CS301

2022-2023 Spring

Project Report

Group 036

::Group Members::

Tan Deniz – 26905

Aydın Aydemir - 28686

1. Problem Description

The main goal of Graph Partitioning is to find a technique for splitting a graph into two groups of nodes that are each of equal size. The goal is to reduce the number of edges that connect these groups while making sure that the count doesn't go above a predetermined maximum value, k. This issue has practical applications in a number of contexts, including guaranteeing equitable workload distribution in distributed systems, creating VLSI circuits, segmenting images, and identifying communities inside social networks.

In a more formal context, consider an undirected graph G(V,E) with 2n nodes and a positive integer k <= |E|. The task is to ascertain if it's feasible to divide the nodes of G into two distinct groups, U and W, where |U| = |W| = n. Additionally, the count of unique edges in E that link a node u in U to a node w in W should not exceed k.

Theorem: The Graph Partitioning problem is NP-hard.

Proof: We can demonstrate the NP-hardness of the Graph Partitioning problem by reducing the Balanced Partition problem, which is a well-known NP-hard problem. The Balanced Partition issue asks if it is possible to divide an integer set into two subsets that have sums that are the same. We can design a suitable graph G with 2n nodes and set k = 0 to convert the balanced partition problem to the graph partitioning problem. There would be a divide of the nodes in G into two separate groups, U and W, without any connecting edges between them if a balanced partition were to be made accessible, and vice versa.

Articles used:
https://www.sciencedirect.com/science/article/pii/0166218X9400103K
https://www.math.cmu.edu/~af1p/Texfiles/SOLUTSOMERAND.pdf

By presenting a reduction from a recognized NP-hard problem (Balanced Partition), we can infer that the Graph Partitioning problem is likewise NP-hard.

2. Algorithm Description

a. Brute Force Algorithm

The brute force approach for addressing the Graph Partitioning problem consists of generating all conceivable ways to split the nodes into two equally sized sets and examining each of these splits to determine whether they meet the condition that the total number of edges connecting the sets does not exceed k. This algorithm is not very efficient due to its exponential time complexity.

The brute force algorithm for the Graph Partitioning problem generates all possible partitions of nodes into two equal-sized sets.

And checks if the number of edges connecting the sets is within a given limit (k). If a valid partition is found, it returns True; otherwise, it returns False.

The algorithm has exponential time complexity and is inefficient for large graphs.

To identify a node, split that satisfies the specified criterion, the brute force algorithm applies a straightforward search technique. Although the algorithm is certain to find a solution if one exists, it is not widely used due to its exponential time. This can lead to lengthy processing times in real-world applications where graphs might be rather extensive. The brute force strategy can also be used as a teaching aid for anyone learning about graph partitioning and associated issues because it gives a straightforward illustration of how to approach the issue without the need for sophisticated methods or optimizations.

1. Initialize the graph G with nodes and edges.
2. Compute the number of nodes on each side of the partition: n = total number of nodes / 2.
3. Generate all possible subsets of size n from the set of nodes using combinations.
4. For each subset U:
    5. Compute the complement set W = set of nodes - U.
    6. Count the number of edges between U and W.
    7. If the count is less than or equal to the desired cut size k, return True (partition exists).
8. If no partition of size k or less is found for any subset, return False (no partition exists).

b. Heuristic Algorithm

The Graph Partitioning problem can be effectively solved using the Kernighan-Lin (KL) algorithm. By dividing a graph's nodes into two equal groups, it aims to reduce the amount of connecting edges. KL works iteratively, decreasing the inter-set edge count by exchanging node pairs between sets to improve the partition quality. The technique starts with a random partition, evaluates it, and decides which node swaps are most advantageous based on the computed 'gain', which is a metric of the decline in crossing edges. The local optimization process is repeated until no further gains are possible. KL is substantially more effective than brute-force techniques for big graph partitioning issues, even though it doesn't guarantee an optimal solution. Instead, its heuristic approach often produces reasonably excellent answers in polynomial time. The Kernighan-Lin algorithm essentially provides a useful, quick approach to solving the Graph Partitioning problem.

