Lab_III

February 18, 2024

1 Computer Science 2XC3 - Graded Lab II

Please refer to the pdf for detailed instructions. The below file contains all the preliminary code you will need to work on the lab. You can copy paste instructions here to create one cohesive lab and organize it that best suits your teams workflow.

[ ]:
```
import random
import copy
import timeit
import matplotlib.pyplot as plt
import numpy as np
import math
from collections import deque
```

[ ]:
```
class Graph:
    def __init__(self, nodes):
        self.graph = [[] for _ in range(nodes)]
    def has_edge(self, src, dst):
        return src in self.graph[dst]
    def add_edge(self,src,dst):
        if not self.has_edge(src,dst):
            self.graph[src].append(dst)
            self.graph[dst].append(src)
    def get_graph(self):
        return self.graph

    # 1.3
    def has_cycle(self):
        # iterate over all nodes
        g = self.get_graph()
        for node in range(len(g)):
            cycle = self.__has_cycle_helper__(node, node, set())
            if cycle:
                return True
        return False
    def __has_cycle_helper__(self, n1:int, parent:int, visited:set):
        # init visited
        if len(visited) == 0:
```

```
            visited.add(n1)

        # get adjacency list
        g = self.get_graph()

        # dfs on all child nodes
        for node in g[n1]:
            # check if we've seen it before, aside from the parent
            if node == parent:
                continue
            if node in visited:
                return True

            visited.add(node)
            has_cycle = self.__has_cycle_helper__(node, n1, visited)
            if has_cycle:
                return True

        return False

    # 1.4
    def is_connected(self):
        g = self.get_graph()
        # only need to check the first half
        for n1 in range(len(g)//2 + 1):
            for n2 in range(len(g)):
                if n1 != n2:
                    connected = self.is_connected_two(n1, n2)
                    if not connected: return False
        return True

    def is_connected_two(self, n1:int, n2:int):
        connection = bfs2(self, n1, n2)
        return connection is not None and len(connection) > 0


# 1.1
def bfs2(graph: Graph, n1: int, n2: int):
    g = graph.get_graph()
    if n1 == n2 or len(g) <= n1:
        return []

    # have we seen this node?
    visited = set()
    visited.add(n1)
    q = deque()
    q.extend(g[n1])
```

