CS301

2022-2023 Spring

Project Report

Group 094

Ege Zorlutuna 27853

Furkan Kerim Yıldırım 28138

1. Problem Description

Our Problem:

Input: 2n-node undirected graph G(V,E); positive integer k<=|E|.

Question: Can the nodes of G be partitioned into 2 disjoint sets U and W each of size n and such that the total number of distinct edges in E that connect a node u in U to a node w in W is at most k?

The aforementioned problem is asking whether it is possible to divide the nodes of G into two equal-sized groups such that the number of edges that cross the partition (i.e., have one endpoint in U and the other endpoint in W) is at most k. Since the solution to this problem would be yes or no indicating the existence of such partition, it is a decision problem. Stockmeyer (1976, pp. 242-243) in their theorem of Simple Max Cut α Minimum Cut Into Equal-Sized Subsets, describes this problem as the "Minimum Cut Into Equal-Sized Subsets", where the input is Graph G = (N, A), two distinguished nodes s and t, positive integer W, there is a partition $N = S_1 \cup S_2$ with $S_1 \cap S_2 = \varnothing$, $|S_1| = |S_2|$, $s \in S_1$, $t \in S_2$, and $|\{\{u,v\} \in A: u \in S_1, v \in S_2\}| <= W$. After describing the problem as mentioned, they conclude that this problem is NP-complete from the completeness of Simple Max Cut and the following proof by reduction:

Given a graph G = (N, A) and positive integer W, as input for Simple Max Cut, let n = |N| and $U = \{u_1, u_2, \ldots, u_n\}$ satisfy $U \cap N = \varnothing$. The corresponding input for Minimum Cut Into Equal-Sized Subset is the graph G' = (N', A'), nodes u_1 and u_n, and positive integer W', defined as follows:

$N' = N \cup U$;

$A' = \{\{u, v\}: u, v \in N' \text{ and } \{u, v\} \notin A\}$;

$W' = n^2 - W$.

Supposing there is a partition $N = S_1 \cup S_2$ such that $|\{\{u, v\} \in A: u \in S_1, v \in S_2\}| \geq W$. Since W is positive, both S_1 and S_2 are nonempty. Let $j = n - |S_1|$. Form $S'_1 = S_1 \cup \{u_1,u_2, \ldots,u_j\}$ and $S'_2 = N' - S'_1$. Then $N' = S'_1 \cup S'_2$ is a partition for G' with $|S'_1| = |S'_2| = n$, $u_1 \in S'_1$, $u_n \in S'_2$, and

$$|\{\{u, v\} \in A': u \in S'_1, v \in S'_2\}| = n^2 - |\{\{u, v\} \notin A': u \in S'_1, v \in S'_2\}|$$

$$= n^2 - |\{\{u, v\} \in A: u \in S_1, v \in S_2\}|$$
$$\leq n^2 - W = W'.$$

Now suppose there is a partition $N' = S'_1 \cup S'_2$, with $u_1 \in S'_1$, and $u_n \in S'_2$, and $|S'_1| = |S'_2| = n$ such that $|\{\{u, v\} \in A': u \in S'_1, v \in S'_2\}| \leq n^2 - W = W'$. Then $N = S_1 \cup S_2$, where $S_1 = S'_1 \cap N$ and $S_2 = S'_2 \cap N$, is a partition for G satisfying

$$|\{\{u, v\} \in A: u \in S_1, v \in S_2\}| = |\{\{u, v\} \notin A': u \in S'_1, v \in S'_2\}|$$
$$= n^2 - |\{\{u, v\} \in A': u \in S'_1, v \in S'_2\}|$$
$$\geq n^2 - (n^2 - W) = W.$$

Therefore G has a cut of weight greater than or equal to W if and only if G' has a cut weight not exceeding W', which separates u_1 and u_n and divides the nodes of the graph into two equal-sized subsets. The reduction is proven. □

This problem can also be described as an optimization problem that is intended to partition the vertices of a given graph into two equal halves so as to minimize the number of those edges with exactly one end in each half. This version of the problem is called "the minimum bisection problem" which is a well-known problem in the field of computer science.

The problem has a variety of real-life applications. As Kerninghan (1970) mentioned, the problem arises in many physical conditions such as "in assigning the components of electronic circuits to circuit boards to minimize the number of connections between boards". Besides from this example, some other applications of this problem might be used in the areas of supply chain optimizations, image segmentations, transportation planning, and etc.

2. Algorithm Description

a. Brute Force Algorithm

A brute force algorithm for this problem would be to enumerate all possible bipartitions of the graph into two sets of size n and then check if the number of crossing edges is at most k. Since the number of possible bipartitions is exponential in the number of nodes, the running time of this algorithm is exponential and not practical for large graphs.

Algorithm:

1 - Generate all possible partitions of the set V into two disjoint sets U and W, each of size n.

2 - For each partition (U, W), count the number of edges connecting nodes in U to nodes in W.