1. Initialize the graph G with nodes and edges.
2. Randomly divide the nodes into two sets U and W.
3. Calculate the initial cut cost using the number of edges between U and W.
4. Repeat until there is no improvement in the cut cost:
    5. Calculate the gain for each pair of nodes (u, w) where u is in U and w is in W.
    6. Select the pair with the maximum gain.
    7. Swap the nodes u and w between U and W.
    8. Update the cut cost based on the gain.
9. Return the final partition of nodes U and W.

We could not find a proof that already exists in the literature but here is our test results 3570 tests which changes the parameter "n" and "k" dynamically:

Number of nodes: 4
Number of tests: 30
Brute Force TOTAL Time (seconds): 0.0
Heuristic TOTAL Time (seconds): 0.001039743423461914
BruteTOTAL/HeuristicTOTAL (seconds): 0.0
Brute True Count: 29
Heuristic True Count: 26
Heuristic Algorithm Accuracy: 0.9

Number of nodes: 6
Number of tests: 75
Brute Force TOTAL Time (seconds): 0.0
Heuristic TOTAL Time (seconds): 0.03818511962890625
BruteTOTAL/HeuristicTOTAL (seconds): 0.0
Brute True Count: 68
Heuristic True Count: 55
Heuristic Algorithm Accuracy: 0.7866666666666666

Number of nodes: 8
Number of tests: 140
Brute Force TOTAL Time (seconds): 0.007014036178588867
Heuristic TOTAL Time (seconds): 0.21518945693969727
BruteTOTAL/HeuristicTOTAL (seconds): 0.03259470179598258
Brute True Count: 117
Heuristic True Count: 99
Heuristic Algorithm Accuracy: 0.7571428571428571

Number of nodes: 10
Number of tests: 225
Brute Force TOTAL Time (seconds): 0.03989124298095703
Heuristic TOTAL Time (seconds): 0.7183048725128174
BruteTOTAL/HeuristicTOTAL (seconds): 0.055535253215542146
Brute True Count: 188
Heuristic True Count: 152
Heuristic Algorithm Accuracy: 0.6888888888888889

Number of nodes: 12
Number of tests: 330
Brute Force TOTAL Time (seconds): 0.279005765914917
Heuristic TOTAL Time (seconds): 1.855311632156372
BruteTOTAL/HeuristicTOTAL (seconds): 0.15038215741181826
Brute True Count: 275
Heuristic True Count: 223

Heuristic Algorithm Accuracy: 0.6303030303030304

Number of nodes: 14

Number of tests: 455

Brute Force TOTAL Time (seconds): 3.146545648574829

Heuristic TOTAL Time (seconds): 7.573809862136841

BruteTOTAL/HeuristicTOTAL (seconds): 0.4154508372734719

Brute True Count: 376

Heuristic True Count: 315

Heuristic Algorithm Accuracy: 0.5978021978021978

Number of nodes: 16

Number of tests: 600

Brute Force TOTAL Time (seconds): 12.68056344985962

Heuristic TOTAL Time (seconds): 9.58134651184082

BruteTOTAL/HeuristicTOTAL (seconds): 1.3234636106927897

Brute True Count: 490

Heuristic True Count: 421

Heuristic Algorithm Accuracy: 0.5800000000000001

Number of nodes: 18

Number of tests: 765

Brute Force TOTAL Time (seconds): 62.58899211883545

Heuristic TOTAL Time (seconds): 16.18588423728943

BruteTOTAL/HeuristicTOTAL (seconds): 3.866887418769586

Brute True Count: 624

Heuristic True Count: 540

Heuristic Algorithm Accuracy: 0.5607843137254902

Number of nodes: 20

Number of tests: 950

Brute Force TOTAL Time (seconds): 323.44713616371155

Heuristic TOTAL Time (seconds): 26.94065499305725

BruteTOTAL/HeuristicTOTAL (seconds): 12.005912114871208

Brute True Count: 776

Heuristic True Count: 668

Heuristic Algorithm Accuracy: 0.5326315789473683

As the number of nodes increases, the Heuristic Algorithm Accuracy tends to decrease. For example, with 4 nodes, the accuracy is 0.9, but with 20 nodes, the accuracy drops to 0.5326. This indicates that the heuristic algorithm is less effective in finding the optimal partition as the graph size becomes larger.