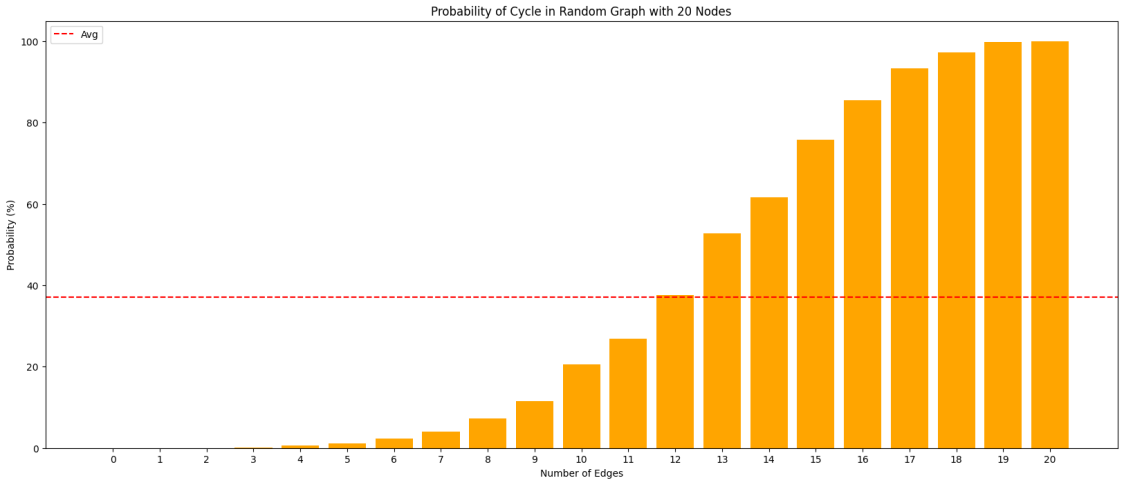
```
    found = False

    # construct a path for all nodes so we can backtrack to the first node
    # each key is a child and the value is the parent node
    path = {n1: n1}
    for first_children in g[n1]:
        path[first_children] = n1

    while len(q) > 0 and not found:
        # ensure we look at the oldest elements first
        n = q.popleft()

        # run checks
        if n == n2:
            found = True
        if n in visited:
            continue

        # save info of node
        visited.add(n)
        q.extend(g[n])
        for child in g[n]:
            if child not in visited:
                path[child] = n

    if not found:
        return None

    # generate our list path
    path_out = []
    n = n2
    while n != n1:
        path_out.append(n)
        n = path[n]
    path_out.append(n1)
    path_out.reverse()
    return path_out

def dfs2(graph: Graph, n1: int, n2: int, visited:set = set(), path:list = []):
    # handle case where client passes n1=n2
    if n1 == n2:
        return path
    if n1 in visited:
        return None
    # init the path; only run on depth=0
    if path == []:
        path = [n1]
```

```
    # get adjacency list and visit
    g = graph.get_graph()
    visited.add(n1)

    for node in g[n1]:
        # found node
        if node == n2:
            return path + [n2]

        if node not in visited:
            # dfs, if none then n2 was not found
            dfs_res = dfs2(graph, node, n2, visited, path + [node])
            if dfs_res is not None:
                return dfs_res

    # no path found
    return None

# 1.2
# n1 is the optional starting point
def bfs3(graph: Graph, n1:int = 0):
    g = graph.get_graph()
    if len(g) <= n1:
        return {}

    # have we seen this node?
    visited = set()
    visited.add(n1)
    q = deque()
    q.extend(g[n1])

    # construct a path for all nodes so we can backtrack to the first node
    # each key is a child and the value is the parent node
    path = {n1: n1}
    for first_children in g[n1]:
        path[first_children] = n1

    while len(q) > 0:
        # ensure we look at the oldest elements first
        n = q.popleft()

        # run check
        if n in visited:
            continue

        # save info of node
```

```
        visited.add(n)
        q.extend(g[n])
        for child in g[n]:
            if child not in visited:
                path[child] = n

    return path

def dfs3(graph: Graph, n1:int = 0, visited:dict = dict()):
    # init visited
    if len(visited) == 0:
        visited[n1] = n1

    # get adjacency list
    g = graph.get_graph()

    # dfs on all child nodes
    for node in g[n1]:
        # visit node and keep track of parent
        if node not in visited:
            visited[node] = n1
            dfs3(graph, node, visited)

    return visited

# 1.5
def create_random_graph(n, num_edges):
    g = Graph(n)

    # clean input, any more edges and we'd have repeats
    max_edges = (n*(n-1))/2
    adjusted_edges = int(min(num_edges, max_edges))

    for _ in range(adjusted_edges):
        while True:
            start = random.randint(0, n-1)
            end = random.randint(0, n-1)

            if start == end or g.has_edge(start, end):
                continue
            g.add_edge(start, end)
            break

    return g
```

