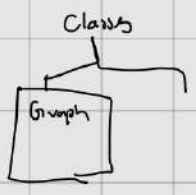
- Classes for the Graph, or using a graph library



- Algorithm for branch & bound Graph
- Could try TSP/ for the multicut problem.

Have to make a weighted graph, normal Graph will not work.

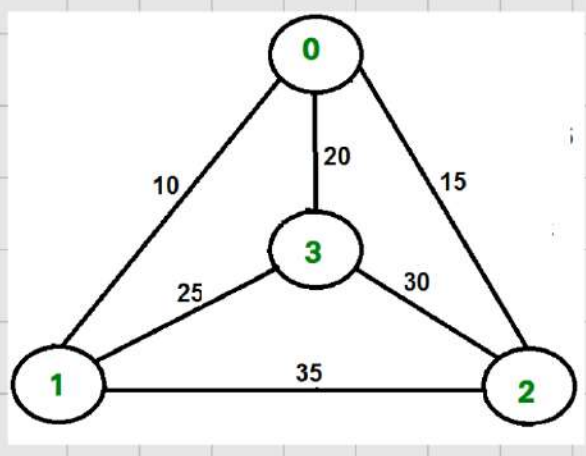
1. Determine whether there is an edge from vertex i to vertex j.
– An adjacency matrix supports this operation more efficiently.
2. Find all vertices adjacent to a given vertex i.
– An adjacency list supports this operation more efficiently.

- TSP will not work, as we are not finding the shortest path, but we are trying to find the minimum edges we could cut so that we have multiple nodes in the graphs, so basically graph

The solution is Graph decomposition algorithm but the way we do it should be in a branch & bound setting.

- We don't need weighted graph as we are focusing on the edges and not the weight of the edges

- Connected Components algorithm using this to understand the structure of the graph, maybe return the edges from each vertex, after that we could maybe use the formula. After that we could maybe use the Graph Partition Algorithm, using the number from before



Lemma. Estimating maximally probable decisions y, given attributes x, given the set of feasible decisions $\mathcal{Y}$, and given parameters θ, i.e.,

$$\underset{y \in \{0,1\}^S}{\operatorname{argmax}} \quad p_{Y|X,Z,\Theta}(y, x, \mathcal{Y}, \theta) \tag{7}$$

assumes the form of the **minimum cost multicut problem**:

$$\underset{y:\, E \to \{0,1\}}{\operatorname{argmin}} \quad \sum_{e \in E} (-\langle \theta, x_e \rangle)\, y_e \tag{8}$$

$$\text{subject to} \quad \forall C \in \text{cycles}(G)\ \forall e \in C: \quad y_e \leq \sum_{e' \in C \setminus \{e\}} y_{e'} \tag{9}$$

Theorem. The minimum cost multicut problem is NP-hard.

Bansal et al. (2004) reduce this problem to the k terminal cut problem whose NP-hardness is an important result Dahlhaus et al. (1994).

Greedy joining algorithm:

- The greedy joining algorithm is a local search algorithm that starts from any initial decomposition.
- It searches for decompositions with lower cost by joining pairs of **neighboring (!)** components recursively.
- As components can only grow and the number of components decreases by one in every step, one typically starts from the finest decomposition Π_0 of A into one-elementary components.

Greedy moving algorithm:

- The greedy moving algorithm is a local search algorithm that starts from any initial decomposition, e.g., the fixed point of greedy joining.
- It searches for decompositions with lower cost by recursively moving individual nodes from one component to a **neighboring!** component, possibly a new one.
- When a **cut node** is moved out of a component or a node is moved to a new component, the number of components increases. When the last element is moved out of a component, the number of components decreases.

- The overall algorithm could still be NP-hard as we are looking at the Graph first and then we are cutting the edges, which could be easily increase the complexity of the graph

- If not for the first algorithm then we could use the algorithm given in the ML-class, but not to sure how that would work.

- The ticket from complexity theory

- The problem big will the start of the algorithm have what kind of complexity, depend on that we could further examine what we could do, cause if it increases the complexity, then there would be no point

- Next thing to do,

↳ research the algorithm of connected component, the complexity and maybe modify the algorithm

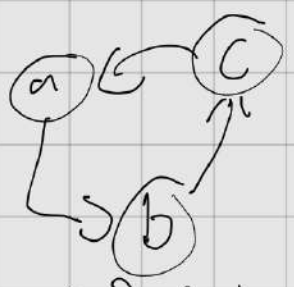
- Implement Graph Partition Algorithm -

The complexity is non linear as the data-structure itself is non-linear

What is the connection between strongly connected component & multicast algorithm?

Does a strongly connected component mean that there is a possible multicast?

This point might be mute, as it takes a look at the path from a → B



strongly connected

$V = \{a, b\}, \{b, c\}, \{c, a\}$

The results will give us the number of connected component, this could help us with the next multicast algorithm, since we know the edges and by giving the algorithm a reference point, by scanning how many edges there are, the algorithm could potentially be more efficient. Also if edge = 1, then we could just ignore it there this might reduce the complexity of the whole Problem.

