CSE 462

Algorithm Report

# Maximum Clique Problem

**Group 1**

| | |
|---|---|
| Rudaiba Adnin | 1505032 |
| Abhik Bhattacharjee | 1505040 |
| Kazi Samin Mubashir | 1505041 |
| Rafi-Ur-Rashid | 1505045 |
| Soumit Kanti Saha | 1505047 |

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
**(BUET)**
Dhaka 1000

December 18,2020

# Contents

## 1 Problem Definition of Maximum Clique Problem

A clique, C, in an undirected graph G = (V, E) is a subset of the vertices such that every two distinct vertices are adjacent. The maximum clique problem seeks to find a clique of the largest possible size in a given graph.

Problems that are reduced to clique problem are 3-SAT,SAT, Circuit SAT and Independent Set. Problems That Clique Problem Reduces to are Vertex Cover, Hamiltonian Cycle, Travelling Salesman Problem, 3-coloring.

## 2 Applications of Clique Problem

- Application in Chemistry
  Find chemicals that match a target structure, model molecular docking and binding sites of chemical reaction, find similar structures within different molecules.
- Application in Bioinformatics
  Clique finding algorithms are used to infer evolutionary tree, minimize gene duplication and losses, predict protein structures, find closely interacting clusters of protein.
- Application in Automatic Test Pattern Generation
  Finding cliques can help to bound the size of a test set in this case.
- Some Other Applications
  Listing the cliques in a dependency graph is an important step in the analysis of some random processes. Cliques finding algorithm is used to find a group of mutual friends in a social network where vertices represent people and edge among them represent acquaintance.

## 3 Proof of NP-completeness of Maximum Clique Problem

Clique problem is NP complete.To prove this, we have to show,

- The clique problem is in NP.
- It is NP-hard or in other words a known NP-hard problem can be reduced to it in polynomial time

The clique problem is in NP.

### 3.1 Proof of Maximum Clique Problem is in NP

Given a set of k vertices we can easily verify if all of them are adjacent by checking the adjacency matrix of the graph. Since this takes polynomial time, clique problem is in NP.

### 3.2 Proof of NP-hardness Using 3-SAT

**Y is satisfiable if and only if G has a k-clique**

If Y is satisfiable, let A be a satisfying assignment. We construct a set S by taking a literal from each clause that is True in A. Since no two literals in S are from the same clause and all of them are simultaneously true, all the corresponding nodes in the graph are connected to each other forming a k-clique. Hence the graph has k-clique.

If the graph G has a k-clique, the clique has exactly one node from each cluster. All nodes in a clique are connected hence all corresponding nodes can be assigned true simultaneously. Each literal belong to exactly one of the k-clauses. Hence Y is satisfiable.

### 3.3 Proof of NP-hardness Using Independent Set

**There is an independent set of size k in G' if and only if G has a k-clique**

If there is an independent set of size k in the graph G, it implies no two vertices share an edge in G which further implies all of those vertices share an edge with all others in the complement graph G' forming a clique that is there exists a clique of size k in G'.

If there is a clique of size k in the complement graph G', it implies all vertices share an edge with all others in G' which further implies no two of these vertices share an edge in G forming an Independent Set that is there exists an independent set of size k in G.

## 4 Exact Algorithms of Maximum Clique Problem

Exact algorithms are algorithms that always solve an optimization problem to optimality. Unless P = NP, an exact algorithm for an NP-hard optimization problem cannot run in worst-case polynomial time.

### 4.1 Brute-force Algorithm

The brute force algorithm for max clique problem which have exponential time complexity is shown in Algorithm 1. For loop in *MAXCLIQUE(G)* in Algorithm 1 runs in O($2^n$) since power set of a set of size n has 2n elements. *Is_Clique* in Algorithm 1 function runs in O($n^2$). Therefor, total time complexity is O($2^n n^2$) which is exponential.

**Algorithm 1** Brute Force algorithm

```
procedure MAXCLIQUE($G$)                                    ▷ The graph G
    $max\_size \leftarrow 0$
    $max\_clique \leftarrow \{\}$
    for each set $X$ in power set of all vertices of $G$ do
        $clique\_size \leftarrow$ IS_CLIQUE($X$)
        if $clique\_size > max\_size$ then
            $max\_size \leftarrow clique\_size$
            $max\_clique \leftarrow X$
        end if
    end for
    return $max\_clique$
end procedure
procedure IS_CLIQUE($V$)                                    ▷ Set V
    for each vertex $X$ in set $V$ do
        if $X$ is adjacent to all vertices of set $V$-$X$ then
            return $size(V)$
        end if
    end for
    return $-1$
end procedure
```

## 4.2 Literature Review

Carraghan and Pardalos introduced a simple-to-implement algorithm that avoids enumerating all cliques via a pruning strategy which reduces the search space tremendously.The algorithm works by performing at each step a depth first search from each vertex [1].

Ostergard devised an algorithm that incorporated an additional pruning strategy to the one by Carraghan and Pardalos. The opportunity for the new pruning strategy is created by reversing the order in which the search is done by the Carraghan-Pardalos algorithm [3].

A number of existing branch-and-bound algorithms for maximum clique use a vertex coloring of the graph to obtain an upper bound on the maximum clique. A popular and recent algorithm based on this idea is the algorithm of Tomita and Seiku [5].