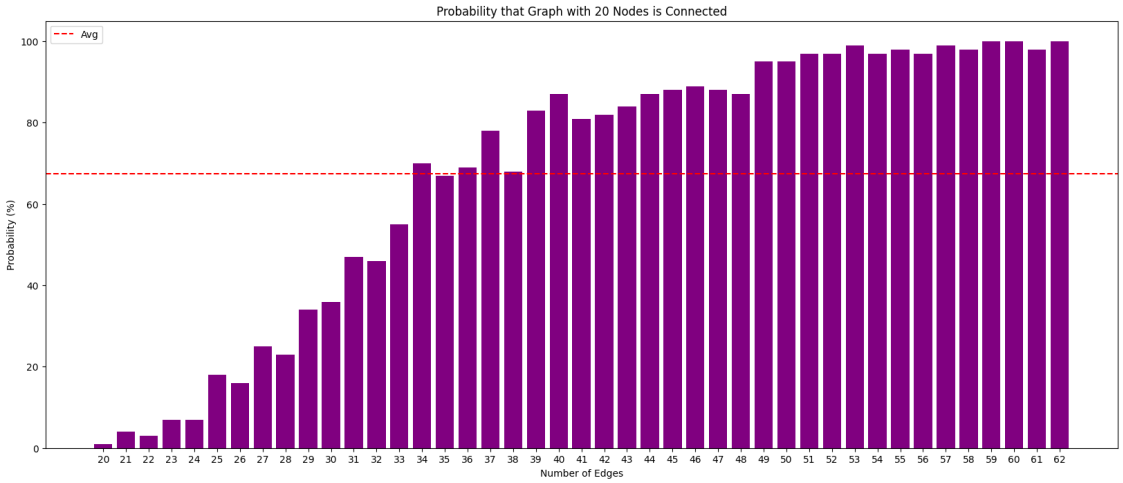
[ ]:
```
# 1.6
```

```python
nodes = 20
iterations_per_edge = 1000
probabilities = []

for i in range(0, nodes+1):
    total = 0
    # check x iterations if this graph has a cycle
    for _ in range(iterations_per_edge):
        g = create_random_graph(nodes, i)
        if g.has_cycle():
            total += 0.1
    probabilities += [round(total, 2)]

x_axis = np.arange(0, len(probabilities), 1)
plt.figure(figsize=(20, 8))
plt.bar(x_axis, probabilities, color='orange')
plt.axhline(np.mean(probabilities), color="red", linestyle="--", label="Avg")
plt.xticks(x_axis, x_axis)
plt.title("Probability of Cycle in Random Graph with " + str(nodes) + " Nodes")
plt.ylabel("Probability (%)")
plt.xlabel("Number of Edges")
plt.legend()
plt.show()
```



Reflection: We decided on using 20 nodes to compute this as it's a high enough number to see a variable change of edges, while not being too computationally expensive. For every node we run 1000 iterations to accurately compute the probability of that specific number of edges having a cycle. There is no need to check when the number of edges > number of nodes as we will be guaranteed to find a cycle. It seems that we break the 50% chance of having a cycle when we have about 66% amount of edges to nodes.

```
[ ]: # 1.7

nodes = 20
iterations_per_edge = 100
probabilities = []

for i in range(nodes-1, 190):
    total = 0
    # check x iterations if this number of edges is connected
    for _ in range(iterations_per_edge):
        g = create_random_graph(nodes, i)
        if g.is_connected():
            total += 1
    probabilities += [round(total, 2)]
    # if the last 5 runs have on average > 99%
    if (len(probabilities) > 10 and (sum(probabilities[-5:])/5 > 99)):
        break

x_axis = np.arange(nodes, len(probabilities) + nodes, 1)
plt.figure(figsize=(20, 8))
plt.bar(x_axis, probabilities, color='purple')
plt.axhline(np.mean(probabilities), color="red", linestyle="--", label="Avg")
plt.xticks(x_axis, x_axis)
plt.title("Probability that Graph with " + str(nodes) + " Nodes is Connected")
plt.ylabel("Probability (%)")
plt.xlabel("Number of Edges")
plt.legend()
plt.show()
```



Reflection: Once again we chose to run this experiment on 20 nodes. This time, we must start our edge iteration at the number of nodes, as you cannot have a connected graph when the number

of edges is less than the number of nodes - 1. Computing whether two vertices are connected is much more computationally expensive so we've lowered the iterations per edge from 1000 to 100. In our graph code we've also optimized the connected function to ensure we're not running unneeded checks. The chance of being connected starts off incredibly low and not until we have about 2-3x as many edges as nodes do we see a high chance of having a connected graph.

