copy and update the copy
    universal_set_copy = copy.deepcopy(universal_set)
    family_of_subsets_copy = copy.deepcopy(family_of_subsets)

    if len(weights) == 0:
        for i in range(len(family_of_subsets)):
            # As a list is not hashable, it is coverted to a tuple
            weights[tuple(family_of_subsets[i])] = 1

    while (len(universal_set_copy) > 0):
        min_obj = 1000000                     # Weights/Degree
        min_i = -1
        for i in range(len(family_of_subsets_copy)):
            if (len(family_of_subsets_copy[i]) == 0):
                continue
            curr_obj = weights[tuple(family_of_subsets[i])]/len(family_of_subsets_copy[i])
            if curr_obj < min_obj:
                min_obj = curr_obj
                min_i = i
        # Add to Set Cover
        set_cover.append(family_of_subsets[min_i])

        # Remove the element from Universal Set
        for element in family_of_subsets_copy[min_i]:
            universal_set_copy.remove(element)

        # Remove the element from other subsets in the family of subsets
        for j in range(len(family_of_subsets)):
            if j != min_i:
                family_of_subsets_copy[j] = [x for x in family_of_subsets_copy[j] if x not in family_of_subsets

        family_of_subsets_copy[min_i] = []

    return set_cover
```

Functions for Comparision

In [ ]:
```
# Colours for lines in the graph
colors = {"brute":'k',"ilp":'#2f85ed',"g2":'#e36634'}

# Print the datapoints in a table format
def print_table(wrt, lst, ratios, title):
    ratios[wrt] = lst
    df = pd.DataFrame.from_dict(ratios)
    df.columns = df.columns.str.strip()
    cols = df.columns.tolist()
    cols = cols[-1:] + cols[:-1]
    df = df[cols]
    del ratios[wrt]
    df.columns = pd.MultiIndex.from_product([[title], df.columns])
    return df

# Plot all the ratios in one single line graph
def plot_ratios(wrt,ratios,lst,ax,title = "Ratio of Costs",std_dev = {}):
    for key in ratios.keys():
        ax.errorbar([x for x in range(len(lst))], ratios[key], yerr=std_dev[key], fmt='-o', label=key, color=co
    plt.sca(ax)
    ax.set_xlabel(wrt)
    ax.set_ylabel(title)
    ax.set_title(title+" vs "+wrt)
    ax.legend()
    plt.xticks(list(range(len(lst))), [ceil(100*x)/100 for x in lst])

 # Plot ratios in 3d graphs
def plot_3d(title,ps,ns,Z,ax,z_max):
    X, Y = np.meshgrid(ps, ns)
    ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none')
    ax.set_title(title);
    ax.set_xlabel('M')
    ax.set_ylabel('N')
    ax.set_zlabel('Ratio')
    ax.set_zlim(0, z_max)
    ax.view_init(30, 20)
```

Parameterization:

- *PRINT* is used to specify whether to print or not.
- *lp* is a parameter which can be specified to be "ilp" or "none".
- *COMPARE* is used to specify whether we are comparing cost or time.

In [ ]:
```
# This function compares both times and costs of g2 algorithm with brute or ILP.
def compare(universal_set, family_of_subsets, weights = {}, PRINT = False, lp = "none", COMPARE = "cost"):
    WEIGHTED = True
    if len(weights) == 0:
        WEIGHTED = False
        for i in range(len(family_of_subsets)):
            weights[tuple(family_of_subsets[i])] = 1

    t_brute_start = time.time()
    if lp == "ilp":
        set_cover_brute = ilp(universal_set, family_of_subsets, weights)
    else:
        set_cover_brute = brute_force(universal_set, family_of_subsets, weights)
    t_brute_end = time.time()

    t_g2_start = time.time()
    set_cover_g2 = g2_algo(universal_set, family_of_subsets, weights)
    t_g2_end = time.time()

    t_brute = t_brute_end - t_brute_start
    t_g2 = t_g2_end - t_g2_start

    if COMPARE == "time":
        t_ilp_start = time.time()
        set_cover_ilp = ilp(universal_set, family_of_subsets, weights)
        t_ilp_end = time.time()
        t_ilp = t_ilp_end - t_ilp_start
        time_dict = {'brute':t_brute,'g2':t_g2,'ilp':t_ilp}
        return time_dict

    if len(set_cover_brute) == 0:
        return {'g2':0}

    if WEIGHTED == False:
        g2_ratio = len(set_cover_g2)/len(set_cover_brute)
    else:
        g2_ratio = sum([weights[tuple(x)] for x in set_cover_g2]) / sum([weights[tuple(x)] for x in set_cover_b

    if (PRINT):
        info = {}
        info["Set Covers"] = [set_cover_brute,set_cover_g2]
        if WEIGHTED == False:
            info["Length"] = [len(set_cover_brute),len(set_cover_g2)]
        else:
            info["Cost"] = [sum([weights[tuple(x)] for x in set_cover_g2]),sum([weights[tuple(x)] for x in set_

        info["Ratio wrt Brute Force"] = [1,g2_ratio]
        df = pd.DataFrame.from_dict(info)

        if lp == "ilp":
            df.index = ["Integer Lin Prog","G2 Algo"]
        else:
            df.index = ["Brute Force","G2 Algo"]
        df.index.name = "Algorithms"
        print("Comparision of Algorithms")
        display(df)

    ratio_dict = {'g2':g2_ratio}
    return ratio_dict
```

Implementation:

- For the same size of universal set(n) and probability(p), we run the compare() function for 10 different sizes of family of subsets(m) picked uniformly from range n/2 to n^2.

Parameterization:

- *n_start* and *n_end* specify the range of number of vertices.
- *p_start, p_end* and *p_increment* determine the prob parameter for set generation, for each n, all algorithms are run 10*(PROB_START - PROB_END)/PROB_INC times.
- *lp* is a parameter which can be specified to be "ilp" or "none".
- *COMPARE* is used to specify whether we are comparing cost or time.
- *WEIGHTED* is used to specify weather sets should be weighted or unweighted

In [ ]:
```
# This function calls the compare() function over a range of n and probabilities
def compare_algos(n_start,n_end,p_start,p_end,p_increment,lp = "none",COMPARE = "size",WEIGHTED = False):
    ns = []
    g2_ratios = []
    brute_ratios = []
    ilp_ratios = []

    for n in range(n_start,n_end+1):
        g2_ratios_p = []
        brute_ratios_p = []
        ilp_ratios_p = []
        prob = p_start
        while (prob <= p_end+0.001):
            g2_ratio_sum = 0
            brute_ratio_sum = 0
            ilp_ratio_sum = 0
            no = 0

            # Run this loop for itr_end times, for values between n/2 and n^2
            m_start = n//2
            itr_end = 10
            m_inc = max(((n**2) - (n//2))//itr_end,1)
            if COMPARE == "time":
                itr_end = 0
            for itr in range(0,itr_end+1):
                m = m_start + m_inc*itr

                # Generate sets
                universal_set, family_of_subsets = generate_sets(n,m,prob)

                # Compare ratios and Update the ratios list
                if WEIGHTED == True:
                    weight_arr = generate_weights(family_of_subsets)
                else:
                    weight_arr = {}
                ratios = compare(universal_set, family_of_subsets, weight_arr, PRINT = False, lp = lp, COMPARE
                g2_ratio = ratios['g2']
                if COMPARE == "time":
                    g2_ratio = ratios['g2']
                    brute_ratio = ratios['brute']
                    ilp_ratio = ratios['ilp']

                g2_ratio_sum += g2_ratio
                if COMPARE == "time":
                    brute_ratio_sum += brute_ratio
                    ilp_ratio_sum += ilp_ratio
                if (g2_ratio != 0):
                    no += 1

            prob += p_increment
            g2_ratios_p.append(g2_ratio_sum/no)
            brute_ratios_p.append(brute_ratio_sum/no)
            ilp_ratios_p.append(ilp_ratio_sum/no)

        ns.append(n)
        g2_ratios.append(g2_ratios_p)
        brute_ratios.append(brute_ratios_p)
        ilp_ratios.append(ilp_ratios_p)