## 4.3 Fast Algorithm for the Maximum Clique Problem on Massive Sparse Graphs by Pattabiraman and Gebremedhin

Bharath Pattabiraman and Gebremedhin (2012) proposed an exact algorithm that employs novel pruning techniques to very quickly find maximum cliques in large sparse graphs [13].

This algorithm consists of several pruning steps. In most existing algorithms, we prune in the case where even if all vertices of U (the set containing neighbours of a vertex) were added to get a clique, its size would not exceed that of the largest clique encountered so far in the search. This algorithm adopts some novel pruning steps.

The main idea of the algorithm is the maximum clique in a graph can be found by computing the largest clique containing each vertex and picking the largest among these. To obtain the largest clique containing a vertex v, it is sufficient to consider only the neighbors of vertex v. When computing the largest clique containing vertex $v_i$ , we don't need to consider its neighbor $v_j$ if the largest clique for $v_j$ has already been found. The pseudocode of the algorithm is shown in Algorithm 2.

### 4.4 Polynomial Time Variants

- **Circle Graph and Perfect graph** A circle graph is the intersection graph of a set of the chords of a circle. An algorithm which requires O($n^2$) steps to generate one maximum clique in a circle graph is presented in. A perfect graph is a graph in which the chromatic number of every induced subgraph equals the size of the largest clique of that subgraph. In all perfect graphs, the graph coloring problem, maximum clique problem, and maximum independent set problem can all be solved in polynomial time.

- **Planar Graph** Kuratowski's theorem states that a finite graph G is planar, if there exists no possible subdivisions of K5 or K3,3 in G In other word, in a planar graph, there are no cliques of size 5 or more. Suppose the problem states "is there a clique of size at least k in a given planar graph G?" Now, if $k \geq 5$, the immediate answer is NO. If $k \leq 4$, we can simply check every subset of the vertices of size k. There are O($n^k$) such subsets. If any one of them induces a clique, we answer YES. Otherwise, we answer NO. Hence, the problem is in P.

## 5 Approximation Algorithms of Maximum Clique Problem

Approximation algorithms are efficient algorithms that find approximate solutions to optimization problems (in particular NP-hard problems) with provable guarantees on the distance of the returned solution to the optimal one.

### 5.1 No PTAS for Maximum Clique Problem unless P=NP

Firstly we need to accept the Corollary:

**MAX3SAT** does not have a PTAS unless P = NP

Now we need to prove:

**Algorithm 2** Algorithm for fiding the maximum clique of a given graph. *Input* : Graph $G = (V, E)$, lower bound on clique $lb$ (default,0). *Output*: Size of maximum clique.

```
procedure MAXCLIQUE($G = (V, E), lb$)
    $max \leftarrow lb$
    for i: 1 to n do
        if $d(v_i) \geq max$ then                      ▷ Pruning 1
            $U \leftarrow \emptyset$
            for each set $v_j \in N(v_i)$ do
                if $j > i$ then                        ▷ Pruning 2
                    if $d(v_j) \geq max$ then          ▷ Pruning 3
                        $U \leftarrow U \cup \{v_j\}$
                    end if
                end if
                CLIQUE($G, U, 1$)
            end for
        end if
    end for
end procedure
```

~ Subroutine

```
procedure CLIQUE($G = (V, E), U, size$)
    if $U = \emptyset$ then
        if $size > max$ then
            $max \leftarrow size$
        end if
        return
    end if
    while $|U| > 0$ do
        if $size + |U| \leq max$ then                  ▷ Pruning 4
            return
        end if
        Select any vertex $u$ from $U$
        $U \leftarrow U \setminus \{u\}$
        $N'(u) := \{w | w \in N(u) \wedge d(w) \geq max\}$   ▷ Pruning 5
    end while
    CLIQUE($G, U \cap N'(u), size + 1$)
end procedure
```

There is a polynomial time reduction from **MAX3SAT** to **CLIQUE** such that for all 3CNF expressions $\phi$ with m clauses, a graph G$\phi$ with 3m nodes can be constructed with the property that

$\sigma(\phi) = c \Leftrightarrow (G\phi) = c.m$ for any $c \in [0, 1]$

Here,

$\sigma(\phi)$ = the maximum number of satisfiable clauses (s) divided by the total number of clauses (m)

#### 5.1.1 Construction:

- Let $\phi = C1 \wedge C2 ... \wedge Cm$ be a 3CNF with variables $x_1, x_2 ..., x_n$.
- Now we represent each $C_i$ based on the Boolean variables $x_{i1}, x_{i2}, x_{i3}$ by the graph $G_i = (V_i, E_i)$ with $V_i = \{x_{i1}, x_{i2}, x_{i3}\}$ and $E_i = \phi$
- We construct the graph $G\phi = (V, E)$ by taking the union of the $V_i$ 's and drawing an edge from $x_{ik} \in V_i$ to $x_{jl} \in V_j$ for $i \neq j$ if and only if the literals they represent are not negations of each other.

#### 5.1.2 First Part

$\sigma(\phi) = c \Rightarrow w(G\phi) = c.m$

- If $\sigma(\phi) = c \qquad c.m$ is the the maximum number of satisfiable clauses (s)
- Then there must be s different sets of Vi that contain a node representing a satisfied literal
- Since by the construction no one of these nodes can represent the negation of another, they must form a completely connected graph.

