An Efficient Binary Particle Swarm Optimization Algorithm for Effectively Solving the Max Clique Problem

PARITOSH CHANDRA RAY

FORKAN AHMED

SUSHANTO RAY PAPON

A thesis submitted for the degree of
Bachelor of Science in Computer Science and Engineering

Department of Computer Science and Engineering
Dhaka University of Engineering & Technology, Gazipur

June 2023

An Efficient Binary Particle Swarm Optimization Algorithm for Effectively Solving the Max Clique Problem

PARITOSH CHANDRA RAY

Student No: 174102

Reg: 10403, Session: 2020-2021

FORKAN AHMED

Student No: 174042

Reg: 10343, Session 2020-2021

SUSHANTO RAY PAPON

Student No: 174104

Reg: 10405 ,Session 2020-2021

Supervisor: **Dr. Md. Jakirul Islam**

Associate Professor

A thesis submitted for the degree of
BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

Department of Computer Science and Engineering
Dhaka University of Engineering & Technology, Gazipur

June 2023

Declaration

We hereby declare that, except as otherwise noted by reference, this thesis entitled "An Efficient Binary Particle Swarm Optimization Algorithm for Effectively Solving the Max Clique Problem" is our own work completed during the session 2020-2021 under the supervision of **Dr. Md. Jakirul Islam,** and that neither it nor the work it contains has been used to support the receipt of any other degree or credential from this university or any other academic institution. We looked over the university's current research ethics regulations and assume responsibility for carrying out the procedures in compliance with the Department of Computer Science and Engineering (CSE) at Dhaka University of Engineering and Technology (DUET), Gazipur.

Signature

…………………………………

(PARITOSH CHANDRA RAY)

…………………………………

(FORKAN AHMED)

…………………………………

(SUSHANTO RAY PAPON)

…………………………………

Dr. Md. Jakirul Islam
Associate Professor
Department of Computer Science and Engineering.
DUET, Gazipur

Acknowledgments

All praise and appreciation to Almighty God, who has blessed us with the ability to finish this thesis. We would like to express our heartfelt gratitude to our supervisor, **Dr. Md. Jakirul Islam, Associate Professor**, Department of Computer Science and Engineering at Dhaka University of Engineering & Technology, Gazipur, who is a genius in his own right. He convincingly instructed and consistently encouraged us to reach our desired aim. Continuous assistance, suggestions, valuable remarks, and patience throughout the thesis have been helpful to us. His guiding approach and passion for the subject left an indelible mark on us.

We would also like to extend our deepest thankful to **Dr. Momotaz Begum,** Professor, Department of Computer Science and Engineering, DUET, Gazipur, for checking and refined our paper. In addition to our supervisor, we would like to express our heartfelt gratitude to **Dr. Md. Obaidur Rahman, Head of the Department** of Computer Science and Engineering, DUET, Gazipur, for his assistance in completing our Bachelor of Science Degree and constructively managing our thesis.

Finally, and most significantly, we want to thank all of the faculty members in the Department of Computer Science and Engineering. Each member of this faculty has supplied us with considerable personal and professional guidance, as well as a wealth of knowledge about scientific research and life in general. Kindness and support have made our studies and lives more enjoyable.

Abstract

The Maximum Clique problem is a well-known and extensively studied NP-hard problem in graph theory, poses significant challenges due to its computational complexity. This problem has diverse applications in various domains, including social network analysis, bioinformatics, and telecommunication networks. In this research, we propose an effective Binary Particle Swarm Optimization (BPSO) algorithm to address the limitations of existing algorithms while solving the maximum clique problem. Our BPSO algorithm controls the binary nature of the problem by representing each potential clique as a binary string. For this modified BPSO, we propose a linearly decreasing function for the position updating equation to effectively explore the search space and converges towards optimal or near-optimal clique solutions. The purpose of this modification is to enhance the exploration and exploitation capabilities of the BPSO algorithm. To evaluate the performance of our proposed algorithm, some experiments are conducted on synthetic datasets consisting various number of cliques. The results demonstrate the effectiveness of our approach, as our algorithm consistently outperforms state-of-the-art algorithms in terms of both solution quality and runtime efficiency.

Contents

List of Figures

List of Algorithms

List of Tables

List of abbreviations

GA	Genetic Algorithm
PSO	Particle Swarm Optimization
ACO	Ant Colony Optimization
BPSO	Binary Particle Swarm Optimization
MBPSO	Modified Binary Particle Swarm Optimization

Chapter 1

Introduction

1.1 Motivation

The Max Clique problem is a fundamental and extensively studied problem in graph theory, known for its computational complexity and wide range of applications in various domains [6]. It involves finding the largest complete subgraph (clique) within a given graph, where every pair of vertices is connected by an edge. Let G = (V, E) be an undirected graph, with V representing the set of vertices and E representing the set of edges. The graph may be represented as an adjacency matrix A, where A[i][j] = 1 if an edge exists between vertices i and j, and A[i][j] = 0 otherwise. Let C be a subset of vertices, i.e., $C \subseteq V$. C is a clique if and only if for every pair of vertices $u, v \in C$, there exists an edge between them, i.e., A[u][v] = 1.The goal is to find the maximum clique, which is the largest possible subset $C^* \subseteq V$ that is a clique.

Mathematically, the max clique problem can be expressed as:

$$\text{maximize } |C^*| \tag{1.1}$$

$$\text{subject to: } C^* \text{ is a clique} \tag{1.2}$$

$$C^* \subseteq V$$

Here, $|C^*|$ represents the cardinality (size) of the clique C^*.

The graph G in Fig. 1.1 has 7 vertices, 11 edges, and the largest clique is shown by the vertices v2, v4, v5, and v6. The problem is NP-hard, making it challenging to find exact solutions in a reasonable amount of time.

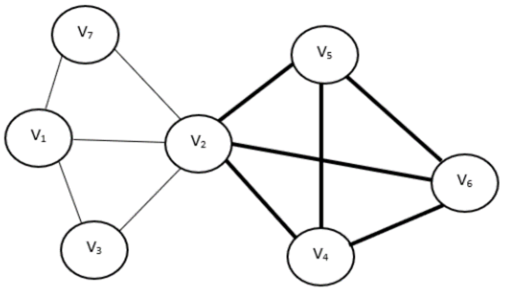

Figure 1.1. Illustration of a max clique in a graph.

In literature, there are two different methods for solving the Max Clique problem. The first is an exact approach that provides an exact solution, while the second is a meta-heuristic method that, when the problem size is large, provides a solution in the appropriate amount of time. Meta-heuristic methods are mostly used in cases using an exact method has a very high computational cost, there is no proper exact method to solve the problem. Meta-heuristic methods may produce a solution, but it is not always the optimal one.

In the last decades, to solve the max clique problem, a number of heuristics and meta-heuristic techniques have been developed, including genetic algorithms [2], Tabu search [13], local search [7], Variable neighbor-hood search [14], K-opt heuristic [15], and hyper heuristics [6]. This is an NP-Hard problem and therefore in meta-heuristic methods there are some difficulties with Genetic Algorithm (GA) for complex implementation. Particle Swarm Optimization (PSO) does not work well for discrete optimization problems since it requires conversion to convert them to continuous optimization problems, which slows down the convergence rate on some optimization problems [3]. We need an effective algorithm to solve this problem. One such algorithm is Binary Particle Swarm Optimization (BPSO), which is inspired by the collective behavior of bird flocking or fish schooling. BPSO has demonstrated promising results in solving binary optimization problems by leveraging the principles of swarm intelligence.

The binary particle swarm optimization is an extension of the PSO method which is mainly developed to handle discrete optimization problems. BPSO is designed to find global optimal solution rather than getting stuck in local optimal solution [1]. It can quickly converge to good solutions, which is particularly important for large graphs where other algorithms may be impractical. BPSO is a flexible algorithm that can be adapted to handle different types of constraints and objective functions and it can be parallelized to handle large search spaces. Despite these benefits, as far our knowledge, the BPSO did not used for the Max Clique problem. Therefore, this study intend to effectively find Max Clique using the BPSO method. However, the original BPSO has the difficulties to provide good quality solutions due to its position updating equation. To overcome these difficulties, we established the following objectives, which are detailed in the following section.

1.2 Research Objectives

The specific objectives of this paper are as follows:

a) To design the Max Clique problem as an optimization problem.

b) To develop a Modified Binary Particle Swarm Optimization (MBPSO) algorithm to find the solution to the Max Clique problem effectively.

c) To assess the performance of the proposed and well-known meta-heuristics approaches on Max Clique instances.

1.3 Contribution

The contributions of this research are as follows:

a) Presentation of a new mathematical model addressing the Max Clique problem.

b) A modified binary Particle Swarm Optimization (PSO) approach can be designed to efficiently find solutions for the Max Clique problem while balancing exploration and exploitation capabilities.

c) Evaluate the proposal over a set of Max Clique problem instances.

1.4 Thesis Outline

The remaining part of this thesis is structured as follows. Chapter 2 describes the background of the thesis. Chapter 3 describes the proposed methodology. Chapter 4 describes the experimental result of this thesis. Chapter 5 contains the conclusion.

Chapter 2

Literature Review

In this chapter, various topics will be discussed like basic combinatorial optimization, the NP-hard problem: Backtracking Approach, Local Search, Genetic Algorithm, Ant Colony Optimization, Particle Swarm Optimization, Binary Particle Swarm Optimization Algorithm, and more methods have been discussed for solving the max clique problem.

2.1 Combinatorial Optimization

Hard combinatorial optimization problems (NP-hard, NP-complete) involve huge discrete search spaces. Combinatorial Optimization is a type of problem in which the solution must optimize a function over a set of discrete objects. Thus we say that the combinatorial optimization is a technique for improving an algorithm by constructing mathematical approaches that limit the number of viable answers while also speeding up the search procedure. A combinatorial optimization problem finds an optimal solution from a large number of possibilities. In order to find a solution, many optimization problems contain an objective function as well as logical requirements and constraints.

2.2 NP-hard Problem

An NP-hard problem is a computational complexity theory problem that is as challenging as or more challenging than any problem in NP. It is classified as NP-hard if every problem in NP can be transformed into it within polynomial time. The max clique problem, often known as the NP-Hard problem, is a constraint satisfaction problem (CSP).

A constraints satisfaction problem consists of the following components:

- A set of variables like $x_0, x_i \ldots\ldots x_{n-1}$
- A domain $D_i = \{d_0, d_1 \ldots d_{m-1}\}$.
- For each pair of variables (i, j), with $0 \leq i < j < n$, a subset constraint $C_{i,j} = D_i \times D_j$ is defined, where $D_i \times D_j$ represents the Cartesian product of D_i and

D_j. These subset constraints are applicable when they deviate from the standard Cartesian product.

For clustering problem that is used to separate decision problems which is exists in the class A, that requires for a decision either "yes" or "no" response. The NP-Hard problem class that emphasis on the problem with the condition where each has a "yes" response and proof that this answer can be verified in polynomial time techniques, In the reasonable time this type of decision problem cannot be resolved only for its higher complexity. Because its complexity (O (n!), $O(2^n)$, and $O(n^n)$) .