    # Convert Lists to numpy arrays for plotting
    ms = np.linspace(p_start, p_end, num = ceil((p_end-p_start)/p_increment) + 1)
    g2_ratios = np.array(g2_ratios)

    g2_ratios_m = np.sum(g2_ratios,axis=0)/len(g2_ratios)
    g2_ratios_n = np.sum(g2_ratios,axis=1)/len(g2_ratios[0])
    g2_std_dev_m = np.std(g2_ratios,axis=0)
    g2_std_dev_n = np.std(g2_ratios,axis=1)

    if COMPARE == "time":
        brute_ratios = np.array(brute_ratios)
        brute_ratios_m = np.sum(brute_ratios,axis=0)/len(brute_ratios)
        brute_ratios_n = np.sum(brute_ratios,axis=1)/len(brute_ratios[0])
        ilp_ratios = np.array(ilp_ratios)
        ilp_ratios_m = np.sum(ilp_ratios,axis=0)/len(ilp_ratios)
        ilp_ratios_n = np.sum(ilp_ratios,axis=1)/len(ilp_ratios[0])

    # Plot and display ratios and standard deviation
    fig = plt.figure(figsize=(30,10))
    ax1 = fig.add_subplot(1, 2, 1)
    ax2 = fig.add_subplot(1, 2, 2)

    ratios_m = {"g2":g2_ratios_m}
    ratios_n = {"g2":g2_ratios_n}
    std_dev_m = {"g2":g2_std_dev_m}
    std_dev_n = {"g2":g2_std_dev_n}
    title = "Ratio of set Covers"
    if COMPARE == "time":
        ratios_m["brute"] = brute_ratios_m
        ratios_n["brute"] = brute_ratios_n
        std_dev_m["brute"] = np.std(brute_ratios,axis=0)
        std_dev_n["brute"] = np.std(brute_ratios,axis=1)
        ratios_m["ilp"] = ilp_ratios_m
        ratios_n["ilp"] = ilp_ratios_n
        std_dev_m["ilp"] = np.std(ilp_ratios,axis=0)
        std_dev_n["ilp"] = np.std(ilp_ratios,axis=1)
        title = "Time taken"

    # Display Tables
    print("\n---------------------",title,"vs","N","---------------------")
    df1 = print_table("N", ns, ratios_n, "Average")
    df2 = print_table("N", ns, std_dev_n, "Standard Deviation")
    dft = pd.concat([df1, df2], axis=1)
    display(dft)

    print("\n---------------------",title,"vs","M","---------------------")
    df1 = print_table("M", ms, ratios_m, "Average")
    df2 = print_table("M", ms, std_dev_m, "Standard Deviation")
    dft = pd.concat([df1, df2], axis=1)
    display(dft)

    # Show 2D plots
    if COMPARE == "time":
        g2_std_dev_m = np.zeros(g2_ratios_m.shape)
        g2_std_dev_n = np.zeros(g2_ratios_n.shape)
        std_dev_m["brute"] = np.zeros(brute_ratios_m.shape)
        std_dev_n["brute"] = np.zeros(brute_ratios_n.shape)
        std_dev_m["ilp"] = np.zeros(ilp_ratios_m.shape)
        std_dev_n["ilp"] = np.zeros(ilp_ratios_n.shape)
    plot_ratios("P",ratios_m,ms,ax1,title,std_dev_m)
    plot_ratios("N",ratios_n,ns,ax2,title,std_dev_n)
    fig.show()

    # 3D Plots
    if COMPARE == "time":
        no_of_plots = 3
        fig = plt.figure(figsize=(24,6))
        ax1 = fig.add_subplot(1, no_of_plots, 1,projection='3d')
        ax2 = fig.add_subplot(1, no_of_plots, 2, projection='3d')
        ax3 = fig.add_subplot(1, no_of_plots, 3, projection='3d')

        plot_3d('G2',ms,ns,g2_ratios,ax1,0.001)
        if WEIGHTED == True:
            plot_3d('Brute',ms,ns,brute_ratios,ax2,2.6)
        else:
            plot_3d('Brute',ms,ns,brute_ratios,ax2,0.4)

        plot_3d('ILP',ms,ns,ilp_ratios,ax3,0.1)

        fig.show()
    else:
        fig = plt.figure(figsize=(6,6))
        ax1 = fig.add_subplot(1, 1, 1,projection='3d')

        plot_3d('G2',ms,ns,g2_ratios,ax1,2)

        fig.show()
```

Example Instance

Sample run for a small n.

In [ ]:
```
example_universal_set = [1,2,3,4,5]
example_family_of_subsets = [[4,1,3],[2,5],[1,4,3,2]]
example_weights = {(4,1,3):5,(2,5):10,(1,4,3,2): 3}

print("Unweighted:")
_ = compare(example_universal_set, example_family_of_subsets, {}, PRINT = True)

print("\nWeighted:")
_ = compare(example_universal_set, example_family_of_subsets, example_weights, PRINT = True)
```

```
Unweighted:
Comparision of Algorithms
```

Algorithms	Set Covers	Length	Ratio wrt Brute Force
Brute Force	[[4, 1, 3], [2, 5]]	2	1.0
G2 Algo	[[1, 4, 3, 2], [2, 5]]	2	1.0

```
Weighted:
Comparision of Algorithms
```

Algorithms	Set Covers	Cost	Ratio wrt Brute Force
Brute Force	[[2, 5], [1, 4, 3, 2]]	13	1.0
G2 Algo	[[1, 4, 3, 2], [2, 5]]	13	1.0

Comparision of Algorithms

Execution Time

Unweighted Set Cover

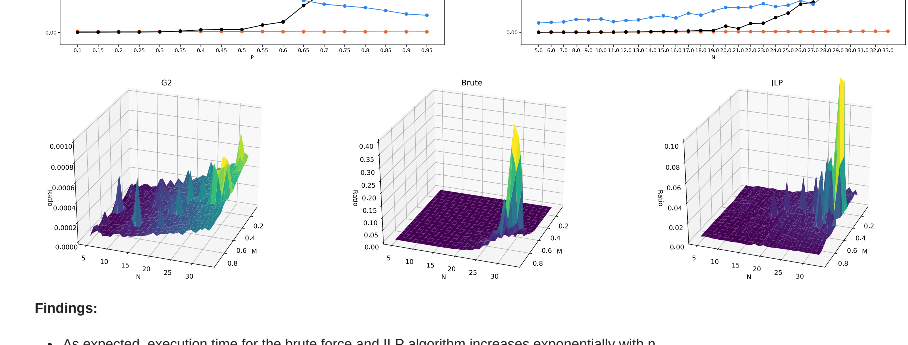
In [ ]:
```
compare_algos(5, 33, 0.1, 0.95, 0.05, lp="none", COMPARE = "time", WEIGHTED = False)
```

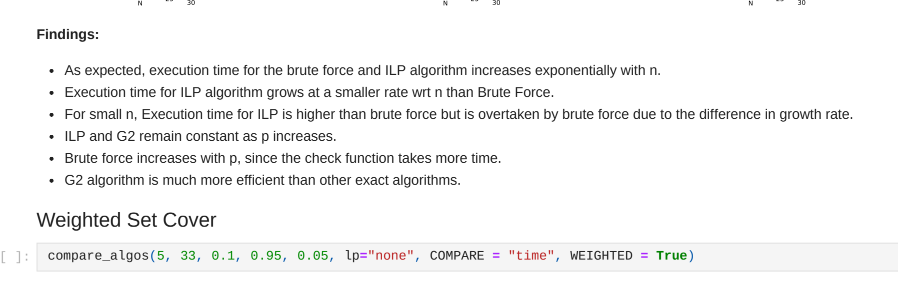
```
--------------------- Time taken vs N ---------------------
```

	Average				Standard Deviation			
	N	g2	brute	ilp	N	g2	brute	ilp
0	5	0.000050	0.000024	0.003454	5	0.000014	0.000008	0.000619
1	6	0.000084	0.000039	0.003689	6	0.000049	0.000031	0.000922
2	7	0.000101	0.000043	0.003851	7	0.000092	0.000035	0.001038
3	8	0.000062	0.000036	0.004746	8	0.000015	0.000012	0.001499
4	9	0.000065	0.000042	0.004595	9	0.000015	0.000030	0.001396
5	10	0.000072	0.000055	0.004915	10	0.000007	0.000023	0.001836
6	11	0.000081	0.000073	0.003913	11	0.000021	0.000045	0.000593
7	12	0.000104	0.000120	0.004379	12	0.000035	0.000088	0.001264
8	13	0.000103	0.000138	0.004584	13	0.000034	0.00011	0.001348
9	14	0.000152	0.000233	0.005515	14	0.000113	0.000194	0.001788
10	15	0.000117	0.000247	0.006100	15	0.000017	0.000182	0.001939
11	16	0.000139	0.000423	0.005321	16	0.000034	0.000271	0.002110
12	17	0.000138	0.000473	0.007059	17	0.000032	0.000514	0.006593
13	18	0.000157	0.000695	0.006343	18	0.000019	0.000744	0.001420
14	19	0.000175	0.000671	0.007917	19	0.000052	0.000681	0.007124
15	20	0.000206	0.002258	0.009180	20	0.000047	0.002619	0.002620
16	21	0.000193	0.001427	0.009087	21	0.000031	0.001731	0.002762
17	22	0.000231	0.003276	0.009341	22	0.000083	0.003536	0.003309
18	23	0.000231	0.003366	0.010586	23	0.000079	0.003951	0.008220
19	24	0.000242	0.005445	0.009499	24	0.000053	0.006908	0.001929
20	25	0.000247	0.007152	0.010038	25	0.000042	0.008673	0.001771
21	26	0.000278	0.010422	0.011760	26	0.000072	0.015156	0.009096
22	27	0.000281	0.011097	0.010362	27	0.000050	0.017041	0.002659
23	28	0.000306	0.024437	0.014017	28	0.000048	0.037189	0.012407
24	29	0.000354	0.026984	0.022035	29	0.000108	0.038937	0.019673
25	30	0.000386	0.046953	0.020568	30	0.000123	0.074603	0.021008
26	31	0.000399	0.048799	0.022244	31	0.000141	0.070642	0.021633
27	32	0.000422	0.084937	0.026036	32	0.000118	0.155865	0.032939
28	33	0.000387	0.092375	0.016959	33	0.000080	0.154950	0.019517

