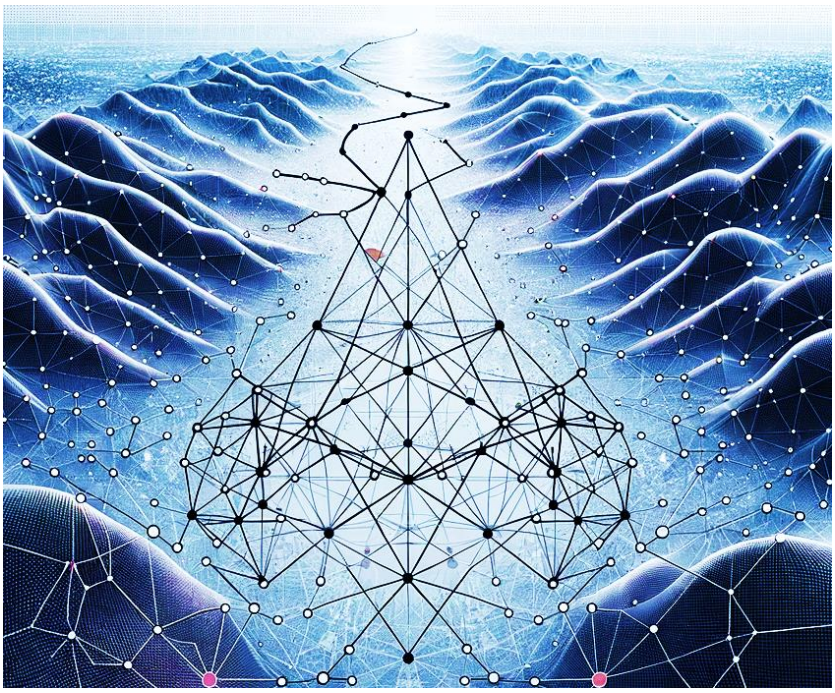

BREAK UP

Theoretical Foundations of Computer Science

Agam Mann - 20885863
Matthew Backhouse - 20898706
Jevi Waugh - 21008315

Unit: COMP3002
Group number: 14
Problem: 3 - Break-up

Curtin University – Department of Computing

Assignment Cover Sheet / Declaration of Originality

Complete this form if/as directed by your unit coordinator, lecturer or the assignment specification.

Last name:	Waugh	Student ID:	21008315
Other name(s):	Jevi		
Unit name:	Theoretical Foundations of Computer Science	Unit ID:	COMP3002
Lecturer / unit coordinator:	Hannes Herrmann	Tutor:	Hannes Herrmann
Date of submission:	26/10/2023	Which assignment?	(Leave blank if the unit has only one assignment.)

I declare that:

- The above information is complete and accurate.
- The work I am submitting is *entirely my own*, except where clearly indicated otherwise and correctly referenced.
- I have taken (and will continue to take) all reasonable steps to ensure my work is *not accessible* to any other students who may gain unfair advantage from it.
- I have *not previously submitted* this work for any other unit, whether at Curtin University or elsewhere, or for prior attempts at this unit, except where clearly indicated otherwise.

I understand that:

- Plagiarism and collusion are dishonest, and unfair to all other students.
- Detection of plagiarism and collusion may be done manually or by using tools (such as Turnitin).
- If I plagiarise or collude, I risk failing the unit with a grade of ANN ("Result Annulled due to Academic Misconduct"), which will remain permanently on my academic record. I also risk termination from my course and other penalties.
- Even with correct referencing, my submission will only be marked according to what I have done myself, specifically for this assessment. I cannot re-use the work of others, or my own previously submitted work, in order to fulfil the assessment requirements.
- It is my responsibility to ensure that my submission is complete, correct and not corrupted.

Signature: J.WAUGH Date of signature: 26/10/2023

(By submitting this form, you indicate that you agree with all the above text.)

Table of Contents

1 Abstract

This report was created to provide a solution to breaking up a shipping network into an integer ***n*** of unconnected components by removing edges and maximising the total graph weight. This report will outline assumptions made and transform problem 3 (Break Up), into an equivalent decision problem. Following, the problem will be classified for its general computability, decidability, and complexity. This report provides two algorithms, being a naive brute-force approach and a grouping solution. Each solution will show a pseudocode implementation without a Turing Machine, a Turing Machine pseudocode, classification for time and space complexity, and proof of decidability. Finally, the report will discuss similar problems and best practice.

2 Glossary

Term	Definition
Bell Number	The number of possible partitions of a set [1].
Direct Shipping	Shipping between a hub and a feeder [2], [3].
Feeder Port	A port that trades with only ports within its own port cluster [3].
Feeder Vessel	A ship that travels to and from feeder ports.
Hub Port	A port that trades within its own cluster and other hubs.
Liner	A large ship operated by a transport company [4]. In the context of this report, liners are specifically cargo ships.
Port Cluster	A collection of nearby ports based on geographic proximity [3].
Powerset	Given a set S, the power set of S is the set that contains all subsets of S. The power set of S is denoted by P(S). P(S) = { x | x is a subset of S }
Transhipping	Shipping between two feeders [2], [3].

3 Breakup Problem

This section will discuss the breakup problem, assumptions, and the transformation to a decision problem.

“A company that specialises in corporate takeovers is looking for an automated way to redistribute shipping networks of companies whose assets they are stripping. Your team will be given an undirected weighted graph that represents such a network, and an integer n and your software’s task will be to separate the graph into n unconnected components by removing edges.

To minimise lost opportunities, a further requirement is that if there are multiple ways to achieve the above aim, your project should remove a set of edges whose total weight is minimal.”

3.1 Assumptions

The following assumptions will be made:

3.1.1 Integer n

- It is assumed integer ***n*** is a positive integer.
- The algorithms must find a solution that breaks the graph into exactly ***n*** unconnected components if there exists one.

3.1.2 Nodes

- Each node will have a weight of zero.
    - The weight of a node in this context refers to the value of having a node in a component. This will be further explained for the grouping approach.
- The input graph will contain at least one node.
- Nodes will be interpreted as a shipping port.

3.1.3 Edges

- The input graph edges will not contain any reflexive edges.
- The weight of each edge will be greater than zero and will also refer to how valuable an edge is.
    - For example, an edge weight of 5 is more valuable than an edge weight of 3.
- No more than one edge will connect two nodes of the input graph.
- Edges will be interpreted as a shipping route between two ports.

3.1.4 Input Graph

- An input graph will always conform to the assumptions of sections 3.1.2 and 3.1.3.
- If the input graph is already in ***n*** components, our solution must recognise this.
- Our solution must also cover an input graph that is already broken down into ***x*** components where $0 < x \leq n$.
    - Essentially should be able to handle disconnected graphs.
    - If $x > n$, no solution exists.
- The input graph will always be an undirected, weighted graph.
- The input graph refers to a maritime shipping network consisting of ports and routes between them.

3.1.5 Unconnected Components

- It is assumed that each separate subgraph must contain at least two nodes and must consist of an edge between them.
    - Each sub-graph must have some trade between ports. A single isolated node indicates a port with no routes and hence should be considered an unacceptable component.
    - The reason for this is because the graph represents a shipping network. Having a shipping port with no routes would be a significant lost opportunity [5], as almost all port income is dependent on trade [6]. Since the overall idea is to redistribute the network, the complete isolation of a node will be considered unacceptable.

3.2 Decision Problem

The problem is considered solved if we can break the input graph into ***n*** components where each component has at least two nodes with an edge between them. Additionally, if a solution exists, we must find the optimal solution, store it, and enter an accept state. The optimal solution is where the

total value of all the remaining edges is the highest. If multiple optimal solutions exist, the first solution calculated will take precedence.

If no solution exists, we must enter the reject state and not provide a solution.

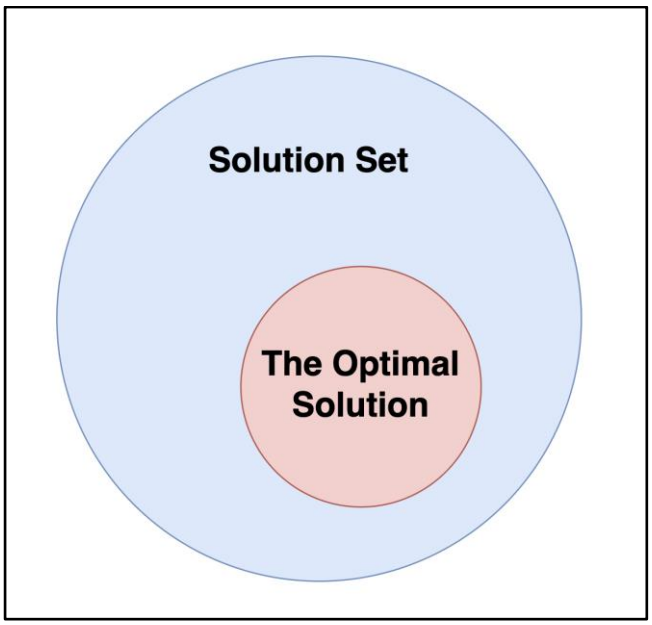

Fig. 3.1. Venn diagram of solution set and optimal solution.

The optimal solution is a subset to the solution to the problem as shown in [Fig 3.1].

The optimal solution will always be of size one.

- If no solution exists, there cannot be an optimal solution.
- If a solution exists, there must be an optimal solution.

If we are focused on only finding a solution, we do not necessarily find the optimal solution. But if we focus on finding the optimal solution, we will always solve the problem, adhere to constraints, and provide the optimal solution.

The decider will be:

"Does a solution exist on an undirected graph with an integer ***n*** where the graph is separated into ***n*** unconnected components by removing edges such that the edge weight loss is minimal, and each unconnected component has 2 nodes with connecting edges".

If so, the machine should accept and store the optimal solution. If no optimal solution exists, the machine should reject.

3.3 Example Input Graphs

The section covers possible input graphs that may exist. These graphs are separated based on validity. A valid graph is defined as a graph that is in the correct format. An invalid graph is a graph that will always be rejected due to not being in the correct format. These examples assume that ***n*** always follow 3.1.1's constraints. Numbers in nodes represent node weight.