2.3 The Max Clique Problem

The Max Clique problem, a widely recognized combinatorial optimization problem in graph theory, revolves around an undirected graph G = (V, E). Here, V denotes the vertex set, and E represents the edge set. The objective is to identify the largest complete subgraph (clique) present in G. A clique is defined as a subset of vertices in which every pair of vertices is connected by an edge. In formal terms, a clique $C \subseteq V$ is a collection of vertices where for every pair of vertices $u, v \in C$, there exists an edge $(u, v) \in E$. The goal of the max clique problem is to identify a clique C that possesses the highest number of vertices. Consider the below examples of problem instances, feasible solutions, and optimal solutions for the maximum clique problem:

Problem instance: The graph with vertices {1, 2, 3, 4, 5} and edges {(1, 2), (1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5), (3, 4), (3, 5), (4, 5)}.

Feasible solutions: {1, 2}, {1, 3}, {1, 4}, {1, 5}, {2, 3}, {2, 4}, {2, 5}, {3, 4}, {3, 5}, {4, 5}, {1, 2, 3}, {1, 2, 4}, {1, 2, 5}, {1, 3, 4}, {1, 3, 5}, {1, 4, 5}, {2, 3, 4}, {2, 3, 5}, {2, 4, 5}, {3, 4, 5}, {1, 2, 3, 4}, {1, 2, 3, 5}, {1, 2, 4, 5}, {1, 3, 4, 5}, {2, 3, 4, 5}, {1, 2, 3, 4, 5}.

Optimal solution: {1, 2, 3, 4, 5}.

2.4 Approaches to solve the Max Clique problem

Two types of approach available to solve the max clique problem [6]. The first one is the exact method which gives exact solution and the second one is the meta-heuristic method which gives a solution within the desired amount of time when the problem size is large. Meta-heuristic methods are mostly used in cases whether using an exact method has a very high computational cost or there is no proper exact method to solve the problem. Meta-heuristic methods can lead to a solution but it is not always optimal. In the last decades, several heuristic and meta-heuristic methods have been developed for max clique problem namely, genetic algorithms, Ant colony optimization, BPSO which are described in the following sections.

2.4.1 Backtracking Approach

The max clique problem is a widely recognized computational challenge in the field of computer science. Its objective is to locate the largest complete sub-graph within an undirected graph [8]. The backtracking algorithm for solving the max clique problem is a brute-force algorithm, which means that it tries all possible combinations of vertices. This can be very time-consuming for large graphs. However, the algorithm is guaranteed to find the maximum clique, even if the graph is very large.

The algorithm starts with the empty set as the clique. Then, it recursively adds vertices to the clique, one at a time. In each iteration, the algorithm verifies whether the newly added vertex is connected to all vertices within the clique. If the newly added vertex is not connected to all existing vertices in the clique, the algorithm backtracks its step and attempts to add a different vertex instead. The algorithm continues recursively adding vertices until no more vertices can be added. The clique at the end of the recursion is the max clique.

Steps of Backtracking Approach:

1. Start with an empty set S.
2. Select a vertex v in the graph and add it to S.
3. Find all vertices in the graph that are adjacent to v, and add them to a list called Adj_v.
4. Remove all vertices in S that are not in Adj_v. This is because if a vertex is not adjacent to v, it cannot be part of the maximum clique.
5. If S is now empty, return 1. This means that we have found a maximal clique of size 1.
6. If S is not empty, recursively apply the previous steps to S. If this returns a clique of size p, then return p+1, since we have found a clique of size p+1 that contains v.

7. Remove v from S, and repeat the previous steps for the remaining vertices in the graph.
8. Return the largest clique found.

Algorithm 2.1 Pseudo code of Backtracking Approach

```
function find_max_clique(graph G):
    max_size = 0
    for each vertex v in G:
        S = {v}
        max_size = max(max_size, backtrack(S, G))
    return max_size
function backtrack(S, G):
    if S is empty:
        return 1
    max_size = 0
    for each vertex v in G:
        if v not in S:
            Adj_v = {u in G : (u, v) is an edge}
            S_new = S.intersection(Adj_v)
            if not S_new:
                continue
            size = backtrack(S_new, G)
            max_size = max(max_size, size+1)
    return max_size
```

Problem with Backtracking:

- It does not work efficiently for solving strategy problem
- More comparisons are needed.
- When problem size is large it takes lot of memory.
- It takes more time thus performance speed very slow.

2.4.2 Local search

Local search refers to a meta-heuristic optimization technique that operates by iteratively enhancing a candidate solution through the exploration of its nearby neighborhood. This approach focuses on making incremental improvements within the local region of the solution space. In the context of the max clique problem, local search can be used to refine a given candidate subset of vertices that forms a clique [16].Here is a simple outline of how local search can be applied to the max clique problem:

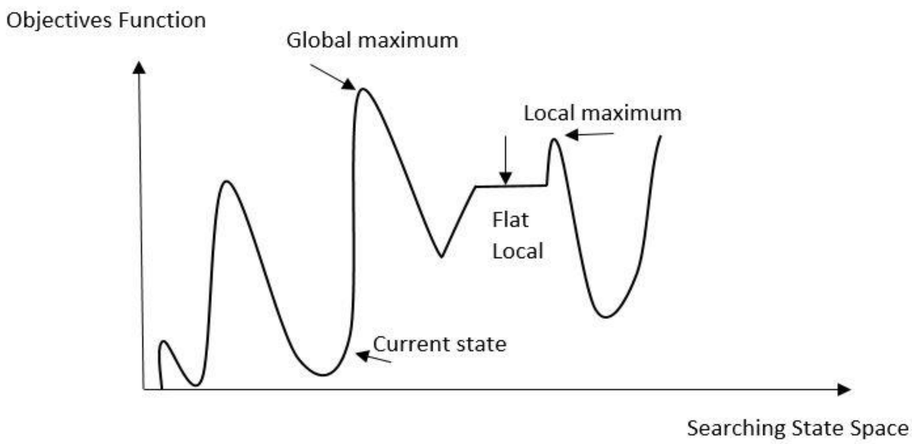

Figure 2.3: Local search algorithm view.

1. **Initialization:** Create an initial candidate solution, such as a random subset of vertices or a heuristic solution.
2. **Neighborhood:** Define a neighborhood function that generates neighboring solutions by applying local modifications to the current solution. In the context of the max clique problem, a frequently used neighborhood function is the vertex removal operator. This operator selectively eliminates vertices from a given subset if their inclusion violates the clique condition. Conversely, it incorporates neighboring vertices into the subset if they adhere to the condition, thus improving the potential for a larger clique.
3. **Local search:** Apply a local search strategy to improve the present solution repeatedly through exploring its surroundings and selecting the best neighboring solution. The hill climbing algorithm is a common local search technique that finds the best neighboring solution that improves the objective function and repeats the process until a local optimum is obtained.
4. **Stopping criterion:** When a stopping requirement has been met, such as reaching a maximum number of iterations, finding a clique of maximum size, or hitting a time limit, the local search process is terminated.

2.4.3 Genetic Algorithm

Genetic algorithms are based on genetics principles and Darwin's theory of evolution, both of which involve the process of natural selection of biological systems [2]. The Genetic Algorithm operates through the following process:

1. Initialize Population
2. Loop
    a. Evaluation
    b. Selection
    c. Reproduction
    d. Crossover
    e. Mutation
3. Convergence

The genetic algorithm's optimization process begins with a set of separately generated at random populations known as chromosomes. After a few generations, the goal is to have a set that has appropriate chromosomes. A fitness function is used to measure the quality of a chromosome. In order to produce new children each generation, we must use genetic crossover and mutation operators. Subsequently, the mutation operator introduces random changes to the genes of a chromosome. Mutation is important in the genetic algorithm because it maintains population diversity and explains new genes for future generations to inherit [5]. In conclusion, the next generation is created by choosing the most suitable chromosomes and discarding the rest, ensuring a consistent population size. The genetic algorithm incrementally converges regarding the best chromosome, indicating an optimal or nearly optimal solution to the problem, over multiple iterations [2]. The following steps illustrate the functioning of a genetic algorithm [5].

Steps of Genetic Algorithm:

Step-1: Initialization

- Create an initial set of chromosomes through a random selection process.

Step-2: Evaluation

- Using a fitness function, evaluate the fitness of each chromosome in the population.

Step-3: Reproduction

- Based on their fitness, select parent chromosomes from the current generation.
- Create new offspring through crossover and mutation operations.

Step-4: Replacement

- To create the new population for the following generation, opt for individuals from the parent population and offspring population.
- Apply selection strategies like tournament, Roulette Wheel selection.

Step-5: Termination

- Verify whether the termination requirements such as the maximum number of generations or the desired fitness level have been met.
- If the termination criteria are satisfied, stop the algorithm; otherwise, go back to Step 2.

Genetic Algorithms (GA) have shown promising results in many different kinds of combinatorial problems involving optimization, including the max clique problem [2]. However, there are several key limitations to consider when utilizing GA:

- **Representation:** The initial and crucial step in implementing a genetic algorithm is defining an appropriate representation for the problem at hand. Choosing an effective representation can significantly impact the algorithm's performance and success.
- **Fitness Function Coding:** Developing an accurate fitness function can be challenging. The fitness function determines how well a solution performs and guides the GA towards optimal or near-optimal solutions. Designing and coding a fitness function that properly captures the problem's objectives can be a complex task.
- **Population Size:** Using a small population size restricts the solution space available for the Genetic Algorithm. Inadequate exploration of the solution space may hinder the algorithm's ability to converge towards optimal solutions. A sufficiently large population size is typically required to achieve accurate and reliable results.
- **Analytical Problems:** Genetic Algorithms are not well-suited for analytical problems that can be effectively solved using mathematical or analytical

methods. GA excels in tackling optimization problems with complex and non-linear search spaces where traditional methods may struggle.