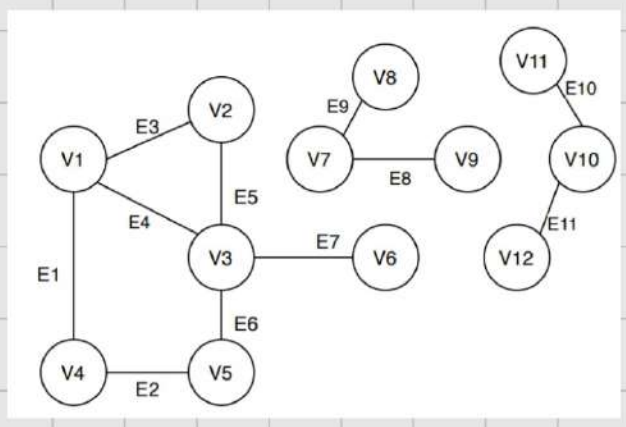
- Also could be seen as a ticket

Algorithm 1: Finding Connected Components using DFS

```
Data: Given an undirected graph G(V, E)
Result: Number of Connected Components
Component_Count = 0;
for each vertex k ∈ V do
    Visited[k] = False;
end
for each vertex k ∈ V do
    if Visited[k] == False then
        DFS(V,k);
        Component_Count = Component_Count + 1;
    end
end
Print Component_Count;
Procedure DFS(V,k)
Visited[k] = True;
for each vertex p ∈ V.Adj[k] do
    if Visited[p] == False then
        DFS(V,p);
    end
end
```

- Implement an undirected Graph
- Implement the connected component
- Graph partition algorithm

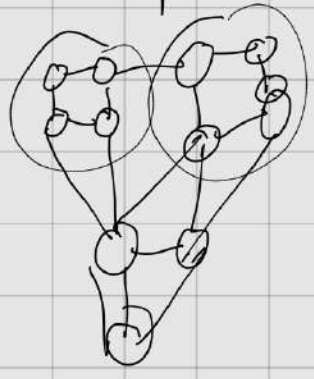
Strongly connect, might not be applicable for the problem above.

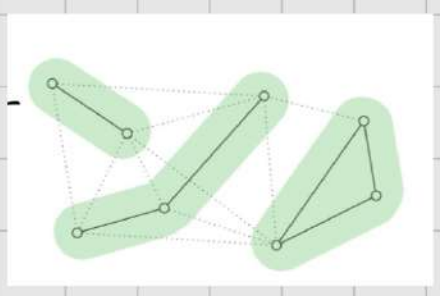


connected components
O(n+k)

↓

Get the result, now we know the component





Knowing the component
(then we use the min algorithm
(compare the complexity with the complexity of the multicut algorithm.

Implement the algorithm

connected component → Graph partition algorithm.
(multicut algorithm)

connected components would not be useful as there could be a lot of combination & Doing it on a larger scale would not be feasible

```cpp
// C++ program to print connected components in
// an undirected graph
#include <bits/stdc++.h>
using namespace std;

// Graph class represents a undirected graph
// using adjacency list representation
class Graph {
    int V; // No. of vertices

    // Pointer to an array containing adjacency lists
    list<int>* adj;

    // A function used by DFS
    void DFSUtil(int v, bool visited[]);

public:
    Graph(int V); // Constructor
    ~Graph();
    void addEdge(int v, int w);
    void connectedComponents();
};

// Method to print connected components in an
// undirected graph
void Graph::connectedComponents()
{
    // Mark all the vertices as not visited
    bool* visited = new bool[V];
    for (int v = 0; v < V; v++)
        visited[v] = false;

    for (int v = 0; v < V; v++) {
        if (visited[v] == false) {
            // print all reachable vertices
            // from v
            DFSUtil(v, visited);

            cout << "\n";
        }
    }
    delete[] visited;
}
```

BFS/DFS
$O(V+E)$ linear

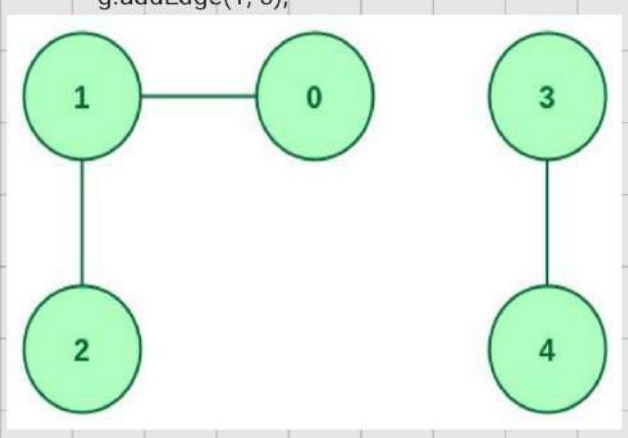
↳ Does not count as computing lower bounds

```cpp
void Graph::DFSUtil(int v, bool visited[])
{
    // Mark the current node as visited and print it
    visited[v] = true;
    cout << v << " ";

    // Recur for all the vertices
    // adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
        if (!visited[*i])
            DFSUtil(*i, visited);}
Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}
Graph::~Graph() { delete[] adj; }

// method to add an undirected edge
void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w);
    adj[w].push_back(v);
}
// Driver code
int main()
{
    // Create a graph given in the above diagram
    Graph g(5); // 5 vertices numbered from 0 to 4
    g.addEdge(1, 0);
```

↳ return value change



```
g.addEdge(2, 1);
g.addEdge(3, 4);



cout << "Following are connected components \n";
g.connectedComponents();

return 0;
```

$G = (V, E)$ [Exploring the search space

Connected components return nodes & connection

$O(V+E)$

node - storage?

ejection chains algorithm -?

options

Greedy (Local optima)

Heuristic Approach (escape local)