```
--------------------- Time taken vs M ---------------------
```

	Average				Standard Deviation			
	M	g2	brute	ilp	M	g2	brute	ilp
0	0.10	0.000254	0.000069	0.009023	0.10	0.000178	0.000058	0.003869
1	0.15	0.000231	0.000076	0.009872	0.15	0.000143	0.000054	0.004440
2	0.20	0.000228	0.000106	0.009976	0.20	0.000141	0.000082	0.004536
3	0.25	0.000221	0.000103	0.009427	0.25	0.000133	0.000077	0.004172
4	0.30	0.000216	0.000144	0.009274	0.30	0.000124	0.000115	0.003758
5	0.35	0.000204	0.000338	0.016057	0.35	0.000119	0.000436	0.020256
6	0.40	0.000219	0.000624	0.023955	0.40	0.000162	0.000896	0.035642
7	0.45	0.000244	0.000706	0.014091	0.45	0.000189	0.000778	0.017540
8	0.50	0.000223	0.000714	0.009546	0.50	0.000164	0.001038	0.006209
9	0.55	0.000192	0.001749	0.012727	0.55	0.000102	0.002996	0.010967
10	0.60	0.000188	0.002465	0.009024	0.60	0.000117	0.003388	0.009468
11	0.65	0.000167	0.006283	0.007510	0.65	0.000095	0.010520	0.005848
12	0.70	0.000160	0.009278	0.006621	0.70	0.000088	0.015123	0.003099
13	0.75	0.000185	0.019255	0.006195	0.75	0.000112	0.035355	0.002982
14	0.80	0.000165	0.030183	0.005829	0.80	0.000087	0.058318	0.002470
15	0.85	0.000157	0.048496	0.005126	0.85	0.000083	0.097805	0.002189
16	0.90	0.000154	0.058464	0.004322	0.90	0.000083	0.127689	0.001331
17	0.95	0.000167	0.051960	0.004034	0.95	0.000075	0.107892	0.000810





Findings:

- As expected, execution time for the brute force and ILP algorithm increases exponentially with n.
- Execution time for ILP algorithm grows at a smaller rate wrt n than Brute Force.
- For small n, Execution time for ILP is higher than brute force but is overtaken by brute force due to the difference in growth rate.
- ILP and G2 remain constant as p increases.
- Brute force increases with p, since the check function takes more time.
- G2 algorithm is much more efficient than other exact algorithms.

Weighted Set Cover

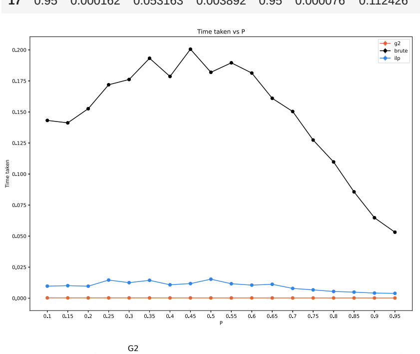
In [ ]:
```
compare_algos(5, 33, 0.1, 0.95, 0.05, lp="none", COMPARE = "time", WEIGHTED = True)
```

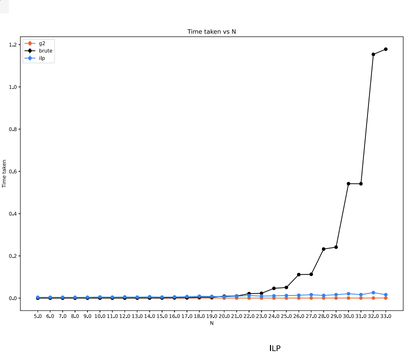
-------------------- Time taken vs N --------------------

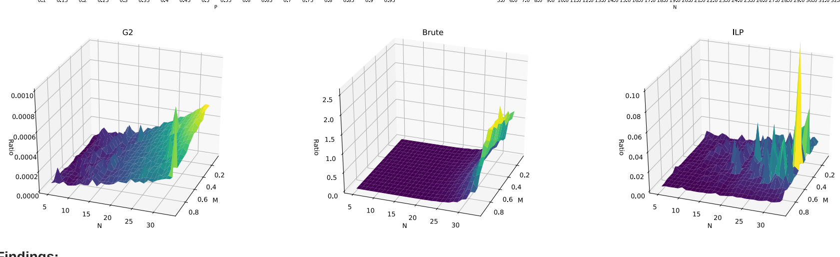
	Average				Standard Deviation			
	N	g2	brute	ilp	N	g2	brute	ilp
0	5	0.000050	0.000036	0.003903	5	0.000010	0.000034	0.001623
1	6	0.000068	0.000047	0.004025	6	0.000034	0.000020	0.001552
2	7	0.000089	0.000048	0.004042	7	0.000030	0.000011	0.001222
3	8	0.000080	0.000085	0.003883	8	0.000017	0.000035	0.000951
4	9	0.000082	0.000086	0.004054	9	0.000029	0.000033	0.001454
5	10	0.000111	0.000190	0.005354	10	0.000041	0.000084	0.001820
6	11	0.000097	0.000165	0.004643	11	0.000024	0.000051	0.002608
7	12	0.000128	0.000386	0.005699	12	0.000040	0.000176	0.003284
8	13	0.000117	0.000372	0.004723	13	0.000026	0.000102	0.001238
9	14	0.000139	0.000822	0.006393	14	0.000042	0.000313	0.003371
10	15	0.000138	0.000860	0.005055	15	0.000029	0.000265	0.001416
11	16	0.000155	0.001990	0.005790	16	0.000035	0.000741	0.001980
12	17	0.000143	0.001918	0.007530	17	0.000014	0.000575	0.006238
13	18	0.000177	0.004453	0.008910	18	0.000043	0.001930	0.004983
14	19	0.000173	0.004338	0.008138	19	0.000024	0.001507	0.007460
15	20	0.000195	0.009284	0.006589	20	0.000030	0.002293	0.001672
16	21	0.000201	0.010042	0.008874	21	0.000026	0.003626	0.004699
17	22	0.000222	0.021296	0.012483	22	0.000033	0.006550	0.010092
18	23	0.000239	0.022841	0.009745	23	0.000036	0.007246	0.004536
19	24	0.000263	0.046368	0.010902	24	0.000037	0.012881	0.005596
20	25	0.000269	0.050492	0.012323	25	0.000044	0.015884	0.007055
21	26	0.000297	0.111479	0.013635	26	0.000051	0.035396	0.008391
22	27	0.000322	0.112841	0.016332	27	0.000085	0.033342	0.013330
23	28	0.000322	0.232390	0.012789	28	0.000063	0.068236	0.007831
24	29	0.000348	0.241299	0.016541	29	0.000071	0.076254	0.009081
25	30	0.000366	0.541815	0.020893	30	0.000071	0.172182	0.015255
26	31	0.000384	0.541425	0.016598	31	0.000070	0.159663	0.010999
27	32	0.000412	1.153941	0.026095	32	0.000082	0.345696	0.029231
28	33	0.000436	1.178442	0.016263	33	0.000129	0.341238	0.013031

-------------------- Time taken vs M --------------------

	Average				Standard Deviation			
	M	g2	brute	ilp	M	g2	brute	ilp
0	0.10	0.000253	0.143189	0.009712	0.10	0.000150	0.314545	0.004194
1	0.15	0.000248	0.141242	0.010089	0.15	0.000145	0.288009	0.005869
2	0.20	0.000244	0.152630	0.009669	0.20	0.000142	0.307096	0.004238
3	0.25	0.000248	0.171917	0.014613	0.25	0.000134	0.366899	0.010987
4	0.30	0.000235	0.176162	0.012515	0.30	0.000152	0.365597	0.009273
5	0.35	0.000228	0.193260	0.014401	0.35	0.000132	0.418265	0.015031
6	0.40	0.000215	0.178605	0.010795	0.40	0.000123	0.365917	0.008355
7	0.45	0.000216	0.200573	0.011771	0.45	0.000119	0.416593	0.010170
8	0.50	0.000201	0.181953	0.015317	0.50	0.000112	0.377923	0.024492
9	0.55	0.000200	0.189597	0.011642	0.55	0.000107	0.407184	0.010702
10	0.60	0.000200	0.181351	0.010537	0.60	0.000110	0.394374	0.009812
11	0.65	0.000191	0.161009	0.011209	0.65	0.000093	0.340160	0.011837
12	0.70	0.000183	0.150383	0.007875	0.70	0.000091	0.323315	0.007055
13	0.75	0.000185	0.127387	0.006700	0.75	0.000086	0.270029	0.004190
14	0.80	0.000170	0.109751	0.005417	0.80	0.000086	0.231255	0.002210
15	0.85	0.000193	0.085643	0.004884	0.85	0.000148	0.186815	0.002151
16	0.90	0.000165	0.064789	0.004124	0.90	0.000074	0.135751	0.001289
17	0.95	0.000162	0.053163	0.003892	0.95	0.000076	0.112426	0.001057







Findings:

- Since we are not early stopping in the weighted set cover, the execution time for brute force is considerably higher.
- As expected, execution time for the brute force increases exponentially with n.
- For small n, Execution time for ILP is higher than brute force but is overtaken by brute force due to the huge difference in growth rate.
