REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE MOHAMED BOUDIAF - M'SILA

FACULTE :

DEPARTEMENT : Informatique

N° : ……

DOMAINE : Mathématiques informatique

FILIERE : Informatique

OPTION : IDO

Mémoire présenté pour l'obtention
Du diplôme de Master Académique

Par : REZZAG LEBZA MOHAMED AYMEN

Intitulé

Meta-heuristic based approach for Minimum Vertex Cover Problem

Soutenu devant le jury composé de :

……	Université de M'sila	Président
BOUZAROURA Ahlam	Université de M'sila	Rapporteur
……	Université de M'sila	Examinateur

Année universitaire : 2018 /2019

Acknowledgement

with the name of ALLAH, the most clement
the most merciful

first of all, I thank ALLAH the ablest who
gives me the power to complete this work

Then to my supervisor bouzaroura ahlem who
did encourage and help me to do this work and
all my family who stand with me

To everyone that helped me …

I dedicate this overmodest work

Table of contents

Chapter 2 Graph Theory

Chapter 3 Minimum vertex cover problem

Chapter 4 **Experimental results**

List of Figure

List of Table

With the huge explosion of data, finding an optimal solution became a great challenge, societies and companies start to suffer to extract knowledge and important information, while scientists were watching nature facing complexed problems but they were solving these problems with simple ways.

Optimization is a scientific term that is concerned with obtaining the best result under given circumstances. Many challenging applications in business, economics, and engineering can be modeled as optimization problems. In fact, real-life problems solving are often complex and difficult to solve to which optimization plays a key role in finding feasible solutions to these problems.

Combinatorial optimization is a lively field of applied mathematics, combining techniques from combinatorics, linear programming, and the theory of algorithms, to solve optimization problems over discrete structures. A combinatorial optimization problem is defined over a set of instances (admissible input data); each instance has a finite set of feasible solutions associated with it. Given an instance, the aim is to determine solutions that maximize or minimize a certain measure function.

the domain of bio-inspired or evolutionary algorithms (EA) , came to simulate this stochastic algorithms and trying to use it to hard problems, and this domain became the main subject for researchers ,those algorithms showed us a new era of computing because of its capabilities to learn and use probabilities to find meaningful information from a large search space in an optimal time.

Vertex cover problem is about to find a set of vertices that cover all other vertices, finding the minimal vertex cover in a large graph is in NP class problems.

The question is how to find a good solution within respected times? And for this reason we have proposed this work.

In this thesis we study the MVC on an undirected graphs, finding a minimum vertex cover of a general graph is an NP-complete problem, and its applications are in routers problems and race circuits and biochemistry.

The aim of this thesis is to apply two algorithms, heuristic which is genetic algorithm and metaheuristic which is bat algorithm then doing a comparison between them to show the effectiveness of each algorithm.

The thesis is structured in four chapters:

Chapter 1 reviews some the basic concept of combinatorial optimization problems algorithms, including the complexity class of problems (Class P, class NP and Class NP-complete), methods of resolution such as exact methods, approximate methods such as heuristics and metaheuristics.

Chapter 2 gives a general introduction to the basic terminology of graph theory pertinent to our study.

Chapter 3 explains the minimum vertex cover problem, and the proposed solutions

Chapter 4 reports the results of an experimental performance evaluation on random generated graphs and several well-known benchmark datasets.

Finally, a general conclusion, that resumes all.

Chapter 1

Combinatorial optimization

1. Introduction:

Combinatorial optimization is huge and large field of study, finding and optimal solution by minimizing or maximizing an objective function became so hard with the growth of data, and big companies were facing a serious problems with exact methods to get optimality from a big size data, a lot of papers and books published and in a small period it became the main search for researchers.

Combinatorial optimization problems are manifested in all fields of technology and industry, it had an important application in mathematics and artificial intelligence and airline companies, and we can say that it appears in several fields in our world today.

2. Concepts of optimization problems:

In this section, we will introduce some of the important concepts and definitions that appears very often in the field of combinatorial optimization:

- Optimization:

  Is the process of starting from inputs to find the minimum or maximum result using a fitness method to test the solutions given by the process.

- Algorithm:

An algorithm is a well-defined procedure that allows a computer to solve a problem using inputs and some criteria, it starts from an initial state usually random state and it maximize or minimize the solutions throughout a given iterations is a procedure that allows computer to solve a problem (minimize or maximize), it start from an initial state and it takes number of iterations and inputs and criteria depending on the need.

- The complexity of an algorithm:

The complexity of an algorithm is a way to classify how efficient an algorithm is, compared to alternative ones. The focus is on how execution time increases with the data set to be processed. The computational complexity and efficient implementation of the algorithm are important in computing, and this depends on suitable data structures.

- Big O notation:

Big O is used to describe the worst-case complexity time or space.

- The time complexity:

Time complexity is a way to describe how long an algorithm takes time to solve a problem, it usually counts on how long each iteration takes or a line takes, it helps to know if this algorithm is efficient or not.

- The space complexity:

The space complexity of an algorithm describes how much memory the algorithm needs in order to operate.

- Polynomial time:

An algorithm is said to be solvable in polynomial time if the number of steps required to complete the algorithm for a given input is O (Nk) for some non-negative integer k, where n is the complexity of the input [a].

- Exponential time:

An algorithm is said to be exponential time, if T (n) is upper bounded by 2poly (n), where poly (n) is some polynomial in n. More formally, an algorithm is exponential time if T (n) is bounded by O (2nk) for some constant k [a].

- Decision problem:

A decision problem that has only two possible answers: yes, or no, for

- Nondeterministic algorithm:

A nondeterministic algorithm can provide different outputs for the same input on different executions. Unlike a deterministic algorithm, which produces only a single output for the same input even on different runs, a nondeterministic algorithm travels in various routes to arrive at the different outcomes. Nondeterministic algorithms are useful for finding approximate solutions.

- Local Optimal:

When a nonlinear model is solved, we say the solution is a local optimum. We have to be aware that objective can get a better solution.

- A globally optimal:

Solution is a feasible solution with an objective value that is as good as or better than all other feasible solutions to the model.

3. Complexity classes:

A complexity class contains a set of problems that take a similar range of space and time to solve; Problems are usually proven to be in a particular complexity class by running the problem on an abstract computational model, usually a Turing machine. Some of the largest unsolved problems in computer science have to do with proving whether or not two complexity classes are equivalent or not, the most notable example of this is the infamous P = NP problem , scientists have so far been unable to prove or disprove the equality of the P class and the NP class[b].

Some complexity classes are a subset of others. For example, the class of problems solvable in deterministic polynomial time, P is a subset of the class of problems solvable in nondeterministic polynomial time NP.

3-1 Class P:

The class P contains decision problems (problems with a “yes” or “no” answer) that are solvable in polynomial time by a deterministic Turing machine. Sometimes this is phrased as: P is the class of languages that are decidable in polynomial time.

Many problems can be solved using a brute-force approach, but this often takes exponential time. If a problem has a polynomial time algorithm, it can be solved much more efficiently than by the brute force method. Many of the algorithms that are actually used in practice are polynomial time algorithms — exponential time algorithms are typically less useful and are avoided as much as possible[b].

A problem in P can be solved in Nk time for some constant K and where N is the size of the input[b].

Once some polynomial time algorithm is discovered for a problem, it is often possible to improve its running time.

Some problems in P

- Finding if numbers are relatively prime.
- Maximum matching.
- Linear programming.

3.2 Class NP:

The class NP contains decision problems (problems with a “yes” or “no” answer) that are solvable by a Turing machine.

- In nondeterministic polynomial time this includes problems that are solvable in polynomial time up to problems that are solvable in exponential time. While they can take a long time to solve, problems in NP can be verified by a Turing machine in polynomial time. This means that if given a “yes” answer to an NP problem, you can check that it is right in polynomial time. This “yes” answer is often called a witness or a certificate[b].
- Proving that a problem is in NP typically involves showing that you can provide a witness string that can be verified in polynomial time.
- P is contained within NP, which can be written $P \in NP$, but it is unknown if P and NP are equal. However, most theoretical computer scientists believe that $P \neq NP$.

Some Problems in NP:

- Graph isomorphism.
- The traveling salesman problem.
- Factoring.
- Boolean satisfiability.
- Graph coloring.

3.3 NP-Complete Problems:

There is two important classes, which are NP-Complete and NP-Hard

3.3.1 NP-complete:

NP-Complete problems are very special because any problem in the NP class can be transformed or reduced into NP-Complete problems in polynomial time. This means that if you can solve an NP-Complete problem, you can solve any other problem in NP. An important consequence of this is that if you could solve an NP-Complete problem in polynomial time, then you could solve any NP problem in polynomial time[b].

3.3.2 NP-Hard:

An NP-Hard problem is at least as hard as the hardest problems in NP. A problem A is NP-Hard if for every problem L in NP, there is a polynomial-time reduction from L to A. Another way to put it, more generally is, a problem A is reducible to problem B if an algorithm for solving problem B could also be used to solve problem [b].

NP-Hard problems do not have to be in NP, and they do not have to be decidable. For example, the halting problem is an NP-Hard problem, but is not an NP problem[b].

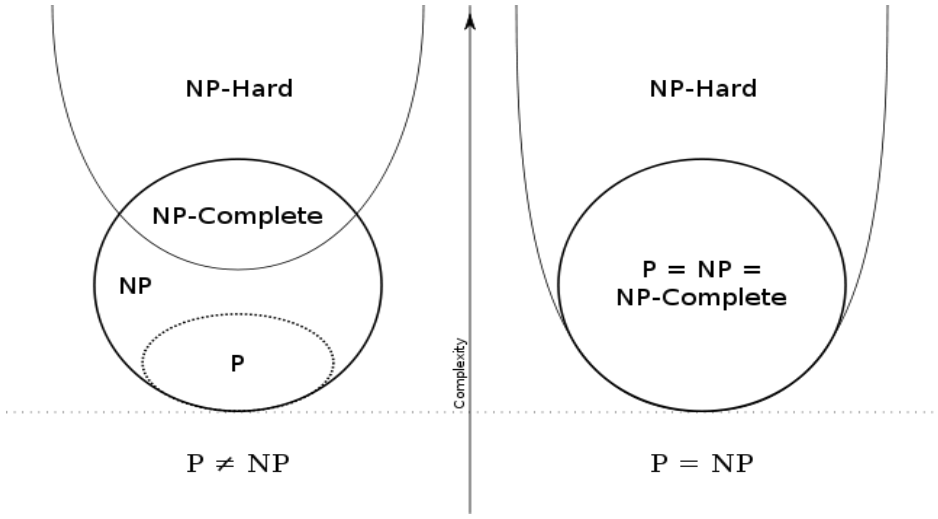


Figure 1.1: the Cook-Levin Theorem[b]

Some problems that are NP-complete:

- Knapsack problem.
- Graph isomorphism.
- Vertex cover.
- Clique problem.
- Hamiltonian path problem.
- Graph coloring.

4. Karps 21 problem:

In computational complexity theory, Karp's 21 NP-complete problems are a set of computational problems which are NP-complete. In his 1972 paper, "Reducibility Among Combinatorial Problems"[1], depending on the theorem that the Boolean satisfiability problem is NP-complete to show that there is a polynomial time many-one reduction from the Boolean satisfiability problem to each of 21 combinatorial and graph theoretical computational

problems, thereby showing that they are all NP-complete. This was one of the first demonstrations that many natural computational problems occurring throughout computer science are computationally intractable, and it drove interest in the study of NP-completeness and the P versus NP problem.

The problems Karp's 21 problems are shown below, many with their original names:

- Satisfiability: the Boolean satisfiability problem for formulas in conjunctive normal form (often referred to as SAT).
- integer programming (A variation in which only the restrictions must be satisfied, with no optimization)
- Clique (see also independent set problem)
  - Set packing
  - Vertex cover
  - Set covering
  - Feedback node set
  - Feedback arc set
  - Directed Hamilton circuit (Karp's name, now usually called Directed Hamiltonian cycle)
  - Undirected Hamilton circuit (Karp's name, now usually called Undirected Hamiltonian cycle)
  - Satisfiability with at most 3 literals per clause (equivalent to 3-SAT)
    - Chromatic number (also called the Graph Coloring Problem)
      - Clique cover
      - Exact cover
        - Hitting set
        - Steiner tree
        - 3-dimensional matching
        - Knapsack (Karp's definition of Knapsack is closer to Subset sum)
          - Job sequencing
          - Partition
          - Max cut

As time went on it was discovered that many of the problems can be solved efficiently if restricted to special cases, or can be solved within any fixed percentage of the optimal result. However.