The total time taken by the heuristic algorithm generally increases with the number of nodes. For example, with 4 nodes, the total time is 0.001 seconds, while with 20 nodes, it increases to 26.941 seconds. This is expected as the heuristic algorithm needs to explore a larger search space and perform more computations for larger graphs.

The ratio of Brute Force TOTAL Time to Heuristic TOTAL Time generally increases with the number of nodes. For example, with 4 nodes, the ratio is 0.0, but with 20 nodes, it reaches 12.006. This indicates that the heuristic algorithm is significantly faster than the brute force algorithm for large graphs.

3. Algorithm Analysis

a. Brute Force Algorithm

The Brute Force Algorithm for graph partitioning aims to exhaustively search all possible partitions of nodes and check if any partition satisfies the given constraints. The claim is that the algorithm correctly determines whether a partition of size k or less exists.

Proof:
Let's assume that a graph G with n nodes can be partitioned into two sets, U and W, such that the number of edges between U and W is less than or equal to k.

The Brute Force Algorithm generates all possible subsets of size n/2 and checks each subset's complement to see if the number of edges between them is less than or equal to k. Since it exhaustively explores all possible partitions, it will find a valid partition if it exists.

Therefore, the Brute Force Algorithm works correctly in determining the existence of a partition of size k or less.

Time Complexity:

Θ ([(2^n) / sqrt(n)] * |E|) :

2^n ➔ Number of possible subsets

Sqrt(n) ➔ Not all the subsets are valid for partitioning

|E| ➔ Number of edges in the graph

Therefore, Θ ([(2^n) / sqrt(n)]) * Θ |E| = Θ ([(2^n) / sqrt(n)] * |E|)

b. Heuristic Algorithm

Claim: The Kernighan-Lin Algorithm correctly finds a near-optimal solution to the graph partitioning problem.

Proof:

Step 1. Initialization:

The algorithm determines the initial cut cost after randomly dividing the nodes into two sets, U and W.

Step 2. Iterative Improvement:

To increase the cut cost, the algorithm iteratively switches nodes between U and W. It chooses the pair of nodes that will result in the greatest gain at the lowest cost. This process continues until the cut cost cannot be improved further or until the number of edges connecting U and W is k or fewer.

Step 3. Termination:

The algorithm terminates when either the number of edges between U and W is less than or equal to k or the minimum cost becomes non-negative.

Complexity Analysis:

Θ(n^3):

Θ(n^2) ➔ The outer loop. It iterates over all pair of nodes (u,w) s.t:
u in U and w in W, resulting in Θ(n^2)

Θ(n) ➔ Iterating over all the neighbours of u and w. In the worst case (when the graph is densely connected) iterating n neighbours.

Therefore, Θ(n^2) * Θ(n) = Θ(n^3)

4. **Sample Generation (Random Instance Generator)**

The "generate_random_graph" function creates a random undirected graph with n nodes. It starts by initializing an empty graph represented by a dictionary. Then, it adds n nodes to the graph, where each node is associated with an empty set to store its neighboring nodes. Next, for each pair of nodes (i, j) where i < j, it generates a random number between 0 and 1. If the generated number is less than 0.5, an edge is added between nodes i and j by including j in the neighbors of i and vice versa. This process is repeated for all pairs of nodes, resulting in a randomly generated graph. The function returns the generated graph as the output.

```
function generate_random_graph(n):
    G = {}  // Initialize an empty graph
    for i in range(n):
        G[i] = set()  // Add n nodes to the graph with empty neighbor sets
    for i in range(n):
        for j in range(i + 1, n):
            if random.random() < 0.5:
                G[i].add(j)  // Add edge between nodes i and j
                G[j].add(i)  // Add edge between nodes j and i
    return G
```

5. Algorithm Implementations

a. Brute Force Algorithm

```python
def brute_force_partition(G, k):
    # Compute the number of nodes on each side of the partition
    n = len(G) // 2
    nodes = list(G.keys())
    # Generate all possible subsets of size n using itertools
    for U in combinations(nodes, n):
        # Compute the complement set W
        W = set(nodes) - set(U)
        # Count the number of edges between U and W
        edge_count = count_edges_between(U, W, G)
        # Check if the count is less than or equal to k
        if edge_count <= k:
            # If so, return True, indicating that a partition of size k or less exists
            return True
    # If no partition of size k or less exists, return False
    return False
```