- ILP and G2 remain constant as p increases.

Performance Ratios

Unweighted Set Cover

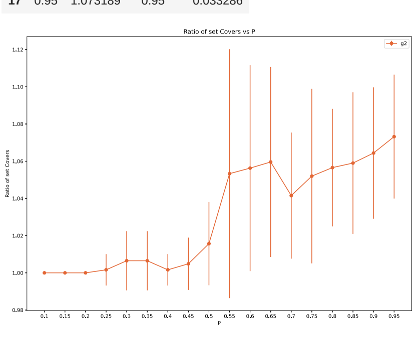
```
In [ ]: compare_algos(5, 32, 0.1, 0.95, 0.05, lp = "ilp", COMPARE = "size", WEIGHTED = False)
```

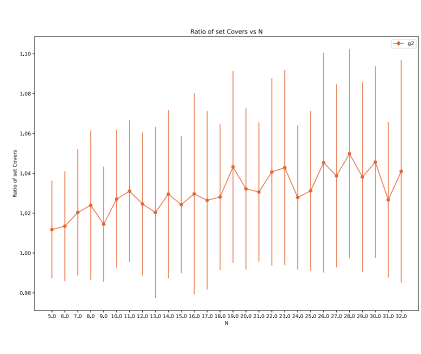
-------------------- Ratio of set Covers vs N --------------------

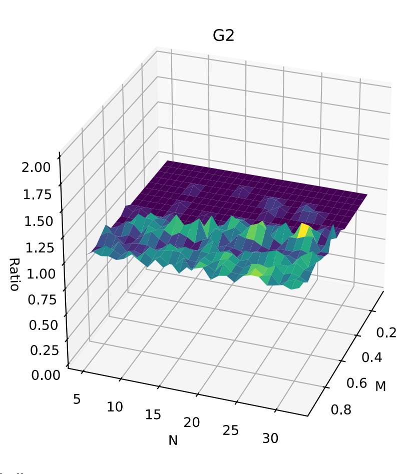
	Average		Standard Deviation	
	N	g2	N	g2
0	5	1.011785	5	0.024512
1	6	1.013468	6	0.027611
2	7	1.020370	7	0.031682
3	8	1.023990	8	0.037477
4	9	1.014478	9	0.028904
5	10	1.027104	10	0.034539
6	11	1.031061	11	0.035670
7	12	1.024627	12	0.035873
8	13	1.020370	13	0.043004
9	14	1.029557	14	0.042273
10	15	1.024299	15	0.034481
11	16	1.029684	16	0.050457
12	17	1.026431	17	0.044819
13	18	1.028114	18	0.036560
14	19	1.043238	19	0.048144
15	20	1.032203	20	0.040422
16	21	1.030640	21	0.034913
17	22	1.040711	22	0.047010
18	23	1.042857	23	0.048894
19	24	1.027874	24	0.036206
20	25	1.031241	25	0.040139
21	26	1.045330	26	0.055167
22	27	1.038733	27	0.045939
23	28	1.049864	28	0.052439
24	29	1.038157	29	0.047568
25	30	1.045647	30	0.048090
26	31	1.026673	31	0.039027
27	32	1.040995	32	0.055864

-------------------- Ratio of set Covers vs M --------------------

	Average		Standard Deviation	
	M	g2	M	g2
0	0.10	1.000000	0.10	0.000000
1	0.15	1.000000	0.15	0.000000
2	0.20	1.000000	0.20	0.000000
3	0.25	1.001623	0.25	0.008435
4	0.30	1.006494	0.30	0.015906
5	0.35	1.006494	0.35	0.015906
6	0.40	1.001623	0.40	0.008435
7	0.45	1.004870	0.45	0.014059
8	0.50	1.015693	0.50	0.022357
9	0.55	1.053301	0.55	0.066891
10	0.60	1.056277	0.60	0.055354
11	0.65	1.059578	0.65	0.051081
12	0.70	1.041504	0.70	0.033888
13	0.75	1.051979	0.75	0.046893
14	0.80	1.056548	0.80	0.031567
15	0.85	1.058987	0.85	0.038083
16	0.90	1.064378	0.90	0.035338
17	0.95	1.073189	0.95	0.033286







Findings:

- Ratio of G2 to optimal is generally close to 1, so it is a pretty good approximation algorithm.
- There is a slight increase as n (number of elements in universal set) and p (size of subsets in the family of subsets) increase, which is expected.

Weighted Set Cover

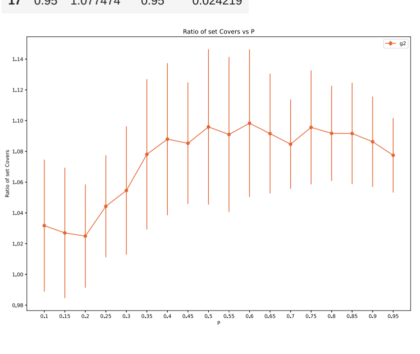
```
In [ ]: compare_algos(5, 32, 0.1, 0.95, 0.05, lp = "ilp", COMPARE = "size", WEIGHTED=True)
```

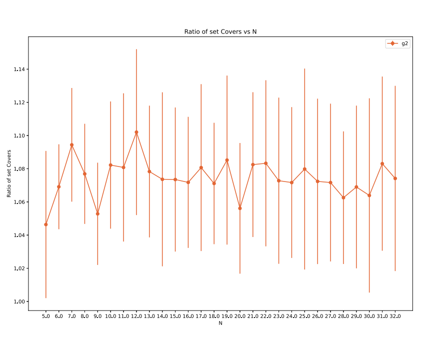
-------------------- Ratio of set Covers vs N --------------------

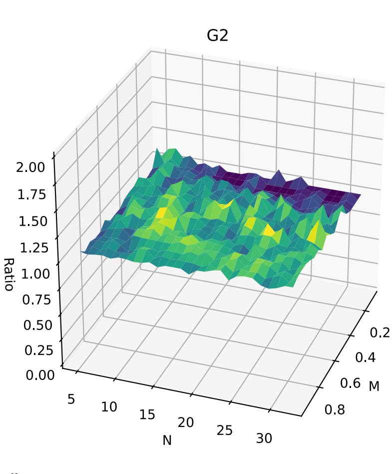
	Average		Standard Deviation	
	N	g2	N	g2
0	5	1.046352	5	0.044351
1	6	1.069125	6	0.025591
2	7	1.094386	7	0.034255
3	8	1.076892	8	0.030174
4	9	1.052811	9	0.030807
5	10	1.082189	10	0.038338
6	11	1.080772	11	0.044670
7	12	1.102044	12	0.049990
8	13	1.078303	13	0.039694
9	14	1.073614	14	0.052437
10	15	1.073480	15	0.043416
11	16	1.071755	16	0.039497
12	17	1.080669	17	0.050294
13	18	1.071093	18	0.036577
14	19	1.085197	19	0.050932
15	20	1.056112	20	0.039374
16	21	1.082464	21	0.043632
17	22	1.083266	22	0.050047
18	23	1.072759	23	0.050074
19	24	1.071670	24	0.045457
20	25	1.079795	25	0.060556
21	26	1.072364	26	0.049837
22	27	1.071654	27	0.047537
23	28	1.062534	28	0.039958
24	29	1.068991	29	0.049028
25	30	1.063887	30	0.058548
26	31	1.083045	31	0.052492
27	32	1.074128	32	0.055813

-------------------- Ratio of set Covers vs M --------------------

	Average		Standard Deviation	
	M	g2	M	g2
0	0.10	1.031728	0.10	0.042924
1	0.15	1.026993	0.15	0.042407
2	0.20	1.024911	0.20	0.033602
3	0.25	1.044306	0.25	0.033158
4	0.30	1.054556	0.30	0.041765
5	0.35	1.078090	0.35	0.048900
6	0.40	1.087962	0.40	0.049423
7	0.45	1.085267	0.45	0.039557
8	0.50	1.095923	0.50	0.050529
9	0.55	1.091003	0.55	0.050386
10	0.60	1.098291	0.60	0.047997
11	0.65	1.091563	0.65	0.038931
12	0.70	1.084674	0.70	0.029046
13	0.75	1.095641	0.75	0.037095
14	0.80	1.091727	0.80	0.030923
15	0.85	1.091636	0.85	0.032922
16	0.90	1.086266	0.90	0.029422
17	0.95	1.077474	0.95	0.024219







Findings:

- Ratio of weighted G2 to optimal is generally close to 1, so it is a pretty good approximation algorithm.
- Even as n (number of elements in universal set) and p (size of subsets in the family of subsets) increase, the ratio of weights_G2 / weights_optimal does not change much.

Travelling Salesman Problem

Introduction

The Travelling Salesman Problem:

Given a complete weighted graph, the problem is to find the shortest possible hamiltonian tour.

The Metric Traveling Salesman Problem:

The special case of the TSP where the input instances satisfy the triangle inequality.
$\forall i, j, k \in V,$

$d(i, k) \leq d(i, j) + d(j, k)$

Assumptions:

- The graph is complete.

Implementation:
We use two metrics for the of algorithms:

1. Performance Ratio: ratios of lengths of vertex covers (wrt Brute force or ILP).
2. Execution time of various algorithms.

Import

In [1]:
```
%%capture
import random
from math import sqrt,ceil
from itertools import permutations, product
import matplotlib.pyplot as plt
from sys import stdout as out
import numpy as np
import networkx as nx
import copy
import pandas as pd
import time
from IPython.display import set_matplotlib_formats
set_matplotlib_formats('pdf','svg')
```

In [2]:
```
%%capture
!pip install mip
from mip import Model, xsum, minimize, BINARY
```

Graph Generation

This function generates a symmetric Adjacency Matrix representation of a randomly generated complete weighted graph.

The generated weights should follow triangle inequality (metric TSP).

Implementation:

- We generate weights randomly in the range ($\lceil MAX\_WEIGHT/2 \rceil$, MAX_WEIGHT-1) which ensures that any 3 numbers generated will follow triangle inequality.

In [3]:
```
def generate_graph(n, MAX_WEIGHT = 100):

