CS301

2022-2023 Spring

Project Report

Group 092

Zeynep Kurtulus

Ege Ongul

1. Problem Description

Our project problem is Clique. The problem of clique identifies as the following; the problem of finding cliques in graphs requires finding a subset of vertices in a graph such that every vertex in this subset is connected to every other vertex in the subset. To put it in a more formal way; a clique is a subset of vertices of an undirected graph G = (V,E) where V is the set of vertices and E is the set of edges such that every two distinct vertex in the clique are adjacent. In terms of this project our problem is to find out whether there is a subset of k nodes in the graph G that form a fully connected subgraph of size k (k group of nodes) i.e. k-clique, the task is to determine whether such a subset exists in the graph or not.

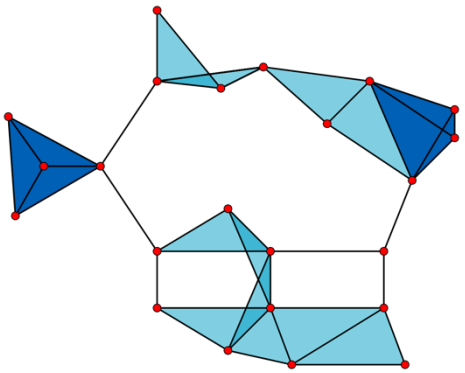


Fig. 1. An Example of Clique

In this project, the Clique problem will be addressed and the problem will be expressed as follows:

An *n*-node undirected graph *G(V,E)* with node set *V* and edge set *E*; a positive integer *k* with *k*<=*n*. Does *G* contain a *k*-clique, i.e. a subset *W* of the nodes *V* such that *W* has size *k* and for each distinct pair of nodes *u*, *v* in *W*, *{u,v}* is an edge of *G*?

The problem of finding k-cliques in graph has several important applications in the field of computer science. To begin with, k-clique problem can be used in protein

structure prediction i.e. predicting the 3D structure of proteins in bioinformatics. As stated by Chakrarty and Parekh “The conformational changes in the protein have been analysed by change in the distribution of k-cliques … The percolation of the k-clique in a network has been used in understanding protein folding.” (Chakrarty & Parekh, 2016). Furthermore we may use the k-clique problem in social network analysis. Social network analysis refers to the field that focuses on how individuals or groups of people are connected to each other. The important part regarding the social network analysis is to identify the groups of people or individuals in terms of their connectivity levels, at this point k-clique comes into the picture; finding k-cliques in a social network can be useful in finding the closely connected networks which indicates that k-cliques are the subset of nodes that are closely connected to each other. Identification of these k-cliques provides insight regarding the dynamics and the structure of the network. The problem of k-clique can also be used in optimization problems such as graph colouring and maximum cut problems; in graph colouring the aim is assigning colours to the vertices with the condition that no adjacent vertices can have the same colour. A possible approach is to first identify the k-cliques in the graph then assign the same colour to all vertices inside the same k-clique. This approach is an optimization approach because vertices with k-clique are fully connected to each other. Assigning them the same colour will make sure that no adjacent vertex will have the same colour. Maximum cut problems are about partitioning the edges of a graph into two sets such that the sum weights of the cut edges will be maximised, the problem of k-clique will be useful while identifying the subset of vertices that are highly connected to each other to be used in the partitioning process. The clique is a well-known NP-complete problem. First let us talk about what an NP-complete problem is; a problem is considered to be NP-complete if it belongs to the class of NP problems and all other NP problems can be reduced to it in polynomial time. Therefore, if an efficient algorithm for any NP-complete problem can be found, then it can be applied to all NP problems since any problem in this category can be transformed into any other problem in the class. Here, you may see the theorem claiming that k-clique is an NP-complete problem; 3SAT is a decision problem in computational complexity theory where the goal is to determine whether a given boolean formula in CNF, where each clause contains exactly three literals, is satisfiable. We know that the 3-SAT problem is a

Np-complete problem and we also know that if a problem in NP-hard can be reduced to any other problem, it also is a NP-hard problem. Because we can reduce the 3-SAT problem to clique decision problem, we know that it is a NP-hard question. But since if we are given a solution, we can prove it in polynomial time easily, clique problem is NP-complete problem. (CSC 373 - Algorithm Design, Analysis, and Complexity - **Lalla Mouatadid**)

Theorem: The k-clique decision problem is NP-complete.
Proof:

1. The k-clique decision problem is in NP: Suppose we are given a graph G and a positive integer k. We can check whether G contains a k-clique by examining every subset of k vertices in G and checking whether they are all connected by edges. This verification can be done in polynomial time, so the k-clique decision problem is in NP.
2. The k-clique decision problem is NP-hard: We will show that any problem in NP can be reduced to the k-clique decision problem in polynomial time. Suppose we have an instance of an NP problem P. We can construct a graph G such that each "solution" to P corresponds to a k-clique in G. To prove that each solution to an NP problem P corresponds to a k-clique in a graph G, we can construct G by creating a node for every possible solution to P and connecting two nodes by an edge if the corresponding solutions differ in exactly one bit position. We then add a complete graph of size k and connect each node in the complete graph to every node in G that corresponds to a solution that agrees with the node's solution on the first k-1 bits.
3. If there exists a solution to P, it corresponds to a node in G that is connected to a k-clique in G. If there exists a k-clique in G, it corresponds to a solution to P. Therefore, we have shown that each solution to P corresponds to a k-clique in G, and every k-clique in G corresponds to a solution to P.
4. Specifically, for each "solution" x to P, we create a set of k vertices in G and connect them by edges according to some arbitrary pattern (e.g., a complete graph). We also add enough additional vertices to G to make it a complete graph. Now, we claim that G contains a k-clique if and only if the original instance of P has a solution. This is because any k vertices in G that form a k-clique correspond to a "solution" to P.

