博士論文

Optimization by Metaheuristic Methods: Spy Algorithm and B-VNS

(メタヒューリスティック手法による最適化： SPY アルゴリズムと B-VNS)

令和　4 年　9 月

DHIDHI PAMBUDI

山口大学大学院創成科学研究科

Optimization by Metaheuristic Methods: Spy Algorithm and B-VNS

Author:
DHIDHI PAMBUDI

Supervisor:
Masaki Kawamura

Graduate School of Sciences and Technology for Innovation
Yamaguchi University

Summary

The role of optimization can be found in almost all aspects of human life. Optimization is common in but not limited to the fields of engineering, economics, design, and planning. Although the optimization problems to be solved change, the optimization goal never changes. That is to find effective solutions efficiently. In modern optimization studies, the metaheuristic algorithm has been one of the most interesting methods, considering the demands of a reasonable computational time.

Many metaheuristic algorithms have been introduced. However, based on the number of tentative solutions used in the search process, metaheuristic algorithms can be categorized into (1) population-based or (2) single-trajectory-based algorithms. The searching with single-trajectory-based metaheuristic algorithms manipulates and modifies a single solution point in every iteration. In contrast, the population-based metaheuristic algorithms combine a set of solution points to create new solutions in every iteration.

A metaheuristic algorithm usually consists of two components, i.e., exploration and exploitation. Exploration means searching for solutions in the global space. In contrast, exploitation means searching for a solution by focusing on a small area or an area near an already known solution. The single-trajectory-based metaheuristic algorithm is exploitation-oriented. On the other side, the population-based metaheuristic algorithm is exploration-oriented because of searching by many points distributed on all search spaces. Balance settings between exploration and exploitation are needed to produce good solutions. In fact, most population-based algorithms will encounter decreasing in exploration and become too exploitation-oriented as the iteration increase. Any metaheuristic algorithm applies parameters to control the behavior. However, the parameters usually do not provide a good intuition of the rate of exploration and exploitation. Hence, reaching a balance between them is hard to predict just by the algorithm parameters.

This dissertation proposes a conceptual design combining the spy algorithm and B-VNS. The spy algorithm is a population-based metaheuristic algorithm that mimics the strategy of a group of spies, the spy ring. The spy algorithm is a new concept with the main idea to ensure the benefit of exploration and exploitation, and cooperative and non-cooperative searches always exist. This goal is implemented by utilizing three kinds of dedicated search operators and regulating them in a fixed portion. The occurrences of exploration and exploitation are controlled by algorithm parameters. Thus, the spy algorithm parameters provide good before-running intuition to easier reach the balance between exploration and exploitation. The spy algorithm is first designed to be used in the continuous optimization model.

The spy algorithm was compared to the genetic algorithm, improved harmony search, and particle swarm optimization on a set of non-convex functions by aiming at accuracy, the ability to detect many global optimum points, and computation time. The Kruskal–Wallis tests, followed by Games–Howell post hoc comparison tests, were conducted using $\alpha = 0.05$ for the comparison. The statistical analysis results show that the spy algorithm outperformed the other algorithms by providing the best accuracy and detecting more global optimum points within less computation time. Furthermore, those results indicate that the spy algorithm is more robust and faster than other algorithms tested.

On the other hand, the B-VNS algorithm is a modification of the variable neighborhood search (VNS) algorithm. The benefit of VNS comes from its thorough search while avoiding the local optimum trap by moving to the neighboring point called shaking. The local search after shaking is another benefit of VNS that makes VNS a prominent algorithm. However, the

thorough search has the drawback of long computation time. This dissertation introduces a modified neighborhood structure to reduce the computation times. The main idea is to apply the binomial distribution to create the neighboring point. As a result, the neighborhood distance has a random pattern. However, it follows a binomial distribution instead of a strictly monotonic increase like in VNS. The B-VNS is a modification of VNS and is classified as a single solution-based algorithm. The B-VNS is intended to solve combinatorial optimization problems, particularly the quadratic unconstrained binary optimization (QUBO) problems categorized as NP-hard problems.

The B-VNS and VNS algorithms were tested on standard QUBO problems from Glover and Beasley, on standard max-cut problems from Helmberg–Rendl, and those proposed by Burer, Monteiro, and Zhang. Finally, Mann–Whitney tests were conducted using $\alpha = 0.05$ to compare the performance of the two algorithms statistically. It was shown that the B-VNS and VNS algorithms are able to provide good solutions, but the B-VNS algorithm runs substantially faster. Furthermore, the B-VNS algorithm performed better in all of the max-cut problems regardless of problem size and in QUBO problems with sizes less than 500.

The spy algorithms and B-VNS have different designs in the process and the domain of the solved problems. However, considering the benefit of the spy algorithm and B-VNS, their combination has the potential to provide good results. Conceptually, the spy algorithm can be seen as the first step of B-VNS. Conversely, B-VNS can be considered an additional refinement for the spy algorithm.

Contents

Title i

Summary ii

Contents iv

List of Figures vi

List of Tables vii

1 Introduction 1
- 1.1 Metaheuristic Classification . . . 1
- 1.2 Main Steps in Metaheuristic . . . 2
- 1.3 Challenge . . . 3

2 Novel Metaheuristic: Spy Algorithm 4
- 2.1 Introduction . . . 4
- 2.2 Population-Based Metaheuristic . . . 5
  - 2.2.1 Genetic Algorithm . . . 5
  - 2.2.2 Particle Swarm Optimization . . . 6
  - 2.2.3 Improved Harmony Search Algorithm . . . 6
  - 2.2.4 Common Drawback . . . 6
- 2.3 Spy Algorithm . . . 6
  - 2.3.1 Concepts . . . 8
  - 2.3.2 Implementation . . . 10
- 2.4 Evaluation . . . 12
  - 2.4.1 Test Condition . . . 12
  - 2.4.2 Experimental Result . . . 13
- 2.5 Conclusion . . . 18

3 Constructing the Neighborhood Structure of VNS Based on Binomial Distribution for Solving QUBO Problems 20
- 3.1 Introduction . . . 20
- 3.2 VNS Algorithm . . . 22
- 3.3 Proposed Neighborhood Structure . . . 23
- 3.4 Benchmarking . . . 25
  - 3.4.1 Test on QUBO problems . . . 27
  - 3.4.2 Test on Max-Cut problems . . . 28
  - 3.4.3 Discussion . . . 31

3.5 Conclusion 33

4 A Conceptual Design: Combination of Spy Algorithm and B-VNS 34

5 Concluding Remarks 36

Bibliography 38

List of Figures

2.1 Generating new solution in GA 7
2.2 PSO movement, influenced by personal best and global best 7
2.3 Creating new solution in IHS 7
2.4 Agent movements 9
2.5 Pseudocode of the spy algorithm 11
2.6 Solution distribution in each category 12
2.7 2-dimensional version of the test functions 14
2.8 Boxplot of error, the red dots show the average values 15
2.9 Detecting global optimums on Shubert function having 18 GOPs (red dots are solutions, diamonds means the solutions lie near a certain GOP while triangles are missed GOPs) 18
2.10 Boxplots of computation time 19

3.1 VNS algorithm. 23
3.2 Distribution of $\boldsymbol{X}'$ related to p. 25
3.3 Neighborhood structure. 26
3.4 Local search for QUBO [71]. 28

4.1 Conceptual design 34

List of Tables

2.1 Test functions 14
2.2 Algorithm parameters 15
2.3 Average error ± Standard deviation 16
2.4 P-values of Games–Howell post hoc error comparison 17
2.5 Average MPR 18
2.6 P-values of post hoc time comparison 19

3.1 Results for Glover [44] problems. 29
3.2 Results for Beasley [71] problems. 30
3.3 Results for Helmberg–Rendl [73] problems. 31
3.4 Results for Burer et al. [75] problems. 32

5.1 Common metaheuristic steps apply to spy algorithm and B-VNS 36

Chapter 1

Introduction

Optimization is an essential part of the history of human civilization's progress, even today. Fields such as engineering, economics, design and planning are some areas where optimization is particularly intense. The optimization benefits are so great in almost all aspects of life that many researchers and practitioners from multidisciplinary fields work on it. However, the challenges in optimization studies remain the same; getting solutions that are more accurate, faster, and, if possible, in a more straightforward manner.

So many optimization problems arise according to the nature of the problem. Thus it needs a suitable method for an appropriate problem while considering demands such as time limitations and other resource constraints. Therefore, optimization problems are categorized according to several aspects to make them easier to understand and solve. Based on the variables used in the model, the optimization is categorized into three models: continuous, discrete, and mixed.

Many methods have been proposed to solve optimization problems, but basically, these methods can be categorized into two groups. First is the exact method in which the solution is obtained through a procedure based on a mathematical theorem so that we can prove the optimality of the obtained solution. The second is the approximation method, where the procedure applies tolerance to mathematical theorems, or even the procedure is often difficult to explain mathematically. However, many practices show that approximations can obtain reasonable solutions.

In the approximation group, the metaheuristic method is one of the methods that has attracted much attention in the recent optimization world. Apart from the practicality and flexibility of implementation in various problems, the metaheuristic method can provide outstanding solutions in a relatively short time. Some well-known metaheuristic methods include genetic algorithm, simulated annealing, tabu search, and variable neighborhood search (VNS).

1.1 Metaheuristic Classification

Many metaheuristic algorithms have been introduced. Metaheuristic methods often use a solution-finding process that imitates natural phenomena in the real world. Some apply a group search model, while some apply a single model. Thus, metaheuristic methods are classified into (1) population-based and (2) single-trajectory-based algorithms [1, 2]. The searching with single-trajectory-based metaheuristic algorithms manipulates and modifies a single solution point in every iteration. The well-known simulated annealing (SA) [3] is an example of a successful single-trajectory-based metaheuristic algorithm. In contrast, the population-based metaheuristic algorithms combine a set of points to create new solutions in every iteration.

A metaheuristic algorithm usually consists of two components, i.e., exploration and exploitation. Exploration means searching for solutions in the global space, while exploitation means searching for solutions by focusing on a small area or an area near an already known solution. The single-trajectory-based metaheuristic algorithm is exploitation-oriented [1]. Population-based metaheuristic algorithm executes searching by using many points distributed on all search spaces; therefore, it is exploration-oriented [1]. Some metaheuristic algorithms use both exploration and exploitation orientation. Exploration and exploitation should be balanced by choosing the proper value of algorithm parameters to obtain a good result. However, this is sometimes time-consuming.

1.2 Main Steps in Metaheuristic

A metaheuristic algorithm may be inspired by a unique phenomenon that results in different strategies. Some metaheuristic algorithms may have simple processes, while others do not. Although many metaheuristic algorithms have different processes, regardless of whether it is population-based or single-trajectory-based, the framework in most consists of the following five main steps.

1. **Initialization**. Initialization is commonly started from random positions since there is no information about the solution space. However, an initialization method, such as the greedy procedure in GRASP [4, 5], can be used.

2. **Solution refinement**. Better solutions are generated in this step. Each metaheuristic algorithm has its refinement strategy inspired by real-world natural phenomena, such as ant colony optimization [6] and cuckoo search [2]. The new solutions that are expected to be better than the previous solutions are produced. In exceptional cases, algorithms such as SA can temporarily result in worse solutions [7]. Based on its strategies, metaheuristic algorithms, such as the genetic algorithm (GA) and harmony search (HS), may apply sorting mechanisms [8].

3. **Solution update**. When a new solution was created from the previous solution by refinement strategy, the algorithm should update its solution by selecting a new one or old one on the basis of a certain rule. The updated solutions will be transferred to the next iteration.

4. **Termination**. As the solution needs to be obtained in a reasonable time, the algorithm should be stopped in a reasonable manner. Usually, the algorithm is terminated when the solutions are not expected to improve. In addition, a maximum number of iterations is set to avoid an infinite loop.

5. **Finalization**. After the algorithm is stopped, one of the possible solutions is reported as the optimal solution.

The most important steps are the refinement strategy and the updating mechanism since they make a metaheuristic algorithm unique from others. The other steps are almost similar to any metaheuristic algorithms. The refinement strategy focuses on the mechanism for obtaining new solutions. This step can be categorized into two strategies; cooperative and non-cooperative.

The cooperative strategy means that new solutions are obtained by involving and manipulating two or more previous solutions. An example of cooperative search is the crossover operator in the GA. The non-cooperative strategy requires only one previous solution for obtaining new

solutions, such as in SA [3] and tabu search [9, 10]. The previous solution is not necessary for the random search.

1.3 Challenge

Although many metaheuristic algorithms have been introduced, the demand for better and faster solutions has encouraged researchers to introduce new metaheuristic algorithms to solve optimization problems. This dissertation is written to contribute to the optimization field by introducing two metaheuristic algorithms and later their conceptual combination design.

The first algorithm design will be described in Chapter 2. It is a population-based metaheuristic algorithm called the spy algorithm. There is a well-accepted principle that a metaheuristic algorithm can give good results by balancing the exploration and exploitation search. However, most metaheuristics have an unclear classification of search operator [11]. Indeed, the rate of exploration and exploitation is hard to predict just by their parameters. The rates can be known only after running the algorithm. Thus, most algorithm parameters do not give a strong before-running intuition of the balance between exploration and exploitation. Considering that many traditional population-based-metaheuristic algorithms encounter decreasing benefits of exploration search and become too strong in exploitation search as iterations increase [12, 11], the main idea of the spy algorithm is to maintain the exploitation and exploration by separating each type of searching and regulate their occurrence by fixed ratio parameters. Hence, the spy algorithm guarantees the occurrence of exploitation and exploration in each iteration. The spy algorithm parameters can give a good intuition of exploration/exploitation rate so that a balance between them can be easier to reach.

The spy algorithm was originally designed to solve the continuous optimization model. However, as the metaheuristic algorithm is a high-level strategy for finding solutions, the spy algorithm has the potential for solving discrete models. Regarding the nature design of the spy algorithm, combining it with a specific algorithm for solving discrete models will improve the results. Chapter 3 will introduce that specific algorithm which is a single-trajectory-based algorithm called B-VNS, to solve combinatorial problems, specifically the quadratic unconstrained binary optimization (QUBO) problems. B-VNS is a modified variable neighborhood search (VNS) that is a prominent algorithm for solving QUBO problems. However, the VNS has a drawback in long computation times. Therefore, the B-VNS introduces the application of Binomial distribution to construct the neighborhood structure to reduce the computation time.

The spy algorithm and B-VNS come from different designs, but they can be combined into a single algorithm. The spy algorithm can be seen as the first step of B-VNS. Conversely, the B=VNS can be seen as an additional refinement of spy algorithm. Chapter 4 will propose a conceptual design of how to combine the spy algorithm and B-VNS. Finally, the concluding remarks will be given in Chapter 5.

Chapter 2

Novel Metaheuristic: Spy Algorithm

2.1 Introduction

Humans have long benefited from optimization. Optimization will always be needed in line with our desire to always obtain the best solutions by considering the constraints. We can find optimization cases in almost all fields. Optimization is common in the fields of engineering design, management, economics, physics, and biology. One of the optimization applications is the very large-scale integration (VLSI) design [13] for creating an integrated circuit (IC) that supports hardware and technology development. The demand to obtain a high-capability chip requires increasingly more components to be embedded. This is made even more complicated by demands that the chip size should be small, consume little power, yet work fast and be able to handle noise. All processes must be done in a short time if the manufacturer wants to launch a new chip by adjusting the time-to-market strategy.

Solving optimization cases commonly starts with modeling the problem to make it easier to understand. Unfortunately, the obtained models often tend to be complicated and hard to solve. As optimization is equal to decision problems, a large-scale model often takes too long to solve. Moreover, many problems are intractable and categorized as non-deterministic polynomial (NP)-complete problems [14]. VLSI design is just one real-world application of the graph-partitioning problem, which is categorized as NP-complete. One of the basic partitioning problems is the maximum cut problem which is included in the original Karp's 21 NP-complete problems [15] and is equivalent to the Ising model [16], which is very influential in the fields of physics and mechanical statistics. Another example of an NP-complete problem that exists is complex job scheduling.