3 - If there is a partition (U, W) with the total number of distinct edges connecting nodes in U to nodes in W being at most k, return True. Otherwise, return False.

As mentioned before, the given algorithm is an instance of a brute force search algorithm, which searches all possible solutions to a problem one by one to find the solution to the problem. Such algorithms work in a straightforward way that involves generating all possible solutions. With the brute force search algorithm mentioned above, the solution is guaranteed no matter the computational complexity.

b. Heuristic Algorithm

An efficient heuristic algorithm that would solve the problem is Kernighan–Lin algorithm. The Keringhan-Lin algorithm is an optimization algorithm that aims to find a partitioning of a graph into two equal-sized sets while trying to minimize the number of edges that crosses the partition (the total number of distinct edges in E that connect a node u in U to a node w in W in our problem description) (Kerninghan, 1970). Since the aforementioned algorithm is an optimization algorithm, it tries to minimize the number of crossing edges. In order to apply this optimization algorithm to our decision problem, the algorithm is slightly updated with a check after the partitioning. As it can be seen in the implementation part of this heuristic algorithm (Section 5-b), it is checked if the number of crossing edges is smaller than the k value after partitioning the graph with the Keringhan-Lin algorithm. If it is smaller than the k value, our algorithm returns the message "Found a partition with at most k distinct edges." and the number of distinct crossing edges, otherwise it returns the message "No partition with at most k distinct edges found.". So, mostly the main logic of the Kernighan-Lin algorithm is preserved, but with an additional check added about the number of distinct edges between partitions in this implementation to solve our decision problem.

The main logic behind our implementation and the Kernighan-Lin algorithm could be easily explained by looking at the pseudocode.

Algorithm:

1- Initial Partition: the algorithm starts by partitioning the graph by arbitrarily dividing the given set of nodes into two disjoint sets.

2- Compute D (difference) Values: For each node, calculate the difference between the edges of that node that crosses the partition and the ones that don't. In other words, the D value is the difference between the number of edges connecting the node to nodes in the other set and the number of edges connecting it to nodes in its own set.

3- Compute Max Gain: Find two nodes in two different sets partitioned earlier that would result in the maximum gain if swapped, and find the gain value.

4- Swap Nodes: If the maximum gain is more than 0, swap the nodes resulting in the maximum gain and update the sets.

5- Termination: If the maximum gain is less or equal to 0, no further improvements can be made by swapping the nodes. Calculate the number of distinct edges between two disjoint sets, and check if it is smaller or equal to the k value, print appropriate message.

This algorithm is a version of greedy algorithms, specifically using the iterative improvement method since it starts with an initial solution and tries to improve it with each iteration hoping that with the local modification and improvements would result in a global improvement. In this case, the Kernighan-Lin algorithm swaps pairs of nodes from different partitions with maximum gain to decrease the number of edges crossing partitions. Since this is a heuristic algorithm, it is not guaranteed that the partition made by the algorithm will result in the optimal solution.

3. Algorithm Analysis

a. Brute Force Algorithm

Correctness:

While proving that a brute force algorithm works correctly, we need to show that it will always output the correct answer. In this case, it needs to decide correctly whether such partitioning exists or not.

Claim:

The algorithm works correctly and returns True if and only if there is a partition of the nodes into disjoint sets U and W such that the number of distinct edges connecting nodes in U to nodes in W is at most k.

Proof:

To see why this is true, suppose that there exists a valid partitioning of the graph into two sets U and W such that the number of crossing edges is at most k. Then, the brute force algorithm will eventually consider this partitioning and output a positive answer. On the other hand, if there does not exist a valid partitioning of the graph, then the brute force algorithm will consider all possible partitions and output a negative answer. This is because the algorithm checks all possible partitions, and if none of them satisfies the condition, then there does not exist a valid bipartition. Therefore the algorithm works correctly.

Complexity:

Given brute force algorithm consists of two main parts that would have an impact on its complexity: generating all possible partitions and counting the number of edges connecting nodes in U to nodes in W.

Generating all possible partitions: The number of possible partitions of the set V into two disjoint sets U and W, each of size n, is given by the binomial coefficient C(|V|, n), which can be computed as:

$$C(|V|, n) = (|V|!)/(n! * (|V|-n)!)$$

This is the number of ways to choose n elements from a set of size |V|, and it has exponential growth with respect to |V|. Since the algorithm must generate and check each of these partitions, the time complexity is at least O(C(|V|, n)).

Counting the number of edges connecting nodes in U to nodes in W: For each partition, we need to iterate through all edges in E to count those connecting nodes in U to nodes in W. This step has a time complexity of O(|E|) for each partition. Combining both steps, the worst-case time complexity is O(C(|V|, n) * |E|). While it is not a tight upper bound, it is a reasonable estimate for the algorithm's time complexity.

As for the space complexity, we need to store the graph G(V,E), which takes O(|V| + |E|) space. Additionally, we need to store the partitions, which can be done using O(|V|) space. Therefore, the space complexity is O(|V| + |E|).