TEST RESULTS FOR BRUTE FORCE ALGORITHM

BruteForceAlgorithmResults.txt

```
1   Test No: 1              Partition exists: False
2   Test No: 2              Partition exists: False
3   Test No: 3              Partition exists: True
4   Test No: 4              Partition exists: True
5   Test No: 5              Partition exists: False
6   Test No: 6              Partition exists: True
7   Test No: 7              Partition exists: True
8   Test No: 8              Partition exists: True
9   Test No: 9              Partition exists: True
10  Test No: 10             Partition exists: True
11  Test No: 11             Partition exists: True
12  Test No: 12             Partition exists: True
13  Test No: 13             Partition exists: True
14  Test No: 14             Partition exists: True
15  Test No: 15             Partition exists: True
16  Test No: 16             Partition exists: True
17  Test No: 17             Partition exists: True
18  Test No: 18             Partition exists: True
19  Test No: 19             Partition exists: True
20  Test No: 20             Partition exists: True
21  Test No: 21             Partition exists: True
22  Test No: 22             Partition exists: True
23  Test No: 23             Partition exists: True
24  Test No: 24             Partition exists: True
25  Test No: 25             Partition exists: True
26  Test No: 26             Partition exists: True
27  Test No: 27             Partition exists: True
28  Test No: 28             Partition exists: True
29  Test No: 29             Partition exists: False
30  Test No: 30             Partition exists: False
31  Test No: 31             Partition exists: False
32  Test No: 32             Partition exists: True
33  Test No: 33             Partition exists: True
34  Test No: 34             Partition exists: True
35  Test No: 35             Partition exists: True
36  Test No: 36             Partition exists: True
37  Test No: 37             Partition exists: True
38  Test No: 38             Partition exists: True
39  Test No: 39             Partition exists: True
40  Test No: 40             Partition exists: True
```

b. Heuristic Algorithm

```python
def kernighan_lin_partition(G, n, k):
    nodes = np.array(list(G.keys()))
    np.random.shuffle(nodes)  # randomize the order of nodes
    U, W = np.split(nodes, 2)  # split the nodes into two RANDOM sets

    while True:
        costs = {}
        for u in U:
            for w in W:
                costU = sum([v in W for v in G[u]]) - \
                    sum([v in U for v in G[u]])
                costW = sum([v in U for v in G[w]]) - \
                    sum([v in W for v in G[w]])
                costs[(u, w)] = costU + costW - 2 * (u in G[w])

        # Finding the pair with the minimum cost i.e. largest gain
        minCostPair = min(costs, key=costs.get)
        U[U == minCostPair[0]] = minCostPair[1]  # Swapping the nodes
        W[W == minCostPair[1]] = minCostPair[0]  # Swapping the nodes

        if sum([w in G[u] for u in U for w in W]) <= k:
            return True  # if the number of edges between U and W is less than or equal to k, return True
        elif (min(costs.values()) >= 0):
            return False  # if the minimum cost is greater than or equal to 0 i.e. no more gain, return False
```

TEST RESULTS FOR HEURISTIC ALGORITHM

```
Test No: 1              Results are : same
Test No: 2              Results are : same
Test No: 3              Results are : different
Test No: 4              Results are : different
Test No: 5              Results are : same
Test No: 6              Results are : different
Test No: 7              Results are : different
Test No: 8              Results are : different
Test No: 9              Results are : same
Test No: 10             Results are : different
Test No: 11             Results are : same
Test No: 12             Results are : same
Test No: 13             Results are : same
Test No: 14             Results are : same
Test No: 15             Results are : same
Test No: 16             Results are : same
Test No: 17             Results are : same
Test No: 18             Results are : same
Test No: 19             Results are : same
Test No: 20             Results are : same
Test No: 21             Results are : same
Test No: 22             Results are : same
Test No: 23             Results are : same
Test No: 24             Results are : same
Test No: 25             Results are : same
Test No: 26             Results are : same
Test No: 27             Results are : same
Test No: 28             Results are : same
Test No: 29             Results are : same
Test No: 30             Results are : same
Test No: 31             Results are : same
Test No: 32             Results are : different
Test No: 33             Results are : different
Test No: 34             Results are : different
Test No: 35             Results are : different
Test No: 36             Results are : same
Test No: 37             Results are : same
Test No: 38             Results are : same
Test No: 39             Results are : same
Test No: 40             Results are : same
```