In optimization, it is common to pursue accurate solutions, but it is also practical to find a solution within a reasonable and acceptable time. We do not want to waste time in a highly competitive world. The demand for obtaining good solutions in a short time leads us to make a compromise and accept the solution obtained through an approximation approach. A well-known approximation method is the metaheuristic algorithm, which usually includes a stochastic search. Compared with deterministic searches, metaheuristics have the advantages that they do not require any information about the function to be optimized. Many metaheuristics do not necessarily take into account the gradient of the function. Most are also flexible or problem-independent, which means they can be applied to various kinds of problems.

Metaheuristic algorithms have an important role in optimization in academic research and in real-world practice. Many metaheuristic algorithms have been introduced, but the demand for better, more accurate, and faster algorithms continues. Since there are various goals in the optimization problems, it is difficult to develop a single algorithm that can solve a wide range of

problems and always outperform other algorithms. On the basis of the 'no free lunch theorem' [17], it is challenging to derive a single metaheuristic algorithm that fits any problems since a certain metaheuristic algorithm may be strong for certain problems but weak for others.

The major challenge in the field of optimization is to develop a robust algorithm that can solve a wide range of problem types as accurately and efficiently as possible. A robust algorithm is one that ensures convergence even when starting from an arbitrary initial solution [18, 19]. Even after many runs, the obtained solutions should not be sensitive to parameters [20] and have low variation [21]. Certain fast algorithms are so focused on speed that they lack robustness [21]. To obtain good results, any metaheuristic algorithm should maintain the exploration and exploitation search. However, the balance is usually hard to achieve. Even though any metaheuristic is controlled by a set of parameters, the parameters can not give a before-running intuition of the occurrence of exploration and exploitation.

In an effort to develop a robust metaheuristic algorithm, we propose an algorithm called the spy algorithm. The spy algorithm is a population-based metaheuristic algorithm that mimics the strategy of a group of spies, the spy ring. Within a spy ring, each spy agent considers three types of movements, which are: (i) movement within a small perimeter, (ii) movement by considering another spy agent, or (iii) just making a random move. The main idea of the spy algorithm is to set a fixed portion of three dedicated search operators by setting the algorithm parameter. Hence, the occurrences of exploration or exploitation can be guaranteed in each iteration.

To demonstrate its robustness, it was tested to solve two types of problems; (i) optimization problems with multi-dimensional function, and (ii) multimodal optimization. The non-convex functions were used on the test as non-convex optimizations are NP-hard problems [22], so they are suitable for testing metaheuristic algorithms. A total of 12 standard benchmark functions were used to demonstrate the performance of the spy algorithm and the results were compared with the GA, IHS, and PSO.

2.2 Population-Based Metaheuristic

In a population-based algorithm, many tentative solutions involved allow the algorithm to explore simultaneously many points in the search space in one iteration and combine much of the information obtained to improve the solutions. Solution combination is one of the advantages of a population-based metaheuristic algorithm. However, a larger memory capacity is needed to keep a number of solutions at the same time. Each metaheuristic algorithm will be unique and has specific characteristics depending on its design.

2.2.1 Genetic Algorithm

The genetic algorithm (GA) was inspired by Darwin's theory of evolution [23]. It mimics the natural evolution of a population by the process of solution reproductions, creates new solutions, and competes for survival [24] based on the operators of selection, crossover, mutation, and sometimes elitism [25]. The main operators in GA are crossover and mutation. The crossover occurrence is controlled by a crossover probability p_c and the mutation is controlled by mutation probability p_m. The implementation first starts from generating random numbers $r1, r2 \in (0, 1)$. If $r1 < p_c$ then the crossover should be performed and the mutation will be performed when $r2 < p_m$. Those two parameters have a substantial role to control the exploration and exploitation search. Figure 2.1 shows the schema for generating new solutions in real-coded GA using simple arithmetic operators.

2.2.2 Particle Swarm Optimization

Another well-known algorithm is particle swarm optimization (PSO), which was originally intended for simulating social behavior, as it mimics the movement of organisms such as a flock of birds or a school of fish [26]. PSO is a more recent algorithm than GA and can be an alternative to GA since it is simpler and converges faster than GA [27, 28]. In the solution refinement, the new solution is created by considering the global best solution and personal best solution. Considering the personal best is called cognitive movement and is controlled by a parameter $c1$. The global best consideration is called social movement and is controlled by parameter $c2$. Figure 2.2 shows the movement applies in PSO.

2.2.3 Improved Harmony Search Algorithm

Another more recent metaheuristic algorithm that has attracted much attention is improved harmony search (IHS). IHS is a modification of HS, the process of which is inspired by a jazz musician finding a good harmony of musical notes [29, 30, 31]. A new solution is generated by considering the previous solution kept in harmony memory. This process is controlled by the harmony memory consideration rate (HMCR) parameter. A random number $r1 \in (0, 1)$ is generated at first. The solution is taken from harmony memory when $r1 < HMCR$. Otherwise, the new solution is randomly generated. After creating new solutions from harmony memory, the next step is to decide whether it needs an adjustment. This process is controlled by the pitch adjustment rate (PAR) parameter. Whenever a generated random number $r2 \in (0, 1)$ is less than PAR, then an adjustment will be made at some portion of the bandwith (BW) parameter. The process for creating a new solution is shown in Fig. 2.3.

2.2.4 Common Drawback

A balance between the exploration and exploitation search is the key principle to obtaining good results. However, most metaheuristics have an unclear classification of search operator [11]. Any metaheuristic controls its operators using some parameters. In fact, the rate of exploration and exploitation is hard to predict just by their parameters. The rates can be known only after running the algorithm. In GA, setting the parameters p_c and p_m will give a weak clue of the occurrence of exploration and exploitation searches. The weakness comes from the fact that it depends on the generated random number $r1$ and $r2$. Moreover, the selection operator will have a great impact because it determines which solutions will be taken to create new solutions. IHS and PSO also suffer from the same thing as in GA.

Most algorithm parameters do not give a strong before-running intuition of the balance between exploration and exploitation. The exploration and exploitation rate can be known just after running the algorithm. In fact, many traditional population-based-metaheuristic algorithms encounter decreasing benefits of exploration search and become too strong in exploitation search as iterations increase [12, 11]. As the occurrence of exploration and exploitation searches is hard to predict, achieving a balance between them is tough. As a result, certain algorithms may lack robustness.

2.3 Spy Algorithm

The spy algorithm was inspired by the strategy used by a spy ring to locate an enemy base. A spy ring is a group of spies cooperating with each other by sharing intelligence. It should be noted that the strategy adopted in the spy algorithm is not the same as real-life espionage since a

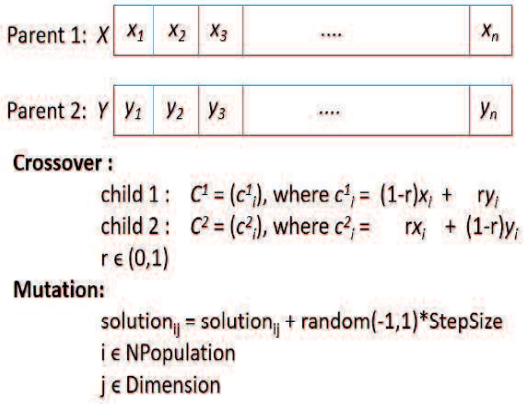


Figure 2.1: Generating new solution in GA

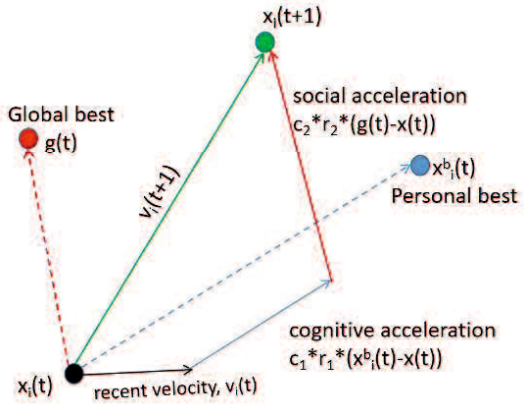


Figure 2.2: PSO movement, influenced by personal best and global best

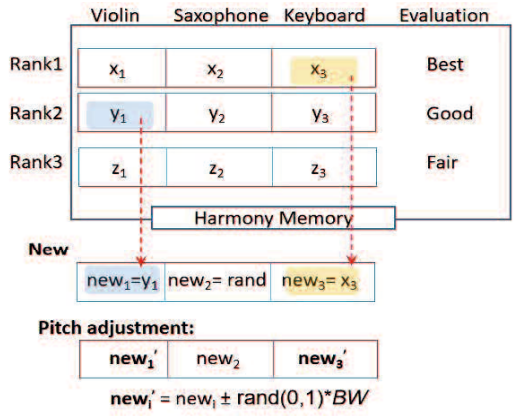


Figure 2.3: Creating new solution in IHS

real spy agent, alone or in a ring, will make many considerations to determine movements. The spy algorithm maintains the exploitation and exploration by separating each type of searching and regulating their occurrence by fixed ratio parameters. Hence, the spy algorithm guarantees the occurrence of exploitation and exploration in each iteration. A set of algorithm parameters is designed to provide a good intuition of exploration/exploitation rate so that a balance between

them can be easier to reach.

2.3.1 Concepts

The spy algorithm is based on a scenario in which a country has been infiltrated by an enemy, but its base is unknown, so the spy agency tries to find the enemy base. The information quality of the enemy base is evaluated by a given objective function. We consider the minimization problems. Therefore, the smaller the value of the function, the better the quality. The spy agents should follow the following steps to find the enemy base.

1. *Initialization.* The agents are sent to random locations and then evaluate the information quality of the locations by using the objective function.

2. *Refinement*

   - *Agent classification.* On the basis of information quality, the agents are classified into three classes: high-rank, mid-rank, and low-rank. The agent which gives the smaller value of the function is the criteria for the high-rank agent in the context of minimization. In the case of maximization, this selection can be easily switched or adjusted.
   - *Movement.* On the basis of agent class, each agent searches for a new location on the basis of the following rules.
     (a) High-rank agents perform *SwingMove*, i.e., they move within a small perimeter on the basis of their own location.
     (b) Mid-rank agents perform *MoveToward*, i.e., they move toward another agent location.
     (c) Low-rank agents perform the random search.

     After performing these movements, each agent evaluates the information quality of the new locations.

3. *Update.* If the new location has better information quality, the agent adopts the location; otherwise goes back to the previous location.

4. *Termination.* Repeat main steps 2 to 3 until the stop command is issued.

5. *Finalization.* The location and the information obtained in the last iteration are reported as the final solution. The best agent provides the best solution achieved while the rest serve as the alternative solutions.

In the context of optimization, a spy agent can be seen as a solution, and the unknown enemy base is the optimum point. The location or position of an agent forms a solution vector, and the information regarding its location is evaluated by the objective function. In this scenario, the cooperative strategy is implemented in *MoveToward*, and a non-cooperative strategy is implemented into two movements, i.e., *SwingMove* and random search. To increase the convergence speed, the agent with higher rank can be chosen in the *MoveToward* movement. The movement for each agent is illustrated in Fig. 2.4. The updating mechanism is a one-to-one comparison between the newly generated solutions and previous ones.

The concept of the spy algorithm is based on that rational thinking that, on the high-rank agents, we should make a slight refinement to avoid the risk of a significant decrease in the quality of information. However, we need to make a progressive movement as well as take

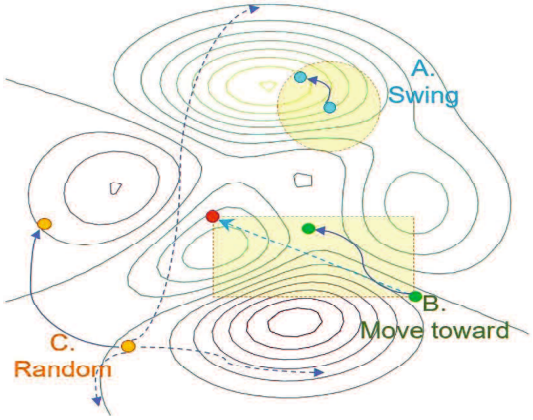


Figure 2.4: Agent movements

advantage of the benefit of the already known solutions. Implementing this movement for mid-rank agents is the right choice so that they can improve the solution obtained by considering other agents. However, the movements carried out by high and mid-rank agents may lead to the local optimum. Therefore, we need a mechanism that does not require taking into account the already obtained solutions so as not to be trapped at the local optimum. This is where the low-rank agents play the role of performing random searches so they can explore new locations. Since the random search sometimes leads to uncertain results, assigning this search to low-rank agents is appropriate. This design enables us to never lose the advantage of exploratory search even when other solutions start converging at certain points. While many traditional population-based-metaheuristic algorithms encounter decreasing benefits of exploration search as iterations increase and solutions tend to converge to certain points [12, 11], the spy algorithm is able to maintain and improve the solutions that have been obtained while maintaining exploration.

The whole design of the spy algorithm is to provide assurance that each type of movement will always exist in every iteration. Each solution is only allowed to perform one type of movement and the consideration is determined on the basis of its solution quality that is converted into a ranking system. Although the spy algorithm applies a ranking system, its application differs from rank selection applied in other metaheuristics, e.g., the GA [32]. To fit the problem to be solved, the balance of occurrence of the three types of movements is set with the parameters used to determine which solution is categorized as high, medium, or low quality.

Exploitation is implemented by *SwingMove* while *random movement* is for exploration. The *MoveToward* tends to be exploratory at the beginning but gradually turns into exploitation as the iteration increases and the solutions get closer to each other. The ability of the spy algorithm to converge relies on *SwingMove* and *MoveToward*. This concept enables the spy algorithm to take advantage of exploration and exploitation as well as use cooperative and non-cooperative strategies for solution refinement. The spy algorithm organizes all these aspects to occur separately while guaranteeing its presence in each iteration.

The spy algorithm can be simply implemented using sorting to arrange the solutions. There are four main parameters that affect the performance, i.e., the number of solutions ($NSol$) that represents the number of spy agents, the maximum index for high-rank (HMI), the maximum index for mid-rank (MMI), and the swing factor (SF) to bound the perimeter. The HMI, MMI is the key parameter that can give before-running intuition of the occurrence of exploration and exploitation. As the result, a balance between exploration and exploitation is easy to reach just by setting those parameters. Another parameter is the number of iterations (NI) as a stopping criterion commonly used in metaheuristic algorithms.

Based on the design, the occurrence of exploration and exploitation searches is easy to predict. As the spy algorithm applies fixed portions, it is clear that the exploration search will always exist with the rate of $1 - MMI$ at minimum, which is implemented by random moves by low-rank agents. Further, the exploitation will always occur at the rate of HMI at minimum, which is implemented by *SwingMove* by high-rank agents. The *MoveToward* will add a large amount of exploration in the early iterations and tends to turn into exploitation as iterations increase. However, the lack of exploration power will not occur as the algorithm maintain the random moves performed by low-rank agents.

2.3.2 Implementation

The objective function f should be defined by using the variable X. We assume the dimension of variable X is $D \in \mathbb{N}$ so that $X = (x_1, x_2, ..., x_D)$. This variable represents the location of an agent in D dimensional space. Since there are $NSol$ agents, $NSol$ values of the objective function are obtained from $NSol$ locations. We denote each location and its value by X^μ and $f(X^\mu)$, respectively, where $\mu = 1, 2, ..., NSol$. The main concepts of the spy algorithm are very simple and can be implemented in many various ways. One of its implementations can be seen on the pseudocode in Fig. 2.5. The ascending sorting can be used to simply determine the rank of each agent. The best agent that gives the smallest value should be placed at the first position and the largest at the last position. It should be noted that refinements can be carried out starting from the agent with the lowest to the highest rank. This strategy is to maintain the position of the higher rank agent to ensure that the rank, as well as its location, does not change before it is used as a reference by other lower rank agents. For clarity, we propose simple movements for *SwingMove* and *MoveToward* on the basis of the assumption that the algorithm works in discrete time t related to the iteration. We denote the location of the μ-th agent $X^\mu = (x_1^\mu, x_2^\mu, ..., x_D^\mu)$ at time t by $X^\mu(t)$, for $t = 1, 2, .., (NI - 1)$.