As a result, the brute force algorithm to solve this problem has an exponential worst-case time complexity of O(C(|V|, n) * |E|) and space complexity of O(|V| + |E|). This algorithm is not efficient for large graphs (large values of n), but it can be used for small instances or as a baseline for comparing more sophisticated algorithms.

b. Heuristic Algorithm

Correctness:

While showing that the algorithm works correctly, a proof by contradiction will be used.

Claim:

The Kernighan-Lin algorithm will always find a partition where it is not possible to reduce the number of crossing edges with a node swap operation.

Proof by Contradiction:

Assuming that the Keringhan-Lin algorithm finds a partition such that there is a pair of nodes in disjoint sets whose swap would reduce the number of crossing edges. This means that their gain of swapping would be bigger than 0. But the algorithm terminates only if there are no possible swaps such that their maximum gain is smaller or equal to 0. This is a contradiction, our assumption is false.

Therefore the algorithm will always find a partition where it is not possible to reduce the number of crossing edges with a single node swap operation. Then it will compare the cut size (number of crossing edges) with the k value and return the proper output. This step is trivial since it will return true if the cut size is smaller or equal to the k value, or it will return false if the cut size is bigger than k. Therefore the algorithm works correctly.

Complexity:

Making the analysis of the algorithm step by step. Most of the steps defined in the algorithm description refer to a method in the implementation.

1- Initial Partition: The complexity of this method is O(n). The list of the nodes is shuffled (O(n)) and split into two disjoint sets, which is also O(n).

2- Computing D Values: In this method, for each node, a summation value (the D (difference) value) over its neighbors is calculated. In the worst-case scenario, the graph will be dense and each node will have exactly n-1 neighbors, resulting in a complexity of $O(n^2)$ for this method. If the graph is sparse, it can be closer to O(n).

3- Computing Max Gain: This method will iterate over all pairs of nodes in our disjoint sets, making the complexity $O((n/2)^2) = O(n^2)$, for each pair of nodes, it does a constant amount of work.

4- Swap Nodes: Since this method does a constant amount of work, its complexity is O(1).

5- Termination of K-L Algorithm: our run() method forms the main loop of the Keringhan-Lin algorithm part of our code. Its complexity depends on the size of our graph and how quickly the Keringhan-Lin will converge into a solution. In the worst-case scenario for the number of iterations would be that in each iteration, there is a pair of nodes that can be swapped to improve the gain, and the algorithm continues to find these pairs until it has swapped every node once. This operation would result in n/2 swaps since the size of each disjoint set is n/2 (running K-L the algorithm n/2 times).

Up until now, looking at the Keringhan-Lin part of the algorithm, in the worst case the complexity will be $O(n/2 * (n + n^2 + n^2 + 1)) = O (n^3)$.

After these steps, the algorithm will continue with the calculation of the number of cuts by the calculate_cut method. This method has a worst-case time complexity of $O(n^2)$ since it considers every pair of nodes in the disjoint sets after the partitioning, and for each pair checks if one node is in the neighbors of the other. Making the total time complexity $O(n^3+n^2)$. Therefore the overall time complexity would be $O(n^3)$.

The space complexity of the algorithm depends mainly on the graph representation which is O(n + e), and the D dictionary that also uses O(n) space, so in total, the space complexity is O(n + e).

4. Sample Generation (Random Instance Generator)

The implementation of the sample generator, which will generate a graph to input the number of nodes n and returns the graph.

Algorithm:

1- Create an empty graph object

2- Add nodes to the graph

3- Select each pair of nodes in the graph and randomly decide whether to add an edge between them

The implementation of this algorithm on python is simple:

```
import networkx as nx
import random

def sample_generator(n: int) -> nx.Graph:
    """
    Generates a random graph with n nodes and returns it.

    Args:
        n (int): The number of nodes in the graph. It has to be even.

    Returns:
        A random graph with n nodes.
    """

    # create an empty graph object
    G = nx.Graph()

    # add nodes to the graph
    for i in range(n):
        G.add_node(i)

    # add edges to the graph
    for i in range(n):
        for j in range(i+1, n):
            # randomly decide whether to add an edge between nodes
```

```
            if random.random() < 0.5:
                G.add_edge(i, j)

    # return the graph
    return G
```

5. Algorithm Implementations

a. Brute Force Algorithm

The algorithm is implemented using Python 3.11 according to the pseudocode in Section 2.a can be found in “brute.py” and it is as follows:

```
start_time = time.time()
for comb in itertools.combinations(G.nodes, n//2):
    U = set(comb)
    W = set(G.nodes)- U

    distinct_edges = 0
    for i, j in G.edges:
        if (i in U and j in W) or (i in W and j in U):
            distinct_edges += 1

    if distinct_edges <= k:
        print("Found a partition with at most k distinct edges.")
        print("U = ", U)
        print("W = ", W)
        print("Distinct edges = ", distinct_edges)
        break
else:
    print("No partition with at most k distinct edges found.")

print("--- %s seconds ---"% round(time.time() - start_time, 3))
```

In order to measure the efficiency of the algorithm, the time from the beginning to the end of the algorithm was calculated. The evaluated graph is generated with the samples_generator created in Section 4. The results are as follows.