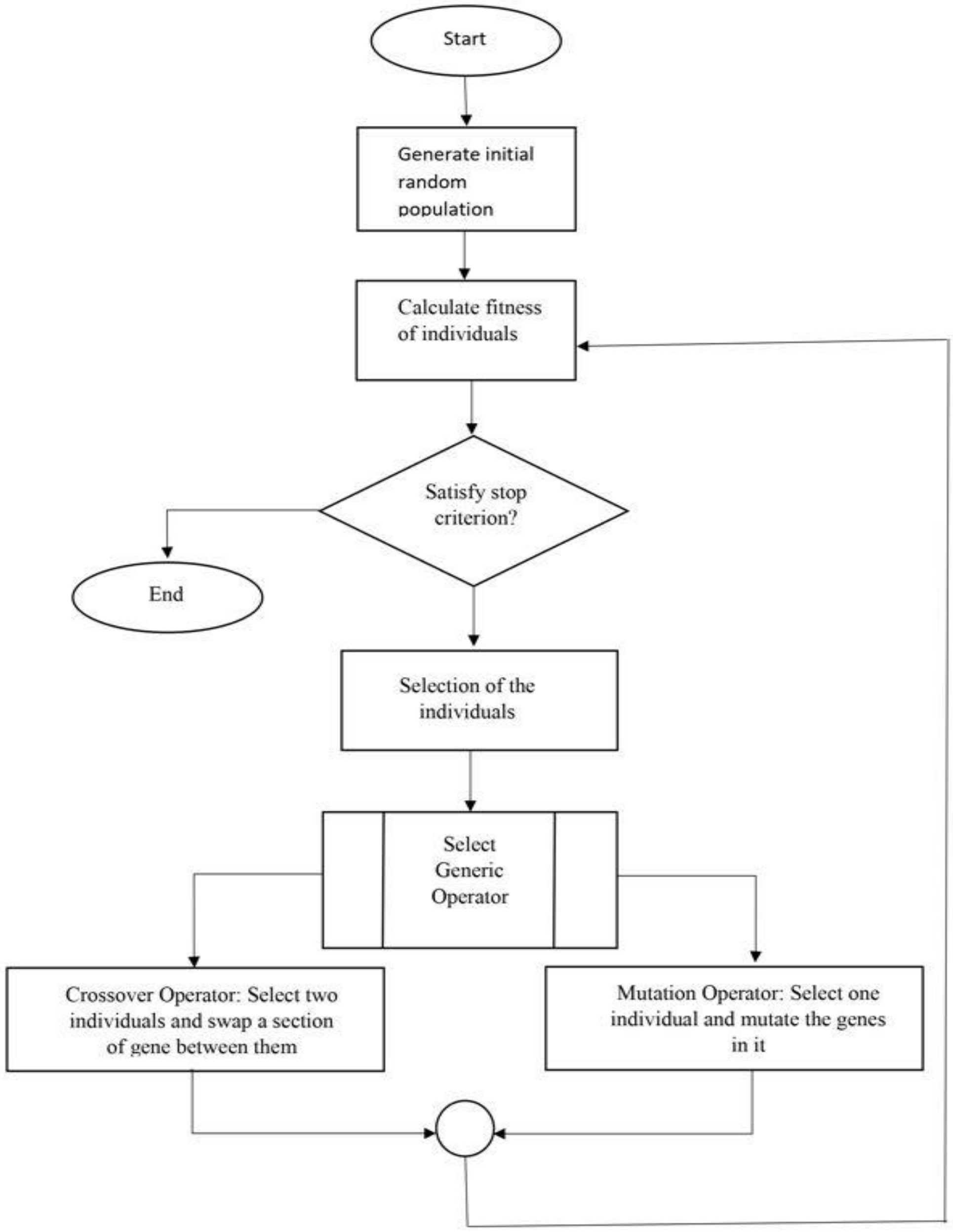

Figure 2.2: The Genetic Algorithm flow chart.

2.4.4 Ant Colony Optimization (ACO)

Ant Colony Optimization (ACO) is a metaheuristic algorithm based on ant habits of foraging. It was suggested in the early 1990s by Marco Dorigo and has since gained significant interest due to its usefulness in solving various combinatorial optimization problems, including the Max Clique problem. ACO algorithms simulate the collective behavior of ants as they search for food, applying pheromone-based communication and local heuristics to guide their search [3].

The following steps illustrate the functioning of ACO algorithm to solve the Max Clique problem:

Step-1: Representation

- The Max Clique problem can be represented as an undirected graph, where vertices represent nodes and edges represent connections between nodes.

Step-2: Initialization

- Initialize a colony of ants, where each ant represents a potential clique solution.
- Assign each ant to a random vertex in the graph as its starting point.

Step-3: Pheromone Initialization

- Associate a pheromone value with each edge in the graph to represent the desirability of that edge for constructing cliques.
- Initialize the pheromone values to a small positive constant.

Step-4: Ant Movement and Construction

- Each ant moves from its current vertex to a neighboring vertex based on a probabilistic rule.
- The probability of selecting a particular neighbor is determined by a combination of the pheromone level on the edge connecting the current vertex and a local heuristic value.
- The local heuristic encourages ants to choose vertices that are more likely to lead to larger cliques.

Step-5: Clique Construction

- As an ant moves to a new vertex, it constructs a partial clique by including the current vertex and its adjacent vertices that are already part of the partial clique.
- The ant continues to build the partial clique by selecting neighbors that form a clique with the existing vertices.

Step-6: Pheromone Update

- After all ants have completed their movement and construction phase, update the pheromone values on the edges.
- The amount of pheromone deposited is proportional to the quality (size) of the cliques constructed by the ants.
- Evaporate a certain fraction of the pheromone on all edges to avoid convergence to suboptimal solutions.

Step-7: Termination

- Repeat steps 4 to 6 for a specified number of iterations or until a termination criterion is met (e.g., a maximum number of iterations without improvement).

Step-8: Solution Extraction

- After termination, select the best clique found by any of the ants as the final solution.

ACO for the Max Clique problem leverages the pheromone trails to guide the search towards high-quality cliques. Over time, the pheromone trails will be reinforced on edges that are part of large cliques, allowing the algorithm to converge towards better solutions. The local heuristic helps ants explore promising areas of the search space.

Pheromone Trails and Heuristic Information

Pheromone trails play a crucial role in ACO by guiding the ants towards better solutions. Ants typically deposit a quantity of pheromone that is directly proportional to the quality of the solution they construct [3]. Conversely, pheromone evaporation ensures that less desirable paths gradually lose their influence over time. Heuristic information, such as the distance between nodes or problem-specific knowledge, is often incorporated into the decision-making process to balance exploration and exploitation.

Algorithm 2.2 Pseudo code of ACO Approach

---

1. Set initial values for variables like the number of ants, iterations, pheromone evaporation rate etc.
2. Initialize pheromone trails on all edges to a small positive value.
3. **Repeat** until termination condition is met:

4. **For** each ant:
5. Choose a starting node randomly.
6. **Repeat** until a solution is constructed:
7. Select the following node probabilistically based on the pheromone and heuristic values of the edges.
8. Move to the selected node and update the visited nodes and edges.
9. End repeat
10. Evaluate the solution and update the best solution if needed.
11. End for
12. **For** each edge:
13. Update the pheromone trail by reducing it by a factor and adding the contribution of the ants that used it.
14. End for
15. End repeat
16. **Return** the best solution found.

Parameters and Tuning

ACO algorithms require careful parameter tuning to achieve good performance. The parameters include the pheromone update rate, the influence of pheromone and heuristic information on ant decisions, and the number of ants and iterations. Proper parameter setting can significantly impact the convergence speed and solution quality.

ACO has demonstrated its efficacy in solving diverse combinatorial optimization problems, such as the well-known max clique problem [4]. One key advantage of ACO is its inherent parallelism. Since each ant constructs its solution independently, ACO algorithms can be easily parallelized, allowing for efficient computation on parallel and distributed systems. This feature makes ACO suitable for solving problems with large solution spaces that require substantial computational resources.

Despite its strengths, ACO does have some limitations. The algorithm's performance heavily depends on parameter settings, and finding the optimal parameter values can be challenging. Additionally, ACO may struggle with problems that have complex constraints or nonlinear objective functions.

2.4.5 Particle Swarm optimization (PSO)

Kennedy and Eberhart proposed the Particle Swarm Optimization (PSO) algorithm in 1995 as a technique for optimizing ongoing problems [9]. PSO is classified as an evolutionary algorithm that represents individuals in the search process by using a population of particles. It is inspired by observed collective behavior in nature, such as fish schooling and bird flocking. PSO has gained widespread acceptance for solving a variety of optimization problems. Particles in the PSO algorithm explore a complex search space by changing their positions dynamically until they either get a relatively stable position or exceed mathematical limitations [10]. In each iteration, the velocity and position of each particle are updated by considering the previous best position, which is determined through the exchange of information among neighboring particles. The following equations are employed to adjust the velocity and position:

$$v_{i,j}(t+1) = wv_{i,j}(t) + c_1 r_1 \left(p_{best,i,j} - x_{i,j}(t)\right) + c_2 r_2 \left(g_{best,i,j} - x_{i,j}(t)\right) \quad (2.1)$$

$$x_{i,j}(t+1) = x_{i,j}(t) + v_{i,j}(t+1) \quad (2.2)$$

Where i represents the index of a particle in the swarm and ranges from 1 to n, and j represents the index of a position within a particle and ranges from 1 to m. The variable t represents the number of iterations ranges from 1 to maximum iteration. The term $v_{i,j}(t)$ represents the i-th particle's velocity at iteration t, while $x_{i,j}(t)$ represents the i-th particle's position at iteration t. r_1 and r_2 are uniformly distributed random numbers ranging from 0 to 1. The acceleration coefficients c_1 and c_2 are used in the update equations. Finally, w is a positive inertia weight that influences how the particle's current velocity affects the update. The PSO algorithm is outlined below [11]:

PSO algorithm steps:

1. **Initialize the swarm.**

The swarm is initialized by randomly generating a set of particles. Each particle is represented as a binary vector, where each element indicates whether the corresponding vertex is included or excluded from the clique solution.

2. **Initialize the personal best and global best positions.**

A particle's personal best position reflects the best solution that the particle observed during the optimization process. In contrast, the global best position denotes the best solution found by any particle within the entire swarm. Both the personal and global best positions are initially set to the generated at random position of the first particle.

3. **Update the velocities and positions of the particles.**

The velocity and position of each particle are updated based on the following formulas:

$$v_{i,j}(t+1) = wv_{i,j}(t) + c_1r_1\left(p_{best,i,j} - x_{i,j}(t)\right) + c_2r_2\left(g_{best,i,j} - x_{i,j}(t)\right) \quad (2.3)$$

$$x_{i,j}(t+1) = x_{i,j}(t) + v_{i,j}(t+1) \quad (2.4)$$

4. **Update the personal best and global best positions.**

A particle's personal best position is updated if the fitness of its current position exceeds the fitness of its personal best position. Similarly, if the fitness of the current position exceeds the fitness of the global best position, the global best position is updated.

5. **Iterate until the stopping criterion is met.**

The algorithm continues to iterate until a specified stopping criterion is satisfied. This criterion can be defined as reaching a maximum number of iterations, achieving a minimum change in the fitness function, or attaining a specific level of accuracy.

6. **Return the global best solution.**

The global best solution is the solution with the highest fitness. The global best solution is returned by the algorithm.

Algorithm-2.3: Pseudo code of PSO Approach:

Require: *max_iterations*	The maximum number of iterations
Require: *swarm_size*	The population size

```
procedure pso_max_clique(graph):
    // Initialize the swarm.
    particles = []
    for i in range(swarm_size):
        particle = []
        for j in range(graph.num_vertices):
            particle.append(random.randint(0, 1))
        particles.append(particle)
    // Initialize the personal best and global best positions.
    pbest = particles[:]
    gbest = particles[0]
    // Iterate until the stopping criterion is met.
    for t in range(max_iterations):
        // Update the velocities and positions of the particles.
        for i in range(swarm_size):
            velocity = w * velocity + c1 * r1 * (pbest[i] - particles[i]) + c2 *
            r2 * (gbest - particles[i])
            particles[i] += velocity
        // Update the personal best and global best positions.
        for i in range(swarm_size):
            if fitness(particles[i]) > fitness(pbest[i]):
                pbest[i] = particles[i]
            if fitness(particles[i]) > fitness(gbest):
                gbest = particles[i]
    // Return the global best solution.
    return gbest
end procedure.
```

2.4.6 The Binary Particle Swarm Optimization (BPSO)

Kennedy and Eberhart developed a binary PSO algorithm in 1995 as an optimization technique for permitting the PSO algorithm to operate in binary problem space [11]. The particle position in the BPSO is not a real value, but rather either 0 or 1. In BPSO, this method was used to calculate velocity. The search space of BPSO is regarded as a hypercube for moving a particle, so that a particle can be seen to move to near and far corners of the hypercube by flipping various numbers of bits.