*** SwingMove**

$$X^\mu(t+1) := X^\mu(t) + rand(-1, 1)(SF/t) \tag{2.1}$$

*** MoveToward (assume that agent X^μ move toward X^ν)**

$$\nu := randint(1, \mu - 1) \tag{2.2}$$

$$X^\mu(t+1) := X^\mu(t) + rand(-1, 1)(X^\nu(t) - X^\mu(t)) \tag{2.3}$$

The function $rand(-1, 1)$ is for generating a vector of real random numbers within $[-1, 1]$ that allows movement in any directions. In *MoveToward*, the considered agent is a randomly selected better agent X^ν. The function $randint(1, \mu - 1)$ is used for generating an integer random number ν within $[1, \mu - 1]$ where μ is the index of the agent that performs *MoveToward* movement. Based on the pseudocode in Fig. 2.5, the process of selecting agent X^ν can be done before going to the *MoveToward* function.

Considering that the new solutions must be within the search space, there needs to be an additional simple mechanism to control the *SwingMove* and *MoveToward*. This mechanism returns the value of each element of the solution vector that is outside the search space to the nearest bound of the search space.

Unlike most metaheuristic algorithms, the spy algorithm implements a concept that each solution is only allowed to perform one type of movement at the refinement step, which is (*SwingMove*), (*MoveToward*), or random search. Nevertheless, the spy algorithm assures all these

```
//set up the problem
objective function: a D-dimensional function f
search space I= <I_i>, where I_i=[a_i,b_i] for i ∈ D

//set up the algorithm parameter value
NSol                  //number of solutions
HMI,MMI ∈ (0,1]       //as the ratio of the number of solutions
SF ∈ ℝ+               //Swing Factor
NI                    //number of iterations

HMI = int(HMI*NSol) //High.MaxIndex
MMI = int(MMI*NSol) //Mid.MaxIndex

//init
Sol = random_search(I) //create the initial solutions by global search
fSol = f(Sol)          //get the objective values
sort(Sol, fSol)        //best solution get smallest index

//refinement
for iter: 1 to NI
    for i: NSol downto (MMI-1)                  //low-rank agents
        NewSol[i] = random_search(I)
    for i: MMI downto (HMI-1)                   //mid-rank agents
        j= randint(1, i-1)          //select other solution j, 1≤j<i
        NewSol[i] = MoveToward(Sol[i], Sol[j])
    for i: HMI downto 1                         //high-rank agents
        NewSol[i] = SwingMove(Sol[i], SF)

    fNewSol=f(NewSol)                   //get the objective values

    //updating
    for i:1 to NSol
        if fNewSol[i] <= fSol[i]
            Sol[i] = NewSol[i]
            fSol[i] = fNewSol[i]
    sort(Sol,fSol)

//finalization
FinalSol=Sol[1]      //the best solution
```

Figure 2.5: Pseudocode of the spy algorithm

movements will always occur in each iteration. Therefore, a proper arrangement is needed to guarantee the occurrence of each movement. One of the strengths of population-based meta-heuristic algorithms is the cooperative search which combines previously obtained solutions to create better new solutions. Considering this benefits, it is necessary to set that there is an adequate number of agents in the category of mid-rank agent to perform *MoveToward*. As *SwingMove* and *random search* are non-cooperative searches that do not take benefit from other solutions, it is reasonable to set the number of high-rank and low-rank agents small. Figure 2.6 shows how the parameters affect the number of agents in each category.

It is common that, in all population-based algorithms, cooperative search dominates the search processes. In the case of GA, new solutions are created more through crossover than through mutation. The spy algorithm accommodates the cooperative search by *MoveToward* performed by mid-rank agents. To get the benefits of population-based algorithms, it is necessary to arrange for the spy algorithm to have a sufficient number of mid-rank agents. From our experiments, it is suggested that the number of the mid-rank agent is around 75%-95% of the total solution ($NSol$), with the rest being for either high or low-rank agents.

The value of HMI parameter that regulates the number of high-rank agents will affect the performance of the spy algorithm. A small HMI value will make the spy algorithm run longer than using a large HMI. A large HMI value will reduce the number of cooperative searches so that the algorithm can run faster but also risks reducing accuracy. In the tuning process, the accuracy decreases quite a lot when the HMI value is more than 0.15 which means the number of high-quality agents is 15% of the total solution ($NSol$). Note that in order to still get the

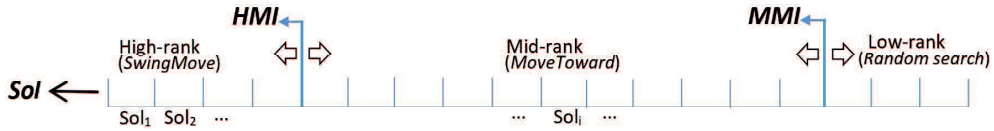


Figure 2.6: Solution distribution in each category

benefits of exploratory search, the spy algorithm needs to maintain the presence of the low-rank agents.

We propose two variants of the spy algorithm. The first variant uses only a single agent in the high-rank category. Since the GA variant may use elitism, where the algorithm preserves the best individual and pass it over to the new generation, this idea became a motivation to be loosely adapted into the spy algorithm. However, GA and the spy algorithm apply a different strategy. The best solution in GA may not be modified as the changes only apply to the selected solutions, but the spy algorithm always tries to improve each solution including the best one. Another inspiration came from PSO where the swarm move by considering the global-best location. This situation also inspired the global-best HS, which is a variant of IHS [31]. As the best solution should be only a single solution, we designed the first variant of the spy algorithm, where the high-rank category consists of a single solution only. This case is also common in spy agencies where each agent competes to be the best one. Applying a single high-rank agent will result in increasing the progressive search implemented by *MoveToward* on mid-rank agents. This design will increase the occurrence of referrals to the best agent in the solution refinement step. The first variant can be obtained by setting parameter $HMI = 1/NSol$ or by setting it in the code, so the number of the high-rank agent is directly set to 1. To make it simple, the first variant is referred to as Spy1. The second variant is the case where the high-rank category consists of more than one agent, and it is referred to as Spy2.

The Spy2 allows the algorithm to have several distinct solutions that are considered high quality. In the case of a problem having many solutions, it is advantageous to have many agents to exploit many basins. In the context of the spy algorithm, exploitation is accommodated by *SwingMove*, which is performed by high-rank agents. When some agents have more information than others, they have a greater chance of finding the best solution. Therefore, a slight movement is recommended for those agents rather than always moving progressively and exploring large areas. Considering the whole process of the spy algorithm, increasing the number of high-rank agents will reduce the number of mid-rank agents assuming that the low-rank agents are constant. The result is an increase in the occurrence of *SwingMove* and a decrease in *MoveToward*. Since the *SwingMove* process is simpler than *MoveToward*, the algorithm may run faster.

Spy1 and Spy2 will have different portions of exploration and exploitation searches. As Spy2 has more high-rank agents, the exploitation power will be stronger than Spy1. However, both will have an equal minimum exploration rate. Using two versions, we can investigate how to achieve a balance between exploration and exploitation.

2.4 Evaluation

2.4.1 Test Condition

Two tests were conducted for investigating the performance of the spy algorithm, i.e., (i) optimization on multi-dimensional function, and (ii) multimodal optimization. Test (i) also involved multimodal functions, but the focus was on finding the global optimum. Test (ii) focused on finding all global optimums, which means how many global optimum points can be detected

with a certain algorithm. For comparison, Spy1 and Spy2 were tested with the three population-based metaheuristic algorithms of the GA [33], IHS [30], and PSO [34]. A real-coded GA with simple arithmetic operators was used to give an equal condition, as the other tested algorithms used simple arithmetic operators as well. Tournament selection was used in the GA because of its stability compared with roulette wheel selection. The elitism was applied to GA. All tested algorithms were in their basic version and did not use any specific approach enabling us to investigate the original potential of each algorithm.

The optimum points for each test function were known so that the performance of each algorithm could be properly measured. We made all tested functions have an optimum value of 0 by normalizing several functions. On test (i), we set the dimension size to 30 to gain adequate insight into how the algorithm works on large dimensional problems. The dimension size was set to 2 for the test (ii) so we could get visual results for better understanding. All test functions are non-convex functions and listed in Table 2.1. The plots of the 2-dimensional version of the test functions are shown in Fig. 2.7. Unlike optimizations on convex functions, which can be solved in polynomial time, optimizations on non-convex functions are more difficult to solve since there are many local optimums, valleys, or plateaus that can trap the algorithm so that it fails to find a global optimum. The optimization of the non-convex function is categorized as NP-hard [22], so it is suitable for testing metaheuristic algorithms.

To adjust to the test and the characteristics of the problem, a different set of algorithm parameter values was tuned. The solution (population) size was set to be equal for each algorithm. The NI was set at 50 times the size of the problem dimension. Since IHS only generates one new solution per iteration, the NI for IHS was set at 50 times the dimension size times the harmony memory size (HMS) to make it equal. Other algorithm parameter values are listed in Table 2.2. These values were tuned to obtain an equal condition that takes into account the number of function evaluations and the search balance. All algorithms were implemented in Python code. The code was run using Python 3.9.4 on a Windows 10 PC powered by Intel i7-9750H and 16-GB of RAM.

Each algorithm was run in 100 independent repetitions to obtain sufficient data to observe its behavior. The performance criteria were mainly based on the average error and its standard deviation, but because test (ii) is also for investigating the potential for finding all global optimum points (GOPs), it has an additional criterion, which is the maximum peak ratio (MPR).

$$MPR = \frac{Number\ of\ detected\ GOPs}{Number\ of\ all\ actual\ GOPs}. \tag{2.4}$$

When a solution falls near a certain GOP location, which means the error E is less than ε, that is,

$$E = \|X_{opt} - \hat{X}\| < \varepsilon, \quad \varepsilon \in \mathbb{R}^{+}, \tag{2.5}$$

and falls within the same narrow basin of the GOP, we count it as able to detect the GOP even though the obtained solution might differ from the exact one. We used $\varepsilon = 0.1$ for test (ii) considering that all tested algorithms were set to use a small number of solutions and iterations. The main purpose of test (ii) is to investigate the ability of the tested algorithms to distribute their set of solutions to reach many distinct GOPs. The last criterion for each test is the computation time taken by the algorithm.

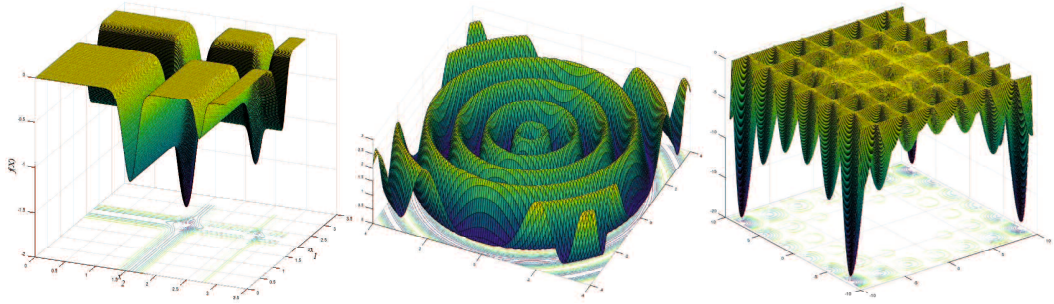
2.4.2 Experimental Result

We tested the spy algorithm, GA, IHS, and PSO on various problems having various characteristics to investigate their accuracy, speed, and robustness. The descriptive statistical results

Table 2.1: Test functions

Test	Name	Test function	Dim (D)	Interval I_i	GOP
(i)	Michalewicz*	$f_1 = -\sum_{i=1}^{n} \sin(x_i) sin^{20}(\frac{ix_i^2}{\pi})$	30	[0,π]	1
	Rosenbrock	$f_2 = \sum_{i=1}^{n} \left(100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right)$	30	[0, 10]	1
	Alpine01	$f_3 = \sum_{i=1}^{n} \lvert x_i \sin(x_i) + 0.1x_i \rvert$	30	[-10, 10]	1
	Ackley	$f_4 = -20e^{\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right)} - e^{\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right)} + 20 + e$	30	[-30, 30]	1
	Salomon	$f_5 = 1 - \cos\left(2\pi\sqrt{\sum_{i-1}^{n} x_i^2}\right) + 0.1\sqrt{\sum_{i-1}^{n} x_i^2}$	30	[-100, 100]	1
	Griewank	$f_6 = 1 + \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} \cos\left(x_i/\sqrt{i}\right)$	30	[-600, 600]	1
(ii)	Bird*	$f_7 = \left(\sin x_1 e^{(1-\cos x_2)^2} + \cos x_2 e^{(1-\sin x_1)^2} + (x_1 - x_2)^2\right)$	2	[-2π, 2π]	2
	Cross in Tray*	$f_8 = -0.0001\left(\left\lvert \sin x_1 \sin x_2 e^{\left\lvert 100 - \frac{\sqrt{x_1^2+x_2^2}}{\pi}\right\rvert}\right\rvert + 1\right)^{0.1}$	2	[-10, 10]	4
	Holder Table*	$f_9 = -\left\lvert \sin x_1 \cos x_2 e^{\left\lvert 1 - \frac{\sqrt{x_1^2+x_2^2}}{\pi}\right\rvert}\right\rvert$	2	[-9.7, 9.7]	4
	Himmelblau	$f_{10} = \left(x_1^2 + x_2 - 11\right)^2 + \left(x_1 + x_2^2 - 7\right)^2$	2	[-6, 6]	4
	Shubert*	$f_{11} = \prod_{i=1}^{n}\left(\sum_{j=1}^{5} j\cos\left((j+1)x_i + j\right)\right)$	2	[-10, 10]	18
	Inv. Vincent	$f_{12} = \frac{1}{n}\sum_{i=1}^{n}\sin\left(10\log x_i\right)$	2	[0.2, 10]	36

*: normalized by subtracting it with the optimum value



(a) f_1, Michalewicz (b) f_5, Salomon (c) f_9, Holder Table

Figure 2.7: 2-dimensional version of the test functions

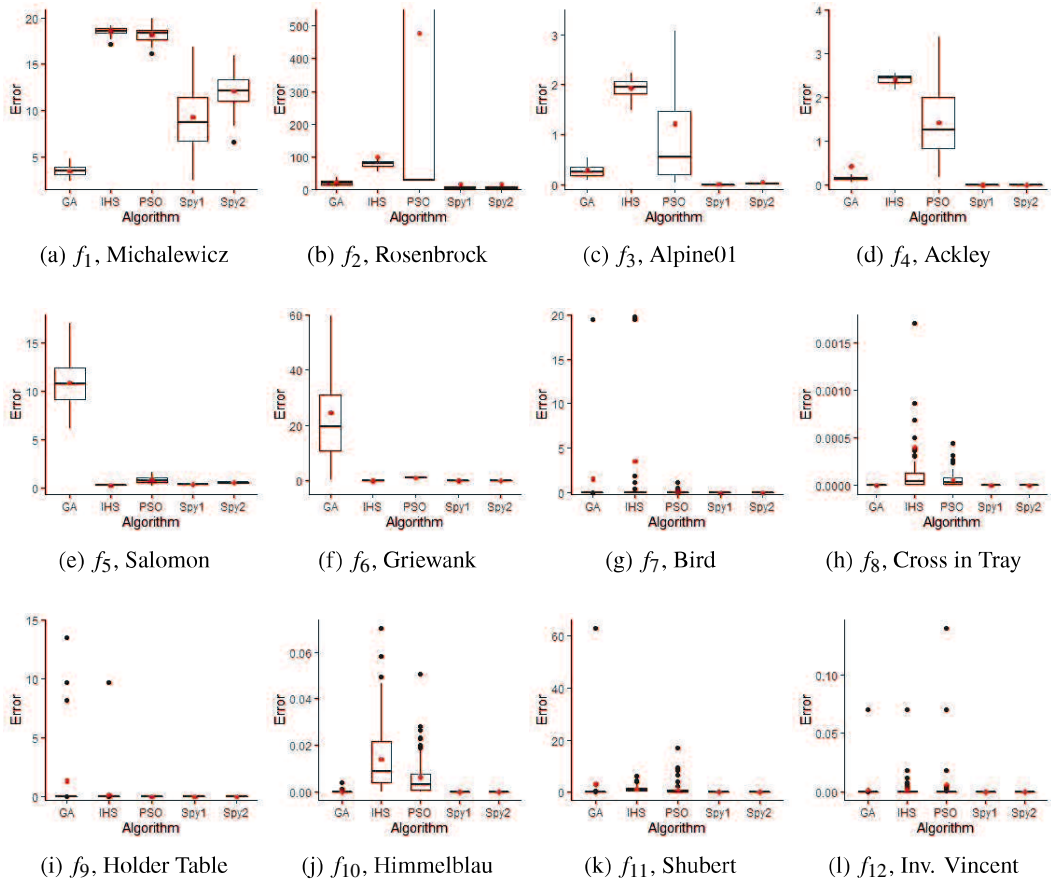
are listed in Table 2.3. The spy algorithm provided the most accurate results in most test functions. Even though the spy algorithm performed poorly in two test functions (f_1 and f_5), Spy1 and Spy2 did not perform the worst. The boxplot of error and the average error are shown in Fig. 2.8. From these results, the spy algorithm tended to have small errors and small standard deviations. These results differed from GA, IHS, and PSO, whose results tended to vary markedly at different test functions. The results indicate that the spy algorithm was more robust than the other tested algorithms.