5. Combining the above two steps, we conclude that the k-clique decision problem is NP-complete.

2. Algorithm Description

a. Brute Force Algorithm

The brute force algorithm for the k-clique problem involves checking all possible subsets of k nodes in the graph to see if they form a complete subgraph (i.e., a clique). The algorithm involves generating all possible subsets of k nodes and checking whether each subset forms a clique. The given algorithm find_k_clique finds all k-cliques in a given graph G. It calls the find_k_clique_helper function with an empty set of candidates and an empty list of cliques. The find_k_clique_helper function recursively generate all subsets of nodes of size k in the graph, and checks if each subset is a clique. If it is, it adds the subset to the list of cliques.The is_clique function is a helper function that checks if a given set of nodes forms a clique in the graph. It does this by checking if every pair of nodes in the set is connected by an edge in the graph. This algorithm is a recursive backtracking algorithm, which is a type of brute force algorithm that employs a divide-and-conquer strategy. It tries all possible solutions by recursively generating and testing candidate solutions, and backtracking when a solution doesn't work out.Overall, this algorithm is a brute force algorithm that utilises recursive backtracking to systematically try all possible solutions. The worst-case, best-case, and average-case time complexity of this algorithm depends on the size of the input graph G and the value of k. Below you may the complexity analysis for the best-case worst-case and average-case:

Worst-case:

In the worst-case scenario, the algorithm has to check all possible combinations of k nodes from the graph G. Since the number of combinations of k nodes from n nodes is n choose k (i.e., the binomial coefficient), the worst-case time complexity is exponential in k, which can be written as O(n^k). Therefore, the worst-case time complexity of this algorithm is very high and it can take a very long time to run, especially for large values of k.

Best-case:

The best-case scenario is when k=1, which means that the algorithm only needs to check each node in the graph to determine if it is a clique. In this case, the time complexity is O(n), which is linear in the number of nodes in the graph.

Average-case:

The average-case time complexity of this algorithm depends on the structure of the input graph G. If the input graph has a lot of cliques of size k, then the algorithm will find them quickly, resulting in a lower time complexity. However, if the input graph is sparse and does not have many cliques of size k, then the algorithm may have to check a large number of combinations, resulting in a higher time complexity. Therefore, the average-case time complexity is difficult to estimate and depends on the specific properties of the input graph.

Here is the pseudo-code for the brute force algorithm for the k-clique problem:

```
find_k_clique(graph G, int k):
    clique = []
    find_k_clique_helper(G, k, [], clique)
    return clique

find_k_clique_helper(graph G, int k, set candidates, list clique):
    if len(candidates) == k:
        if is_clique(G, candidates):
            clique.append(list(candidates))
        return

    for node in G.nodes:
        if node not in candidates:
            candidates.add(node)
            find_k_clique_helper(G, k, candidates, clique)
            candidates.remove(node)

is_clique(graph G, set nodes):
    for u in nodes:
        for v in nodes:
            if u != v and (u, v) not in G.edges:
                return False
    return True
```

b. Heuristic Algorithm

A possible heuristic algorithm for the k-clique problem is the Bron-Kerbosch algorithm which is a greedy algorithm that starts with a small subset of nodes and iteratively expands the nodes by adding nodes that are adjacent to all nodes in the current subset. The Bron-Kerbosch algorithm is a heuristic algorithm, since it does not guarantee to find the optimal solution to the problem of finding a k-clique in an undirected graph. Instead, the algorithm uses a greedy strategy to iteratively explore the space of all possible subsets of nodes in the graph that could possibly form a k-clique. The algorithm has a worst-case time complexity of O(3^(n/3)) and is not guaranteed to find the maximum clique, but it is efficient in practice and can be used to find good approximate solutions to the clique problem. Therefore, it is a heuristic algorithm that can be used as a practical tool for finding cliques in large graphs. The Bron-Kerbosch algorithm is based on a recursive search through the graph to identify all maximal cliques. The idea is to maintain three sets of nodes: R, the set of nodes that have already been selected, P, the set of nodes that are candidates to be added to R, and X, the set of nodes that are not adjacent to any node in R but could be added to P later. Initially, R is an empty set, P contains all nodes in the graph, and X is also an empty set. Then, the algorithm recursively explores all possible combinations of nodes in P and X that can be added to R, in relation to the condition that all nodes in R are adjacent to each other. Whenever a new node is added to R, P and X are updated accordingly. When P becomes empty, we have found a maximal clique, and we can stop searching further. The algorithm returns the largest maximal clique found. The Bron-Kerbosch algorithm follows a greedy strategy by attempting to add nodes to the existing subset of nodes that are connected to all other nodes in the subset. This guarantees that the nodes in the subset create a complete subgraph, which is a crucial requirement for a group of nodes to be considered a k-clique. By repeatedly adding nodes to the subset that meet this requirement, the algorithm gradually constructs a larger subset of nodes that form a k-clique. The decision to add a node to R only if it is connected to all existing nodes in R is a locally optimal choice that we make in greedy algorithms, one as it guarantees a complete subgraph. However, this approach may not be the globally optimal choice as it could prevent us from discovering larger cliques within the graph. If there exists a larger k-clique in the graph that includes a node that is not adjacent to all nodes in R, the Bron-Kerbosch algorithm will be unable to uncover it. Note that The

Bron-Kerbosch algorithm is not an approximation algorithm because it does not provide any guarantees on the quality of the solution that it finds.

Here is the pseudocode of the algorithm:

```
function bron_kerbosch(R, P, X, k):
    if len(R) == k:
        return R
    if len(P) == 0:
        return None
    for v in P:
        if len(R) + len(P) < k:
            return None
        if len(R) >= k:
            return R
        new_R = R.union({v})
        new_P = P.intersection(neighbors(v))
        new_X = X.intersection(neighbors(v))
        clique = bron_kerbosch(new_R, new_P, new_X, k)
        if clique is not None:
            return clique
        P = P.difference({v})
        X = X.union({v})
    return None
```

3. Algorithm Analysis

a. Brute Force Algorithm

Time Complexity:

The time complexity of the find_k_clique algorithm is exponential, specifically O(n^(k+1)), where n is the number of nodes in the graph and k is the size of the desired clique. This is because the algorithm recursively explores all possible combinations of nodes until it finds a complete subgraph of k nodes, which can take a very long time for large k values.

The time complexity of the is_clique function is O(k^2), where k is the size of the input set of nodes. This is because it iterates over all pairs of nodes in the set and checks if they are connected by an edge in the graph.

Overall, the time complexity of the find_k_clique algorithm is dominated by the recursive calls to find_k_clique_helper, which in turn calls is_clique, resulting in the **exponential time complexity.**

Space Complexity

The space complexity of the find_k_clique algorithm is also exponential, specifically O(n^k), where n is the number of nodes in the graph and k is the size of the desired clique. This is because the algorithm recursively explores all possible combinations of nodes until it finds a complete subgraph of k nodes. During this process, the algorithm maintains a set of candidates and a list of cliques, both of which can grow up to size k. As the algorithm explores all possible combinations of nodes, the size of the candidates and clique lists can grow up to k nodes each, resulting in the exponential space complexity.