Approximatic Algorithm

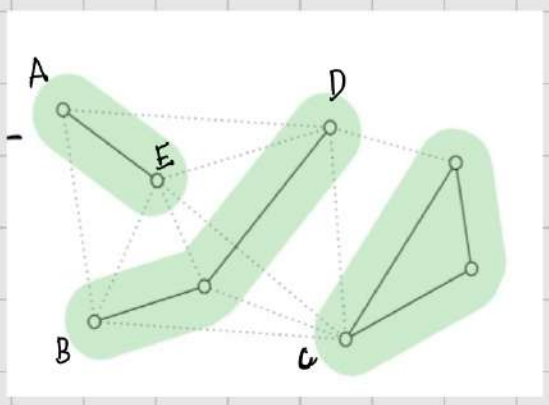
The other approach would be through exploitation of the search space without knowing the connected component

- Maybe greedy falls under this category
- Heuristics could be tried here
- Tabu search, maybe can be used with greedy -?
  - That could be one test case the other would be.
  - dichotomic search

$$\text{node} = (\{A, B\}, \{B, E\}, \{A, E\}, \{A, D\}$$

Greedy Algorithm

the sum of the cost of the edges that are cut is minimal.



How would you calculate the impact of a cut edge as it doesn't have any weight, or maybe the connected edges could be used as weights, but how would you calculate the impact?

Don't need to use DFS, as it's already done in connected component

Connected component (G=(V,E)) = Node

Multicut = (Si, ti)

Greedy (Nodes, Multicut)

A L DFS (Si, ti)

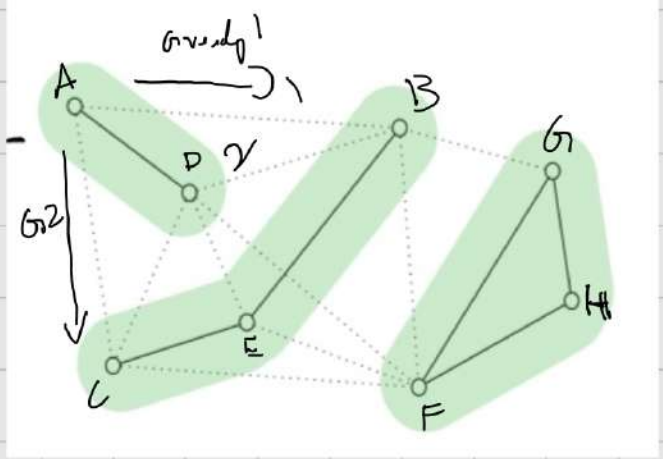
return new node ;

M ⊆ E

M = (Si, ti)

= (A, D), (G, D), (E, B)

Greedy traversal



To segment A & D we have to cut all nodes connected to them, this means have to keep track of the connections

Greedy least amount of edges traversed?

The other solution could be through the path taken to get to the target, depending on the amount of edges traversed but this could mean we need to scope the vertex choices, maybe recursion or for loop

A — C = 4

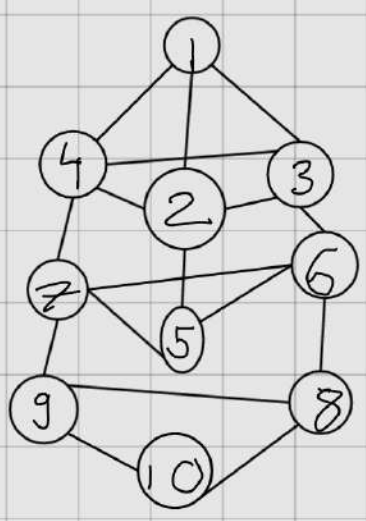
D(6) E(4) F(4)

DFS & Greedy

M = (Si, ti)

|

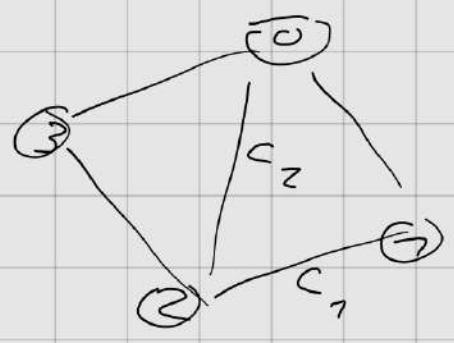
5



M

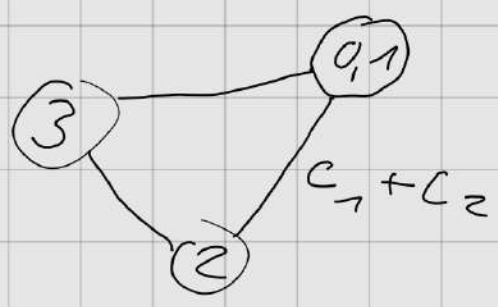
(1, 5)
(2, 6)
(3, 7)
(4, 8)
(5, 9)
(6, 10)