Of all tests, the spy algorithm performed worse in accuracy for the Michalewicz (f_1) function. This algorithm was worse than the GA, although it was better than IHS and PSO. As the representation of the characteristic, the 2-dimensional Michalewicz function has a contour in

Table 2.2: Algorithm parameters

GA	**IHS**	**PSO**	**Spy1**	**Spy2**
Pop= 40	HMS=40	NSwarm=40	NSol=40	NSol=40
Tourn= 10	HMCR= 0.85	c1= 0.9	HMI= 1/NSol$^{(*)}$	HMI= 0.1
Parent= 10	minPAR= 0.7	c2= 1	MMI=0.9	MMI= 0.9
pc= 0.9	maxPAR= 0.85		SF=1	SF=1
pm= 0.2	minBW= 0.5			
StepSize=1	maxBW= 1.5			
NI= 50D	NI= 50D*HMS	NI= 50D	NI= 50D	NI= 50D

D: dimension size
(*): the high-rank agent can also directly be set fix at 1



(a) f_1, Michalewicz (b) f_2, Rosenbrock (c) f_3, Alpine01 (d) f_4, Ackley

(e) f_5, Salomon (f) f_6, Griewank (g) f_7, Bird (h) f_8, Cross in Tray

(i) f_9, Holder Table (j) f_{10}, Himmelblau (k) f_{11}, Shubert (l) f_{12}, Inv. Vincent

Figure 2.8: Boxplot of error, the red dots show the average values

the form of valleys as well as plateaus which can trap the algorithm so that it fails to approach the global optimum point. Such a characteristic did not exist in other tested functions. On the Michalewicz function, GA showed its superiority over the others. For functions similar to the Michalewicz function, the spy algorithm may perform worse than the GA. Considering that the spy algorithm was better than IHS and PSO in this case, it is possible to improve the performance of the spy algorithm by adjusting the parameters.

To provide a more in-depth investigation of the differences between each algorithm, we conducted a Kruskal–Wallis H test using $\alpha = 0.05$ on each test function. Our test results on

Table 2.3: Average error ± Standard deviation

f	GA	IHS	PSO	Spy1	Spy2
f_1, Michalewicz	**3.501**	18.556	18.181	9.311	12.387
	±0.564	±0.423	±0.776	±3.378	±1.461
f_2, Rosenbrock	21.674	98.344	477.124	**17.157**	22.856
	±10.316	±60.779	±1429.226	±29.856	±28.084
f_3, Alpine01	0.284	1.922	1.229	**0.019**	0.250
	±0.141	±0.267	±1.681	0.079	0.999
f_4, Ackley	0.431	2.396	1.434	**7.617e-6**	4.213e-4
	±1.742	±0.115	±0.776	±.972e-6	±1.504e-4
f_5, Salomon	10.909	**0.285**	0.856	0.426	0.62
	±2.170	±0.031	±0.336	±0.056	±0.077
f_6, Griewank	24.640	0.203	1.013	0.004	**0.002**
	±20.167	±0.115	±0.195	±0.011	±0.008
f_7, Bird	1.559	3.595	0.053	1.041e-6	**1.415e-7**
	±7.155	±7.521	±0.140	±5.553e-6	±1.045e-6
f_8, Cross in Tray	2.855e-7	3.973e-4	5.553e-5	3.592e-9	**5.826e-10**
	±4.595e-7	±0.003	±7.386e-5	±2.251e-8	±3.681e-9
f_9, Holder Table	1.368	0.199	0.011	2.186e-6	**2.983e-7**
	±3.318	±1.364	±0.004	±6.678e-6	±1.442e-6
f_{10}, Himmelblau	2.286e-4	0.014	0.007	8.502e-7	**6.126e-7**
	±4.968e-4	±0.014	±0.008	±4.269e-6	±2.536e-6
f_{11}, Shubert	3.207	1.083	0.961	0.003	**7.885e-4**
	±13.836	±1.057	±2.216	±0.004	±0.001
f_{12}, Inv. Vincent	0.001	0.002	0.007	7.759e-7	**4.342e-7**
	±0.010	±0.007	±0.021	±2.800e-6	±2.139e-6

*__in bold__: smallest among others

each test function indicate that there were significant differences among these algorithms. To know the detail, we conducted the Gomes-Howell post hoc comparison. The P-values of this comparison are listed in Table 2.4. Considering the average error values and these P-values, the spy algorithm significantly gave the best results or was always in the best group. These results also showed that the two variants of the spy algorithm had significant differences in several test functions. Table 2.3 shows that Spy1 tended to be better than Spy2 on the high-dimensional test functions while Spy2 was slightly better than Spy1 on the 2-dimensional test functions.

Without changing the algorithm parameter values, on test (ii), we tested all algorithms to detect as many GOPs as possible. Combined with test (i), test (ii) will provide a more deep investigation of the robustness of the algorithm. The visual results of detecting GOPs on f_{11} are shown in Fig. 2.9. The diamonds denote that there are solutions that fall near the location of the global optimum, and triangles indicate that no solution lies near the considered location of the global optimum. The averages of all maximum peak ratios (MPRs) are listed in Table 2.5. In this test, the GA and IHS performed poorly with low MPR results. These results indicate that the GA and IHS were weak in detecting many GOPs and tended to converge to a few. PSO provided a good enough MPR so that it was suitable to use to find many GOPs. From these results, the spy algorithm outperformed PSO, as seen from the MPR, which was greater than that of PSO. Both Spy1 and Spy2 had similar accuracy in detecting many GOPs.

In terms of time performance, we computed the aggregate computation times regardless of the test function. Although the complexity of the test function affected computation time, the differences were not very large. As the time difference is more affected by the dimensions of the problem, we separated test (i), which applied the dimension size of 30, and test (ii), which applied the dimension size of 2. The computation times are shown in the boxplot chart in Fig. 2.10. We again performed Kruskal–Wallis H test using $\alpha = 0.05$ to compare computation

Table 2.4: P-values of Games–Howell post hoc error comparison

Comparison	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}
GA - IHS	<.001	<.001	<.001	<.001	<.001	<.001	0.780	<.001	<.001	<.001	0.534	0.958
GA - PSO	<.001	0.017	<.001	<.001	<.001	<.001	0.063	<.001	<.001	<.001	0.167	0.013
GA - Spy1	<.001	0.609	<.001	0.106	<.001	<.001	0.003	<.001	<.001	<.001	0.095	0.851
GA - Spy2	<.001	0.995	0.997	0.106	<.001	<.001	0.003	<.001	<.001	<.001	0.095	0.851
IHS - PSO	<.001	0.071	<.001	<.001	<.001	<.001	0.002	0.009	0.669	<.001	0.303	0.016
IHS - Spy1	<.001	<.001	<.001	<.001	<.001	<.001	<.001	<.001	0.609	<.001	0.055	<.001
IHS - Spy2	<.001	<.001	<.001	<.001	<.001	<.001	<.001	<.001	0.609	<.001	0.055	<.001
PSO - Spy1	<.001	0.015	<.001	<.001	<.001	<.001	0.398	<.001	<.001	<.001	<.001	0.002
PSO - Spy2	<.001	0.017	<.001	<.001	<.001	<.001	0.398	<.001	<.001	<.001	<.001	0.002
Spy1 - Spy2	<.001	0.634	0.154	<.001	<.001	0.471	0.613	0.383	0.058	1.000	0.012	0.951

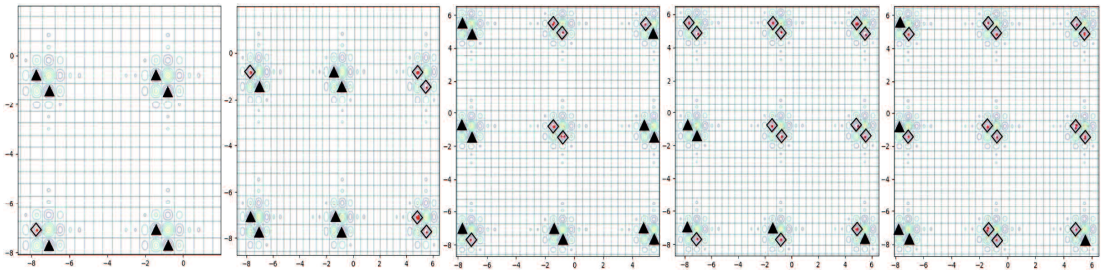
The difference is significant if $P - value < \alpha, (\alpha = 0.05)$

times. As the output of Kruskal–Wallis H test showed that there were significant differences, we followed it up with a post hoc comparison. The results are listed in Table 2.6. From these P-values, the only insignificant difference was between IHS and Spy1 for the dimension size of 2. From the results in Fig. 2.10 and Table 2.6, the spy algorithm was the fastest among the algorithms. Another important result was that Spy2 was faster than Spy1. The shortest computation time of Spy2 could be easily understood because Spy2 used more *SwingMove* and less *MoveToward* than Spy1. *SwingMove* took less computation time because the *SwingMove* has simpler operation than *MoveToward*.

The spy algorithm, GA, IHS, and PSO were tested on two tests without changing the algorithm parameters. The tests applied an equal condition for all tested algorithms, and all algorithm parameters were tuned up to suit the set of test functions as a whole. Based on the descriptive results and statistical analysis, the overall spy algorithm showed the best performances in accuracy, MPR, and computation time. These results indicate that the spy algorithm was more robust than the GA, IHS, and PSO. Changes in parameter values in the spy algorithm have a direct effect on the number of occurrences of *SwingMove*, *MoveToward*, and *random search*. These changes may have an impact on the performance. Spy2 was substantially faster than Spy1, but Spy1 tended to have better accuracy on high-dimensional test functions. Both Spy1 and Spy2 showed almost equally good ability for detecting many GOPs.

The spy algorithm performed well because its rule and all processes are very simple, so it can achieve a low computational cost. The spy algorithm is accurate because it uses two types of solution refinements. *SwingMove* is a slight refinement to avoid a sudden drop in solution quality. However, the SF has an important role in managing its change. A small SF tends to provide a new solution that is not much different from the previous solution. While the *SwingMove* preserves solution quality, the *MoveToward* performs a progressive search by benefiting the previously obtained location. These two strategies are well managed so that the spy algorithm can achieve high accuracy while reducing computational costs.

The similar good results between Spy1 and Spy2 show that different parameter settings did not have a significant impact. Based on this fact, it is shown that the algorithm has a consistent behavior. Spy1 and Spy2 have different settings for HMI, where Spy2 has stronger exploitation power. However, the exploitation searches in both Spy1 and Spy2 will increase as the iteration increase. Based on this common phenomenon, it is reasonable that they have similar results. Moreover, they have equal minimum exploration rates preserved from the first to the last iteration. So, it is easy to reach a balance state as the parameters HMI and MMI are fixed so that they can provide before-running intuition of the occurrence of exploration and exploitation searches.



(a) GA, detected=1 (b) IHS, detected=5 (c) PSO, detected=6 (d) Spy1, detected=13 (e) Spy2, detected=13

Figure 2.9: Detecting global optimums on Shubert function having 18 GOPs (red dots are solutions, diamonds means the solutions lie near a certain GOP while triangles are missed GOPs)

As the balance can be reached, the spy algorithm became the most robust algorithm compared to GA, IHS, and PSO. The robustness can be seen that Spy1 and Spy2 are the most successful algorithm for tackling two types of problems without changing algorithm parameters.

2.5 Conclusion

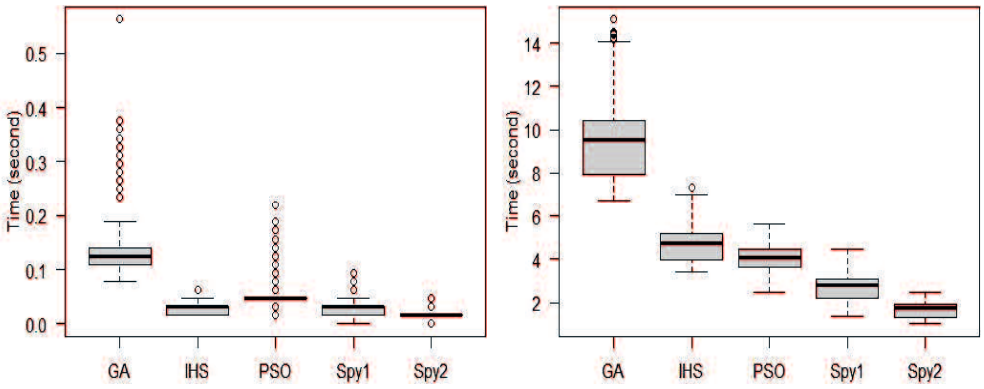
We proposed the spy algorithm, which is a population-based metaheuristic algorithm that ensures the benefit of exploration and exploitation as well as cooperative and non-cooperative searches in each iteration. Unlike many traditional population-based metaheuristic algorithms that loses exploration as the iteration increase, the spy algorithm is able to maintain exploitation as well as exploration search to improve the solutions. As the spy algorithm applies a fixed portion on three dedicated search operators, it provide a before-running-intuition of the occurrences of exploration and exploitation to achieve a balance state between them.

We tested the GA, IHS, PSO, and spy algorithm on a set of non-convex functions that focus on accuracy, the ability to detect many global optimum points, and computation time. We conducted a statistical analysis to gain insight into accuracy and computational cost. As a result, the spy algorithm outperformed GA, IHS, and PSO. It resulted in fewer errors and higher maximum peak ratios within less computation time, indicating that the spy algorithm was more robust than other tested algorithms.

Table 2.5: Average MPR

f	**GA**	**IHS**	**PSO**	**Spy1**	**Spy2**
f_7, Bird	0.47	0.49	0.495	0.99	0.96
f_8, Cross in Tray	0.25	0.2575	0.5675	0.925	0.9075
f_9, Holder Table	0.1425	0.2375	0.2525	0.9875	1
f_{10}, Himmelblau	0.25	0.255	0.305	0.7525	0.78
f_{11}, Shubert	0.054	0.0961	0.18	0.4556	0.4828
f_{12}, Inv. Vincent	0.0272	0.0464	0.1228	0.2925	0.2903

*larger is better, maximum is 1



(a) $D = 2$ (b) $D = 30$

Figure 2.10: Boxplots of computation time

Table 2.6: P-values of post hoc time comparison

Comparison	$D = 2$	$D = 30$
GA - IHS	<.001	<.001
GA - PSO	<.001	<.001
GA - Spy1	<.001	<.001
GA - Spy2	<.001	<.001
IHS - PSO	<.001	<.001
IHS - Spy1	0.891	<.001
IHS - Spy2	<.001	<.001
PSO - Spy1	<.001	<.001
PSO - Spy2	<.001	<.001
Spy1 - Spy2	<.001	<.001

The difference is significant if $P - value < \alpha, (\alpha = 0.05)$

Chapter 3

Constructing the Neighborhood Structure of VNS Based on Binomial Distribution for Solving QUBO Problems

3.1 Introduction

Combinatorial optimization has attracted a good deal of attention, as it has many applications in various fields [35]. However, it is not always easy to solve combinatorial optimization problems, especially in some cases classified as NP-hard problems. In this kind of situation, the use of the approximation method is a reasonable option [36]. One of the approximation methods that has attracted a great deal of attention in modern optimization is the metaheuristic method [37], which is designed to obtain good solutions within a reasonable time frame [38]. Even though it is not easy to prove that a solution obtained using the metaheuristic method is a global optimum [39], the results are often very close to the global optimum.

