CS 170 HW 12

Due **2021-4-27, at 10:00 pm**

1 Study Group

List the names and SIDs of the members in your study group. If you have no collaborators, write "none".

2 Local Search for Max Cut

Sometimes, local search algorithms can give good approximations to NP-hard problems. In the Max-Cut problem, we have a graph $G(V, E)$ and we want to find a cut (S, T) with as many edges crossing as possible. One local search algorithm is as follows: Start with any cut, and while there is some vertex $v \in S$ such that more edges cross $(S - v, T + v)$ (or some $v \in T$ such that more edges cross $(S + v, T - v)$), move v to the other side of the cut. Note that when we move v from S to T, v must have more neighbors in S than T.

(a) Give an upper bound on the number of iterations this algorithm can run for (i.e. the total number of times we move a vertex).

(b) Show that when this algorithm terminates, it finds a cut where at least half the edges in the graph cross the cut.

Solution:

(a) $|E|$ iterations. Each iteration increases the number of edges crossing the cut by at least 1. The number of edges crossing the cut is between 0 and $|E|$, so there must be at most $|E|$ iterations.

(b) $\delta_{in}(v)$ be the number of edges from v to other vertices on the same side of the cut, and $\delta_{out}(v)$ be the number of edges from v to vertices on the opposite side of the cut. The total number of edges crossing the cut the algorithm finds is $\frac{1}{2}\sum_{v\in V}\delta_{out}(v)$, and the total number of edges in the graph is $\frac{1}{2}\sum_{v\in V}(\delta_{in}(v)+\delta_{out}(v))$. We know that $\delta_{out}(v) \geq \delta_{in}(v)$ for all vertices when the algorithm terminates (otherwise, the algorithm would move v across the cut), so the former is at least half as large as the latter.

3 Coffee Shops

A rectangular city is divided into a grid of $m \times n$ blocks. You would like to set up coffee shops so that for every block in the city, either there is a coffee shop within the block or there is one in a neighboring block. (There are up to 4 neighboring blocks for every block). It costs r_{ij} to rent space for a coffee shop in block ij.

Write an integer linear program to determine which blocks to set up the coffee shops at, so as to minimize the total rental costs.

(a) What are your variables, and what do they mean?

(b) What is the objective function?

(c) What are the constraints?

(d) Solving the linear program gets you a real-valued solution. How would you round the LP solution to obtain an integer solution to the problem? Describe the algorithm in at most two sentences.

(e) What is the approximation ratio obtained by your algorithm?

(f) Briefly justify the approximation ratio.

Solution:

(a) There is a variable for every block x_{ij}, i.e., $\{x_{ij} | i \in \{1, \ldots, m\}, j \in \{1, \ldots, n\}\}$. This variable corresponds to whether we put a coffee shop at that block or not.

(b) $\min \sum_{i=1}^{m} \sum_{j=1}^{n} r_{ij} x_{ij}$. Alternatively, $\min t$ is correct as well as long as the correct constraint is added.

(c) (i) $x_{ij} \geq 0$: This constraint just corresponds to saying that there either is or isn't a coffee shop at any block. $x_{ij} \in \{0, 1\}$ or $x_{ij} \in \mathbb{Z}_+$ is also correct.

(ii) For every $1 \leq i \leq m, 1 \leq j \leq n$:

$$x_{ij} + x_{(i+1),j} \mathbb{1}_{\{i+1 \leq m\}} + x_{(i-1),j} \mathbb{1}_{\{i-1 \geq 1\}} + x_{i,(j+1)} \mathbb{1}_{\{j+1 \leq n\}} + x_{i,(j-1)} \mathbb{1}_{\{j-1 \geq 1\}} \geq 1$$

This constraint corresponds to that for every block, there needs to be a coffee shop at that block or a neighboring block.