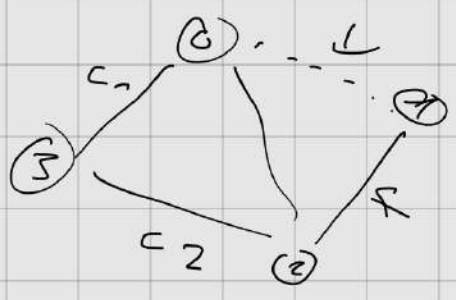
Contract the graph



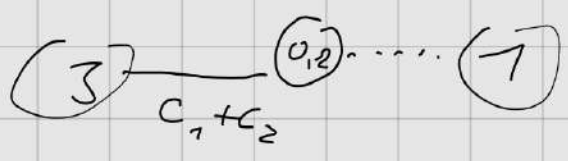
0 and 1 are connected

0 and 1 not connected





02 is connected



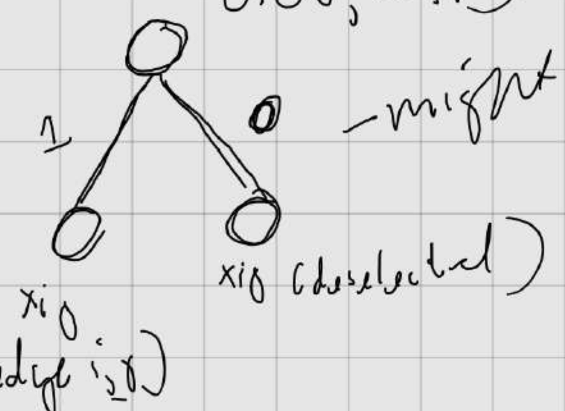
The edges are weighted (used in red pair)

So, we have positive & negative edges where the positive are willing to be traversed & the negatives are willing to be cut

Finish the paper after that implement the different algorithms, once that is done then move on to the other different papers or, do it together with the algorithm.

1 means same
Partision

$G(V, E, W)$



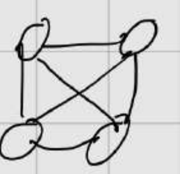
- might not be correct.

2 nodes are derived from the same node if the node is not measured, meaning it exceeds the overall lower bound,
2^{nodes}, the total number is unknown

$$\frac{4(4-1)}{2}$$

6

$$\frac{12}{2}$$



$$\frac{n \cdot (n-1)}{2}$$

the most amount of edges.

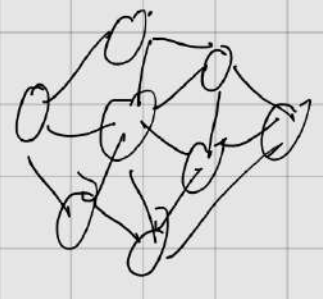
$$\nabla(i,j,k) := \begin{cases} \min\{|w_{ij}|, |w_{ik}|, |w_{jk}|\} & \text{if } (w_{ij} < 0 \wedge w_{ik}, w_{jk} > 0) \\ & \vee (w_{ik} < 0 \wedge w_{ij}, w_{jk} > 0) \\ & \vee (w_{jk} < 0 \wedge w_{ij}, w_{ik} > 0) \\ 0 & \text{else.} \end{cases}$$

-50 68 45

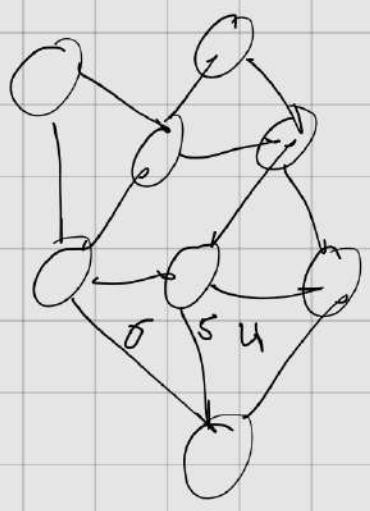
w_{ij} - negative
w_{ik} - "
w_{jk} - neg

minimize it

minimize the initial upper bound $\bar{g}_C < \bar{g}^*_0$
→ reduction of multicut through triples.



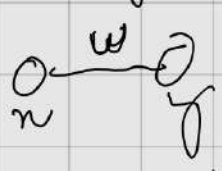
Branch & Bound for a graph with weights
- Constraints
- weights
- Negative constraint
- Implement a graph.
- knapsack could work but the problem being we do not have the constraint of vertices



• Does the multicut need to maximize the sums of it's partisions

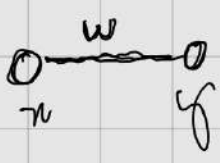
Approches

- Limit the number of vertex (k-mapcut)
- Largest sum, deaves which one would lead to the largest sum



remove edge (cutting the edge)

have to remove u from v &
v from u.



The reason we need the [ ], is to keep the multiple edges from a single node

```
// To add an edge
void addEdge(vector <pair<int, int> > adj[], int u,
                                        int v, int wt)
{
    adj[u].push_back(make_pair(v, wt));
    adj[v].push_back(make_pair(u, wt));
}
```

the u & v are the indexes and in that index they have pair with other vertex

• Needed an Array that can store the information about the edges

struct Vertex [] ← Has to be an array

Maybe could use auto at this segment

- Edge
- target

Add vertex

could Potential be used for removal too.

```
// Print adjacency list representation of graph
void printGraph(vector<pair<int,int> > adj[], int V)
{
    int v, w;
    for (int u = 0; u < V; u++)
    {
        cout << "Node " << u << " makes an edge with \n";
        for (auto it = adj[u].begin(); it!=adj[u].end(); it++)
        {
            v = it->first;
            w = it->second;
            cout << "\tNode " << v << " with edge weight ="
                 << w << "\n";
        }
        cout << "\n";
    }
}
```

the it is suppose to go to the next index

methods for traversal., removing could be done with this method. ↳ Find method, that could work will need more experimentation.

