CS301

2022-2023 Spring

Project Final Report

Group 132

Murat Demiraslan 27869

Selim Gül 29200

1. Problem Description

Problem: Clique.

Intuitive: Finding k many adjacent nodes in a given undirected graph G, in other words, find a complete sub-graph with k nodes from a given undirected graph G.

Formal: For given undirected graph G (V, E) having n nodes, a positive integer k such that $k \leq n$. Find a k-clique, which for a subset W of nodes V, such that W has size k, and for any u, v in W, E = {u, v} is an edge of G.

Applications:

- Cliques were used firstly by social scientists to model social cliques, groups of people who knows each other.[1]
- It can be used in chemistry to find similar and different formulas from the existing formulas.
- Also, bioinformatics is one of the fields cliques are used.

Theorem: Clique problem is np-complete.

Proof: it is proven by Richard Karp on his well-known paper "Reducibility Among Combinatorial Problems"[2]. Also, it is a clique decision problem, therefore, it is np-complete.

2. Algorithm Description

a. Brute Force Algorithm

A brute force algorithm for the k-clique decision problem may be looking every k-size sub-graph and check whether it is a clique or not, if there is a clique among them, return true, otherwise, return false.

This algorithm's time complexity is factorial since it needs to generate k sized combinations in n sized nodes. $C(n, k) = n! / (k! * (n-k)!)$

[1] *Luce and Perry, 1949, "A Method of Matrix Analysis of Group Structure".*

[2] *Karp, 1972, p. 94, 97, 100. Also, he refers Stephen Cook's (1971) "The Complexity of Theorem-Proving Procedure".*

The Algorithm:

1. Look at all the k-sized sub-graph combinations of G.
2. Check every sub-graph for being clique by looking each node if it has an edge or not, if any node which does not have an edge is detected in the sub graph return false. If there is not any node such that then return true, referring this sub-graph is a clique
3. Once encounter a clique in the sub-graphs, return true. If there is not any clique, return false.

This algorithm is directly an iterative algorithm, and not efficient since it can be used for only small n.

b. Heuristic Algorithm

A heuristic algorithm for the k-clique problem is a greedy algorithm. The algorithm will take the node with highest degree, meaning that the node with the most edges, and check whether there is a clique or not with the given node count, and if there is, return true, or false if not.

The Algorithm:

1. Start with an empty clique and add an arbitrary node from the graph.
2. While the clique has fewer k nodes, select a node that is connected to all current clique members and has the highest degree.
3. Add the selected node to the clique and repeat the process until the clique has size k or there are no more nodes to add.

This algorithm is much faster than the brute force method, with a time complexity of O(n*k), where n is the number of nodes in the graph and k is the size of the combinations.

Pseudo-code:

```
Function greedy_k_clique(graph, k)
    Initialize empty set "clique"

    Set "start" as node with highest degree in "graph"
    Add "start" to "clique"

    While size of "clique" is less than k do
        Initialize "next_node" as None
        Initialize "max_degree" as -1

        For each "node" in "graph" do
            If "node" is not in "clique" and all nodes in "clique" are neighbors of "node" then
                If degree of "node" is greater than "max_degree" then
                    Update "max_degree" as degree of "node"
                    Update "next_node" as "node"

        If "next_node" is None then
            Return None (No k-clique exists)

        Add "next_node" to "clique"

    Return "clique"
EndFunction
```

3. Algorithm Analysis

a. Brute Force Algorithm

Theorem: The brute force algorithm for k-clique detection correctly identifies whether a k-clique exists in an undirected graph G (V, E) with node set V and edge set E.

Proof: It is enough to show that the algorithm considers all possible k-node pairs in the G and correctly decide if it is a clique or not to prove the correctness of the brute force algorithm.

- The algorithm iterates for all possible combination of k nodes from the set V. It can be said that all the possible combination is considered if they are clique or not.
- Checking every k node combination if they are clique or not uses the nodes of the subset and the E to determine all nodes are connected to each other or not. Therefore, for any pair (u, v) nodes the edge {u, v} must be in E. If there is an edge for all pairs in the subset it will return true for being a clique.

Because of these reasons, if there is a k-clique in the G, the brute force algorithm will exactly determine it, and if there is not any k-clique, after checking all possible k-nodes subgraph it will return false.