Depending on the case, a combinatorial problem can be formulated in various ways so that it can be be easily solved. One method is the use of a Boolean or binary vector, which, despite its simplicity, is a compelling method of solving many combinatorial problems. A specific binary formulation forms a major optimization problem category called "quadratic unconstrained binary optimization (QUBO)", or "Ising model optimization" in some of the literature [40]. In the QUBO problem, given $\boldsymbol{Q} = (q_{ij})$ is a symmetric n -square matrix of coefficients, the objective is to maximize the function:

$$f(\boldsymbol{X}) = \boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{Q}\boldsymbol{X} = \sum_{i=1}^{n}\sum_{j=1}^{n} q_{ij}x_i x_j \tag{3.1}$$

where $\boldsymbol{X} = (x_i)$ is an n-dimensional vector of binary variables, i.e., $x_i \in \{0, 1\}$ for $i = 1, 2, ..., n$. QUBO problems are categorized as NP-hard problems [41], and their decision form is NP-complete [40]. The implementation of the QUBO formulation can be found in many combinatorial problems, e.g., graph coloring, partition, and maximum cut (max-cut) [40], which were included in Karp's original 21 NP-complete problems [15]. More applications of QUBO were described by Glover and Kochenberger [42].

It is as difficult to solve QUBO problems as it is to solve other NP-hard problems. Many metaheuristic algorithms have been devised to solve them, based on the requirement that they are

solved within a reasonable amount of time. For the example, the simulated annealing applying 1-*opt* local search showed good performances for solving QUBO problems[43]. Another hybrid form build from genetic algorithm and 1-*opt* local search also provided good results [45]. The adaptive memory tabu search that select particular variable that give best improvement, so that the concepts is similar to local search, is also can be used for solving QUBO problem [44]. It can be seen that the local search has been a substantial role for enhancing other metaheuristic algorithm. However, the pure local search algorithm itself show a great performance [46].

One particular metaheuristic algorithm that uses local search is the variable neighborhood search (VNS) algorithm [47]. The VNS algorithm can solve various problems, including QUBO problems. As a single-trajectory-based algorithm, the VNS algorithm has the advantage that it is resource-efficient. Furthermore, its memory requirements are relatively low compared to population-based metaheuristic algorithms. Therefore, the VNS algorithm is suitable for use in solving large-scale problems and does not need large memory allocations. It is a simple concept, and the VNS algorithm is often used in its original or a modified form or is hybridized with another algorithm [48][49]. Moreover, there is an algorithm similar to VNS called greedy adaptive search procedure (GRASP) for solving QUBO problem. Unlike the VNS that start from random initial point and shifting to its neighboring point before local search, GRASP start from a greedy initial point and directly perform local search based on that point.

The VNS algorithm has been shown to perform well in various problems. Its applications include the max-cut combinatorial problem [50, 51], the scheduling problem [52, 53], the layouting problem [54], the vehicle routing problem [55], and the multiprocessor scheduling problem with communication delays [56]. It has also been used to tune proportional–integral–derivative controllers in cyber–physical systems [57]. Not only has the VNS algorithm been applied in numerous ways, but it has also been often used in combination with other algorithms to obtain combined performance. It is possible to hybridize the VNS algorithm with other metaheuristic algorithms, such as with the genetic algorithm [58], the particle swarm optimization algorithm [59], the migrating birds optimization algorithm [60], and the simulated annealing algorithm [61]. The VNS algorithm can also be implemented in parallel programming by using a graphical processing unit [62] to leverage its performance.

Many factors determine the performance of the VNS algorithm, i.e., the initialization method, neighborhood structure construction, local search procedures, and update mechanisms. Although there are many determining factors, neighborhood structure is the central concept of the VNS algorithm that significantly influences its performance. One version of the VNS algorithm is based on the dynamic neighborhood model proposed by Mladenović and Hansen [47]. Changing the neighborhood structure during a search enables the algorithm to escape the local optimum trap [63], as it allows the algorithm to move from one basin of search to another. The construction of a neighborhood is thus crucial to the performance of the VNS algorithm. However, in terms of solving the QUBO problem, previous research reports regarding neighborhood construction are difficult to find. The Hamming distance is commonly used in the basic VNS algorithm to construct the neighborhood when solving a QUBO problem [64, 65, 50].

The VNS algorithm uses a strictly monotonic increasing neighborhood structure when the local search does not yield a better solution. As a result, the basic VNS algorithm takes quite a long time to yield a good solution. A new version called "Jump VNS" was introduced to speed up the neighborhood construction process. It enables the neighborhood structure to leap ahead in accordance with parameter $k_{step} \in \mathbb{N}$ [66]. The basic VNS algorithm that does not have the jump ability is obtained by setting $k_{step} = 1$. The performance of the Jump VNS algorithm does not appear to have been previously reported.

The VNS algorithm is simple, and the ability to slowly change the neighborhood structure is

an advantage of its use. However, this is also a weakness of the VNS algorithm. This gradual but slow change may impact the length of computation time. Although it may be sped up with Jump VNS, the thorough search behavior may be lost. For example, setting the maximum distance on a VNS algorithm to 100 results in 100 times neighborhood structure change at most, but setting $k_{step} = 2$ on Jump VNS results in only 50 times neighborhood structure change. This value is even less if a larger k_{step} is used. So, the VNS and Jump VNS algorithms are not flexible.

This paper elaborates on the basic VNS algorithm and introduces a new method of improving it by focusing on the neighborhood structure by implementing binomial distribution. Instead of a strictly monotonic increase in neighborhood structure, the neighborhood distance follows a binomial distribution. Although the binomial distribution will cause a non-monotonic increase, the trend of a widening structure will remain the same. We investigated the potential of our proposed algorithm to be used in some QUBO and max-cut problems.

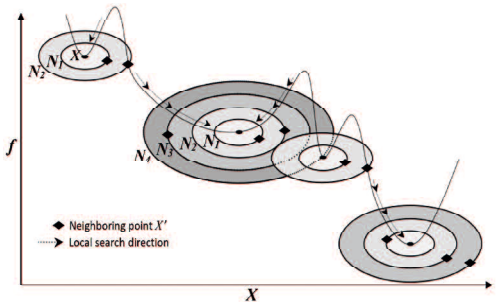
3.2 VNS Algorithm

The VNS algorithm is a well-known metaheuristic algorithm that utilizes dynamic neighborhood structure changes. The simple implementation of the VNS algorithm starts from a non-deterministic *guest* initial point and then attempts refinements using a local search. The algorithm shifts from its initial point to a neighboring point before a local search is carried out. The algorithm should move to another neighborhood structure when the local search does not yield a better solution. The algorithm systematically exploits the following observations [67]:

> *Observation 1*: A local minimum for one neighborhood structure is not necessarily so for another;
>
> *Observation 2*: A global minimum is a local minimum for all of the possible neighborhood structures;
>
> *Observation 3*: For many problems, local minimums for one or several neighborhoods are relatively similar to each other.

In other words, according to observation 1, a local minimum for a specific neighborhood structure is not necessarily a local minimum for other neighborhood structures. Other neighborhoods may have other local optimums. The second observation means that the global minimum will only be found after examining all of the possible local optimums, which requires the examination of all of the possible neighborhood structures. Other neighborhoods should be examined if a local minimum is not the global minimum. The last observation is an empirical observation that suggests a local optimum usually provides information that helps determine the global optimum [67]. For example, in the case of a multi-variable function, several variables often have the same value in the local optimum as in the global optimum [67].

There are several versions of the VNS algorithm; the most prominent is the dynamic neighborhood model proposed by Mladenović and Hansen [47]. They proposed a random shift to a neighboring point $\boldsymbol{X'}$ which would be used instead of the initial point $\boldsymbol{X}$ as the base point of a local search. They called this shifting "shaking." In this model, if the local search does not yield an improvement, the neighborhood structure is expanded. This change enables the algorithm to move to another basin to be exploited. To retain efficiency, the change must be limited by a parameter that defines the maximum number of neighborhood structures examined. If the local search yields an improvement, the structure is suddenly shrunk back to the first structure. The use of this clever expand-and-shrink neighborhood structure during a search enables the VNS



(a) Searching process. [66]

```
procedure VNS( f, MaxIter, k_max )
1   Solution ← ∅
2   for i=1,MaxIter:
3       randomly generate X                    //initial point
4       k ← 1
5       while k ≤ k_max:
6           randomly generate X' ∈ N_k(X)      //shaking
7           X'' ← local_search(X')
8           if f(X'') better than f(X):
9               X ← X''                        //make a move
10              k ← 1                          //back to first neihgborhood
11          else:
12              k ← k + 1                      //next neighborhood
13      Sol ← (x, f(X))
14      Update(Solution, Sol)
15  return Solution
end procedure VNS
```

(b) Pseudocode.

Figure 3.1: VNS algorithm.

algorithm to avoid the local optimum trap [63] because the algorithm moves from one basin to another. Figure 3.1(a) shows the searching process [66] in the context of minimization. The pseudocode of the VNS algorithm is shown in Fig. 3.1(b).

A change in neighborhood structure can be illustrated as an expanded disc with point $\boldsymbol{X}$ at the center. The neighborhood structure expands following variable $k \in \mathbb{N}$. As proposed elsewhere [50, 68, 65], neighborhood structure N is associated with variable k, which is limited by parameter $k_{max} \in \mathbb{N}$. Thus, Hamming distance d increases following k when solving QUBO problems. Accordingly, a point $\boldsymbol{X'} \in N_k(\boldsymbol{X})$ means that the Hamming distance $d(\boldsymbol{X'}, \boldsymbol{X}) = |\boldsymbol{X'} - \boldsymbol{X}|$ is exactly k. This neighborhood structure is the standard that is used in solving QUBO problems. The neighborhood structure of the basic VNS algorithm for use in solving QUBO problems is formulated as

$$N_k(\boldsymbol{X}) = \{\boldsymbol{Y} : |\boldsymbol{Y} - \boldsymbol{X}| = k\}, \tag{3.2}$$

for $k = 1, 2, ..., k_{max}$.

3.3 Proposed Neighborhood Structure

The strictly monotonic expanding neighborhood structure used in the VNS algorithm is tenable. However, there is no guarantee that this expansion type will always result in a better solution. To exploit this technique to solve any QUBO problem, we propose the use of binomial distribution to create a neighborhood structure. As the proposed algorithm is based on the VNS algorithm, it takes advantage of the characteristics of the VNS algorithm while aiming to reduce the computation time required.

Our proposed structure enables the algorithm to expand the neighborhood by following a random schema. However, it is good to maintain a gradual expansion trend. In the basic VNS algorithm, the neighboring point $\boldsymbol{X'} \in N_k(\boldsymbol{X})$ is obtained by flipping k numbers of the element of $\boldsymbol{X}$. The flipped elements are chosen randomly. As a result, some elements are changed, from 1 to 0 or vice versa, and some remain. Instead of applying this basic neighborhood structure, we propose a mechanism of determining whether each element should be flipped or not. Each element will be flipped by considering a flip probability. The proposed neighborhood structure is as follows: We define a trial T to flip each element of a vector $\boldsymbol{X} = (x_1, x_2, ..., x_n)$ to obtain $\boldsymbol{X'} = (x'_1, x'_2, ..., x'_n) \in N(\boldsymbol{X})$ in accordance with the following rule:

$$x'_i = \begin{cases} flip(x_i) & \text{, with probability } p \in [\,0, 1] \\ x_i & \text{, with probability } (1 - p) \end{cases}, \tag{3.3}$$

where

$$flip(x_i) = \begin{cases} 1 - x_i & \text{, for } \{0, 1\} \text{ encode} \\ -x_i & \text{, for } \pm 1 \text{ encode} \end{cases}. \tag{3.4}$$

This trial satisfies binomial requirements as it is repeated n times, where n corresponds to the vector length and sets a fixed probability p. Suppose a random variable A represents the number of flipped elements, and let a random variable D represent the obtained Hamming distance d. Then, there is a clear correspondence between random variable A and D. Take any vector X and generate a vector Y by following Equation 3.3, and suppose that m random elements of X were flipped based on this process; from this supposition, we have $d(X, Y) = |X - Y| = m$. As the flip is controlled by probability p, by following binomial distribution, we have

$$\mu_D = np, \tag{3.5}$$

$$\sigma_D = \sqrt{np(1-p)}. \tag{3.6}$$

Based on binomial distribution, the distance at approximately np has a high probability of occurring.

In the VNS algorithm, the distance is equal to variable k and is limited by the parameter k_{max}. Therefore, k_{max} is the maximum distance that can be reached when constructing a neighborhood structure. To replace this concept, our proposed method uses a parameter $p_{max} \in [0, 1]$ as a limitation. To apply the gradual expansion mechanism, we divide the p_{max} into several equal chunks. Then, we ensure that the flip probability p corresponds to these chunks.

$$p_c = cp_{max}/C, \tag{3.7}$$

for $c = 1, 2, .., C$, where $C \in \mathbb{N}$ is a parameter for chunk size. This construction is applied every time a neighboring point $\boldsymbol{X'}$ needs to be generated. Thus, this binomial neighborhood structure does not depend on variable k but instead on flip probability p_c which corresponds to chunk c.

$$N_c(X) = \{Y : D_{(Y,X)} \sim Binom(p_c, n)\}, \tag{3.8}$$

for $c = 1, 2, ..., C$ considering (3.7), $D_{(Y,X)}$ is the Hamming distance between Y and X, where Y is generated by applying Equation 3.3.

The distribution of obtained neighboring point $\boldsymbol{X'}$ corresponds to the probability density function of the binomial distribution, as shown in Fig. 3.2. The concept is advantageous because the distance of neighboring points $\boldsymbol{X'}$ can be estimated even though they are random. The proposed neighborhood structure is illustrated in Fig. 3.3(a). Note that the illustration is a simple version, as $\boldsymbol{X'}$ can be obtained in any direction. The neighborhood expansion corresponds to flip probability p_c following the variable chunk c. The distance between neighboring point $\boldsymbol{X'}$ and $\boldsymbol{X}$ will be random, even though it will follow the characteristic of binomial distribution. In contrast, in the basic VNS algorithm, the neighborhood structure can be illustrated as an expanded disc, as shown in Fig. 3.3(b). In the basic VNS algorithm, the expansion corresponds to variable k and results in $d(\boldsymbol{X'}, \boldsymbol{X})$ being exactly k.

