UNSW SYDNEY

COMP6741
Algorithms for Intractable Problems
Educational Material for
Branching Algorithms
2024 T1

Tianrui Wang z5407459

Yuedong Li z5158488

Sixiang Qiu z5386034

1. Work split and Contributions

1.1 Preliminary phase

After our team was formed, we immediately started working on the assignment. Rather than internet conversation, we have our first meeting in person. Following a vigorous discussion, we unanimously decided to select the "**Alternative 2 – educational**" as our assignment 3 option. We take one day to review the topics in courses respectively and explore which topics would be most appropriate to address in this assignment. During our second meeting, Tianrui suggested choosing the topic of branching algorithms because this topic was explained in class using three different problems, which perfectly suited us to divide and conquer the task. And finally, we choose the topic: **Branching Algorithm.**

1.2 Work split and collaboration

As we form a 3-member team, we split the work into three parts: vertex cover, feedback vertex set and maximum leaf spanning tree. Each of us tries to design the visualization of our own part.

In the process of working, we divided the whole assignment into several phases: initial coding, peer review and code optimization, code integration, writing individual sections of the report, and finally, integrating the report. At the end of each phase, we held a meeting to discuss and summarize our current phase, and to set deadlines for the next phase. Thanks to a well-considered schedule and phase planning, our project was executed smoothly.

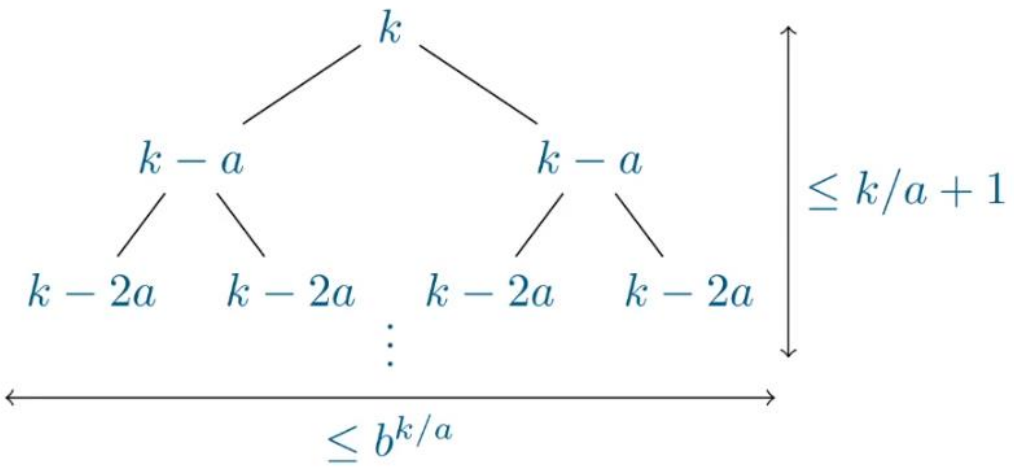
1.3 Contributions of each team member

Tianrui Wang(z5407459): in charge of maximum leaf spanning tree problem, test design, report writing and integrating.
Yuedong Li(z5158488): in charge of vertex cover problem, test design, report writing and code integrating.
Sixiang Qiu(z5386034): in charge of feedback vertex set problem, test design, README.md, report writing and review.

2. Project design

2.1 Overview

The original concept of Branching algorithms is simple: Compute a solution of the instance based on the solutions of the subinstances. So the main design of our implements will be in detailed problem part. Here is branching algorithm tree:



2.2 Vertex Cover

I implemented vertex cover in three ways: a running algorithm, visualize simplification rules for vertex cover and an interactable widget to play with.

An implementation is considered as working if it outputs a correct answer within a reasonable time. There are simple and complex graphs available in Sage math library. We can also create our own graphs with Sage math functions. So, it is relatively easy to test if the algorithm is working. Since Sage math library also provides functions to plot graphs, we can examine the output straightforward by observation.

The running algorithm part is to convert the pseudo-code on lecture slides into a working code. The way to test if the code or the algorithm is running is give it graphs and a corresponding k that we know if it is a YES-instance or a NO-instance and see if it outputs the expected answers. However, the algorithm is a recursive call(branching), so it will be gradually slower as the graph becomes larger. But we are still able to manually test if the output is correct by observing the result graph.

The simplification rule is to visualize the steps on lecture slides into a step-by-step procedure with corresponding input graph and output graph, this will be shown in detail in next part. The way to test if this is working is comparing the graphs before and after applying the rule, which is relatively easy and straightforward as the can be printed out. This part does not

require complicated graphs. The benchmark of saying if this is working is to say if the algorithm correctly finds corresponding vertices with given conditions (degree-0, degree-1 and so on).

The interactable widget is a game that user can set a budget k at the beginning for a given graph, then they can decide which vertex they would like to put into vertex cover in each step, the selected vertices would be coloured in red. If the budget runs out the user will get a warning message indicating that or a message saying this is a YES-instance as the user found a vertex cover within the budget. To test if this works is by playing with the widget and see if there is any bug in it. Edge cases will be tested like non-existing vertices or double adding. The widget also provides functionality to remove vertices from the vertex cover.

2.3 Feedback Vertex Set

The design of the project revolves around the implementation of an algorithm to find a feedback vertex set (FVS) of a given size for a graph and somehow display it to users via @interact feature provided in SageMath.

Besides, reasonable test design is also necessary during implementation to validate the correctness of the algorithm itself.

In such case, the work was split into three parts:

1. FVS function
2. Interactive function
3. Test cases (merged to integrated version)

For the first part, slides from the course were examined several times which turns out the best approach is to use a recursive function including appropriate reduction rules, such as removing loops, merging multiple edges and handling vertices of degree 1 and 2, to find the FVS of size at most k.

Also, an acyclicity check is applied to check if the graph is acyclic, which is the key terminate condition for the recursive function. In other words, if the graph becomes acyclic (a forest) at any point, the function would returns true, indicating that an FVS of size k or less has been found.

For visualization purposes, the function keeps track of the steps taken. For each step, there are relevant text messages together with the current graph object. More specifically, in order to achieve good educational outcomes, the messages include:

1. Step ID, counting how many steps it took.
2. k, stepwise decreasing until solution found or $k < 0$.
3. State message (inserted in the algo), showing current state.
    a) Whether the reduction rules are applied or,
    b) The function is terminating itself.