Worst-case time complexity:

- The worst case can be achieved when there is not any k-clique because it will check all possible k-nodes sub graphs. Also, when checking the k-node sub-graphs, it will be determined that it is not a clique by checking all possible pairs.
- Checking all possible n-nodes subgraphs ➔ C (n, k) = n! / (k! * (n-k)!)
- Deciding it is not a clique in the worst case ➔ O(k^2) since there are k*(k-1)/2 pairs.
- **ϴ** (n! / (k! * (n-k)!) * k^2) = **ϴ** (n!)

b. Heuristic Algorithm

Theorem: The greedy k-clique detection algorithm identifies a subset of k nodes that form a clique, if such a subset exists, in a given undirected graph G(V, E) with node set V and edge set E. The greedy algorithm is a heuristic approach that does not guarantee finding a k-clique if one exists, because it makes locally optimal choices without considering the overall structure of the graph.

Proof: To prove the correctness of this algorithm, we need to prove two properties:

1. The algorithm terminates: The main loop of the algorithm continues until the size of the "clique" set is equal to k, or no more nodes can be added to the "clique" set. Since we are only considering finite graphs, the number of nodes is finite and the size of the "clique" set is always strictly increasing. Thus, the loop will either reach the target size k or exhaust all possibilities, proving the algorithm will always terminate.
2. Correctness: Whenever a node is added to the "clique", the algorithm checks that the node is connected to every other node already in the "clique". This means that the "clique" set always represents a clique in the graph. Furthermore, the algorithm only selects the next node that is connected to all nodes currently in the clique and has the highest degree among such nodes. Therefore, it is seeking to add nodes that would contribute to forming a k-clique. If the algorithm is unable to find a next node that connects with all the current nodes in the clique, it returns None, correctly indicating that no k-clique exists.

Worst-case Time Complexity:

- The primary operations of our greedy algorithm involve iterating over the nodes in the graph, and for each node, checking its connections to nodes in the current clique.
- Selecting a node with maximum degree to start with, takes O(n) time, where n is the number of nodes in the graph.
- Inside the while loop, the algorithm could potentially go through all the nodes in the graph for each node already in the clique, resulting in a complexity of O(n*k), where k is the size of the desired clique.
- In the worst-case scenario, each node is connected to all others, so checking the connections takes O(n) time.

- Combining all those operations, the overall worst-case time complexity of the algorithm is O(n^2*k).

4. Sample Generation (Random Instance Generator)

a. We used **networkx** library in the python to generate random graph to test the brute force algorithm.

b. Giving the parameters n (number of nodes), and the parameter p (probability that an edge is constructed between nodes)

```
def generate_random_graph(n, p):

    G = nx.gnp_random_graph(n, p)
    graph = {node: set(neighbors) for node, neighbors in G.adj.items()}
    return graph

      n = 10
      p = 0.5
      random_graph = generate_random_graph(n, p)
      print(random_graph)
```

5. Algorithm Implementations

a. Brute Force Algorithm

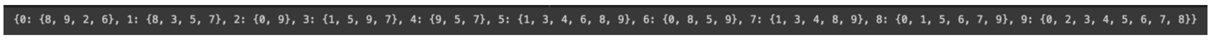
In order to iterate through all the nodes, we used itertools library's combinations method in python.

```
import itertools

def is_clique(nodes, graph):
    for u, v in itertools.combinations(nodes, 2):
        if v not in graph[u]:
            return False
    return True

def contains_k_clique_bruteforce(graph, k):
    nodes = graph.keys()
    for combination in itertools.combinations(nodes, k):
        if is_clique(combination, graph):
            return Tru
    return False
```

We found the itertools idea and its implementation from the internet.

Initial Tests:

1- n = 10, p = 0.5, k = 3, result = True (there is a 3-clique in this graph), the sample graph;



```
{0: {8, 9, 2, 6}, 1: {8, 3, 5, 7}, 2: {0, 9}, 3: {1, 5, 9, 7}, 4: {9, 5, 7}, 5: {1, 3, 4, 6, 8, 9}, 6: {0, 8, 5, 9}, 7: {1, 3, 4, 8, 9}, 8: {0, 1, 5, 6, 7, 9}, 9: {0, 2, 3, 4, 5, 6, 7, 8}}
```

2- The same graph, k = 4, result = True

3- The same graph, k = 5, result = False

4- n = 15, p = 0.5, k = 5, result = True

5- Last graph, k = 6, result = False

6- **n = 30**, **p = 0.5**, **k = 10**, result = False, time = 14s

7- Last graph, k = 12, result = False, time = 44s (**NOT EFFICIENT**)

8- Last graph, k = 6, result = True, time = 0.009s

9- **n = 30**, **p = 0.7 (Only changed the p)**, **k = 10**, result = True, time = 2s (**previous case is the worst case**)

10- Last graph, k = 20, result = False

11- n = 100, p = 0.8, k = 20, **result =???,** time = 4m 14s, we cancelled it at this time, The time needed increases so fast.