And therefore, $w(G\phi) \geq s = c.m$

#### 5.1.3 Second Part

$w(G\phi) = c.m \Rightarrow \sigma(\phi) = c$

- $w(G\phi) = c.m$, then there must be $c.m$ nodes from different $V_i$ that form a completely connected graph
- Then there must be $c.m$ different sets of $V_i$ that contain a node representing a satisfied literal
- Since no one of these nodes represents the negation of another, it is possible to have an assignment that satisfies all clauses represented by these $V_i$'s. And therefore, $s \geq c.m$ or $\sigma(\phi) \geq c$

### 5.2 No Constant Factor PTAS for Maximum Clique Problem Unless P=NP

To prove this, we will exploit a self-improvement property of CLIQUE which is defined by means of a graph product.
Now we assume that the graph has a self-loop at every one of its nodes.That means, for a graph $G = (V, E)$ , we define $E' = E \cup \{\{u, u\} : u \in V\}$ and consider then instead of G the graph $G' = (V, E')$

**Product graph:** For two graphs $G1 = (V1, E1)$ and $G2 = (V2, E2)$ their product $G1 \times G2$ is the graph whose vertex set is V1 $\times$ V2 and edge set is :

$\{\{(u1, v1), (u2, v2)\} : \{u1, u2\} \in E1' and \{v1, v2\} \in E2'\}$

We can show without much effort that, $w(G1 \times G2) = w(G1) \ . \ w(G2)$

Now consider the graph $G^K = G \times ... \times G$ (k times).

Then,

$w(G^K) = m^K$ where $w(G) = m$

There can be an efficient algorithm S that, given a clique of size c in $G^k$ , can also find a clique of size $\sqrt[k]{c}$ in G:

Let C be the clique found in $G^K$ , and suppose $U_i$ be the set of nodes used in the $i^{th}$ coordinate of the nodes in C. Then each $U_i$ must be a clique in G.

Since, $\prod^{i=K} |U_i| \geq |C|$, there must be a $U_i$ with $|U_i| \geq \sqrt[k]{|c|}$

We have already proved that there cannot be a polynomial time approximation scheme for CLIQUE with an approximation ratio $1 + \epsilon$

Suppose that we have some $1 + \delta$-approximation scheme **A** for CLIQUE for some constant$\delta$. Then we transform algorithm **A** into the following algorithm **B**:

Given any graph G, we compute the graph product $G^k$ , then use **A** to compute a clique C in $G^k$ , and then apply algorithm **S** to transform C into a clique C' for G.

Since A is a $1 + \delta$-approximation scheme it holds that :

$$|C| \geq \frac{1}{1+\delta}.w(G^k) = \frac{1}{1+\delta}w(G)^k$$

Then using algorithm S, we obtain a clique C' with

$$|C'| \geq \sqrt[k]{\frac{1}{1+\delta}.w(G^k)} = \sqrt[k]{\frac{1}{1+\delta}}w(G)$$

Let k now be chosen so that $\frac{1}{1+\delta} < (\frac{1}{1+\epsilon})^k$

$$|C'| > \frac{1}{1+\epsilon}.w(G)$$

But this implies that we found a polynomial time approximation scheme for CLIQUE with an approximation ratio of less than $1+\epsilon$, which is a contradiction to our assumption.

## 5.3 Literature Review

The problem of max independent set is strongly related to the max clique problem (by complementing the graph), and hence shares the same approximation ratio. The chromatic number of a graph is the smallest number of independent sets that cover all vertices of the graph. It shares essentially the same hardness of approximation results as max clique (this is an empirical observation rather than a theorem). In terms of approximation algorithms, Halldorsson (1993) shows that the chromatic number can be approximated within a ratio of $O(n(loglogn)^2/(logn)^3)$, which is better than the best approximation ratio known for max clique. Feige (2004) describes a polynomial time algorithm that finds a clique of size $t*(log_3k(n/t)-3)$, in any graph that has clique of size $n/k$. Feige provides an approximation algorithm that finds a clique with a number of vertices within a factor of $O(n(loglogn)^2/(logn)^3)$ of the maximum,matching the known approximation ratio for the chromatic number. Although the approximation ratio of this algorithm is weak, it is the best known to date [6].

Fayaz, Abdul and Asadullah proposed a technique to use the approximation algorithms of Minimum Vertex Cover (MVC) for the solution of the Maximum Clique (MC) problem [14].

## 5.4 Approximation Algorithm by Fayaz, Abdul and Asadullah

Fayaz, Abdul and Asadullah (2018) proposed a technique to use the approximation algorithms of Minimum Vertex Cover (MVC) for the solution of the Maximum Clique (MC) problem [14].

We take input graph G with vertices V and edges E. Then, we take the complement of G which is G' and apply the approximation algorithm for MVC and save the result to P We remove P from G and save the cardinality to T. Then we decreament T by 1 and calculate the degree of each vertex in G. If the degree is less than T we remove the vertex from G and repeat the procedure. If no, then we calculate $K_n$ which is $n(n-1)/2$ and if kn is equal to the number of edges in G we have our maximum clique.