The complete implementation of our proposed method for solving QUBO problems is similar to that of the VNS algorithm, except for differences in the neighborhood construction. Unlike the basic VNS algorithm, we use parameter p_{max} to control the distance. However, just like VNS, neighborhood structure change should be limited. Therefore, we use parameter chunk

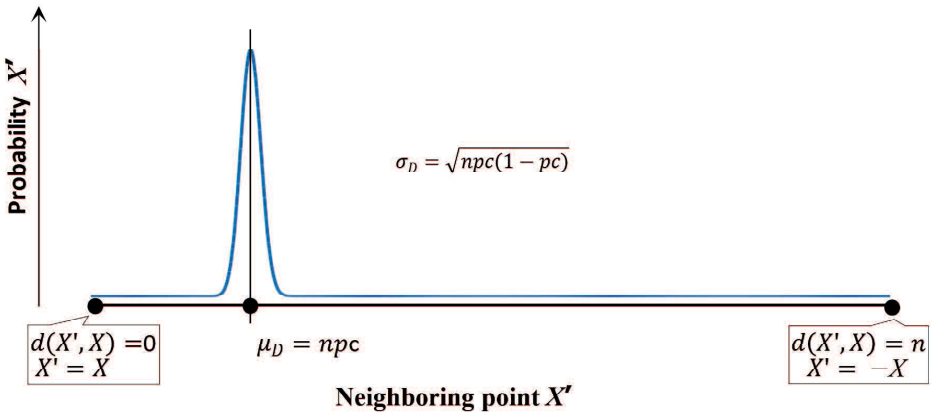


Figure 3.2: Distribution of $\boldsymbol{X'}$ related to p.

size C. Thus, we only change the construction of the neighborhood structure. For simplicity, we call our proposed algorithm "B-VNS", while "VNS" refers to the basic VNS algorithm [47, 50, 68, 65].

Changing the construction of the neighborhood structure will change the behavior. The B-VNS algorithm is more flexible than the VNS algorithm. In terms of k_{max} in the VNS algorithm, the B-VNS algorithm can reach almost the same neighborhood structure by setting up the parameter p_{max} that

$$\mu_D = np_{max} \equiv k_{max}. \tag{3.9}$$

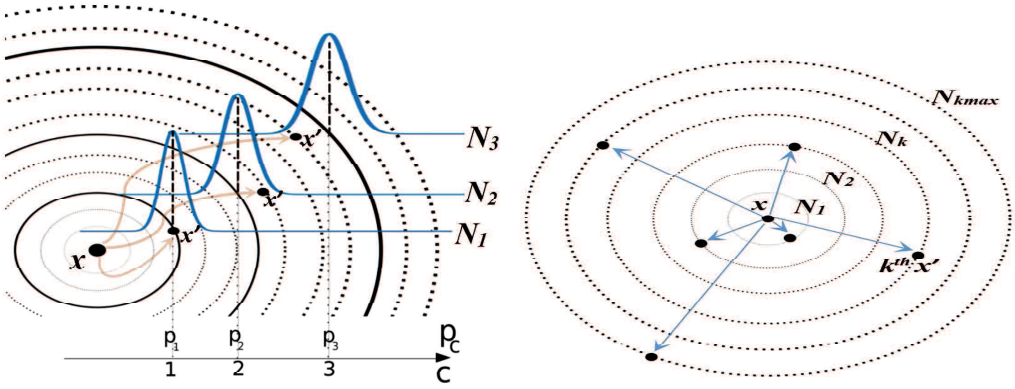
However, the B-VNS algorithm is more flexible than the VNS algorithm. The chunk size C can be set as equal to k_{max} to give an almost equal condition. However, the chunk size C can be set as either larger or less than k_{max}. Setting the chunk size C to less than k_{max} may have the potential to speed up the computation time.

In addition to the potential advantages of B-VNS, there are also potential weaknesses. Like the Jump VNS algorithm, in the B-VNS algorithm, thorough search behavior may be lost. The VNS algorithm, as described by Hansen and Mladenović [64, 65] and Festa et al. [50], performs a thorough search by starting with the nearby neighborhood structure and slowly expanding it when the local search fails to improve the solution. In contrast, in the proposed algorithm, a random pattern of the distance of the neighborhood structures is seen. This random characteristic can be advantageous as it can increase the exploration search. However, a drawback is that it causes B-VNS to miss basins that the global optimum may be in. Consequently, this random characteristic has a possibility of incurring a longer searching time than the basic VNS algorithm.

3.4 Benchmarking

Considering that the B-VNS algorithm is a modification of the basic VNS algorithm, it is fair and appropriate to investigate the performance of our proposed B-VNS algorithm alongside the VNS algorithm [50, 63] alone. The investigation did not involve any other algorithms. Therefore, the impact of our modification can be evaluated by using VNS algorithm performances as the bases.

The investigation was conducted by running simulations on some standard QUBO problems. The QUBO problems were taken from OR-Library [69] available at [70]. The best-



(a) B-VNS. (b) Basic VNS [66].

Figure 3.3: Neighborhood structure.

known objective functions for those problems were compiled from [43, 45, 44, 71, 72]. We also tested the B-VNS algorithm to solve some standard max-cut problems. We used problems from Helmberg–Rendl [73] that can be downloaded from [74]. Those problems were generated using a machine-independent graph generator called rudy, which Giovanni Rinaldi developed. We also tested the B-VNS algorithm on problems proposed by Burer, Monteiro, and Zhang [75], as they have different problem constructions. Helmberg–Rendl problems consist of random, planar, and toroidal graphs, while those from Burer et al. are cubic lattice graphs that represent Ising spin glass models [68]. Problems by Burer et al. can be downloaded from [76]. The best-known max-cut values were summarized from [68, 77, 78, 79, 80]. It is worth noting that all of those best-known values for QUBO and max-cut problems are open for improvement and may change in the future.

The simulation program used practically the same program code written in Fortran for both the VNS and B-VNS algorithms. The only difference was in the neighborhood construction section, and the remaining sections were the same. Therefore, we could accurately measure the performance difference between our proposed neighborhood structure and the standard VNS algorithm structure.

The simulation was compiled and run on a CentOS 8 system powered by an Intel Core i7-8700 processor with 16 GB RAM. The simulation was independently run 30 times for each problem ($NSim = 30$). Therefore, the sample size for both the VNS and B-VNS algorithms on each test item was 30, which was sufficient for the conduction of statistical tests. The best objective value and computation time obtained for each iteration and simulation were recorded. We used three criteria to evaluate the performance. After obtaining the best-known values, we calculated the difference ($BestDif$) between the best-known and the best value obtained from 30 simulations. If $BestDif = 0$, then the algorithm obtained the best-known value. A value of $BestDif > 0$ means the algorithm failed to obtained the best-known value. On the other hand, $BestDif < 0$ means the algorithm exceeded the best-known value.

$$BestSim = max(\{f_{Sim_i} : i = 1, 2, ..., NSim\}), \tag{3.10}$$

$$BestDif = BestKnown - BestSim. \tag{3.11}$$

We calculated the average differences ($AvgDif$) between the best-known and obtained objective values involving all 30 simulations.

$$AvgDif = \frac{\sum_{i=1}^{NSim}(f_{Sim_i} - BestKnown)}{NSim}. \tag{3.12}$$

Lastly, we calculated the average computation time ($AvgT$). Regarding all of the criteria, the algorithm that gives the lowest results is the best one. However, as the direct comparison of samples by their average values may have led to a biased conclusion, we conducted statistical analysis to precisely compare the results using JASP [81].

3.4.1 Test on QUBO problems

The local search as described in [71] was used in tested algorithms. Figure 3.4 shows the pseudocode of the local search used for QUBO problems. We applied equal conditions for the VNS and B-VNS algorithms. We aimed to reduce the values of all of the parameters to reduce the computation time while obtaining high-quality solutions.

In the preliminary step, we used four problems for parameter tuning: two problems each from Glover and Beasley, with sizes of 100. We started from larger parameter values and gradually reduced them. We found that $k_{max} = 0.02n$ was adequate for the VNS algorithm to obtain good solutions within short computation times. Hence, we set $p_{max} = 0.002$ on the B-VNS algorithm to make it equal. The number of iterations was set at $0.2n$ for both the VNS and B-VNS algorithms. The variable n was the problem size that was equal to the length of the solution vector. For the B-VNS algorithm, parameter chunk size C was set at $0.02n$ so that it was equal to k_{max} in the VNS algorithm. All of these settings made an equal condition for the VNS and B-VNS algorithms. Table 3.1 shows the test results for Glover problems [44], while Table 3.2 shows test results for Beasley problems [71].

Tests regarding Glover problems show that the B-VNS algorithm was able to give the same good results as the VNS algorithm in terms of objective function values. Both algorithms obtained the best-known value in almost all of the problems and only failed in the *6b* problem. The statistical analysis shows that the differences between the B-VNS and VNS algorithms were insignificant in all of the Glover test cases, as seen from the p-value greater than 0.05. The Mann–Whitney test ($\alpha = 0.05$) was used because most samples did not satisfy the normality and homoscedasticity assumption. In terms of computation time for tests on Glover problems, statistical analyses show that the B-VNS algorithm was faster than the VNS algorithm, specifically in problems with sizes up to 200. In problems with sizes of 500, the B-VNS algorithm was comparable to the VNS algorithm.

The test results regarding the Beasley problems also show a similar trend to the tests on the Glover problems. However, the B-VNS algorithm could obtain all of the best-known values, while the VNS algorithm failed on problems $bqp50\_1$, $bqp100\_1$, and $bqp250\_8$ [1]. Like the test on the Glover problem, as the differences were insignificant, both B-VNS and VNS algorithms were shown to be good algorithms for use in solving Beasley problems.

Statistical analyses for computation time regarding Beasley problems with $n < 500$ show that the B-VNS algorithm was significantly faster than the VNS algorithm. The computation times of the two algorithms were only statistically the same in problems $bqp50\_1$, $bqp50\_5$, $bqp50\_6$, $bqp100\_8$, and $bqp250\_7$. For problems where $n > 500$, the computation speeds for the B-VNS and VNS algorithms were comparable.

All of the tests regarding QUBO problems under equal conditions show that the the B-VNS and VNS algorithms are good, as they reached most of the best-known values, with some exceptions for the VNS algorithm due to its failures. Moreover, the B-VNS algorithm ran substantially faster than the VNS algorithm, particularly on problems with sizes less than 500. Therefore, the B-VNS algorithm suit for problems with sizes less than 500 and is comparable to the VNS algorithm for larger problems.

[1]The notation $bqp\boldsymbol{n\_m}$ refers to the Beasley problem, which has size n and number m.

```
procedure local_search( X, fX, f )
1  improved=true
2  while (improved):
3     improved= false            //set back the flag to start
4     T= X                       //set temporary T equal to X
5     fT= fX
6     for i=1,n:
7           T_i = 1 - T_i        //flip the variable i
8           fT=f(T)              //evaluate new solution
9           if fT>fX:
10                X= T           //update solution
11                fX= fT
12                improved=true
13          else
14                T_i = 1 - T_i  //reset variable i
15 return X
end procedure local_search
```

Figure 3.4: Local search for QUBO [71].

3.4.2 Test on Max-Cut problems

The representation of the max-cut problem in the QUBO problem can be found in [42]. Instead of using the QUBO form, we used the common formula for max-cut. Given undirected graph $G = (V, E)$ with node set $V = \{v_1, v_2, ..., v_n\}$ and non-negative weight $w_{ij} = w_{ji}$ on edge $(i, j) \in E$, a partition of G into two disjoint node subsets S and S^c that maximized the cut value was found.

$$cut(S, S^c) = \sum_{\alpha \in S, \beta \notin S} w_{\alpha\beta}. \tag{3.13}$$

This definition corresponds to the following forms:

$$max\ cut(S, S^c) = \frac{1}{2} \sum_{i<j} w_{ij}(1 - x_i x_j), \tag{3.14}$$

$$s.t.\ \ x_i, x_j \in \{-1, 1\}, \tag{3.15}$$

or

$$max\ cut(S, S^c) = \sum_{i<j} w_{ij}(x_i - x_j)^2, \tag{3.16}$$

$$s.t.\ \ x_i, x_j \in \{0, 1\}. \tag{3.17}$$

The ± 1 encoding was used on this test. We applied the local search process reported elsewhere [63][48] on both the VNS and B-VNS algorithms. The local search process was carried out as follows: With X as the current solution that corresponds to partition (S, S^c), a new partition was defined $(S', S^{c\prime})$.

$$(S', S^{c\prime}) = \begin{cases} (S \setminus \{i\},\ S^{c\prime} \cup \{i\}) & \text{if node } i \in S \\ (S \cup \{i\},\ S^{c\prime} \setminus \{i\}) & \text{if node } i \in S^c \end{cases} \tag{3.18}$$

For each node $i \in V$, a function δ associated with solution X was defined as

$$\delta(i) = \sum_{j \in S} w_{ij} - \sum_{j \in S^c} w_{ij}. \tag{3.19}$$

In order to improve the objective value, a node i made a movement from a subset of V to another subset regarding these situations:

Table 3.1: Results for Glover [44] problems.

Problem number	n	Best Known	VNS BestDif	VNS AvgDif	VNS Time**	B-VNS BestDif	B-VNS AvgDif	B-VNS Time**	test (p-value)* Dif	test (p-value)* Time
1a	50	3414	0	1.667	0.003	0	1	0.003	0.459	***
2a	60	6063	0	0	0.012	0	0	0.011	-	0.006
3a	70	6037	0	8.9	0.017	0	11.467	0.016	0.773	***
4a	80	8598	0	0	0.035	0	0	0.030	-	0.009
5a	50	5737	0	0	0.004	0	3.867	0.003	-	0.041
6a	30	3980	0	0	***	0	0	***	-	***
7a	30	4541	0	0	***	0	0	***	-	***
8a	100	11109	0	1.467	0.128	0	0	0.121	-	***
1b	40	133	0	18.033	***	0	21	***	0.512	-
2b	50	121	0	0.733	***	0	2.667	***	0.096	***
3b	60	118	0	4.667	0.001	0	8.533	0.001	0.065	0.401
4b	70	129	0	13.867	0.004	0	18.733	0.003	0.112	0.005
5b	80	150	0	0	0.012	0	0	0.011	-	***
6b	90	146	13	35.933	0.021	13	37.833	0.016	0.277	***
7b	80	160	0	0	0.027	0	2.533	0.025	-	***
8b	90	145	0	6.633	0.062	0	5.433	0.053	0.307	***
9b	100	137	0	2	0.156	0	2.433	0.141	1	***
10b	125	154	0	0.233	0.467	0	0.233	0.414	1	***
1c	40	5058	0	0	0.001	0	0	0.001	-	0.507
2c	50	6213	0	0	0.003	0	0	0.003	-	0.107
3c	60	6665	0	0	0.015	0	0	0.013	-	0.003
4c	70	7398	0	0	0.020	0	0	0.017	-	***
5c	80	7362	0	0.867	0.035	0	0	0.028	-	***
6c	90	5824	0	27.467	0.048	0	21.167	0.047	0.186	0.043
7c	100	7225	0	0	0.134	0	0	0.123	-	***
1d	100	6333	0	16.9	0.135	0	13.733	0.129	0.583	0.006
2d	100	6579	0	31.967	0.152	0	19.633	0.145	0.214	0.022
3d	100	9261	0	14.567	0.157	0	16.067	0.144	0.658	***
4d	100	10727	0	5.367	0.166	0	9.067	0.151	0.056	***
5d	100	11626	0	11.633	0.179	0	14.7	0.166	0.471	0.001
6d	100	14207	0	5	0.171	0	1.667	0.155	0.313	***
7d	100	14476	0	8.9	0.194	0	7.733	0.173	0.763	***
8d	100	16352	0	0	0.176	0	0	0.162	-	***
9d	100	15656	0	1.13	0.180	0	0.3	0.164	0.305	***
10d	100	19102	0	0	0.184	0	0	0.170	-	***
1e	200	16464	0	12.833	4.689	0	11.767	4.237	0.576	***
2e	200	23395	0	8	5.635	0	7.067	5.232	0.579	0.001
3e	200	25243	0	0	6.172	0	0	5.731	-	***
4e	200	35594	0	0.533	5.071	0	0.533	4.697	1	***
5e	200	35154	0	20.33	5.995	0	31.233	5.924	0.127	0.264
1f	500	61194	0	2	578.194	0	1.2	559.644	0.679	0.004
2f	500	100161	0	0.1	545.884	0	0.2	521.028	0.570	***
3f	500	138035	0	38.967	521.415	0	37.9	521.502	0.594	0.971
4f	500	172771	0	33.6	440.721	0	18	450.550	0.354	0.050
5f	500	190507	0	2.833	499.021	0	3.3	511.853	0.513	0.04

*: Mann–Whitney test; the difference is significant if $p - value < \alpha$, ($\alpha = 0.05$).
**: average computation time (second).
***: <0.001.