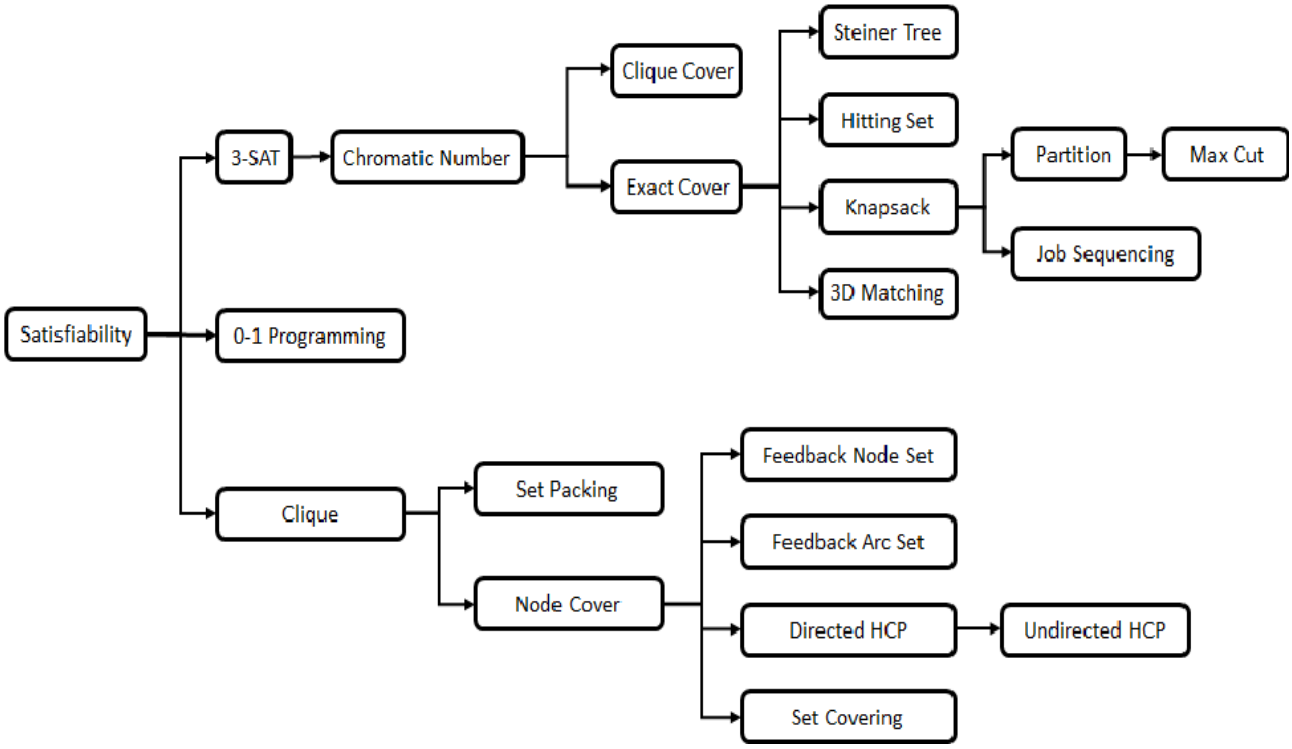


Figure1.2: Karp's 21 NP-complete problems[c]

5. Exact methods in combinatorial optimization:

The exact methods aim to find optimal solutions, but they are often extremely time-consuming when solving real-world problems (i.e. problems with large dimensions, hardly constrained problems, multimodal and/or time-varying problems).

6. Heuristic and metaheuristic:

Heuristic and metaheuristic techniques are powerful and flexible search methodologies that have successfully tackled practical difficult problems. Heuristic and meta-heuristic algorithms seek to produce good-quality solutions in reasonable computation times and good enough for practical purposes.

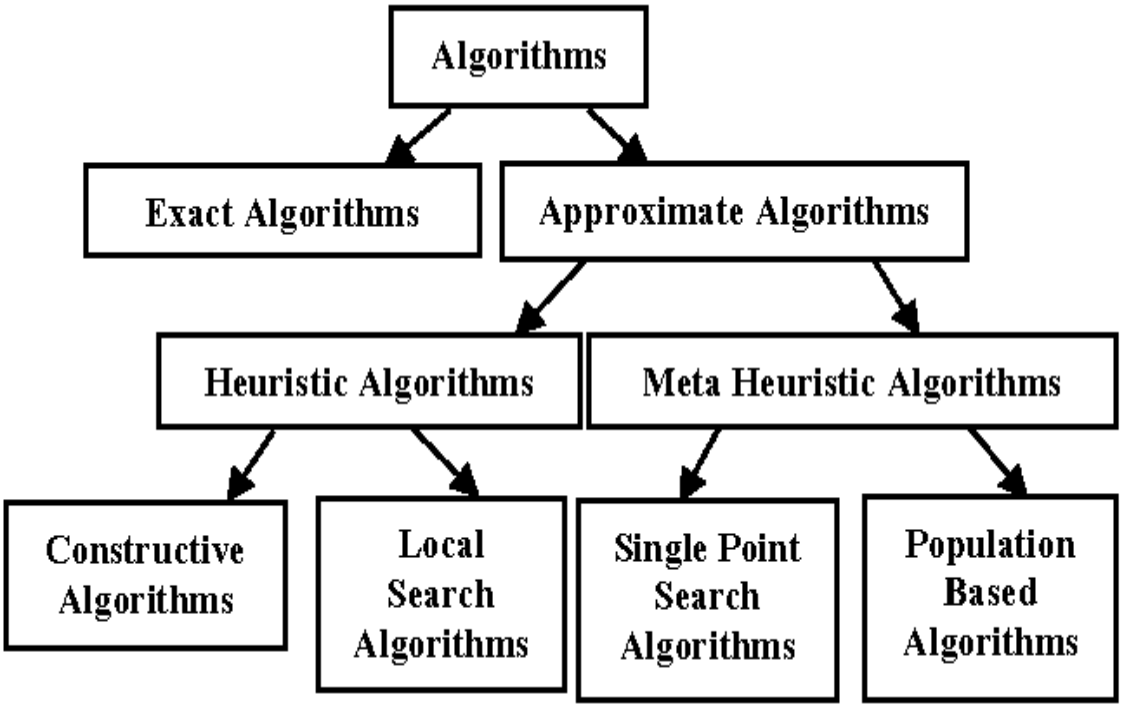


Figure 1.3: Heuristic and metaheuristic algorithms[c]

6.1 Heuristics:

Heuristic refers to a problem-solving method executed through learning-based techniques and experience. When exhaustive search methods are impractical, heuristic methods are used to find efficient solutions.

The objective of a heuristic algorithm is to apply a rule to produce a solution in a reasonable time frame that is good enough for solving the problem at hand. There is no guarantee that the solution found will be the most accurate or optimal solution for the given problem. We often refer the solution as “good enough” in most cases.

- **Ways to Use Heuristics in Everyday Life:**

Here are some examples of real-life heuristics that people use as a way to solve a problem or to learn something:

- "Educated guess" is a heuristic that allows a person to reach a conclusion without exhaustive research. With an educated guess a person considers what they have observed in the past, and applies that history to a situation where a more definite answer has not yet been decided.

- "Working backward" allows a person to solve a problem by assuming that they have already solved it, and working backward in their minds to see how such a solution might have been reached.

6.2 Metaheuristic:

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.

Metaheuristics is a rather unfortunate term often used to describe a major subfield, indeed the primary subfield, of stochastic optimization. Stochastic optimization is the general class of algorithms and techniques which employ some degree of randomness to find optimal (or as optimal as possible) solutions to hard problems. Metaheuristics are the most general of these kinds of algorithms, and are applied to a very wide range of problems [2].

Metaheuristics are applied to I know it when I see it problems. They're algorithms used to find answers to problems when you have very little to help you: you don't know beforehand what the optimal solution looks like, you don't know how to go about finding it in a principled way, you have very little heuristic information to go on, and brute-force search is out of the question because the space is too large. But if you're given a candidate solution to your problem, you can test it and assess how good it is. That is, you know a good one when you see it [2].

6.2.1 Types of metaheuristics:

- **Local search metaheuristics**

Local search (LS), or iterative improvement, finds good solutions by iteratively making changes to a single solution, called the current (or incumbent) solution, applying a single move to a given solution is called the neighborhood of that solution, In each iteration, the current solution is replaced by the best solution a solution from its neighborhood. in which the best move from the neighborhood is selected, A solution that is better than any solution in its neighborhood is called a local optimum, When the current solution is a local optimum, a metaheuristic will use a strategy to "escape" this local optimum[3].

- **Constructive metaheuristics:**

Constructive metaheuristics, as their name suggests, construct solutions from their constituting elements rather than improving complete solutions. This is done by adding one element at a time to a partial solution. Constructive metaheuristics are often adaptations of

greedy algorithms that add the best possible element at each iteration. To improve the quality of the final solutions, most constructive metaheuristics include a local search phase after the construction phase[3].

- **Population-based metaheuristics:**

Evolutionary algorithms operate on a set or population of solutions and use two mechanisms to search for good solutions: the selection of predominantly high-quality solutions from the population and the recombination of those solutions into new ones, using specialized operators that combine the attributes of two or more solutions. After recombination, new solutions are reinserted into the population, possibly requiring them to satisfy conditions such as feasibility or minimum quality demands, to replace other (usually low-quality) solutions. Operators used in evolutionary algorithms (selection, recombination and reinsertion) almost without exception make heavy use of randomness. A mutation operator that randomly makes a (small) change to a solution after it has been recombined, is also frequently applied. Most evolutionary algorithms iterate the selection, recombination, mutation, and reinsertion phases a number of times, and report the best solution in the population. Evolutionary algorithms generally require some form of "population management" to ensure that the best solutions survive through the various iterations, while at the same time diversity is maintained in the population [3].

- **Hybrid metaheuristics**

In recent years, there is a tendency to view metaheuristic frameworks as providing general ideas or components that can be used to build optimization methods, As a result, most recent metaheuristic algorithms combine ideas from different classes and the term hybrid metaheuristic has lost most of its discriminatory power. Many modern metaheuristics use specialized heuristics to efficiently solve sub problems produced by the metaheuristic method. Similarly, a large number of local search metaheuristics use a construction phase to find an initial solution (or a set of initial solutions) from which to start the neighborhood search [3].

6.2.2 Classification of metaheuristics

- **Deterministic metaheuristic:**

A deterministic metaheuristic solves an optimization problem by making deterministic decisions. In deterministic algorithms, using the same initial solution will lead to the same final solution.

- **Stochastic metaheuristics:**

In stochastic metaheuristics, some random rules are applied during the search, in stochastic metaheuristics, different final solutions may be obtained from the same initial solution. And this section is divided in two types:

- Single solution-based method.
- Population-based approaches evolutionary computation, genetic algorithms, and particle swarm optimization which we will discuss later.

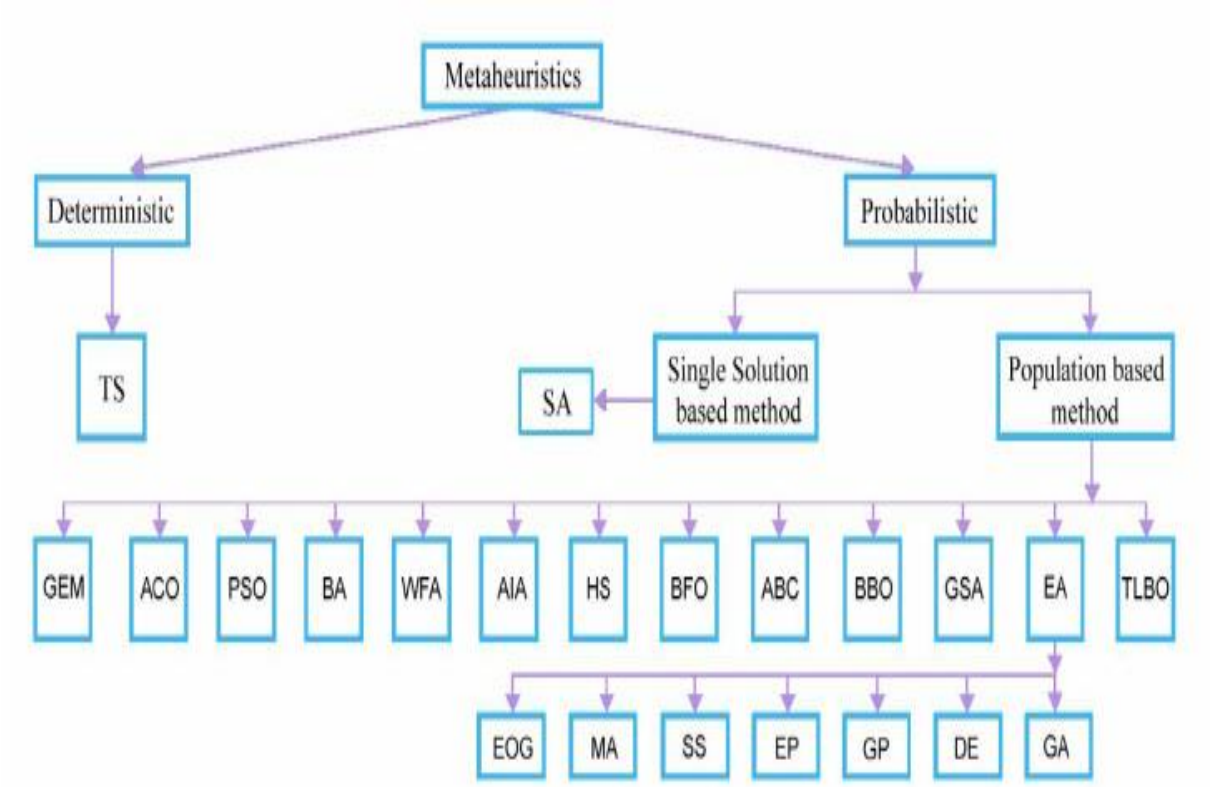


Figure 1.4: Classification of metaheuristics[d].