**Approximation Ratio**
The Approximation Ratio of the MVC algorithm is the Approximation Ratio of this approximation algorithm for Max CLIQUE. Some of the MVC algorithms: Maximum Degree Greedy (MDG), Vertex Support Algorithm (VSA), Nearly Optimal Vertex Cover (NOVAC-1), A dvanced Vertex Support Algorithm (AVSA), Modified Vertex Support Algorithm (MVSA)

## 5.5 Approximation Algorithm by Feige

Feige (2004) describes a polynomial time algorithm that finds a clique of size $t*(log_{3k}(n/t)-3)$, in any graph that has clique of size $n/k$ [6].

Let G(V,E) be an input graph with n vertices which contains a clique of size $n/k$. In this graph we wish to find a clique large as possible. For a parameter t, we shall give an algorithm that finds a clique of size at least $t(log_{3k}(n/t)-3)$. For every $b>0$, whenever $k<(logn)b$, we can choose $t=\theta(logn/loglogn)$, and then our algorithm finds a clique of size $O((logn/loglogn)^2)$ in polynomial time.

In the course of this algorithm, it will be considered vertex induced subgraphs of G. Definition: Let G be a graph with a clique of size $n/k$. A vertex induced subgraph S is called poor if it does not contain a clique of size $|S|/2k$. Lemma: Let G be a graph with a clique of size $n/k$. Let S1,S2,... be arbitrary disjoint poor subgraphs of G (with no clique of size $|Si|/2k$, respectively). Let G'(V',E')

**Algorithm 3** Approximate Maximum Clique Algorithm (AMCA). *Input* :Graph $G = (V, E)$ . *Output*: Max Clique (MC).
$AP\_MVC$ : An approximation algorithm for Minimum Vertex Cover

```
1:  procedure AMCA($G = (V, E)$)
2:      $G'(V, E') \leftarrow$ complement of $G(V, E)$
3:      $MC \leftarrow \emptyset$
4:      $Pi \leftarrow \emptyset$
5:      $Pi \leftarrow AP\_MVC(G(V, E'))$
6:      $T \leftarrow |G| \setminus |Pi|$
7:      $T \leftarrow T - 1$
8:      for each $v_i \in V$ do
9:          calculate $deg(v_i)$
10:         if $deg(v_i) < T$ then
11:             $G \leftarrow |G| \setminus v_i$
12:         end if
13:         if $|G| = Kn$ then
14:             $MC \leftarrow V$
15:             return $MC$
16:         end if
17:     end for
18: end procedure
```

be the vertex induced subgraph of G that remains after removing the poor subgraphs. Then $|V'| \geq n/2k$, and G' contains a clique of size at least $|V'|/k$.

Our algorithm works in phases. The input to a phase is a vertex induced subgraph $G'(V', E')$ of $G$. (The input to the first phase is the graph $G$ itself.) This subgraph contains a clique of size $|V'|/k$. A phase is completed when one of the following two conditions hold:

1. A clique of size $tlog_{3k}(|V'|/6kt)$ is found.

2. A poor subgraph is found.

If upon finishing a phase the first condition holds, then the algorithm terminates. If upon finishing a phase the second condition holds, then the poor subgraph is removed from G' and a new phase begins with the resulting graph.

There are several iterations in a phase.

**Input to an iteration**

Subgraph $G"(V", E")$ from G',a set of vertices C that for a clique in $V' \backslash V"$

**Output of iteration**

At the end of iteration either C increases or V" is declared as poor.

**Iteration Steps**

Following are the iteration steps.

1. If $|V"| < 6kt$, phase ends and output is C

2. Partition $V"$ in disjoint sets all having 2kt size.

3. In each part Pi, consider all possible subset $S_{ij}$ of vertices of cardinality t.
4. Let $N(S_{ij})$ be the set of vertices that are neighbors in G" with the vertices of $S_{ij}$ which means $N(S_{ij})$ and $S_{ij}$ form the two sides of a complete bipartite graphs of G".
5. Call $S_{ij}$ good, if the subgraph of G" induced on $S_{ij}$ is a clique and $N(S_{ij}) \geq V"/2k - t$
6. If some $S_{ij}$ is good, set $C = C \cup S_{ij}$ and go to next iteration with inputs: Subgraph induced on $N(S_{ij})$ as new G" and C.
7. If no $S_{ij}$ is found good, declare V" poor and end the phase.

**Interpretation of output of a phase**
$V"$ declared poor subgraph of G induced on V" does not have a clique of size $|V"|/2k$. Output a set C C contains at least $tlog_{3k}(n'/6kt)$ vertices and these vertices form a clique in $G'$, the input graph of that phase.
**Time complexity**
There are total $|V"|/2kt$ total partitions and 2ktCt subsets in each part. Therefore, total running time O(2ktCt * nc), where c is some universal constant.We are generally interested in the cases where $k \leq (logn)b$, for $b > 0$, in which we take t=log n / log log n, thus ensuring polynomial running time. Approximation Ratio of the algorithm is $O(n(loglogn)^2/(logn)^3)$.

## 6 Meta-heuristic Algorithms of Maximum Clique Problem

A metaheuristic is defined as an iterative approach which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting search space. They are inspired by observing the phenomena occurring in nature. Popularly algorithms include genetic algorithms, particle swarm optimization, ant algorithms, and bee algorithms.

### 6.1 Literature Review

A Reactive Local Search (RLS) is an algorithm to the MCP which is based on local search complemented by a feedback scheme to determine the amount of diversification [2].