1. if $i \in S \wedge \delta(i) > 0$, then $S = S \setminus \{i\},\ S^{c\prime} = S^{c\prime} \cup \{i\}$;
2. if $i \in S^{c\prime} \wedge \delta(i) < 0$, then $S^{c\prime} = S^{c\prime} \setminus \{i\},\ S = S \cup \{i\}$.

This local search examined all of the possible movements starting from the first node.

For Burer et al. problems, parameter k_{max} was set at $0.1n$, while p_{max} was set at 0.1. The

Table 3.2: Results for Beasley [71] problems.

n	Problem number	Best Known	VNS BestDif	VNS AvgDif	VNS Time**	B-VNS BestDif	B-VNS AvgDif	B-VNS Time**	test (p-value)* Dif	test (p-value)* Time
50	1	2098	68	127.3	0.003	0	93.867	0.003	0.062	0.305
	2	3702	0	15	0.003	0	22.967	0.003	0.497	0.006
	3	4626	0	11.367	0.003	0	19	0.003	0.248	0.025
	4	3544	0	21.533	0.003	0	19.733	0.003	0.863	0.006
	5	4012	0	10.667	0.003	0	2.933	0.003	0.170	0.677
	6	3693	0	1.933	0.003	0	2.9	0.003	0.654	0.190
	7	4520	0	4.6	0.003	0	4.867	0.003	0.288	0.031
	8	4216	0	18	0.003	0	7.333	0.003	0.117	-
	9	3780	0	19.367	0.005	0	20.267	0.003	0.887	***
	10	3507	0	27.733	0.005	0	32.867	0.003	0.602	***
100	1	7970	42	150.867	0.080	0	173.133	0.076	0.163	0.034
	2	11036	0	15.333	0.083	0	19.333	0.078	0.732	0.001
	3	12723	0	0	0.071	0	0	0.068	-	0.039
	4	10368	0	8.333	0.078	0	11.533	0.072	0.984	***
	5	9083	0	44.467	0.085	0	49.167	0.079	0.682	0.006
	6	10210	0	2.067	0.088	0	4.767	0.080	0.910	0.006
	7	10125	0	24.467	0.082	0	26.533	0.072	0.770	***
	8	11435	0	8.867	0.079	0	9	0.077	0.820	0.139
	9	11455	0	0.6	0.081	0	0.6	0.075	1	0.004
	10	12565	0	18.667	0.078	0	12.933	0.069	0.247	***
250	1	45607	0	8	11.873	0	10.133	10.890	0.458	***
	2	44810	0	59.267	12.123	0	45.033	11.429	0.147	0.005
	3	49037	0	0	8.996	0	0	8.662	-	0.045
	4	41274	0	20.6	10.539	0	33.133	9.743	0.067	***
	5	47961	0	15.933	9.672	0	10.933	8.808	0.611	***
	6	41014	0	8.6	11.766	0	11.5	10.973	0.876	***
	7	46757	0	0	10.624	0	0	10.191	-	0.050
	8	35726	52	214.200	13.311	0	177	12.196	0.297	***
	9	48916	0	23.100	11.330	0	27.233	10.341	0.433	***
	10	40442	0	3.533	12.526	0	2.2	11.211	0.688	***
500	1	116586	0	5.333	586.406	0	6.267	592.798	0.677	0.398
	2	128339	0	2.5	459.926	0	4.4	455.013	0.402	0.374
	3	130812	0	0	501.773	0	0	496.806	-	0.432
	4	130097	0	28.933	518.133	0	26.733	523.351	0.486	0.321
	5	125487	0	10.4	521.381	0	2.6	516.252	0.380	0.300

*: Mann–Whitney test, the difference is significant if $p - value < \alpha$, ($\alpha = 0.05$).
**: average computation time (second).
***: <0.001.

number of iterations was set at $0.5n$ for both the VNS and B-VNS algorithms. We designed different test conditions for Helmberg–Rendl problems. Parameter k_{max} in the VNS algorithm was set at 100 for all of the Helmberg–Rendl problems used in the test regardless of problem size, as suggested in [68]. Therefore, the parameter p_{max} in the B-VNS algorithm was set at $100/n$, which enabled the B-VNS algorithm to reach an equal maximum neighborhood structure as in the VNS algorithm. We found that setting the iterations to $0.2n$ was adequate to obtain good solutions while reducing computation times.

Unlike the tests for QUBO problems, we applied a non-equal condition by setting the chunk size $C = 0.9k_{max}$ for Helmberg–Rendl as well as for Burer et al. problems. This non-equal condition was used to investigate the impact of setting the chunk size to less than k_{max}. This setting has a risk that the B-VNS algorithm will be much less thorough than the VNS algorithm, but it was used with the aim to be faster. Table 3.3 shows the test results for Helmberg–Rendl problems, while Table 3.4 shows results for those from Burer et al. The Mann–Whitney test with

$\alpha = 0.05$ was conducted, as most samples did not satisfy the normality and homoscedasticity assumption.

Although the B-VNS algorithm is much less thorough, the results show that the B-VNS algorithm was still able to provide solutions as satisfactorily as the VNS algorithm. Setting the chunk size C to smaller than k_{max} had an insignificant effect on the ability of the B-VNS algorithm to achieve the objective values. Moreover, the B-VNS algorithm was shown to be substantially faster than the VNS algorithm in all of the tested problems, regardless of the size. Therefore, the B-VNS algorithm was shown to clearly perform better than the VNS algorithm on max-cut problems.

Table 3.3: Results for Helmberg–Rendl [73] problems.

Graph	Problem	n	Best	VNS			B-VNS			test (p-value)*	
			Known	BestDif	AvgDif	Time**	BestDif	AvgDif	Time**	Dif	Time
Random	G1	800	11624	0	0.033	180.13	0	2.533	158.702	***	***
	G2	800	11620	0	7.133	185.804	0	6.8	162.854	0.830	***
	G3	800	11622	0	1.733	195.169	0	2.833	172.003	0.222	***
	G4	800	11646	0	0.567	193.595	0	0.633	175.664	0.507	***
	G5	800	11631	0	5.133	191.32	0	4.433	169.104	0.654	***
Random (±1)	G6	800	2178	0	1.867	203.065	0	2.2	136.156	0.299	***
	G7	800	2006	0	4.533	195.770	0	5.2	169.933	0.286	***
	G8	800	2005	0	3.4	154.650	0	2.733	164.133	0.845	***
	G9	800	2054	0	4.3	195.341	1	4.6	169.740	0.622	***
	G10	800	2000	0	3.2	160.173	0	2.333	141.589	0.191	***
Toroidal	G11	800	564	14	25.267	66.274	20	27.733	44.726	0.001	***
	G12	800	556	18	24.4	66.716	16	24.133	57.920	0.916	***
	G13	800	582	16	22.8	68.079	18	24.067	62.023	0.127	***
Planar	G14	800	3064	29	37.733	78.656	32	39.6	69.983	0.087	***
	G15	800	3050	31	38.633	77.195	32	39.867	67.350	0.179	***
Random	G43	1000	6660	1	8.967	431.314	1	9.733	381.073	0.323	***
	G44	1000	6650	3	8.467	428.178	2	9.533	375.822	0.114	***
	G45	1000	6654	0	12.067	425.304	1	11.033	374.124	0.445	***
	G46	1000	6654	9	16.1	410.047	5	16.633	373.864	0.494	***
	G47	1000	6654	9	20.433	415.607	13	20.267	370.332	0.472	***
Planar	G51	1000	3846	39	47.433	194.891	39	49.533	192.633	0.070	0.007
	G52	1000	3849	41	49.1	126.208	42	50.033	110.035	0.265	***

*: Mann–Whitney test; the difference is significant if $p - value < \alpha$, ($\alpha = 0.05$).
**: average computation time (second).
***: <0.001.

3.4.3 Discussion

We tested the basic VNS and B-VNS algorithms using simulations on several QUBO and max-cut problems. QUBO problems from Glover and Beasley and max-cut problems from Helmberg–Rendl and Burer et al. were chosen for the test, because they are benchmarking standards. Thus, our simulations gave a good overview of the performance of the two algorithms.

When solving the QUBO problems, the parameters of the two algorithms were set to be equivalent. Compared to the problem size n, the parameters of both algorithms were set to very small values. We used $k_{max} = 0.02n$ and $p_{max} = 0.02$. As a result, the k_{max} in the VNS algorithm and the maximum μ_D in the B-VNS algorithm only ranged from 1 to 10, and n ranged from 30 to 500. As the number of iterations was set at $0.2n$, the number of iterations was in the range of 6 to 100. However, both the VNS and B-VNS algorithms were able to obtain the best-known values. The failure on just a few problems that was observed is understandable because

Table 3.4: Results for Burer et al. [75] problems.

Problem	n	Best	VNS			B-VNS			test (p-value)*	
		Known	BestDif	AvgDif	Time**	BestDif	AvgDif	Time**	Dif	Time
sg3dl052000	125	112	0	1.2	0.042	0	1.8	0.039	0.058	***
sg3dl054000	125	114	0	1.333	0.043	0	2.133	0.039	0.158	***
sg3dl056000	125	110	0	1.333	0.041	0	1.467	0.038	0.803	***
sg3dl058000	125	108	0	1.333	0.043	0	1.6	0.039	0.312	***
sg3dl0510000	125	112	0	3.267	0.042	0	2.4	0.039	0.030	***
sg3dl102000	1000	900	20	28.867	402.421	18	30.200	350.943	0.237	***
sg3dl104000	1000	896	18	28.667	431.393	20	28.733	351.443	0.928	***
sg3dl106000	1000	886	24	31.267	359.999	28	34.467	322.743	0.004	***
sg3dl108000	1000	880	16	26.867	380.613	20	28.600	311.638	0.058	***
sg3dl1010000	1000	890	18	28.800	390.797	20	29.733	354.691	0.397	***

*: Mann–Whitney test, the difference is significant if $p - value < \alpha$, ($\alpha = 0.05$).
**: average computation time (second).
***: <0.001.

of the small values used for the parameters. The result would be improved by enlarging the parameter values, i.e., by increasing k_{max} in the VNS algorithm, p_{max} in the B-VNS algorithm, and the number of iterations. However, increasing the parameter values may result in a longer computation time.

For QUBO problems that apply equal conditions, the B-VNS algorithm was shown to be substantially faster than the VNS algorithm for problems with sizes of less than 500. Moreover, the B-VNS algorithm was able to provide good solutions to all of the tested problems. Solving the standard QUBO problem under equal conditions showed that the B-VNS algorithm is able to match the VNS algorithm even better and faster. This trend also occurred in the tests for max-cut problems, even though the parameter settings were under non-equal conditions that risked the B-VNS algorithm being less thorough. However, the experiments show that reducing the thorough search behavior by setting the chunk size $C = 0.9k_{max}$ in the B-VNS algorithm did not lessen the solution quality, and it even ran substantially faster in all of the max-cut problems tested.

Setting the parameter chunk size C to less than k_{max} has the risk of lowering the accuracy of the B-VNS algorithm. The characteristic of binomial distribution, which results in a pattern of randomly increasing distances, also may reduce accuracy. Small chunk size and binomial distribution can cause the algorithm to jump too far and miss some basins. However, our experiments and statistical analysis show that the B-VNS algorithm was still able to provide solutions as satisfactorily as the VNS algorithm. Therefore, setting the chunk size parameter to slightly less than k_{max} is advantageous for solution quality and efficiency.

The flexible design implemented by parameter chunk size C gives the B-VNS algorithm the potential to be faster than the VNS algorithm. Even though it is known that there is a risk of the B-VNS algorithm losing its thorough search characteristic, which may result in longer computation times, our experiments showed that this was not the case. The experiment results indicate that setting the B-VNS algorithm's parameters as equal to the VNS algorithm's parameters can avoid this risk.

All of the tests involving standard QUBO and standard max-cut problems show that the B-VNS and VNS algorithms are good. The VNS and B-VNS algorithms achieved most of the best-known values or were very close to them. It should be noted that the referenced best-known values were obtained using specific modified algorithms or by setting larger parameter values. The tests show that the B-VNS algorithm performed better in all of the max-cut problems regardless of problem size, and it was shown to be suited to QUBO problems with sizes less than

500.

We tested the B-VNS and VNS algorithm to investigate their performances. However, there are limitations to this study. First, we used the most basic form of VNS algorithm. We did not use the advanced or hybrid form because we focused on the construction mechanism of the neighborhood structure. Many studies regarding the VNS algorithm have been carried out previously, but reports on the use of alternative neighborhood structures to solve QUBO problems are very rare. Therefore, our study is useful for fundamentally developing the VNS algorithm. Even though we used small parameter settings, this was sufficient to observe the potential of both the B-VNS and VNS algorithms. Festa reported that the VNS algorithm is sensitive to problem characteristics, so early iterations can be used to predict the potential of the algorithm [68]. The results in this paper were also inseparable from the simulation we used. One of the crucial factors in metaheuristic simulation is the random number generator. We used the Fortran language and applied a dynamic random seed. Random seeds change over time. Each time a random number function is called, it will use a different seed to avoid generating specific patterns of random numbers. It is worth considering that the programming approach applied in simulation codes may also impact the results. Our tests only involved small- to medium-sized problems. However, our experiments show that using binomial distribution can potentially enhance the VNS algorithm. Thus, experimenting on much larger cases becomes a challenge. Considering that hybrid models tend to be more powerful, the hybrid form of the B-VNS algorithm is worth studying. Considering the successful implementation of parallel programming of the VNS algorithm [82], the potential of the B-VNS algorithm can be further developed by applying a suitable hybrid model.

3.5 Conclusion

The B-VNS algorithm is a VNS algorithm that is modified by applying binomial distribution to construct the neighborhood. As a result, the expansion of the neighborhood structure is no longer strictly monotonous but random, following the characteristics of binomial distribution. Our experiments used QUBO problems from Glover [44] and Beasley [71] and max-cut problems from Helmberg–Rendl [73] and Burer et al. [75]. We confirmed that the B-VNS and VNS algorithms are suitable for use in solving QUBO and max-cut problems. The experiment results show that both algorithms can provide good solutions, but the B-VNS algorithm runs faster. Furthermore, the B-VNS algorithm performed better in all of the max-cut problems regardless of problem size, and it performed better in QUBO problems with sizes less than 500. Although we did not test large-sized problems, our results suggest that the use of binomial distribution to construct neighborhood structures can improve the performance by reducing the speed.

Chapter 4

A Conceptual Design: Combination of Spy Algorithm and B-VNS

The spy algorithm ensures the benefit of exploration and exploitation as well as cooperative and non-cooperative searches in each iteration, as described in Chapter 2. Although the spy algorithm's original design has the nature of continuous optimization, the spy algorithm design can be expanded for solving discrete optimization models. Regarding the nature of the spy algorithm, combining it with B-VNS, as described in Chapter 3, will improve the performance.

For combining the spy algorithm and B-VNS, we should consider their processes. The spy algorithm is population-based, while B-VNS is a single-trajectory-based metaheuristic. Hence, using the spy algorithm as the main framework is more appropriate. As a result, the combination is a population-based metaheuristic. The high-level concept is shown in Fig. 4.1.