12- n = 50, p = 0.8, k = 10, result = True

13- Last graph, k = 5, result = True

14- n = 200, p = 0.5, k = 5, result = True

15- Last graph, k = 8, result = True

b. Heuristic Algorithm

For the implementation of the greedy algorithm, we did not use any external libraries.

```
def greedy_k_clique(graph, k):
    clique = set()

    start = max(graph, key=lambda node: len(graph[node]))
    clique.add(start)
```

```
while len(clique) < k:
    next_node = None
    max_degree = -1
    for node in graph:
        if node not in clique and all(neighbor in graph[node] for neighbor in clique):
            if len(graph[node]) > max_degree:
                max_degree = len(graph[node])
                next_node = node

    if next_node is None:
        return None

    clique.add(next_node)

return clique
```

Tests:

1- n = 10, p = 0.5, k = 3, result = True, the sample graph:

```
{0: {2, 3, 5, 7, 9}, 1: {7}, 2: {0, 9, 4, 5}, 3: {0, 5, 6, 7}, 4: {9, 2, 5, 6}, 5: {0, 2, 3, 4, 8}, 6: {3, 4, 7}, 7: {0, 1, 3,
6, 8, 9}, 8: {9, 5, 7}, 9: {0, 2, 4, 7, 8}}
```

2- The same graph, k = 4, result = False

3- The same graph, k = 5, result = False

4- n = 15, p = 0.5, k = 5, result = False

5- Last graph, k = 6, result = False

6- **n = 30**, **p = 0.5**, **k = 10**, result = False

7- Last graph, k = 12, result = False

8- Last graph, k = 6, result = False

9- **N = 30**, **p = 0.7 (Only changed the p)**, **k = 10**, result = True, time = 0.000216 seconds

10- Last graph, k = 20, result = False

11- n = 100, p = 0.8, k = 20, result = False, time = 0.0014 seconds

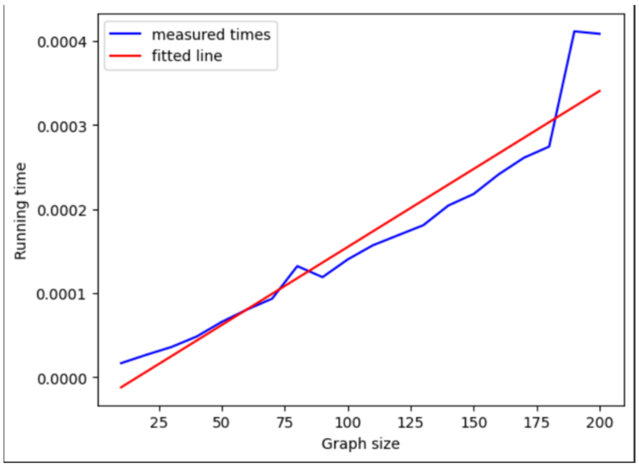
12- n = 50, p = 0.8, k = 10, result = True

13- Last graph, k = 5, result = True

14- n = 200, p = 0.5, k = 5, result = True

15- Last graph, k = 8, result = True

6. Experimental Analysis of The Performance (Performance Testing)



100 runs per each graph sizes (10, 20, 30, 40, …., 200) are executed to test the performance of the algorithm. As a result, we got the statistics for each graph sizes as follows:

Size	Mean Time	Std Dev	Std Errors	%95 CL
10	2.074e-05	3.73e-06	8.34e-07	(1.91e-05, 2.23e-05)
20	2.75e-05	7.26e-06	1.62e-06	(2.43e-05, 3.07e-05)
30	3.75e-05	4.27e-06	9.56e-07	(3.56e-05, 3.94e-05)
50	6.40e-05	7.88e-06	1.76e-06	(6.06e-05, 6.75e-05)
100	1.58e-04	4.87e-05	1.08e-05	(1.37e-04, 1.8e-04)
150	2.25e-04	3.95e-05	8.84e-06	(2.07e-04, 2.42e-04)
200	4.08e-04	1.30e-04	2.91e-05	(3.50e-04, 4.65e-04)