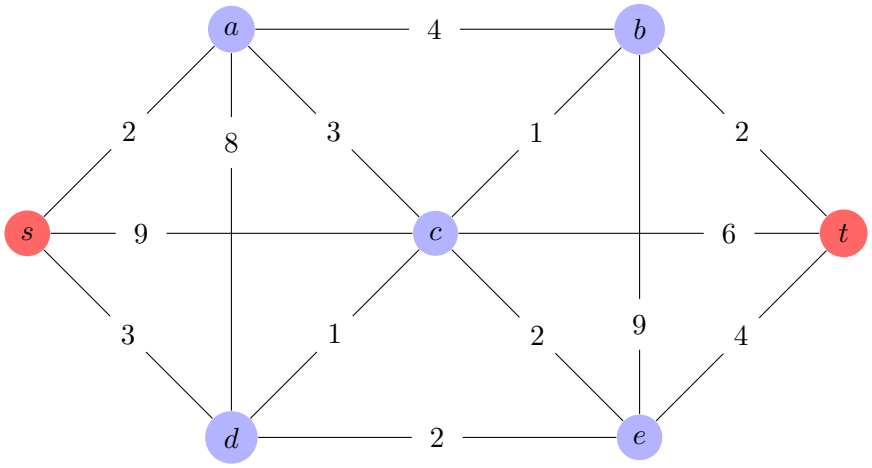
$\mathbb{1}_{\{i+1 \leq m\}}$ means "1 if $\{i+1 \leq m\}$, and 0 otherwise". It keeps track of the fact that we may not have all 4 neighbors on the edges, for instance.

(iii) If the objective was $\min t$, then the constraint $\sum_{i=1}^{m} \sum_{j=1}^{n} r_{ij} x_{ij} \leq t$ needs to be added.

(d) Round to 1 all variables which are greater than or equal to 1/5. Otherwise, round to 0. In other words, put a coffee shop on (i, j) iff $x_{i,j} \geq 0.2$.

(e) Using the rounding scheme in the previous part gives a 5-approximation.

(f) Notice that every constraint has at most 5 variables. So for every constraint, there exists at least one variable in the constraint which has value $\geq 1/5$ (not everyone is below average). The number of coffee shops in the rounded solution is at most $5 \cdot$ LP-OPT, (the number of $x_{i,j} \geq 0.2$ is at most $5 \cdot \sum_{i,j} x_{i,j}$). Since Integral-OPT $\geq$ LP-OPT (the LP is more general than the ILP), our rounding gives value at most 5 LP-OPT ≤ 5 Integral-OPT. So we get a 5-approximation.



4 Mechanical Project Problem

In this problem, you will look at a few mechanical examples of the project problem, to build intuition. Consider the graph $G = (V, E)$ below.

Your answers to the following *do not* require any justification.

(a) What is the shortest path from s to t in G and what is its length? **Solution:** The shortest path is $s \rightsquigarrow d \rightsquigarrow c \rightsquigarrow b \rightsquigarrow t$, which has a length of 7.

(b) Which edge should be removed to maximize the length of the shortest path if you are allowed to remove $k = 1$ edge? What is the new shortest path length from s to t? **Solution:** Removing the edge $b \rightsquigarrow t$ would leave the graph with a shortest path length of 9.

(c) Which edges should be removed to maximize the length of the shortest path if you are allowed to remove $k = 2$ edges? What is the new shortest path length from s to t? **Solution:** Removing the edges $s \rightsquigarrow a$ and $s \rightsquigarrow d$ would leave the graph with a shortest path length of 12.

(d) Suppose that instead of being allowed to remove individual edges, you are allowed to remove exactly one node (not s or t) from the graph. Which node should be removed to maximize the length of the shortest path? What is the new shortest path length from s to t? **Solution:** Removing node b would leave the graph with a shortest path length of 9.

5 Reduction Coding (Extra Credit)

This semester, we are trying something new: questions which involve coding.

This link will take you to a python notebook, hosted on the Berkeley datahub, in which you will implement a reduction from the 3-coloring problem to 3SAT, and then use the provided SAT solver, along with your reduction, to solve 3-coloring problems. Once you have finished, download a PDF of your completed notebook via File → Download as → PDF via

HTML, and append the downloaded pdf to the rest of your homework submission. Be careful when selecting pages on gradescope.