7. Bio inspired algorithms:

Bio inspired or evolutionary algorithms aim to mimic the biological evolution in a digital analogue of it. However, despite of being called evolutionary algorithms, they are more similar to artificial breeding– where several generations of solutions are combined together in order to grew something that is better. One can also say that evolutionary algorithm is a method to work on a problem that you do not know much about [e].

The following schema shows a brief history of the development of bio inspired algorithms:

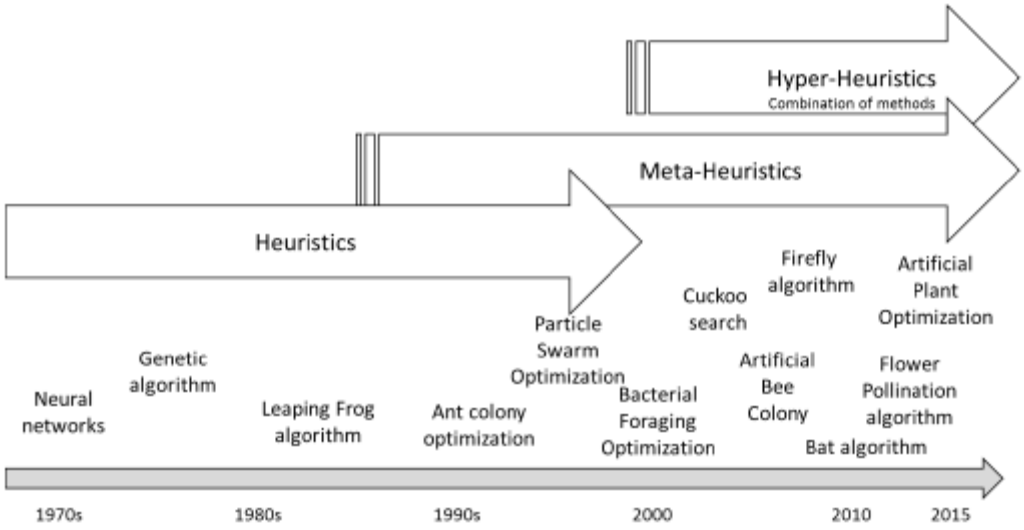


Figure 1.5: Development focus of bio-inspired algorithms [f]

7.1 Genetic Algorithms:

Genetic algorithms (GA) were invented by John Holland in the 1960s and were developed by Holland and his students and colleagues at the University of Michigan in the 1960s and the 1970s.

GA is a method for moving from one population of "chromosomes" to a new population by using a kind of "natural selection" together with the genetics−inspired operators of crossover, mutation, and inversion

The chromosomes in a GA population typically take the form of bit strings. Each locus in the chromosome has two possible alleles: 0 and 1. Each chromosome can be thought of as a point in the search space of candidate solutions. The GA processes populations requires a fitness function that assigns a score (fitness) to each chromosome in the current population. The fitness of a chromosome depends on how well that chromosome solves the problem at hand. [5]

- **The basic steps of Genetic algorithms [5]:**

The simplest form of genetic algorithm involves three types of operators: selection, crossover (single point), and mutation.

Selection this operator selects chromosomes in the population for reproduction. The fitter the chromosome, the more times it is likely to be selected to reproduce.

Selection types:

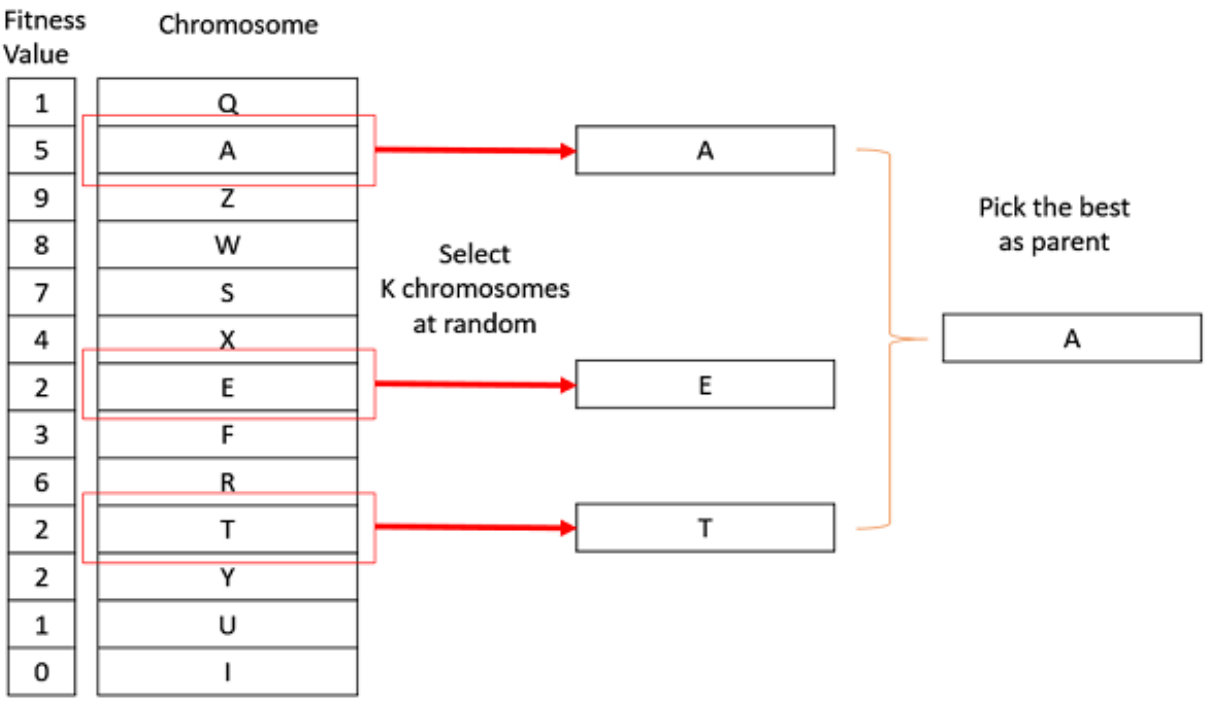
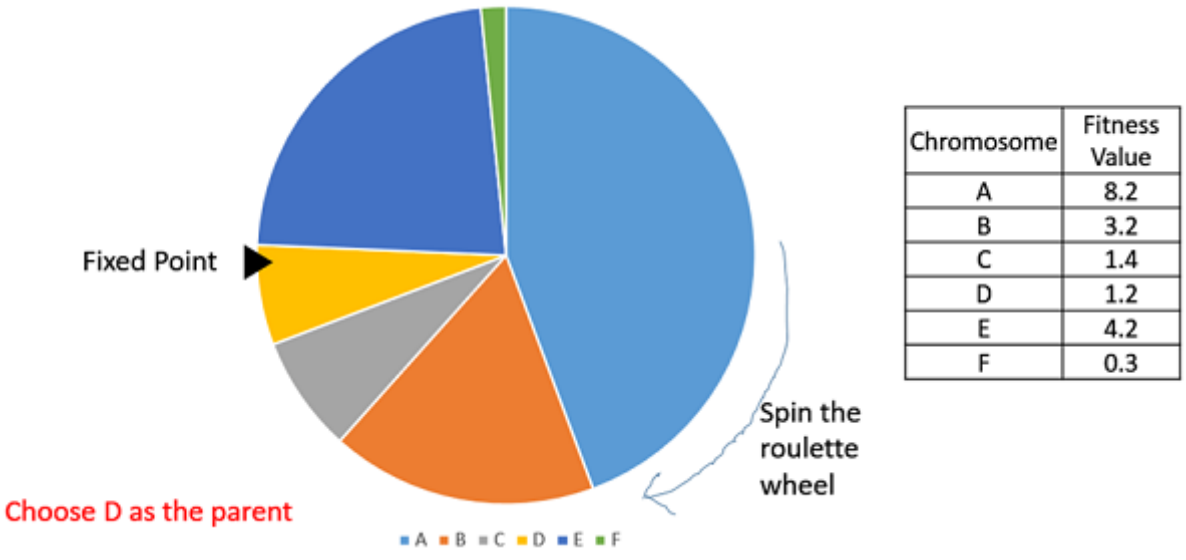


Figure 1.6: Tournament Selection [h]



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

Figure 1.7: Wheel Selection [h]

- **Crossover:**

This operator randomly chooses a locus and exchanges the subsequences before and after that locus between two chromosomes to create two offspring. For example, the strings 10000100 and 11111111 could be crossed over after the third locus in each to produce the two offspring 10011111 and 11100100. The crossover operator roughly mimics biological recombination between two single−chromosome (haploid) organisms.

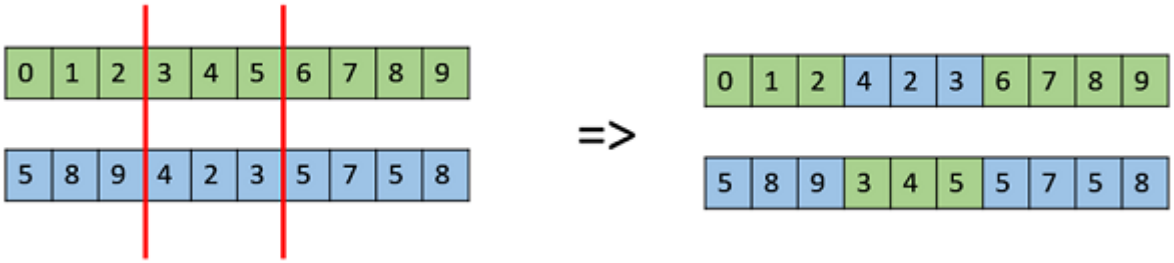


Figure 1.8- Multi Point Crossover [h]

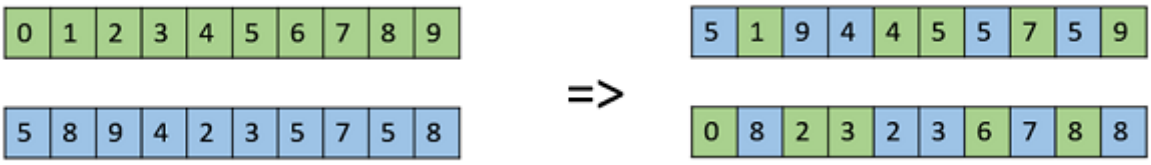


Figure 1.9- Uniform Crossover [h]

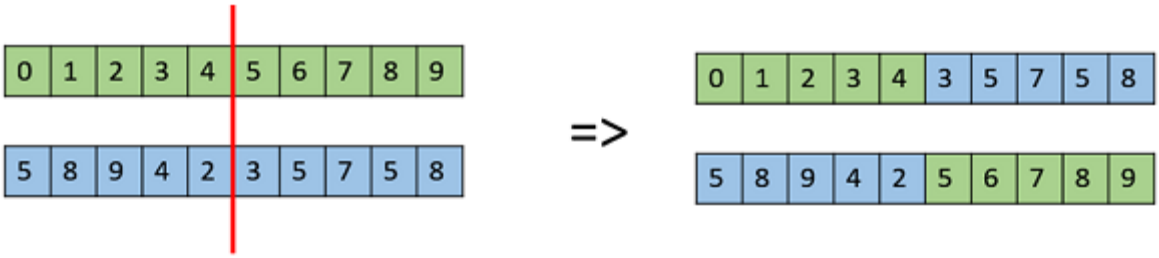


Figure 1.10- One Point Crossover [h]

- **Mutation**

This operator randomly flips some of the bits in a chromosome. For example, the string 00000100 might be mutated in its second position to yield 01000100. Mutation can occur at each bit position in a string with some probability, usually very small (e.g., 0.001).

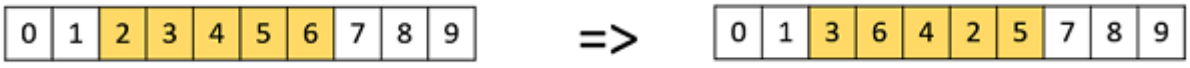


Figure 1.11- Scramble Mutation [h]

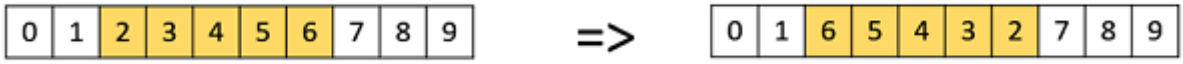


Figure 1.12- Inversion Mutation [h]

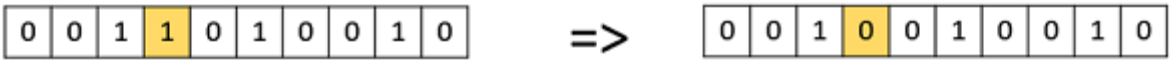


Figure 1.13- Bit Flip Mutation [h]

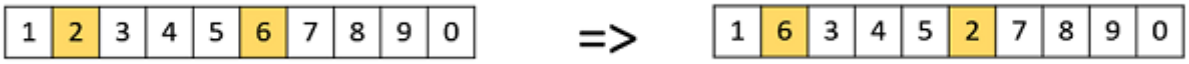


Figure 1.14: Swap Mutation [h]

- **Simple Genetic Algorithm [5]:**

Given a clearly defined problem to be solved and a bit string representation for candidate solutions, a simple

GA works as follows:

1. Start with a randomly generated population of an l−bit chromosome (candidate solutions to a problem).
2. Calculate the fitness $f(x)$ of each chromosome x in the population.
3. Repeat the following steps until n offspring have been created:
   a. Select a pair of parent chromosomes from the current population, the probability of selection being an increasing function of fitness. Selection is done "with replacement," meaning that the same chromosome can be selected more than once to become a parent.
   b. With probability pc (the "crossover probability" or "crossover rate"), cross over the pair at a randomly chosen point (chosen with uniform probability) to form two offspring. If no crossover takes place, form two offspring that are exact copies of their respective parents.
   (Note that here the crossover rate is defined to be the probability that two parents will cross over in a single point. There are also "multi−point crossover" versions of the GA in which the crossover rate for a pair of parents is the number of points at which a crossover takes place.)

c. Mutate the two offspring at each locus with probability pm (the mutation probability or mutation rate), and place the resulting chromosomes in the new population.