Case 1:

Enter number of nodes (must be even): 6

Number of nodes = 6, Number of edges = 10

Enter k (must be less than number of edges): 4

No partition with at most k distinct edges found.

--- 0.001 seconds ---

Case 2:

Enter number of nodes (must be even): 10

Number of nodes = 10, Number of edges = 24

Enter k (must be less than number of edges): 8

No partition with at most k distinct edges found.

--- 0.008 seconds ---

Case 3:

Enter number of nodes (must be even): 10

Number of nodes = 10, Number of edges = 20

Enter k (must be less than number of edges): 15

Found a partition with at most k distinct edges.

U = {0, 1, 2, 3, 4}

W = {5, 6, 7, 8, 9}

Distinct edges = 10

--- 0.0 seconds ---

Case 4:

Enter number of nodes (must be even): 16

Number of nodes = 16, Number of edges = 68

Enter k (must be less than number of edges): 10

No partition with at most k distinct edges found.

--- 0.338 seconds ---

Case 5:

Enter number of nodes (must be even): 16

Number of nodes = 16, Number of edges = 55

Enter k (must be less than number of edges): 20
Found a partition with at most k distinct edges.
U = {0, 1, 2, 4, 6, 7, 13, 15}
W = {3, 5, 8, 9, 10, 11, 12, 14}
Distinct edges = 20
--- 0.026 seconds ---

Case 6:
Enter number of nodes (must be even): 20
Number of nodes = 20, Number of edges = 100
Enter k (must be less than number of edges): 30
No partition with at most k distinct edges found.
--- 7.656 seconds ---

Case 7:
Enter number of nodes (must be even): 20
Number of nodes = 20, Number of edges = 92
Enter k (must be less than number of edges): 40
Found a partition with at most k distinct edges.
U = {0, 1, 2, 3, 4, 5, 7, 8, 18, 19}
W = {6, 9, 10, 11, 12, 13, 14, 15, 16, 17}
Distinct edges = 40
--- 0.024 seconds —

Case 8:
Enter number of nodes (must be even): 26
Number of nodes = 26, Number of edges = 161
Enter k (must be less than number of edges): 80
Found a partition with at most k distinct edges.
U = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 13}
W = {12, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25}
Distinct edges = 78
--- 0.0 seconds ---

Case 9:

Enter number of nodes (must be even): 26

Number of nodes = 26, Number of edges = 156

Enter k (must be less than number of edges): 70

Found a partition with at most k distinct edges.

U = {0, 1, 2, 3, 4, 5, 6, 8, 9, 10, 17, 20, 23}

W = {7, 11, 12, 13, 14, 15, 16, 18, 19, 21, 22, 24, 25}

Distinct edges = 70

--- 0.515 seconds ---

Case 10:

Enter number of nodes (must be even): 26

Number of nodes = 26, Number of edges = 167

Enter k (must be less than number of edges): 50

No partition with at most k distinct edges found.

--- 796.39 seconds ---

Case 11:

Enter number of nodes (must be even): 26

Number of nodes = 26, Number of edges = 162

Enter k (must be less than number of edges): 60

No partition with at most k distinct edges found.

--- 770.291 seconds ---

Case 12:

Enter number of nodes (must be even): 30

Number of nodes = 30, Number of edges = 212

Enter k (must be less than number of edges): 100

Found a partition with at most k distinct edges.

U = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 19}

W = {13, 15, 16, 17, 18, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29}

Distinct edges = 99

--- 0.002 seconds ---

Case 13:

Enter number of nodes (must be even): 30

Number of nodes = 30, Number of edges = 217

Enter k (must be less than number of edges): 90

Found a partition with at most k distinct edges.

U = {0, 1, 2, 3, 4, 5, 6, 7, 10, 12, 15, 20, 23, 25, 26}

W = {8, 9, 11, 13, 14, 16, 17, 18, 19, 21, 22, 24, 27, 28, 29}

Distinct edges = 90

--- 8.457 seconds —

Case 14:

Enter number of nodes (must be even): 30

Number of nodes = 30, Number of edges = 231

Enter k (must be less than number of edges): 90

--- Terminated after 10800 second ---

Case 15:

Enter number of nodes (must be even): 30

Number of nodes = 30, Number of edges = 208

Enter k (must be less than number of edges): 85

Found a partition with at most k distinct edges.