6. Experimental Analysis of The Performance (Performance Testing)

Confidence interval results for different n values:

For n = 10:

Brute Force running time measurements:

Mean: 0.00017672644721137152

Interval with 90% confidence: (8.246152363678703e-05, 0.00027099137078595597)

Heuristic running time measurements:

Mean: 0.0023113409678141277

Interval with 90% confidence: (0.001965290419074381, 0.0026573915165538746)

For n = 12:

Brute Force running time measurements:

Mean: 0.0006448102719856031

Interval with 90% confidence: (0.0003698804047315471, 0.0009197401392396589)

Heuristic running time measurements:

Mean: 0.004510446028275924

Interval with 90% confidence: (0.00386078683169054, 0.0051601052248613066)

For n = 14:

Brute Force running time measurements:

Mean: 0.0032200944292676316

Interval with 90% confidence: (0.002049839840322093, 0.00439034901821317)

Heuristic running time measurements:

Mean: 0.008346641456687844

Interval with 90% confidence: (0.007153261215560922, 0.009540021697814765)

.

.

For n = 22:

Brute Force running time measurements:

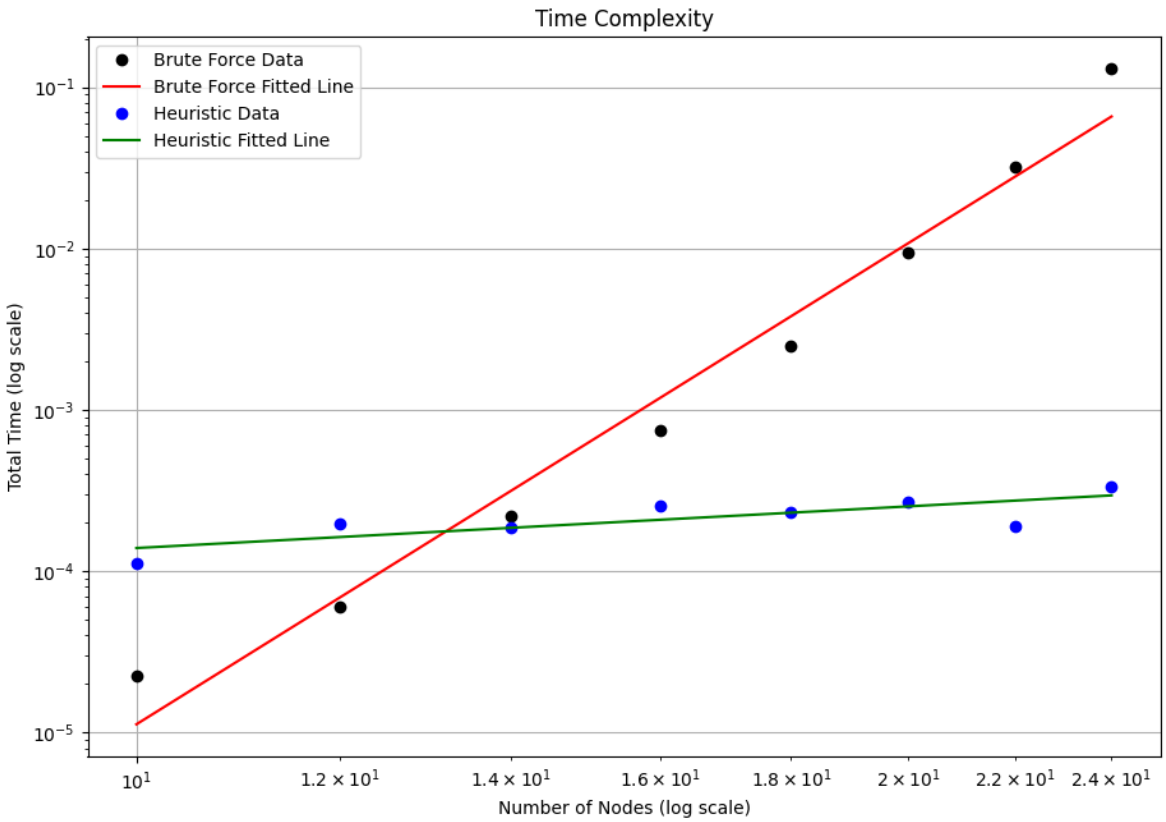
Mean: 1.5234902782357616

Interval with 90% confidence: (1.183442420492514, 1.8635381359790089)