If n is odd, one new population member can be discarded at random.

4. Replace the current population with the new population.
5. Go to step 2.

7.2 Ant colony optimization (ACO)

(ACO) is a population-based metaheuristic that can be used to find approximate solutions to difficult optimization problems.

In ACO, a set of software agents called artificial ants search for good solutions to a given optimization problem. To apply ACO, the optimization problem is transformed into the problem of finding the best path on a graph. The artificial ants (hereafter ants) incrementally build solutions by moving on the graph. The solution construction process is stochastic and is biased by a pheromone model, that is, a set of parameters associated with graph components (either nodes or edges) whose values are modified at runtime by the ants.

- **Steps of ACO [6]:**

The ACO algorithm is essentially based on three procedures ConstructAntsSolutions, update Pheromones, and DeamonActions, as represented by Algorithm 1.

1 ConstructAntsSolutions is the process by which artificial ants construct walks on the construction graph incrementally and stochastically. For a given ant, the probability pal to go from a node k to a feasible successor node l is an increasing function of kl and kl (u), where kl is the pheromone on arc (k, l), and kl (u) is the heuristic value of arc (k, l), which should be a reasonable guess of how good arc (k, l) is.

2 Update Pheromones is the process by which pheromone is modified on arcs. Pheromone may be both increased and decreased. Pheromone is modified (decreased) by each ant on each arc as soon as it is added to a partial walk on the construction graph, this operation is called local update. Moreover, pheromone is further modified (increased) on selected good solutions to more strongly bias the search in future iterations, and this operation is called global update. Decreasing pheromone on selected arcs is important, in order to avoid too rapid convergence of the algorithm to suboptimal solutions. Interestingly, pheromone decreases also in the biological environment, due to evaporation.

3 DeamonActions are centralized operations, such as comparing solution values among ants in order to find the best solution, or running a local search procedure.

While termination condition not met, do
ScheduleActivities
ConstructAntsSolutions
UpdatePheromone
DeamonActions
End Schedule Activities
End while

Figure 1.15: Algorithm Ant Colony Optimization (ACO)

7.3 Particle Swarm Optimization PSO:

PSO is a swarm intelligence meta-heuristic inspired by the group behavior of animals, for example bird flocks or fish schools. Similarly to genetic algorithms (GAs), it is a population-based method, that is, it represents the state of the algorithm by a population, which is iteratively modified until a termination criterion is satisfied. In PSO algorithms, the population $P = \{p1, \dots, pn\}$ of the feasible solutions is often called a swarm. The feasible solutions $p1, \dots, pn$ are called particles. The PSO method views the set Rd of feasible solutions as a "space" where the particles "move". For solving practical problems. [7]

PSO has been proposed by Eberhart and Kennedy in 1995, subsequently developed in thousands of scientific papers, and applied to many diverse problems, for instance neural networks training, data mining, signal processing, and optimal design of experiments. [7]

- **Characteristics of PSO :**
  - Xi: The position.
  - Vi: The speed of particle.
  - Pb: The current best position.
  - Pg: The entire swarm best position.
  - $w(t)$ … inertia weight; a damping factor, usually decreasing from around 0.9 to around 0.4 during the computation

- $\varphi 1, \varphi 2$ acceleration coefficients

Update the speed:

$$vi(t+1) = w(t)vi(t) + \varphi 1u1(pi(t) - xi(t)) + \varphi 2u2(li(t) - xi(t)).$$

Update the position :

$$xi(t+1) = xi(t) + vi(t+1).$$

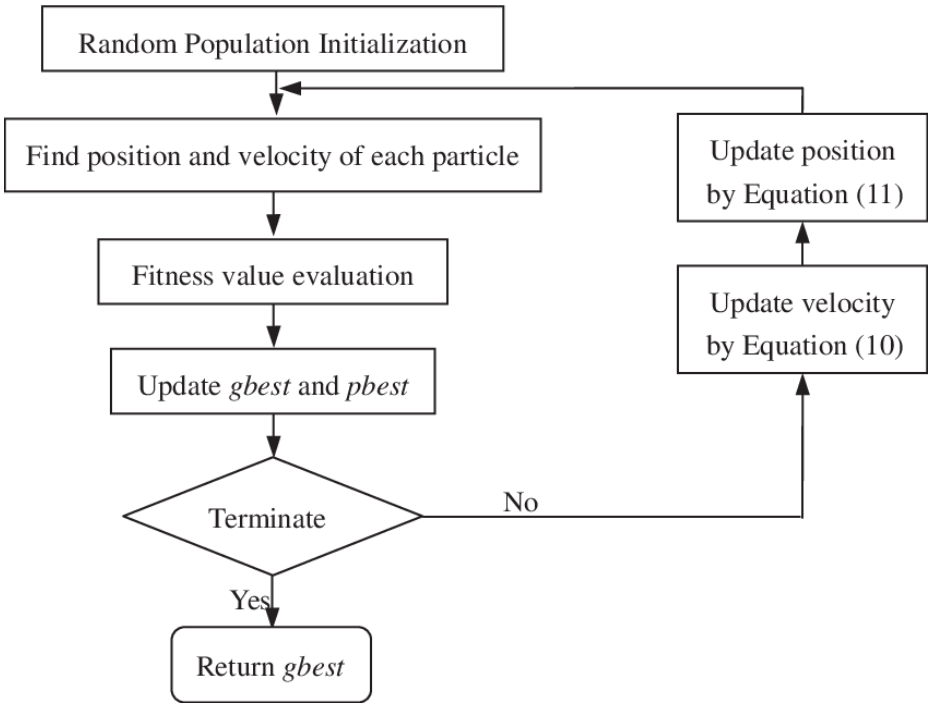


Figure 1.16: The PSO Flow Chart[g]

Different algorithms are derivative from PSO such as Cuckoo search, Firefly, and Bat algorithm.

7.4 Harmony Search (HS)

In order to explain the Harmony Search in more detail, let us first idealize the improvisation process by a skilled musician. When a musician is improvising, he or she has three possible choices: (1) play any famous piece of music (a series of pitches in harmony) exactly from his or her memory; (2) play something similar to a known piece (thus adjusting the pitch slightly); or (3) compose new or random notes. Zong Woo Geem et al. formalized these three options into quantitative optimization process in 2001, and the three corresponding components become: usage of harmony memory, pitch adjusting, and randomization [8].

The usage of harmony memory is important, as it is similar to the choice of the best-fit individuals in genetic algorithms (GA). This will ensure that the best harmonies will be carried over to the new harmony memory. In order to use this memory more effectively, it is typically assigned as a parameter raccept Î[0,1], called harmony memory accepting or considering rate. If this rate is too low, only few best harmonies are selected and it may converge too slowly. If this rate is extremely high (near 1), almost all the harmonies are used in the harmony memory, then other harmonies are not explored well, leading to potentially wrong solutions. Therefore, typically, we use raccept=0.7 ~ 0.95[9].

The second component is the pitch adjustment determined by a pitch bandwidth brange and a pitch adjusting rate rpa. Though in music, pitch adjustment means to change the frequencies, it corresponds to generate a slightly different solution in the Harmony Search algorithm [1]. In theory, the pitch can be adjusted linearly or nonlinearly, but in practice, linear adjustment is used. So we have

$$X_{new}=X_{old}+b_{range} * \varepsilon$$

Where x old is the existing pitch or solution from the harmony memory, and x_{new} is the new pitch after the pitch adjusting action. This essentially produces a new solution around the existing quality solution by varying the pitch slightly by a small random amount [1,2]. Here ε is a random number generator in the range of [-1,1]. Pitch adjustment is similar to the mutation operator in genetic algorithms. We can assign a pitch-adjusting rate (rpa) to control the degree of the adjustment. A low pitch adjusting rate with a narrow bandwidth can slow down the convergence of HS because the limitation in the exploration of only a small subspace of the whole search space. On the other hand, a very high pitch-adjusting rate with a wide bandwidth may cause the solution to scatter around some potential optima as in a random search. Thus, we usually use rpa=0.1 ~0.5 in most applications[9].

Harmony Search
begin
Objective function f(x), x=(x1,x2, ...,xd)T
Generate initial harmonics (real number arrays)
Define pitch adjusting rate (rpa), pitch limits and bandwidth
Define harmony memory accepting rate (raccept)
while (t<Max number of iterations)
Generate new harmonics by accepting best harmonics
Adjust pitch to get new harmonics (solutions)
if (rand>raccept), choose an existing harmonic randomly
else if (rand>rpa), adjust the pitch randomly within limits
else generate new harmonics via randomization
end if
Accept the new harmonics (solutions) if better
end while
Find the current best solutions
end

Figure 1.17: Pseudo code of the Harmony Search algorithm [9].

7.5 BAT Algorithm

In 2010, Yang developed a bio-inspired algorithm called bat algorithm (BA) based on the echolocation behavior of bats. The echolocation capability of the micro-bats is fascinating as these bats can find their prey and discriminate different types of insects even in complete darkness. [10]

- **-The Echolocation Characteristics of micro-bats[10]**

  1. All bats use echolocation to sense distance, and they also 'know' the difference between food/prey and background barriers in some magical way;

  2. Bats fly randomly with velocity vi at position xi with a fixed frequency fmin, varying wavelength _ and loudness $A0$ to search for prey. They can automatically adjust the wavelength (or frequency) of their emitted pulses and adjust the rate of pulse emission r 2 $[0, 1]$, depending on the proximity of their target.

3. Although the loudness can vary in many ways, we assume that the loudness varies from a large (positive) $A0$ to a minimum constant value Amin.

- **Movement of Bats:**

1. Update frequency: $f_i = f_{min} + (f_{max} - f_{min})\,\beta$

2. Update speed: $v_i^t = v_i^{t-1} + (x_i^t - x')f_i$

3. Update positions: $x_i^t = x_i^{t-1} + v_i^t$

Objective function $f(x), x = (x1, \ldots, xd)T$
Initialize the bat population xi $(i = 1, 2, \ldots, n)$ *and* vi
Define pulse frequency fi *at* xi
Initialize pulse rates ri *and the loudness* Ai
$while\ (t < Max\ number\ of\ iterations)$
Generate new solutions by adjusting frequency, and updating velocities and locations/solutions [equations (1) to (3)]
$if\ (rand > ri)$
Select a solution among the best solutions
Generate a local solution around the selected best solution
$end\ if$
Generate a new solution by flying randomly
$if\ (rand < Ai\ \&\ f(xi) < f(x'))$
Accept the new solutions
Increase ri *and reduce* Ai
$end\ if$
Rank the bats and find the current best x_i
$end\ while$

Figure 1.18: Pseudo code of the bat algorithm (BA) **[10]**.

8. Conclusion:

In this chapter we have introduce the domain of combinatorial optimization and its importance in all fields of solving hard problems , next we will represent the Graph theory and we will try to give you brief overview on its concepts and how it have a lot of uses in all fields.

Chapter 2
Graph Theory

1. INTRODUCTION

Many world problems and situations are represented as points and links joining certain points, it used to describe relations (distances, flow, amount, . ..) or a shared things between this points depending on what the points are represented by, if we said for example that this points is Facebook users and the links describes friends or a book they booth likes that could make it easier to big companies and researchers to search or to detect this shared links that's why graph theory term has risen.

One of the usages of graph theory is to give a unified formalism for many very different looking problems. It then suffices to present algorithms in this common formalism. This has led to the birth of a special class of algorithms, the so-called graph algorithms. Half of the text of these notes deals with graph algorithms, again putting emphasis on network-theoretic methods. Only basic algorithms, applicable to problems of moderate size, are treated here. [11]

2. Definition and Fundamental Concepts

In this section we will introduce and explain some terms in graph theory.

2.1 Graph

Conceptually, a graph is formed by vertices and edges connecting the vertices.

Formally, a graph is a pair of sets(V, E), where V is the set of vertices and E is the set of edges, formed by pairs of vertices. E is a multiset, in other words, its elements can occur more than once so that every element has a multiplicity. Often, we label the vertices with letters (for example: a, b, c, or v1, v2 . . .) or numbers 1, 2 . . . Throughout this lecture material, we will label the elements of V in this way. [11]

Vertices: nodes or points in the graph.

Edge: an edge is a path (bridge, line) between two regions or a relation between two objects.

2.2 Graph Representations:

Here we going to see some of the important representations of graphs.

- Adjacency Matric:

Adjacency Matrix is a 2D array of size $V \times V$ where V is the number of vertices in a graph. Let the 2D array be adj [ ] [ ], a slot $adj[i]$ $[j] = 1$ indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If $adj[i]$ $[j] = w$, then there is an edge from vertex i to vertex j with weight w.