The Deep Adaptive Greedy Search (DAGS) algorithm uses an iterated greedy construction procedure [7]. The k-opt algorithm (KLS) is based on a conceptually simple variable depth search procedure that uses elementary search steps in which a vertex is added to or removed from the current clique [8]. The Dynamic Local Search (DLS-MC) scheme is actually a slight simplification of the Deep Adaptive Greedy Search (DAGS) [10].

The Phased Based local search (PLS), to cope with graphs of different structures, combines three sub-algorithms which use different vertex selection rules: random selection, random selection among those with the highest vertex degree, and random selection within those with the lowest vertex penalty [9].The Co-operating Local search (CLS) further improves over PLS by incorporating four low level heuristics which are effective for different instance types [11].

The capabilities of Ant Colony Optimization (ACO), a bio-inspired approach for solving the maximum clique problem was investigated by Fenet and Solnon.The main idea of ACO is to model the problem as the search for a minimum cost path in a graph [4]. Roger, Kristopher, Roman have proposed a Hybrid Genetic Algorithm, a solution that implements a migration mechanism composed of two sub-population [12].

## 6.2 Co-operative Local Search Algorithm by Pullan, Mascia and Brunato

CLS is a parallel maximum clique hyper-heuristic that incorporates four low level heuristics [11].

- Greedy Search (GREEDY) which uses random selection within vertex degree, is biased towards higher degree vertices and performs limited plateau search.
- Level Search (LEVEL) which uses random selection within vertex degree, is biased towards higher degree vertices and performs extensive plateau search.
- Focus Search (FOCUS) which identifies a vertex degree as being the focus of the current iteration and selects vertices with a goal of obtaining an average vertex degree for the clique as close as possible to this focus vertex degree.
- Penalty Search (PENALTY), which performs random selection within minimum vertex penalties and is biased towards lower degree nodes

The pseudocode of the algorithm is shown in Algorithm 4

## 6.3 Genetic Algorithm by Roger, Kristopher and Roman

Roger, Kristopher, Roman have proposed a Hybrid Genetic Algorithm, a solution that implements a migration mechanism composed of two sub-population [12].

Given a set of vertices, this algorithm finds the maximal clique using

- A “Relax”, (adds random vertices to a sub-graph)
- A “Repair”(extracts a small maximal clique from the current sub-graph)

**Algorithm 4** Co-Operating Local Search. *Input* : Integer tcs(target clique size), max-time. *Output*: Clique of cardinality tcs or 'failed'

```
1: procedure CLS-Heuristic($tcs, max\_time$)
2:     <Randomly select a vertex $v \in V, K \leftarrow \{v\}$ >
3:     while $time < max\_time$ do
4:         while $true$ do
5:             while $C_0(K) \neq \emptyset$ do
6:                 $v \leftarrow Select(C_0(K))$
7:                 $K \leftarrow K \cup \{v\}$
8:                 if $|K| = tcs$ then
9:                     return $K$
10:                end if
11:            end while
12:            if $C_1(K) \neq \emptyset$ then
13:                $v \leftarrow Select(C_1(K))$
14:                $K \leftarrow [K \cup \{v\}] \setminus \{i\}, where\{i\} = K \setminus N(v)$
15:            end if
16:        end while
17:        restart
18:    end while
19:    return 'failed'
20: end procedure
```

- An "Extend" step(generates a bigger maximal clique by adding random nodes to the clique obtained in the repair step)

The HGA is then obtained by combining the heuristic local search with a genetic algorithm. The heuristic function is added after the crossover operation to find maximal clique.

A migration mechanism based on two sub-populations, a main sub-population and a secondary sub-population. The main population will have the role of a standard HGA.

The Main sub-population uses a generic HGA defined previously. This sub-population specializes in the exploitation of the data. The secondary sub-population will serve as a pool that generates the new chromosomes for migrating into the main sub-population.

The secondary sub-population will be dedicated to exploration of the data. An effort is made to diversify the population. Transfer the best of its population to the main sub-population through a migration mechanism. The pseudocode of the algorithm is shown in Algorithm 5.

## 6.4 Ant Colony Optimization by Fenet and Solnon

Ant colony optimization is a bio inspired approach, models the problem as the search for a minimum cost path in a graph, ants walk through the graph to

**Algorithm 5** Genetic Algorithm for finding the maximum clique of a given graph

```
procedure GA
    Initialize population()
    Evaluate population()
    while end condition == $false$ do
        Select parent()
        Crossover()
        Mutation()
        Evaluate population()
    end while
end procedure
```

find different paths, co-operation among ants is performed through pheromone laying. Ant Clique characteristics-

- Uses a simple greedy heuristic to generate maximal cliques.
- ACO is used to select a vertex to enter the clique at each time step.
- vertex is chosen with a probability dependent on pheromone trails between it and the current clique.
- Pheromone trails deposited by the ants are proportional to the quality of the previously computed cliques.

The pseudocode of the algorithm is shown in Algorithm 6.