Note: Datahub does not guarantee 100% reliability when you save your notebook, and recommends downloading a local copy occasionally to backup progress (via File → Download as → Notebook (.ipynb)).

REDUCTIONS IN NP COMPLETE PROBLEMS: 3-COL to 3-SAT

A Graph G is 3-colorable if you can color all of its vertices using no more than 3 colors such that no two adjacent vertices that are connected by an edge are the same color.

Given a Graph G with Nodes V and Edges E and access to a solver class for 3-SAT, use a reduction from 3-COL to 3-SAT to determine whether G is 3-colorable. If G is 3-colorable, output a valid assignment of colors for the vertices of G. Otherwise state that G is not colorable.

Note that the Graph and SAT classes have already been fully implemented. Your task is to implement the reduction function COL_to_SAT. More details about skeleton code and your task can be found in the respective docstrings.

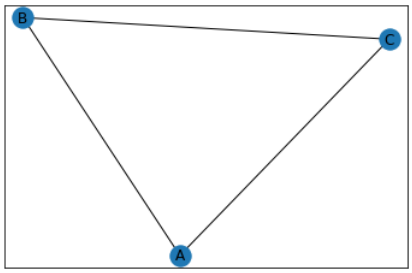
In [1]:
```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
import itertools
import random
```

GRAPH CLASS AND EXAMPLES

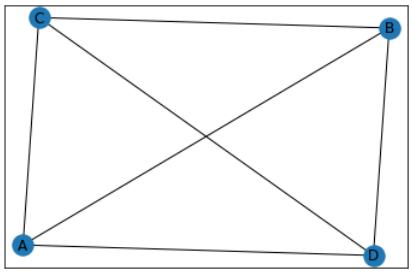
In [2]:
```
class Graph(object):

    def __init__(self, V = [], E = []):
        """
        Initializes a graph object G with vertices from set V and edges from list of tuples E

        Vertices should be represented as strings of alphanumeric characters only

        Sample usage:
        >>> small_graph = Graph(['A', 'B', 'C'], [('A', 'B'), ('A', 'C'), ('B', 'C')])
        """
        assert all([v.isalnum() for v in V])
        self.G = nx.Graph()
        self.G.add_nodes_from(V)
        self.G.add_edges_from(E)

    def vertices(self):
        """
        Returns a set containing all vertices of G

        Sample usage:
        >>> small_graph.vertices()
        {'A', 'B', 'C'}
        """
        return set(self.G.nodes())

    def edges(self, vertex = None):
        """
        Returns a list of tuples containing all edges that are incident to vertex of G

        If vertex is set to None the function simply returns all the edges of G

        Sample usage:
        >>> small_graph.edges()
        [('A', 'B'), ('A', 'C'), ('B', 'C')]
        >>> small_graph.edges('A')
        [('A', 'B'), ('A', 'C')]
        """
        return list(self.G.edges(vertex))

    def add_vertices(self, vertices):
        """
        Adds every new vertex in vertices to G

        Vertices should be represented as strings of alphanumeric characters only

        Sample usage:
        >>> small_graph.vertices()
        {'A', 'B', 'C'}
        >>> small_graph.add_vertex({'D'})
        >>> small_graph.vertices()
        {'A', 'B', 'C', 'D'}
        >>> small_graph.add_vertex({'E', 'F'})
        >>> small_graph.vertices()
        {'A', 'B', 'C', 'D', 'E', 'F'}
        """
        assert all([v.isalnum() for v in vertices])
        self.G.add_nodes_from(vertices)

    def add_edges(self, edges):
        """
        Adds an edge for every tuple in the edges list to G

        Sample usage:
        >>> small_graph.edges()
        [('A', 'B'), ('A', 'C'), ('B', 'C')]
        >>> small_graph.add_edges([('A', 'D')])
        >>> small_graph.edges()
        [('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C')]
        >>> small_graph.add_edges([('B', 'D'), ('C', 'D')])
        >>> small_graph.edges()
        [('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D')]
        """
        self.G.add_edges_from(edges)

    def visualize(self):
        """
```

```
        Displays a visual representation of the Graph

        Sample usage:
        >>> small_graph.visualize()
        < cool image appears here >
        """
        nx.draw_networkx(self.G)
        plt.show()
```