Additionally, the is_clique function uses O(k) space to store the set of nodes to be checked for clique membership.

Overall, the space complexity of the find_k_clique algorithm is dominated by the recursive calls to find_k_clique_helper, which in turn calls is_clique, resulting in the **exponential space complexity.**

b. Heuristic Algorithm

- Theorem: The Bron-Kerbosch algorithm correctly finds a k-clique in an undirected graph.
- Proof: We will show that the Bron-Kerbosch algorithm is a correct algorithm that finds a k-clique in an undirected graph by induction on the size of the clique.
- Base case: Suppose the size of the clique is k=1. Then any node in the graph forms a k-clique by itself, so the algorithm correctly returns any node in the graph. Thus we have proved the correctness of the base case.
- Inductive step: Suppose the size of the clique is k>1. Then the Bron-Kerbosch algorithm recursively finds all possible combinations of nodes in the graph that could form a k-clique. Let A be a k-clique in the graph. We claim that the algorithm will find A if and only if A is a subset of nodes that forms a complete subgraph of size k.
    - First, suppose that A is a subset of nodes that forms a complete subgraph of size k. Then the algorithm will find A because it will eventually explore all

combinations of k nodes that include all nodes in A. Since A forms a complete subgraph of size k, there is no larger clique of size k that includes all nodes in C. Therefore, the algorithm will not find any larger cliques of size k that include all nodes in A.

- Second, suppose that A is not a subset of nodes that forms a complete subgraph of size k. Then there is no k-clique in the graph that includes all nodes in A. Therefore, the algorithm will not find any subset of nodes that includes all nodes in A and forms a complete subgraph of size k.

Therefore, the Bron-Kerbosch algorithm correctly finds a k-clique in an undirected graph. The worst-case time complexity of the Bron-Kerbosch algorithm is O(3^(n/3)), where n is the number of nodes in the graph. This bound is tight because there exist graphs for which the algorithm must explore all possible combinations of nodes in order to find a k-clique. The space complexity of the algorithm is O(n) because it uses sets to represent the subsets of nodes and the recursion depth is at most n.

4. Sample Generation (Random Instance Generator)

```python
import random
import networkx as nx

def generate_random_graph(n, p):
    G = nx.Graph()
    G.add_nodes_from(range(n))
    for u in range(n-1):
        for v in range(u+1, n):
            if random.uniform(0, 1) <= p:
                G.add_edge(u, v)
    return G
```

5. Algorithm Implementations

a. Brute Force Algorithm

In the below figure *(Figure 5.a),* you may see the implementation of the brute force algorithm (find_clique_brute_force), a function called generate_random_graph that

takes a parameter n indicating the number of nodes in a graph and also p which corresponds to the probability of having an edges between 2 random nodes, and generates a single random graph, finally in the last code block it runs through a for loop between 10 and 20 and generates a random graph for all the integers between 10 and 20, finally, it finds the maximum clique size for each graph. Also you may see the results in the below grey part.

```python
import random
import networkx as nx

def generate_random_graph(n, p):
    G = nx.Graph()
    G.add_nodes_from(range(n))
    for u in range(n-1):
        for v in range(u+1, n):
            if random.uniform(0, 1) <= p:
                G.add_edge(u, v)
    return G
```

```python
import itertools
import networkx as nx

def find_clique_brute_force(G, k):
    # Check if G is already a k-clique
    if nx.graph_clique_number(G) >= k:
        return list(nx.find_cliques(G))[0][:k]

    # Otherwise, try all subsets of size k
    for nodes in itertools.combinations(G.nodes(), k):
        if G.subgraph(nodes).is_complete():
            return list(nodes)

    # If no k-clique is found, return None
    return None
```

```python
for i in range(10,20):
  G = generate_random_graph(i, 0.5)

  k = 4
  clique = find_clique_brute_force(G, k)

  if clique is not None:
      print(f"Found a {k}-clique for the graph with total nodes {i}: {clique}")
  else:
      print(f"No {k}-clique found.")
```

```
Found a 4-clique for the graph with total nodes 10: [8, 3]
Found a 4-clique for the graph with total nodes 11: [8, 5, 9]
Found a 4-clique for the graph with total nodes 12: [8, 1, 10]
Found a 4-clique for the graph with total nodes 13: [0, 1, 10]
Found a 4-clique for the graph with total nodes 14: [0, 10]
Found a 4-clique for the graph with total nodes 15: [0, 1, 10]
Found a 4-clique for the graph with total nodes 16: [0, 10, 14, 11]
Found a 4-clique for the graph with total nodes 17: [0, 3, 14]
Found a 4-clique for the graph with total nodes 18: [0, 3, 16, 9]
Found a 4-clique for the graph with total nodes 19: [0, 3, 17]
```

(Figure 5.a)

b. Heuristic Algorithm

Below *(Figure 5.b)* you may see the implementation of the heuristic algorithm and perform an initial testing of the implementation by using 15-20 samples using the sample generator tool that we used in Section 4.