**Algorithm 6** Ant Colony Algorithm for finding the maximum clique of a given graph

```
procedure ANT-CLIQUE
    Initialize pheromone trails
    repeat
        for k: 1 to nbAnts do
            construct a clique $C_k$
            Update pheromone trails w.r.t $C_1$.....$C_{nbAnts}$
        end for
    until max cycles reached or optimal solution found
end procedure
```

```
1: procedure CONSTRUCTION OF CLIQUE BY ANTS
2:     choose randomly a first vertex $v_f \in V$
3:     $C \leftarrow v_f$
4:     $Candidates \leftarrow \{v_i/(v_f, v_i)\} \in E$
5:     while $Candidates \neq \phi$ do
6:         choose a vertex $v_i \in Candidates$ with probability
           $p(v_i) = \frac{[T_c(v_i)]^\alpha}{\sum_{v_j \in Candidates}[T_c(v_j)]^\alpha}$        ▷ $T_C(v_i) = \sum_{v_j \in C} T(v_i, v_j)$
7:         $C \leftarrow C \cup \{v_i\}$
8:         $Candidates \leftarrow Candidates \cap \{v_j/(v_i, v_j)\} \in E$
9:     end while
10:    return C
11: end procedure
```

## 7 Implementation of Algorithms to Solve Maximum Clique Problem

We have implemented two algorithms to solve Maximum Clique Problem

- Exact Algorithm: Branch and Bound by Pattabiraman and Gebremedhin (2012)
- Meta-heuristic Algorithm: Ant Colony Optimization( ACO) by Fenet and Solnon (2003).

### 7.1 Implementation of Exact Algorithm: Branch and Bound by Pattabiraman and Gebremedhin

We have used python as the programming language to implement the exact algorithm for max clique problem.

```python
import logging
import copy
import time
import random
from tqdm import tqdm
import numpy as np


logging.basicConfig(level=logging.INFO, format='[%(asctime)s] - %(
    message)s', datefmt='%H:%M:%S')

class BranchAndBound:
    def __init__(self, lb=0):
        self.lb = lb

        self.best_clique = []
        self.cur_max = 0

    def Clique(self, graph, U, size, cur_clique):
        if len(U) == 0:
            if size > self.cur_max:
                self.cur_max = size
                self.best_clique = cur_clique
            return

        while len(U) > 0:
            if size + len(U) <= self.cur_max: # pruning 4
                return

            vertex = U.pop()
            new_cur_clique = cur_clique[:]
            new_cur_clique.append(vertex)

            # pruning 5
            neib_vertex = set(v for v in graph[vertex] if len(graph
    [v]) >= self.cur_max)
            new_U = U.intersection(neib_vertex)

            self.Clique(graph, new_U, size + 1, new_cur_clique)

    def MaxClique(self, graph, lb=0):
        # nodes are labeled as 1, 2, ....no_of_vertices
        no_of_vertices = len(graph)
        self.cur_max = lb
        self.best_clique = []

        for i in tqdm(range(1, no_of_vertices + 1), desc='Running
    bnb loops'):
            if str(i) not in graph:
                continue
            neib_vi = graph[str(i)]

            # pruning 1
            if len(neib_vi) >= self.cur_max:
                U = set()
                cur_clique = [str(i)]

                for j in neib_vi:
                    # pruning 2
                    if int(j) > i:
                        # pruning 3
                        if len(graph[j]) >= self.cur_max:
                            U.add(j)

                self.Clique(graph, U, 1, cur_clique)


    def run(self, graph):
        start_time = time.time()
        self.MaxClique(graph, self.lb)
        end_time = time.time()

        s = self.cur_max
        t = (end_time - start_time) * 1000

        logging.info(f"clique size: {s}, time(ms): {t:.3f}")
        return (s, t)
```