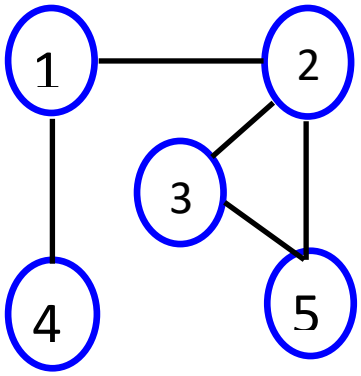
- Incidence Matrix:

The rows of the matrix $[AC]$ represent the number of nodes and the column of the matrix $[AC]$ represent the number of branches in the given graph. If there are 'n' number of rows in a given incidence matrix, that means in a graph there are 'n' number of nodes. Similarly, if there are 'm' number of columns in that given incidence matrix, that means in that graph there are 'm' number of branches.

- Adjacency List:

An array of lists is used. Size of the array is equal to the number of vertices. Let the array be $array$ [ ]. An entry $array[i]$ represents the list of vertices adjacent to the ith vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is adjacency list representation of the above graph.

The graph can be represented with three ways as a drawing representation or as adjacency matrix representation or as an adjacency list representation as shown in the next page:



v	1 2 3 4 5
1	0 1 0 1 0
2	1 0 1 0 1
3	0 1 0 0 1
4	1 0 0 0 0
5	0 1 1 0 0

v	Adj_v
1	2,4
2	1,3,5
3	2,5
4	1
5	2,3

Figure 2.1: Graph representation

2.3 Terminologies

We have the following terminologies [11]:

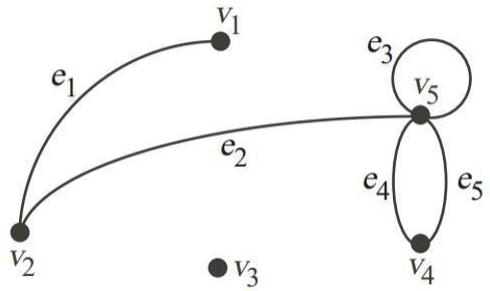


Figure 2.2: Graph with label edges

- The two vertices u and v are end vertices of the edge(u, v): ($v4$ and $v5$ are end vertices of $e5$.)
- Edges that have the same end vertices are parallel: ($e4$ and $e5$ are parallel.)
- An edge of the form (v, v) is a loop: ($e3$ is a loop.)
- A graph is simple if it has no parallel edges or loops: (The graph is not simple.)
- A graph with no edges (i.e. E is empty) is empty.
- A graph with no vertices (i.e. V and E are empty) is a null graph
- A graph with only one vertex is trivial.
- Edges are adjacent (neighbors) if they share a common end vertex: ($e1$ and $e2$ are adjacent.)
- Two vertices u and v are adjacent if they are connected by an edge, in other words, (u, v) is an edge: ($v1$ and $v2$ are adjacent.)
- The degree of the vertex v, written as d (v), is the number of edges with v as an end vertex.
- By convention, we count a loop twice and parallel edges contribute separately.
- A pendant vertex is a vertex whose degree is 1: (The degree of $v1$ is 1 so it is a pendant vertex.)
- An edge that has a pendant vertex as an end vertex is a pendant edge: ($e1$ is a pendant edge).

Degrees of vertices: Let $G = (V, E, \varphi)$ be a graph and $v \in V$ a vertex. The degree of v, d (v) is the number of $e \in E$ such that $v \in \varphi(e)$; i.e., e is incident on v. According to figure 2.2 The degree of $v2$ is 2 because $v2$ has two neighbors.

2.4 Graph types

There are several types in the graph representation we will mention some of them in this section.

- Simple graph

  A simple graph is the undirected graph with no parallel edges and no loops.

  A simple graph which has n vertices, the degree of every vertex is at most n -1.

- Directed graph

A directed graph is a graph in which the edges are directed by arrows.

Directed graph is also known as digraphs.

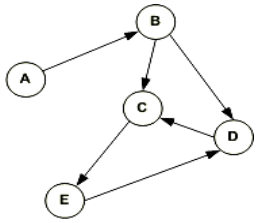


Figure2.3: directed graph

- Undirected graph

An undirected graph is a graph whose edges are not directed.

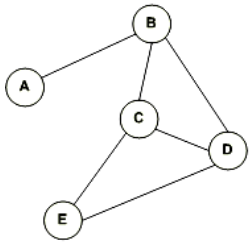


Figure 2.4: Undirected graph

- Bipartite graph

A bipartite graph is a graph in which the vertex set can be partitioned into two sets such that edges only go between sets, not within them.

A graph $G\ (V, E)$ is called bipartite graph if its vertex-set $V\ (G)$ can be decomposed into two non-empty disjoint subsets $V1\ (G)$ and $V2\ (G)$ in such a way that each edge $e \in E\ (G)$ has its one last joint in $V1\ (G)$ and other last point in $V2\ (G)$.

The partition $V = V1 \cup V2$ is known as bipartition of G.

- Complete graph

A graph in which every pair of vertices is joined by exactly one edge is called complete graph. It contains all possible edges.

A complete graph with n vertices contains exactly nC2 edges and is represented by Kn.

- Weighted Graph

A weighted graph is a graph whose edges have been labeled with some weights or numbers.

The length of a path in a weighted graph is the sum of the weights of all the edges in the path.

3. Applications Of Graph [12]

Graph theoretical concepts are widely used to study and model various applications, in different areas.

They include, study of molecules, construction of bonds in chemistry and the study of atoms. Similarly, graph theory is used in sociology for example to measure actor's prestige or to explore diffusion mechanisms. Graph theory is used in biology and conservation efforts where a vertex represents regions where certain species exist and the edges represent migration path or movement between the regions. This information is important when looking at breeding patterns or tracking the spread of disease, parasites and to study the impact of migration that affect other species.

Graph theoretical concepts are widely used in Operations Research. For example, the traveling salesman problem, the shortest spanning tree in a weighted graph, obtaining an optimal match of jobs and men and locating the shortest path between two vertices in a graph. It is also used in modeling transport networks, activity networks and theory of games. The network activity is used to solve large number of combinatorial problems. The most popular and successful applications of networks in OR is the planning and scheduling of large complicated projects. The best well known problems are PERT (Project Evaluation Review Technique) and CPM (Critical Path Method). Next,

Game theory is applied to the problems in engineering, economics and war science to find optimal way to perform certain tasks in competitive environments. To represent the method of finite game a digraph is used. Here, the vertices represent the positions and the edges represent the moves.

3.1 Example of uses of graphs in networks

As networks began to grow, it became very difficult to manually manage every connection between sites. Configuring routers to direct traffic to a specific location was taking too long, it needed to be dynamically finding its shortest path, Dijkstra's algorithm used later to calculate the bandwidth between routers, a lot of algorithms of graphs used on this problems and we see today how it is a complex operation but we always get what we want in seconds.

4. Algorithms of Graph

Here we going to introduce to you some popular algorithms of graph

4.1 Depth-First and Breadth-First Searches

Breadth First Search (BFS) is the traversing method used in graphs. It uses a queue for storing the visited vertices. In this method the emphasize is on the vertices of the graph, one vertex is selected at first then it is visited and marked. The vertices adjacent to the visited vertex are then visited and stored in the queue sequentially. Similarly, the stored vertices are then treated one by one, and their adjacent vertices are visited. A node is fully explored before visiting any other node in the graph, in other words, it traverses shallowest unexplored nodes first.

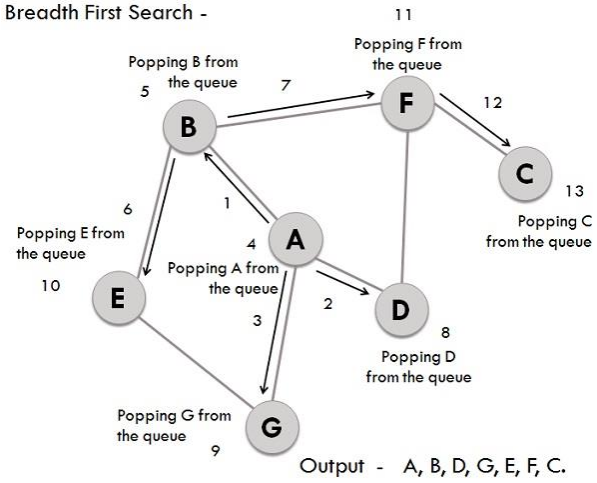


Figure 2.5: BFS[i]

Depth First Search (DFS) traversing method uses the stack for storing the visited vertices. DFS is the edge based method and works in the recursive fashion where the vertices are explored along a path (edge). The exploration of a node is suspended as soon as another unexplored node is found and the deepest unexplored nodes are traversed at foremost. DFS traverse/visit each vertex exactly once and each edge is inspected exactly twice.

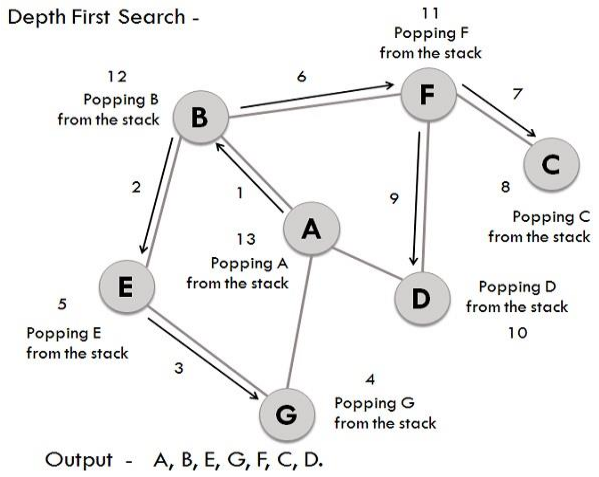


Figure 2.6: DFS [i]

4.2 Dijkstra's algorithm[13]

Dijkstra's algorithm is a graph search algorithm that solves the single source shortest path problem for a graph with non-negative edge cost path costs, producing a shortest path tree. Conceived by edsger Dijkstra

Let the node at which we are starting be called the initial node. Let the distance of node Y be the distance from the initial node to Y. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

- Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
- Mark all nodes unvisited. Set the initial node as current. Create a set of the unvisited nodes called the unvisited set consisting of all the nodes.
- For the current node, consider all of its unvisited neighbors and calculate their tentative distances. For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbor B has length 2, then the distance to B (through A) will be 6 + 2 = 8.
- When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the unvisited set. A visited node Will Never be checked again.
- If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the unvisited set is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.

- Select the unvisited node that is marked with the smallest tentative distance, and set it as the new "current node" then go back to step 3.

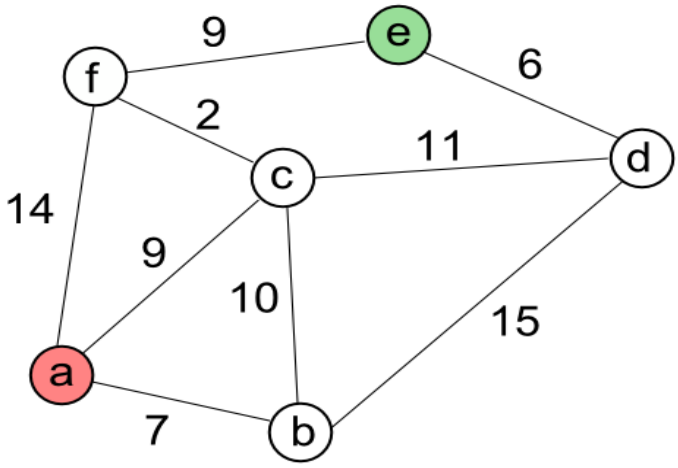


Figure 2.7: Dijkstra shortest path [j]

5. Common Problems in Graphs

Here we going to see some popular problems in graph

5.1 Minimum Dominating Set problem (MWDS)

Given an undirected graph $G = (V, E)$ where V is the set of vertices and E is the set of edges, a dominating set is a subset $D \subseteq V$ such that every vertex $v \in V$ is either a member of D or is adjacent to a vertex in D. A dominating set D is called the minimum dominating set if its cardinality is minimum of all possible dominating sets. [13]

- **Applications of the MWDS**

The dominating set plays an important role in computer and communication networks to route the information between the nodes. For example, wireless network. A wireless network is a type of computer network that uses wireless data transmission by connecting the wireless network nodes. The examples of wireless network include cell-phone network, mobile Ad-hoc network, wireless sensor network etc. [14]

5.2 Vertex cover problem:

Vertex cover is one of the important known covering problems, a vertex cover in a graph is a set of vertices such that every edge in the graph is incident to at least one vertices in the set, the problem of finding a minimum vertex cover is a classical optimization problem in computer science and is a typical example of an NP-hard optimization problem that has an approximation algorithm. Its decision version, the vertex cover problem, was one of Karp's 21 NP-complete

problems and is therefore a classical NP-complete problem in computational complexity theory. Furthermore, the vertex cover problem is fixed-parameter tractable and a central problem in parameterized complexity theory.

We going to explain this problem more in the next chapter.

5.3 Clique problem [11]:

A clique in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E. In other words, a clique is a complete subgraph of G. The size of a clique is the number of vertices it contains. The Clique problem is the optimization problem of finding a clique of maximum size in a graph.

As a decision problem, we ask simply whether a clique of a given size k exists in the graph.

Instance: a graph $G = (V, E)$ and a positive integer $k \leq |V|$.

Question: is there a clique $V' \subseteq V$ of size $\geq k$?