3.3.1 Valid Graphs

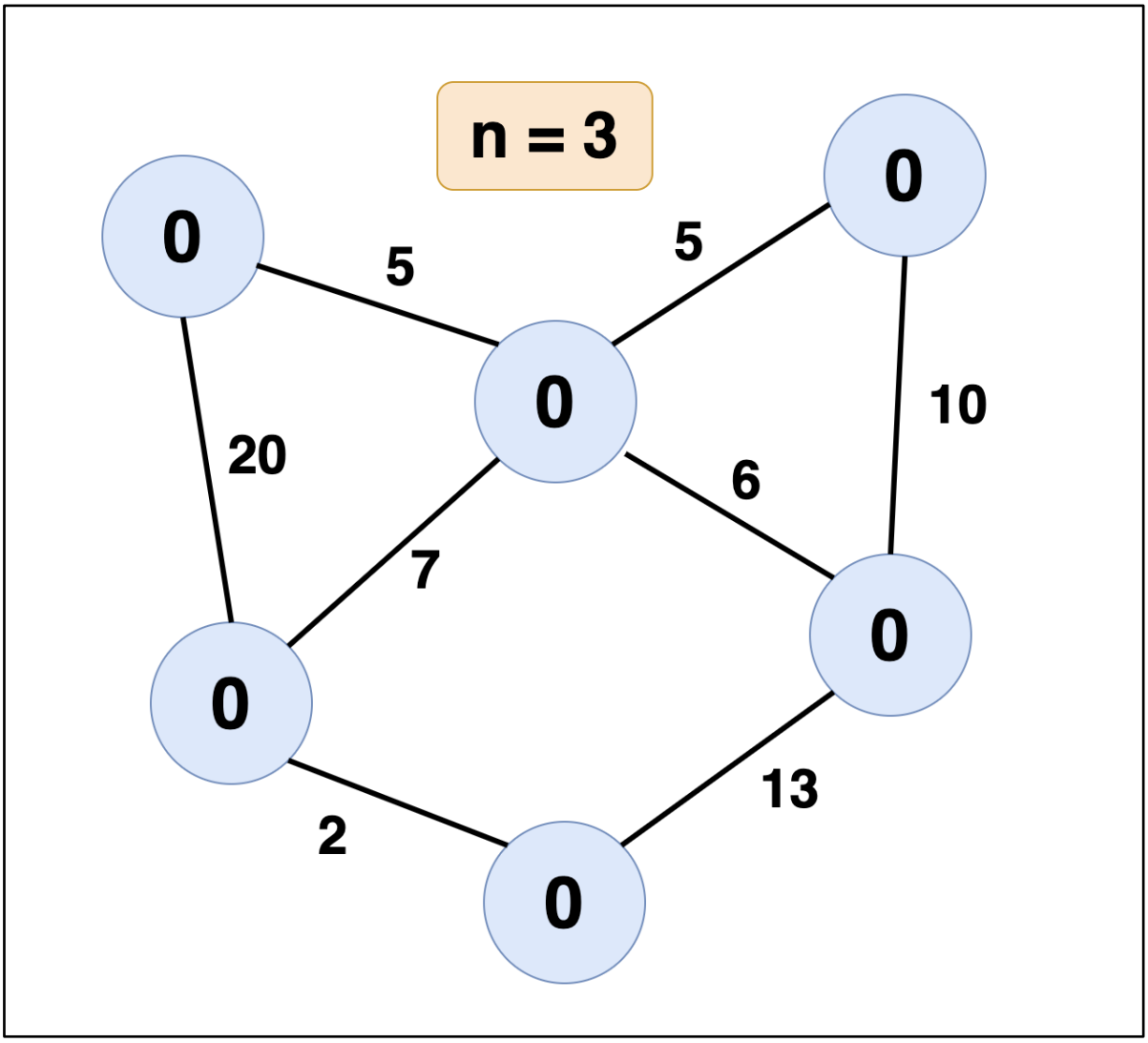

Fig. 3.2. Valid input graph example

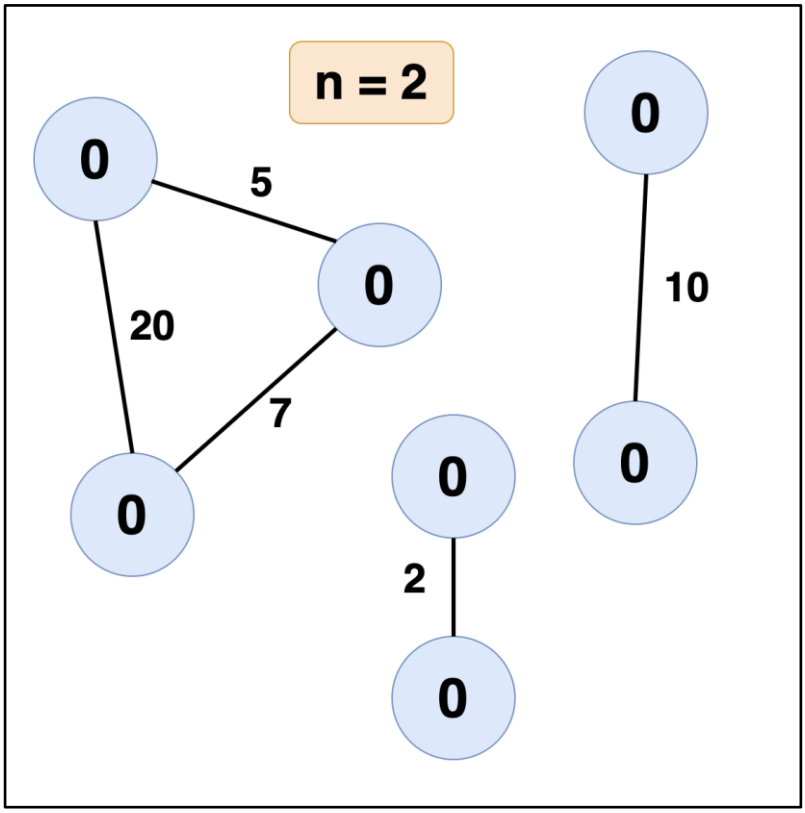

Fig. 3.3. Valid input graph on impossible solution

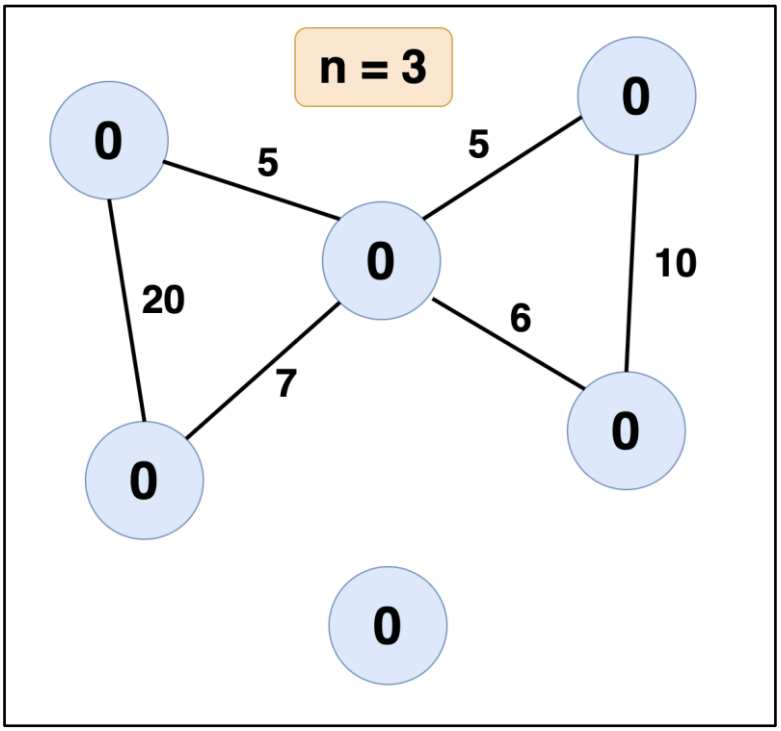

Fig. 3.4. Valid input graph when breaking 3.1.5 constraint.

- [Fig. 3.2] displays an example graph with a solution.
- [Fig. 3.3] is an example of a graph where it is impossible to break the graph into 2 unconnected components since it already has 3 unconnected components.
- [Fig. 3.4] is an example of a graph that doesn't follow constraint 1.1.5. Each unconnected component must have at least two nodes and an edge connecting them.
- [Fig. 3.2], [Fig. 3.3] and [Fig. 3.4] are considered valid graphs. Even though [Fig. 3.3] and [Fig. 3.4] have no solution, they must be rejected by the algorithms after checking format due to not following the constraints in section 3.1.

3.3.2 Invalid Graphs

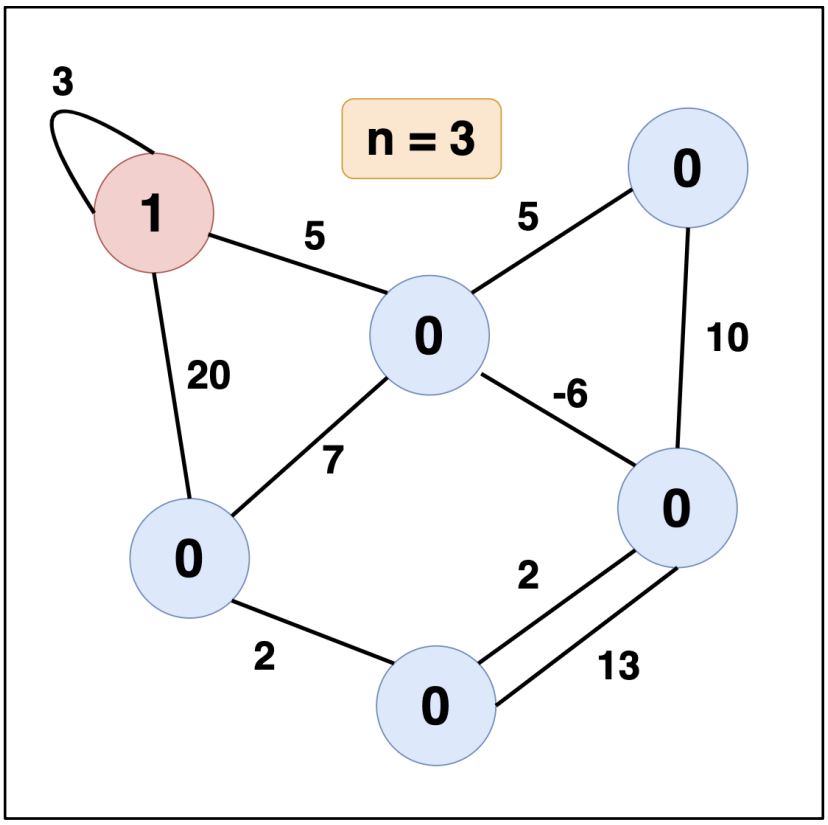

Fig. 3.5. Invalid input graph breaking multiple constraints.

[Fig. 3.5] displays an example graph that breaks multiple constraints:

- Red Node has a node weight that is not zero, which violates 3.1.2.
- Red Node has a reflexive edge, which violates 3.1.3.
- There exists a negative edge weight, which violates, 3.1.3
- There exists more than one edge between two nodes, which violates 3.1.3

4 Classification

4.1 Computability

The given problem accepts a graph as input, which must be parsed over multiple times to determine a solution due to the complex nature of the problem. Hence, the problem is tier 3 and requires at least a Turing machine to solve. By contrast, tier 1 and 2 machines are only able to consider each piece of input a single time making them unable to solve this problem for an arbitrary input. A Turing machine can iterate over input any number of times which allows it to, for example, generate the power set of edges to brute force the problem, or merge nodes to solve the problem in a dynamic manner.