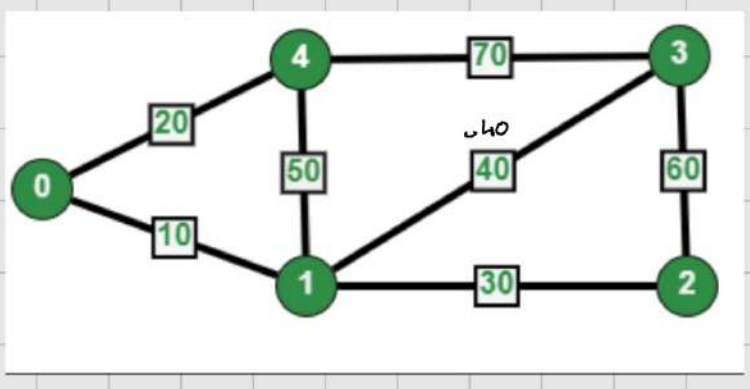
finding the element O(n) not going to be O(1)

The search would not work as we are not looking for a specific element beet, instead at the edge we need to cut, so maybe we could use an equals where a > b or a < b depending on the format , we just have to look at the different edges, as they are stored in pairs

this would work as we could traverse the graph and it will always give the maximal number, but because this is towards 1 based their how to deal alternatives

that would be useful for the multicut

Pose a simple remove method



• Constraints for multicut

will require 2 loops as the other variable will have different size

loop {

for { auto it = adj[v].begin(); it! = adj [v].end(); it++

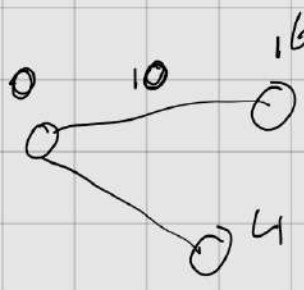
this has to be the variable we remove,

at that index we remove it

destroy the pair;

}

using search algorithm will not decrease the complexity



int x = it. first

int y = it. second

the vertex we passing towards

if (x == B)

{ ; this would mean we are at that index now we would like to delete

How would you delete the pair - ?

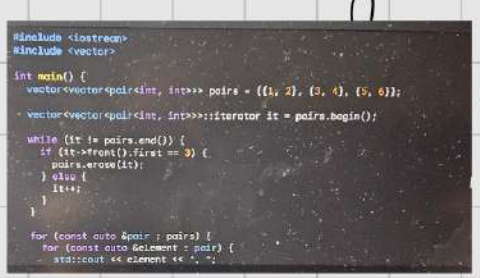
}

The deletion could be done in one loop, but we will need to check which variable is bigger.

using if, then we could use the values to configure the problem

thing to do the size -?

for the delete method we could use a counter variable that, only lets the variable enter as long as the counter is true (kind of like a while loop)



```
#include <iostream>
#include <vector>

int main() {
  vector<vector<pair<int, int>>> pairs = {{1, 2}, {3, 4}, {5, 6}};

  vector<vector<pair<int, int>>>::iterator it = pairs.begin();

  while (it != pairs.end()) {
    if (it->front().first == 3) {
      pairs.erase(it);
    } else {
      it++;
    }
  }

  for (const auto &pair : pairs) {
    for (const auto &element : pair) {
      std::cout << element << " ";
```

→ deleting a pair

Next test case would be print the delete, before returning the value.

- The erase method does remove the index but, the space does not seem to be cleared up —?
  I could potentially

We are working with numbers, so we could work off the size, we don't specifically need the numbers

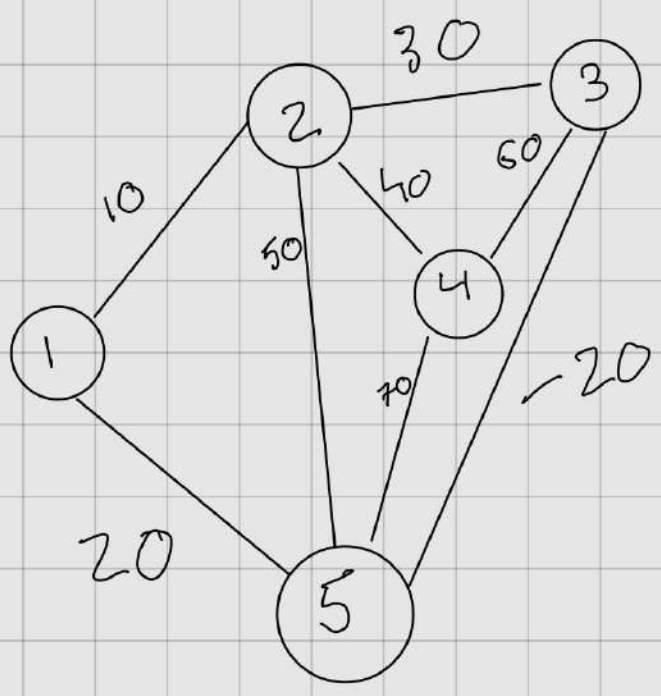
first, we check the size adj[], init x, init y