U = {0, 1, 2, 3, 7, 13, 14, 15, 16, 17, 18, 19, 21, 24, 26}

W = {4, 5, 6, 8, 9, 10, 11, 12, 20, 22, 23, 25, 27, 28, 29}

Distinct edges = 85

--- 574.393 seconds —

Case 16:

Enter number of nodes (must be even): 30

Number of nodes = 30, Number of edges = 189

Enter k (must be less than number of edges): 60

--- Terminated after 10800 second ---

As a result of the samples, it is possible to see that the brute force algorithm gives very fast results up to 20 nodes. Since it tries all possible combinations, it outputs between 20 nodes and 30 nodes in an average and acceptable time, while giving a very slow result above 30 nodes. It would be more appropriate to use heuristic algorithms to speed up this time.

b. Heuristic Algorithm

The algorithm is implemented using Python 3.11 according to the pseudocode in Section 2.b can be found in “heuristic.py” and it is as follows:

```
import random
import networkx as nx
from typing import Any, Dict, List, Set, Tuple

class KernighanLinAlgorithm:
    def __init__(self, G: nx.Graph, k: int):
        self.Graph = G
        self.k = k

    def run(self):
        U, W = self.initial_partition()
        while True:
            D = self.compute_D_values(U, W)
            a, b, gain = self.compute_max_gain(U, W, D)

            if gain <= 0:
                distinct_edges = self.calculate_cut(U, W)

                if distinct_edges <= self.k:
                    print("Found a partition with at most k distinct
edges.")
                    print("U = ", U)
                    print("W = ", W)
                    print("Distinct edges = ", distinct_edges)
                    return True


                else:
                    print("No partition with at most k distinct edges
found.")
```

```
                    return False

            U, W = self.swap_nodes(U, W, a, b)

    def initial_partition(self) -> Tuple[Set[Any], Set[Any]]:
        nodes = list(self.Graph.nodes)
        random.shuffle(nodes)  # Ensure randomness of initial
partition
        mid = len(nodes) // 2
        return set(nodes[:mid]), set(nodes[mid:])

    def compute_D_values(self, U: Set[Any], W: Set[Any]) -> Dict[Any,
int]:
        D = {}
        for node in U:
            D[node] = sum(1 if neighbor in W else -1 for neighbor in
self.Graph[node])

        for node in W:
            D[node] = sum(1 if neighbor in U else -1 for neighbor in
self.Graph[node])

        return D

    def compute_max_gain(self, U: Set[Any], W: Set[Any], D: Dict[Any,
int]) -> Tuple[Any, Any, int]:
        max_gain = float('-inf')
        for a in U:
            for b in W:
                cost = D[a] + D[b] - 2 * (1 if b in self.Graph[a] else
0)
                if cost > max_gain:
                    max_gain = cost
                    max_pair = a, b

        return max_pair[0], max_pair[1], max_gain

    def swap_nodes(self, U: Set[Any], W: Set[Any], a: Any, b: Any) ->
Tuple[Set[Any], Set[Any]]:
        # Ensure the swap doesn't imbalance the sets
        if len(U) == len(W):
```

```
                U.remove(a)
                W.remove(b)
                U.add(b)
                W.add(a)

        return U, W

    def calculate_cut(self, U: Set[Any], W: Set[Any]) -> int:
        cut = sum(1 for a in U for b in W if a in self.Graph[b])
        return cut
```

The heuristic algorithm terminates faster than our heuristic algorithm. The algorithm ran on the inputs for n = 10, 12, 14, 16, 18, 20, 22 and 200 times for each input. The full results of the run samples can be found in the "heuristic_sample_results.csv". Some test cases and results can be found here as an example:

n	k value	k selection	run time	result
10	23	k	0.000161	TRUE
10	11	k/2	0.000121	TRUE
10	10	random	0.000161	TRUE
12	0	0	0.000524	FALSE
12	32	k	0.000461	TRUE
12	16	k/2	0.000286	TRUE
14	48	k	0.000403	TRUE
14	24	k/2	0.000339	TRUE
14	9	random	0.000446	FALSE
16	0	0	0.000284	FALSE
16	60	k	0.000569	TRUE
16	30	k/2	0.000413	TRUE
18	76	k	0.000941	TRUE

18	38	k/2	0.000773	TRUE
18	67	random	0.000546	TRUE
20	0	0	0.000863	FALSE
20	95	k	0.000879	TRUE
20	47	k/2	0.000537	TRUE
22	124	k	0.001173	TRUE
22	62	k/2	0.001411	TRUE
22	87	random	0.001071	TRUE

6. Experimental Analysis of The Performance (Performance Testing)

The Heuristic algorithm has been performed 200 times for n=10, 20, 50, 100, 200, and 500. Then, the low and upper bounds with a 90% confidence interval were calculated. The "main.py" file is used to call the heuristic algorithm with the intended input (n) and the number of iterations (200). The performance test results have been saved into the "performance_test.csv" file. The analysis made on the data file can be found in the "analysis.ipynb" file. The performance test results are as follows:

n = 10, sample mean = 0.00022448658943171162, sample std = 9.147545214519521e-05, lower bound = 0.00021381390886525607, upper bound = 0.00023515926999816716

n = 20, sample mean = 0.001045851707458447, sample std = 0.0015551870240527873, lower bound = 0.0008644039644894734, upper bound = 0.0012272994504274207

n = 50, sample mean = 0.007574962377548165, sample std = 0.0014754115495123465, lower bound = 0.007402822248117478, upper bound = 0.007747102506978852