2.4 Maximum Leaf Spanning Tree

For the maximum leaf spanning tree part, after repeatedly reviewing the lecture, I decided on the design of implementation steps. Initially, I need to outline the algorithm process as pseudocode and then implement it in code. After ensuring the code's functionalities were correctly implemented, I should save the state of the graph at points where changes to the graph's state might occur. Finally, in the “@interact“ section, I can display the changes in the graph's state according to the algorithm's process.

After the primary algorithm has been implemented, I start to review my code design. Initially, I thought of recording the state changes at the beginning of the recursive function, but this approach failed to correctly capture the simplifications made by various simplification rules, so I incorporated the recording function into each branch to ensure every step was accurately recorded. After correctly displaying the algorithm's process, I felt that merely showing the different states of the graph was insufficient to clearly demonstrate the functionality of branching algorithm in the maximum leaf spanning tree question. Thus, I enhanced the recording function to log the type of operation being performed and added extensive explanatory text in the “@interact“ part to ensure each operation of the algorithm was correctly explained and demonstrated.

Subsequently, I further elaborated on specific terminologies within the algorithm, such as the key terms T, I, B, L, X, Non-extendable simplification rule, branching Lemma, and follow path lemma. Explaining these key terms made the steps of the algorithm clearer and easier to understand.

2.5 Test instance design

The key concept of our test design is that the test graph could clearly demonstrate how branching algorithms handle all possible scenarios. And we design both YES-instance and NO-instance for each part. For example, the test example we used in maximum leaf spanning tree section, the algorithm will execute operations of “start from a vertex”, “simplification rule”, “go to branch 1”, “recall branch 1 operation”, “follow path lemma”, “go to branch 2: branch lemma”, “recall branch 2 operation”, “re-select a vertex as start vertex”, “recall”, “give the result”. This algorithm takes 32 steps to give a YES result for YES instance takes 60 steps to give a NO result for NO instance. The test instances will execute all operations possibly happens in branching algorithm for maximum leaf spanning tree section.

3. Project implementation

3.1 Implementation

The implementation of project will be explained detailedly in different part.

3.1.1 Vertex Cover

For the running algorithm, I just converted the pseudo-code on lecture slides into a working code, for a given graph and a k. It will output you with either a message saying it is a NO-instance or a new graph with vertices in vertex cover in red.

Below is the code for the algorithm, comments above each line of code is the explanation of what the code does.

```
# the function takes in three parameters
# G: the input graph
# k: remaining budget
# cover: default as empty but will change if there is a provided non-empty set
def vertex_cover_recursive(G, k, cover=set()):
    # see if there is any edge that is not covered
    uncovered_edges = [e for e in G.edges(labels=False) if not (e[0] in cover or e[1] in cover)]
    if not uncovered_edges:
        return cover  # return current vertex cover
    if k <= 0:
        return None  # run out of budget but there still are edges left
    # select an edge from E
    u, v = uncovered_edges[0]
    # add u into vertex cover and go on
    cover_with_u = vertex_cover_recursive(G, k - 1, cover.union({u}))
    if cover_with_u is not None:
        return cover_with_u
    # add v into vertex cover and go on
    cover_with_v = vertex_cover_recursive(G, k - 1, cover.union({v}))
    if cover_with_v is not None:
        return cover_with_v
    # return no if u and v are both not valid vertex
    return None
```

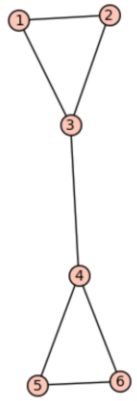
This is the block of code that will print answers to user terminal.

If it is a YES-instance, user will get a resulting graph, if it is a NO-instance, the user will also be notified. Both will be shown below.

```
def vertex_cover(G, k):
    # starting with empty vertex cover
    cover = vertex_cover_recursive(G, k)
    if cover is not None:
        print("Found a vertex cover:", cover)
        # mark the vertices in VC in red and other vertices in white
        vertex_colors = {'red': list(cover), 'white': [v for v in G.vertices() if v not in cover]}
        # plot the resulting graph
        G.show(vertex_colors=vertex_colors)
        return cover
    else:
        print("No vertex cover found with", k, "vertices")
        return None
```

This is testing with a NO-instance and the corresponding graph and answer can be seen from the screenshot.

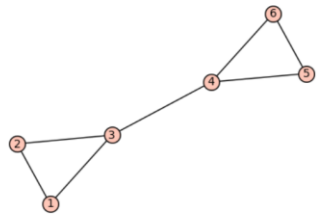
```
# create a graph
# G = graphs.PetersenGraph()
G = Graph([(1, 2), (2, 3), (3, 1), (3, 4), (4, 5), (5, 6), (6, 4)])
show(G)
# try the algorithm with created G and k=3
vertex_cover(G, 3)
```



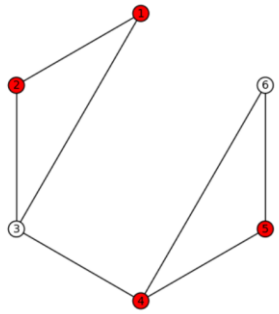

```
No vertex cover found with 3 vertices
```

It will be a YES-instance if we increase the budget by one as shown below.

```
# create a graph
# G = graphs.PetersenGraph()
G = Graph([(1, 2), (2, 3), (3, 1), (3, 4), (4, 5), (5, 6), (6, 4)])
show(G, layout='circular')
# try the algorithm with created G and k=4
vertex_cover(G, 4)
```

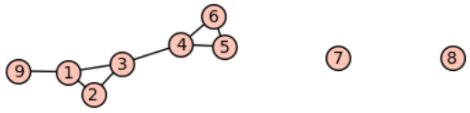



```
Found a vertex cover: {1, 2, 4, 5}
```



For simplification rule, A graph G is created at the beginning.

```
:
# define edges and nodes
vertices = [1, 2, 3, 4, 5, 6, 7, 8]  # including nodes of 0 degree(7 & 8)
edges = [(1, 2), (2, 3), (3, 1), (3, 4), (4, 5), (5, 6), (6, 4), (1, 9)]

# create the graph
G = Graph([vertices, edges])
# G = Graph([(1, 2), (2, 3), (3, 1), (3, 4), (4, 5), (5, 6), (6, 4)])

# define a budget k
k = 5

# show the graph
G.show()
# plot(G)

# obtain all vertices with degree-3
# degree_of_3 = G.degree(3)

# show degree for each node starting from node 0
# all_degrees = G.degree()
```



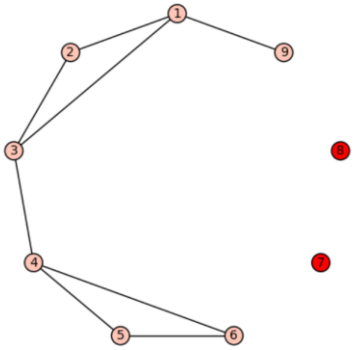
Below is the code and corresponding output before and after applying degree-0 simplification rule.