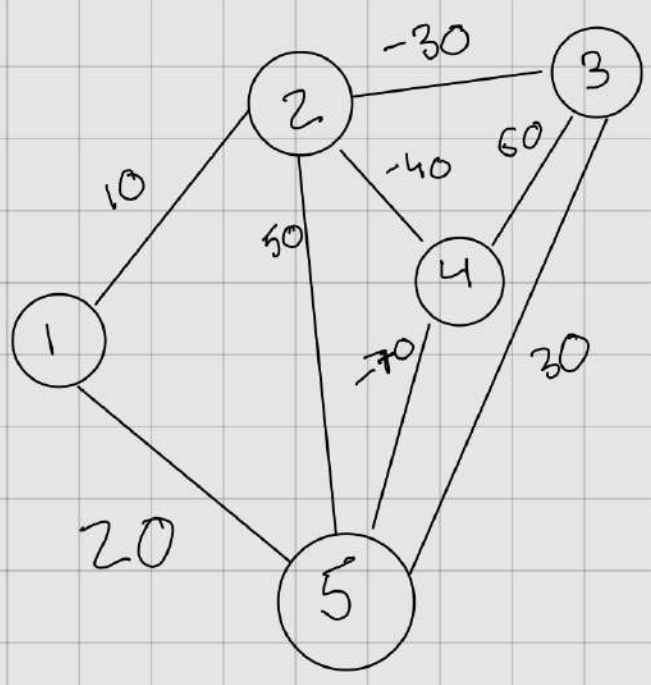
x > y

{ int x

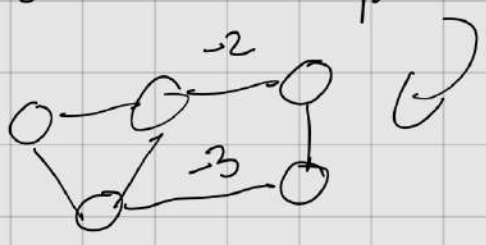
for(auto it = adj[x].begin; it != adj[x].end(); it++)

incrementing the second variable, now we have to figure out about the second part of the variable, specifically the incrementing part.



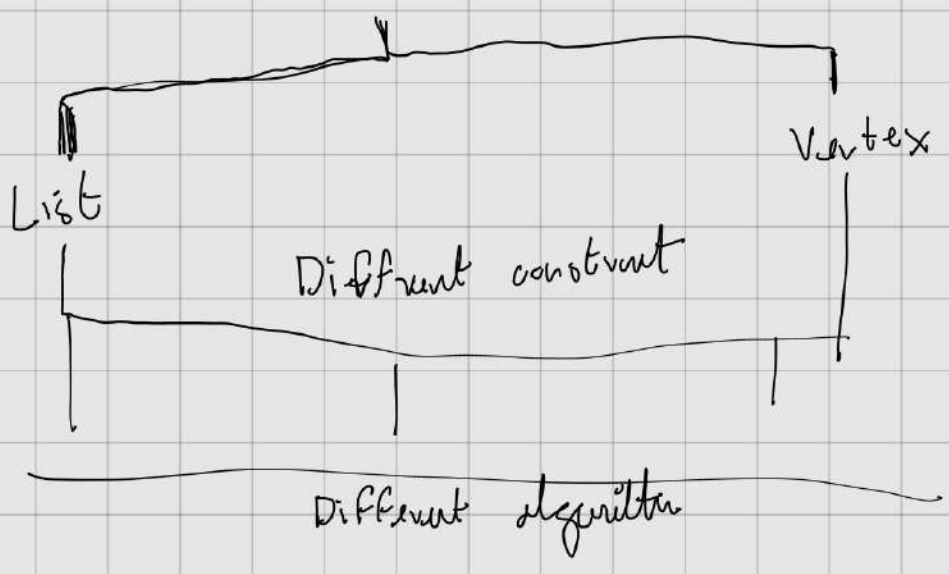


- Simple multicut, have to choose the constraints, and the amount of nodes, can start simple



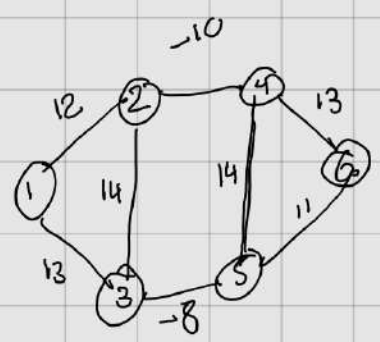
- Then branch & bound, and looking about the different constraint we need to use, to achieve it.
- Take bit's and pieces from the algorithm in the papers, not all of it.
- Also have different constraint
- Try different branch & bound algorithm
  - combining them with Tabu
  - Greedy
  - Simulated annealing
  - Min cut

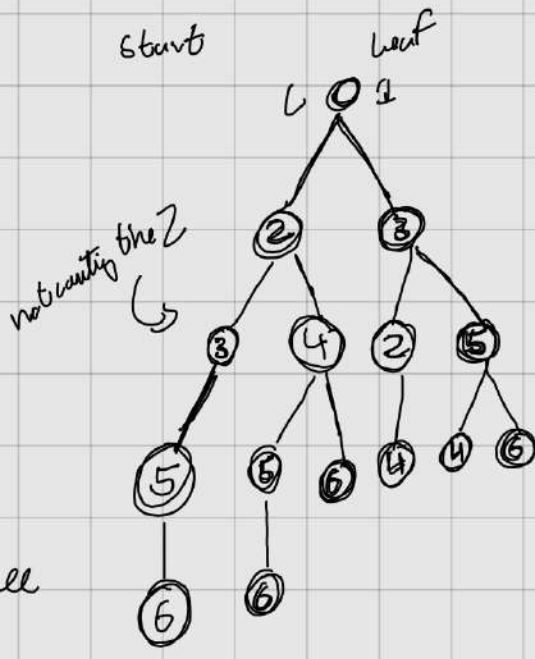
- Also with all of the algorithms try with diffrent data structs