    # Adj Matrix of 0's
    graph = [[0]*n for i in range(n)]
    b = MAX_WEIGHT - 1
    a = MAX_WEIGHT // 2
    for i in range(n):
        for j in range(i+1,n):
            weight = (random.randint(a,b))
            graph[i][j] = weight
            graph[j][i] = weight

    return graph
```

Exact Algorithms

Brute Force

Iterate through all possible hamiltonian tours possible for the graph (Here, a circular permutation of list of vertices represents a possible tour) and take the tour with minimum path length.

Algorithm Brute Force:

Input: Adjacency matrix representation of a graph G.
Output: A Hamiltonian tour in G (A List of vertices) and the cost of this tour.

1. Get all circular permutations of vertex list. Each permutation represents a hamiltonian tour.
2. For every tour, calculate the path length and get the minimum.

Implementation :

- Rather than checking all permutations of vertex list, we fix the first vertex to 0 and get permutations for the rest of the list to avoid multiple representations for a tour. (For example 0 $\rightarrow$ 1 $\rightarrow$ 2 $\rightarrow$ 0 and 1 $\rightarrow$ 2 $\rightarrow$ 0 $\rightarrow$ 1 are the same tour, if we fix 0 as first vertex, we avoid this repetition).
- Doing this still leaves two representations for every tour, clockwise and anticlockwise, increasing execution time by a factor of 2.

In [4]:
```
def brute_force(graph):
    min_length = 100000000
    min_hamiltonian_path = []

    n = len(graph)

    # Generate all permutations from 1 to n-1
    l = list(permutations(range(1, n)))

    # Calculate Path Length of the particular permutation
    for perm in l:
        path_length = graph[0][perm[0]]
        for i in range(len(perm)-1):
            path_length += graph[perm[i]][perm[(i+1)]]
        path_length += graph[perm[len(perm)-1]][0]

        if (path_length < min_length):
            min_length = path_length
            min_hamiltonian_path = list(perm)

    # Go back to the start point
    min_hamiltonian_path.append(0)
    return [0] + (min_hamiltonian_path), min_length
```

Integer Linear Programming

Input: Adjacency matrix representation of a graph G.
Output: A Hamiltonian tour in G (A List of vertices) and the cost of this tour.

$$\min \sum_{i=1}^{n} \sum_{j=1, j \neq i}^{n} c_{ij} x_{ij},$$

subject to

$$\sum_{i=1, i \neq j}^{n} x_{ij} = 1, \quad j = 1, 2, \ldots, n,$$
$$\sum_{j=1, j \neq i}^{n} x_{ij} = 1, \quad i = 1, 2, \ldots, n,$$
$$u_i - u_j + n x_{ij} \leq n - 1, \quad 2 \leq i \neq j \leq n,$$
$$x_{ij} \in \{0, 1\} \quad i, j = 1, 2, \ldots, n, \quad i \neq j,$$
$$u_i \in \mathrm{R}^+ \quad i = 1, 2, \ldots, n.$$

Notation

$$x_{ij} = \begin{cases} 1 & \text{if the route includes a direct link between vertices } i \text{ and } j, \\ 0 & \text{otherwise.} \end{cases}$$

In addition, for each vertex, ***i* = 1,2, ... , n**,
u_i is an auxiliary variable indicating the sequential order of each point in the produced route.
c_{ij} is the distance between the vertices i and j.

- The first set of equalities requires that each vertex is arrived at from exactly one other vertex.
- The second set of equalities requires that from each vertex there is a departure to exactly one next vertex.
- The third set of inequalities prevents the formation of sub-tours and thus guarantees the formation of a single tour, covering all the vertices.

In [5]:
```
def ilp(graph):
    c = graph
    n = len(graph)
    V = set([x for x in range(n)])

    model = Model()

    # binary variables indicating if arc (i,j) is used on the route or not
    x = [[model.add_var(var_type=BINARY) for j in V] for i in V]

    # continuous variable to prevent subtours: each vertex will have a different sequential id in the planned r
    y = [model.add_var() for i in V]

    # objective function: minimize the distance
    model.objective = minimize(xsum(c[i][j]*x[i][j] for i in V for j in V))

    # constraint : leave each vertex only once
    for i in V:
        model += xsum(x[i][j] for j in V - {i}) == 1

    # constraint : enter each vertex only once
    for i in V:
        model += xsum(x[j][i] for j in V - {i}) == 1

    # subtour elimination
    for (i, j) in product(V - {0}, V - {0}):
        if i != j:
            model += y[i] - (n+1)*x[i][j] >= y[j]-n