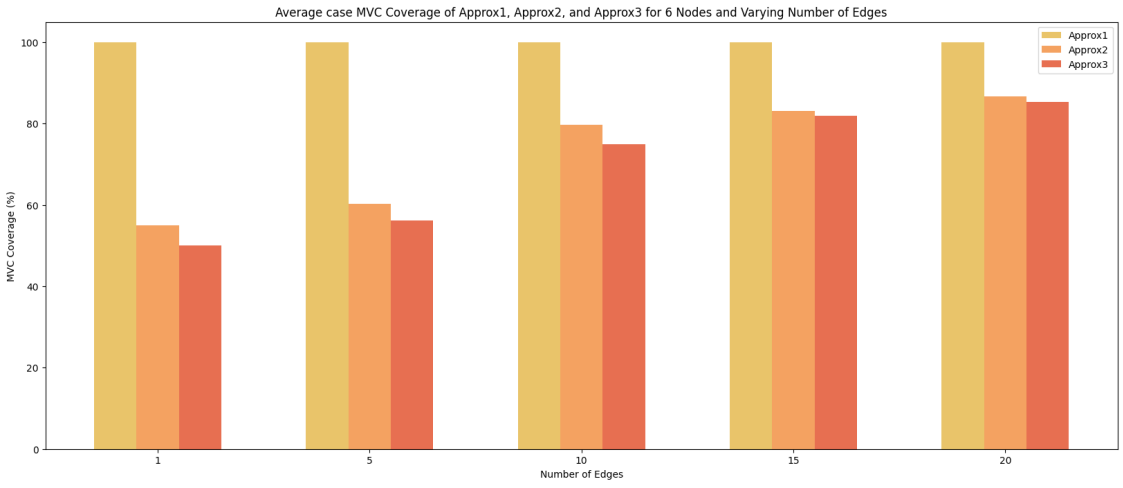
[ ]:
```
#Use the methods below to determine minimum vertex covers

def add_to_each(sets, element):
    copy = sets.copy()
    for set in copy:
        set.append(element)
    return copy

def power_set(set):
    if set == []:
        return [[]]
    return power_set(set[1:]) + add_to_each(power_set(set[1:]), set[0])

def is_vertex_cover(G : Graph, C):
    graph = G.get_graph()
    for start in range(0,len(graph)):
        for end in graph[start]:
            if not(start in C or end in C):
                return False
    return True

def MVC(G):
    nodes = [i for i in range(len(G.get_graph()))]
    subsets = power_set(nodes)
    min_cover = nodes
    for subset in subsets:
        if is_vertex_cover(G, subset):
            if len(subset) < len(min_cover):
                min_cover = subset
    return min_cover
```

[ ]:
```
def Approx1(G : Graph):
    C = set()
    graph = copy.deepcopy(G.get_graph())

    while True:

        # find and add vertex v of highest degree
        v = 0
        for i in range(0,len(graph)):
            if len(graph[i]) > len(graph[v]):
```

```
                v = i
        C.add(v)

        # remove all edges connected to vertex v
        graph[v] = []
        for j in range(0,len(graph)):
            if v in graph[j]:
                graph[j].remove(v)

        # check if vertex cover
        if is_vertex_cover(G, C):
            return C

def Approx2(G: Graph):
    C = set()
    graph = copy.deepcopy(G.get_graph())
    nodes = [i for i in range(len(graph))]

    while True:

        # select and add random node
        v = nodes[random.randrange(0,len(nodes))]
        nodes.remove(v)
        C.add(v)

        # check if vertex cover
        if is_vertex_cover(G, C):
            return C

def Approx3(G: Graph):
    C = set()
    graph = copy.deepcopy(G.get_graph())
    nodes = [i for i in range(len(graph))]

    while True:

        # check if vertex cover (we do this first to account for graphs with no␣
 ↪nodes)
        if is_vertex_cover(G, C):
            return C

        # manage list of selectable nodes
        i = 0
        while i < len(graph):
            if i in nodes and len(graph[i]) == 0:
                nodes.remove(i)
            else:
```

```
                i+=1

        # randomly select edge (u,v)
        v = nodes[random.randrange(0,len(nodes))]
        u = graph[v][random.randrange(0,len(graph[v]))]
        nodes.remove(v)
        nodes.remove(u)


        C.add(v)
        C.add(u)

        # remove all edges connected to v or u
        graph[v] = []
        graph[u] = []
        for j in range(0,len(graph)):
            if v in graph[j]:
                graph[j].remove(v)
            if u in graph[j]:
                graph[j].remove(u)


graphs_list = []
edge_sizes = [1,5,10,15,20]
a1_sums = []
a2_sums = []
a3_sums = []

for edge_sz in edge_sizes:
    for _ in range(0,50):
        graphs_list.append(create_random_graph(5, edge_sz))

    mvc_sum = 0

    for graph in graphs_list:
        size = len(MVC(graph))
        mvc_sum += size

    a1_sum = 0

    for graph in graphs_list:
        size = len(Approx1(graph))
        a1_sum += size

    a2_sum = 0

    for graph in graphs_list:
```

```
        size = len(Approx2(graph))
        a2_sum += size

    a3_sum = 0

    for graph in graphs_list:
        size = len(Approx3(graph))
        a3_sum += size

    a1_sums.append(mvc_sum/a1_sum * 100)
    a2_sums.append(mvc_sum/a2_sum * 100)
    a3_sums.append(mvc_sum/a3_sum * 100)


x_axis = np.arange(0, len(edge_sizes),1)
plt.figure(figsize=(20, 8))
plt.bar(x_axis-0.2, a1_sums, 0.2, color='#E9C46A', label='Approx1')
plt.bar(x_axis, a2_sums, 0.2, color='#F4A261', label='Approx2')
plt.bar(x_axis+0.2, a3_sums, 0.2, color='#E76F51', label='Approx3')
plt.xticks(x_axis, edge_sizes)
plt.title("Average case MVC Coverage of Approx1, Approx2, and Approx3 for 6␣
 ↪Nodes and Varying Number of Edges")
plt.ylabel("MVC Coverage (%)")
plt.xlabel("Number of Edges")
plt.legend()
plt.show()
```

