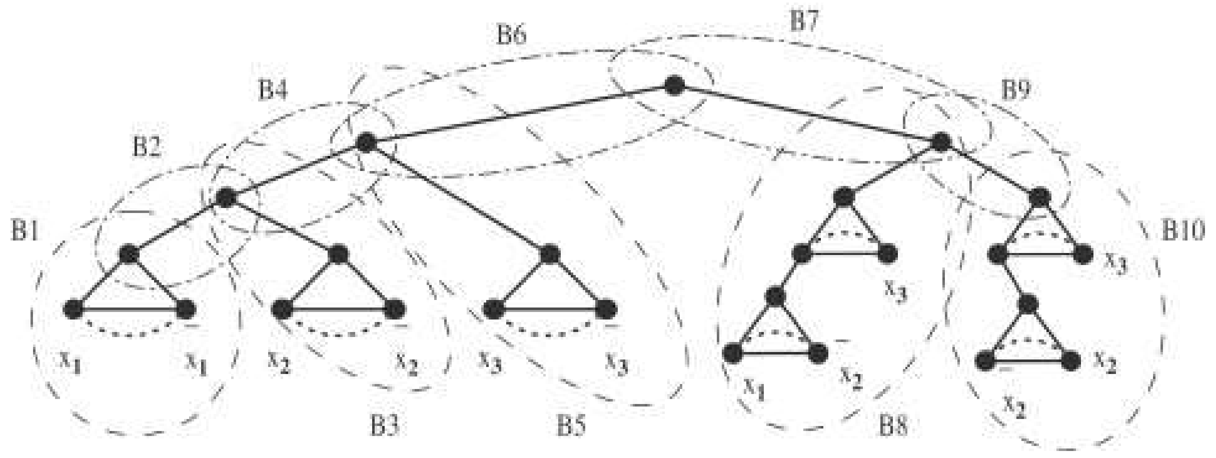
Combinatorial Branch and Bound algorithm for multicut



Penzen Lama
pela250e@tu-dresden.de

Table of content: PG

Penzen Lama
pela250e@tu-dresden.de

Introduction

The multicut problem is a fundamental combinatorial optimization problem with applications in areas such as computer vision and machine learning. Given an undirected graph with weighted edges, the goal is to partition the graph into components by removing edges, while minimizing the total cost of the removed edges or maximizing the sum of the edges in the partition. The project follows the description of the latter.

In the project, we are working with complete graphs, meaning each node is connected to its subsequent node.

This project aims to explore a purely combinatorial branch-and-bound algorithm for solving the multicut problem. In contrast to ILP-based methods, this algorithm will not rely on linear programming, offering potential advantages:

Faster Exploration: The search tree can be traversed more quickly without the overhead of solving linear programs at each node.

Structural Exploitation: The algorithm can directly leverage the specific properties of the multicut problem, such as edge contractions.

Persistence Integration: Techniques to detect persistences (edges that must be included or excluded in any optimal solution) can be easily incorporated.

The primary tradeoff of a combinatorial approach is potentially weaker lower bounds compared to those obtained through linear programming relaxations. To further explore the bounds, we have calculated the bounds in three different ways as will be seen in the testing.

The project has implemented the combinatorial branch-and-bound algorithm in C++ and compares its performance against classical ILP-based solvers results given by [1].

Methods

The algorithm is done in a branch and bound way, the way it has been done can be described as Here's a detailed description of the provided code, focusing on the core algorithm, its methods, and how it aims to find clusters in a graph to maximize the sum of cluster weights.

Data Structures

Struct tulip

The struct is the main structure that holds together the different data structures used for the branch and bound. The main data structure that holds the data is a multimap used with pairs.

multimap<int, pair<int, int>>

The first int is a node A and the first int in the pair is the weight and the second int in the pair is the node B, so it's represented naturally.

The node: A--- weight---> B(connected node)

Since we are working with a complete graph, we can store multiple values in the same node. There are a total of 5 multimaps, their purposes are:

Jyon: Jyon stores the data for the join part of the algorithm, stores all the data from the joined section

Part: The part stores the part of the recursion where we start to cut the graph, it stores all the recursive elements of the cut.

Pcut: The Pcut stores the cuts that have happened in the recursion.

Jcut: It is a reference to the kept for the cuts before the recursive call of the join, it keeps the cuts as they were before the Join call.

bd_ref : This is used to keep a reference to the graph passed into the recursion, as after the recursion the graph data structure will change, and it is used to calculate the actual bound of the cut/join.