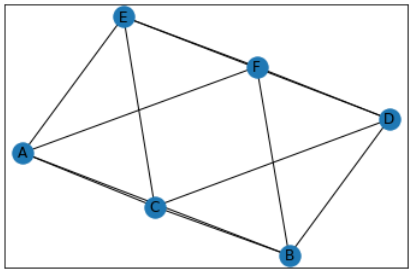
In [3]:
```
G_small1 = Graph(["A", "B", "C"], [("A", "B"), ("A", "C"), ("B", "C")])
G_small1.visualize()
```



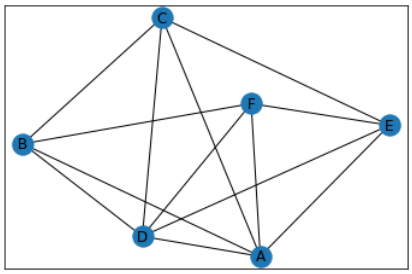
In [4]:
```
G_small2 = Graph(["A", "B", "C"], [("A", "B"), ("A", "C"), ("B", "C")])
G_small2.add_vertices({'D'})
G_small2.add_edges([("A", "D"), ("B", "D"), ("C", "D")])
G_small2.visualize()
```



In [5]:
```
G_medium1 = Graph(["A", "B", "C", "D", "E", "F"],
                  [("A", "B"), ("A", "C"), ("A", "E"), ("A", "F"),
                   ("B", "C"), ("B", "D"), ("B", "F"),
                   ("C", "D"), ("C", "E"),
                   ("D", "E"), ("D", "F"),
                   ("E", "F")])
G_medium1.visualize()
```



In [6]:
```
G_medium2 = Graph(["A", "B", "C", "D", "E", "F"],
                  [("A", "B"), ("A", "C"), ("A", "D"), ("A", "E"), ("A", "F"),
                   ("B", "C"), ("B", "D"), ("B", "F"),
                   ("C", "D"), ("C", "E"),
                   ("D", "E"), ("D", "F"),
                   ("E", "F")])
G_medium2.visualize()
```



SAT CLASS AND EXAMPLES:

In [7]:

```
class SAT(object):

    def __init__(self):
        """
        Initializes a SAT instance to solve 3SAT

        Sample usage:
        >>> sat = SAT()
        """
        self.cnf = []
        self.make_false()

    def make_false(self):
        """
        Adds trivial clauses to SAT instance to force a 'false' variable
        with value False which can be used as a trivial 3rd variable in
        clauses with only 2 real variables so that the 3 variables per
        clause structure of 3SAT is maintained
        YOU SHOULD NOT CALL THIS FUNCTION DIRECTLY
        """
        self.add_clauses([('ignore1', 'ignore2', '~false'),
                          ('ignore1', '~ignore2', '~false'),
                          ('~ignore1', 'ignore2', '~false'),
                          ('~ignore1', '~ignore2', '~false')])

    def add_clause(self, variable_1, variable_2, variable_3 = 'false'):
        """
        Adds a 3SAT clause with variable_1, variable_2, and variable_3 to the
        cnf of the SAT object where each variable is represented as a string

        If only variable_1 and variable_2 are passed in, a trivial false variable
        is used as the third variable so that the 3 variable structure of a 3SAT
        clause is maintained

        Variables can be named anything except 'false', 'ignore1', and 'ignore2'
        as these are provisional variables maintained internally by the SAT instance

        Calling sat.add_clause('x', 'y', 'z') is equivalent to adding the clause
        (x or y or z) to the SAT cnf formula

        To add the complement of any variable to a clause, simply prefix it with
        a '~' and then the variable name. ex: sat.add_clause('~x', 'y', '~z') is
        equivalent to adding the clause (not x or y or not z) to the SAT cnf formula


        You should not use the following three variable names in your SAT clauses:
        'false', 'ignore1', 'ignore2' because these are provisional variables
        defined and maintained by the

        Sample usage:
        >>> sat = SAT()
        >>> sat.add_clause('a', 'b', 'c')   # cnf = (a or b or c)
        >>> sat.add_clause('~a', '~d', 'e') # cnf = (a or b or c) and (not a or not d or e)
        >>> sat.add_clause('b', 'd')        # cnf = (a or b or c) and (not a or not d or e) and (b or d or false)
        """
        clause = set()
        for term in [variable_1, variable_2, variable_3]:
            if term[0] == '~':
                clause.add((term[1:], False))
            else:
                clause.add((term, True))
        return self.cnf.append(clause)

    def add_clauses(self, expressions):
        """
        Adds multiple clauses to the 3SAT cnf formula by adding a clause for
        every tuple expression in the list expressions

        Sample usage:
        >>> sat = SAT()
        >>> sat.add_clauses([('a', 'b', 'c'), ('~a', '~d', 'e'), ('b', 'd')])
        # cnf = (a or b or c) and (not a or not d or e) and (b or d or false)
        """
        for expression in expressions:
            assert len(expression) == 2 or len(expression) == 3
            if len(expression) == 2:
                self.add_clause(expression[0], expression[1])
            else:
                self.add_clause(expression[0], expression[1], expression[2])

    def solve(self, naive = False):
        """
        Solves the SAT instance using a back-tracking based DPLL algorithm and
        returns whether the SAT instance is satisfiable along with a dictionary
        with valid assignments for variables of the SAT instance if it is satisfiable

        Sample usage:
        >>> sat1 = SAT()
        >>> sat1.add_clause('a', 'b')   # cnf = (a or b or false)
        >>> sat1.add_clause('~a', '~b') # cnf = (a or b or false) and (not a or not b)
        >>> satisfiable, assignments = sat1.solve()
        >>> satisfiable
        True
        >>> assignments
        {'b': True, 'a': False}
        >>> sat2 = SAT()
        >>> sat2.add_clauses([('a', 'b'), ('a', '~b'), ('~a', 'b'), ('~a', '~b')])
        # cnf is now (a or b or false) and (a or not b or false) and (not a or b or false) and (not a or not b or false)
        >>> satisfiable, assignments = sat2.solve()
        >>> satisfiable
        False
        >>> assignments
```

```
        None
        """
        if naive:
            satisfiable, assignments = self.brute_force_3SAT_Solver(self.cnf)
        else:
            satisfiable, assignments = self.dpll_3SAT_Solver(self.cnf)
        if satisfiable:
            assignments.pop('false', None)
            assignments.pop('ignore1', None)
            assignments.pop('ignore2', None)
        return satisfiable, assignments

    def dpll_3SAT_Solver(self, cnf, assignments={}):
        """
        Internally used DPLL algorithm used to solve the 3SAT instance
        YOU SHOULD NOT CALL THIS FUNCTION DIRECTLY
        Instead use: sat.solve()

        This function is adapted from https://gist.github.com/davefernig/e670bda722d558817f2ba0e90ebce66f
        """
        def select_literal(cnf):
            for c in cnf:
                for literal in c:
                    return literal[0]
        if len(cnf) == 0:
            return True, assignments
        if any([len(c)==0 for c in cnf]):
            return False, None
        l = select_literal(cnf)
        new_cnf = [c for c in cnf if (l, True) not in c]
        new_cnf = [c.difference({(l, False)}) for c in new_cnf]
        sat, vals = self.dpll_3SAT_Solver(new_cnf, {**assignments, **{l: True}})
        if sat:
            return sat, vals
        new_cnf = [c for c in cnf if (l, False) not in c]
        new_cnf = [c.difference({(l, True)}) for c in new_cnf]
        sat, vals = self.dpll_3SAT_Solver(new_cnf, {**assignments, **{l: False}})
        if sat:
            return sat, vals
        return False, None

    def brute_force_3SAT_Solver(self, cnf):
        """
        Provisional brute force algorithm used to solve the 3SAT instance
        YOU SHOULD NOT CALL THIS FUNCTION DIRECTLY
        Instead use sat.solve(naive = False)

        This function is adapted from https://gist.github.com/davefernig/e670bda722d558817f2ba0e90ebce66f
        """
        literals = set()
        for conj in cnf:
            for disj in conj:
                literals.add(disj[0])
        literals = list(literals)
        n = len(literals)
        for seq in itertools.product([True,False], repeat=n):
            a = set(zip(literals, seq))
            if all([bool(disj.intersection(a)) for disj in cnf]):
                assignments = {item[0]: item[1] for item in a}
                return True, assignments
        return False, None
```