(Degree-0)

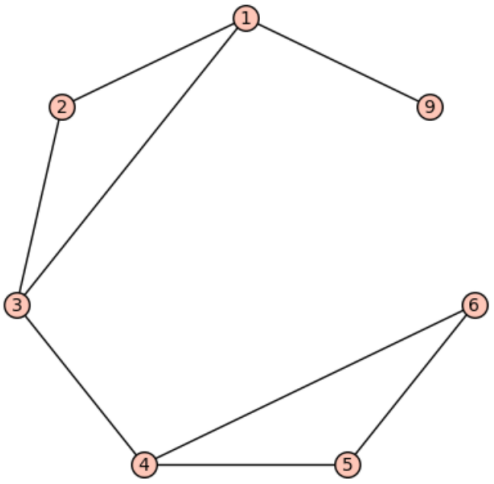
If ∃v ∈ V such that dG(v) = 0, then set G ← G − v

```
# obtain all vertices with degree-0
degree_of_0 = [v for v in G.vertices() if G.degree(v) == 0]
print("Degree of node 0:", degree_of_0)

# mark nodes with degree 0 with color red
vertex_colors = {'red': degree_of_0}

# show the result G
G.show(vertex_colors=vertex_colors, layout='circular')
```

```
Degree of node 0: [7, 8]
```




```
# delete nodes with degree 0
G.delete_vertices(degree_of_0)

# show the new graph
G.show(layout='circular')
```

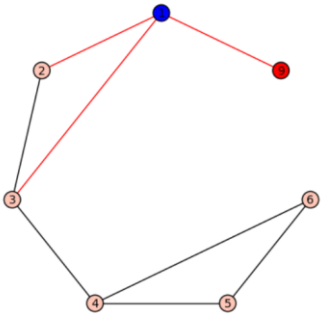


Similar to above, this is the corresponding content for applying degree-1 simplification rule.

(Degree-1)

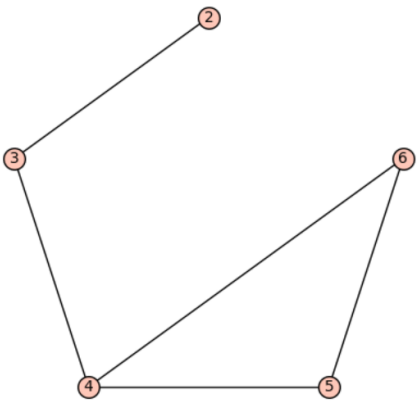
If ∃v ∈ V such that dG(v) = 1, then set G ← G − NG[v] and k ← k − 1

```
# obtain all vertices with degree-1
degree_of_1 = [v for v in G.vertices() if G.degree(v) == 1]

# and their neighbors
neighbor_vertex = []
for vertex in degree_of_1:
    neighbor_vertex.append(G.neighbors(vertex)[0])

# mark nodes with degree 1 with color red
# mark neighbor of nodes with degree 1 with color blue
vertex_colors = {'red': degree_of_1, 'blue': neighbor_vertex}

# find all edges connected to the neighbor vertices and mark them red
neighbor_edges = [(u, v) for u, v in edges if u in neighbor_vertex or v in neighbor_vertex]

edge_colors = {'red': neighbor_edges}

# show the result G
G.show(vertex_colors=vertex_colors, edge_colors=edge_colors, layout='circular')
```




```
# delete nodes with degree 1
G.delete_vertices(degree_of_1)
# and their beighbor
G.delete_vertices(neighbor_vertex)

print("k before delete", k)
k = k - len(degree_of_1)
print("k after delete", k)

# show the new graph
G.show(layout='circular')
```

```
k before delete 5
k after delete 4
```

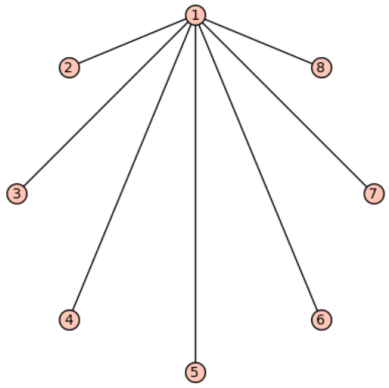


Below is visualize for simplification rule “Large degree”.

(Large Degree)

If ∃v ∈ V such that dG(v) > k, then set G ← G − v and k ← k − 1.

```
# define edges and nodes
vertices = [1, 2, 3, 4, 5, 6, 7, 8]  # including nodes of 0 degree(7 & 8)
edges = [(1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8)]

# create the graph
G = Graph([vertices, edges])

k = 5

G.show(layout='circular')

all_degrees = G.degree()
print("Number of Vertex:", len(vertices))
print("Number of Edges:", G.size())
print("Cuurrent k:", k)
```




```
Number of Vertex: 8
Number of Edges: 7
Cuurrent k: 5
```

The visualization for simplification rule below is not figured out yet, so it is just written like this.

(Number of Edges)

If dG(v) ≤ k for each v ∈ V and |E| > k^2 -> then return No

For interactable widget, A using instruction is provided at the start of the file.