Int drac: Used for the starting case of the first recursion

Int bo: The calculated bounds are stored in this data structure, it is also used to update the lower bound before going into the cut recursion.

Cluster: Used to keep track of the cluster formed, done before the join method to show the nodes in the clusters.

Core Functions

Method: readGraphData

This method is used for getting the data from the text file provided by [1] and converting it into vectors to be used in the multimap.

Method: AddEdge & remove_edge_2

Used to add edges to the multimap, but to make sure it's an undirected graph two additions are happening, it's the same for removal since it's connected from both nodes it needs to be removed from both nodes.

Add/ remove

node: A--- weight---> B

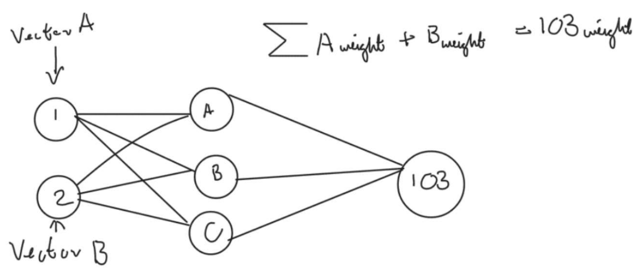
node: B--- weight---> A

Method: printGraph & prtspecifc

Print graph is used to print the whole graph and prtspecific takes in the graph we are trying to print and a specific node we want, and it will print the data for that specific node.

Method: Join

This method takes in a multimap and two nodes we want to join, the join can also be seen as a cluster of nodes, we extract the data from the multimap and store node A and node B's data into two vectors, after that we iterate through each member of the nodes in the vector and combine the weights and the nodes, the format used to name the new nodes are A + B + 100. The weight between A and B is transferred into a node called 0 where it stores the weight between them.



Method: Checkcut

Method takes in a multimap, node A and node B, this method is primarily used to check if two nodes are permitted to join, meaning if Node A and B have been joined, but node B has a cut with node C, this would mean that A and B and C cannot be in the same cluster.

The first check done is for the cut size meaning if there are no cuts then that would mean there is no restriction join.

The next check is to see if the iterator is pointing at node 0(stores information for the cuts) which is not allowed to join.

The third check is to see if the iterator returns the size of the node, which would mean that the node does not exist in the cut, or if the graph is small enough that sometimes the size of the cut and the node could be the same so there is a check to see if it’s truly not in the node or a coincidence.
If all the previous checks are skipped then we get to the loop where it checks if the node has been cut, if it has it returns a false if not then true.

Method: Count
Counts all the connected nodes except for 0 nodes, which stores the joined values of A and B.
Method: initial_bounder
Calculates the initial bound, which is the positive sum of the edges as it’s done in [2].
Method: Bounder
Calculates all the bounds for the current bounds for the graph, this means that all the possible edges that can be cut or joined are checked first using the checkcut method.
After the edges have been checked their weighted values are calculated the positive and the negative numbers. This gives us a bound that is stricter than the others.
Method: nd_bounder
It calculates the bound of the current node, meaning it uses all the feasible solutions(using Checkcut) from the current node and calculates the bound.
Method: frcut
This method has a looser bound because it is similar to the bounder method but the only difference is it only calculates the positive nodes that can be traversed for cut or joined which gives us a bigger bound.