4.2 Decidability

The problem asks for the best way to split a graph into multiple disconnected subgraphs, while maximising the weights of the remaining edges. Note that because the input contains a finite number of nodes, there is a finite number of ways to partition these nodes, which in turn means that a Turing machine can check all these possible partitions in turn. For each partition it can check if it is a valid solution and compare with the last found solution. This means the problem is decidable, since an accept or reject can be reached in all cases, given an appropriate machine to determine this.

4.3 Complexity

The given problem is an instance of the minimum k-cut problem (see section 7.2 for further details). While the initial non-decision version is NP-Hard, the decision version of the problem is NP-Complete. To prove that this is the case, the problem must (1) be shown to be NP, and (2) that it can be reduced to a known NP-Complete problem.

1) A non-deterministic Turing machine can be constructed to non-deterministically consider all possible cuts of the given graph and validate if they cut the graph into the appropriate number of disjoint subgraphs. Then, it would be able to compare them to provide the optimal solution. Hence, the given problem is in NP.
2) To demonstrate that a problem is NP-Complete, it can be reduced to SAT, a known NP-Complete problem. To do so, the minimum k-cut problem can be reduced to the maximum clique problem for arbitrary k [7]. From there the problem can be reduced to SAT [8]. The SAT problem is known to be NP-C, and reducing another problem to SAT demonstrates that it is also NP-C [9].

5 Brute force solution

5.1 General Notion

The general idea is to separate the graph into ***n*** unconnected components by removing a set of edges whose total weight is minimal. A way to implement that is by utilising a Brute-Force approach to go through all the possible subsets of edges (powerset) and removing the subset of edges that are the most minimal, which will result in a graph with the remaining edges that are maximised. This will leave the graph ***G*** divided into ***n*** unconnected components. Because this algorithm is heavily dependent upon traversing every edge within the graph, the time complexity of the powerset would be $O(2^n)$ because it computes all the possible combinations of edges singularly or together and iterates through each subset. This Brute force approach will endeavour to find a solution for the decision problem and basically, for each subset, it will remove the edges by using a temporary graph

and will also check the number of disconnected subgraphs by utilising a depth-first search traversal technique. This will solve the decision problem.

The following pseudocode implementation is a non-Turing machine algorithm solving the problem.

5.2 Pseudo Code

```
BruteForce(G, n)
    // Powerset Operation
    powerset = {()}// The empty set [[]]
    for every edge in G.edges
        create new_subsets ()
        for every subset in powerset
            create a copy of subset called new_subset ()
            add edge to new_subset ()
            // To create a new unique powerset
            add new_subset to new_subsets ()

        add new_subsets to powerset {}


    create optimal_subset = ()
    make optimal_weight = -math.infinity

    for every subset in powerset
        create a graph copy G called temp_graph
        // remove the edges and then calculate the weight as well
        total_weight = 0
        for every edge in subset // subset can have multiple edges
            total_weight = total_weight + edge.weight
            remove edge from temp_graph

        count = Subgraphs_DPS(temp_graph)

        if count == n and count > 1 and total_weight > optimal_weight
        // if count != n move on to find the next iteration of edges
            optimal_weight = total_weight
            optimal_subset = subset

        // Remove minimal edges and leave n unconnected graph
    minimal_weight = 0
    for every edge in optimal_subset
        minimal_weight = minimal_weight + edge.weight
        remove edge from graph G
    Accept state with minimal_weight

Subgraphs_DPS(G)
    //Assume the DFS Has been implemented for the moment
    counter = 0
    for every vertex in G.vertices
        make all visit states of graph G's vertices to = False

    for every vertex in G.vertices
        if vertex.visited != True
            counter += 1
            DFS(vertex)
    return counter

```

5.3 Turing Machine

5.3.1 Turing Machine Description

A multi-tape Turing Machine that uses Brute-Force to find a solution of Graph G ⟨⟨G⟩, n⟩.

5.3.2 Powerset

Given a set **S**, the power set of **S** is the set that contains all subsets of **S**. The power set of **S** is denoted by P(S).

P(S) = { x|x is a subset of S }

|P(S)| = 2^n

Time complexity is O(2^n) because brute-force requires iterating over all possible combinations of edges (subsets) within the graph which results in exponential time affecting the overall time complexity of the algo2rithm.

5.3.3 Assumptions

1. Assume $ will have a certain string so that the machine knows it is within the language and will halt.
2. Assume '|' will guarantee an edge of the following: (Vertex 1, Vertex 2, Edge Weight)
3. Assume the input is in the format of $|V1,V2,EW|V1,V2,EW,n1 1$ The ones after n signify the value of n.
4. Assume graph Nodes cannot start with C, W, T and F. They will be used as counter, weights, true and false on certain tapes.

5.3.4 Tape specification and complexity

Tape	Tape Name	Tape Specification	Time Complexity
1	GRAPH_TAPE	This tape will be used to read Graph G ⟨⟨*G*⟩, *n*⟩.	O(n)
2	POWERSET_TAPE	Powerset recording all possible sets. Syntax is {()}{()}{()}{()}	O(2^n)
3	SUBSET_TAPE	Powerset Operations to calculate for new subsets and will be reset to contain the best set of edges.	O(2^{n-1})
4	TEMP_GRAPH_TAPE	Temporary graph for DFS Traversal and edge removals.	O(n)
5	DFS_HISTORY_TAPE	DFS Vertex history and counter. $**F**V1**T**V2,EW. The values before the vertices indicate if they have been visited. "|" and the comma will be modified to Boolean values representing vertex state.	O(n)
6	DFS_STACK_TAPE	This will be used as a stack for Depth-first search (DFS).	N/A

5.3.5 Algorithmic Framework

Due to the notion of brute-force, the powerset is generated which consists of all possible subsets of edges including the empty set.

5.3.5.1 Powerset Operation

The powerset initially assigns an empty set {} on POWERSET_TAPE and includes three phases:

1. **PHASE 1:** Copy all sets from POWERSET_TAPE to SUBSET_TAPE.
2. **PHASE 2:** Add current edge from GRAPH_TAPE to all subsets of SUBSET_TAPE.
3. **PHASE 3:** Add SUBSET_TAPE subsets to POWERSET_TAPE.

For example, consider the operation below executing the following edges (A,B) and (C,D):

1. **PHASE 1:** SUBSET_TAPE has the empty set {}
2. **PHASE 2:** {(A,B)}
3. **PHASE 3:** {}, {(A,B)}
4. **PHASE 1:** SUBSET_TAPE now has {}, {(A,B)}
5. **PHASE 2**: {(C,D)}, {(A,B), (C,D)}
6. **PHASE 3:** {}, {(A,B)}, {(C,D)}, {(A,B), (C,D)}

This is how the powerset is achieved resulting in a time complexity of O(2^n)**.**

Additionally, **SUBSET_TAPE** aids to create the powerset but won't have the time & space complexity of a powerset since the sets from **SUBSET_TAPE** are added to **POWERSET_TAPE** for every iteration, meaning, **SUBSET_TAPE** will have the final sets of the last node with all of the other nodes (look at the difference in phase 2 and phase 3). In the operation above, Phase 2 has {(C,D)}, {(A,B), (C,D)} and then phase 3 has {}, {(A,B)}, {(C,D)}, {(A,B), (C,D)}.

{(C,D)} {(A,B), (C,D)} was added to **POWERSET_TAPE**, therefore **SUBSET_TAPE** results in a time complexity of O(2^{n-1}).

5.3.5.2 Primary Algorithmic Procedure

The architecture of the main algorithm constitutes of 4 high-level schematic phases that are pivotal to find a solution for the problem.

PHASE 1: Find edges from GRAPH_TAPE and record them on TEMP_GRAPH_TAPE and DFS_HISTORY_TAPE.

PHASE 2: Computes edge weights based on POWERSET_TAPE subsets.

PHASE 3: Remove edge from TEMP_GRAPH_TAPE.

PHASE 4: Execute DFS to find the number of subgraphs.

PHASE 5: Check edge validity and update optimal subset.

5.3.6 Depth-first search blueprint

The depth-first search has been implemented in Turing Machine Algorithm whereas the pseudocode below is a summary of the DFS framework designed for readability due to the complexity and high-level description of DFS in the Turing machine pseudocode. Tape numbers have been used rather than their names for simplicity.

Algorithm 1 Iterative DFS with Turing Machine

```
1: Function Iterative_DFS_TM(Tape_5, Tape_6, Start_Vertex)
2: Push Start_Vertex onto Tape_6 (stack).
3: while Tape_6 (our stack) is not empty do
4:    Pop the top vertex "current_vertex" from Tape_6.
5:    Move Tape_5's head to the position of current_vertex.
6:    if "current_vertex" is not visited then
7:       Write "visited" next to current_vertex on Tape_5.
8:    end if
9:    for each adjacent vertex V do
10:      Move Tape_5's head to the position of V.
11:      Read the status of V from Tape_5.
12:      if V is not visited then
13:         Push V onto Tape_6 (stack).
14:      end if
15:   end for
16: end while
17: End Function
```

5.3.7 Complexity class

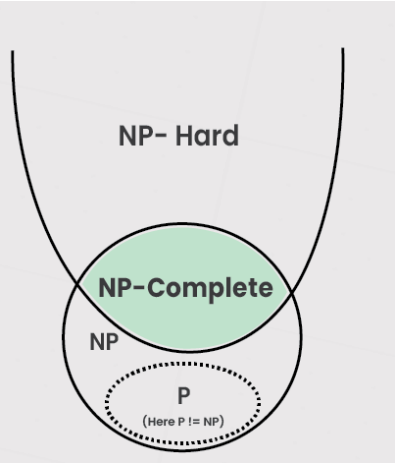

Figure 5.1

(geeksforgeeks, 2023)

The Tape that takes the most memory and time is the POWERSET_TAPE (Tape 2), as it takes in all the possible combinations of edges as a Brute-Force operation. The time and space complexity are O(2^n) Ω(2^n), and Θ(2^n). Its exponential complexity indicates that the Turing Machine Algorithm won't change significantly.

Since, the brute-force solution is exponential given a complexity of O(2^n) for the powerset, a non-deterministic, non-polynomial Turing Machine will solve this problem classified as NP-complete and in NP.