Using instruction

1. To start using this tool fisrt select "Restart"
2. When "Restart" is selected, you can set the k in "Starting k" text box, then click "Run Interact" to set.
3. Then switch to "Select vertex", you can input the vertex that you want to put into Vertex Cover in "Select Vertex" text box
4. If you want to remove any vertex from the Vertex Cover, switch to "Remove Vertex", and do the same as step3

This is the internal setting of the game:

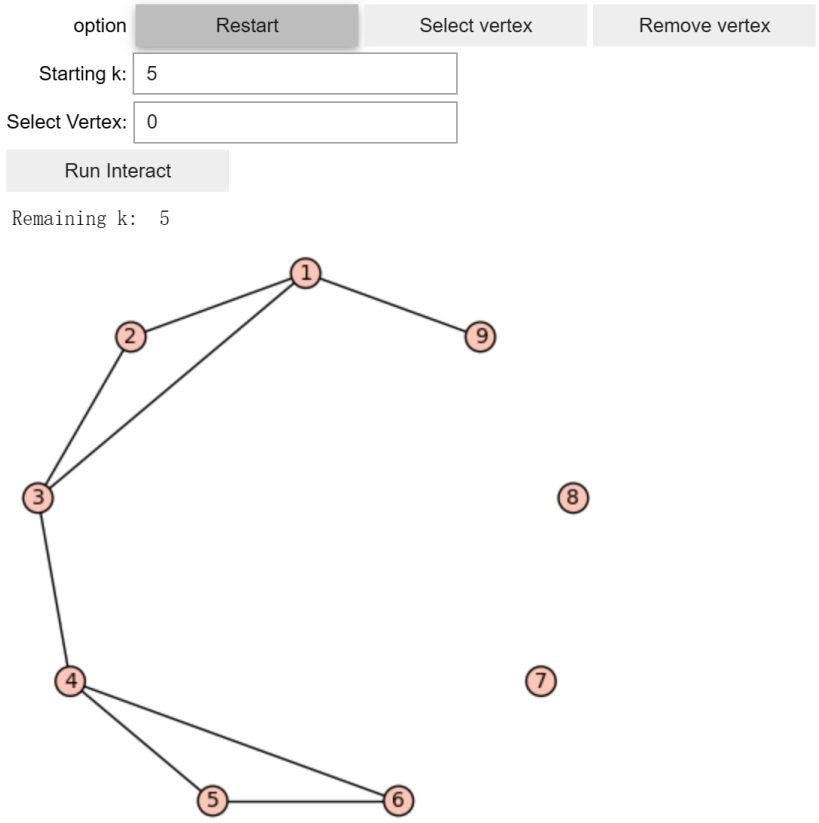
```
import copy

# define edges and nodes
vertices = [1, 2, 3, 4, 5, 6, 7, 8]  # including nodes of 0 degree(7 & 8)
edges = [(1, 2), (2, 3), (3, 1), (3, 4), (4, 5), (5, 6), (6, 4), (1, 9)]
# create the graph
G = Graph([vertices, edges])
# number of edge: G.size()

# store current vertex cover
vertex_cover = []
# store currently covered edges
edge_cover = []
# options for users to use
options = ['Restart', 'Select vertex', 'Remove vertex']
# remaing budget
remaining_k = 0
# flag to indicate the start of the game
initial_state = True

# mark nodes in VC with color red
vertex_colors = {'red': vertex_cover}
# mark edges in EC with color yellow
edge_colors = {'blue': edge_cover}

# obtain all vertices with degree-0
degree_of_0 = [v for v in G.vertices() if G.degree(v) == 0]
```

Below is the code that will be executed as the user is playing with the widget.

```
@interact
def _(option = selector(options, buttons=True),
      set_k = input_box(default=5, label="Starting k:"),
      vertex = input_box(default=0, label="Select Vertex:"),
      auto_update = False):

    global vertex_cover, edge_cover, remaining_k, initial_state, degree_of_0
    if initial_state:
        initial_state = False
        remaining_k = deepcopy(set_k)

    if option=='Restart':
        vertex_cover=[]
        edge_cover = []
        remaining_k = deepcopy(set_k)
    elif option=='Select vertex':
#         if remaining_k <= 0:
#             print('Cannot proceed!')
#         elif len(edge_cover) == G.size():
#             print('You have found a vertex cover!')
#         else:
            if vertex not in vertex_cover:
                if vertex in degree_of_0:
                    print('You have added a vertex of degree-0, you may want to reconsider that!')
                vertex_cover.append(vertex)
                remaining_k -= 1
                # edge with one end is vertex
                edges_connected = [(u, v) for u, v, _ in G.edges() if u == vertex or v == vertex]
                for edge in edges_connected:
                    if edge not in edge_cover:
                        edge_cover.append(edge)
```

```
    else:
        if vertex in vertex_cover:
            vertex_cover.remove(vertex)
            remaining_k += 1
            # edge with one end is vertex
            edges_to_remove = [(u, v) for u, v in edge_cover if (u == vertex and v not in vertex_cover) or (v == vertex and u not in ver
            for edge in edges_to_remove:
                if edge in edge_cover:
                    edge_cover.remove(edge)  # remove edge from edge cover

    vertex_colors = {'red': vertex_cover}
    edge_colors = {'blue': edge_cover}

    if len(edge_cover) == G.size():
        print('You have found a vertex cover!')
    elif remaining_k <= 0:
        print('Cannot proceed!')

    print('Remaining k: ', remaining_k)
    # show the result G
    G.show(vertex_colors=vertex_colors, edge_colors=edge_colors, layout='circular')
```

Below is the interactable interface that s user will see and play with. User can play with it by following the steps on provided using instructions.



A YES-instance with provided graph and a user setting k=5

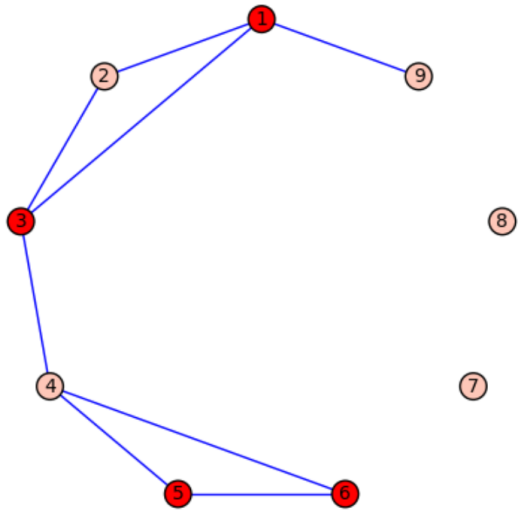
option | Restart | Select vertex | Remove vertex

Starting k: 5

Select Vertex: 6

Run Interact

```
You have found a vertex cover!
Remaining k:  1
```



A NO-instance if k = 3:

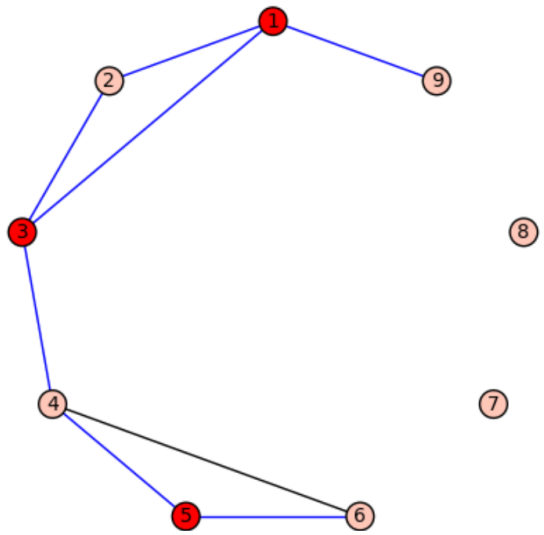
option | Restart | Select vertex | Remove vertex

Starting k: 3

Select Vertex: 5

Run Interact

```
Cannot proceed!
Remaining k:  0
```



3.1.2 Feedback Vertex Set

In the part of implementation, several screenshot would be shown in line with the design part described above, including the FVS function implementation, the interactive implementation without the test cases (merged to integrated version).

```
def is_acyclic(graph):
    """Check if the graph is acyclic (i.e., a forest)."""
    return graph.is_forest()
```

Acyclicity check function returns true if the graph is acyclic.

```
def feedback_vertex_set(graph, k, visited=None):
    """Recursive function to find a feedback vertex set of size at most k."""
    global steps
    if visited is None:
        visited = set()
        steps = [(deepcopy(graph), k, "Original")]  # Initialize steps with the original graph

    if is_acyclic(graph) and k >= 0:
        steps.append((deepcopy(graph), k, "Graph is acyclic, solution found"))
        return True

    if k < 0:
        return False
```

Recursive function to find a feedback vertex set of size at most k. Figure shows the two terminate condition in the function.

```
for v in graph.vertices():
    # Remove Loops
    if graph.has_edge(v, v):
        graph.delete_vertex(v)
        steps.append((deepcopy(graph), k, f"Removed loop at {v}"))
        if feedback_vertex_set(graph, k - 1, visited):
            return True

for u, v in graph.edges(labels=False):
    # Merge multiple edges
    if graph.multiple_edges(u, v):
        graph.merge_edges([(u, v)], loops=True)
        steps.append((deepcopy(graph), k, f"Merged multiple edges between {u} and {v}"))

for v in graph.vertices():
    # Handle degree-1 vertices
    if graph.degree(v) == 1:
        graph_copy = deepcopy(graph)
        graph_copy.delete_vertex(v)
        steps.append((graph_copy, k, f"Removed degree-1 vertex {v}"))
        return feedback_vertex_set(graph_copy, k, visited)

    # Handle degree-2 vertices
    elif graph.degree(v) == 2:
        u, w = graph.neighbors(v)
        graph_copy = deepcopy(graph)
        graph_copy.delete_vertex(v)
        if not graph.has_edge(u, w):
            graph_copy.add_edge(u, w)
            if k >= 1:
                steps.append((graph_copy, k - 1, f"Removed degree-2 vertex {v} and added edge {u}-{w}"))
        else:
            if k >= 1:
                steps.append((graph_copy, k - 1, f"Removed degree-2 vertex {v} "))
        if feedback_vertex_set(graph_copy, k - 1, visited):
            return True
```

The core of the implementation, this recursive function tries to find an FVS of size at most k. It employs several strategies to simplify the graph:

1. Removing loop: Self-loops are removed, as they are trivial cycles.
2. Merging multiple edges: Multiple edges between the same pair of vertices are merged into a single edge.
3. Handling degree-1 vertices: Vertices of degree 1 can be safely removed, as they cannot be part of any cycle.
4. Handling degree-2 vertices: For a vertex of degree 2, its neighbors are connected directly, and the vertex is to be removed.

Key words “steps.append” provides step information for the visualization, the graph and the text messages of what the function is doing currently.

```
# Recursive case: try removing each vertex
for v in graph.vertices():
    if v not in visited:  # Avoid re-visiting the same vertex
        visited.add(v)
        graph_copy = deepcopy(graph)
        graph_copy.delete_vertex(v)
        if k >= 1:
            steps.append((deepcopy(graph_copy), k - 1, f"Trying removal of {v}"))
            if feedback_vertex_set(graph_copy, k - 1, visited):
                return True

return False
```

Recursive part, add vertices into visited set avoiding re-visiting.

```
def main(graph, k, visited=None):
    """Main function to find a feedback vertex set of size at most k."""
    result = feedback_vertex_set(graph, k, visited=None)
    if not result:
        steps.append((deepcopy(graph), k, f"No solution!"))
    return result
```

Main function handles false result for the interactive part, injecting failing messages to the visualization.

```
@interact
def show_step(step=slider(0, len(steps)-1, 1, label="Step")):
    """Interactive function to visualize the steps of the algorithm."""
    graph, k, description = steps[step]
    G_plot = graph.plot(title=f"Step {step}: k={k}\n{description}",layout='circular')
    show(G_plot)
```

Using @interact feature to show the steps specified during the function.

3.1.3 Maximum Leaf Spanning Tree

My final version of the implementation includes 3 main parts: key conceptions, branching algorithm for maximum leaf spanning tree and algorithm demonstration.

Terminologies part. This part gives a simple example to show the T, I, B, L, X in different colors.