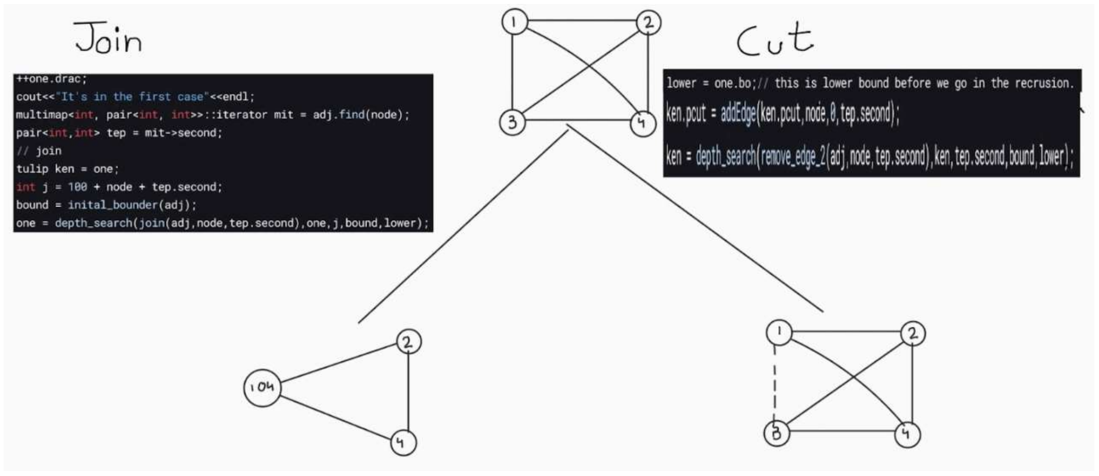
Algorithm

The primary goal of the algorithm is to implement a combinatorial branch-and-bound algorithm to solve the multicut problem. In essence, it aims to:
Partition a graph: Divide the graph into multiple clusters (cliques).
Maximise Weights: Find a partitioning that maximizes the sum of the weights of edges within the clusters.

The depth_search Algorithm

The algorithm has been named depth_Search because the branching happens in a DFS manner; this function employs a recursive branch-and-bound approach to explore different graph partitions.

The first check with **one.drac** is done at the very start of the recursion tree, this is the case where the algorithm is at the root node of the tree, and this check only works once.
The upper bound is calculated with the help of the **initial_bounder** which calculates the upper bound or the best possible solution of the graph.
Then the recursion starts for the join part, where the input node is joined with it's first connected node.
For the cut part of the recursion, we keep a reference to the original graph with the help of **Ken**(multimap), we use the same node we joined with but instead of joining we cut.



```
++one.drac;
cout<<"It's in the first case"<<endl;
multimap<int, pair<int, int>>::iterator mit = adj.find(node);
pair<int,int> tep = mit->second;
// join
tulip ken = one;
int j = 100 + node + tep.second;
bound = inital_bounder(adj);
one = depth_search(join(adj,node,tep.second),one,j,bound,lower);
```

```
lower = one.bo;// this is lower bound before we go in the recrusion.
ken.pcut = addEdge(ken.pcut,node,0,tep.second);
ken = depth_search(remove_edge_2(adj,node,tep.second),ken,tep.second,bound,lower);
```

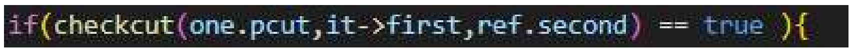
Base Case

If the graph is empty (all nodes joined or cut), or if a cut has removed all connections to the current node, it signals a terminal condition for the recursion.
There is another condition that checks the node provided by the iterator, we make sure that it does not point at 0 and that the connected node is not pointing at 0.

Penzen Lama
pela250e@tu-dresden.de

Recursive call

The recursive call works with the help of a **for loop**, it works with the updated multimap size for each iteration.
 The first check is done to see if, in the iteration, the next connected element to the node is the same node, this would mean we can't join it because it has been cut so we return it.
Before we get into the recursion we do another check to see if we are permitted to join it using **checkcut,** we do this because only if we can join it, then we cut it as that is the basis of the recursion.