For BPSO, the velocity update equation remains unchanged which is as follows:

$$v_{i,j}(t+1) = wv_{i,j}(t) + c_1r_1\left(p_{best,i,j} - x_{i,j}(t)\right) + c_2r_2\left(g_{best,i,j} - x_{i,j}(t)\right) \quad (2.5)$$

Where, the variable *i* denotes the index of a particle in the swarm, where *i* ranges from 1 to n, *j* represents the index of a position within a particle, ranging from 1 to m. the variable *t* denotes the iteration number ranges from 1 to maximum iteration. The term $v_{i,j}(t)$ represents the velocity of the i-th particle at iteration *t*. The variables r_1 and r_2 are uniformly distributed random numbers between 0 and 1. The coefficients c_1 and c_2 are acceleration coefficients used in the update equations. p_{best}, $g_{best,i,j}$ denotes personal best and global best. Lastly, the parameter *w* is a positive inertia weight that influences the impact of the particle's current velocity on the update. But for updating its position in redefined by the rules:

$$x_{i,j}(t+1) = \begin{cases} 0 \ \ if\ rand() \geq S(v_{i,j}(t+1)) \\ 1 \ \ if\ rand() < S(v_{i,j}(t+1)) \end{cases} \quad (2.6)$$

Where, $x_{i,j}(t)$ represents the position of the i-th particle at iteration *t*, $S(v_{i,j}(t))$ is the sigmoid function that is used to transform the velocity as the following expression:

$$S\left(v_{i,j}(t+1)\right) = \frac{1}{1+e^{-v_{i,j}(t+1)}} \quad (2.7)$$

and rand () is the pseudo random number selected from a uniform distribution over [0.1, 1.0].

BPSO is known for its ability to quickly converge to good solutions, making it particularly suitable for solving problems involving large graphs where other algorithms may be impractical. It offers flexibility in handling various types of constraints and objective functions, and its parallelizability allows for efficient exploration of large search spaces. Despite these advantageous features, to the best of our knowledge, BPSO has not been extensively applied to the Max Clique problem. Hence, the objective of this study is to effectively utilize the BPSO method to find the Maximum Clique. However, the original BPSO algorithm faces challenges in providing high-quality solutions due to its position updating equation. To overcome these difficulties and achieve improved performance, we introduce a modified BPSO algorithm in the following chapter.

2.5 Research Gaps and Limitations

Despite the extensive research on metaheuristic algorithms like GA, PSO, and ACO for solving the Max Clique problem, there are still research gaps and limitations that need to be addressed. Two major research gaps identified in this study which are as follows:

- The effective representation of solutions.
- The need for balancing exploration and exploitation in the search space.

To bridge these gaps, this research introduces a modified BPSO algorithm specifically designed for solving the Max Clique problem. The modified BPSO algorithm, which will be detailed in the following chapter, aims to overcome the limitations of existing algorithms and provide more effective solutions. By addressing these research gaps, this research contributes to the advancement of optimization techniques for the Max Clique problem.

Chapter 3

Methodology

This chapter provides the detail methodology of the proposed research. Specifically the proposed model for the Max Clique problem and the BPSO for locating solution to this problem.

3.1 Problem Statement

The max clique problem is a decision problem in computer science and fundamental NP complete problem. This problem consists of an undirected graph G = (V, E), where V is the set of vertices and E is the set of edges. The objective is to find a subset of vertices C ⊆ V such that every vertex in C is connected to every other vertex in C, and |C| is as large as possible. In this research, we want to find the solution of max clique problem using BPSO. To do so, it is necessary to convert the max clique problem form decision making problem to the binary optimization problem, where each vertex is represented by a binary variable x_i. If $x_i = 1$, then vertex i is selected in the clique C, and if $x_i = 0$, then vertex i is not selected in the clique C. Mathematically, we represent the max clique problem as an optimization problem by the following equation [6]:

$$\text{Maximize} \qquad \sum_{i=1}^{n} x_i \tag{3.1}$$

Subject to $\quad x_i + x_j \leq 1$, for all {i, j} not belonging to the set of edges E. (3.2)

$x_i \in \{0, 1\}$, for i = 1,…....., n.

To apply the MBPSO to the max clique problem, the binary string *x = (x_1, x_2,………,x_n) was chosen from {0, 1}.

3.2 The Modified BPSO

This research aims to use the BPSO method to find the solution effectively for the max clique problem. The BPSO described above, this method takes long time to provide solution. To get the solution effectively, we need to modify the original BPSO so that it can provide solution effectively. In this thesis, for getting the max clique problem solution effectively, we have modified the position updating equation of BPSO to handle the max clique problem. The detail of the modified BPSO will provided in the following sections.

3.2.1 Solution Representation

To solve the max clique problem, the BPSO will use the particles where each of the particles consists of a velocity vector (v_i), position vector (x_i), and a personal best position (p_{best}). The position x_i of the i-th particle represents the current solution, while the personal best position (p_{best}) of the i-th particle represents the best solution that the particle has found so far. For the convenience, this study represents the solution vector using binary string. For example, consider the vertex V, edge E the corresponding solutions that have been represented by

$$V = \{v_1, v_2, v_3, v_4, v_5\}$$

$$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9\}$$

$$X_1 = \{0, 1, 1, 0, 1\}$$

$$X_2 = \{1, 0, 0, 1, 1\}$$

$$X_3 = \{0, 1, 1, 0, 0\}$$

$$X_4 = \{1, 1, 1, 0, 0\}$$

………………………………..

………………………………..

………………………………..

………………………………..

$$X_n = \{0, 0, 1, 1, 1\}$$

Each particle contains a solution. Suppose the targeted max clique contain vertex is v_1, v_4, v_5 then the solution is X_2.The solution can be represented by a one dimensional array which is getting after performing permutation from all combination, as shown in Fig. 3.1.

1	0	0	1	1

Figure 3.1: Solution Representation in one dimensional array.

In the above Fig. 3.1, the element, say 0 represent an indicator that the corresponding vertex has not been included in the solution and the element, say 1 represent an another indicator that the corresponding vertex has been included in the solution.

3.2.2 Fitness Function

In our proposed model for the Max Clique problem, the fitness value of each particle is calculated using Algorithm 2. The algorithm operates on a clique list and a 1D array called Position. When the value of a Position element is equal to 1, it is added to the clique list. By examining the elements in the clique list, if it satisfies the conditions for a clique, the algorithm returns the clique itself. Otherwise, if the conditions are not met, the algorithm returns 0, indicating that the particle contains an infeasible solution to the Max Clique problem.

Algorithm-3.1: Pseudo-code for the fitness function

```
for (int i = 0; i < n; i++)
    if (Position[i] == 1)
        clique.Add(i);
    bool isClique = true;
    for (int i = 0; i < clique.Count; i++)
        for (int j = i + 1; j < clique.Count; j++)
            if (adjacencyMatrix[clique[i], clique[j]] == 0)
                isClique = false;
                break;
        if (!isClique)
                break;
  Fitness = isClique ? clique.Count : 0;
Return Fitness
```

3.2.3 Velocity Updating Equation

The modified BPSO updates the velocity of i-th particle t-th iteration using the following equation:

$$v_{i,j}(t+1) = wv_{i,j}(t) + c_1r_1\left(p_{best,i,j} - x_{i,j}(t)\right) + c_2r_2\left(g_{best,i,j} - x_{i,j}(t)\right) \quad (3.3)$$

Where, the variable *i* denotes the index of a particle in the swarm, where *i* ranges from 1 to *n*, *j* represents the index of a position within a particle, ranging from 1 to *m*. the variable *t* denotes the iteration number ranges from 1 to maximum iteration. The term $v_{i,j}(t)$ represents the velocity of the i-th particle at iteration *t*. The variables r_1 and r_2 are uniformly distributed random numbers between 0 and 1. The coefficients c_1 and c_2 are acceleration coefficients used in the update equations. p_{best}, $g_{best,i,j}$ denotes personal best and global best. Lastly, the parameter *w* is a positive inertia weight that influences the impact of the particle's current velocity on the update.

3.2.4 Modified Position Updating Equation

The modified BPSO updates the position of *i-th* particle at *t-th* iteration using the following equations. The first equation is modified sigmoid function $S\left(v_{i,j}(t)\right)$ which is as follows:

$$S\left(v_{i,j}(t+1)\right) = \frac{1}{1+e^{-v_{i,j}(t+1)}} \quad (3.4)$$

This sigmoid function is used to update the particle position in the following way:

$$x_{i,j}(t+1) = \begin{cases} 0 \;\; if\; rand() \geq S\left(v_{i,j}(t+1)\right) * D_f \\ 1 \;\; if\; rand() < S\left(v_{i,j}(t+1)\right) * D_f \end{cases} \quad (3.5)$$

Where, $x_{i,j}(t+1)$ represents the position of the i-th particle at iteration *t*+1, D_f is a linearly decreasing function which provides initially a high probability to comprehensively explore the search space and provides a low probability in the last stages to exploit the search space. As a result, the modified BPSO can offer the good quality solution in at the end of the run. The value of D_f can be calculated using the following equation [1]:

$$D_f = D_f(max) - Itr_t * \left(\frac{D_f(max) - D_f(min)}{Itr_{max}}\right) \quad (3.6)$$

Where, $D_f(max)$ and $D_f(min)$ are the bounds on the control parameter = D_f , Itr_{max} is the maximum number of iterations, and Itr_t is the current iteration, where t = 0, 1, 2… Itr_{max} -1.

3.2.5 Updating Global best Solution

In BPSO algorithm, all the particles are attracted by the global best position. Therefore, they all converge to one of the global best positions at the end of the run. The algorithm for calculating the global best position is demonstrated in Algorithm 3.2. In this algorithm, *globalBest* and particle are Particle type object, which contains the Fitness, Velocity, *BestPosition*, *BestFitness*, **Vmax**, **Vmin** and some other function. For a particle, if the fitness is greater than the *BestFitness* of *globalBest* object or *globalBest* object equal to NULL, then *BestFitness* of *globalBest* object is updated by particle.

Algorithm-3.2: Pseudo-code for updating the global best position

```
for each particle:
    if (particle.Fitness > globalBest.BestFitness)
        globalBest = particle;
```

According to the above algorithm, if the BPSO find a global best solution then it remains unchanged during the course of the iteration.