1. Key conceptions in Maximum Leaf Spanning Tree

1.1. terminologies

T - a tree in G(includes edges and I, B, L)

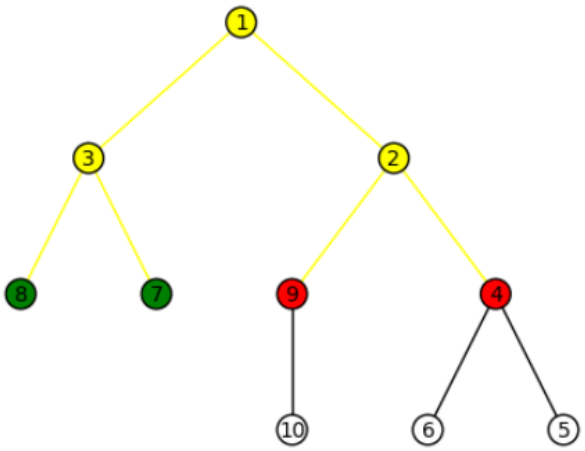
I - the internal vertices of T, with r∈I

B - a subset of the leaves of T where T may be extended: the boundary set

L - the remaining leaves of T

X - the external vertices V\V(T)
Here is a simple exmple of them:

In [17]:
```
G_instruction_example = Graph([(1, 2), (1, 3), (2, 4), (4, 5), (4, 6), (3, 7), (3, 8),
T = Graph([(1, 2), (1, 3), (2, 4), (3, 7), (3, 8), (2, 9)])
I = {1, 2, 3}
L = {7, 8}
B = {4, 9}
X = {5, 6, 10}
# vertex colors
vertex_colors = {'green': list(L), 'red': list(B), 'white': list(X), 'yellow': list(I)

# edge_colors
edge_colors = {'yellow': T.edges(labels=False)}
G_instruction_example.show(vertex_colors=vertex_colors, edge_colors=edge_colors, layou
```



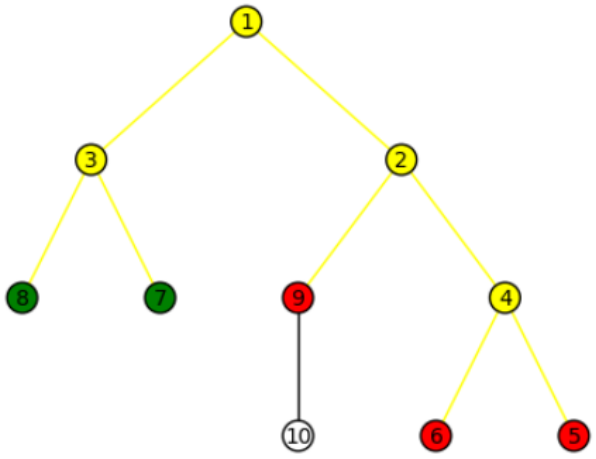
Branching Lemma. This part use the same graph to show what branching lemma do in the graph.

1.2.1. Branching Lemma

Suppose u ∈ B and there exists a k-leaf tree T' extending T where u is an internal vertex. Then, there exists a k-leaf tree T' extending (V(T)∪N_G(u),E(T)∪{uv:v∈N_G(u)∩X}).

For example, if we choose node 4 in Graph above, here is result after execute the branching lemma.

In [18]:
```
T = Graph([(1, 2), (1, 3), (2, 4), (3, 7), (3, 8), (2, 9), (4, 6), (4, 5)])
I = {1, 2, 3, 4}
L = {7, 8}
B = {5, 6, 9}
X = {10}
# vertex colors
vertex_colors = {'green': list(L), 'red': list(B), 'white': list(X), 'yellow': list(I)

# edge_colors
edge_colors = {'yellow': T.edges(labels=False)}
G_instruction_example.show(vertex_colors=vertex_colors, edge_colors=edge_colors, layou
```



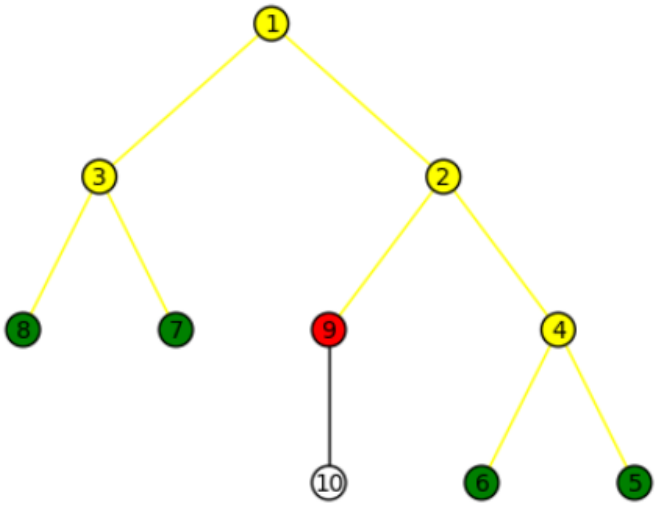
Non-extendable simplification rule. This part shows which vertex will be moved after applying non-extendable simplification rule.

1.2.2. Non-extendable simplification rule

If exists v ∈ B with N_G(v) ∩ X = Ø, then move v to L.

For example, if we apply this simplification rule to the graph above, here is result.

```
T = Graph([(1, 2), (1, 3), (2, 4), (3, 7), (3, 8), (2, 9), (4, 6), (4, 5)])
I = {1, 2, 3, 4}
L = {5, 6, 7, 8}
B = {9}
X = {10}
# vertex colors
vertex_colors = {'green': list(L), 'red': list(B), 'white': list(X), 'yellow': list(I)

# edge_colors
edge_colors = {'yellow': T.edges(labels=False)}
G_instruction_example.show(vertex_colors=vertex_colors, edge_colors=edge_colors, layou
```



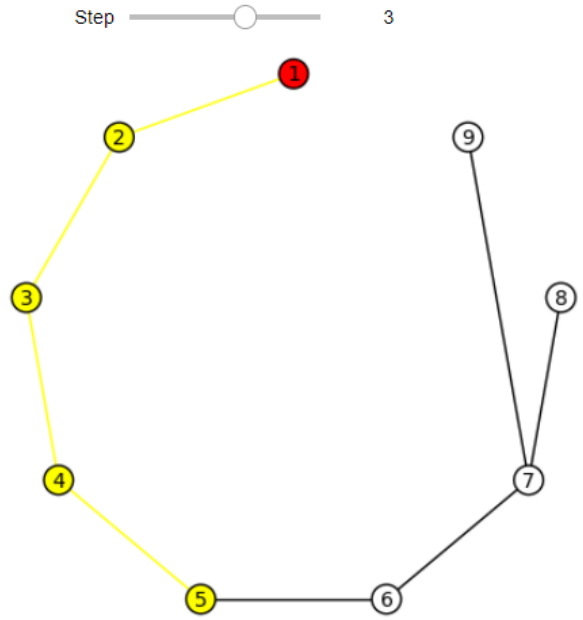
Follow path lemma. This part I use the @interact component to dynamically demonstrate how the follow path lemma works and what time it should be terminated.