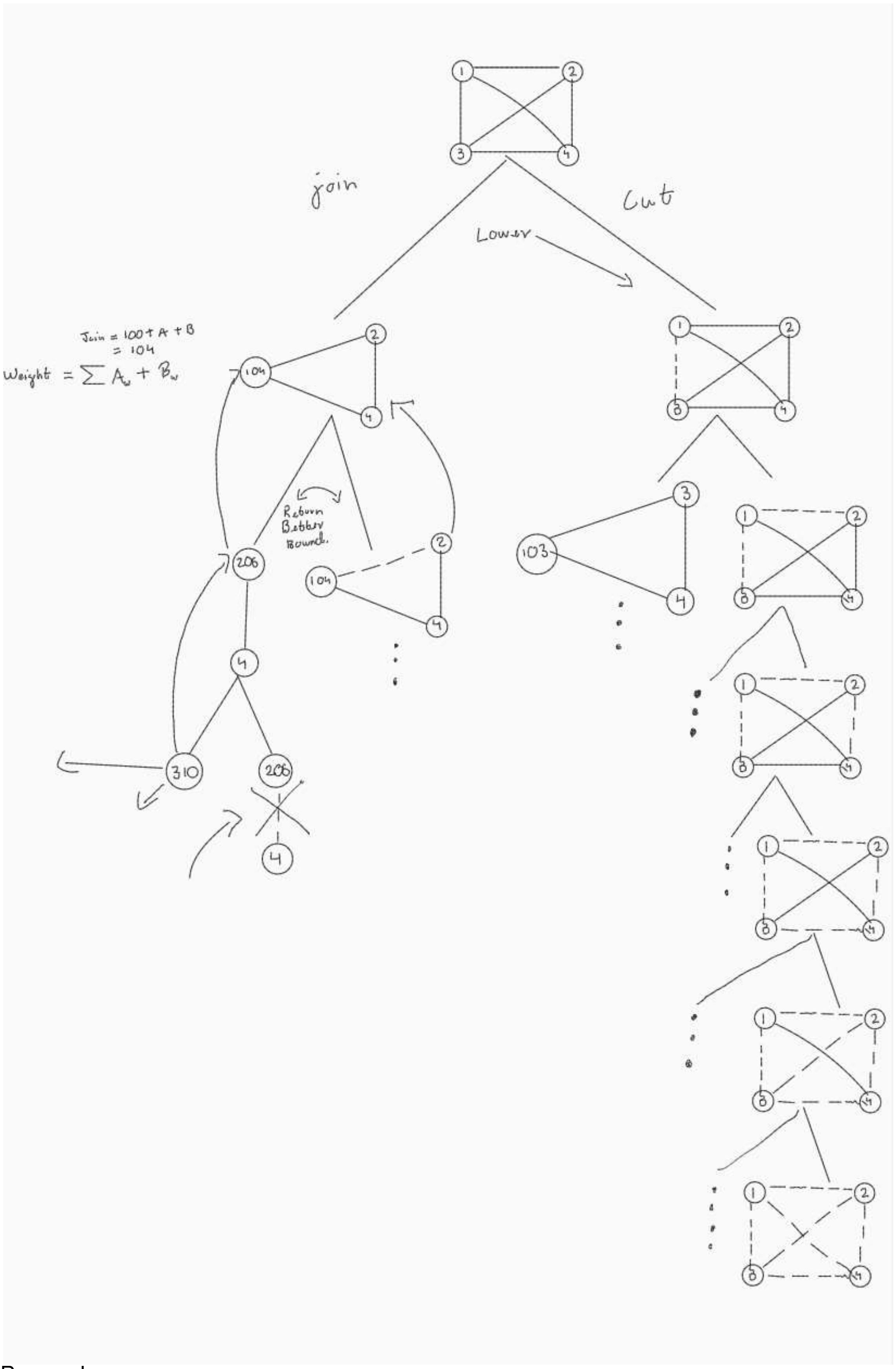
```
if(checkcut(one.pcut,it->first,ref.second) == true ){
```

Before we start the recursion we keep a reference to the struct passed in, this is because we will use it for the cut recursion, and after the join recursion, the one passed in will be altered, so we keep a reference **ken.** We also keep a reference to the cuts with **one.jcut.**

Bounding Strategies

For the bound there are two ways we can do it, using the **bounder** or **frcut,** bounder provides a tighter restriction as compared to frcut as seen in the tests and thus there is less traversal than the frcut, thus the bound is not as good as the frcut in larger graphs.
The idea for the two bounds comes from [2], where the initial **bound**(upper) is just the sum of all positive edges in the graph. In the paper they have taken a lower bound for a node to further examine if branching is feasible or not, but in the case of the algorithm the idea is the same, but instead of looking at just the node we calculate the bound of the overall graph at that point of the recursion, but this will only include feasible solutions, meaning we only add the bound if **checkcut** returns true.
The final check the upper bound of a particular branch is lower than the current best solution (lower bound), that branch is pruned – it cannot lead to a better solution
After the Join recursion has ended the **lower** bound is updated to the bound returned from the join.
The lower bound is compared to the current bound for the cut, if it's greater we return, else we traverse.
After both the join and cut recursion are finished we compare the bounds of both recursion and return the better bound up to the parent node on the recursion tree.

Distinct clusters are then shown, and the Joined Components Groups of nodes in the jyon structure represent the clusters. Each joined component forms a cluster within the final solution. The goal of maximizing the sum of weights within clusters is achieved as the algorithm tends to join nodes with higher weights.



Penzen Lama
pela250e@tu-dresden.de

Testing

The testing is done with the use of the data sets provided by [1] and some test cases created by myself, the results are based on three different bounds.
The first bound is a stricter bound done using **Bounder.**
The second test is done using **Frcut,** it has a loser bound as compared to **Bounder,** so if the solution in **bounder** is not optimal **Frcut,**will find a better bound.
The third test is done using **nd_bounder,** this is a test done in paper [2], where we calculate the bound based on the current node.