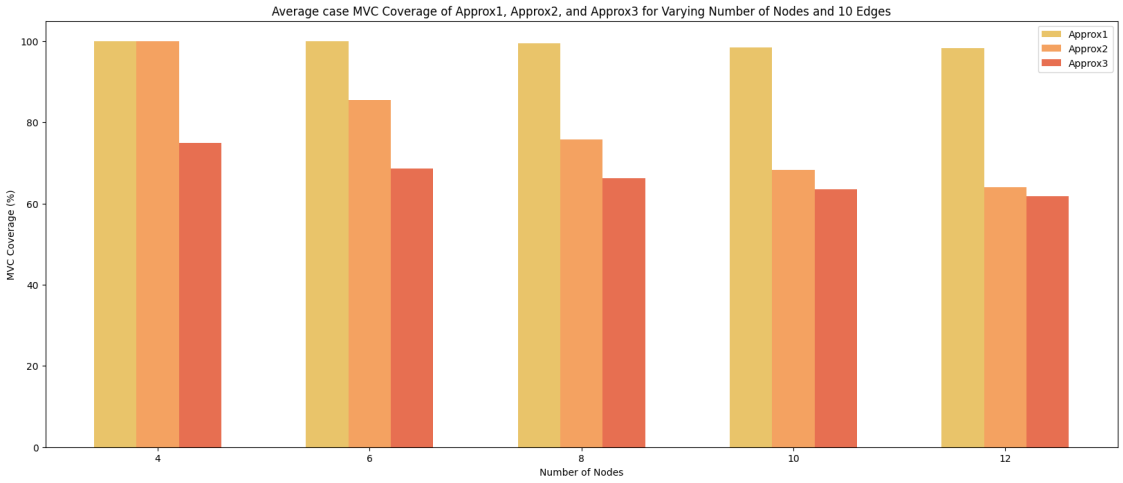


From the above graph, we can observe that Approx1 will always return a minimum vertex cover. As for the other functions, Approx2 is likely to randomly add unnecessary nodes (nodes with few or zero edges) to its vertex cover list, and Approx3 will always select 2 nodes connected to each other, which will often be redundant. We can see that the randomized functions improve their accuracy as

the number of edges in the graph increases, because the chance of them randomly selecting useful nodes (nodes with many edges) increases with more edges in the graph.