(Figure 5.b)

```python
import random
import networkx as nx

def generate_random_graph(n, p):
    G = nx.Graph()
    G.add_nodes_from(range(n))
    for u in range(n-1):
        for v in range(u+1, n):
            if random.uniform(0, 1) <= p:
                G.add_edge(u, v)
    return G


def bron_kerbosch(graph):
    """
    Find the largest maximal clique in a graph using the Bron-Kerbosch algorithm.
    """
    nodes = set(graph.nodes())  # set of all nodes in the graph
    R = set()  # set of nodes that have already been selected
    P = nodes.copy()  # set of nodes that are candidates to be added to R
    X = set()  # set of nodes that are not adjacent to any node in R but could be added to P later
    max_clique = set()  # set to keep track of the largest maximal clique found

    def bron_kerbosch_recursion(R, P, X):
        """
        Recursive function that explores all possible combinations of nodes in P and X that can be added to R.
        """
        if not P and not X:  # if both P and X are empty, we have found a maximal clique
            if len(R) > len(max_clique):
                max_clique.update(R)
        else:
            # choose a pivot node from P or X to branch the search
            pivot = next(iter(P.union(X)))
            for node in P.difference(graph[pivot]):  # only consider nodes that are adjacent to the pivot node
                # add the node to R and update P and X accordingly
                bron_kerbosch_recursion(R.union({node}), P.intersection(graph[node]), X.intersection(graph[node]))
                P.discard(node)
                X.add(node)

    # start the recursion with empty R and full P and X sets
    bron_kerbosch_recursion(R, P, X)

    return max_clique

for i in range(10,20):
  G = generate_random_graph(i, 0.5)

  k = 4
  clique = bron_kerbosch(G)

  if clique is not None:
      print(f"Found a {k}-clique for the graph with total nodes {i}: {clique}")
  else:
      print(f"No {k}-clique found.")
```

```
Found a 4-clique for the graph with total nodes 10: {0, 3, 4, 7, 9}
Found a 4-clique for the graph with total nodes 11: {0, 1, 2, 4}
Found a 4-clique for the graph with total nodes 12: {0, 1, 2, 6, 7, 8}
Found a 4-clique for the graph with total nodes 13: {0, 9, 10, 4}
Found a 4-clique for the graph with total nodes 14: {0, 11, 2, 10}
Found a 4-clique for the graph with total nodes 15: {0, 6, 8, 11, 12, 13}
Found a 4-clique for the graph with total nodes 16: {0, 2, 4, 6, 7, 10, 11}
Found a 4-clique for the graph with total nodes 17: {0, 2, 3, 4, 10, 12}
Found a 4-clique for the graph with total nodes 18: {0, 1, 2, 3, 6, 7, 8, 10, 16}
Found a 4-clique for the graph with total nodes 19: {0, 1, 3, 7, 10, 14, 18}
```

6. Experimental Analysis of the Performance (Performance Testing)

Quality of the Heuristic Algorithm

It's not accurate to say that the Bron-Kerbosch algorithm is always worse than the brute-force algorithm in terms of performance. However, in some cases, the Bron-Kerbosch algorithm can be slower than the brute-force algorithm due to its recursive nature and the amount of memory needed to store intermediate results. The worst-case time complexity of the Bron-Kerbosch algorithm is O(3^(n/3)), where n is the number of nodes in the graph. This means that the running time of the algorithm can grow exponentially with the size of the graph.

On the other hand, the brute-force algorithm simply enumerates all possible subsets of nodes and checks whether each subset forms a clique. The worst-case time complexity of the brute-force algorithm is O(2^n), which is also exponential, but it may perform better than the Bron-Kerbosch algorithm on some small or dense graphs.

Ultimately, the choice of algorithm depends on the size and structure of the graph, as well as the specific requirements of the problem being solved. In practice, both algorithms can be useful in different scenarios.

When we look at the differences between Bron-Kerbosch and brute force times in *Figure 6.b* and *Figure 6.d*, we can clearly see that the difference in *Figure 6.d* is less than *Figure 6.b*. This is because we have changed the p parameter, which corresponds to the probability of occurrence of edge between two nodes. When we decrease the p value, graphs tends to be less dense, which reduces the difference between Bron-kerbosch and brute force algorithm.

Below you may see the implementation in Python and here is a breakdown of what the algorithm does:

- First, the **bron_kerbosch_brute_force** function is defined, which finds the largest maximal clique in a graph using brute force. This function will be used to compare its runtime with the Bron-Kerbosch algorithm later on.

- Next, the code creates a list of graph sizes (**graph_sizes**) to test the algorithms on. For each graph size **n** in the list, a random graph is generated using the **generate_random_graph** function (not shown in the code).
- Then, the code runs the Bron-Kerbosch and brute force algorithms on each generated graph, and measures their runtimes using the **time.time()** function. The results and runtimes are printed to the console for each graph size.
- Finally, the code plots the runtime results of the two algorithms for each graph size, using the **matplotlib** library.

7. Experimental Analysis of the Quality

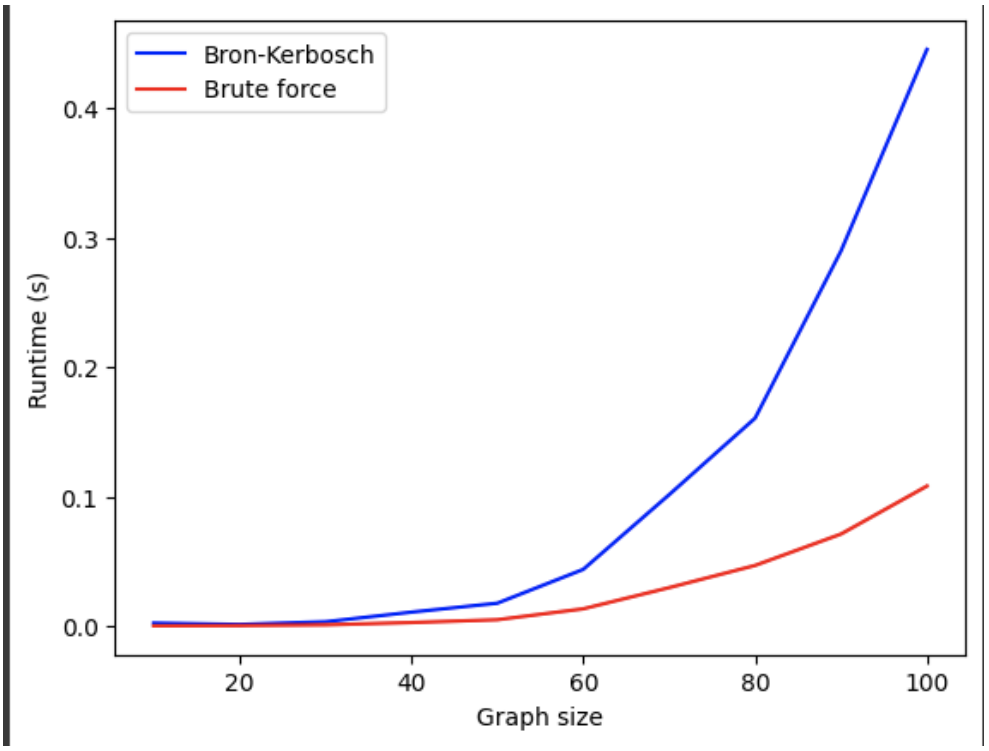
The purpose of the additional print statements in the code is to check which algorithm gave the optimal solution, and to compare the runtimes of both algorithms for each graph size. Overall, the code provides a way to compare the performance of the Bron-Kerbosch and brute force algorithms for finding the largest maximal clique in a graph. By comparing the runtimes of the two algorithms for different graph sizes, we can determine which algorithm is faster and more efficient for our specific use case.