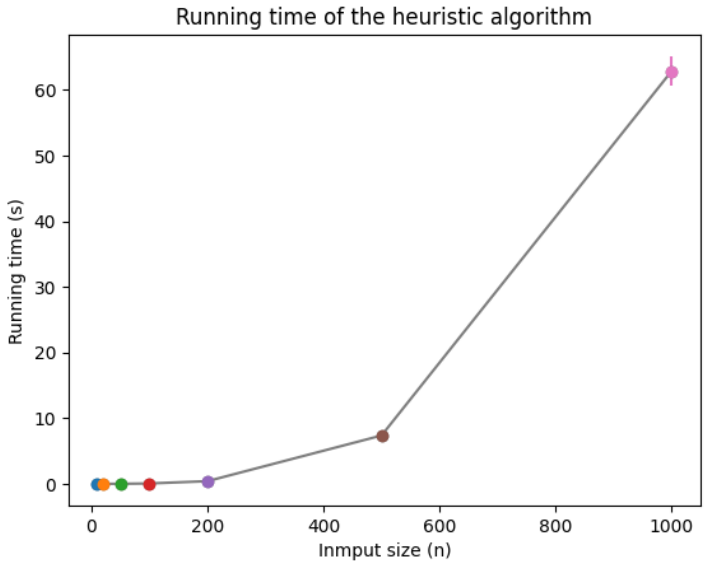
n = 100, sample mean = 0.05453288555145258, sample std = 0.008722429628794716, lower bound = 0.053515216843526954, upper bound = 0.05555055425937821

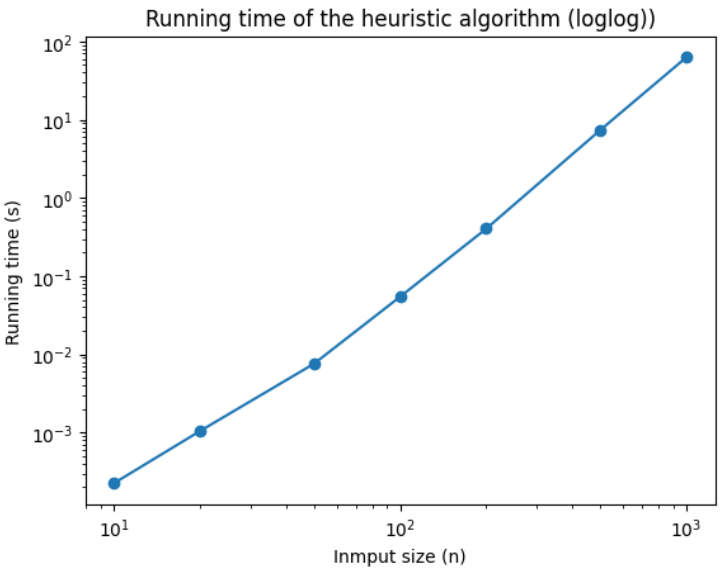
n = 200, sample mean = 0.4046731173992157, sample std = 0.06239476673936145, lower bound = 0.39739335655834307, upper bound = 0.4119528782400883

n = 500, sample mean = 7.361952663660049, sample std = 0.9673213011598399, lower bound = 7.249092754140057, upper bound = 7.474812573180041

n = 1000, sample mean = 62.868611347675326, sample std = 9.627741845994773, lower bound = 61.745317492450624, upper bound = 63.99190520290003

After that, graphs of the average working time depending on the input width has been created.





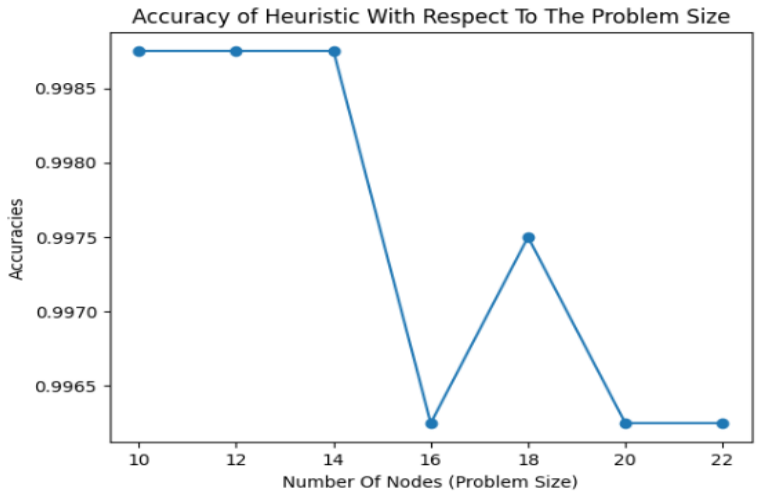
Finally, the slope of the log-log graph is found as 2.7408698055199254.