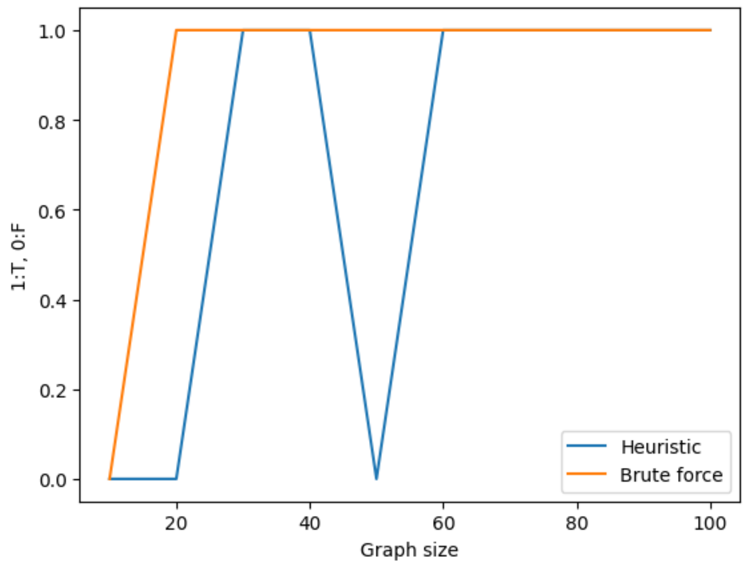
7. Experimental Analysis of the Quality

Since we have used a greedy algorithm, while it is significantly fast in execution in comparison to the Brute-force algorithm, due to how it is designed, it may not always find the exact solutions to the problem. To test how the result from the heuristic algorithm may deviate from the actual result, we have implemented the following:

1. Create an array with the desired sizes for the array. In our implementation, we used numbers from 10 to 101 with a step of 10.
2. Generate a random graph using the sample generator in Section 4 with the resulting size and p = 0.5

3. Try to find clique with 6 nodes using both the brute-force and the greedy algorithm
4. Add the results to separate arrays
5. Plot the results

While the variables such as p or k can change, the results will be accurate. In one of our runs, we have reached the following plot with False represented with 0.0 and True represented with 1.0:



As it can be seen at the plot above, while the findings of both Heuristic and Brute-force algorithms are mostly consistent, in some cases even though there exists a clique, the Heuristic algorithm could not find it. This is because of how our algorithm works. It always checks if there is a clique with given k starting with the node with a maximum degree, however there may also exists a clique that doesn't include the said node. In those cases, the Heuristic algorithm cannot find the exact result.

8. Experimental Analysis of the Correctness (Functional Testing)

For the testing part, we have used **unittest** library. We used the White Box testing method as we knew what the edge cases are and what the possible outcomes are. Our test cases are the following:

a. Test with a simple graph with 3-clique.

```
graph = {1: {2, 3}, 2: {1, 3}, 3: {1, 2}, 4: {5}, 5: {4}}
```

b. Test with an empty graph.

```
graph = {1: set(), 2: set(), 3: set()}
```

c. Test with a graph with isolated nodes.

```
graph = {1: set(), 2: set(), 3: set()}
```

d. Test with a larger graph that has 5-clique.

```
graph = {1: {2, 3, 4, 5}, 2: {1, 3, 4, 5}, 3: {1, 2, 4, 5}, 4: {1, 2, 3, 5}, 5: {1, 2, 3, 4}}
```

e. Test with a graph that has two 3-cliques.

```
graph = {1: {2, 3}, 2: {1, 3}, 3: {1, 2}, 4: {5, 6}, 5: {4, 6}, 6: {4, 5}}
```

9. Discussion

The results of the experimental analysis align quite well with the theoretical expectations for both the brute-force and the greedy heuristic algorithms. The detailed discussions on three aspects of the analysis are the following:

a. Performance Analysis: As expected, the heuristic algorithm performs significantly better than the brute-force algorithm as the number of nodes (n) and clique size (k) increases. This is because the time complexity of the brute-force algorithm grows in factorial with the input size, as it needs to check every possible combination of k nodes in the graph. On the other hand, the heuristic algorithm only checks a subset of these combinations, leading to a much better performance especially for larger inputs. This matches with the theoretical time complexity analysis where the heuristic approach has a polynomial time complexity (O(n^2 * k^2)), which would be significantly more efficient than an factorial time complexity of a brute-force approach.

b. Quality Analysis: The brute-force algorithm guarantees to find a clique if one exists because it checks all possible combinations. However, the heuristic algorithm might miss some cliques because it always starts with the node of highest degree. There may exist cliques that don't include this highest-degree node, or there could be multiple k-cliques, in which case the heuristic algorithm will find one but not necessarily all. This is a trade-off made by the heuristic algorithm to gain speed in comparison to the brute-force approach.

c. Correctness Analysis: Given that both algorithms produce expected results in all tested cases, this indicates that they are implemented correctly. The algorithms' correctness is not dependent on the size or complexity of the graph, but on the logic of the algorithm. Therefore, as long as the input graph is structured correctly and the algorithms are implemented correctly, they will produce expected results for cliques of size k in any graph. As already discussed in Quality Analysis, the heuristic algorithm doesn’t always give the exact result, however this does not mean that it is implemented incorrectly.