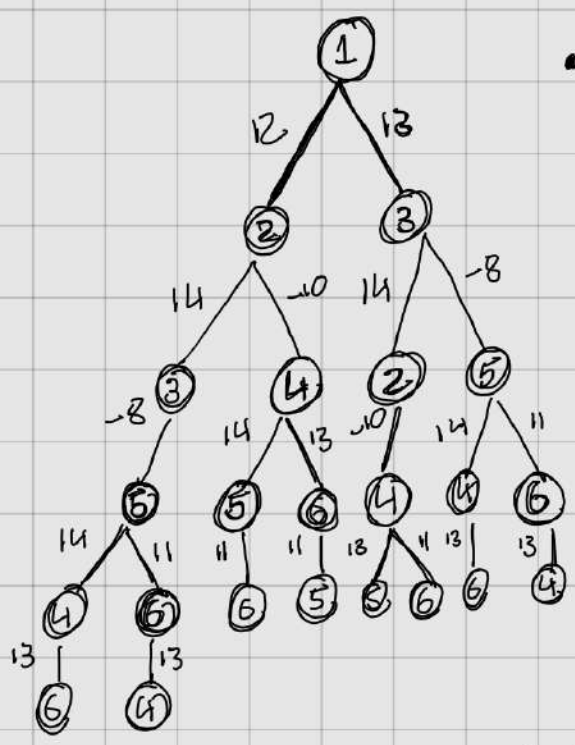
upper bound, can be used as a constraint, may be limiting the amount of sum that could be there else node be maximin number of nods in the graph.

Least cost - Branch & Bound





State - space tree

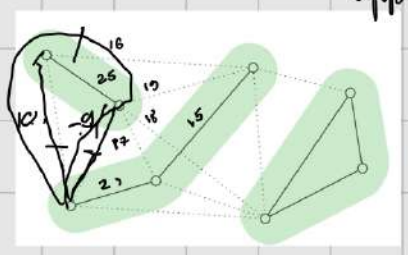


- This will just give the shortest path, pretty much.

- use a map, where the weights would be the map.

upper bound is 40

$\Sigma_{xij\ max}$ ↑
- see how close we can get to 40



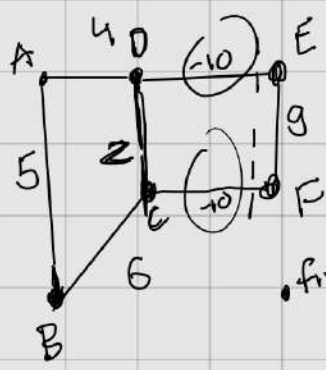
Constraint Propagation
- Can be used to make the cuts
- Number of vertex
- Number of clusters

- minimal multicut
- sum of the edges are cuts to a minimal

- initialize the upper bound
- Bell number for branching
  - How many ways are there to partition a set, with n - number of elements

- 8The tree for the branch and bound can be used but will have to check all the solutions also keep the negative number edge that way we can ignore the branching from their and make a cut.
- Checking if they can be in the same partition, the paper reduces the branches that way could potentially be use ful for the branching, after the negative edges we could maybe perform a new cut

B D, E = optimal startpoint



B/t + t/t D = P

- fixed node
- $\Sigma_{x_{ij}}^{max} = 40$
- weight = 0 ↓ has to be minimal

$\Sigma_{xij\ max} = 10$ (try to get as close to 10 as possible)

$16 \geq \quad \geq 10$

min - multicut

- fixed node

$\Sigma_{xij} = AB(5) + BC(6) = 11$

Pop (AB)

$\Sigma_{xij} = BC(6)$

- Now that the sum exceeds the maximal upper bound, we could potentially ignore the previous branch (AB), and move forwards many pop (AB) keep BC
- The pop is possible as there is another path to be taken.

$\Sigma_{xij} = AD + DC = 6$

- A condition that makes sure, if there is any cubic edge it's ignored
- it produces 2 cuts EF, BCD and because the cuts have to be unique we can't count ADC

(How to be sorted into a new subgraph)

- The biggest sum between the common edges will be chosen.

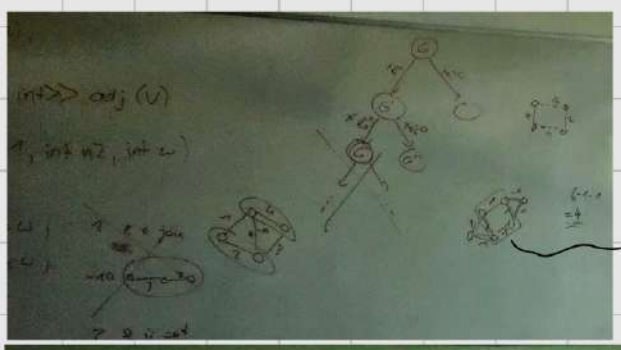
- The stack would have a new graph BCD, now because we only have negative edges the cut could be made here as traversal would mean there we need to add the negative numbers

- Since the best possible answer would be
- They can't have a common node DAB & EF would be the ideb cut
- would have an optimal solution if the starting node was b (must see how to deal with the

- storage can be done with vector <pair> (int, int) [..], then the comparision be done after it is sorted.