[ ]:
```
graphs_list = []
node_sizes = [4,6,8,10,12]
a1_sums = []
a2_sums = []
a3_sums = []

for node_sz in node_sizes:
    for _ in range(0,50):
        graphs_list.append(create_random_graph(node_sz, 10))

    mvc_sum = 0

    for graph in graphs_list:
        size = len(MVC(graph))
        mvc_sum += size

    a1_sum = 0

    for graph in graphs_list:
        size = len(Approx1(graph))
        a1_sum += size

    a2_sum = 0

    for graph in graphs_list:
        size = len(Approx2(graph))
        a2_sum += size

    a3_sum = 0

    for graph in graphs_list:
        size = len(Approx3(graph))
        a3_sum += size

    a1_sums.append(mvc_sum/a1_sum * 100)
    a2_sums.append(mvc_sum/a2_sum * 100)
    a3_sums.append(mvc_sum/a3_sum * 100)

x_axis = np.arange(0, len(node_sizes),1)
plt.figure(figsize=(20, 8))
plt.bar(x_axis-0.2, a1_sums, 0.2, color='#E9C46A', label='Approx1')
plt.bar(x_axis, a2_sums, 0.2, color='#F4A261', label='Approx2')
plt.bar(x_axis+0.2, a3_sums, 0.2, color='#E76F51', label='Approx3')
plt.xticks(x_axis, node_sizes)
```

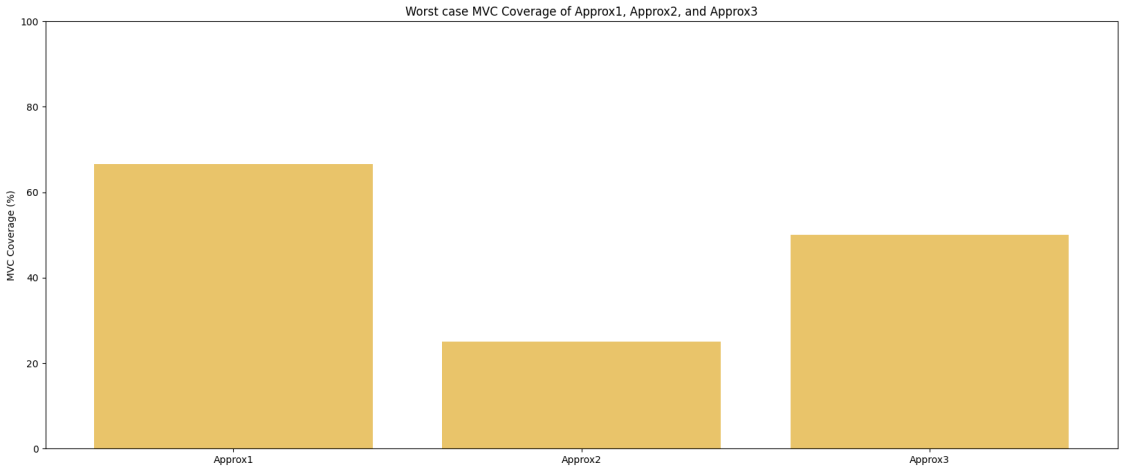
```
plt.title("Average case MVC Coverage of Approx1, Approx2, and Approx3 for␣
 ↪Varying Number of Nodes and 10 Edges")
plt.ylabel("MVC Coverage (%)")
plt.xlabel("Number of Nodes")
plt.legend()
plt.show()
```



Interestingly, for small enough graph sizes, the MVC is so large that Approx2 is guaranteed to be 100% accurate. Approx3, on the other hand, probably becomes guaranteed to add redundant nodes. For larger numbers of nodes, both Approx3 seems to keeps its accuracy static, while Approx2 notices a decrease in accuracy. This is likely because with more nodes but the same number of edges, the Approx2 function has more chances to "guess wrong" (to randomly select a node with no edges).