Algorithm-3.3: Pseudo-code for MBPSO

Require: v_{max} the upper bound of velocity
Require: *max_iterations* the maximum number of iterations
Require: *swarm_size* the population size

```
// Initialize the swarm.
particles = []
for i=1 to popSize do
    Vi ∈ {-Vmax , Vmax}
    Xi ∈ {0, 1}
    Pariticles[i] = Xi
```

```
// Initialize the personal best and global best positions.
pbest = particles[:]
gbest = particles[0]
// Iterate until the stopping criterion is met.
for t in range(max_iterations):
    Update the velocities and positions of the particles;
    for i in range(swarm_size):
        Update velocity using Equ. (3.3);
        Update Position using equation-3.4 & 3.5
        Calculate I_f by using equation-3.6
    // Update the personal best and global best positions.
    for i in range(swarm_size):
        if fitness(particles[i]) > fitness(pbest[i]):
            pbest[i] = particles[i]
        if fitness(particles[i]) > fitness(gbest):
            gbest = particles[i]
// Return the global best solution.
return gbest
```

Chapter 4

Experimental Result

GA	ACO	BPSO	MBPSO
Crossover rate:0.8 Mutation rate:0.01	alpha :1.0 beta :2.0 evaporationRate:1.0 initialPheromone:1.0	C1 : 2.0 C2 : 2.0 V_{max} : 4 V_{min} : -4	C1 : 2.0 C2 : 2.0 V_{max} : 4 V_{min} : -4 D_f(max) : 5 D_f(min) : 1

Table 4.1: Parameter setting for different algorithms.

4.1 Experimental Setup

Table 4.1 shows the parameter setting for different algorithms. To do the experiments, all the algorithms have used the same number of population and number of iterations which are 20 and 1000, respectively.

4.2 Performance Matrices

In this research, we use following performance matrices for the evaluation of the modified BPSO algorithm.

- **Average max clique size:** The average maximum clique size by summing up the sizes of all the maximum cliques found and dividing it by the total number of runs. The formula for calculating the average maximum clique size (Avg) can be expressed as:

$$\text{Avg} = \frac{\text{Sum of maximum clique sizes}}{\text{Total number of runs}}$$

  The higher the average max clique size value, the better.
- **Average Number of Function Evaluations:** The average functions evaluations referred to the number of times the algorithm calls the fitness function to find the optimal solution. A lower value of the average function evaluation for an algorithm indicates that this algorithm is better in terms of computational efficiency.

4.3 Result and Discussion

Instance	No. of nodes	Best Known Max Clique Size
I-1	10	3
I-2	20	6
I-3	30	6
I-4	60	7
I-5	120	10
I-6	20	4
I-7	40	4
I-8	20	13
I-9	40	14
I-10	60	14
I-11	120	14

Table 4.2: Information for the benchmark max clique instances.

We provide the results of the compared algorithms in the following sections. To provide results, we use 11 challenging max clique instances having multiple cliques. The information for these instances is given Table 4.2.

Table 4.3 provides results for different algorithms in terms of clique size and function evaluations. The algorithms compared are BPSO, GA, ACO, and Modified BPSO. Each row represents a specific run, while the average clique size and function evaluation are provided at the bottom. For clique size, all algorithms consistently achieved a size of 3, indicating that they were equally effective in finding cliques. In terms of function evaluations, the best results are highlighted in bold. The BPSO algorithm consistently outperformed the other algorithms with the lowest number of function evaluations, achieving 1 in all runs. GA and ACO both achieved relatively good results, with an average of 4.5 and 2.4 function evaluations, respectively. The Modified BPSO algorithm had the highest number of function evaluations, with an average of 5.5. Overall, the BPSO algorithm performed the best in terms of both clique size and function evaluations, consistently achieving the largest cliques with the fewest function evaluations.

No. of run	Clique Size				Function Evaluations			
	BPSO	GA	ACO	Modified BPSO	BPSO	GA	ACO	Modified BPSO
1	3	3	3	3	1	2	1	1
2	3	3	3	3	1	2	1	13
3	3	3	3	3	1	2	1	4
4	3	3	3	3	1	6	1	2
5	3	3	3	3	1	1	1	12
6	3	3	3	3	1	1	1	12
7	3	3	3	3	1	12	1	1
8	3	3	3	3	1	1	1	11
9	3	3	3	3	1	3	1	3
10	3	3	3	3	1	10	3	1
11	3	3	3	3	1	2	3	10
12	3	3	3	3	1	6	3	11
13	3	3	3	3	1	6	3	3
14	3	3	3	3	1	2	3	1
15	3	3	3	3	1	8	3	1
16	3	3	3	3	1	1	3	2
17	3	3	3	3	1	14	3	4
18	3	3	3	3	1	1	3	6
19	3	3	3	3	1	5	3	15
20	3	3	3	3	1	4	3	1
21	3	3	3	3	1	1	3	3
22	3	3	3	3	1	11	3	3
23	3	3	3	3	1	5	3	1
24	3	3	3	3	1	3	3	15
25	3	3	3	3	1	6	3	4
26	3	3	3	3	1	3	3	7
27	3	3	3	3	1	6	3	3
28	3	3	3	3	1	4	3	3
29	3	3	3	3	1	2	3	9
30	3	3	3	3	1	5	3	3
Average	**3**	3	3	3	**1**	4.5	2.4	5.5

Table 4.3: Results on the test instance I-1 for the different algorithms.

Table 4.4 presents results for different algorithms (BPSO, GA, ACO, Modified BPSO) based on two metrics: Clique Size and Function Evaluations. The bold results indicate the best values. For the metric "Clique Size," all algorithms consistently achieved a size of 6, except for BPSO, which achieved an average size of 5. In terms of "Function Evaluations," the Modified BPSO algorithm achieved the best average performance with only 67.9 evaluations, followed by GA with 160.57 evaluations. ACO had an average of 10005.5 evaluations, while BPSO had the highest average of 12861 evaluations. In summary, the Modified BPSO algorithm had the best performance in terms of both Clique Size and Function Evaluations. The GA algorithm also performed well in terms of Function Evaluations. BPSO had a slightly lower average Clique Size compared to other algorithms.

No. of run	Clique Size				Function Evaluations			
	BPSO	**GA**	**ACO**	**Modified BPSO**	**BPSO**	**GA**	**ACO**	**Modified BPSO**
1	5	6	5	6	20000	71	20000	65
2	4	6	5	6	20000	11	20000	3
3	4	6	4	6	20000	193	20000	179
4	4	6	5	6	20000	107	20000	112
5	4	6	5	6	20000	279	20000	11
6	4	6	4	6	20000	41	20000	72
7	5	6	5	6	20000	85	20000	71
8	4	6	5	6	20000	66	20000	75
9	4	6	5	6	20000	121	20000	64
10	5	6	6	6	20000	536	11	5
11	5	6	6	6	20000	128	11	103
12	5	6	6	6	20000	25	11	193
13	5	6	6	6	20000	685	11	49
14	5	6	6	6	20000	48	11	4
15	4	6	6	6	20000	21	11	53
16	6	6	6	6	1300	140	11	89
17	6	6	6	6	4161	467	11	115
18	4	6	6	6	20000	31	11	1
19	4	6	6	6	20000	202	11	87
20	4	6	6	6	20000	82	11	68
21	5	6	6	6	20000	60	11	73
22	6	6	6	6	41	202	11	84
23	6	6	6	6	41	369	11	2
24	6	6	6	6	41	3	11	2
25	6	6	5	6	41	250	20000	10
26	6	6	5	6	41	418	20000	8
27	6	6	5	6	41	44	20000	88
28	6	6	5	6	41	20	20000	65
29	6	6	4	6	41	31	20000	109
30	6	6	5	6	41	81	20000	177
Average	5	6	5.4	**6**	12861	160.57	10005.50	**67.9**

Table 4.4: Results on the test instance I-2 for the different algorithms.

No. of run	Clique Size				Function Evaluations			
	BPSO	GA	ACO	Modified BPSO	BPSO	GA	ACO	Modified BPSO
1	5	6	5	6	20000	370	20000	166
2	6	6	5	6	6180	61	20000	233
3	5	6	5	6	20000	58	20000	77
4	5	6	5	6	20000	293	20000	34
5	5	6	5	6	20000	43	20000	230
6	5	6	5	6	20000	667	20000	23
7	5	6	5	6	20000	466	20000	28
8	5	6	6	6	20000	717	11	17
9	5	6	6	6	20000	11	11	21
10	5	6	6	6	20000	91	11	741
11	5	6	6	6	20000	209	11	274
12	5	6	6	6	20000	151	11	33
13	5	6	6	6	20000	144	11	190
14	5	6	6	6	20000	58	11	23
15	6	6	6	6	16401	134	11	6
16	5	6	6	6	20000	70	11	36
17	5	6	6	6	20000	63	11	112
18	5	6	5	6	20000	702	20000	131
19	5	6	5	6	20000	732	20000	69
20	4	6	5	6	20000	287	20000	42
21	5	6	5	6	20000	75	20000	103
22	4	6	5	6	20000	230	20000	17
23	6	6	5	6	5640	166	20000	201
24	5	6	5	6	20000	115	20000	260
25	5	6	6	6	20000	39	12	62
26	5	6	6	6	20000	76	12	158
27	5	6	5	6	20000	161	20000	316
28	6	6	5	6	2454	209	20000	80
29	5	6	5	6	20000	71	20000	13
30	5	6	5	6	20000	146	20000	125
Average	5.07	6	5.4	**6**	18355.83	220.5	12004.47	**127.37**

Table 4.5: Results on the test instance I-3 for the different algorithms.

Table 4.5 provides results for different algorithms .Looking at the "Clique Size" metric, the algorithms achieved varying results. BPSO had an average clique size of 5.07, while the other algorithms consistently achieved a size of 6. In terms of "Function Evaluations," the Modified BPSO algorithm performed the best with an average of 127.37 evaluations. GA had an average of 220.5 evaluations, ACO had an average of 12004.47 evaluations, and BPSO had the highest average of 18355.83 evaluations. To summarize, BPSO achieved a slightly lower average clique size compared to other algorithms. Modified BPSO had the best performance in terms of Function Evaluations, followed by GA. ACO had a significantly higher number of evaluations on average.

No. of run	Clique Size				Function Evaluations			
	BPSO	GA	ACO	Modified BPSO	BPSO	GA	ACO	Modified BPSO
1	6	7	6	7	20000	539	20000	974
2	6	7	5	7	20000	502	20000	656
3	6	7	6	7	20000	1050	20000	406
4	6	7	6	7	20000	79	20000	31
5	5	7	5	7	20000	124	20000	886
6	5	7	6	7	20000	61	20000	128
7	6	7	7	7	20000	636	20	76
8	6	7	7	7	20000	439	20	380
9	6	7	5	7	20000	20	20000	166
10	6	7	5	7	20000	1465	20000	140
11	6	7	5	7	20000	1219	20000	244
12	6	7	5	7	20000	61	20000	107
13	6	7	7	7	20000	519	15	985
14	6	7	7	7	20000	1912	15	605
15	5	7	6	7	20000	465	20000	363
16	6	7	6	7	20000	505	20000	306
17	6	7	6	7	20000	779	20000	93
18	7	7	7	7	16608	363	20	810
19	5	7	7	7	20000	355	20	704
20	5	7	7	7	20000	190	20	661
21	6	7	6	7	20000	286	20000	311
22	6	7	6	7	20000	1271	20000	801
23	5	7	6	7	20000	948	20000	63
24	5	7	6	7	20000	689	20000	306
25	6	7	6	7	20000	155	20000	245
26	6	7	6	7	20000	593	20000	316
27	6	7	5	7	20000	2863	20000	84
28	7	7	7	7	12498	467	8	346
29	6	7	7	7	20000	690	8	291
30	6	7	7	7	20000	1842	8	816
Average	5.83	7	6.1	**7**	19636.87	702.9	13338.47	**410**

Table 4.6: Results on the test instance I-4 for the different algorithms.