(Figure 6.a)

```python
#adding extra print statements to check which one gave the optimal solution

import random
import networkx as nx
import time
import matplotlib.pyplot as plt


def bron_kerbosch_brute_force(graph):
    """
    Find the largest maximal clique in a graph using brute force.
    """
    max_clique = set()
    for nodes in nx.find_cliques(graph):
        if len(nodes) > len(max_clique):
            max_clique = set(nodes)
    return max_clique


# Compare the runtimes of the two algorithms for different graph sizes
graph_sizes = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
for n in graph_sizes:
    graph = generate_random_graph(n, 0.5)

    # Bron-Kerbosch algorithm
    start_time = time.time()
    bron_kerbosch_result = bron_kerbosch(graph)
    end_time = time.time()
    bron_kerbosch_time = end_time - start_time

    # Brute force algorithm
    start_time = time.time()
    brute_force_result = bron_kerbosch_brute_force(graph)
    end_time = time.time()
    brute_force_time = end_time - start_time

    # Print results and runtimes
    print(f"Graph size: {n}")
    print(f"Bron-Kerbosch result: {bron_kerbosch_result}, Time: {bron_kerbosch_time:.4f}s")
    print(f"Brute force result  : {brute_force_result}, Time: {brute_force_time:.4f}s")
    print("--------------------")


# Plot
plt.plot(graph_sizes, bron_kerbosch_times, label='Bron-Kerbosch', color='blue')
plt.plot(graph_sizes, bron_kerbosch_brute_force_times, label='Brute force', color='red')
plt.xlabel('Graph size')
plt.ylabel('Runtime (s)')
plt.legend()
plt.show()

```

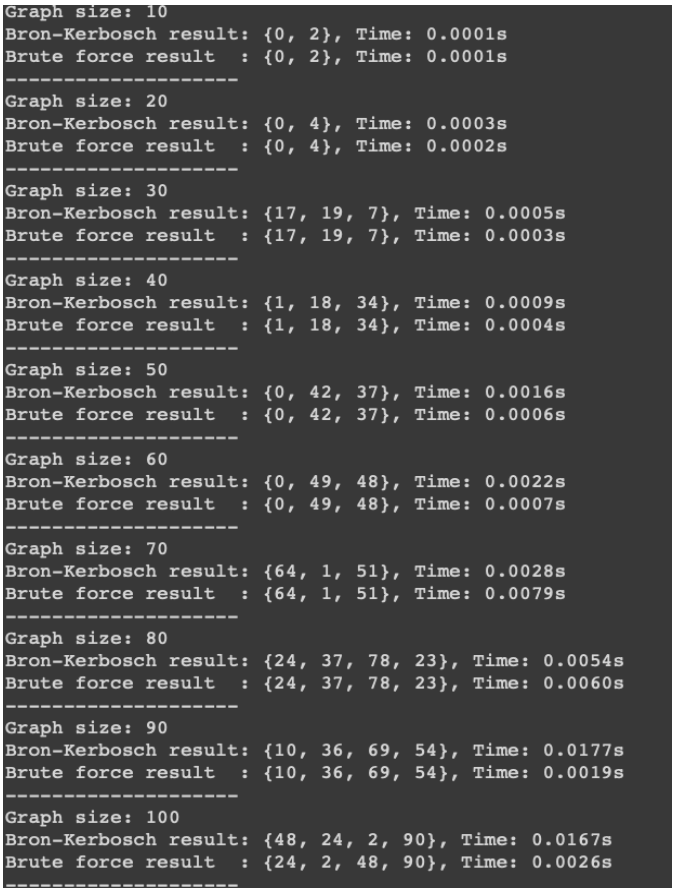
```
Graph size: 10
Bron-Kerbosch result: {8, 0, 2, 7}, Time: 0.0003s
Brute force result  : {8, 0, 2, 7}, Time: 0.0002s
--------------------
Graph size: 20
Bron-Kerbosch result: {0, 16, 19, 4, 10, 14}, Time: 0.0012s
Brute force result  : {0, 4, 10, 14, 16, 19}, Time: 0.0004s
--------------------
Graph size: 30
Bron-Kerbosch result: {17, 26, 22, 9, 10, 25}, Time: 0.0070s
Brute force result  : {9, 10, 17, 22, 25, 26}, Time: 0.0018s
--------------------
Graph size: 40
Bron-Kerbosch result: {1, 18, 19, 38, 39, 29}, Time: 0.0221s
Brute force result  : {1, 38, 18, 19, 24, 26}, Time: 0.0045s
--------------------
Graph size: 50
Bron-Kerbosch result: {34, 3, 9, 13, 14, 47, 49, 27}, Time: 0.0568s
Brute force result  : {34, 3, 9, 13, 14, 47, 49, 27}, Time: 0.0134s
--------------------
Graph size: 60
Bron-Kerbosch result: {35, 5, 6, 47, 50, 20, 24, 27}, Time: 0.1490s
Brute force result  : {5, 11, 49, 20, 54, 24, 27, 31}, Time: 0.0261s
--------------------
Graph size: 70
Bron-Kerbosch result: {0, 36, 5, 42, 43, 46, 19, 52}, Time: 0.2109s
Brute force result  : {0, 36, 5, 42, 43, 46, 19, 52}, Time: 0.0381s
--------------------
Graph size: 80
Bron-Kerbosch result: {0, 65, 36, 50, 20, 57, 29, 31}, Time: 0.3700s
Brute force result  : {0, 65, 36, 50, 20, 57, 29, 31}, Time: 0.0637s
--------------------
Graph size: 90
Bron-Kerbosch result: {2, 5, 9, 75, 44, 47, 17, 55, 57}, Time: 0.4180s
Brute force result  : {2, 5, 9, 75, 44, 47, 17, 55, 57}, Time: 0.0574s
--------------------
Graph size: 100
Bron-Kerbosch result: {0, 64, 34, 78, 15, 18, 83, 84, 53, 24}, Time: 0.5855s
Brute force result  : {0, 64, 34, 78, 15, 18, 83, 84, 53, 24}, Time: 0.1006s
--------------------
```

(Figure 6.b)