Listing 1: Python example

## 7.2 Implementation of Meta-heuristic Algorithm: Ant Colony Optimization( ACO) by Fenet and Solnon

We have used python as the programming language to implement the meta-heuristic algorithm for max clique problem.

```

import logging
import copy
import time
import random
from tqdm import tqdm
import numpy as np
from multiprocessing import Pool

random.seed(0)
np.random.seed(0)

logging.basicConfig(level=logging.INFO, format='[%(asctime)s] - %(
    message)s', datefmt='%H:%M:%S')

class AntClique:
    def __init__(self, num_ants=7, taomin=0.01, taomax=4, alpha=2,
    rho=.995, max_cycles=3000):
        self.num_ants = num_ants
        self.taomin = taomin
        self.taomax = taomax
        self.alpha = alpha
        self.rho = rho
        self.max_cycles = max_cycles

        self.graph = None
        self.best_clique_info = None

    def initialize_pheromone_trails(self, graph):
        self.graph = copy.deepcopy(graph)
        self.best_clique_info = {
            'clique': set(),
            'req_time': -1,
            'req_cycles': -1,
        }

        # initialize all edges with taomax pheromone
        for n, nbrs in graph.items():
            for nbr, attrs in nbrs.items():
                attrs['pheromone'] = self.taomax
                self.graph[n][nbr] = attrs


    def construct_clique(self, ant_idx):
        clique = set()
        candidates = set()
```

```python
        neigbors = lambda node: set(self.graph[node].keys())
        pheromone_factor = lambda node: sum(self.graph[node][clique_node]['pheromone'] for clique_node in clique)

        initial_vertex = random.sample(self.graph.keys(), 1)[0]
        clique.add(initial_vertex)
        candidates.update(neigbors(initial_vertex))

        while candidates:
            pheromone_factors = [pheromone_factor(node) ** self.alpha for node in candidates]
            pheromone_probs = [factor / sum(pheromone_factors) for factor in pheromone_factors]

            selected_vertex = np.random.choice(list(candidates), size=1, p=pheromone_probs)[0]

            clique.add(selected_vertex)
            candidates = candidates.intersection(neigbors(selected_vertex))

        return (ant_idx, clique)

    def update_pheromone_trails(self, iteration_no, start_time, cliques):
        best_ant_idx, best_clique = max(cliques, key=lambda t: len(t[1]))

        # update global info
        if len(best_clique) > len(self.best_clique_info['clique']):
            self.best_clique_info = {
                'clique': best_clique,
                'req_time': (time.time() - start_time) * 1000,
                'req_cycles': iteration_no + 1
            }

        c_best = len(self.best_clique_info['clique'])
        c_k = len(best_clique)

        # evaporate pheromone on all edges
        for n, nbrs in self.graph.items():
            for nbr, attrs in nbrs.items():
                attrs['pheromone'] = max(self.taomin, self.rho * attrs['pheromone'])
                self.graph[n][nbr] = attrs

        # deposit pheromone for best ant
        for n in best_clique:
            for nbr in best_clique:
                if n == nbr:
                    continue

                attrs = self.graph[n][nbr]
                attrs['pheromone'] = min(self.taomax, (1 / (1 + c_best - c_k)) + attrs['pheromone'])
                self.graph[n][nbr] = attrs


    def run(self, graph, use_threading=False):
        start_time = time.time()

        self.initialize_pheromone_trails(graph)

        for iteration_no in tqdm(range(self.max_cycles), desc='Running ant-clique loops'):
            if use_threading:
                with Pool(self.num_ants) as pool:
                    formed_cliques = pool.map(self.construct_clique, range(self.num_ants))
            else:
                formed_cliques = [self.construct_clique(i) for i in range(self.num_ants)]

            self.update_pheromone_trails(iteration_no, start_time, formed_cliques)

        s = len(self.best_clique_info['clique'])
        t = self.best_clique_info['req_time']
        c = self.best_clique_info['req_cycles']

        logging.info(f"clique size: {s}, req cycles: {c}, req time(ms): {t:.3f}")
        return (s, t, c)
```
Listing 2: Python example

## 7.3 Results of the Implemented Algorithms

We will show the results by running the two implemented algorithms for 21 input graphs as we can see in Table 1.

Table 1: Test Graphs Characteristics

| Graph name | Nodes | Edges | Known Max Clique |
|---|---|---|---|
| anna | 138 | 986 | 11 |
| brock200_2 | 200 | 9876 | 12 |
| brock200_4 | 200 | 13089 | 17 |
| C125.9 | 125 | 6963 | 34 |
| hamming84 | 256 | 20864 | 16 |
| homer | 561 | 3258 | 13 |
| huck | 74 | 602 | 11 |
| keller4 | 171 | 9435 | 11 |
| le450_15b | 450 | 8169 | 15 |
| le450_15c | 450 | 16680 | 15 |
| le450_15d | 450 | 16750 | 15 |
| le450_25a | 450 | 8260 | 25 |
| le450_25c | 450 | 17343 | 25 |
| le450_25d | 450 | 17425 | 25 |
| le450_5a | 450 | 5714 | 5 |
| le450_5b | 450 | 5734 | 5 |
| le450_5c | 450 | 9803 | 5 |
| le450_5d | 450 | 9757 | 5 |
| miles250 | 128 | 774 | 8 |
| p_hat300-1 | 300 | 10933 | 8 |
| p_hat300-2 | 300 | 21928 | 25 |

As we can see from Table 2 for dense graphs the running time increases for the branch and bound algorithm because it searches for the optimal solution. For small and less dense graphs it takes less time to find maximum clique than the meta-heuristic algorithm Ant Colony Optimization. The Ant Colony Optimization always comes up with a solution even if its not optimal in the given time which is 20 minutes.

Table 2: **Running Times (in ms)**

| Graph name(best) | Ant-clique nbCy(sdv) | Ant-clique time(sdv) | Branch and bound time |
|---|---|---|---|
| anna (11) | 7.67(2.05) | 17.38(2.91) | **2.86** |
| brock200_2 (12) | 380.67(428.10)) | **5119.13**(5818.34) | 10194.34 |
| brock200_4 (17) | 214.00(115.34) | **4587.07**(2452.60) | 579099.92 |
| C125.9 (34) | 180.00(35.33) | **7312.11**(1569.49) | * |
| hamming8-4 (16) | 99.66(40.54) | **2972.39**(1211.32) | 898296.82 |
| homer (13) | 8.33(2.86) | 35.60(9.52) | **2.69** |
| huck (11) | 1.66(0.94) | 6.33(2.99) | **0.51** |
| keller4 (11) | 22.66(14.29) | **430.47**(268.46) | 158878.92 |
| le450_15b (15) | 1.00(0.00) | **46.25**(5.40) | 54.47 |
| le450_15c (15) | 1.00(0.00) | **88.35**(12.46) | 556.35 |
| le450_15d (15) | 1.00(0.00) | **97.60**(13.06) | 557.51 |
| le450_25a (25) | 1.00(0.00) | 54.30(7.13) | **41.02** |
| le450_25c (25) | 1.00(0.00) | **93.10**(4.86) | 466.93 |
| le450_25d (25) | 1.00(0.00) | **92.79**(10.94) | 494.23 |
| le450_5a (5) | 1.33(0.47) | **32.34**(3.23) | 41.55 |
| le450_5b (5) | 1.00(0.00) | **29.13**(3.77) | 41.11 |
| le450_5c (5) | 1.00(0.00) | **51.25**(9.27) | 145.15 |
| le450_5d (5) | 1.00(0.00) | **48.91**(1.72) | 140.18 |
| miles250 (8) | 4.66(2.49) | 10.19(3.34) | **0.78** |
| p_hat300-1 (8) | 81.00(38.28) | **1119.92**(493.61) | 1408.27 |
| p_hat300-2 (25) | 162.00(26.19) | **5640.97**(916.14) | * |