7. Experimental Analysis of the Quality

On the experimental analysis of the quality of the heuristic algorithm, an experiment having the input sizes n = 10, 12, 14, 16, 18, 20, and 22 have been conducted. For each input size, both the brute force and heuristic algorithms have been run for 200 times. For each run, 4 different k values have been selected (k=0, k= random, k=E/2, k =E where E is the number of edges). Therefore for each input size, 800 results have been collected making 5600 results in total for each of the algorithms. In order to run the algorithms, the "main.py" file has been used and the results of the quality analysis experiments have been saved to the "quality_test.csv" file. The analysis steps made on the data file can be found in the "analysis.ipynb" file.

In order to understand the quality of the heuristic algorithm with respect to the problem size, the accuracies of the results of the heuristic for each input size with respect to the results of the brute force algorithms have been found.

```
n = 10, accuracy = 0.99875
n = 12, accuracy = 0.99875
n = 14, accuracy = 0.99875
n = 16, accuracy = 0.99625
n = 18, accuracy = 0.9975
n = 20, accuracy = 0.99625
n = 22, accuracy = 0.99625
```



By looking at the results of each input size, it can be seen that as the input size increases, the accuracy of the heuristic algorithm tends to decrease.

Looking at the overall accuracy of our heuristic algorithm, even though the accuracy tends to decrease as the input size grows, the overall accuracy for small input sizes is still around 0.9975.

```
n = all samples, accuracy = 0.9975
```

8. Experimental Analysis of the Correctness (Functional Testing)

Python's Pytest library with white box test methods have been used to do functional testing of the code and achieved 100% test coverage. Edge cases such as k=max edge count or no edge have also been implemented on the test codes, and the algorithm implementation was successful. The codes for the experimental analysis of the correctness can be found in the "test.py" file. Applied test functions are as follows:

```
import pytest
import random
import networkx as nx
from heuristic import KernighanLinAlgorithm

random.seed(0)

def test_swap_nodes_1():
    U, W = {1, 2, 3}, {4, 5, 6},
    a, b = 1, 4
    expected_U, expected_W = {2, 3, 4}, {1, 5, 6}

    algo = KernighanLinAlgorithm(None, None)
    U, W = algo.swap_nodes(U, W, a, b)

    assert U == expected_U
    assert W == expected_W


def test_swap_nodes_2():
    U, W = {1, 2}, {4, 5, 6},
    a, b = 2, 5
    expected_U, expected_W = {1, 2}, {4, 5, 6}

    algo = KernighanLinAlgorithm(None, None)
```

```
    U, W = algo.swap_nodes(U, W, a, b)

    assert U == expected_U
    assert W == expected_W


def test_compute_D_values_1():
    G = nx.Graph()
    G.add_edges_from([(1, 2), (1, 3), (1, 4), (2, 3), (3, 4)])
    U, W = {1, 2}, {3, 4}
    expected_D = {1: 1, 2: 0, 3: 1, 4: 0}

    algo = KernighanLinAlgorithm(G, None)
    function_D = algo.compute_D_values(U, W)

    assert function_D == expected_D

def test_compute_D_values_2():
    G = nx.Graph()
    G.add_edges_from([(1, 2), (1, 3), (2, 4), (2, 3), (3, 4)])
    U, W = {1, 4}, {2, 3}
    expected_D = {1: 2, 4: 2, 2: 1, 3: 1}

    algo = KernighanLinAlgorithm(G, None)
    function_D = algo.compute_D_values(U, W)

    assert function_D == expected_D


def test_initial_partition_1():
    G = nx.Graph()
    G.add_edges_from([(1, 2), (1, 3), (1, 4), (2, 3), (3, 4)])
    expected_U, expected_W = {1, 3}, {2, 4}

    algo = KernighanLinAlgorithm(G, None)
    U, W = algo.initial_partition()

    assert U == expected_U
    assert W == expected_W

def test_initial_partition_2():
```

```
    G = nx.Graph()
    G.add_edges_from([(1, 2), (1, 3), (1, 5), (2, 6),
                      (3, 6), (3, 5), (4, 5), (4, 6),
                      ])
    expected_U, expected_W = {1, 2, 4}, {3, 5, 6}

    algo = KernighanLinAlgorithm(G, None)
    U, W = algo.initial_partition()

    assert U == expected_U
    assert W == expected_W

def test_compute_max_gain_1():
    G = nx.Graph()
    G.add_edges_from([(1, 2), (1, 3), (1, 4), (2, 3),
                      (3, 4), (3, 5), (4, 5), (4, 6),
                      ])

    U, W = {1, 2, 3}, {4, 5, 6}
    D = {1: 1, 2: 0, 3: 2, 4: 0, 5: -1, 6: -1}
    expected_a, expected_b, expected_gain = 3, 6, 1

    algo = KernighanLinAlgorithm(G, None)
    a, b, gain = algo.compute_max_gain(U, W, D)

    assert a == expected_a
    assert b == expected_b
    assert gain == expected_gain

def test_compute_max_gain_2():
    G = nx.Graph()
    G.add_edges_from([(1, 2), (1, 3), (1, 5), (2, 6),
                      (3, 6), (3, 5), (4, 5), (4, 6),
                      ])

    U, W = {1, 3, 4}, {2, 5, 6}
    D = {1: 1, 3: 1, 4: 2, 2: 0, 5: 3, 6: 1}
    expected_a, expected_b, expected_gain = 4, 5, 3

    algo = KernighanLinAlgorithm(G, None)
    a, b, gain = algo.compute_max_gain(U, W, D)
```

```
    assert a == expected_a
    assert b == expected_b
    assert gain == expected_gain

def test_calculate_cut_1():
    G = nx.Graph()
    G.add_edges_from([(1, 2), (1, 3), (1, 4), (2, 3),
                      (3, 4), (3, 5), (4, 5), (4, 6),
                      ])

    U, W = {1, 2, 3}, {4, 5, 6}
    expected_cut = 3

    algo = KernighanLinAlgorithm(G, None)
    cut = algo.calculate_cut(U, W)

    assert cut == expected_cut


def test_calculate_cut_2():
    G = nx.Graph()
    G.add_edges_from([(1, 2), (1, 3), (1, 5), (2, 6),
                      (3, 6), (3, 5), (4, 5), (4, 6),
                      ])

    U, W = {1, 3, 4}, {2, 5, 6}
    expected_cut = 6

    algo = KernighanLinAlgorithm(G, None)
    cut = algo.calculate_cut(U, W)

    assert cut == expected_cut

def test_case_1():
    G = nx.Graph()
    G.add_edges_from([(1, 2), (1, 3), (1, 4), (2, 3),
                      (3, 4), (3, 5), (4, 5), (4, 6),
                      ])
    k = 3
```

```
    algo = KernighanLinAlgorithm(G, k)
    result = algo.run()

    assert result == True

def test_case_2():
    G = nx.Graph()
    G.add_edges_from([(1, 2), (1, 3), (1, 5), (2, 6),
                      (3, 6), (3, 5), (4, 5), (4, 6),
                      ])

    k = 2

    algo = KernighanLinAlgorithm(G, k)
    result = algo.run()

    assert result == False

def test_case_3():
    G = nx.Graph()

    for i in range(1, 7):
        G.add_node(i)

    k = 0

    algo = KernighanLinAlgorithm(G, k)
    result = algo.run()

    assert result == True

def test_case_4():
    G = nx.Graph()
    G.add_edges_from([(1, 2), (1, 3), (1, 4), (2, 3),
                      (3, 4), (3, 5), (4, 5), (4, 6),
                      ])
    k = 8

    algo = KernighanLinAlgorithm(G, k)
    result = algo.run()
```

```
    assert result == True

def test_case_5():
    G = nx.Graph()
    G.add_edges_from([(1, 2), (1, 3), (1, 4), (2, 3),
                      (3, 4), (3, 5), (4, 5), (4, 6),
                      ])
    k = 1

    algo = KernighanLinAlgorithm(G, k)
    result = algo.run()

    assert result == False
```