    # optimizing
    model.optimize()
    V = list(V)

    # checking if a solution was found
    if model.num_solutions:
        distance = model.objective_value
        hamiltonian_cycle = [V[0]]
        nc = 0
        while True:
            nc = [i for i in V if x[nc][i].x >= 0.99][0]
            hamiltonian_cycle.append(V[nc])
            if nc == 0:
                break
        return hamiltonian_cycle,int(distance)
```

Approximation Algorithms

MST Approximation

Algorithm MST:

Input: Adjacency matrix representation of a graph G.
Output: A Hamiltonian tour in G (A List of vertices) and the cost of this tour.

1. Find a minimum spanning tree T in G.
2. Construct a multigraph T' by making two copies of each edge in T.
3. Find an Eulerian tour ET in T'.
4. Construct a Hamiltonian tour by short-circuiting the Eulerian tour. That is, starting at any vertex, follow the Eulerian tour as long as new vertices are being visited. At any point where the Eulerian tour repeats a vertex, jump directly to the next unvisited vertex. Finally, complete the cycle by returning to the starting vertex.

Perfomance Ratio: 2

- Given any collection of edges H from G, $cost(H)$ denotes the sum of all the edge lengths/weights for the edges in H.
- T is the MST, T' is the multigraph made from doubling all the edges, ET is the Eulerian Tour.
- $cost(T) \leq cost(OPT(G))$ because any Hamiltonian cycle with an edge removes gives a spanning tree.
- Thus, $cost(ET) = cost(T') \leq 2 * cost(OPT(G))$ because $cost(T') = 2 * cost(T)$ and $cost(T) \leq cost(OPT(G))$.
- Finally short-cut procedure ensures that $cost(approx_{mst}(G)) \leq cost(ET)$.
- Therefore $cost(approx_{mst}(G)) \leq 2 * cost(OPT(G))$.
- This gives an upper bound of **2**.

Implementation :

- We convert the adjacency matrix representation of a graph into a networkx graph object in order to use multiple useful functions from the networkx library.

In [6]:
```
def mst_algo(graph):

    # Make an Networkx Graph
    nx_graph = nx.to_networkx_graph(np.array(graph))

    # Find Minimum Spanning Tree of the graph
    mst = nx.minimum_spanning_tree(nx_graph)

    # Construct a multigraph by making 2 copies of each edge
    G = nx.MultiGraph()
    edge_list = nx.to_edgelist(mst)
    for edge in edge_list:
        G.add_edge(edge[0],edge[1],weight=edge[2]['weight'])
        G.add_edge(edge[0],edge[1],weight=edge[2]['weight'])
    multigraph_mst = G

    # Find an euler tour
    euler_tour = list(nx.eulerian_circuit(multigraph_mst))

    # Construct a hamiltonian tour
    hamiltonian_cycle = []

    for edge in euler_tour:
        if edge[0] not in hamiltonian_cycle:
            hamiltonian_cycle.append(edge[0])

    hamiltonian_cycle.append(euler_tour[0][0])

    # Find cost of the tour
    distance = 0
    for i in range(len(hamiltonian_cycle)-1):
        distance += graph[hamiltonian_cycle[i]][hamiltonian_cycle[i+1]]
    return hamiltonian_cycle, distance
```

Christofides Approximation

We convert the adjacency matrix representation of a graph into a networkx graph object in order to use multiple useful functions from the networkx library.

Algorithm Christofides:

Input: Adjacency matrix representation of a graph G.
Output: A Hamiltonian tour in G (A List of vertices) and the cost of this tour.

1. Create a minimum spanning tree T of G.
2. Let O be the set of vertices with odd degree in T. By the handshaking lemma, O has an even number of vertices.
3. Find a minimum-weight perfect matching M in the induced subgraph given by the vertices from O.
4. Combine the edges of M and T to form a connected multigraph H in which each vertex has even degree.
5. Form an Eulerian circuit in H.
6. Make the circuit found in previous step into a Hamiltonian circuit by skipping repeated vertices (shortcutting).

Performance Ratio: 1.5

- Given any collection of edges H from G, $cost(H)$ denotes the sum of all the edge lengths/weights for the edges in H.
- Let $OPT(G)$ be the optimal TSP tour and Let T be the Minimum spanning tree.
- Let M be the the minimum weight matching on the set O of odd degree vertices in the MST T.
- **Claim 1**: $cost(T) \leq cost(OPT(G))$
  Removal of an edge from C gives a spanning tree.
- **Claim 2**: $cost(M) \leq cost(OPT(G))/2$
- Take a shortcut tour X only on the set of odd-degree vertices O.
- These edges partition into two matchings M_1 and M_2 such that $cost(M_1) + cost(M_2) \leq cost(OPT(G))$
- $cost(approx_{ch}) \leq cost(M) + cost(T) \leq 1.5 * cost(OPT(G))$
- $cost(approx_{ch})/cost(OPT(G)) \leq 1.5$

Implementation :

- As the neworkx subgraph function does not work properly with max_weight_matching function, we create a copy of the graph and remove odd degree vertices.
- Find a minimum weight matching for our subgraph:
  - Since networkx only has a maximum weight matching function, we subtract our weights from a large number (10^8).
  - An alternative is to just negate the weights but the networkx function only accepts positive weights.

In [7]:
```
def christofides(graph):

    # Make an Networkx Graph
    nx_graph = nx.to_networkx_graph(np.array(graph))

    # Create a minimum spanning tree
    mst = nx.minimum_spanning_tree(nx_graph)

    # Construct a multigraph
    multigraph_mst = nx.MultiGraph()
    edge_list = nx.to_edgelist(mst)
    for edge in edge_list:
        multigraph_mst.add_edge(edge[0],edge[1],weight=edge[2]['weight'])

    # Find the set of vertices with odd degree in the MST and remove from original graph
    odd_subgraph = copy.deepcopy(nx_graph)
    nodes = [x for x in nx.nodes(nx_graph)]
    odd_nodes = [x for x in nx.nodes(mst) if len(list(nx.neighbors(mst,x))) % 2 == 1]
    for n in nodes:
        if n not in odd_nodes:
            odd_subgraph.remove_node(n)

    # Find a minimum-weight perfect matching of the odd subgraph
    # As networkx only has a max_weight_matching function, we inverted the weights by subtracting them from a 
    inf = 100000000
    edge_list = nx.to_edgelist(odd_subgraph)
    for edge in edge_list:
        odd_subgraph[edge[0]][edge[1]]['weight'] = inf - odd_subgraph[edge[0]][edge[1]]['weight']

    perf_matching = nx.max_weight_matching(odd_subgraph, maxcardinality=False, weight='weight')

    # Reverting the weights back to normal and adding to the multigraph
    for edge in perf_matching:
        for edge_l in edge_list:
            if ((edge_l[0] == edge[0] and edge_l[1] == edge[1]) or (edge_l[0] == edge[1] and edge_l[1] == edge[
                multigraph_mst.add_edge(edge[0],edge[1],weight = inf - edge_l[2]['weight'])

    # Find an euler tour
    euler_tour = list(nx.eulerian_circuit(multigraph_mst))

    # Construct a hamiltonian tour
    hamiltonian_cycle = []

    for edge in euler_tour:
        if edge[0] not in hamiltonian_cycle:
            hamiltonian_cycle.append(edge[0])

    hamiltonian_cycle.append(euler_tour[0][0])

    # Find cost of the tour
    distance = 0
    for i in range(len(hamiltonian_cycle)-1):
        distance += graph[hamiltonian_cycle[i]][hamiltonian_cycle[i+1]]
    return hamiltonian_cycle, distance
```

Functions for comparison

In [8]:
```
# Colours for lines in the graph
colors = {"brute":'k',"ilp":'#2f85ed',"mst":'#e36634',"christof":'#1fde1f'}

# Print the datapoints in a table format
def print_table(wrt, lst, ratios, title):
    ratios[wrt] = lst
    df = pd.DataFrame.from_dict(ratios)
    df.columns = df.columns.str.strip()
    cols = df.columns.tolist()
    cols = cols[-1:] + cols[:-1]
    df = df[cols]
    del ratios[wrt]
    df.columns = pd.MultiIndex.from_product([[title], df.columns])
    return df