In Table 4.6, the results indicate that the Modified BPSO algorithm achieved the best performance in terms of both Clique Size and Function Evaluations. GA also performed well in terms of Function Evaluations. BPSO had a significantly higher number of evaluations on average.

No. of run	Clique Size				Function Evaluations			
	BPSO	GA	ACO	Modified BPSO	BPSO	GA	ACO	Modified BPSO
1	9	10	7	10	20000	14803	20000	5061
2	8	9	8	9	20000	20000	20000	20000
3	9	9	8	9	20000	20000	20000	20000
4	9	9	7	9	20000	20000	20000	20000
5	8	9	8	10	20000	20000	20000	7561
6	8	10	7	9	20000	8791	20000	20000
7	9	9	8	10	20000	20000	20000	14990
8	8	9	8	9	20000	20000	20000	20000
9	8	9	7	9	20000	20000	20000	20000
10	10	9	8	9	571	20000	20000	20000
11	9	9	8	10	20000	20000	20000	14826
12	8	9	8	9	20000	20000	20000	20000
13	10	10	8	10	630	11804	20000	3920
14	8	10	8	9	20000	4045	20000	20000
15	8	9	9	10	20000	20000	20000	103
16	9	9	8	9	20000	20000	20000	20000
17	7	9	8	10	20000	20000	20000	798
18	8	9	7	10	20000	20000	20000	6553
19	7	9	8	10	20000	20000	20000	6753
20	7	9	9	9	20000	20000	20000	20000
21	8	9	7	10	20000	20000	20000	8448
22	8	9	7	9	20000	20000	20000	20000
23	8	9	8	9	20000	20000	20000	20000
24	8	10	7	9	20000	15561	20000	20000
25	9	10	7	10	20000	19999	20000	18521
26	8	10	9	10	20000	9270	20000	3905
27	10	10	7	10	3676	6040	20000	4592
28	9	10	8	9	20000	9663	20000	20000
29	9	10	8	9	20000	11164	20000	20000
30	8	9	8	9	20000	20000	20000	20000
Average	8.4	9.33	7.77	**9.43**	18162.57	17038	20000	**14534.37**

Table 4.7: Results on the test instance I-5 for the different algorithms.

In Table 4.7, the results indicate that the Modified BPSO algorithm achieved the best performance in terms of both Clique Size and Function Evaluations. GA and BPSO also performed well in terms of Function Evaluations, but ACO had a lower average Clique Size. BPSO had a higher average Clique Size and a relatively higher number of evaluations.

No. of run	Clique Size				Function Evaluations			
	BPSO	GA	ACO	Modified BPSO	BPSO	GA	ACO	Modified BPSO
1	4	4	4	4	1114	11	4	3
2	4	4	4	4	21	14	4	4
3	4	4	4	4	21	14	4	4
4	4	4	4	4	21	13	4	14
5	4	4	4	4	21	3	4	38
6	4	4	4	4	21	18	4	2
7	4	4	4	4	21	2	4	1
8	4	4	4	4	209	7	4	9
9	4	4	4	4	21	1	15	1
10	4	4	4	4	21	17	15	4
11	4	4	4	4	21	4	15	114
12	4	4	4	4	21	2	15	26
13	4	4	4	4	21	54	18	3
14	4	4	4	4	90	30	18	13
15	4	4	4	4	21	18	18	74
16	4	4	4	4	21	30	18	6
17	4	4	4	4	21	27	18	64
18	4	4	4	4	21	38	18	78
19	4	4	4	4	249	5	18	1
20	4	4	4	4	21	5	18	1
21	4	4	4	4	21	5	18	2
22	4	4	4	4	21	35	18	5
23	4	4	4	4	21	11	18	18
24	4	4	4	4	21	11	18	6
25	4	4	4	4	21	5	18	11
26	4	4	4	4	21	13	18	7
27	4	4	4	4	21	16	18	10
28	4	4	4	4	46	40	18	5
29	4	4	4	4	46	8	18	42
30	4	4	4	4	41	104	18	3
Average	4	4	**4**	4	75.93	18.7	**13.87**	18.97

Table 4.8: Results on the test instance I-6 for the different algorithms.

In Table 4.8, the results indicate that the ACO algorithm achieved the best result followed by GA. The Modified BPSO and BPSO had higher average evaluations. Therefore, the best values are indicated by the bold results, with ACO having the lowest average evaluations and all algorithms having the same clique size.

No. of run	Clique Size				Function Evaluations			
	BPSO	GA	ACO	Modified BPSO	BPSO	GA	ACO	Modified BPSO
1	4	4	4	4	15344	4	11	29
2	4	4	4	4	644	21	11	15
3	4	4	4	4	68	5	11	1
4	4	4	4	4	190	41	11	4
5	4	4	4	4	603	12	11	29
6	4	4	4	4	106	7	11	11
7	4	4	4	4	208	17	11	20
8	4	4	4	4	596	1	11	14
9	3	4	4	4	20020	8	3	2
10	4	4	4	4	146	3	3	16
11	4	4	4	4	45	31	3	8
12	4	4	4	4	45	4	3	2
13	4	4	4	4	1106	3	3	2
14	4	4	4	4	5303	19	3	2
15	4	4	4	4	751	13	3	6
16	4	4	4	4	51	9	3	4
17	4	4	4	4	21	29	3	2
18	4	4	4	4	21	7	3	30
19	4	4	4	4	4709	9	20	1
20	4	4	4	4	890	1	20	6
21	4	4	4	4	50	5	20	14
22	4	4	4	4	110	52	20	21
23	4	4	4	4	353	7	20	24
24	4	4	4	4	1695	45	20	36
25	4	4	4	4	250	4	20	6
26	4	4	4	4	19030	24	20	20
27	4	4	4	4	237	2	6	1
28	4	4	4	4	3556	43	6	36
29	4	4	4	4	6119	10	6	26
30	4	4	4	4	1506	130	6	1
Average	3.97	4	**4**	4	2792.43	18.87	**10.07**	12.97

Table 4.9: Results on the test instance I-7 for the different algorithms.

In Table 4.9 the algorithms achieved similar clique sizes, mostly 4 with one run of Algorithm 9 achieving a clique size of 3. However, in terms of the Function Evaluations metric, ACO had the best result, followed by Modified BPSO. GA and BPSO had higher average evaluations. Therefore, the best values are indicated by the bold results, with ACO having the lowest average evaluations and all algorithms having similar clique sizes.

No. of run	Clique Size				Function Evaluations			
	BPSO	GA	ACO	Modified BPSO	BPSO	GA	ACO	Modified BPSO
1	12	13	9	13	20000	845	20000	263
2	12	13	9	13	20000	69	20000	566
3	10	13	10	13	20000	2787	20000	66
4	13	13	11	13	17552	5475	20000	322
5	13	13	10	13	15100	110	20000	316
6	13	13	9	13	12394	341	20000	487
7	12	13	9	13	20000	2246	20000	222
8	13	13	10	13	12767	410	20000	98
9	12	13	10	13	20000	52	20000	86
10	12	13	11	13	20000	158	20000	586
11	12	13	11	13	20000	330	20000	347
12	11	13	10	13	20000	149	20000	124
13	12	13	9	13	20000	750	20000	24
14	13	13	10	13	16993	364	20000	313
15	11	13	10	13	20000	693	20000	261
16	13	13	11	13	12230	975	20000	78
17	11	13	10	13	20000	312	20000	272
18	12	13	10	13	20000	584	20000	174
19	13	13	10	13	14664	2505	20000	370
20	13	13	10	13	2654	969	20000	127
21	13	13	8	13	4509	1646	20000	234
22	11	13	10	13	20000	4531	20000	98
23	12	13	8	13	20000	1743	20000	231
24	12	13	9	13	20000	2348	20000	354
25	13	13	9	13	12694	574	20000	138
26	13	13	10	13	15946	143	20000	213
27	11	13	10	13	20000	8700	20000	153
28	11	13	9	13	20000	1015	20000	199
29	11	13	9	13	20000	1695	20000	232
30	13	13	9	13	4369	315	20000	186
Average	12.1	13	9.67	**13**	16729.07	1427.8	20000	**238**

Table 4.10: Results on the test instance I-8 for the different algorithms.

In Table 4.10, the results indicate that the Modified BPSO algorithm outperformed the other algorithms in terms of both Clique Size and Function Evaluations. It consistently achieved the largest cliques and required the fewest function evaluations. The GA algorithm also performed well in terms of Function Evaluations. ACO had the lowest clique sizes and the highest number of function evaluations.

No. of run	Clique Size				Function Evaluations			
	BPSO	**GA**	**ACO**	**Modified BPSO**	**BPSO**	**GA**	**ACO**	**Modified BPSO**
1	13	13	9	13	20000	20000	20000	20000
2	12	13	10	13	20000	20000	20000	20000
3	12	13	10	13	20000	20000	20000	20000
4	10	13	11	13	20000	20000	20000	20000
5	13	13	10	13	20000	20000	20000	20000
6	13	13	9	13	20000	20000	20000	20000
7	13	13	8	13	20000	20000	20000	20000
8	13	13	9	14	20000	20000	20000	729
9	13	13	10	13	20000	20000	20000	20000
10	13	13	12	14	20000	20000	20000	1711
11	13	13	9	13	20000	20000	20000	20000
12	12	13	10	13	20000	20000	20000	20000
13	13	13	10	13	20000	20000	20000	20000
14	11	13	10	13	20000	20000	20000	20000
15	13	13	11	13	20000	20000	20000	20000
16	14	13	11	13	4555	20000	20000	20000
17	13	13	12	13	20000	20000	20000	20000
18	13	13	12	13	20000	20000	20000	20000
19	12	13	8	13	20000	20000	20000	20000
20	13	13	11	13	20000	20000	20000	20000
21	14	13	9	13	2028	20000	20000	20000
22	12	13	11	13	20000	20000	20000	20000
23	11	13	11	13	20000	20000	20000	20000
24	12	13	8	13	20000	20000	20000	20000
25	12	13	8	13	20000	20000	20000	20000
26	13	13	10	13	20000	20000	20000	20000
27	13	14	9	13	20000	15902	20000	20000
28	13	13	10	13	20000	20000	20000	20000
29	12	13	9	13	20000	20000	20000	20000
30	13	13	9	14	20000	20000	20000	1242
Average	12.57	13.03	9.87	**13.1**	18886.1	19863.4	20000	**18122.73**

Table 4.11: Results on the test instance I-9 for the different algorithms.