Turing Machine Algorithm

```
Check Format
Initialize tapes -> GRAPH_TAPE, POWERSET_TAPE, SUBSET_TAPE, TEMP_GRAPH_TAPE,
DFS_HISTORY_TAPE, DFS_STACK_TAPE
// POWERSET
WRITE {} to POWERSET_TAPE.head //Empty set
MOVE POWERSET_TAPE.head to the right
WRITE ',' to POWERSET_TAPE.head
RESET GRAPH_TAPE.head to the left most position // To find the links from beginning
MOVE GRAPH_TAPE.head to the right
if GRAPH_TAPE.head.value == "$":
    continue the Algorithm
else: REJECT STATE as there are no strings in the language
    // $ means that is the start of the string
if GRAPH_TAPE is not Null && != n: //the n value is not needed yet.
    MOVE GRAPH_TAPE.head to the right
    for every "|" on GRAPH_TAPE in |^x:
    // Read current edge from tape 1 >> can only be done one at a time
        RESET POWERSET_TAPE.head to the left most position
        // PHASE 1: Copies all sets from tape 2 to tape 3
        While POWERSET_TAPE is not Null:
            for each cell in POWERSET_TAPE:
                MOVE POWERSET_TAPE.head to the right
                WRITE POWERSET_TAPE.head.value to SUBSET_TAPE
                MOVE SUBSET_TAPE.head to the right

        // PHASE 2: This while loop will add the current edge of tape 1 to every subset of
tape 3
        while SUBSET_TAPE READS all '}' in }^y:
            // Add current edge of tape 1 to every subset on tape 3 for e.g. (V1,V2,EW)
            // empty set will be ignored
            MOVE SUBSET_TAPE.head to the left && WRITE ',' to SUBSET_TAPE.head if not {}
            MOVE SUBSET_TAPE.head to the right // currently still reading '}'
            // NOW tape 3 has the correct syntax such as {(SomeRandomEdge),}

            WRITE '(' to SUBSET_TAPE.head && MOVE SUBSET_TAPE.head to the right
            for 5 times COMPUTE: //every 5 cells including commas have a new edge -> V1,
V2,EW
                MOVE GRAPH_TAPE.head to the right
                // for e.g. Tape 1 has V1,V2,EW and Tape 2 and tape 3 has {}
                WRITE GRAPH_TAPE.head.value to SUBSET_TAPE.head
                // for e.g. Tape 3 now has {(A,B,1)}
                MOVE SUBSET_TAPE.head to the right
                // No need to worry about ',' because tape 3 is temporary
                // Tape 3 will always be heading towards '}' so the loop will still
iterate
            WRITE ')' to SUBSET_TAPE.head && MOVE SUBSET_TAPE.head to the right

        // PHASE 3: WRITING back all the new sets to tape 2 for e.g.{(A,B,1)}
        MOVE POWERSET_TAPE.head to the right
        RESET SUBSET_TAPE.head to the left most position
        for every "{" on SUBSET_TAPE:
            for 8 times COMPUTE:
            // every 8 times there is an edge, 8 because of syntax
            // for example Tape 2 now has {}, {(A,B,1)}
            // 1 -> ( 2 -> A -> 3 -> , 4 -> B 5 -> , 6 -> 1 -> 7 ) -> 8 -> }
                MOVE SUBSET_TAPE.head to the right
                WRITE SUBSET_TAPE.head.value to POWERSET_TAPE.head
                MOVE POWERSET_TAPE.head to the right
```

```
            // Comma is gonna be skipped by the loop because of '{'

for every cell on SUBSET_TAPE:
    // We can reuse this tape to store the best set of edges
    RESET SUBSET_TAPE.head.value to blank
    RESET SUBSET_TAPE.head to the left most position

// Primary Algorithmic Procedure

// Tape 2 is currently pointing to the last set of the powerset
// We copied all the data from tape 3 last time
MOVE POWERSET_TAPE.head to the right && WRITE 'W' to POWERSET_TAPE.head.value
// W indicates weights that we will be calculating and storing
RESET POWERSET_TAPE.head to the left most position
for every subset {}^i on POWERSET_TAPE:
    //PHASE 4: Finding edges and recording them as a temporary graph on Tape 4
    RESET GRAPH_TAPE.head && DFS_HISTORY_TAPE.head to the left most positicn
    for every cell on GRAPH_TAPE:
        MOVE GRAPH_TAPE.head to the right
        //copying the graph to TEMP_GRAPH_TAPE
        WRITE GRAPH_TAPE.head.value to TEMP_GRAPH_TAPE.head
        MOVE TEMP_GRAPH_TAPE.head to the right
        //copying the graph to DFS_HISTORY_TAPE to keep track of vertex.visited history
        WRITE GRAPH_TAPE.head.value to DFS_HISTORY_TAPE.head
        MOVE DFS_HISTORY_TAPE.head to the right

    for each '(X,Y,EW)' edge in subset(skipCells=6): //skip every 6 cells for new edge
         // PHASE 5: Computing Edge Weights
         // subset can have multiple edges due to powerset nature.
        RESET POWERSET_TAPE.head to the leftmost position
        RESET TEMP_GRAPH_TAPE.head to the left most position // for traversing operations
        for every "W" in cells^Z ON POWERSET_TAPE:
        // Z is 1 because we have only 1 W that represents the start of edge weight counts
            if POWERSET_TAPE.head.value == 'W' ||  POWERSET_TAPE.head.value == '_': //
total weight will be after
                MODIFY POWERSET_TAPE.head.value to "@" // so that the next iterations don'
t pick W (only pick blank)
                MOVE POWERSET_TAPE.head to the left Z times // now we have the edge weight
                // Because we are doing 2 things (UNBOUNDED) on the tape we will need to
read the data
                READ POWERSET_TAPE.head.value as edgeWeight // record that cell as a
reference
                MOVE POWERSET_TAPE.head to the right Z+1 times // Z+1 is after W to write
data
                // Currently the head is blank

                // head is pointing to where it needs already
                for (POWERSET_TAPE.head.value + edgeWeight) times COMPUTE:
                    // total_weight = total_weight + edge.weight
                    //the head is null (0) so 0 + EW is valid
                    WRITE 1 to POWERSET_TAPE.head.value
                    MOVE POWERSET_TAPE.head to the right
            // for the loop to continue for the next subset
            MOVE POWERSET_TAPE.head to the left Z + 1 + (POWERSET_TAPE.head.value +
edgeWeight) times
            MOVE POWERSET_TAPE.head 2 times to the right
            // to skip the extra brackets '),(' in {(A,B,1),(A,B,1)} then the loop will
iterate again 6 times

        // PHASE 6: Let's remove the edge
        MOVE POWERSET_TAPE.head 2 times to the left // because after the last subset,
there aren't any brackets
```

```
        // Now we have the edge weight value from the subset of a powerset
        MOVE TEMP_GRAPH_TAPE.head to the right // Skipping $
        // The point of this loop is to find a specific edge
        for each '|V1,V2,EW' in edges on TEMP_GRAPH_TAPE(skip=6): // it will skip the
cells 6 times
            // comparing the edge weights
            if TEMP_GRAPH_TAPE.head.value == POWERSET_TAPE.head.value:
                // To have the value of the first verticies
                MOVE POWERSET_TAPE.head && TEMP_GRAPH_TAPE.head to the left 4 times
                    // Comparing first verticies
                    if TEMP_GRAPH_TAPE.head.value == POWERSET_TAPE.head.value:
                        // To have the value of the Second verticies
                        MOVE POWERSET_TAPE.head && TEMP_GRAPH_TAPE.head to the right 2
times
                        // Comparing Second verticies
                        if TEMP_GRAPH_TAPE.head.value == POWERSET_TAPE.head.value:
                            // removing the edge
                            MOVE TEMP_GRAPH_TAPE.head to the left 3 times // finding the
start of the edge '|'
                            for every cell till first "|" on TEMP_GRAPH_TAPE: // Terminate
once '|' has been seen
                                EMPTY TEMP_GRAPH_TAPE.head

    // PHASE 7: Depth-First Search
    RESET TEMP_GRAPH_TAPE.head && DFS_HISTORY_TAPE.head to the left most position
    MOVE TEMP_GRAPH_TAPE.head && DFS_HISTORY_TAPE.head to the right // skip $
    FOR every cell on DFS_HISTORY_TAPE:
     // Instead of "|" which indicates the start of a string, we put "F" as its vertex
history
        if cell == "|":
            MODIFY DFS_HISTORY_TAPE.head.value to "F" //Vertex 1 visit history of edge 1
        elif cell == ",": //so it will skip the commas before EW to account for the next
edge
            MODIFY DFS_HISTORY_TAPE.head.value to "F" //Vertex 2 visit history of edge 1

    MOVE DFS_HISTORY_TAPE.head to the rightmost position + 1 // Move on to the next unused
cell
    WRITE "C" to DFS_HISTORY_TAPE.head // To indicate the start of a counter
    FOR each '|V1,V2,EW' edge^w on TEMP_GRAPH_TAPE(skip=6):
        FOR each "F" in cells^Y on DFS_HISTORY_TAPE: // Considers each visited vertex
            IF DFS_HISTORY_TAPE.head.value == "F": // If vertex.visited is False
                MODIFY DFS_HISTORY_TAPE.head.value to "T" // Update Vertex.visited to True
                FOR every cell in cells^D on DFS_HISTORY_TAPE:
                    // Increment Counter
                    IF cell == "C": // There is only one 'C', so skip the cells to find C
                    //There could be ones in there already
                        FOR every cell in cells^G on DFS_HISTORY_TAPE(until=_): //find the
empty cell
                            MOVE DFS_HISTORY_TAPE.head to the right //Move on to the next
unused cell
                            WRITE 1 to DFS_HISTORY_TAPE.head // Keep track and
incrementing count with 1's
                MOVE DFS_HISTORY_TAPE.head to the left D + G times // RESET head for
future use
                RESET GRAPH_TAPE.head to the left most position
                // USING TAPE 1 HERE to not complicate other tapes
                MOVE GRAPH_TAPE.head to the right 3 times // to get the start vertex
                // Push Start_Vertex onto DFS_STACK_TAPE (our stack). start_vertex is just
the first edge
                WRITE (GRAPH_TAPE.head.value) to DFS_STACK_TAPE.head
                // MOVE DFS_STACK_TAPE.head to the right prolly don't need it
                RESET GRAPH_TAPE.head to the left most position