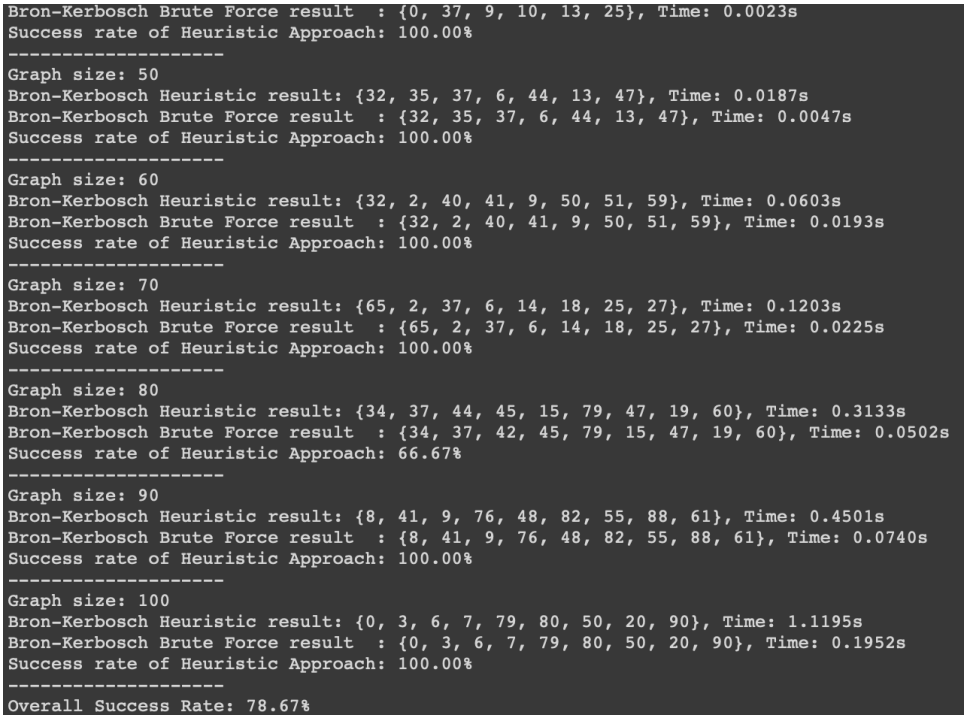
(Figure 6.c)

(Figure 6.d)

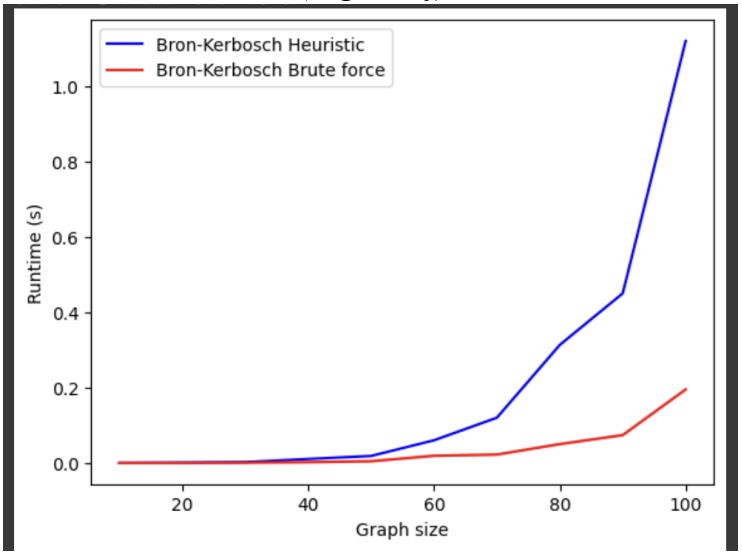


We have also written a code that performs the necessary calculations to simply compare the success rate of the heuristic algorithm to the brute force algorithm for each graph consisting of different sizes and also overall success rate when all graphs consisting of different sizes are taken into account. In *Figure 6.e* you may see the results and in *Figure 6.f* you may see the graph representing the success rate.

(Figure 6.e)



(Figure 6.f)



Moving on, we have used some statistical measures such as variance, mean, standard deviation, confidence interval. Below are two tables showing the statistical measures for the brute-force and heuristic approaches for graphs with different sizes:

(Table 6.a : Statistical Analysis for the Brute-Force Approach)

Size	Standard Deviation	Standard Error	Mean Time	Confidence Level (90.0%)	Variance
10	0.000089	0.000040	0.000195s	(0.00011014102146443544, 0.0002806747133988458)	0.000000
20	0.000124	0.000055	0.000381s	(0.00026290016812 99734, 0.000499180978110261)	0.000000
30	0.000206	0.000092	0.001058s	(0.00086136102738 7619, 0.001254365443559 6466)	0.000000
80	0.003627	0.001622	0.048509s	(0.04505108853066 506, 0.051966676514500 96)	0.000013
100	0.013202	0.005904	0.132636s	(0.12004898495484 82, 0.145222201873765 08)	0.000174

(Table 6.b : Statistical Analysis for the Brute-Force Approach)

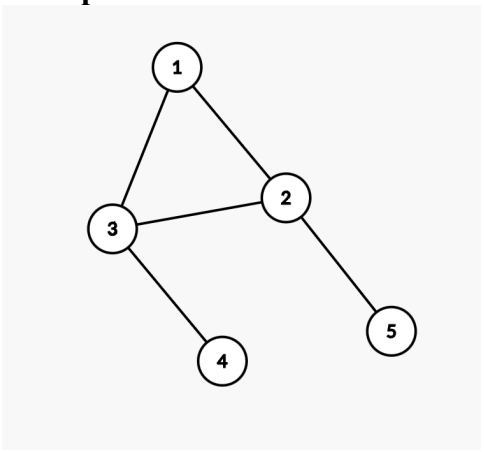
Size	Standard Deviation	Standard Error	Mean Time	Confidence Level (90.0%)	Variance
10	0.000050	0.000022	0.000253s	(0.00020590419561859942, 0.00030078296868804117)	0.000000
20	0.000312	0.000140	0.001026s	(0.00072831288962 63464, 0.001323326667258 4194)	0.000000
30	0.001325	0.000593	0.003786s	(0.00252259672506954, 0.005049863449491007)	0.000002
80	0.238813	0.106801	0.400187s	(0.17250412774609283, 0.6278691403813486)	0.057032
100	0.100700	0.045034	0.677065s	(0.58105826658255 33, 0.773071333942349 1)	0.010141