In Table 4.11, the results indicate that the Modified BPSO algorithm outperformed the other algorithms in terms of both Clique Size and Function Evaluations. It achieved the largest clique sizes and required fewer function evaluations compared to the other algorithms. GA also performed well in terms of Clique Size, but had slightly higher function evaluations. BPSO had lower clique sizes compared to the Modified BPSO and GA algorithms. ACO had the lowest clique sizes and a higher number of function evaluations.

No. of run	Clique Size				Function Evaluations			
	BPSO	GA	ACO	Modified BPSO	BPSO	GA	ACO	Modified BPSO
1	11	13	9	13	20000	20000	20000	20000
2	13	13	10	13	20000	20000	20000	20000
3	12	13	10	13	20000	20000	20000	20000
4	12	13	10	14	20000	20000	20000	2064
5	13	13	10	13	20000	20000	20000	20000
6	13	13	9	13	20000	20000	20000	20000
7	13	13	10	13	20000	20000	20000	20000
8	13	13	9	13	20000	20000	20000	20000
9	13	13	9	13	20000	20000	20000	20000
10	13	14	11	13	20000	5929	20000	20000
11	13	13	10	13	20000	20000	20000	20000
12	12	13	9	13	20000	20000	20000	20000
13	13	13	10	13	20000	20000	20000	20000
14	13	13	10	13	20000	20000	20000	20000
15	13	13	9	13	20000	20000	20000	20000
16	13	13	10	13	20000	20000	20000	20000
17	12	14	11	13	20000	9927	20000	20000
18	13	13	11	14	20000	20000	20000	3265
19	11	13	11	13	20000	20000	20000	20000
20	13	13	9	13	20000	20000	20000	20000
21	13	13	10	13	20000	20000	20000	20000
22	13	13	11	13	20000	20000	20000	20000
23	12	13	9	13	20000	20000	20000	20000
24	11	13	9	13	20000	20000	20000	20000
25	13	13	12	13	20000	20000	20000	20000
26	13	13	12	13	20000	20000	20000	20000
27	11	13	10	13	20000	20000	20000	20000
28	9	13	10	13	20000	20000	20000	20000
29	11	13	10	13	20000	20000	20000	20000
30	13	13	9	13	20000	20000	20000	20000
Average	12.37	13.07	9.97	**13.07**	20000	19195.2	20000	**18844.30**

Table 4.12: Results on the test instance I-10 for the different algorithms.

In Table 4.12, the results indicate that both the GA and Modified BPSO algorithms performed well in terms of Clique Size, achieving the highest average sizes. The Modified BPSO algorithm also outperformed the other algorithms in terms of Function Evaluations, requiring the fewest evaluations on average. BPSO had lower clique sizes compared to GA and Modified BPSO, and ACO had the lowest clique sizes but required the same number of function evaluations for all runs.

No. of run	Clique Size				Function Evaluations			
	BPSO	GA	ACO	Modified BPSO	BPSO	GA	ACO	Modified BPSO
1	12	13	10	13	20000	20000	20000	20000
2	13	13	10	13	20000	20000	20000	20000
3	13	13	9	13	20000	20000	20000	20000
4	13	14	9	13	20000	17351	20000	20000
5	13	13	11	13	20000	20000	20000	20000
6	13	13	11	13	20000	20000	20000	20000
7	13	13	11	13	20000	20000	20000	20000
8	12	13	9	13	20000	20000	20000	20000
9	13	13	10	13	20000	20000	20000	20000
10	13	13	11	13	20000	20000	20000	20000
11	12	14	9	13	20000	9713	20000	20000
12	13	13	11	13	20000	20000	20000	20000
13	13	13	9	13	20000	20000	20000	20000
14	13	13	10	14	20000	20000	20000	161
15	13	13	9	14	20000	20000	20000	75
16	13	13	9	13	20000	20000	20000	20000
17	12	14	11	13	20000	14157	20000	20000
18	13	13	9	13	20000	20000	20000	20000
19	13	13	8	13	20000	20000	20000	20000
20	13	13	10	13	20000	20000	20000	20000
21	13	13	10	13	20000	20000	20000	20000
22	13	13	11	13	20000	20000	20000	20000
23	13	13	8	13	20000	20000	20000	20000
24	13	13	11	13	20000	20000	20000	20000
25	13	13	10	13	20000	20000	20000	20000
26	13	13	9	13	20000	20000	20000	20000
27	13	13	10	13	20000	20000	20000	20000
28	13	13	10	13	20000	20000	20000	20000
29	13	14	10	13	20000	18275	20000	20000
30	13	13	9	13	20000	20000	20000	20000
Average	12.87	**13.13**	9.8	**13.07**	20000	19316.53	20000	**18674.53**

Table 4.13: Results on the test instance I-11 for the different algorithms.

In Table 4.13, the results indicate that GA and Modified BPSO algorithms performed well in terms of Clique Size, achieving the highest average sizes. The Modified BPSO algorithm also outperformed the other algorithms in terms of Function Evaluations, requiring the fewest evaluations on average. BPSO had lower clique sizes compared to GA and Modified BPSO, while ACO had the lowest clique sizes but required the same number of function evaluations for all runs.

In summary, the Modified BPSO algorithm had the best performance in terms of both Clique Size and Function Evaluations for test instance that are used through the experiments. The GA algorithm also performed well in terms of Function Evaluations, while ACO had the highest number of evaluations. BPSO had a slightly lower average Clique Size compared to other algorithms.

Chapter 5

Conclusion

In conclusion, this study proposed an effective Binary Particle Swarm Optimization (BPSO) approach for solving the Maximum Clique problem. The objective was to leverage the power of swarm intelligence to efficiently find cliques in large graphs. The modified BPSO algorithm addressed the limitations of the original BPSO by enhancing exploration and exploitation and incorporating problem-specific constraints. Through extensive experimentation and comparative analysis, the effectiveness of the proposed approach was demonstrated. The modified BPSO algorithm showed superior performance in terms of solution quality and convergence speed compared to existing state-of-the-art algorithms for the Maximum Clique problem. The algorithm successfully handled large search spaces and provided feasible solutions adhering to the problem's constraints.

The contributions of this study go beyond the development of the modified BPSO algorithm. The research highlighted the potential of swarm intelligence techniques, specifically BPSO, in tackling challenging combinatorial optimization problems. By extending the application of BPSO to the Maximum Clique problem, we provided valuable insights into the capabilities and limitations of the algorithm in this specific domain.

The findings of this study have practical implications in various fields where clique identification is crucial, such as social network analysis, bioinformatics, and telecommunications. The ability to efficiently find cliques in large graphs can contribute to better understanding community structures, optimizing network resources, and improving decision-making processes.

However, there are still areas for future research and improvement. Further investigations can focus on fine-tuning the parameters of the modified BPSO algorithm and exploring additional enhancements to boost its performance. Additionally, testing the algorithm on more diverse and complex datasets would provide a more comprehensive evaluation of its scalability and robustness.

Overall, the proposed effective Binary Particle Swarm Optimization approach for the Maximum Clique problem has demonstrated promising results. It opens up possibilities for future research in swarm intelligence and combinatorial optimization, ultimately contributing to the advancement of optimization algorithms and their applications in real-world problem-solving.

References

[1] Islam, M. J., Li, X., & Mei, Y. (2017). A time-varying transfer function for balancing the exploration and exploitation ability of a binary PSO. Applied Soft Computing, 59, 182-196.

[2] Bui, T.N., Eppley, P.H.: A hybrid genetic algorithm for the maximum clique problem. In: Proceedings of the 6th International Conference on Genetic Algorithms, pp. 478–484 (1995).

[3] Mohammad Soleimani-Pouri, Alireza Rezvanian, and Moham-mad Reza Meybodi. An ant based particle swarm optimization al-gorithm for maximum clique problem in social networks. State of the art applications of social network analysis, pages 295–304, 2014.

[4] Christine Solnon and Serge Fenet. A study of aco capabilities for solving the maximum clique problem. Journal of Heuristics, 12:155–180, 2006.

[5] Marchiori, E.: A simple heuristic based genetic algorithm for the maximum clique problem. In: Proceedings of the 1998 ACM Symposium on Applied Computing – SAC'98 (1998)

[6] Qinghua Wu and Jin-Kao Hao. A review on algorithms for maxi-mum clique problems. European Journal of Operational Research, 242(3):693–709, 2015

[7] Roberto Battiti and Marco Protasi. Reactive local search for the maximum clique problem 1. Algorithmica, 29:610–637, 2001.

[8] Deniss Kumlander. A new exact algorithm for the maximum-weight clique problem based on a heuristic vertex-coloring and a backtrack search. In Proc. 5th Int'l Conf. on Modelling, Computation and Optimization in Information Systems and Management Sciences, pages 202–208, 2004.

[9] Mohammad Soleimani-Pouri, Alireza Rezvanian, and Moham-mad Reza Meybodi. Finding a maximum clique using ant colony optimization and particle swarm optimization in social networks. arXiv preprint arXiv:1311.7213, 2013

[10] James Kennedy and Russell C Eberhart. A discrete binary version of the particle swarm algorithm. In 1997 IEEE International conference on systems, man, and

cybernetics. Computational cybernetics and simulation, volume 5, pages 4104–4108. IEEE, 1997.

[11] Eberhart, R. C., and Kennedy, J. (1995). A new optimizer using particle swarm theory. Proceedings of the Sixth International Symposium on Micro Machine and Human Science, Nagoya, Japan, 39-43. Piscataway, NJ: IEEE Service Center.

[12] Zwe-Lee Gaing. Discrete particle swarm optimization algorithm for unit commitment. In 2003 IEEE Power Engineering Society General Meeting (IEEE Cat. No. 03CH37491), volume 1, pages 418–424. IEEE, 2003

[13] Qinghua Wu and Jin-Kao Hao. An adaptive multistart tabu search approach to solve the maximum clique problem. Journal of Combina-torial Optimization, 26:86–108, 2013.

[14] Pierre Hansen, Nenad Mladenovi´c, and Dragan Uroˇsevi´c. Variable neighborhood search for the maximum clique. Discrete Applied Mathematics, 145(1):117–125, 2004.

[15] 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, pages 1021–1025, 2004.

[16] Emile HL Aarts and Jan Karel Lenstra. Local search in combinatorial optimization. Princeton University Press, 2003.