```

```
                // NOTE: we only traversed Y times so we have not lost track
                RESET DFS_HISTORY_TAPE.head to the leftmost position
                // NOTE: we only traversed w times so we have not lost track
                RESET TEMP_GRAPH_TAPE.head to the leftmost position

                while DFS_STACK_TAPE is ! Null: // ! is 'not' // While stack is not empty
                    // for the following 2 loops, we will either find vertex 1 or vertex 2
                    for each "|" on DFS_HISTORY_TAPE:
                        for each "|" on GRAPH_TAPE:// we have start vertex from tape 1
                            MOVE DFS_HISTORY_TAPE.head && GRAPH_TAPE.head to the right//
Finding V1
                            // IF tape's 5 vertex 1 == tape's 1 vertex 1
                            if DFS_HISTORY_TAPE.head.value == GRAPH_TAPE.head.value:
                                    // We now have our current vertex position
                                    continue the Algorithm
                            else: // if it cannot find the current vertex from vertex 1
then find it from the second vertex
                                    // for e.g (V1,V2,EW) is our edge, our current vertex
could be in either V1 or V2
                                RESET DFS_HISTORY_TAPE.head && GRAPH_TAPE.head to the
leftmost position
                                for each "," on DFS_HISTORY_TAPE:
                                    for each "," on GRAPH_TAPE:// we have start vertex
from tape 1
                                        MOVE DFS_HISTORY_TAPE.head && GRAPH_TAPE.head to
the right// Finding V2
                                        // IF tape's 5 vertex 2 == tape's 1 vertex 2
                                        if DFS_HISTORY_TAPE.head.value ==
GRAPH_TAPE.head.value:
                                                // We now have our current vertex position
                                                continue the Algorithm
                    // If "current_vertex" is not visited:
                    //I am only checking the status of the first vertex of an edge, not
the second yet
                    IF DFS_HISTORY_TAPE.head.value == "F":
                        MODIFY DFS_HISTORY_TAPE.head.value to "T" //Mark it as visited

                    // Pop the top vertex "current_vertex" from DFS_STACK_TAPE
                    MOVE DFS_STACK_TAPE.head to the rightmost position
                    EMPTY DFS_STACK_TAPE.head.value to blank // Pop the vertex
                    ADJUST DFS_STACK_TAPE.head // This will make sure that the right-most
position is recalculated

                    RESET DFS_HISTORY_TAPE.head to the leftmost position
                    // Read all adjacent vertices of current_vertex
                    for every cell on DFS_HISTORY_TAPE:
                        // tape 1 has the current vertex that guides this part
                        if DFS_HISTORY_TAPE.head.value == GRAPH_TAPE.head.value:
                            // Reading the second vertex visited history
                            MOVE DFS_HISTORY_TAPE.head to the right  // To read the status
of the adjacent vertex
                            // If V is not visited:
                            if DFS_HISTORY_TAPE.head.value == "F":
                                // Push V onto Tape_D (our stack).
                                WRITE (DFS_HISTORY_TAPE.head.value) to DFS_STACK_TAPE.head
                                // END of DFS

            //Move head for the next iterations of loop
            MOVE DFS_HISTORY_TAPE.head to the left Y times
        MOVE TEMP_GRAPH_TAPE.head to the left W times

    // PHASE 8: Checking edge validity and Updating Optimal subset
```

```
    RESET GRAPH_TAPE.head to the left most position

    MOVE GRAPH_TAPE.head to the right until (GRAPH_TAPE.head.value == "n") // START OF N
    MOVE DFS_HISTORY_TAPE.head to the right until (GRAPH_TAPE.head.value == "C") // START OF COUNTER

    //currently counting
    for every cell on DFS_HISTORY_TAPE:
        // checking compatibility
        MOVE GRAPH_TAPE.head to the right  //stores n
        MOVE DFS_HISTORY_TAPE.head to the right // stores count
    // if n == count
    if GRAPH_TAPE.head == 1 && DFS_HISTORY_TAPE.head == 1:
        MOVE DFS_HISTORY_TAPE.head to the left
        // count > 1
        if DFS_HISTORY_TAPE.head != "@" // then count is greater than 1
            // anything after @ on tape 2 is the total weight
            MOVE POWERSET_TAPE.head to the right till "@" is found
            for every cell on DFS_HISTORY_TAPE:
                MOVE DFS_HISTORY_TAPE.head to the right // Serves as total_weight
                MOVE SUBSET_TAPE.head to the right // Tape 3 will serves as our optimal weight
            MOVE DFS_HISTORY_TAPE.head to the right
            // total_weight > optimal_weight then

            if DFS_HISTORY_TAPE.head == 1: // A continous number of ones signified the total weight
            // We don't need to care about the weight because it is in the set and we have a poweset
                EMPTY SUBSET_TAPE to blank for all cells // RESET tape to store optimal subset
                // store the subset
                for each '(X,Y,EW)' edge in subset {}:
                    //optimal subset
                    WRITE edge subvalues (X,Y,EW) to SUBSET_TAPE.head.value && MOVE SUBSET_TAPE.head to the right
                    WRITE "," to SUBSET_TAPE.head.value && MOVE SUBSET_TAPE.head to the right
            else:
                continue the Algorithm

    // RESETINGtape 2's head for the next iteration
    MOVE POWERSET_TAPE to next subset {} i

ACCEPT STATE on SUBSET_TAPE // Tape 3 has the optimal subset with minimal weights
```

5.4 Time and space complexity

The image below is a summary of all the important lines of computation that take significant time complexity excluding constants throughout the whole program.

```
1     for every "|" on GRAPH_TAPE in |^x: //O(n)
2         While POWERSET_TAPE is not Null: // O(2^n)
3         while SUBSET_TAPE READS all '}' in }^y: // O(2^n-1)
4         for every "{" on SUBSET_TAPE: // O(2^n-1)
5 for every cell on SUBSET_TAPE: // O(2^n-1)
6 for every subset {}^i on POWERSET_TAPE: // O(2^n)
7     for every cell on GRAPH_TAPE: //O(n)
8     for each '(X,Y,EW)' edge in subset(skipCells=6)://O(n)
9         for every "W" in cells^Z ON POWERSET_TAPE: // O(2^n)
10                MOVE POWERSET_TAPE.head to the right Z+1 times //O(Z+1)
11        for each '|V1,V2,EW' in edges on TEMP_GRAPH_TAPE(skip=6): //O(n)
12            for every cell till first "|" on TEMP_GRAPH_TAPE: //O(n)
13    FOR every cell on DFS_HISTORY_TAPE: //O(n)
14    FOR each '|V1,V2,EW' edge^w on TEMP_GRAPH_TAPE(skip=6): //O(n)
15        FOR each "F" in cells^Y on DFS_HISTORY_TAPE: //O(n)
16                FOR every cell in cells^D on DFS_HISTORY_TAPE://O(n^2)
17                MOVE DFS_HISTORY_TAPE.head to the left D + G times //O(D) + 0(G)
18                for each "|" on DFS_HISTORY_TAPE: //O(n)
19                    for each "|" on GRAPH_TAPE: //O(n)
20                            for each "," on DFS_HISTORY_TAPE: //O(n)
21                                for each "," on GRAPH_TAPE: //O(n)
22                for every cell on DFS_HISTORY_TAPE:  //O(n)
23            MOVE DFS_HISTORY_TAPE.head to the left Y times  //O(Y)
24        MOVE TEMP_GRAPH_TAPE.head to the left W times //O(W)
25    MOVE GRAPH_TAPE.head to the right until (GRAPH_TAPE.head.value == "n") //O(n)
26    MOVE DFS_HISTORY_TAPE.head to the right until (GRAPH_TAPE.head.value == "C") //O(n)
27    for every cell on DFS_HISTORY_TAPE://O(n)
28            for every cell on DFS_HISTORY_TAPE: //O(n)
29                EMPTY SUBSET_TAPE to blank for all cells //O(n)
30                for each '(X,Y,EW)' edge in subset {}: //O(n)
```

5.4.1 Analysis

(O(n) × (O(2^n) + O(2^{n-1}) + O(2^n 1)))

+ (O(2^n) × (O(n) + (O(n) × [O(2^n) × O(Z+1)] + (O(n) + O(n)) + O(n) + O(n)

× [[O(n) × (O(n) × O(n)) + O(D + G) + [O(n) + O(n) + O(n) + O(n)] + O(n) + O(Y))))] + O(W)]

+ O(n) + O(n) + [O(n) x O(n) x O(n) x O(n)]))

= O(n x 2^n) + (O(2^n) × (O(n) × [O(2^n) × O(Z+1)]) + (O(n) x O(n^2)) + O(n^4))

5.4.2 Simplified Time Complexity

O(n x 2^n) + (O(2^n) × (O(n) × [O(2^n) × O(Z+1)]) + (O(n) x O(n^2)) + O(n^4))

= O(n x 2^n) + (O(n x 2^n) + O(n^2x Z) + O(n^3) + O(n^4))

= O(n x 2^n) + O(n x 2^n)

= 2 O(n x 2^n)

= O(n x 2^n)

5.4.3 Space complexity

Tapes	Tape Name	Space Complexity
1	GRAPH_TAPE	O(n)
2	POWERSET_TAPE	O(2^n)
3	SUBSET_TAPE	1. Powerset operation = O(2^{n-1}) 2. Best set of edges after reformatting the tape = O($\|E\|$)
4	TEMP_GRAPH_TAPE	O(n)
5	DFS_HISTORY_TAPE	O(n)
6	DFS_STACK_TAPE	O(n)

Note: **SUBSET_TAPE** was used to help create the powerset. After that operation is done, **SUBSET_TAPE** is not needed anymore. Hence, it was reformatted and used to store the best set of edges resulting in the highest remaining edges. Hence, Space complexity shows 2 space complexities. It used 2^{n-1} for the powerset operation and then, used $|E|$ for the best set for edges.

Since the Turing Machine has a time complexity of *O(n x* 2^n*),* It only accepts the state on "SUBSET_TAPE" where the worst case for space complexity is O($|E|$), where E is the number of edges in the subgraph. In a scenario where all the edges have the same weight, the algorithm has no choice but to select all edges deemed optimal.

5.5 Decidability

In terms of input and output, the algorithms provided receive a finite graph G including a finite integer n of which are a finite set of information to the Turing Machine. The Turing Machine at its early stage of its algorithmic operation acts as a lexical analyser that converts the stream of characters into tokens and uses certain predefined assumptions (for e.g. $ is either the start or end of the string and | is the start of an edge) to decide the semantics of the input at certain locations of the string and allocates whether the character is a vertex or edge weight or combines all together makes the edge between two vertices. Its specific manual input parameter stream and the finite set guarantees that there is an end to the input and the process since there is only a maximum of $2^{|E|}$ edges with a complexity of O(n x 2^n), Ω(n x 2^n), and Θ(n x 2^n).

The algorithm uses the graph G with its data and attempts to remove a set of edges that will altogether weight as the most minimal by Brute-Forcing through and since there is a well systematic procedural design, the Turing machine will halt for every character and also only after it evaluates 2^nsubsets where n is the number of edges in a graph therefore proving it is decidable. ("I" assume that special symbols and comma won't take as much time in comparison to important input such as characters and ones) therefore the time complexity

is still 2^n rather than 2^{n*6} (because there are 6 characters in the input as decided initially for e.g., |V1,V2,EW.

6 Grouping solution

6.1 General Idea

- Instead of simply brute forcing the problem, dynamic programming can be used to cache intermediate results for later use, reducing the overall number of computations that need to be completed.
- The grouping approach is as follows. Given the input graph, merge two connected nodes, with the new node having the weight of those nodes, plus the weights of any edges connecting them. Any edges terminating at the original nodes now terminate at this new node.
    - For the input graph, the grouping solution will perform every possible merge, then from there perform every possible merge of those as well.
    - At each step, store the results and check if a graph has already had its merges calculated to save regenerating them.
- Conceptually, the merged nodes represent a subgraph of the shipping network, and any remaining edges are those which will be pruned.
    - As per the initial assumptions that no node can be isolated, the grouping approach must continue merging until there are no nodes with a weight of 0 remaining, because that indicates that all nodes now represent a sub graph.
    - When the number of remaining nodes after merging is the number of subgraphs that must be generated as per the initial problem, tally up the total weight of the nodes. This is the value that is being maximised.
    - If no appropriate solution is found after all possible merge combinations have been considered, then reject.
    - If a solution has been found, accept, since the solution is updated each time a more optimal alternative is found.
- If checking a graph and there are less nodes than the amount of sub graphs to create, it indicates that the graph has been merged too many times, and no solution will be found, so stop checking further.
- The following pseudo code outlines this process in more detail and was used as a planning tool to implement the Turing machine design.