8. Experimental Analysis of the Correctness (Functional Testing)

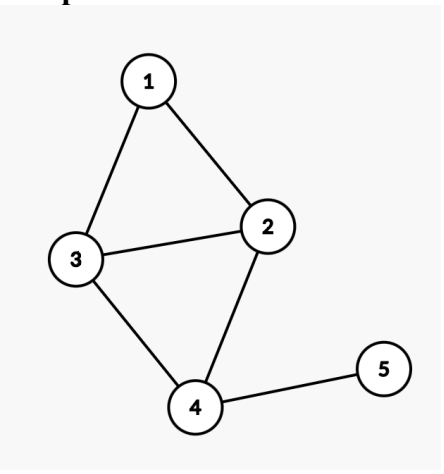
Functional testing is a type of software testing that focuses on verifying that the functionality of an application or system is working as expected. This type of testing involves testing the individual functions or features of an application to ensure that they work correctly and meet the specified requirements.The tests are designed to ensure that the application or system performs as intended and meets the functional specifications. The tests can be automated or manual, and they can be performed at various stages of the development lifecycle.
The purpose of functional testing is to identify any defects or issues in the algorithm. This helps to ensure that the algorithm is working correctly and meets the requirements and restrictions specified. Functional testing is also important for ensuring that the algorithm is reliable, scalable, and easy to use. By ensuring that the algorithm meets the functional testing requirements, it helps to reduce the risk of bugs and errors that could cause problems for the end-users. We have performed a black-box testing; the testing is considered black box testing because the focus is primarily on the external behaviour and functionality of the **bron_kerbosch()** and **bron_kerbosch_brute_force()** functions. The tester evaluates the functions based on the provided inputs (graphs) and compares the actual results (k-cliques) with the expected results. While the tester has access to the code and results, the testing approach does not heavily rely on knowledge of the internal implementation details.

In *Figure 6.a* you may see the code and the results; the code tests the bron_kerbosch() and bron_kerbosch_brute_force() functions on three different graphs. The graphs are defined using the NetworkX library's Graph() class, and the add_nodes_from() and add_edges_from() methods. Each graph is a small example network, with varying numbers of nodes and edges. For each graph, the code calls both the bron_kerbosch() and bron_kerbosch_brute_force() functions to find the maximum cliques. The results are stored in clique1, clique1_brute, clique2, clique2_brute, clique3, and clique3_brute, clique4, clique4_brute, clique5, clique5_brute, clique6, and clique6_brute, respectively. Finally, the code prints the results of the tests for each graph. Specifically, it prints the maximum cliques found by each algorithm for each graph. This allows us to compare the efficiency and accuracy so to perform a functional testing of both algorithms on various types of graphs. Overall, the code is testing the functionality and performance of the bron_kerbosch() and bron_kerbosch_brute_force() functions on different graphs. This helps to ensure that these functions are correctly identifying the maximum cliques in different types of graphs. Below you may see the visualisation of the graphs that we have used for testing.