[ ]:
```
from itertools import combinations

def generate_all_graphs(n):
    # Generate all possible edges for a graph with n nodes
    all_edges = list(combinations(range(n), 2))

    # Generate all combinations of edges
    all_graphs = []
    for i in range(2 ** len(all_edges)):
        graph = Graph(n)
        for j in range(len(all_edges)):
            if i & (1 << j):
                # graph.append(all_edges[j])
                graph.add_edge(all_edges[j][0],all_edges[j][1])
        all_graphs.append(graph)

    return all_graphs
```

```
# Generate all graphs with 5 nodes
all_5_node_graphs = generate_all_graphs(5)

min1 = 1
min_graph = Graph(5)
for graph in all_5_node_graphs:
    mvc = MVC(graph)
    approx = Approx1(graph)
    if len(mvc) > 0 and len(mvc) / len(approx) < min1:
        min1 = len(mvc) / len(approx)
        min_graph = graph

min2 = 1
min_graph = Graph(5)
for graph in all_5_node_graphs:
    mvc = MVC(graph)
    approx = Approx2(graph)
    if len(mvc) > 0 and len(mvc) / len(approx) < min2:
        min2 = len(mvc) / len(approx)
        min_graph = graph

min3 = 1
min_graph = Graph(5)
for graph in all_5_node_graphs:
    mvc = MVC(graph)
    try:
        approx = Approx3(graph)
    except:
        print("approx3 error")
        print(graph.get_graph())
    if len(mvc) > 0 and len(mvc) / len(approx) < min3:
        min3 = len(mvc) / len(approx)
        min_graph = graph

x_axis = np.arange(0, 3, 1)
plt.figure(figsize=(20, 8))
plt.bar(x_axis, [min1*100, min2*100, min3*100], color='#E9C46A')
plt.xticks(x_axis, ["Approx1", "Approx2", "Approx3"])
plt.title("Worst case MVC Coverage of Approx1, Approx2, and Approx3")
plt.ylabel("MVC Coverage (%)")
plt.ylim(0, 100)
plt.show()
```



Contrary to our earlier observations, it seems Approx1 does not always return the minimum vertex cover, having a worst case of 75% accuracy. Approx2 has the poorest worst case, which makes sense for a completely random algorithm.

[ ]:
```
def is_indep_set(G : Graph, S):
    graph = G.get_graph()
    for u in S:
        for v in S:
            if v in graph[u]:
                return False
    return True

def MIS(G : Graph):
    nodes = [i for i in range(len(G.get_graph()))]
    subsets = power_set(nodes)
    max_set = []

    for subset in subsets:
        if is_indep_set(G, subset):
            if len(subset) > len(max_set):
                max_set = subset
    return max_set
```

[ ]:
```
size_list = []
for _ in range(0,20):
    newGraph = create_random_graph(6,5)
    size_list.append(len(MIS(newGraph)) + len(MVC(newGraph)))

print(size_list)
```

```
size_list = []
for _ in range(0,20):
    newGraph = create_random_graph(7,5)
    size_list.append(len(MIS(newGraph)) + len(MVC(newGraph)))

print(size_list)
```

```
[6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6]
[7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7]
```

From the above experiment, we can see that the size of a MIS of a graph + the size of a MVC of the same graph is always equal to the number of nodes in the graph.

[ ]:
```
bool_list = []
for _ in range(0,20):
    newGraph = create_random_graph(6,5)

    mvc_set = set(MVC(newGraph))
    nodes_set = {i for i in range(0,6)}

    new_mis = nodes_set.difference(mvc_set)
    bool_list.append(is_indep_set(newGraph, new_mis) and len(new_mis) ==␣
 ↪len(MIS(newGraph)))

print(bool_list)

bool_list = []
for _ in range(0,20):
    newGraph = create_random_graph(6,5)

    mis_set = set(MIS(newGraph))
    nodes_set = {i for i in range(0,6)}

    new_mvc = nodes_set.difference(mis_set)
    bool_list.append(is_vertex_cover(newGraph, new_mvc) and len(new_mvc) ==␣
 ↪len(MVC(newGraph)))

print(bool_list)
```

```
[True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True]
[True, True, True, True, True, True, True, True, True, True, True, True,
True, True, True, True, True, True, True, True]
```

From the above experiment, we can see that for any MVC of a graph G, the set of all nodes of G which are NOT in the MVC make up a MIS of G. The same can be said for the set of all nodes not in an MIS of G making up a MVC of G.

As there are typically several possible MVCs and MISs for a single graph, it follows that any MVC

+ any MIS do not always form the set of all nodes in G.

Any disconnected node (a node with no edges) will always be in the MIS of a graph. From our earlier observations, it follows that a disconnected node will never be in the MVC of a graph.