6.2 Pseudo Code

```
Given the input graph G, and the number of disjoint subgraphs to create is n

If count of nodes in G < 2 * n
    Fail, since there are not enough nodes for all of them to merge

If count of edges in G < n
    Fail, since there are not enough edges to merge everything properly

If any unconnected nodes in graph G
    Fail, invalid graph, see assumptions

Let the set of all checked graphs be V
Let the set of all graphs to check be W
Let the best known merged graph be Z

Execute Function PossibleMerges G

//  Main loop
while W has a graph
    Remove last graph of W and place it in X

    If X is not in V
        V += X

        If count of nodes in X > n
            Execute Function PossibleMerges X

        Else If count of nodes in X == n and no node of X has a weight of 0
            If Z.NodeWeight < X.NodeWeight
                Z = X

        //  Otherwise, do nothing. The current graph either has too few nodes,
        //  or the right number of nodes but one has a value of 0,
        //  neither of which will ever produce a solution.

Output result Z

//  Note, graphs may have multiple edges between the same nodes (after merging)
Function PossibleMerges X
    Foreach node A of X
        Foreach node B of X
            If A != B and A is connected to B
                Create Y, a copy of X

                //  Create a new node which is merged from the two nodes its
replacing
                Create node C within Y
                Set weight of node C as A.weight + B.weight

                //  Update all the edges to go to the newly merged node
                Foreach edge F in X
                    If F connected to A and B
                        C.weight += F.weight


                    Else if edge connected to A
                        Add a copy of edge F to Y, but replace its termination at
node A with node C

                    Else if edge connected to B
                        Add a copy of edge F to Y, but replace its termination at
node B with node C

                    If F connected to A or B
                        Remove edge F from Y

                Remove node A and B from Y

                W += Y
```

6.3 Turing Machine

```
This describes the data format I will use for Turing machines henceforth.
Integers are a continuous string of 1, the quantity is the integer value.
Graphs are formatted as nodes, then edges. Nodes have an ID and weight, and all of the node IDs combined to get them in the first place. Edges have two node IDs that they connect to, and a weight.
#:ID,WEIGHT[]:ID,WEIGHT[]|NODE_ID_1,NODE_ID_2,WEIGHT|NODE_ID_1,NODE_ID_2,WEIGHT
Example below, indicates a triangle graph with nodes 1, 2 and 3 all of weight 0. Edges 1 <-> 2 weighted 4, 2 <-> 3 weighted 2, 1 <-> 3 weighted 3
#:1,[]:11,[]:111,[]|1,11,1111|11,111,11|1,111,111

Input graph is in the format as defined, preceded by the number of disjoint subgraphs to create (which is using the defined integer format)
Using a multitape turing machine, with the following tapes and their names used:
1. INITIAL (initial tape),
2. GRAPH (G and X),
3. GRAPH_COPY (PossibleMerges -> X but a copy of it),
4. GRAPH_COPY_TWO (PossibleMerges -> X but a copy of it),
5. COUNT (n),
6. SOLUTION (Z),
7. ALL_GRAPHS (V),
8. POSSIBLE_MERGES (W),
9. WORKING_GRAPH (PossibleMerges -> Y)
Note the values in the brackets refer to the equivalents from the original pseudo code for better clarity.

The following is the Turing Machine algorithm to solve the problem

// Used to detect the start of the tape
On all tapes, move head 1 space left and place $

Check input format, then return head to $

Copy the sub graph size from INITIAL to COUNT (both scanning right)

Copy the input graph from INITIAL to GRAPH (both scanning right)

Reset GRAPH and COUNT to start (scan left until $)

// Check there are enough nodes present
Compare count of nodes (char :) in GRAPH with two times the size of COUNT
    If nodes is smaller than twice COUNT reject
    Otherwise reset GRAPH and COUNT to start

// Check there are enough edges present
Compare count of edges (char |) in GRAPH with the size of COUNT
    If nodes is smaller than COUNT reject
    Otherwise reset GRAPH and COUNT to start

// Check that there are no isolated nodes, which cannot be merged with anything
Copy GRAPH to WORKING_GRAPH
Scan right through each node in GRAPH
    Reset WORKING_GRAPH to start
    Scan right through WORKING_GRAPH and find an edge connected to current node in GRAPH
        If not found then reject

```

```
51  //  Prepare to run the main loop
52  //  Note, uses the value in GRAPH to build the derivatives
53  Run the routine PossibleMerges
54
55  //  Main loop of the machine
56  While graphs (char #) exists in POSSIBLE_MERGES
57      Copy last graph from POSSIBLE_MERGES to GRAPH
58      Clear the last graph from POSSIBLE_MERGES
59
60      Check if ALL_GRAPHS contains GRAPH (this is done by comparing the weights and
    sorted sub nodes of each node to determine if the graphs contain effectively the
    same data)
61          If it does then
62              Copy GRAPH to after the last graph of ALL_GRAPHS
63
64              Compare count of nodes in GRAPH with COUNT
65              Check if a node in GRAPH has a weight of 0
66                  If more nodes than COUNT
67                      Run the routine PossibleMerges
68
69                  Otherwise if number of nodes in GRAPH and COUNT are equal and no
    node has a weight of 0 then
70                      Compare the total node weight of SOLUTION and GRAPH
71                          If LHS is less then
72                              Clear SOLUTION
73                              Copy GRAPH to SOLUTION
74
75
76
77  // Final component, check that an answer was found
78  Check SOLUTION contains a graph
79      If has a graph then accept
80      Otherwise reject
81
82  //  Effectively a function to generate all the possible merges of a graph
83  Define the routine PossibleMerges as follows
84      CLEAR GRAPH_COPY
85      Copy GRAPH to GRAPH_COPY
86
87      Scan right through the nodes in GRAPH, foreach
88          Scan right through the nodes in GRAPH_COPY, foreach
89              Compare the node IDs of the current node in GRAPH and GRAPH_COPY
90                  If not equal then
91                      Clear WORKING_GRAPH
92                      Copy GRAPH to WORKING_GRAPH
93
94                      Scan right through the edges in WORKING_GRAPH
95                          If A and B are connected then
96
97                              Insert a new node into WORKING_GRAPH with the next ID
98                                  Weight is the sum of the weights of the current
    node in GRAPH and GRAPH_COPY
99                                  Sub nodes of GRAPH node ID, GRAPH_COPY node ID and
    both of their sub nodes.
100                                 This step will require shifting all the edges
    right at the same time as inserting the new node
101
102                             Clear GRAPH_COPY_TWO
103                             Copy GRAPH to GRAPH_COPY_TWO
```

```
104
105                             Scan right through the edges in GRAPH_COPY_TWO,
foreach
106                                 Check if edge at GRAPH_COPY_TWO is connected to
node at GRAPH
107                                 Check if edge at GRAPH_COPY_TWO is connected to
node at GRAPH_COPY
108                                     If connected to both then
109                                         Find the last node in WORKING_GRAPH
110                                         Add the weight of the edge at
GRAPH_COPY_TWO to the node (will need to move other cells over)
111                                     Else If connected to GRAPH then
112                                         Create copy of edge at GRAPH_COPY_TWO in
WORKING_GRAPH, but replace the endpoint at the node of GRAPH with the ID of the
last node of WORKING_GRAPH
113                                     Else If connected to GRAPH_COPY then
114                                         Create copy of edge at GRAPH_COPY_TWO in
WORKING_GRAPH, but replace the endpoint at the node of GRAPH_COPY with the ID of
the last node of WORKING_GRAPH
115
116                                     If connected to either then
117                                         Remove the edge at GRAPH_COPY_TWO from
WORKING_GRAPH, making sure to move any cells to the right over to fill the void
118
119
120                             From WORKING_GRAPH find the node at GRAPH, and clear
it, shuffling all cells to the right over to fill the void
121
122                             From WORKING_GRAPH find the node at GRAPH_COPY, and
clear it, shuffling all cells to the right over to fill the void
123
124                             Copy WORKING_GRAPH to POSSIBLE_MERGES after the last
graph already there, while doing so sort the node IDs of each node, and then the
nodes themselves by their node IDs
```

6.4 Decidability

This Turing machine is a decider and will eventually halt in all cases. The main program loop has a finite stack to process. For each element in this stack, it can either have been seen before, or never seen. If it has never been seen before, it will be marked as seen and its derivatives will be placed onto the stack, causing the stack to grow in size. However, because of the set keeping track of already seen graphs and due to the nature in which derivatives are created, there is a finite number of unique derivatives which will eventually be exhausted, at which point the stack will cease to grow, allowing the main loop to finish processing and make a final decision to accept or reject.

6.5 Time and Space Complexity

The worst case for the merging algorithm is if the input graph is a clique, in which case the total number of merges approaches the Bell number, depending on the number of sub graphs to create.

The bell number can be calculated with the following sequence equation.

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$$

[10]

The main loop of the Turing machine loops over all possible merges of the graph. Within that loop, it runs a check against the number of existing graphs, hence the time complexity is the number of merges squared. Hence, the time complexity is $O(B(n^2))$, $\Theta(B(n^2))$, $\Omega(B(n^2))$

The space complexity is the number of graphs being stored. The ALL_GRAPHS tape stores all graphs that have been generated, which is the number of merges, $O(B(n))$, $\Theta(B(n))$, $\Omega(B(n))$.

7 Naming

This section will mention similar problems and issues to the breakup problem.

7.1 Breakup Problem

There appears to be a problem that is close enough to be considered like the breakup problem. It is called the minimum k-cut. There are also similar problems that will be discussed.

7.2 Minimum k-cut Problem

The minimum ***k***-cut problem is when given a graph and integer ***k***, delete the minimum weight set of edges such that the graph has at least ***k*** connected components. [7] and [11]

Most variations bring forth the idea of a source ***s*** and sink ***t*** of a cut [7] and [11]. A cut must start from ***s*** and end at ***t*** [7] and [11]. ***s*** and ***t*** can be placed anywhere on a graph and a line must connect ***s*** and ***t*** together. The constraints of the line are such that the line cannot cut over an already-cut edge or cross through a node. [Fig 7.1] shows an example of minimum k-cut where k = 2.