6. Conclusion

In this chapter, we have represented the importance of the graph theory and how it can used to represent a lot of problems. In the next chapter, we will introduce minimum vertex cover one of the Karp's problems, and we will propose to solve this problem by a heuristic and a meta-heuristic method.

Chapter 3
Minimum Vertex Cover

1. Introduction

Vertex cover problem is a NP-complete problem. If a problem is NP-complete, we are unlikely to find polynomial-time algorithm for solving it exactly, but this does not simply that all hope is lost. There are two approaches to getting around NP-completeness.

First if the actual inputs are small, an algorithm with exponential running time may be perfectly satisfactory. Second, it may still possible to find near optimal solutions in polynomial time. In practice near optimality is often good enough. An algorithm that returns near-optimal solutions is called an approximation algorithm.

In computer science, the vertex cover problem or node cover problem is one of Karp's 21 NP-complete problems. It is often used in complexity theory to prove NP-hardness of more complicated problems.

2. Minimum Vertex Cover Problem

Consider a graph $G = (V, E)$ where V and E are accordingly vertex and edges. A Vertex cover of an undirected graph is a subset $V' \subseteq V$ such that if (u, v) is an edge of G, Then either $u \in V' or v \in V'$ or both.

The size of a vertex cover is the number of vertices in it. The vertex cover problem is to find a vertex cover of minimum size in a given undirected graph. Such a vertex cover is called an optimal vertex cover.

A minimum vertex cover is a vertex cover of smallest possible size, the number of vertices of a graph is equal to its minimum vertex cover number plus the size of a maximum independent set.

A minimum vertex cover is a vertex cover having the smallest possible number of vertices for a given graph. The size of a minimum vertex cover of a graph *G* is known as the vertex cover number .

As an example, we consider Wireless Sensor Network as an undirected graph G = (V, E) where V represents the set of vertices and E represents the set of edges. The degree of a vertex v, d (v), is the number of edges incident to v. The maximum degree of *G* is denoted by d. The input to the program is an n×n adjacency matrix. If there is an edge between u and v the adjacency matrix is set to 1 otherwise set to 0[16].

Vertex Cover C of G is defined as the set of vertices which contains at least one end vertex of every edge. A vertex v is removable from C if and only if its removal still retains the vertex cover set C of G. the number of removable vertices of a vertex cover C is denoted by ρ(C). A

minimal vertex cover is a vertex cover with the less number of vertices that has no removable vertices [k].

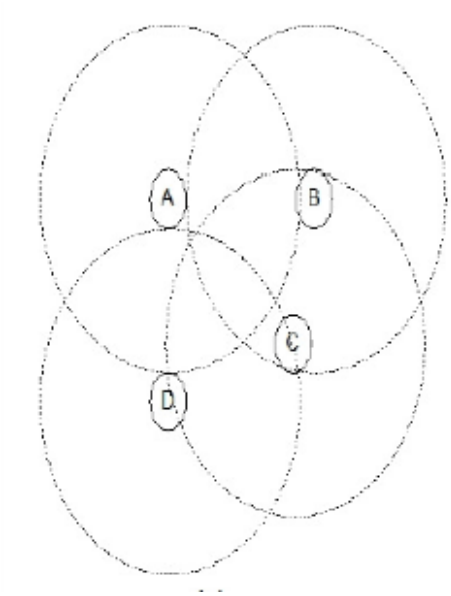


Figure. 3.1: Wireless sensor network [16]

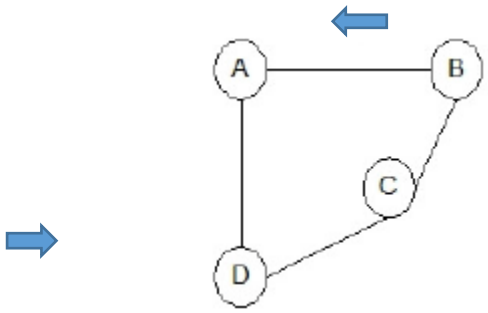


Fig. 3.2: Vertex cover [16]

By using above algorithm, we get the minimal vertex covers of size k. We can make these vertex covers as cluster heads in a wireless sensor networks [16].

3. objective of our work

To solve a vertex cover problem, we need to use either a heuristic or a metaheuristic to obtain an optimal solution in a polynomial time. So, in this work we aim to try to solve MVCP and make a comparison between heuristic and metaheuristics. for heuristics we use genetic algorithm and for metaheuristics we use bat algorithm. In the following section, we will describe in details the basic steps of the two approaches. Also, we have tried to solve MVCP by a hybrid-metaheuristic which combine two meta-heuristics: the harmony search and the bat algorithm. This will be described further

3.1 Genetic algorithm implementation:

Genetic algorithm is a well-known heuristic. We have chosen it for several reasons:

- Most NP hard problems are earlier solved by GA.
- It’s a population-based heuristic. Which is the origin of all bio-inspired algorithms.
- It’s a simple algorithm with simple basic steps.

The population is a group of individuals. Each individual is a potential solution. Each one is represented as binary string that has length equals the number of nodes in the targeted graph. The bit that has a value of 1 means that this node is included in the vertex cover set. Whilst, the node of value 0 means that node is not.

The evaluation of the solution is based on the properties of MVC, the solution must contain the minimum set of nodes and cover the maximum number of nodes. To solve this equation, we must combine of each solution another binary string that has also the length of the given graph where 1 means that the node is cover and 0 otherwise. According to this we have fitness function:

$$f(x) = \sum_{i=1}^{n} x_i + \text{uncovered}(x)$$

Where n is the number of nodes (the size of the solution), and the first term of the equation implies the minimum vertex cover size. While, the last part of the function represents the nodes that reside out of the minimum set coverage.

To ensure the best execution of GA, we have the following parameters:

- iteration number presents the number of generations
- pc presents the probability to have a crossover
- pm presents the probability to have a mutation

the basic steps of GA are described and illustrated by our Java coding:

1- Initialization: a random set of solution is created.

```
        static List<List<Integer>> minvtx = new ArrayList<List<Integer>>();

    public static void initial() {
    for(int k=0;k<gensize;k++){

    List<Integer> l = new ArrayList<Integer>() ;
    Random rand = new Random();


       for (int j = 0; j<countlines; j++)
           {
            int pick = rand.nextInt(2);
            l.add(pick);

            }
       minvtx.add(l);

}
    }
```

Figure 3.3: initialization of GA

2-calculate the fitness of all the population

```
                static List<List<Integer>> vtx ;
static List<Integer> lb;
                public static void evalua(){
                     List<Integer> la ;
                      lb= new ArrayList<Integer>();
                                          List<Integer> lm ;
                      vtx = new ArrayList<List<Integer>>();
                      fitness=new double [gensize];
                      pool=new double [gensize];
          for(int m=0;m<gensize;m++){
              la = new ArrayList<Integer>();
              lm = new ArrayList<Integer>();
           for (int j = 0; j<minvtx.get(m).size(); j++)
          {
              if(minvtx.get(m).get(j)==1){
                  la.add(j);
                  lm.add(j);
                  for (int i = 0; i<adjacencyList.get(j).size();i++){
                  la.add(adjacencyList.get(j).get(i));}
                   o++;
              }              }
           vtx.add(lm);
           fitness[m]=f(o);
            lb = la.stream().distinct().collect(Collectors.toList());
            allvtx=lb.size();
            pool[m]=f(allvtx);
            o=0;
            allvtx=0;
          }           }
```

Figure 3.4: Fitness function

Each individual is evaluated according to our fitness solution, the neighbor is the way to ensure that the maximum number of nodes are covered. The procedure will stop when we cover all the nodes of the graph with the minimum vertex cover.

3-selection: the individuals with the best fitness are selected and stored in a list

```
    static List<List<Integer>> selct ;
    public static void select() {
        selct = new ArrayList<List<Integer>>();
            int ind1,ind2;
            for (int i=0;i<gensize;i++)
            {
             ind1=(int)(Math.random()*(minvtx.size()-1));
             ind2=(int)(Math.random()*(minvtx.size()-1));
            if( pool[i]>=0.8 ){
                good++;
                selct.add(minvtx.get(i));
            }
            else
            selct.add(minvtx.get((int)Math.random()*minvtx.size()));

            }

    }
```

Figure 3.5: Selection

4- The crossover: Knowing that we are already have chosen the most of highest individuals we do the crossover randomly on these individuals

```
public static void croisement() {
        double r=Math.random();
        for (int i=0;i<selct.size();i=i+2)
        if (r<=pc)
        {
            int ind=0;
            for(int j=0;j<selct.get(i).size()/2;j++)
            {
               // System.out.println("    " + selct.size());
                ind=selct.get(i).get(j);
                selct.get(i).set(j,selct.get(i+1).get(j));
                selct.get(i+1).set(j,ind);

        }}
}
```

Figure 3.6: Crossover

5-Mutation: If the random number r is less than pm we pick a random individual and check his fitness if it's good we do only a small random change if it's not we do several random changes

```
public static void mutations() {
double r=Math.random();
for (int i=0;i<selct.size();i++){
if (r<=pm)

        {
            //System.out.println("    " + selct.size());
            int ind=(int)Math.random()*(selct.size()-1);
            if(selct.get(i).get(ind)==1){
                selct.get(i).set(ind,0);
            }
            else
               selct.get(i).set(ind,1);

        }

}
    minvtx=selct;
}
```

Figure 3.7: Mutation

This procedure will be executed to create a new population that contains the best solutions founds so far, the algorithm will stop when we obtain the solution that cover all the graph with the minimum set cover possible.

3.2 Bat algorithm implementation:

Since the bat algorithm is bio-inspired an evolutionary one, it's based on the population solution. So, we keep the same presentation of the GA algorithm. The bat algorithm parameters are:

- iteration MAX: number presents the number of generations
- X the position each bat will have a position which is a binary individual
- V the velocity each bat will have a velocity which is set of random number between 0 and 1, this velocity will move the bat to a new position.
- F the frequency each individual will have his frequency which shows us if he is close to solution or far so then we will decide to do a number of changes on this individual, this number get bigger if the solution is too far and smaller if the solution is too close
- R the pulse emission
- A the loudness

The important steps of the bat algorithm are:

- ✓ Initialize the frequency and velocity as 0, and positions randomly
- ✓ find best solution of the first generation and store it.
- ✓ the fitness function: same as the GA one according to the MVC problem
- ✓ update position and velocity and frequency
- ✓ check if the solution improves and update the best solution of all generation if it improves

```
BAT algorithm
begin
   Objective function f(x), x=(x1,x2, ...,xd)T
   Generate initial Bats (binary number arrays)
   Define A_min and A_max the loudness
   Define R_min and R_max the pulse emission
   Initialize frequency, and velocities
   Generate  locations/solutions(population)
   Store the fittest solution b
   while ( t<Max number of iterations )
     Generate new solutions by adjusting frequency, and
 updating velocities and locations/solutions
   if (rand > R)
   Update solution  X according to its frequency
   End if
   if (f(X) <f(b)) accept the new solution b=X
   end while
   Print b
end
```

Figure 3.8: *Pseudo code of our Bat algorithm*

3.3 Hybrid Harmony-Bats Search implementation:

Harmony search algorithm is a meta-heuristic that seek for the best solution by selection one randomly, evaluate it and compare it with best solution find so far. Our hybrid algorithm will apply this process on solution that are originally a bat-population. So, a bat-population is created by the bat algorithm and be evaluated by the harmony search one.

This hybrid algorithm uses as parameters:

R_MEMO: this rate defines the probability of choosing an individual that already exists in Bat population.

R_Pitch: this rate defines the probability to take an individual that already chosen by R_memo rate and adjust it.

R_Rand: defines the probability to generate a new individual and replace it in place of an individual that have not been chosen by R_MEMO.

Harmon- bats search
begin
*Objective function f(**x**), **x**=(x1,x2, …,xd)T*
Generate initial Bats (binary number arrays)
Define pitch adjusting rate (rpa),
Define harmony memory accepting rate (R_memo)
Define bat parameters (same as previous one)
***while** (t<Max number of iterations)*
Generate new solutions by adjusting frequency, and updating velocities and locations/solutions
$if\ (rand > ri)$
Update solution B according to its frequency
End if
***if** (rand>R_memo), choose an existing X randomly*
***if** (rand>R_pitch), adjust the X randomly put it as H*
end if
***else if**(rand>R_rand),Generate a random H*
end if
Accept the best solution among H and S
Replace X if the previous best is better than X
end while
Print the global best solution
end

Figure 3.9: Pseudo code of our Harmony Bats algorithm

4. Conclusion:

In this chapter, we have introduced the implementation of the genetic algorithm, bat algorithms and the hybrid algorithm harmony-bats algorithm.in the next chapter we will present some experimental results, then try to discuss and compare the solutions proposed.

Chapter 4

Experimental Results

1. Introduction

The previews chapter was dedicated for the description or our work. Where we have explained in details the implementation of genetic algorithm, bat algorithm and the hybrid harmony-bat algorithm. So, we implemented these three algorithms with java language, and we tested it on windows 10 operating system and Intel(R) Core (TM) i5-3317U CPU @ 1.70GHz (4 CPUs), ~1.7GHz and 6 gega-byte of memory and an SSD hard disk 128 gega-byte.

2. Data set

The implemented algorithms tested on graphs that generated randomly, and on a benchmark graphs, the tests were on an undirected graph, and we separated them for two classes.