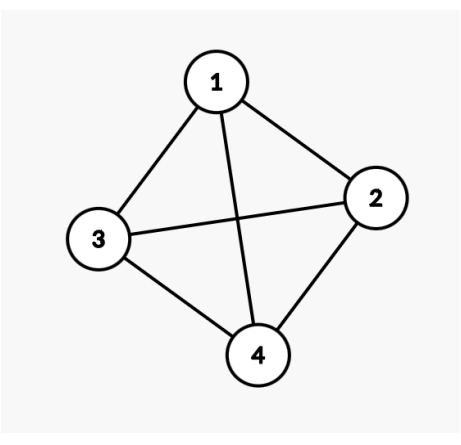
Graph 1



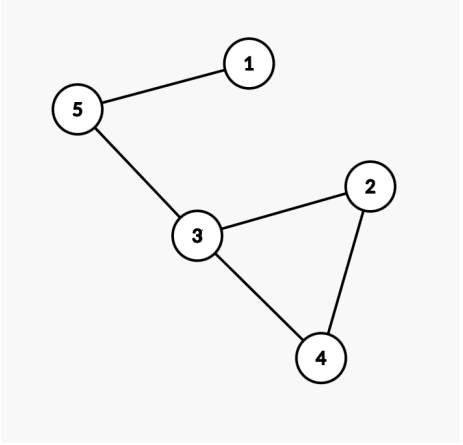
Graph 2



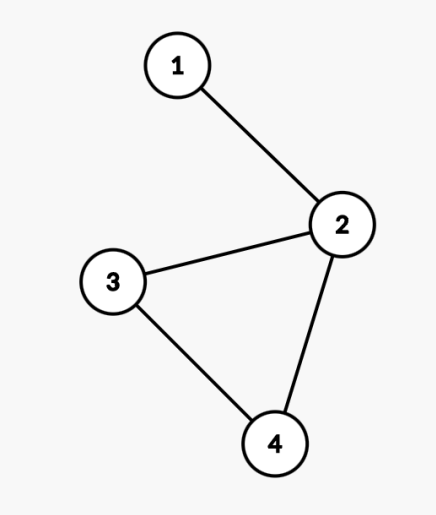
Graph 3



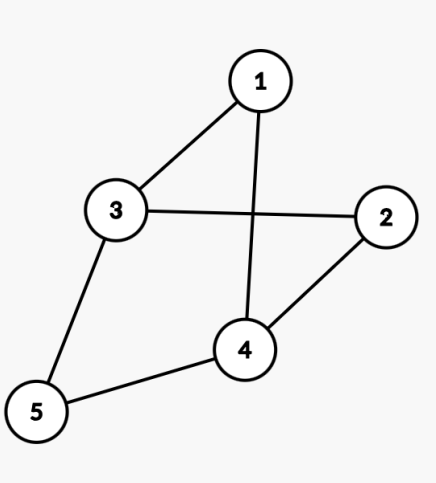
Graph 4



Graph 5



Graph 6




```python
# Test the algorithms on the graphs
clique1 = bron_kerbosch(graph1)
clique1_brute = bron_kerbosch_brute_force(graph1)

clique2 = bron_kerbosch(graph2)
clique2_brute = bron_kerbosch_brute_force(graph2)

clique3 = bron_kerbosch(graph3)
clique3_brute = bron_kerbosch_brute_force(graph3)

clique4 = bron_kerbosch(graph4)
clique4_brute = bron_kerbosch_brute_force(graph4)

clique5 = bron_kerbosch(graph5)
clique5_brute = bron_kerbosch_brute_force(graph5)

clique6 = bron_kerbosch(graph6)
clique6_brute = bron_kerbosch_brute_force(graph6)

# Print the results
print("Graph 1:")
print(f"Bron-Kerbosch Heuristic result: {clique1}")
print(f"Bron-Kerbosch brute force result: {clique1_brute}")
print("--------------------")

print("Graph 2:")
print(f"Bron-Kerbosch Heuristic result: {clique2}")
print(f"Bron-Kerbosch brute force result: {clique2_brute}")
print("--------------------")

print("Graph 3:")
print(f"Bron-Kerbosch Heuristic result: {clique3}")
print(f"Bron-Kerbosch Brute Force result: {clique3_brute}")
print("--------------------")

print("Graph 4:")
print(f"Bron-Kerbosch Heuristic result: {clique4}")
print(f"Bron-Kerbosch Brute Force result: {clique4_brute}")
print("--------------------")

print("Graph 5:")
print(f"Bron-Kerbosch Heuristic result: {clique5}")
print(f"Bron-Kerbosch Brute Force result: {clique5_brute}")
print("--------------------")

print("Graph 6:")
print(f"Bron-Kerbosch Heuristic result: {clique6}")
print(f"Bron-Kerbosch Brute Force result: {clique6_brute}")
print("--------------------")
```

```
Graph 1:
Bron-Kerbosch Heuristic result: {1, 2, 3}
Bron-Kerbosch brute force result: {1, 2, 3}
--------------------
Graph 2:
Bron-Kerbosch Heuristic result: {1, 2, 3}
Bron-Kerbosch brute force result: {1, 2, 3}
--------------------
Graph 3:
Bron-Kerbosch Heuristic result: {1, 2, 3, 4}
Bron-Kerbosch Brute Force result: {1, 2, 3, 4}
--------------------
Graph 4:
Bron-Kerbosch Heuristic result: {2, 3, 4}
Bron-Kerbosch Brute Force result: {2, 3, 4}
--------------------
Graph 5:
Bron-Kerbosch Heuristic result: {2, 3, 4}
Bron-Kerbosch Brute Force result: {2, 3, 4}
--------------------
Graph 6:
Bron-Kerbosch Heuristic result: {1, 3}
Bron-Kerbosch Brute Force result: {1, 3}
--------------------
```

(Figure 6.a)

9. Discussion

We have designed and implemented 2 algorithms for the k-clique problem which is an Np-Complete Problem. First algorithm we designed is the Brute Force algorithm, which iterates through all possible combinations of nodes and returns the clique with the maximum size. The second algorithm was the Heuristic algorithm. In the provided results, an experimental analysis was conducted to evaluate the performance and quality of the heuristic Bron-Kerbosch algorithm compared to the brute-force algorithm. Here is a breakdown of the findings:

Performance Analysis:

The performance analysis suggests that the Bron-Kerbosch algorithm may be slower than the brute-force algorithm in certain cases. The recursive nature of the Bron-Kerbosch algorithm and the memory required to store intermediate results can contribute to its slower performance. The worst-case time complexity of the Bron-Kerbosch algorithm is O(3^(n/3)), meaning the running time can grow exponentially with the graph's size. On the other hand, the brute-force algorithm enumerates all possible subsets of nodes and checks for cliques, with a worst-case time complexity of O(2^n), also exponential. Brute Force algorithm guarantees that the solution is correct. However in situations where the input size is very large and the graph is not dense as it must be, this algorithm takes so much time that it is necessary to reach the optimal or nearly optimal solution. Since the brute force algorithms running time increases exponentially, there is a trade off between finding the optimal solution and runtime. Because of this problem, we designed and implemented another algorithm, which is called the Bron-Kerbosch algorithm. This algorithm is based on a recursive backtracking approach, where at each step, it greedily selects a vertex from the graph and explores its neighbours. In contrast, a brute force algorithm for finding a maximum clique would systematically examine all possible subsets of vertices to determine if they form cliques. This approach involves checking an exponential number of combinations and becomes impractical for large graphs.

Quality Analysis:

The analysis emphasises that it is inaccurate to claim that the Bron-Kerbosch algorithm is always inferior to the brute-force algorithm in terms of performance. Both algorithms have their strengths and weaknesses, and the choice depends on factors such as graph size, structure, and specific problem requirements. The Bron-Kerbosch algorithm is considered heuristic and can be effective in certain scenarios.

Comparison of Experimental Results:

The experimental results, presented in Figure 6.b and Figure 6.d, highlight the differences between the Bron-Kerbosch and brute-force algorithms in terms of runtime. It is noted that the difference in runtime between the two algorithms is less pronounced in Figure 6.d compared to Figure 6.b. This is attributed to modifying the "p" parameter, which affects the probability of edge occurrence between nodes. When the "p" value is decreased, resulting in less dense graphs, the performance gap between the Bron-Kerbosch and brute-force algorithms reduces.

Defects in the Algorithm:

The provided analysis does not explicitly mention any defects in the algorithm. However, it focuses on the runtime and quality aspects of the Bron-Kerbosch and brute-force algorithms. It is essential to note that an algorithm may have defects or limitations that affect its correctness or efficiency, but such defects are not discussed in the provided results.

Consistency between Theoretical and Experimental Analysis:

Based on the information provided, there is no explicit comparison between the theoretical and experimental analysis. Theoretical analysis typically involves evaluating the algorithm's complexity, correctness, and expected behaviour, while the experimental analysis involves measuring performance and quality using real-world data. Without specific details on the theoretical analysis, it is challenging to determine if there is any inconsistency between the two analyses.

Overall, the key difference is that Bron-Kerbosch is a more efficient and optimised algorithm that makes locally optimal choices, whereas a brute force algorithm exhaustively searches through all possible combinations without any optimizations.

References

Chakrabarty, B., & Parekh, N. (2016). NAPS: Network Analysis of Protein Structures. *Nucleic Acids Research*, *44*(W1), W375–W382. https://doi.org/10.1093/nar/gkw383

Estrada, E. (2013). Journal of Complex Networks: Quo Vadis? *Journal of Complex Networks*, *1*(1), 1–2. https://doi.org/10.1093/comnet/cnt008

Journal of Combinatorial Optimization. (2023, April 14). Springer. https://www.springer.com/journal/10878/

Mouatadid, L. (1989). *Introduction to Complexity Theory: CLIQUE is NP-complete*. https://d1b10bmlvqabco.cloudfront.net/attach/hsoyaw6eh1215z/hcf60z2lm1mde/hy4775493rba/clique.pdf

NP-Completeness. (n.d.). https://sites.cs.queensu.ca/courses/cisc365/Record/20191119%20-%20CNF-SAT%20to%20k-Clique.pdf