In [8]:

```
# Feel free to play around with a few different clauses and
# SAT instances here before proceeding with the reduction

sat = SAT()

sat.add_clause('a', 'b')
sat.add_clause('~a', '~b')
# cnf = (a or b or false) and (not a or not b or false)

# Add some more clauses to play around with here

sat.solve()
```

Out[8]: (True, {'a': True, 'b': False})

YOUR SOLVER HERE:

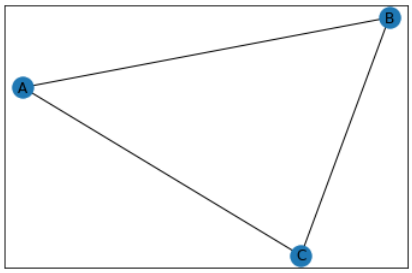
In [9]:

```
def COL_to_SAT(G, naive = False):
    """
    Given a graph G and access to the SAT class, use a reduction from
    3COL to 3SAT to determine whether G is 3 colorable or not. If it is
    satisfiable, output a valid assignment of colors for every vertex
    in the graph in a dictionary. You can choose any 3 distinct colors
    to mark the vertices. It doesn't matter which vertex is colored which
    color as long as neighboring vertices are marked with differently.

    If naive is set to true, use the naive solver for the SAT instance

    Expected behavior:
    >>> G_small1 = Graph(['A', 'B', 'C'], [('A', 'B'), ('A', 'C'), ('B', 'C')])
    >>> satisfiable, assignments = COL_to_SAT(G_small1)
    >>> satisfiable
    True
    >>> assignments
    {'A': '_red', 'B': '_green', 'C': '_blue'}
    >>> G_small2 = Graph(['A', 'B', 'C', 'D'], [('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D')])
```

```
    >>> satisfiable, assignments = COL_to_SAT(G_small2)
    >>> satisfiable
    False
    >>> assignments
    None
    """
    color1 = "_red"
    color2 = "_green"
    color3 = "_blue"
    sat = SAT()
    for v in G.vertices():
        vc1 = v + color1 # boolean representing if vertex v is color1
        vc2 = v + color2 # boolean representing if vertex v is color2
        vc3 = v + color3 # boolean representing if vertex v is color3
        # below line ensures that vertex v is at least one color
        sat.add_clause(vc1, vc2, vc3)
        # below line ensures that vertex v is at most one color
        sat.add_clauses([('~' + vc1, '~' + vc2), ('~' + vc1, '~' + vc3), ('~' + vc2, '~' + vc3)])
    for edge in G.edges():
        u = edge[0]
        v = edge[1]
        uc1, vc1 = u + color1, v + color1
        uc2, vc2 = u + color2, v + color2
        uc3, vc3 = u + color3, v + color3
        # below line ensures that no two adjacent vertices are the same color
        sat.add_clauses([('~' + uc1, '~' + vc1), ('~' + uc2, '~' + vc2), ('~' + uc3, '~' + vc3)])
    satisfiable, assignments = sat.solve(naive = naive)
    if satisfiable:
        assignments = {key.split('_')[0]: key.split('_')[1] for key, value in assignments.items() if value}
    return satisfiable, assignments
```

TESTING:

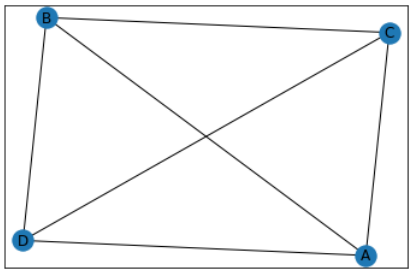
In [10]:
```
%%time
G = G_small1
G.visualize()
COL_to_SAT(G)
```




```
Wall time: 330 ms
```