1. Medium class: contains the graphs with small number of nodes (20, 30, 50, and 75,100,200).

2. Large class: contains the graphs with big number of nodes (300,400,500,800).

3. Benchmark class: contains a graph from DIMACS repository.

3. Results

This table shows us the comparison between genetic and bat algorithms on first class of instances

		Genetic algorithm			Bat algorithm		
Vertices	edges	F	MVC	T_s	F	MVC	T_s
50	250	0.22	11	13	0.14	**7**	7
50	500	0.22	11	13	0.1	**5**	4
50	750	0.18	9	1	0.1	**5**	4
50	1000	0.16	8	1	0.1	5	4
100	500	0.39	39	9	0.18	18	16
100	750	0.27	27	1	0.14	14	14
100	1000	0.25	25	14	0.14	14	14
100	2000	0.27	27	1	0.06	6	13
150	500	0.26	39	6	0.24	36	59
150	750	0.40	61	16	0.18	28	43
150	1000	0.40	61	16	0.16	25	29
150	2000	0.25	38	52	0.1	15	31

150	3000	0.26	39	2	0.08	13	29
200	750	0.42	84	19	0.235	47	7
200	1000	0.43	87	20	0.195	39	7
200	2000	0.255	51	57	0.13	26	7
200	3000	0.25	50	3	0.085	17	7
200	5000	0.25	50	4	0.07	14	7
250	1000	0.46	115	24	0.212	53	13
250	2000	0.42	106	35	0.14	35	10
250	3000	0.24	60	3	0.1	25	10
250	5000	0.24	60	5	0.08	20	10

Table 4.1: class 2 results

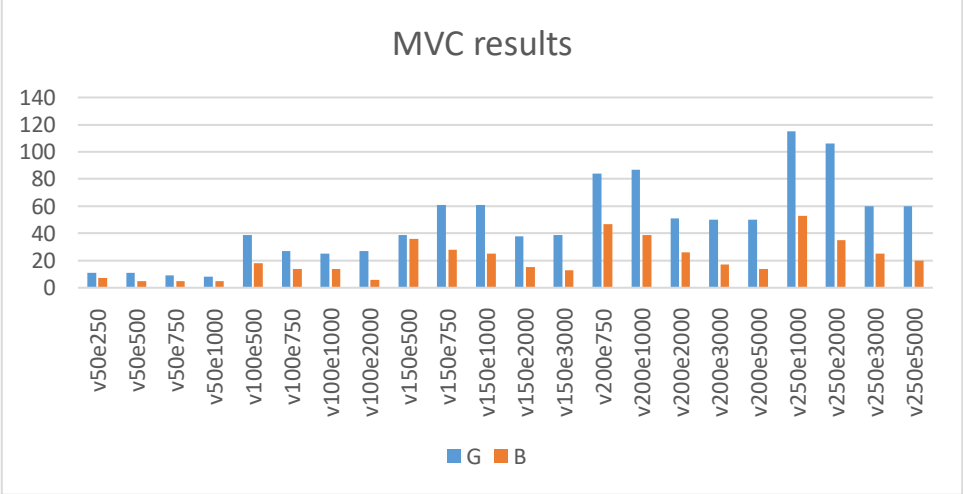


Figure 4.1: MVC comparison

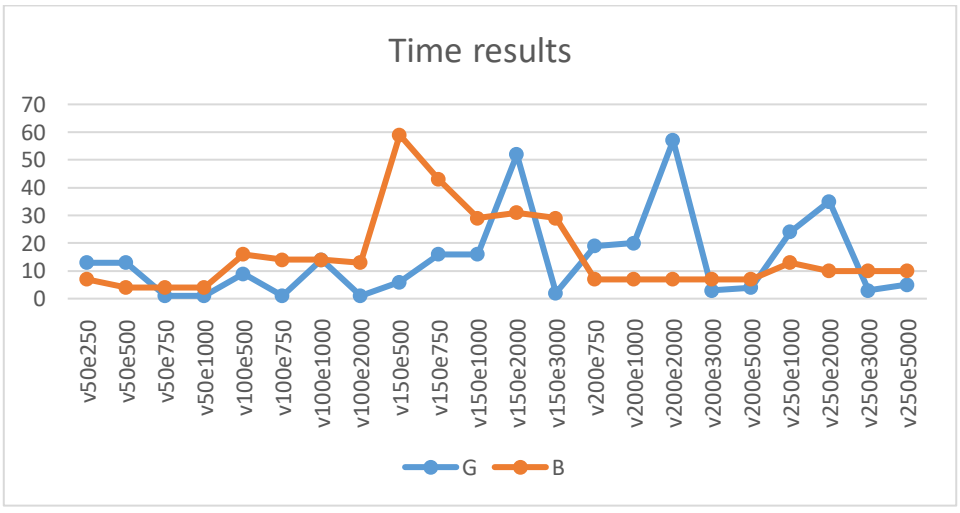


Figure 4.2: Time comparison

This table shows us the comparison between genetic and bat algorithms on second of medium class of instances

		Genetic algorithm			Bat algorithm		
Vertices	edges	F	MVC	T_s	F	MVC	T_s
300	2000	0.23	69	2	0.16	48	32
300	3000	0.23	69	3	0.11	35	30
300	5000	0.23	69	5	0.08	26	30
500	2000	0.49	245	41	0.214	107	89
500	5000	0.268	134	6	0.114	57	77
500	10000	0.268	134	10	0.078	39	44
800	5000	0.45	363	98	0.161	129	192
800	7500	0.45	363	135	0.121	97	185
800	10000	0.45	363	251	0.10	81	179
1000	5000	0.467	467	243	0.196	196	153
1000	10000	0.272	272	397	0.126	126	148
1000	15000	0.248	248	16	0.091	91	151
1000	20000	0.248	248	20	0.075	75	293

Table 4.2: class 2 results

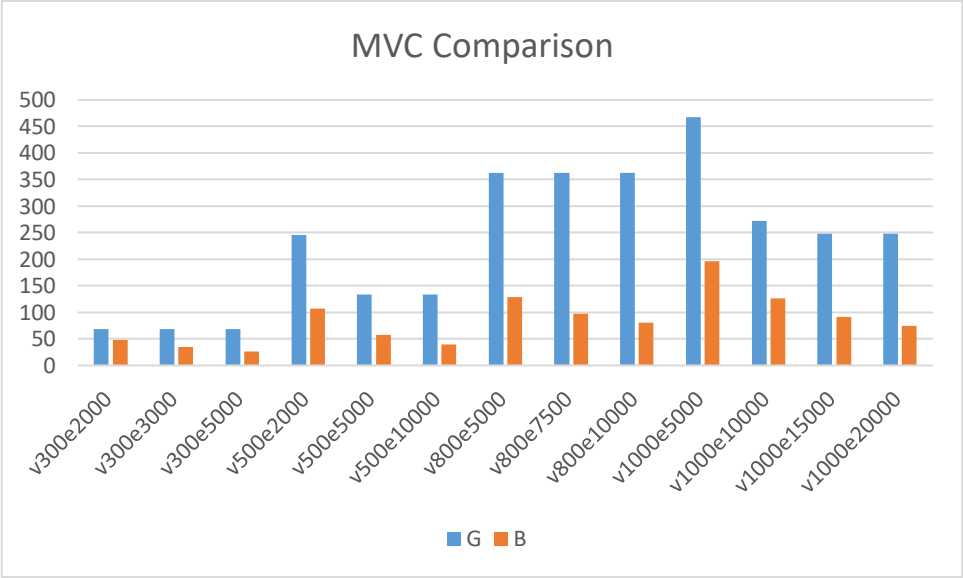


Figure 4.3: MVC comparison

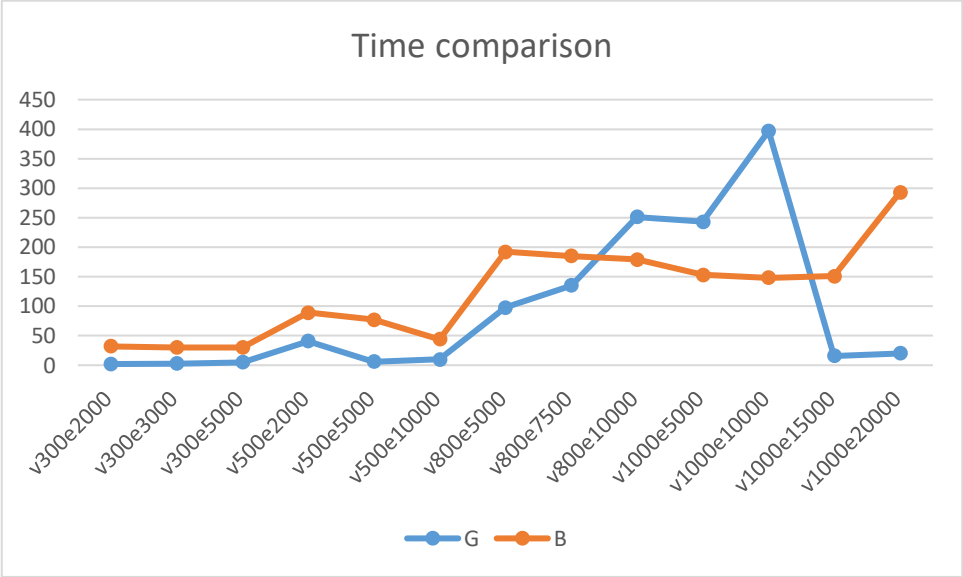


Figure 4.4: Time comparison

This table shows us the comparison between genetic and bat algorithms on benchmark class.

Graph name			Genetic algorithm			Bat algorithm		
	Vertices	edges	F	MVC	T_s	F	MVC	T_s
Frb30-15-2	450	83151	0.2533	0.2533	67	0.008	4	71
frb35-17-3	595	148784	0.247	0.247	121	0.0067	4	161
frb45-21-1	945	386854	0.26	0.26	311	0.003	3	457
frb50-23-1	1150	580603	0.23	0.23	473	0.003	4	525
keller5	776	225990	0.2384	0.2384	174	0.005	4	204
MANN-z27	378	70551	0.2539	0.2539	52	0.0052	2	10
P-hat1500-1	1500	284923	0.256	0.256	258	0.038	58	166
Brock200-1	200	14834	0.27	0.27	10	0.015	3	3
Brock400-1	400	59723	0.2325	0.2325	49	0.01	4	13
Brock800-4	800	207643	0.2525	0.2525	174	0.00875	7	44
C125-9	125	6963	0.224	0.224	4	0.024	3	9
C500-9	500	112332	0.26	0.26	98	0.006	3	18
C1000-9	1000	450079	0.249	0.249	375	0.015	15	17
DSJC1000-5	1000	249826	0.264	0.264	221	0.021	21	68
DSJC500-5	500	62624	0.262	0.262	50	0.016	8	17

c-fat500-1 Frb59-26-5.	500 1534	4459 1049829	0.256	0.256	5	0.066 0.469	33 72	16 229

Table 4.3: class 3 results

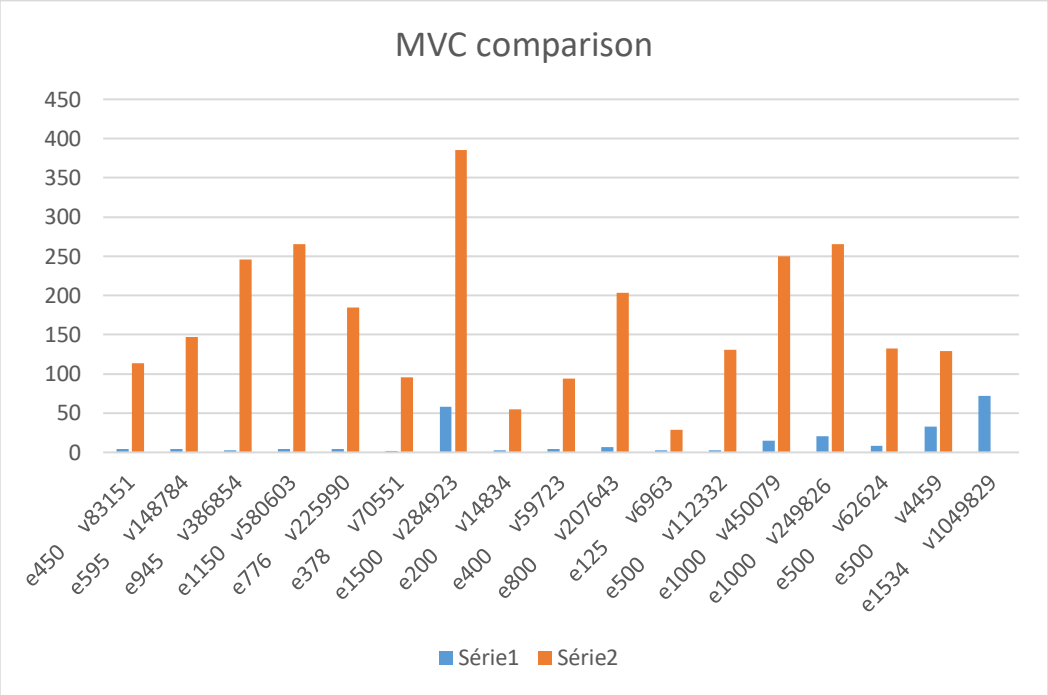


Figure 4.5: MVC comparison

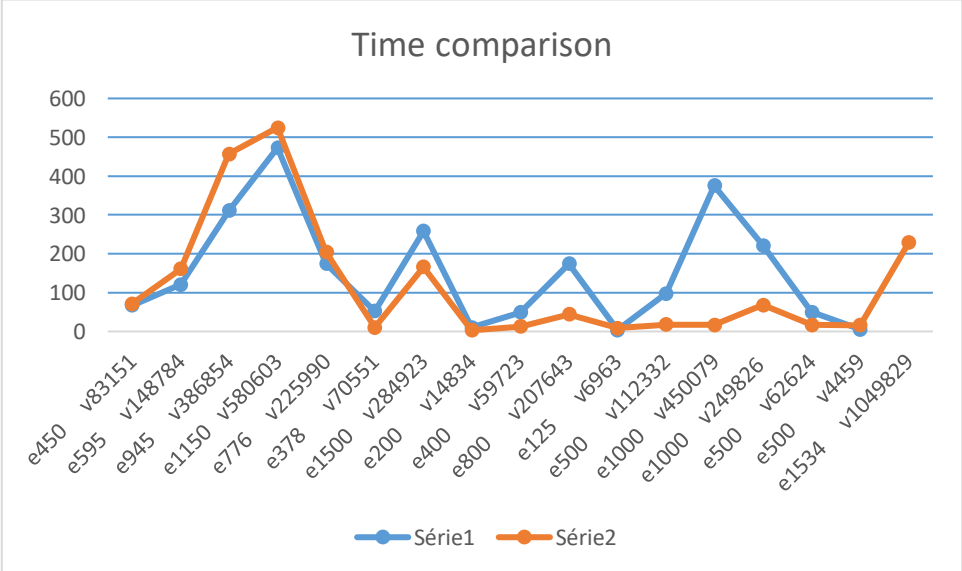


Figure 4.6: Time comparison

This table shows us the comparison between Hybrid bat harmonies algorithm and bat algorithms on benchmark class.