1.2.3. Follow Path Lemma

Suppose u ∈ B and |N_G(u)∩X| = 1. Let N_G(u)∩X = {v}. If there exists a k-leaf tree extending T where u is internal, but no k-leaf tree extending T where u is a leaf, then there exists a k-leaf tree extending T where both u and v are internal.

Here is simple exmple of use of Follow Path Lemma:

```
G_instruction_example = Graph([(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (7, 8),
steps_instruction_example = []
#initial state
u = 1
T = Graph()
T.add_vertex(u)
I = set()
L = set()
B = {u}
X = set(G_instruction_example.vertices()) - {u}
neighbors_X = set(G_instruction_example.neighbors(u)) & X
#Follow Path Lemma
while len(neighbors_X) == 1:
    v = neighbors_X.pop()
    X.remove(v)
    T.add_edge(u, v)
    neighbors_X = set(G_instruction_example.neighbors(v)) & X
    u = v  # update v as new u(current vertex)
    if len(neighbors_X) == 1:
        I.add(u)
    elif len(neighbors_X) != 1:
        B.add(u)
    steps_instruction_example.append([copy.deepcopy(T), copy.deepcopy(I), copy.deepcop
@interact
def show_step(step=slider(0, len(steps_instruction_example)-1, 1, label="Step")):
    T, I, B, L, X, recall, operation_type = steps_instruction_example[step]
    # vertex colors
    vertex_colors = {'green': list(L), 'red': list(B), 'white': list(X), 'yellow': lis

    # edge_colors
    edge_colors = {'yellow': T.edges(labels=False)}

    # plot graph
    G_instruction_example_plot = G_instruction_example.plot(vertex_colors=vertex_color

    show(G_instruction_example_plot)
```



Branching algorithm for maximum leaf spanning tree. After I explained all key concepts, Here is the implementation of branching algorithm.

In [35]:

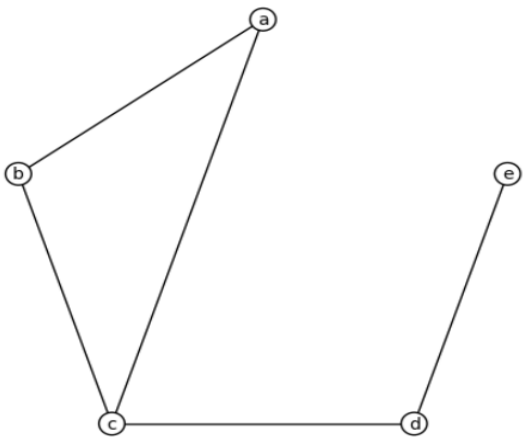
```
steps = []
def extend_tree(G, k, T, I, B, L, X, recall):
    #record all steps
    recall = 0

    # Halt - yes
    if len(L) + len(B) >= k:
        return True

    # Halt - No
    if len(B) == 0:
        steps[-1][5] += 1
        return False

    # Simplication rule: If exist v∈B with N_G(v) ∩ X = 0, then move v to L
    B_to_L = {v for v in B if not set(G.neighbors(v)) & X}
    for v in B_to_L:
        B.remove(v)
        L.add(v)
        steps.append([copy.deepcopy(T), copy.deepcopy(I), copy.deepcopy(B), copy.deepc

    #Select u from B
    for u in list(B):

        # branch 1: try to move a vertex from B to L
        B.remove(u)
        L.add(u)
        steps.append([copy.deepcopy(T), copy.deepcopy(I), copy.deepcopy(B), copy.deepc
        if extend_tree(G, k, T, I, B, L, X, recall):
            return True
        # recall branch 1
        L.remove(u)
        B.add(u)
        steps.append([copy.deepcopy(T), copy.deepcopy(I), copy.deepcopy(B), copy.deepc

        # branch 2: handle u as a internal vertex (Branching Lemma)
        original_u = u  # save the original vertex
        added_edges = []  # save added edges
        B.remove(u)
        I.add(u)
        neighbors_X = set(G.neighbors(u)) & X

        #Follow Path Lemma
        while len(neighbors_X) == 1:
            v = neighbors_X.pop()
            X.remove(v)
            T.add_edge(u, v)
            added_edges.append((u, v))
            neighbors_X = set(G.neighbors(v)) & X
            u = v  # update v as new u(current vertex)
            if len(neighbors_X) == 1:
                I.add(u)
            elif len(neighbors_X) != 1:
                B.add(u)
            steps.append([copy.deepcopy(T), copy.deepcopy(I), copy.deepcopy(B), copy.d

        B.discard(u)
        # process branch 2
        for v in neighbors_X:
            T.add_edge(u, v)
            added_edges.append((u, v))
            B.add(v)
            X.remove(v)
            steps.append([copy.deepcopy(T), copy.deepcopy(I), copy.deepcopy(B), copy.d

        if extend_tree(G, k, T, I, B, L, X, recall):
            return True

        # recall branch 2
        for u, v in added_edges:
            T.delete_edge((u, v))  # delete added edges
            I.discard(v)
            X.add(v)
        I.discard(original_u)  # remove original u
        B.add(original_u)  # add original u to B
        X.update(neighbors_X)  # update unused vertices to X
        for v in neighbors_X:
            B.discard(v)  # remove vertices from B
        steps.append([copy.deepcopy(T), copy.deepcopy(I), copy.deepcopy(B), copy.deepc
    steps[-1][5] += 1
    return False

def max_leaf_spanning_tree(G, k):
    T = Graph()
    I = set()
    L = set()
    B = set()
    X = set(G.vertices())
    recall = 0
    # save initial state
    for r in G.vertices():
        steps.append([Graph(), set(), set(), set(), set(G.vertices()), 0, 8])
        T = Graph() #a tree in G
        T.add_vertex(r) #add initial vertex to T
        I = set()  # the internal vertices of T, with r∈I
        L = set()  # the remaining leaves of T
        B = {r}  # a subset of the leaves of T where T may be extended: the boundary s
        X = set(G.vertices()) - {r}  # the external vertices V|V(T)
        steps.append([copy.deepcopy(T), copy.deepcopy(I), copy.deepcopy(B), copy.deepc
        if extend_tree(G, k, T, I, B, L, X, recall):
            return True, T
    return False, None
```

Algorithm demonstration. This part I visualize a step-by-step execution of branching algorithm and give a plenty of texts to explain every single step.