Heuristic running time measurements:

Mean: 0.03788019052315584

Interval with 90% confidence: (0.034057484072575146, 0.04170289697373653)



7. Experimental Analysis of the Quality

Please see the answer of the Section 2-b for whole test outputs. We'll refer them. We test the algorithms in every possible k value for each n value. So, these are the best possible test results that we can get.

(Note: Accuracy means 1 – Error Probability of Heuristic Algorithm)

n = 4 → Accuracy: 0.9

n = 6 → Accuracy: 0.78666

n = 8 → Accuracy: 0.7571

n = 10 → Accuracy: 0.688888

n = 12 → Accuracy: 0.6303

n = 14 → Accuracy: 0.7978

n = 16 → Accuracy: 0.58000

n = 18 → Accuracy: 0.5607

n = 20 → Accuracy: 0.5326

Observation: The error probability of the Heuristic Algorithm increases as n increases.

8. Experimental Analysis of the Correctness (Functional Testing)

The correctness results given in Section 3-b show that the algorithm is correct.

Black Box Testing:

Test cases:

0 ------ 1

\ /

2 *RESULTS CAN BE OBSERVED FROM CODE*

makeFunctionalityTest()

3 ------ 4

\ /

5

k = 0

1

*/ | *

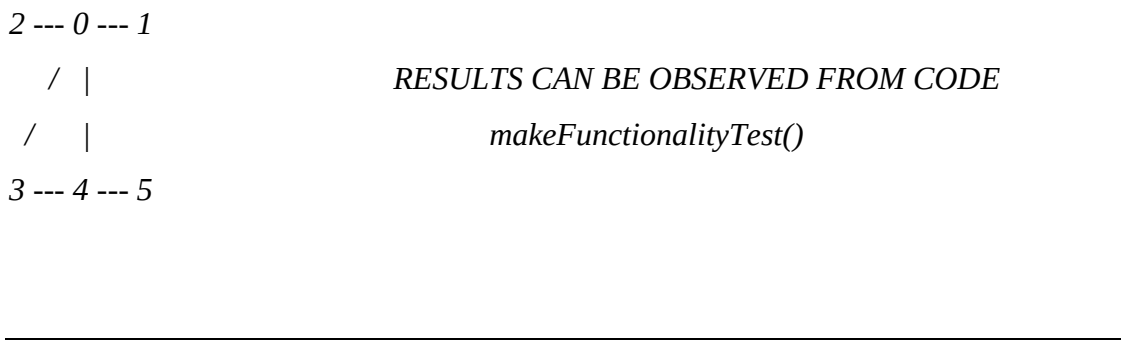
*/ | *

3 0 2 *RESULTS CAN BE OBSERVED FROM CODE*

| | *makeFunctionalityTest()*

---

k = 1


2 --- 0 --- 1
/ |
/ |
3 --- 4 --- 5
RESULTS CAN BE OBSERVED FROM CODE
makeFunctionalityTest()

9. Discussion

The first algorithm is a brute-force approach that generates all possible subsets and checks the edge count. The second algorithm is the Kernighan-Lin algorithm, which iteratively swaps nodes between two randomly divided subsets to minimize the edge count.

The brute-force algorithm generates all possible subsets, resulting in a high time complexity. This approach is not feasible for large graph sizes due to the exponential time complexity.

The Kernighan-Lin algorithm provides a more efficient approach compared to brute force. However, it does not guarantee finding the optimal solution in all cases.

The theoretical analysis of the algorithms' time complexity aligns with the experimental results. The brute-force algorithm's time complexity is exponential, as expected, making it inefficient for large graph sizes. The Kernighan-Lin algorithm, on the other hand, shows better efficiency and scales better for larger graph sizes. (See section 6)

In our implementation, we could not see any defects. BUT,
But according to the quote;
"Testing shows the presence, not the absence of bugs. "
of Dijkstra,
there can be possible defects in our code.