APPENDIX

Implementation of C# Code:

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Linq;
using System.Collections.Generic;
using System.Text;
namespace MBPSO
{
   public class BPSO
   {
       //Initial Parameters for BPSO
         Random Rnd = new Random((int)DateTime.Now.Ticks);
         Particle[] swarm;
         double w;           //Inirtia weight
         double wMax;        //Max inirtia weight
         double wMin;        //Min inirtia weight
         double c1=2;
         double c2=2;
         double Vmax = 4;
         double Vmin=-4;

         double jj = 0;
         double wMax1 = 3.7;//0.9;       //Max inirtia weight
         double wMin1 = 0.5;       //Min inirtia weight
         double[] velocity; // Velocity=zeros(noP,noV);%Velocity vector
         string position; //=zeros(noP,noV);%Position vector
         ////////Cognitive component/////////
         double pBestFit;     // =zeros(noP);
         string pBest;     //=zeros(noP,noV);
         ////////Social component//////////
         double gBestFit;
         string gBest;       //=zeros(1,noV);
         int BPSO_No;
         double[] gbest;
         List<string> pbest;
         List<double[]> vel;
         double[] VelMonitor;
         double [] SigMonitor;
         int[,] adjacencyMatrix;
        // Pop. diversity measure
```

```
double[] diversity;
int[, ] count;
int FE;
public int FEs
{
   get { return FE; }
}
public Particle [] Swarm
{
   get {return swarm; }
}
public string Gbest
{
   get { return gBest; }
}
public double GbestFit
{
   get {return gBestFit;}
}
public double[] W
{
   get { return weight; }
}
public double[] P
{
   get { return profit; }
}
public int I
{
   get { return Item; }
}
public double [] Convergence
{
   get { return gbest; }
}
public  List<string> Pbest
{
   get { return pbest; }
}
public List<double[]> Vel
{
   get { return vel; }
}
public double[] velMonitor
{
   get { return VelMonitor; }
}
public double[] sigMonitor
```

```
{
   get { return SigMonitor; }
}
public double[] Diversity
{
   get { return diversity; }
}
public int[,] Count
{
   get { return count; }
}
//Initialization
 public string Initialization(int NoP, int Max_iteration,int BPSO_num,int
 NoItem)
{
   FE = 0;
  //count
   wMax = 0.9;       //Max inirtia weight
   wMin = 0.4;       //Min inirtia weight
   gbest = new double[Max_iteration];
   pbest = new List<string>();
   vel = new List<double[]>();
   VelMonitor = new double[Max_iteration];
   SigMonitor = new double[Max_iteration];
   gBestFit = double.MinValue;
   gBest = string.Empty;
   BPSO_No = BPSO_num;
   swarm = new Particle[NoP];
   int[,] adjacencyMatrix = {
          {0,1,1,1,0,1,0,1,0,1},
          {1,0,1,0,1,0,0,0,1,0},
          {1,1,0,0,0,0,1,0,0,1},
          {1,0,0,0,1,0,1,0,0,0},
          {0,1,0,1,0,1,0,0,0,1},
          {1,0,0,0,1,0,1,0,0,0},
          {0,0,1,1,0,1,0,1,1,0},
          {1,0,0,0,0,0,1,0,1,0},
          {0,1,0,0,0,0,1,1,0,1},
          {1,0,1,0,1,0,0,0,1,0},
          };
   int KnownBestFiteness = 3;
   Item = adjacencyMatrix.GetLength(1);
   NoItem = Item;
   if (BPSO_num == 2)
   {
      //small dimension
      if (NoItem < 100)
      {
```

```
        Vmax = 4;
        Vmin = -Vmax;
    }
    else
    {
        Vmax = 10;
        Vmin = -10;
    }
}
else if (BPSO_num == 1)
{
        Vmax = 4;
        Vmin = -Vmax;
}
else if (BPSO_num == 6)
{
    Vmax = 6;
    Vmin = -Vmax;
}
else
{
    Vmax = 4;
    Vmin = -Vmax;
}
char c;
StringBuilder g,g1;
velocity = new double[NoItem];
int r;
for (int i = 0; i < NoP; i++)
{
    g = new StringBuilder("");
    g1 = new StringBuilder("");
    for (int j = 0; j < NoItem; j++)
    {
        g1.Append('0');
        velocity[j] = (Vmax - Vmin) * Rnd.NextDouble() + Vmin;
       // r = Rnd.Next(0,2);
      //  MessageBox.Show(velocity[j].ToString ());
        if (Rnd.NextDouble() <= 0.5)
            c = '1';
        // c =(char)(r+'0');
        else
            c = '0';
        g.Append(c);
    }
  //MessageBox.Show(g.ToString());
    position = g.ToString();
    pBest = g.ToString();
```

```
    pBestFit = CostFunction(pBest);
    swarm[i] = new Particle(velocity,position,pBest,pBestFit,NoItem);
}
diversity = new double[Max_iteration];
//Count
int[, ,] x1 = new int[Max_iteration,NoP,NoItem];
int found = 0;
for (int i = 0; i < Max_iteration; i++)
{
    //Measure distance between particles
        //Calculate fitness of each particle
    for (int j = 0; j < swarm.Length; j++) // each Particle
    {
        swarm[j].Fitness = CostFunction(swarm[j].Position);
        FE++;
        if (swarm[j].PbestFit < swarm[j].Fitness)
        {
            swarm[j].PbestFit = swarm[j].Fitness;
            swarm[j].PBest = swarm[j].Position;
            swarm[j].ListOfpBestUpdate.Add(swarm[j].PBest);
            swarm[j].ListOfpBestUpdateGen.Add(i);
        }
        if (gBestFit < swarm[j].Fitness)
        {
            gBestFit = swarm[j].Fitness;
            gBest = swarm[j].Position;
            if (gBestFit == KnownBestFiteness)
            {
                found = 1;
                break;
            }
        }
    }
    //For convergence record
    gbest[i] = gBestFit;
    if (found == 1)
        break;
   // pbest.Add(swarm[0].PBest);
   // vel.Add(swarm[0].Velocity);
    //Update the W of PSO

    jj = wMax1 - i * ((wMax1 - wMin1) / Max_iteration);
    if (BPSO_num > 4 && BPSO_num < 7)
    {
        w = wMax - i * ((wMax - wMin) / Max_iteration);
    }
    else
        w = 1;
```

```
        //Update the Velocity and Position of particles
        string oldPosition = string.Empty;
        for (int j = 0; j < swarm.Length; j++) // each Particle
        {
            oldPosition = swarm[j].Position;
            for (int k = 0; k < Item; k++)
            {
                swarm[j].Velocity[k] =w*swarm[j].Velocity[k] + c1 *
                Rnd.NextDouble() * (swarm[j].PBest[k] - swarm[j].Position[k]) + c2
                * Rnd.NextDouble() * (gBest[k] - swarm[j].Position[k]);

                if (swarm[j].Velocity[k] > Vmax)
                    swarm[j].Velocity[k] = Vmax;
                if (swarm[j].Velocity[k] < -Vmax)
                    swarm[j].Velocity[k] = -Vmax;

            }
            swarm[j].Position = UpdatePosition(swarm[j].Position,
            swarm[j].Velocity, BPSO_No,jj,i);
        }
    }
    string sss=string.Empty ;
    return sss;
}
public double Hamming(Particle[] swarm1)
{
    int result;
    double[] r = new double[swarm1.Length];
    for (int i = 0; i < swarm1.Length; i++)
    {
        result = 0;
        for (int j = 0; j < swarm1.Length; j++)
        {
            result+=Hamming_dist(swarm1[i].Position, swarm1[j].Position);

        }
        r[i] = result / swarm1.Length;
    }
    return r.Average();
}
public int Hamming_dist(string first, string second)
{
    int result = 0;
  //  MessageBox.Show(first +"  "+second);
    for (int i = 0; i < first.Length; i++)
    {
        if (!first[i].Equals(second[i]))
        {
```

```
            result++;
        }
    }
  //  MessageBox.Show(result .ToString ());
    return result;
}


 public string UpdatePosition(string pos, double[] vel, int BPSO_num, double
 j, int xx)
{
    double sig=0.0;
    StringBuilder g = new StringBuilder("");
    char c='0';
    for (int i = 0; i < Item; i++)
    {
        if (BPSO_num == 1)
        {
            sig = 1 / (1 + Math.Exp(-vel[i])); //S1 transfer
        }
        if (BPSO_num == 2)
        {
            sig = 1 / (1 + Math.Exp(-vel[i]) / j); //S2 transfer
        }
        if (BPSO_num == 3)
        {
            sig = 1 / (1 + Math.Exp(-vel[i] / 2)); //S3 transfer
        }
        if (BPSO_num == 4)
        {
            sig = 1 / (1 + Math.Exp(-vel[i] / 3)); //S4 transfer
        }
        if (BPSO_num <= 4)
        {
            double r=Rnd.NextDouble();
            if ( r< sig)
                   c = '1';
                else
                    c = '0';
        }
        if (BPSO_num == 5)
        {
            sig = 1 / (1 + Math.Exp(-vel[i]));
            //sig = Math.Abs(Math.Tanh(vel[i])); //S5 transfer V-Shaped
        }
        if (BPSO_num == 6)
        {
          //  sig = 1 / (1 + Math.Exp(-vel[i]));
```

```
            sig = Math.Abs((2 / Math.PI) * Math.Atan(Math.PI / 2 * vel[i])); //S6
            transfer V-shaped
        }
        if (BPSO_num > 4 && BPSO_num < 7)
        {
            double r = Rnd.NextDouble();
            if (r < sig)
            {
                //  MessageBox.Show("rand="+r.ToString() + " Sig=" +
                sig.ToString()+" V="+vel[i].ToString ());
                if (pos[i] == '1')
                    c = '0';
                else
                    c = '1';
            }
            else
                c = pos[i];
        }
        if (BPSO_num == 7)
        {
            int p = 0;
            if (pos[i] == '1')
                p = 1;
            sig = (p + vel[i] + Vmax) / (1 + 2 * Vmax); //S7 transfer linear
            if (Rnd.NextDouble() < sig)
                c = '1';
            else
                c = '0';
        }
        if(i==0)
            SigMonitor[xx]=sig;
        g.Append(c);
    }
    return g.ToString();
}
public double CostFunction(string pos)
{
    int size = pos.Length;
    List<int> clique = new List<int>();
    for (int i = 0; i < size; i++)
    {
        if (pos[i] == '1')
        {
            // Check if the vertex i is adjacent to all the vertices in the current
            clique
            bool isAdjacentToAll = true;
            foreach (int j in clique)
            {
```

```
                    if (adjacencyMatrix[i, j] == 0)
                    {
                        isAdjacentToAll = false;
                        break;
                    }
                }
                if (isAdjacentToAll)
                {
                    clique.Add(i);
                }
            }
        }
        return clique.Count;
    }
}
public class Particle
{
    double[] velocity;
    string position;
    string pBest;
    double pBestFit;
    double fitness;
    public List<string> ListOfpBestUpdate = new List<string>();
    public List<int> ListOfpBestUpdateGen = new List<int>();
    public string Position
    {
        get { return position; }
        set { position = value; }
    }
    public double[] Velocity
    {
        get { return velocity; }
        set { velocity = value; }
    }
    public double Fitness
    {
        get { return fitness; }
        set { fitness = value; }
    }

    public string PBest
    {
        get { return pBest; }
        set { pBest = value; }
    }
    public double PbestFit
    {
        get { return pBestFit; }
```

```
            set { pBestFit = value; }
        }
        public Particle(double[] vel, string pos, string pbest, double pbestFit,int NoItem)
        {
            velocity = new double[NoItem];
            velocity = vel;
            position = pos;
            pBest = pbest;
            pBestFit = pbestFit;
        }
    }
}
```