Bounder bound

Nodes	**Sum of all**	**Start index**	**B&B**
3	12	2	12
4	6	4	5
6	18	1	14
10	30	6	4
32 (Lung cancer)	3472 (ILP solver)	30	2721
33(cars)	1501 (ILP solver)	20	1346
30(wild cats)	1304 (ILP solver)	25	716
40(soybean 21)	3041 (ILP solver)	15	595

Data Sets

Frcut bound

Nodes	Sum of all	Start Index	B&B
3	12	2	12
4	6	4	6
6	18	1	14
10	30	6	11
32 (Lung cancer)	3472 (ILP solver)	30	2721
33(cars)	1501 (ILP solver)	20	—
30(wild cats)	1304 (ILP solver)	25	716
40(soybean 21)	3041 (ILP solver)	15	967

Bd_bounder

Nodes	Sum of all	Start Node	B&B
3	12	2	9
4	6	4	6
6	18	1	14
10	30	6	8
32 (Lung cancer)	3472 (ILP solver)	30	2635
33(cars)	1501 (ILP solver)	20	1346

Nodes	Sum of all	Start Node	B&B
30(wild cats)	1304 (ILP solver)	25	698
40(soybean 21)	3041 (ILP solver)	15	595

Bounding Methods Comparison

The first bound we use, called **bounder,** gives the best answer in the graphs with the lower bounds,but as the number of nodes increases the results seem to be suboptimal,the algorithm seems to be suitable for small graphs. In smaller graphs (n <12), the cuts and clusters formed by the algorithm are optimal.
The algorithm seems to work best with **bounder**, at least in the case of the bigger graph, when there are more positive weights, the bounds for cars(33) and lung cancer(32) have suboptimal answers as compared to the ILP solver, but as the nodes and the negative weight increase the bounds seems to get worse(soybean).

The second bound, called **frcut,** gives a better bound for few of the cases for the 4,10, 33(combined with **Bd_bounder**) and 40. The problem with it is because **frcut** initially calculates the best possible bound at a certain point in the recursion, it triverses more of the graph to find the better solution, which means it takes more time and resources as compared to the stricter bound, gives a better bound but at the cost of more traversal and time.

The third bound, called **Bd_bounder,** for some of the cases seems to do better than the **bounder,** but is never better than the **frcut**, the idea to calculate bound from the nodes came from[2],this bound follows the rules of the sticker bound but the only difference is it is calculated from the current node and not the whole graph.

As the number of nodes increase the results become suboptimal,the data sets used in the test are considered to be “easy”, but according to paper[1], even the easy data sets take at least an hour to complete for the ILP solvers, the results for the larger graph may be sub-optimal but the trade off is the time.

Conclusion

The combinatorial branch-and-bound algorithm developed for the multicut problem demonstrates promising results for smaller graphs. It successfully identifies graph clusters while optimizing the sum of weights within these clusters.
The implementation includes three distinct bounding methods (bounder, frcut, and nd_bounder), with bounder generally providing the tightest bounds graphs.
Performance on Smaller Graphs: The algorithm appears to perform well on graphs with fewer nodes ($n < 12$), producing optimal or near-optimal solutions as compared to ILP solvers.
Bounding Tradeoffs: The stricter bounder method generally yields superior results for smaller graphs. However, in certain larger graphs, frcut can find better bounds, albeit at the cost of increased traversal and execution time.
Suboptimality with Larger Graphs: As the number of nodes and negative weights increase, the algorithm's solutions tend to become suboptimal compared to ILP solvers.
Time Efficiency: Despite potential suboptimality, the algorithm exhibits a significant advantage in runtime, completing calculations within minutes even for larger data sets.
Overall, the project offers a valuable foundation for further research into combinatorial approaches for the multicut problem. The insights gained on bounding methods and the tradeoffs between solution quality and execution time provide a strong starting point for further potential enhancements.

References.

[1] Michael M Sørensen and Adam N Letchford. Cp-lib: Benchmark instances of the clique partitioning problem. Mathematical Programming Computation, pages 1–19, 2023.
[2] Florian Jaehn and Erwin Pesch. New bounds and constraint propagation techniques for the clique partitioning problem. Discrete Applied Mathematics, 161(13-14):2025–2037, 2013.
[3] Jens Clausen. Branch and Bound Algorithms - Principles and Examples. 1999.
[4] M. Grotschel and Y. Wakabayasi-II. A cutting plane algorithm for a clustering problem . Mathematical Programming 45 (59-96), 1989.
[5] U. Dorndorf, E. Pesch, Fast clustering algorithms, ORSA Journal on Computing 6 (1994) 141–153.
[6] Debmalya Panigrahi. COMPSCI 638: Graph Algorithms,Lecture 5. 2019.

Penzen Lama
pela250e@tu-dresden.de