9. Discussion

During the experiments and testing stages, there were not any unexpected defects in the algorithm. Since the implemented algorithm is a heuristic algorithm, it was working very fast compared to the brute force algorithm, but the accuracy tends to decrease as the input size increases. In other words, the accuracy values obtained for different input sizes indicated a slightly lower accuracy as the problem size increased. On the other hand, looking at the overall quality, it had a good accuracy rate of around 0.9975 for the aforementioned inputs.

According to the theoretical analysis in the Section 3.b, the time complexity of the heuristic algorithm was determined to be $O(n^3)$. This means that the algorithm's runtime should increase cubically with the size of the problem (n). The theoretical analysis suggests that the log-log plot of the algorithm's runtime versus the problem size should have a slope of 3. However, the experimental analysis reveals a slightly different picture. The observed slope of 2.7408698055199254 in the log-log plot suggests a slightly smaller increase in the running time as the input size increases, but it is still in the $O(n^3)$.

Based on the experimental results and the information provided above, there appears to be a slight difference between the theoretical time complexity and the observed performance of the heuristic algorithm in the experiments. This difference can be explained by the usage of an optimization algorithm in the design and implementation of the heuristic algorithm. It also shows that the worst-case scenario used in the analysis of the algorithm complexity is not always the case in the experimental solutions of our problem.

REFERENCES

Garey, M. R., Jhonson., D. S., & Stockmayer, L. (2002, March 26). *Some simplified NP-complete graph problems*. Theoretical Computer Science. Retrieved April 27, 2023, from https://reader.elsevier.com/reader/sd/pii/0304397576900591?token=F157EE042185088 2612496107FE65A4510AE69B193A334FC8D8B0107AA43DC1F3371AABEC7F5D2 6BEFFA50A40E28ED31&originRegion=eu-west-1&originCreation=20230427180412

Kernighan, B. W., & Lin, S. (1970). An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, *49*(2), 291–307. https://doi.org/10.1002/j.1538-7305.1970.tb01770.x