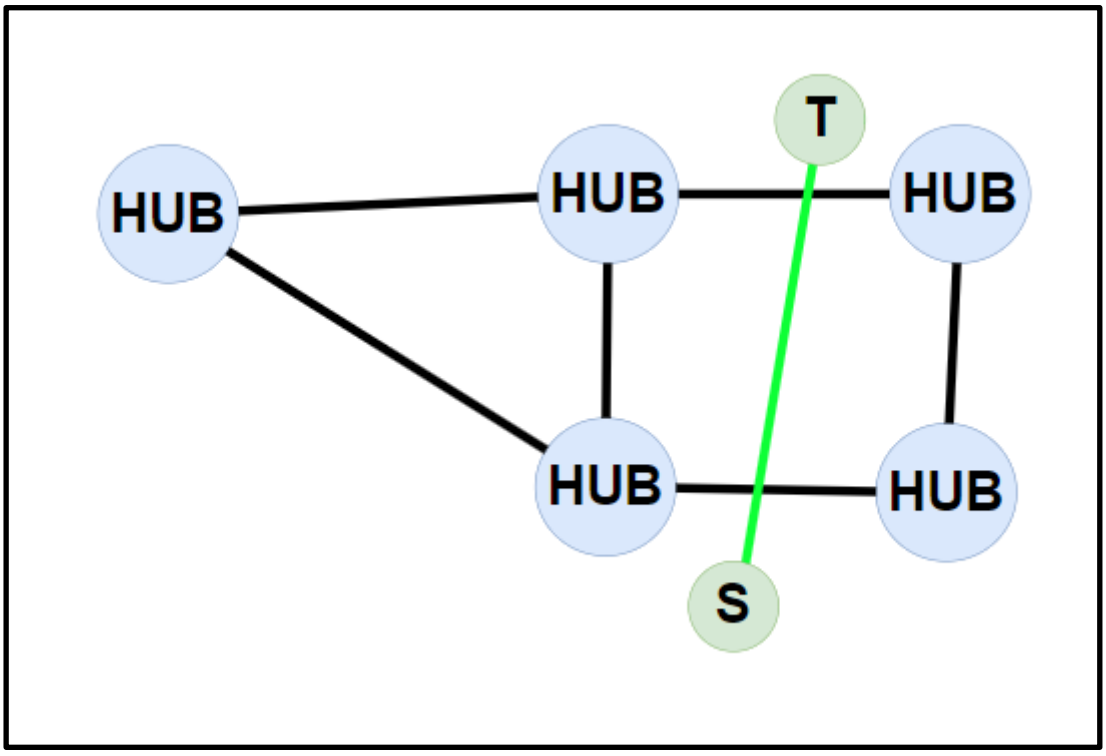

Fig. 7.1. Example of Minimum ***k***-cut where ***k*** = 2. Assume all edge weights are the same.

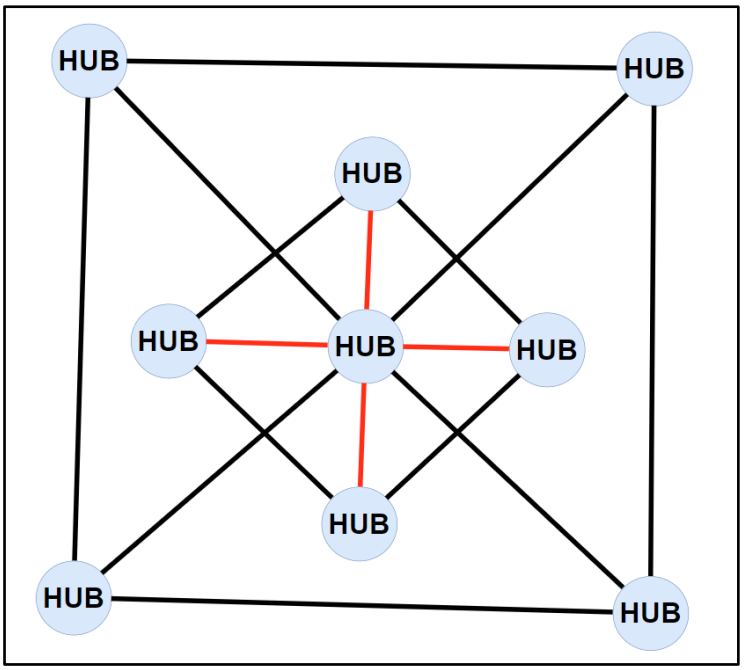

Fig. 7.2. Example of minimum ***k***-cut where it is impossible to cut red lines where ***k*** = 2. Assume the set of red lines is the minimum weight set.

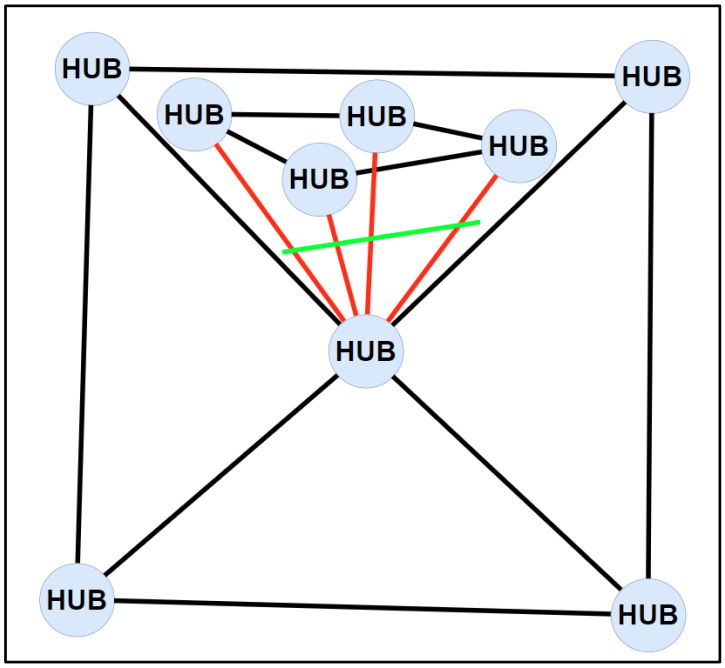

Fig. 7.3. How [Fig. 7.2] can be rearranged to make a cut possible

This is almost identical to the problem at hand. In our graph problem, we can select whatever edges we want to remove without having to adhere to the ***s*** and ***t*** constraints.

As seen in [Fig. 7.2], you can have a graph where you may not be able to cut a specific set of edges without cutting another edge. However, in [Fig. 7.3], you may be able to rearrange the nodes in such a way that the cut is now possible.

This problem is one of the closest to the breakup problem. A potential limiting factor is whether every graph can be rearranged so each set of edges can be cut without cutting an edge not in the set. If any graph can be rearranged, our problem can be treated as the minimum k-cut problem.

For the breakup problem, it will be treated as the minimum k-cut problem.

7.3 Liner Hub-and-Spoke Shipping Network Design

Most liner shipping networks follow the Hub-and-Spoke model with direct shipping, consisting of hub ports and feeder ports [12]. Hubs are connected to other hubs via a mainline service [12], [13]. Feeder ports connect to only the nearest hub port [3]. Liner vessels transport the bulk of cargo between hubs [13] and feeder vessels supply nearby feeder ports from its associated hub [3]. [Fig. 7.4] displays an example of a hub-and-spoke network.

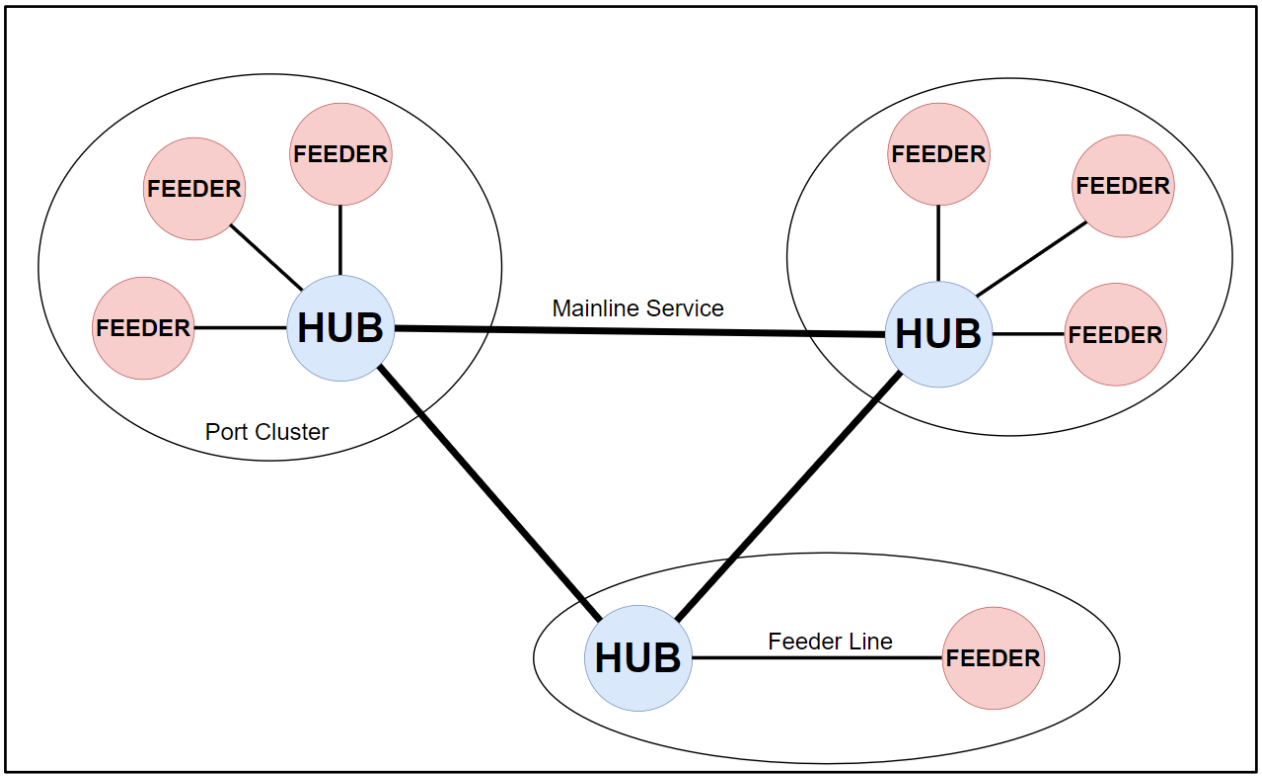

Fig. 7.4. Hub-and-spoke network example.

When making liner shipping networks, a sub-problem that must be considered is the aggregation of ports into a multi-hub network [3]. The steps to solve this sub-problem include assigning hub ports, connecting feeder ports to a hub, and determining inter-hub routes [3].

Each port cluster could be represented as a disconnected component of the input graph. However, the solutions provided in this report do not consider the designation of ports (hub or feeder), hence possibly resulting in a point-to-point network rather than a multi-hub network. Regardless, the solution would separate the graph into ***n*** port clusters where overall graph weight is maximised.