- How to calculate the best starting point?
  I can't use bell number in this scenario
- Greedy search from professor Bpires

• Complete graphs

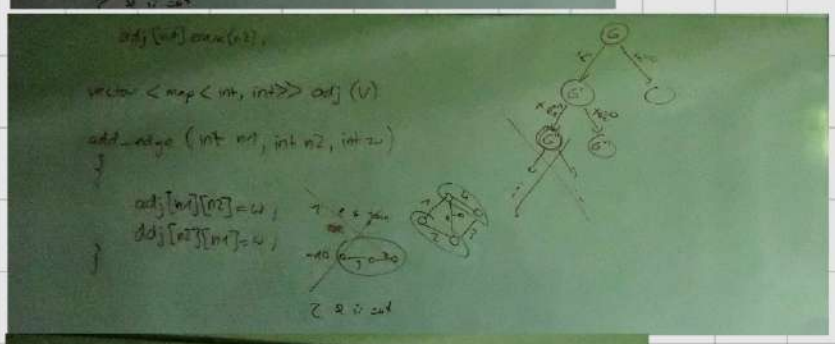


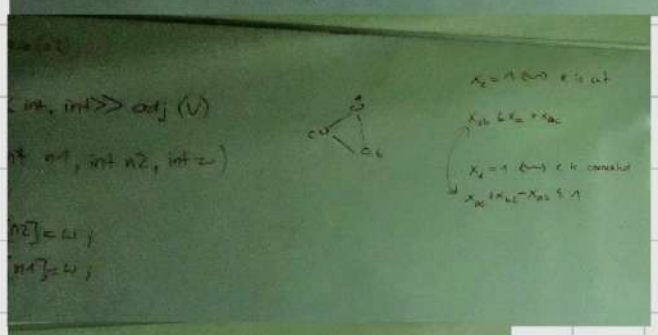
The upper bound is calculated.

In multicut the 1 means that it is in the multicut

$x_{01}=1$

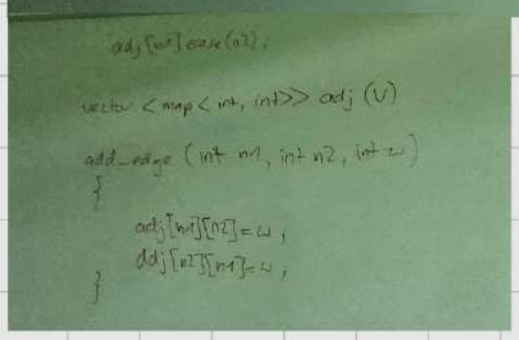
$x_{12}=0$ - not in the multicut





• Keep in mind the cycle of the graph

→ you can't cut the vertex because of the cycle is a multicut



- Connect the graph into array instead of an arrayy
- it is faster.
- Can use the weights to act as the edges it has faster traversal.

Objective

- Try to maximize the $\sum x_{ij}$ of the multicuts
- The upper limit should be calculated

The best we could get from here would be 7, but because of the −, we have to keep track

$4+3-2$

$= 5$ // - the upper bound would be 5

A cycle is a path that starts and ends in the same vertex

$0 \to 1 \to 2 \to 0$ cycle.

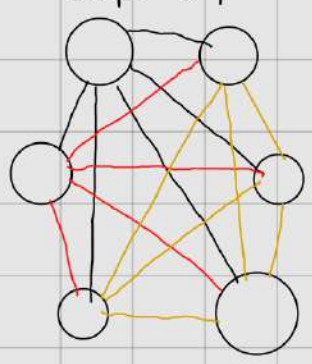
Complete graph edges $\frac{n(n-1)}{2}$ n = vertices

- All the vertices have to be connected

$\frac{4(4-1)}{2}$

$= \frac{4 \times 3}{2} = 6$

complete graphs



$\frac{6(6-1)}{2}$ $\frac{15}{30}$ $\frac{}{21}$

$=$

commit the change do V_1

- Set up a github repository
- Change the vector from array to HashMap
- The greedy algorithm used by 0 bjärs, can be used to find the approximate node
- Create a method to cycle, should be simple enough, can be used after we get a multicut to check the consistency
- Have to set up a upper limit, need ways to figure out that part

$$\max \sum_{1\le i<j\le n} w_{ij}\cdot x_{ij}$$ maximize the sum of the weight of t

$$\text{s.t.}\quad x_{ij}+x_{jk}-x_{ik}\le 1 \quad \text{for } 1\le i<j<k\le n$$
$$x_{ij}-x_{jk}+x_{ik}\le 1 \quad \text{for } 1\le i<j<k\le n$$
$$-x_{ij}+x_{jk}+x_{ik}\le 1 \quad \text{for } 1\le i<j<k\le n$$
$$x_{ij}\in\{0,1\} \quad \text{for } 1\le i<j\le n.$$

$x_{ij}=1$ same cluster if not in the same cluster then

$x_{ij} \to x_{ik} =$ [merge not in the same cluster
$1-x_{ij}=0$

$\overline{g}_0^* := \sum_{1\le i<j\le n} \max\{w_{ij},0\}$. Adding the positive edges

This upper bound can similarly be applied to each node λ of the search. If a positive edge weight is deselected or if a negative edge weight is selected, the upper bound is reduced accordingly:

$$\overline{g}_\lambda^* = \overline{g}_0^* - \sum_{\substack{1\le i<j\le n\\ x_{ij}\text{ fixed}}} \big((1-x_{ij})\cdot\max\{w_{ij},0\} - x_{ij}\cdot\min\{w_{ij},0\}\big).$$

x_{ij} would either be the selected(1) or unselected (0)

Adding the edges from previous visited edges

negative edges

nodes ⇒ All the positive edges

node not in the multicut ? (X) the max weight of the nodes in the cluster ⇒ the minimum edge in the multicut

- remember the upper bound is for the search space.