*** - Graph run did not finish within time limit (20 minutes)**

As we can see from Table 3 for dense graphs the branch and bound algorithm can not find the maximum clique because it searches for the optimal maximum clique. For small and less dense graphs it finds the optimal maximum clique. The Ant Colony Optimization always comes up with a solution of clique even if it is not optimal.

Table 3: Found Clique Sizes

| **Graph name(best)** | **Ant clique-avg(sdv)** | **Branch and bound** |
|---|---|---|
| anna (11) | **11.00(0.00)** | **11** |
| brock200_2 (12) | 11.33(0.47) | **12** |
| brock200_4 (17) | 16.00(0.00) | **17** |
| C125.9 (34) | **34.00(0.00)** | * |
| hamming8-4 (16) | **16.00(0.00)** | **16** |
| homer (13) | **13.00(0.00)** | **13** |
| huck (11) | **11.00(0.00)** | **11** |
| keller4 (11) | **11.00(0.00)** | **11** |
| le450_15b (15) | **15.00(0.00)** | **15** |
| le450_15c (15) | **15.00(0.00)** | **15** |
| le450_15d (15) | **15.00(0.00)** | **15** |
| le450_25a (25) | **25.00(0.00)** | **25** |
| le450_25c (25) | **25.00(0.00)** | **25** |
| le450_25d (25) | **25.00(0.00)** | **25** |
| le450_5a (5) | **5.00(0.00)** | **5** |
| le450_5b (5) | **5.00(0.00)** | **5** |
| le450_5c (5) | **5.00(0.00)** | **5** |
| le450_5d (5) | **5.00(0.00)** | **5** |
| miles250 (8) | **8.00(0.00)** | **8** |
| p_hat300-1 (8) | **8.00(0.00)** | **8** |
| p_hat300-2 (25) | **25.00(0.00)** | * |

*** - Graph run did not finish within time limit (20 minutes)**

## References

[1] Randy Carraghan and Panos M Pardalos. “An exact algorithm for the maximum clique problem”. In: *Operations Research Letters* 9.6 (1990), pp. 375–382.

[2] Roberto Battiti and Marco Protasi. “Reactive local search for the maximum clique problem 1”. In: *Algorithmica* 29.4 (2001), pp. 610–637.

[3] Patric RJ Östergård. “A fast algorithm for the maximum clique problem”. In: *Discrete Applied Mathematics* 120.1-3 (2002), pp. 197–207.

[4] Serge Fenet and Christine Solnon. “Searching for maximum cliques with ant colony optimization”. In: *Workshops on Applications of Evolutionary Computation*. Springer. 2003, pp. 236–245.

[5] Etsuji Tomita and Tomokazu Seki. “An efficient branch-and-bound algorithm for finding a maximum clique”. In: *International conference on discrete mathematics and theoretical computer science*. Springer. 2003, pp. 278–289.

[6] Uriel Feige. “Approximating maximum clique by removing subgraphs”. In: *SIAM Journal on Discrete Mathematics* 18.2 (2004), pp. 219–225.

[7] Andrea Grosso, Marco Locatelli, and Federico Della Croce. “Combining swaps and node weights in an adaptive greedy approach for the maximum clique problem”. In: *Journal of Heuristics* 10.2 (2004), pp. 135–152.

[8] Kengo Katayama, Akihiro Hamamoto, and Hiroyuki Narihisa. “Solving the maximum clique problem by k-opt local search”. In: *Proceedings of the 2004 ACM symposium on Applied computing*. 2004, pp. 1021–1025.

[9] Wayne Pullan. “Phased local search for the maximum clique problem”. In: *Journal of Combinatorial Optimization* 12.3 (2006), pp. 303–323.

[10] Wayne Pullan and Holger H Hoos. “Dynamic local search for the maximum clique problem”. In: *Journal of Artificial Intelligence Research* 25 (2006), pp. 159–185.

[11] Wayne Pullan, Franco Mascia, and Mauro Brunato. “Cooperating local search for the maximum clique problem”. In: *Journal of Heuristics* 17.2 (2011), pp. 181–199.

[12] Roger Ouch, Kristopher Reese, and Roman V Yampolskiy. “Hybrid Genetic Algorithm for the Maximum Clique Problem Combining Sharing and Migration.” In: *MAICS*. 2013, pp. 79–84.

[13] Bharath Pattabiraman et al. “Fast algorithms for the maximum clique problem on massive sparse graphs”. In: *International Workshop on Algorithms and Models for the Web-Graph*. Springer. 2013, pp. 156–169.

[14] Muhammad Fayaz et al. “Approximate Maximum Clique Algorithm (AMCA): A Clever Technique for Solving the Maximum Clique Problem through Near Optimal Algorithm for Minimum Vertex Cover Problem”. In: *International Journal of Control and Automation* 11.3 (2018), pp. 35–44.