The solution from our algorithm could be used later to centralise each port cluster to find the optimal hub to connect all hubs together. Afterwards, inter-hub connections could be created to connect the graph. Therefore, the solution could assist in solving the port aggregation sub-problem by first breaking down an entire provided network.

However, considering feasibility in a real-life context, almost always a shipping network cannot be represented as an undirected graph while maintaining its usefulness [14]. Additionally, a point-to-point network may result in transhipping. Although transhipping may be more efficient for transportation in some cases, it has a significant negative effect on cargo value [15]. Transhipping is often used for lesser supply routes where the cost of setting up direct routes from the hub(s) is too costly or impossible, or cargo value is not a significant factor [15].

7.4 Graph Decomposition

Graph decomposition is a very similar problem. The graph is separated into ***n***-connected components where edges cannot be repeated but nodes can [16]. By repeating nodes, you can maintain the connectivity of the graph [16]. However, the breakup problem has the constraint that each component must be unconnected. This would mean nodes cannot be repeated across the graph to keep components disconnected. Furthermore, graph decomposition does not consider edge weights when separating components, whereas the breakup problem has an edge weighting constraint. [Fig. 7.5] displays an example of graph decomposition.

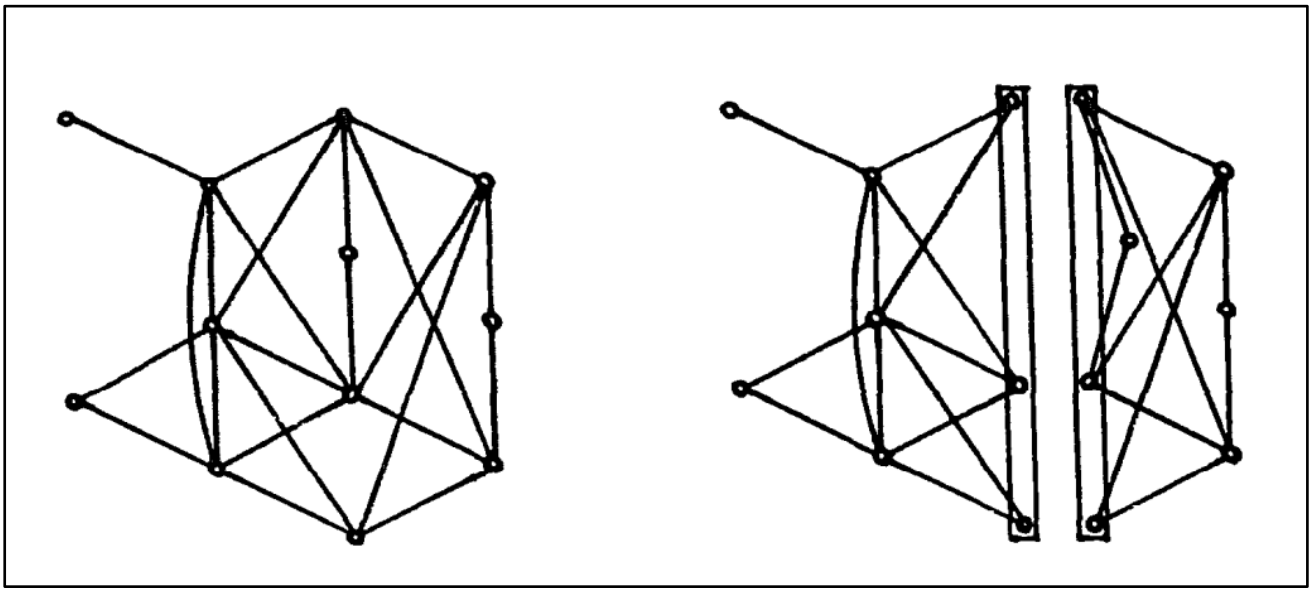

Fig. 7.5. Separating a graph into 2 connected components using graph decomposition. [15, Fig. 1]

8 Best Practice

Best practice to solve minimum k-cut involves using an algorithm that utilises these techniques. These techniques are primarily concerned with simplifying the graph to as much as possible and then iterating over the simplified possibilities:

8.1 Graph Sparsification

Graph sparsification is used to reduce the number of edges present in a graph while minimising significant change to the overall structure of the graph [11]. Fewer edges must be considered during the algorithm which will benefit running time [11].

8.2 Tree Packing

Tree packing is concerned with packing as many vertex-disjoint, isomorphic trees into a single tree [17]. This would mean fewer trees would have to be iterated over to examine possible cuts [11].

8.3 Colour Coding

Colour coding is a dynamic programming technique to categorise similar branches of a tree, so the minimum number of iterations occurs across these branches [11].

9 Advice

9.1 Basic Risk

The basic risks associated with accepting a contract to solve the break-up problem are that shipping is a large industry [19], and errors in the algorithm could result in unnecessary expenses and/or a reduction in potential income for the company.

Errors in the algorithm could result in not providing the optimal solution. Cases may occur where the company's profits are lower than they would have been if the best solution was provided. If a ship takes longer to reach its destination, the company would have extra expenses from crew wages, food and water, and extra fuel.

A reduction from potential income could occur from transhipping, leading to reduced cargo value hence reduced profits. This was mentioned prior in Liner Hub-and-Spoke Shipping Network Design.

9.2 Modifications

Some modifications to the problem could be negotiated to lower the risk of the contract. These may include:

- Upper limit on the number of nodes, edges, and integer n in the input
- Setting a total edge weight limit.

An upper limit on the number of nodes and/or edges in the input graph may help assist in understanding the scope of the problem. Depending on the number of edges and nodes that have to be handled, there may be cases where an exponential algorithm may be considered acceptable. Cases of this may include a low number of edges and nodes where integer **n = 2**. Since the scale is relatively small compared to possibly larger values of integer **n**, an exponential algorithm may be considered feasible.

By setting a total edge weight limit, if a solution is found where the total remaining weight of the graph exceeds the limit, the algorithm can halt early and accept. This would remove the guarantee of

an optimal solution, however, would allow acceptance of reasonable solutions early. By accepting the edge weight limit as input, and storing it, the time can be improved to find an acceptable solution much sooner.

9.3 Existing Solutions

The simplest and fastest solution to implement would be the brute force solution outlined in section 5. This would be an effective proof of concept, especially when smaller inputs are used. Furthermore, both suggested problem modifications can be applied to this solution to improve runtime.

The grouping approach outlined in section 6 is an extension of the brute force solution and is designed to improve the time and space complexity. The same modifications can still be applied to this solution.

If the brute force or grouping solutions are not suitable for the input sizes of the problem being solved, the best practice solution outlined in section 8 should be considered. Its time complexity of $O(n^{(1+O(1))k})$[11] is a further improvement over the grouping approach. However, its more complicated design may pose challenges to implementation of the algorithm. In addition, complications may arise from this algorithm assuming an unweighted graph [11] which is not the case for the breakup problem.

10 Bibliography

[1] R. Grassl and O. Levin, “Bell Numbers,” More Discrete Mathematics: via Graph Theory. Available: https://discrete.openmathbooks.org/more/mdm/sec_adv-bell.html#:~:text=Since%20the%20Bell%20numbers%20count,2%2C%203%2C%20and%207. (Accessed Sep. 20, 2023).

[2] “Direct Shipment vs Transhipment: Which Shipping Method Should I Use?” International Cargo Express. Available: https://icecargo.com.au/direct-shipment-vs-transhipment/ (Accessed Sep. 20, 2023).

[3] J. Mulder and R. Dekker, “Methods for strategic liner shipping network design-Issue 2,” in *European Journal of Operational Research*, Jun. 1. 2014 [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0377221713007960

[4] “Liner” *Cambridge Dictionary*. 2023. Cambridge University. [Online]. Available: https://dictionary.cambridge.org/dictionary/english/liner?q=Liner

[5] “Port Harbor Operating Costs.” FinModelsLab. https://finmodelslab.com/blogs/operating-costs/port-harbor-operating-costs (Accessed Sep. 17, 2023).

[6] The European Sea Ports Organisation, “Trends In EU Ports’ Governance 2022.” ESPO. https://www.espo.be/media/ESPO%20Trends%20in%20EU%20ports%20governance%202022.pdf (Accessed Sep. 17, 2023).

[7] O. Goldschmidt and D.S. Hochbaum, “Polynomial Algorithm for the k-cut problem,” in *29th Annual Symposium on Foundations of Computer Science*, Oct. 26. 1988. [Online]. Available: https://ieeexplore.ieee.org/document/21960

[8] R. M. Karp, “Reducibility Among Combinatorial Problems,” Springer eBooks, pp. 219–241, Jan. 2010, doi: https://doi.org/10.1007/978-3-540-68279-0_8.

[9] M. R. Garey and D. S. Johnson, Computers and intractability : a guide to the theory of NP - completeness. New York: W.H. Freeman And Co, 2003.

[10] H. S. Wilf, Generatingfunctionology. San Diego: Academic Press, 1994.

[11] J. Li, “Faster Minimum k-cut of a simple graph” in *60th Annual Symposium on Foundations of Computer Science*, Oct. 8. 2019. [Online]. Available: https://arxiv.org/pdf/1910.02665.pdf

[12] M. Christiansen, E. Hellsten, D. Pisinger, D. Sacramento, and C. Vilhelmsen, “Liner Shipping Network Design,” in *European Journal of Operational Research*, Oct. 1. 2020. [Online]. Available: https://www.sciencedirect.com/science/article/abs/pii/S0377221719308148

[13] H. Menon, “What Are Liner Services and Tramp Shipping?” Marine Insight. https://www.marineinsight.com/maritime-law/what-are-liner-services-and-tramp-shipping/ (Accessed Sep. 17, 2023).

[14] J. Zheng, Q. Meng, and Z. Sun, “Liner hub-and-spoke shipping network design,” in *Transportation Research Part E: Logistics and Transportation Review*, Jan. 22. 2015. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1366554514002269

[15] M. Fugazza and J. Hoffmann, “Liner Shipping connectivity as determinant of trade-Issue 2,” in *Journal of Shipping and Trade*, Mar. 2. 2017. [Online]. Available: https://jshippingandtrade.springeropen.com/articles/10.1186/s41072-017-0019-5

[16] W. Holberg, “The decomposition of graphs into k-connected components-Issue 3, ” in *Discrete Mathematics*, Nov. 12. 1992. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0012365X9290284M

[17] S. Masuyama, “On the tree packing problem” in *Discrete Applied Mathematics 35*, Jan. 2. 1992. [Online]. https://core.ac.uk/download/pdf/82129258.pdf

[18] geeksforgeeks. (2023, September 5). *Introduction to NP-Complete Complexity Classes*. Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/introduction-to-np-completeness/

[19] Z. Iqbal, “Overview - How the Shipping Industry Works & Key Catalysts”, Alpha Invesco. Oct. 25. 2023. [Online]. https://www.alphainvesco.com/blog/how-does-the-shipping-industry-work/