# Plot all the ratios in one single line graph
def plot_ratios(wrt,ratios,lst,ax,title = "Ratio of Costs",std_dev = {}):
    for key in ratios.keys():
        ax.errorbar([x for x in range(len(lst))], ratios[key], yerr=std_dev[key], fmt='-o', label=key, color=co
    plt.sca(ax)
    ax.set_xlabel(wrt)
    ax.set_ylabel(title)
    ax.set_title(title+" vs "+wrt)
    ax.legend()
    plt.xticks(list(range(len(lst))), [ceil(100*x)/100 for x in lst])
```

Parameterization:

- *lp* is a parameter which can be specified to be "ilp" or "none".
- *PRINT* is used to specify whether to print or not.
- *COMPARE* is used to specify whether we are comparing cost or time.

In [9]:
```
# This function compares both times and costs of mst and christofides algorithm with brute or ILP.
def compare(graph, lp = "none", PRINT = False, COMPARE = "cost"):
    t_brute_start = time.time()
    if lp == "ilp":
        brute_min_ham, brute_min_ham_length = ilp(graph)
    else:
        brute_min_ham, brute_min_ham_length = brute_force(graph)
    t_brute_end = time.time()

    t_mst_start = time.time()
    mst_min_ham, mst_min_ham_length = mst_algo(graph)
    t_mst_end = time.time()
    t_christof_start = time.time()
    christof_min_ham, christof_min_ham_length = christofides(graph)
    t_christof_end = time.time()

    t_brute = t_brute_end - t_brute_start
    t_mst = t_mst_end - t_mst_start
    t_christof = t_christof_end - t_christof_start

    mst_ratio = mst_min_ham_length / brute_min_ham_length
    christof_ratio = christof_min_ham_length / brute_min_ham_length

    if COMPARE == "time":
        t_ilp_start = time.time()
        ilp_min_ham, ilp_min_ham_length = ilp(graph)
        t_ilp_end = time.time()
        t_ilp = t_ilp_end - t_ilp_start
        time_dict = {'brute':t_brute,'mst':t_mst,'christof':t_christof,'ilp':t_ilp}
        return time_dict

    ratios = {"mst" : mst_ratio, "christof" : christof_ratio}

    if (PRINT):
        info = {}
        info["Hamiltonian Cycles"] = [brute_min_ham,mst_min_ham,christof_min_ham]
        info["Distances"] = [brute_min_ham_length, mst_min_ham_length, christof_min_ham_length]
        info["Ratio wrt Brute Force"] = [1,mst_ratio,christof_ratio]
        df = pd.DataFrame.from_dict(info)

        if lp == "rlp":
            df.index = ["Relaxed Lin Prog","MST Algo","Christofides"]
        elif lp == "ilp":
            df.index = ["Integer Lin Prog","MST Algo","Christofides"]
        else:
            df.index = ["Brute Force","MST Algo","Christofides"]
        df.index.name = "Algorithms"
        print("--------------------Comparison of Algorithms--------------------")
        display(df)

    return ratios
```

Parameterization:

- *n_start* and *n_end* specify the range of number of vertices.
- *max_weight_start*, *max_weight_end* and *max_weight_increment* determine the MAX_WEIGHT parameter for Graph generation, for each n, all algorithms are run $(max\_weight\_start - max\_weight\_end)/max\_weight\_increment$ times.
- *lp* is a parameter which can be specified to be "ilp" or "none".
- *COMPARE* is used to specify whether we are comparing cost or time.

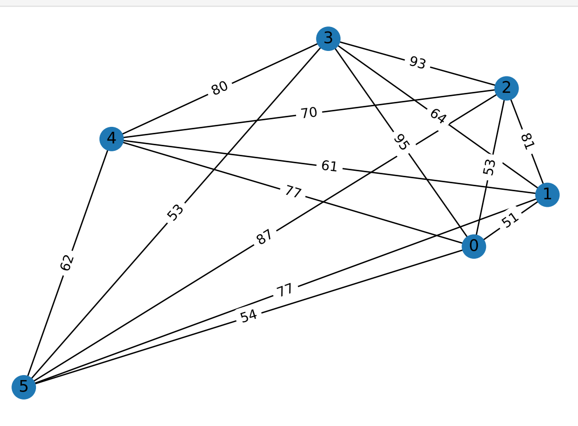
In [10]:
```
# This function calls the compare() function over a range of n and max_weights
def compare_algos(n_start,n_end,max_weight_start,max_weight_end,max_weight_increment,lp = "none",COMPARE = "cos
    ns = []
    mst_ratios = []
    christof_ratios = []
    brute_ratios = []
    ilp_ratios = []

    for n in range(n_start,n_end+1):
        p = max_weight_start
        mst_ratios_p = []
        christof_ratios_p = []
        brute_ratios_p = []
        ilp_ratios_p = []
        while (p <= max_weight_end):
            mst_ratio_sum = 0
            christof_ratio_sum = 0
            brute_ratio_sum = 0
            ilp_ratio_sum = 0

            # Generate Graph and compare ratios
            graph = generate_graph(n, MAX_WEIGHT = 100)
            ratios = compare(graph, lp = lp, PRINT = False, COMPARE = COMPARE)

            # Update the ratios list
            mst_ratio = ratios['mst']
            christof_ratio = ratios['christof']
            mst_ratio_sum += mst_ratio
            christof_ratio_sum += christof_ratio
            if COMPARE == "time":
                brute_ratio = ratios['brute']
                brute_ratio_sum += brute_ratio
                ilp_ratio = ratios['ilp']
                ilp_ratio_sum += ilp_ratio
            p += max_weight_increment
            mst_ratios_p.append(mst_ratio_sum)
            christof_ratios_p.append(christof_ratio_sum)
            brute_ratios_p.append(brute_ratio_sum)
            ilp_ratios_p.append(ilp_ratio_sum)

        ns.append(n)
        mst_ratios.append(mst_ratios_p)
        christof_ratios.append(christof_ratios_p)
        brute_ratios.append(brute_ratios_p)
        ilp_ratios.append(ilp_ratios_p)

    # Convert Lists to numpy arrays for plotting
    mst_ratios = np.array(mst_ratios)
    christof_ratios = np.array(christof_ratios)

    mst_ratios_n = np.sum(mst_ratios,axis=1)/len(mst_ratios[0])
    mst_std_dev_n = np.std(mst_ratios,axis=1)
    christof_ratios_n = np.sum(christof_ratios,axis=1)/len(christof_ratios[0])
    christof_std_dev_n = np.std(christof_ratios,axis=1)

    if COMPARE == "time":
        brute_ratios = np.array(brute_ratios)
        brute_ratios_n = np.sum(brute_ratios,axis=1)/len(brute_ratios[0])
        brute_std_dev_n = np.std(brute_ratios,axis=1)
        ilp_ratios = np.array(ilp_ratios)
        ilp_ratios_n = np.sum(ilp_ratios,axis=1)/len(ilp_ratios[0])
        ilp_std_dev_n = np.std(ilp_ratios,axis=1)

    # Plot and display ratios and standard deviation
    fig = plt.figure(figsize=(30,11))
    ax1 = fig.add_subplot(1, 2, 1)
    ax2 = fig.add_subplot(1, 2, 2)

    if COMPARE == "time":
        ratios_n_1 = {"brute": brute_ratios_n,"ilp": ilp_ratios_n, "mst":mst_ratios_n,"christof":christof_ratio
        std_dev_n_1 = {"brute" : brute_std_dev_n, "ilp" : ilp_std_dev_n, "mst":mst_std_dev_n, "christof":christ
        ratios_n_2 = {"ilp": ilp_ratios_n, "mst":mst_ratios_n,"christof":christof_ratios_n}
        std_dev_n_2 = {"ilp" : ilp_std_dev_n, "mst":mst_std_dev_n, "christof":christof_std_dev_n}
        ratios_n = ratios_n_1
        std_dev_n = std_dev_n_1
        title = "Time Taken"
    else:
        ratios_n_1 = {"christof":christof_ratios_n}
        std_dev_n_1 = {"christof":christof_std_dev_n}
        ratios_n_2 = {"mst":mst_ratios_n}
        std_dev_n_2 = {"mst":mst_std_dev_n}
        ratios_n = {"mst":mst_ratios_n,"christof":christof_ratios_n}
        std_dev_n = {"mst":[0]*len(mst_ratios_n), "christof":[0]*len(christof_ratios_n)}
        title = "Ratio of Costs"
        figall = plt.figure(figsize=(30,12))
        axall = figall.add_subplot(1, 1, 1)
        plot_ratios("N",ratios_n,ns,axall,title,std_dev_n)
        std_dev_n = {"mst":mst_std_dev_n, "christof":christof_std_dev_n}