Out[10]: (True, {'B': 'green', 'A': 'blue', 'C': 'red'})

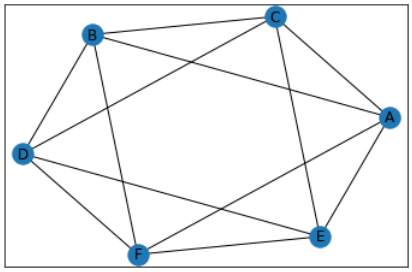
In [11]:
```
%%time
G = G_small2
G.visualize()
COL_to_SAT(G)
```




```
Wall time: 317 ms
```

Out[11]: (False, None)

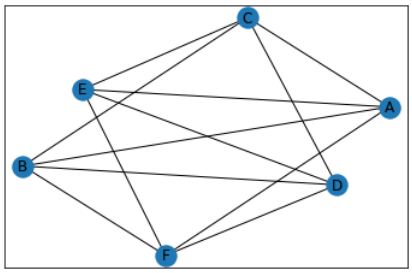
In [12]:
```
%%time
G = G_medium1
G.visualize()
COL_to_SAT(G)
```




```
Wall time: 321 ms
```

Out[12]:
```
(True,
 {'B': 'green',
  'A': 'blue',
  'D': 'blue',
  'F': 'red',
  'E': 'green',
  'C': 'red'})
```

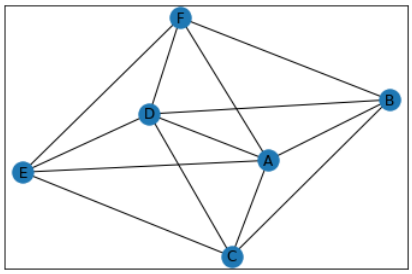
In [13]:
```
%%time
G = G_medium1
G.visualize()
COL_to_SAT(G, naive = True)
```




```
Wall time: 20.8 s
```

Out[13]:
```
(True,
 {'C': 'blue',
  'F': 'blue',
  'E': 'green',
  'B': 'green',
  'D': 'red',
  'A': 'red'})
```

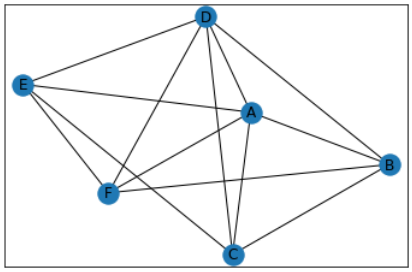
In [14]:
```
%%time
G = G_medium2
G.visualize()
COL_to_SAT(G)
```




```
Wall time: 263 ms
```

Out[14]:
```
(False, None)
```

In [15]:
```
%%time
G = G_medium2
G.visualize()
COL_to_SAT(G, naive = True)
```




```
Wall time: 1min 2s
```

Out[15]:
```
(False, None)
```

TAKEAWAY:

You might wonder why we would want to reduce one NP-Complete problem to another NP-Complete problem since they would both end up taking exponential time to solve in the worst case anyways.

However, some NP-Complete problems such as 3SAT are very well studied. As a result, researchers develop problem-specific strategies over time to find solutions for these problems relatively faster. While the worst case time complexity is still exponential, you probably noticed that in general the brute force solver for 3SAT (which is called when naive is set to True) takes much longer to reach a solution compared to the DPLL solver for 3SAT which is used by our SAT solver by default. Reductions between NP-Complete problems try to take advantage of exactly this!

If we wanted to solve a 3COL problem (which is relatively less well studied) more quickly on average, we would have to spend a lot of time researching the problem to come up with appropriate strategies that would make our search for valid color assignments quicker. However, using a reduction from 3COL to 3SAT, we can directly take advantage of the many sophisticated solvers that already exist for 3SAT and use those to our advantage to get a relatively quicker solver for 3COL.

Kinda cool, right?