Graph name			Hybrid bat harmonies algorithm			Bat algorithm		
	Vertices	edges	F	MVC	T_s	F	MVC	T_s
Frb30-15-2	450	83151	0.006	3	12	0.008	4	71
frb45-21-1	945	386854	0.0031	3	53	0.003	3	457
frb50-23-1	1150	580603	0.002	3	75	0.003	4	525
Brock200-1	200	14834	0.015	3	3	0.015	3	3
C125-9	125	6963	0.016	2	1	0.024	3	9
P-hat1500-1	1500	284923	0.016	24	44	0.03	58	166
frb35-17-3	595	148784	0.005	3	20	0.0067	4	161
keller5	776	225990	0.005	4	31	0.005	4	208
DSJC500-5	500	62624	0.014	7	9	0.016	8	17
Frb59-26-5.	1534	1049829	0.001	3	132	0.005	9	1025

Table 4.4: class 3 results

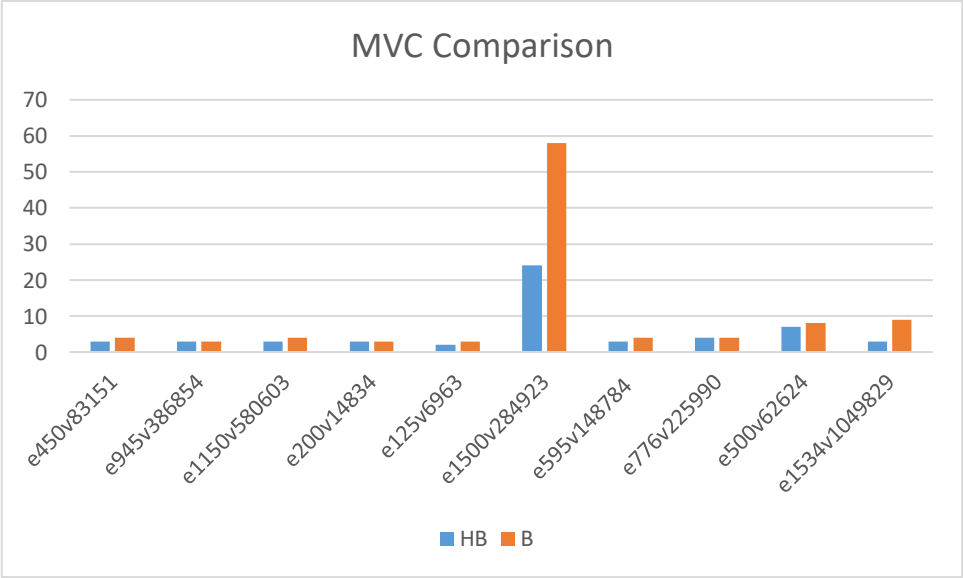


Figure 4.7: MVC comparison

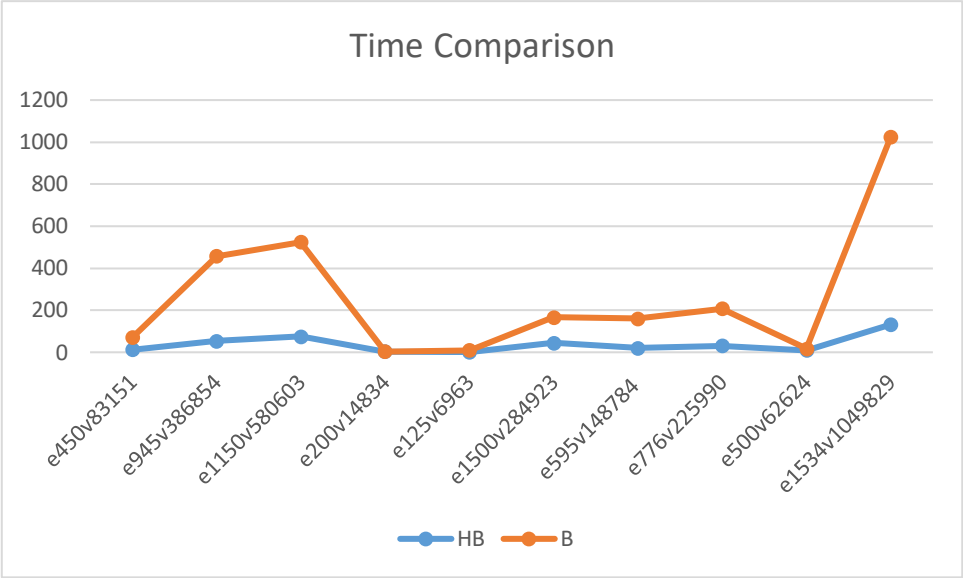


Figure 4.8: Time comparison

4. Discussion of results

From the 4 tables above, we see in charts:

- Minimum Vertex Cover: bat algorithm is always getting better solutions in most times it brings the global optimum with less more iterations and population.
- The Hybrid Harmony Bats algorithm is getting minimal solutions mostly better with less population than bat algorithm
- Time factor: genetic algorithm is better when number of edges is near to the number of vertices but when number of edges get bigger specially in large graphs bat algorithm do better than genetic algorithm.
- The Hybrid Harmony-Bats algorithm takes less time and an optimal solution better in most times than Bat algorithm especially in large graphs.

5. Conclusion:

Bio-inspired are showing us their strength dealing with big data, the bat algorithm is strong with hard cases but it takes longer time due to its high changes every iteration, but with The Hybrid Harmony Bats algorithm it can jumps over a lot of changes that takes long time with its parameters which will saves useless time Bat algorithm is taking.

The minimum vertex cover problem is one of the NP-complete problems that has its real applications everywhere.

The aim of our project is to implement and compare the results of genetic algorithm, bat algorithm and a hybrid harmony-bats search algorithm.

The experimental results prove that meta-heuristics and hybrid meta-heuristics get much butter results, the improvement is in the small number of nodes in the vertex cover set and in the time of execution, even for the large graphs. The hybrid algorithm proposed seek to find best solution and best time, this is obtained in more cases of execution.

As a further work, we propose to improve the hybrid algorithm and to try to combine more meta-heuristics to get optimal results in an efficient time interval. We also suggest to solve other np-hard problem in relation with graph theory such us: dominating set problem.

[1] Richard M. Karp (1972). "Reducibility Among Combinatorial Problems"(PDF). In R. E. Miller; J. W. Thatcher; J.D. Bohlinger (eds.). Complexity of Computer Computations. New York: Plenum. pp. 85–103. doi:10.1007/978-1-4684-2001-2_9.

[2] Sean Luke, 2013, Essentials of Metaheuristics, Lulu, second edition, available at http://cs.gmu.edu/↤sean/book/metaheuristics/

[3] Fred Glover and Kenneth Sörensen (2015) Metaheuristics. Scholarpedia, 10(4):6532.

[4] Sasireka, A. H. Nandhu Kishore , Applications of Dominating Set of Graph in Computer Networks, International Journal Of Engineering Sciences & Research Technology

[5] Melanie Mitchelle,An Introduction to Genetic Algorithm, 1996 Massachusetts Institute of Technology

[6] A survey on metaheuristics for stochastic combinatorial optimization Leonora Bianchi Æ Marco Dorigo Æ Luca Maria Gambardella Æ Walter J. Gutjahr.

[7] Radoslav Harman ,A very brief introduction to particle swarm optimization , Department of Applied Mathematics and Statistics, Faculty of Mathematics, Physics and Informatics Comenius University in Bratislava.

[8] Geem ZW, Kim JH and Loganathan GV (2001) A new heuristic optimization algorithm: Harmony search. Simulation, 76:60-68.

[9] Xin-She Yang, Harmony Search as a Metaheuristic Algorithm, in: Music-Inspired Harmony Search Algorithm: Theory and Applications, (Editor Z. W. Geem), Studies in Computational Intelligence, Springer Berlin, vol. **191** pp. 1-14 (2009)

[10] Xin-She Yang ,A New Metaheuristic Bat-Inspired Algorithm .Springer, 2010, pp. 65–74.

[11] Keijo Ruhonene,Graph theory,2013.

[12] S.G. Shrinivas et. al, APPLICATIONS OF GRAPH THEORY IN COMPUTER SCIENCE AN OVERVIEW, International Journal of Engineering Science and Technology Vol. 2(9), 2010, 4610-4621

[13] A Edsger Dijkstra , A Note on Two Problems in Connexion with Graphs (1959).

[14] A.Potluri ,A.Singh .Hybrid metaheuristic algorithm for minimum weight dominating set Applied soft computing 13(2013) 76-88

[15] g asireka, A. H. Nandhu Kishore , Applications of Dominating Set of Graph in Computer Networks, International Journal Of Engineering Sciences & Research Technology.

[16] Shwetha Kumari V, Vasudeva Pai, Ravi B, International Journal of Engineering Research & Technology (IJERT) Vol. 5 Issue 05, May-2016.

websites :

a. Terr, David. "Polynomial Time." From MathWorld--A Wolfram Web Resource, created by Eric W. Weisstein. http://mathworld.wolfram.com/PolynomialTime.html
b. Complexity Classes. *Brilliant.org*. Retrieved , from https://brilliant.org/complexity-classes/
c. https://www.semanticscholar.org/
d. https://bjopm.emnuvens.com.br/bjopm/article/view/425/633
e. Joseph Brown, head of artificial intelligent in game development, lab http://serious-science.org/evolutionary-algorithms-6330
f. www.tech_talk.com
g. www.researchgate.net
h. https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation.htm
i. https://techdifferences.com/difference-between-bfs-and-dfs.html.
j. https://www.bogotobogo.com/python/python_Dijkstras_Shortest_Path_Algorithm.php
k. http://www.dharwadker.org/vertex_cover.

ملخص

في هذه الدراسة ، نتعامل مع واحدة من المشكلات الأساسية في نظرية الرسوم البيانية والتحسين التوافقي ، وهي مشكلة الحد الأدنى لقمة الرأس (MVCP). نجري مقارنة بين: الخوارزمية الجينية والخوارزمية الجديدة: خوارزمية الخفافيش. أيضًا ، يُقترح خوارزمية هجينة للحصول على نتيجة أفضل بكثير. يتم تطبيق مقارنة لتحديد أي من هذه الخوارزميات الثلاث هي الافضل لإيجاد الحل الأمثل للمشكلة

Abstract

In this study, we deal with one of the fundamental problems in graph theory and combinatorial optimization, that is, the minimum vertex cover problem (MVCP). We make a comparison between well-known heuristic: genetic algorithm and new metaheuristic: bat algorithm. Also, a hybrid algorithm is proposed to obtain much more optimal result. A comparison is applied to decides which of these three algorithms is much butter to find the optimal solution to the minimum vertex cover problem.
Key words: minimum vertex cover problem, heuristics, meta-heuristics, bat algorithm, hybrid algorithm.

Résumé

Dans cette étude, nous traitons l'un des problèmes fondamentaux de la théorie des graphes et de l'optimisation combinatoire, le problème de la couverture de sommet minimale (MVCP). Nous faisons une comparaison entre l’une des l'heuristique les bien connue : algorithme génétique et le nouvel algorithme métaheuristique : bat ou chauve-souris. En outre, un algorithme hybride est proposé pour obtenir un résultat beaucoup plus optimal. Une comparaison est appliquée pour décider lequel de ces trois algorithmes est le meilleur pour trouver la solution optimale au problème de la couverture minimale des sommets.
Mots clés : problème de couverture minimale de vertex, heuristique, méta-heuristique, algorithme bat, algorithme hybride.