    print("\n--------------------",title,"vs","N","--------------------")
    df1 = print_table("N", ns, ratios_n, "Average")
    df2 = print_table("N", ns, std_dev_n, "Standard Deviation")
    dft = pd.concat([df1, df2], axis=1)
    display(dft)

    plot_ratios("N",ratios_n_1,ns,ax1,title,std_dev_n_1)
    plot_ratios("N",ratios_n_2,ns,ax2,title,std_dev_n_2)
    fig.show()
```

Example Instance

Sample run for a small n.

In [11]:
```
# Generate a random graph of 6 vertices and weights from 50 to 99.
random.seed(165)
example_tsp = (generate_graph(n = 6, MAX_WEIGHT = 100))
```

In [12]:
```
# Display the graph
np_example_tsp = np.array(example_tsp)
example_graph = nx.to_networkx_graph(np_example_tsp)
pos = nx.spiral_layout(example_graph, resolution=0.7)  # positions for all nodes
nx.draw(example_graph, pos, with_labels=True)
labels = nx.get_edge_attributes(example_graph,'weight')
nx.draw_networkx_edge_labels(example_graph,pos,edge_labels=labels)
plt.axis()
plt.show()
```



In [13]:
```
_ = compare(example_tsp, lp = "none", PRINT = True)
```

--------------------Comparison of Algorithms--------------------

Algorithms	Hamiltonian Cycles	Distances	Ratio wrt Brute Force
Brute Force	[0, 1, 3, 5, 4, 2, 0]	353	1.000000
MST Algo	[0, 5, 3, 2, 1, 4, 0]	419	1.186969
Christofides	[0, 2, 5, 3, 4, 1, 0]	385	1.090652

Comparison of Algorithms

Execution Time

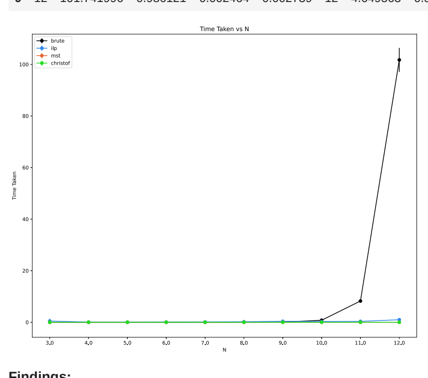
Implementation:

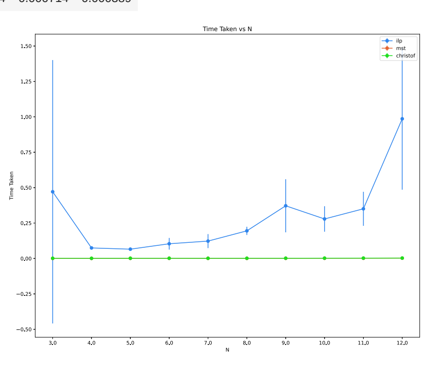
- Since brute force increases highly exponentially, it compresses the other graphs.
- We have displayed 2 other graphs for execution time without brute force.

In [14]:
```
compare_algos(3,12,100,500,100,lp = "none",COMPARE = "time")
```

-------------------- Time Taken vs N --------------------

	Average					Standard Deviation				
	N	brute	ilp	mst	christof	N	brute	ilp	mst	christof
0	3	0.000022	0.471032	0.000844	0.001320	3	0.000004	0.929756	0.000127	0.000475
1	4	0.000031	0.074448	0.000884	0.001101	4	0.000003	0.007475	0.000131	0.000211
2	5	0.000062	0.065678	0.001174	0.001356	5	0.000011	0.009813	0.000507	0.000323
3	6	0.000265	0.103877	0.001396	0.001445	6	0.000051	0.041033	0.000627	0.000448
4	7	0.001262	0.122729	0.001123	0.001248	7	0.000252	0.049384	0.000231	0.000161
5	8	0.026764	0.195055	0.001083	0.001512	8	0.034182	0.028881	0.000017	0.000138
6	9	0.078659	0.372005	0.001235	0.001634	9	0.003723	0.187337	0.000039	0.000163
7	10	0.780092	0.278973	0.001417	0.002055	10	0.022851	0.089953	0.000150	0.000318
8	11	8.247344	0.350626	0.001725	0.002391	11	0.107887	0.120069	0.000304	0.000291
9	12	101.741996	0.986121	0.002464	0.002739	12	4.649563	0.500464	0.000714	0.000389





Findings:

- As expected, execution time for the brute force and ILP algorithm increases exponentially.
- Execution time for ILP algorithm grows with a smaller rate than Brute Force.
- For small n, Execution time for ILP is higher than brute force but is overtaken by brute force due to the difference in growth rate.
- The approximation algorithms are much more efficient than the exact algorithms and take somewhat similar execution time.

Performance Ratios

Executing the respective algorithms 20 times for each n.

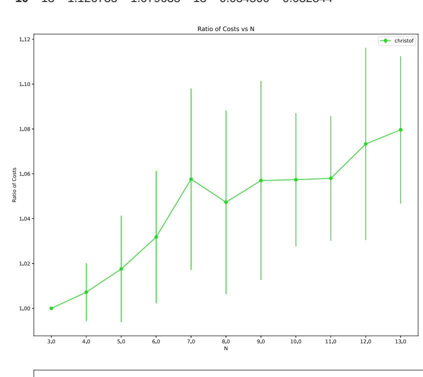
Implementation:

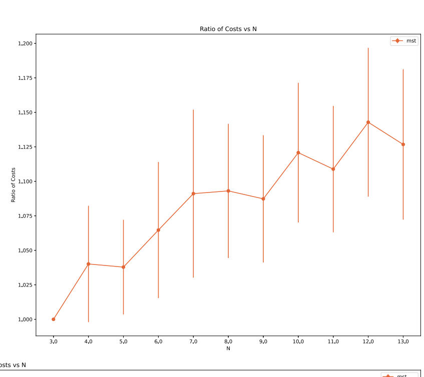
- Since there is a high standard deviation in both the graphs, it overlaps.
- We have displayed 3 separate graphs for costs, with and without standard deviations.

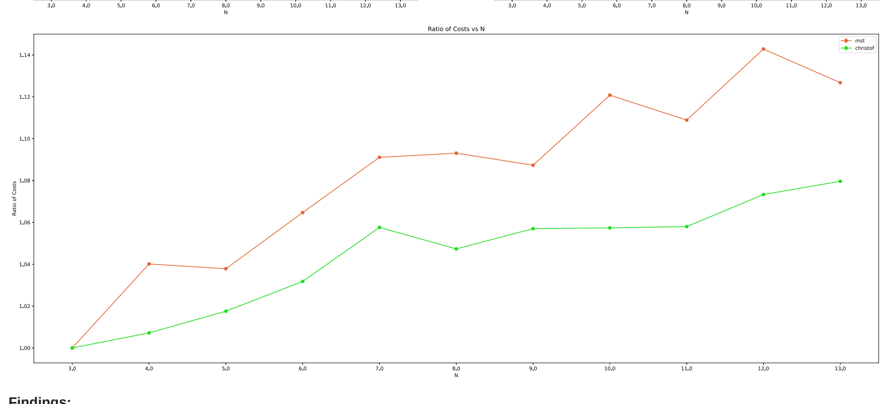
In [15]:
```
compare_algos(3,13,100,4100,200,lp = "ilp",COMPARE = "cost")
```

-------------------- Ratio of Costs vs N --------------------

	Average			Standard Deviation		
	N	mst	christof	N	mst	christof
0	3	1.000000	1.000000	3	0.000000	0.000000
1	4	1.040126	1.007202	4	0.042181	0.012897
2	5	1.037848	1.017558	5	0.034307	0.023717
3	6	1.064684	1.031792	6	0.049344	0.029502
4	7	1.091085	1.057607	7	0.060880	0.040445
5	8	1.093069	1.047332	8	0.048645	0.040911
6	9	1.087316	1.057002	9	0.046121	0.044409
7	10	1.120783	1.057381	10	0.050609	0.029701
8	11	1.108871	1.057987	11	0.045804	0.027859
9	12	1.142826	1.073308	12	0.053940	0.042891
10	13	1.126735	1.079635	13	0.054500	0.032844







Findings:

- There is a gradual increase in the performance ratio for both MST and Christofides algorithms.
- As expected, Christofides algorithm has a better (lower) performance ratio.