In [38]:
```
@interact
def show_step(step=slider(0, len(steps)-1, 1, label="Step")):
    T, I, B, L, X, recall, operation_type = steps[step]
    # vertex colors
    vertex_colors = {'green': list(L), 'red': list(B), 'white': list(X), 'yellow': lis

    # edge_colors
    edge_colors = {'yellow': T.edges(labels=False)}

    # plot graph
    G_plot = G.plot(vertex_colors=vertex_colors, edge_colors=edge_colors, layout='circ

    # show current steps of algorithm

    print("The current tree T are represented in yellow, the nodes in B are in red, th
    print(" and the remaining unchanged parts(X) set to white.\n")
    print(f"Current state: I:{I if I else ' Null'}, B:{B if B else ' Null'}, L:{L if

    # operation explanation
    if operation_type == 1:
        print(f"This state we choose a new vertex as a start vertex.\n")
    elif operation_type == 2:
        print(f"Simplication rule: If exist v∈B with N_G(v) ∩ X = 0, then move v to
        print(f"This means all vertices in B that can not be expanded will be moved to
    elif operation_type == 3:
        print(f"This step we go to branch 1: try to move a vertex from B to L.\n")
    elif operation_type == 4:
        print(f"Recall the branch 1 operation: move the vertex back to B.\n")
    elif operation_type == 5:
        print(f"This step will process follow path lemma.\n")
    elif operation_type == 6:
        print(f"This step we go to branch 2: handle current vertex u as a internal ver
        print(f"Then, put all vertices adjacent to the current vertex u and not yet pr
    elif operation_type == 7:
        print(f"Recall the branch 2 operation: Branching Lemma.\n")
    elif operation_type == 8:
        print(f"This step is initial state of the graph.\n")

    # recall info
    if recall == 1:
        print(f"This branch has no more leaves that may be extended, this state will r
    elif recall == 3:
        print(f"This initial vertex is not available, the state will recall to initial

    #result
    if step == len(steps)-1:
        if result:
            print(f"The number of sum of vertices in B and vertices in L has reached k
            print(f"YES. The branching algorithm find a maximum leaf spanning tree wit
        else:
            print(f"The branching algorithm has test all viable tree, but can't reach
            print(f"NO. The branching algorithm find out there is no maximum leaf span
    show(G_plot)
```

Step 0

```
The current tree T are represented in yellow, the nodes in B are in re
d, the nodes in L are in green,
 and the remaining unchanged parts(X) set to white.

Current state: I: Null, B: Null, L: Null, X:{'c', 'a', 'd', 'e', 'b'},
k:3

This step is initial state of the graph.
```



3.2 Challenges

3.2.1 Vertex Cover

The challenge I met was to visualize simplification rule “number of edges”. It is a conceptual idea that I have k vertices, and each can cover at most k edges, so the amount of edge I can cover with my vertex set is k*k, if my |E| is greater than k^2, then it is a NO-instance.

3.2.2 Feedback Vertex Set

The unfamiliarity of @interact feature constitutes most of the challenge when working on the project. At first, I could only plot the static search tree. After several hours learning through SageMath wiki and discussion with other group members, we decided to use the “slider” key word which is thought fine tuned for visualizing dynamic graph object.

Then, I took some time design the sufficient messages to visualize (see design part) and make efforts to put them into practice using global variable “steps”. Messages are recorded in “steps” during each recursion using the key word “steps.append”. What left to do is implement function with the @interact feature and combine the global variable “steps” with “slider” key word. The result went all good in my codes.

3.2.3 Maximum Leaf Spanning Tree

For my implementation, the first challenge I encountered was how to actualize the code. Although the teacher provided a comprehensive explanation of the code in class, the steps explained still differed somewhat from those needed for implementation. After thoroughly understanding the algorithm, I organized a clear code process and implemented the algorithm based on this process. After verifying the correctness of the code, my next challenge arose from my unfamiliarity with SageMath, particularly in choosing which features to use to display the algorithm's process. Without wanting to significantly alter the original algorithmic function framework, I found that using @interact to display the algorithm was a good choice. I saved the various states of the algorithm and displayed them step by step in sequence using @interact. However, I found this approach still wasn't intuitive enough, so when saving the states, I used recall and operation_type to record more information, including whether the algorithm backtracked at this step, and what operation was performed. During the presentation, based on this information, I provided corresponding explanations of the algorithm to help students understand the operations performed at each step. Finally, I think I give a really informal and clear algorithm demonstration.

4. Conclusion

4.1 Outcomes

Our project gives three main outcomes. In vertex cover part, we provide a running code for Vertex Cover that will give a correct result with a given graph and a k, a visualization for Simplification Rules for vertex cover, a widget that user can play (most intuitive approach for user to get to know about vertex cover).
In Feedback vertex set part, we provide a brief visualization of the algorithm that can be disposed easily in future course material.
In maximum leaf spanning tree part, we provide a detailed visualization of the branching algorithm for the maximum leaf spanning tree. I visualized all the key concepts and displayed each step of the algorithm visually.

4.2 Merits

The advantage of our project is the detailed explanation of the algorithm's steps during the presentation, including what operations the algorithm performed and whether the algorithm needed to backtrack.
By visualizing the algorithms, we presented step-by-step plots of what happens at each stage. User can play with interactable tool to get a better idea of the concepts that are introduced in this project.
Additionally, different variable were marked in the algorithm process with various colors in two of the algorithms' visualization, which greatly aids in understanding the algorithm. Specially, we give a new interactable widget that user can execute algorithm steps by themselves, which help them to have a new experience and deep recognition of the algorithm.

4.2 Future works

We still have a bunch of ideas that is really cool to apply in education. For example, to develop an unified screen that can switch three branching algorithm by push button.
Moreover, we try to make interactable algorithm course more attention-catching. We will do same game widgets in vertex cover for the feedback vertex set and the maximum leaf spanning tree.
Additionally, in vertex cover part, we still have issues that the way to visualize simplification rule "number of edges" needs to improve.
In the future, the running time analysis would be another functionality task which would vastly enhance the educational outcomes if visualized well.