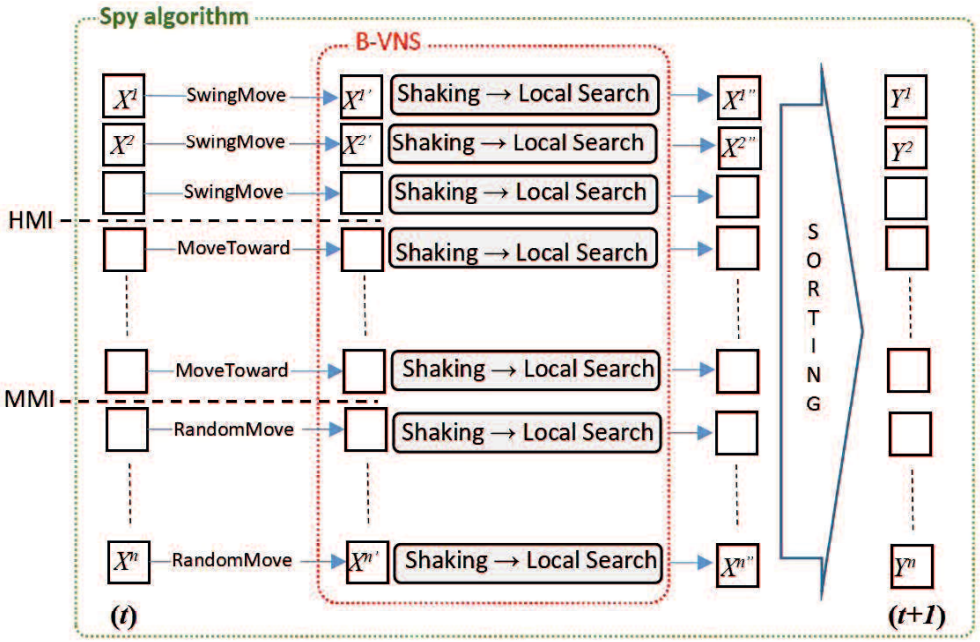


Figure 4.1: Conceptual design

The spy algorithm has a role in providing initial points for B-VNS. Based on this design, some part is exactly the same as B-VNS, where the low-rank solutions provide random points as initial points for B-VNS. In the other part, B-VNS will start from pretty good points generated

by the SwingMove and MoveToward operators. The spy algorithm gets an advantage because the B-VNS act to refine solutions created by the spy algorithm. On the other hand, this design enables the spy algorithm to feed various points for B-VNS. The spy algorithm is the primary framework while providing initial points for B-VNS. B-VNS acts as an additional refinement for the spy algorithm.

Chapter 5

Concluding Remarks

We have introduced two metaheuristic algorithms; the spy algorithm and B-VNS. The spy algorithm is a population-based metaheuristic algorithm that ensures the benefit of exploration and exploitation as well as cooperative and non-cooperative searches in each iteration. The spy algorithm can maintain exploitation and exploration search by applying proper algorithm settings to improve the solutions. The Spy algorithm is first designed to solve the continuous optimization models.

The B-VNS algorithm is a VNS algorithm that is modified by applying binomial distribution to construct the neighborhood. As a result, the expansion of the neighborhood structure is no longer strictly monotonous but random, following the characteristics of the binomial distribution. Based on the encoding, the B-VNS is intended to solve combinatorial optimization problems, particularly in the form of the QUBO problem.

Even though the spy algorithm and B-VNS have different designs in detail, they follow the same global framework of metaheuristics. They consist of the same five steps. Table 5.1 shows that the spy algorithm and B-VNS follow common steps. However, the original design of each of them targeted a different model. Another common thing is that the main idea of each of them applies to the refinement step, in line with the explanation in Chapter 1 that the refinement step plays a significant role in metaheuristics.

Table 5.1: Common metaheuristic steps apply to spy algorithm and B-VNS

	Spy	B-VNS
Class	population	single-trajectory
Step		
1. Initialization	random points	random point
2. Refinement	*SwingMove*, *MoveToward*, and random move	shaking and local search
3. Update	take the better one	take the better one
4. Termination	number of iteration	number of iteration
5. Finalization	propose the best one taken from the last solutions	propose the last solution
Domain	continuous model	discrete model (QUBO)
Main idea	apply fixed portion of *SwingMove*, *MoveToward*, and random move to control the exploration and exploitation searches	apply Binomial distribution to construct neighborhood structure

We tested the GA, IHS, PSO, and spy algorithm on non-convex functions by aiming at accuracy, the ability to detect many global optimum points, and computation time. Statistical analyses were conducted to draw conclusions regarding the accuracy and computational cost. As a result, the spy algorithm outperformed GA, IHS, and PSO by providing more accurate

solutions. Furthermore, the Spy algorithm provided higher maximum peak ratios. All those results were achieved by the Spy algorithm within less computation time. The spy algorithm is the most successful algorithm for solving two types of problems. This performance was obtained without changing the algorithm parameters. These results indicate that the design of the spy algorithm enables it to reach a balance state between exploration and exploitation searches. As a result, empirical experiments proved the spy algorithm more robust than other tested algorithms.

We tested the B-VNS and VNS on QUBO problems from Glover [44] and Beasley [71], and max-cut problems from Helmberg–Rendl [73] and Burer et al. [75]. We confirmed that the B-VNS and VNS algorithms are suitable for solving QUBO and max-cut problems. The experiment results show that both algorithms can provide good solutions, but the B-VNS algorithm runs faster. Furthermore, the B-VNS algorithm performed better in all of the max-cut problems regardless of problem size, and it performed better in QUBO problems with sizes less than 500. Although we did not test large-sized problems, our results suggest that the use of binomial distribution to construct neighborhood structures can improve performance by reducing speed.

The spy algorithms and B-VNS have different designs in the process and the domain of the solved problems. However, their combination has the potential to provide good results by considering their benefits. The spy algorithm can be seen as the first step of B-VNS. Conversely, B-VNS can be considered an additional refinement for the spy algorithm.

Bibliography

[1] El-Ghazali Talbi. *Metaheuristics From Design to Implementation*. John Wiley & Sons, New Jersey, USA, 2009.

[2] Xin-She Yang. *Nature-Inspired Metaheuristic Algorithms*. Luniver Press, 2nd edition, 2010.

[3] C Koulamas, SR Antony, and R Jaen. A survey of simulated annealing applications to operations research problems. *Omega*, 22(1):41 – 56, January 1994.

[4] Thomas A. Feo and Mauricio G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109 – 134, March 1995.

[5] Mauricio G.C. Resende and Celso C. Ribeiro. Greedy randomized adaptive search procedures. In Fred Glover and Gary A. Kochenberger, editors, *Handbook of Metaheuristics*, International Series in Operations Research & Management Science, chapter 8, pages 219–249. Kluwer Academic, Dordrecht, 2003.

[6] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE Computational Intelligence Magazine*, 1(4):28–39, November 2006.

[7] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[8] Rassoul Khosravanian, Vahid Mansouri, David A. Wood, and Masood Reza Alipour. A comparative study of several metaheuristic algorithms for optimizing complex 3-d well-path designs. *Journal of Petroleum Exploration and Production Technology*, 8:1487 – 1503, December 2018.

[9] Fred Glover. Tabu search - Part I. *ORSA Journal on Computing*, 1(3):190–206, August 1989.

[10] Fred Glover. Tabu search - Part II. *ORSA Journal on Computing*, 2(1):4–32, February 1990.

[11] Bernardo Morales-Castañeda, Daniel Zaldívar, Erik Cuevas, Fernando Fausto, and Alma Rodríguez. A better balance in metaheuristic algorithms: Does it exist? *Swarm and Evolutionary Computation*, 54:100671, 2020.

[12] Kashif Hussain, Mohd Najib Mohd Saleh, Shi Cheng, and Yuhui Shi. On the exploration and exploitation in popular swarm-based metaheuristic algorithms. *Neural Computing and Applications*, 31:7665–7683, 2019.

[13] Francisco Barahona, Martin Grötschel, Michael Jünger, and Gerhard Reinel. An application of combinatorial optimization to statistical physics and circuit layout design. *Operations Research*, 36(3):493–513, June 1988.

[14] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, New York, 1979.

[15] Richard M. Karp. *Reducibility Among Combinatorial Problems*, pages 219–241. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[16] J. C. Anglés d'Auriac, M. Preissmann, and A. Sebö. Optimal cuts in graphs and statistical mechanics. *Mathematical and Computer Modelling*, 26(8-10):1 – 11, 1997.

[17] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997.

[18] Jasbir S. Arora. Chapter 14 - practical applications of optimization. In Jasbir S. Arora, editor, *Introduction to Optimum Design (Third Edition)*, pages 575 – 617. Academic Press, Boston, 2012.

[19] P. B. Thanedar, J. S. Arora, G. Y. Li, and T. C. Lin. Robustness, generality and efficiency of optimization algorithms for practical applications. *Structural optimization*, 2(4):203 – 212, December 1990.

[20] Michał Komorowski, Maria J. Costa, David A. Rand, and Michael P. H. Stumpf. Sensitivity, robustness, and identifiability in stochastic chemical kinetics models. *Proceedings of the National Academy of Sciences*, 108(21):8645–8650, May 2011.

[21] Nikos D. Lagaros and Dimos C. Charmpis. Efficiency and robustness of three metaheuristics in the framework of structural optimization. In Harris Papadopoulos, Andreas S. Andreou, and Max Bramer, editors, *Artificial Intelligence Applications and Innovations*, pages 104–111, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[22] Katta G. Murty and Santosh N. Kabadi. Some NP-complete problems in quadratic and nonlinear programming. *Mathematical Programming*, 39:117 – 129, June 1987.

[23] Paul C. Jennings, Steen Lysgaard, Jens Strabo Hummelshøj, Tejs Vegge, and Thomas Bligaard. Genetic algorithms for computational materials discovery accelerated by machine learning. *npj Computational Materials*, 5, April 2019.

[24] Sangit Chatterjee, Matthew Laudato, and Lucy A. Lynch. Genetic algorithms and their statistical applications: an introduction. *Computational Statistics & Data Analysis*, 22(6):633 – 651, October 1996.

[25] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, Cambridge, MA, USA, 1996.

[26] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948, 1995.

[27] Shahid Shabir and Ruchi Singla. A comparative study of genetic algorithm and the particle swarm optimization. *International Journal of Electrical Engineering*, 9(2):215 – 223, 2016.

[28] Navid Reza Tayebi, Fereidoon Moghadas Nejad, and Mahmood Mola. Comparison between GA and PSO in analyzing pavement management activities. *Journal of Transportation Engineering*, 140(1):99–104, 2014.

[29] Zong Woo Geem, Joong Hoon Kim, and G.V. Loganathan. A new heuristic optimization algorithm: Harmony search. *SIMULATION*, 76(2):60–68, February 2001.

[30] M. Mahdavi, M. Fesanghary, and E. Damangir. An improved harmony search algorithm for solving optimization problems. *Applied Mathematics and Computation*, 188(2):1567 – 1579, May 2007.

[31] Mahamed G.H. Omran and Mehrdad Mahdavi. Global-best harmony search. *Applied Mathematics and Computation*, 198(2):643–656, 2008.

[32] James Edward Baker. Adaptive selection methods for genetic algorithms. In *Proceedings of an International Conference on Genetic Algorithms and their applications*, volume 101, page 111. Hillsdale, New Jersey, 1985.

[33] F. Herrera, M. Lozano, and J.L. Verdegay. Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis. *Artificial Intelligence Review*, 12(4):265–319, August 1998.

[34] Bruno Seixas Gomes de Almeida and Victor Coppo Leite. Particle swarm optimization: A powerful technique for solving engineering problems. In Javier Del Ser, Esther Villar, and Eneko Osaba, editors, *Swarm Intelligence*, chapter 3. IntechOpen, Rijeka, 2019.

[35] Vangelis Th. Paschos. *Applications of Combinatorial Optimizations*. John Wiley & Sons, Ltd, 2014.

[36] El-Ghazali TalbiTalbi. *Metaheuristics*. John Wiley & Sons, Ltd, 2009.

[37] Xin-She Yang. Metaheuristic optimization: Algorithm analysis and open problems. In Panos M. Pardalos and Steffen Rebennack, editors, *Experimental Algorithms*, pages 21–32, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[38] Kenneth Sörensen and Fred W. Glover. *Metaheuristics*, pages 960–970. Springer US, Boston, MA, 2013.

[39] Fred Glover and Gary A. Kochenberger, editors. *Handbook of Metaheuristics*. Springer, 2003.

[40] Christos Papalitsas, Theodore Andronikos, Konstantinos Giannakis, Georgia Theocharopoulou, and Sofia Fanarioti. A QUBO model for the traveling salesman problem with time windows. *Algorithms*, 12(11), 2019.

[41] Prasanna Date, Davis Arthur, and Lauren Pusey-Nazzaro. QUBO formulations for training machine learning models. *Scientific reports*, 11(1):10029–10029, May 2021.

[42] Fred Glover, Gary Kochenberger, and Yu Du. A tutorial on formulating and using QUBO models. *CoRR*, abs/1811.11538, 2018.

[43] Kengo Katayama and Hiroyuki Narihisa. Performance of simulated annealing-based heuristic for the unconstrained binary quadratic programming problem. *European Journal of Operational Research*, 134(1):103–119, 2001.

[44] Fred Glover, Gary A. Kochenberger, and Bahram Alidaee. Adaptive memory tabu search for binary quadratic programs. *Management Science*, 44(3):336–345, 1998.

[45] Peter Merz and Bernd Freisleben. Genetic algorithms for binary quadratic programming. GECCO'99, page 417-424, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[46] Endre Boros, Peter L. Hammer, and Gabriel Tavares. Local search heuristics for quadratic unconstrained binary optimization (QUBO). *Journal of Heuristics*, 13(2):99–132, Apr 2007.

[47] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100, 1997.

[48] Abraham Duarte, Ángel Sánchez, Felipe Fernández, and Raúl Cabido. A low-level hybridization between memetic algorithm and VNS for the max-cut problem. In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*, GECCO '05, page 999-1006, New York, NY, USA, 2005. Association for Computing Machinery.

[49] Su-Hyang Kim, Yong-Hyuk Kim, and Byung-Ro Moon. A hybrid genetic algorithm for the max cut problem. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, GECCO'01, page 416-423, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[50] Paola Festa, P Pardalos, M Resende, and C Ribeiro. GRASP and VNS for max-cut. In *Extended Abstracts of the Fourth Metaheuristics International Conference*, pages 371–376, 2001.

[51] MG Resende. GRASP with path re-linking and VNS for maxcut. In *Proceedings of 4th MIC, Porto*, 2001.

[52] Makbul A. M. Ramli and Houssem R. E. H. Bouchekara. Solving the problem of large-scale optimal scheduling of distributed energy resources in smart grids using an improved variable neighborhood search. *IEEE Access*, 8:77321–77335, 2020.

[53] Fucai Wang, Guanlong Deng, Tianhua Jiang, and Shuning Zhang. Multi-objective parallel variable neighborhood search for energy consumption scheduling in blocking flow shops. *IEEE Access*, 6:68686–68700, 2018.

[54] L. Garcia-Hernandez, L. Salas-Morera, C. Carmona-Muñoz, A. Abraham, and S. Salcedo-Sanz. A hybrid coral reefs optimization—variable neighborhood search approach for the unequal area facility layout problem. *IEEE Access*, 8:134042–134050, 2020.

[55] Meiling He, Zhixiu Wei, Xiaohui Wu, and Yongtao Peng. An adaptive variable neighborhood search ant colony algorithm for vehicle routing problem with soft time windows. *IEEE Access*, 9:21258–21266, 2021.

[56] Abdessamad Ait El Cadi, Rabie Ben Atitallah, Nenad Mladenović, and Abdelhakim Artiba. A variable neighborhood search (VNS) metaheuristic for multiprocessor scheduling problem with communication delays. In *2015 International Conference on Industrial Engineering and Systems Management (IESM)*, pages 1091–1095, 2015.

[57] Guilherme Silva, Pedro Silva, Valéria Santos, Alan Segundo, Eduardo Luz, and Gladston Moreira. A VNS algorithm for PID controller: Hardware-in-the-loop approach. *IEEE Latin America Transactions*, 19(9):1502–1510, 2021.

[58] R. K. Phanden, H. I. Demir, and R. D. Gupta. Application of genetic algorithm and variable neighborhood search to solve the facility layout planning problem in job shop production system. In *2018 7th International Conference on Industrial Technology and Management (ICITM)*, pages 270–274, 2018.

[59] Dharmesh Dabhi and Kartik Pandya. Uncertain scenario based microgrid optimization via hybrid levy particle swarm variable neighborhood search optimization (HL_PS_VNSO). *IEEE Access*, 8:108782–108797, 2020.

[60] Sujun Zhang, Xingsheng Gu, and Funa Zhou. An improved discrete migrating birds optimization algorithm for the no-wait flow shop scheduling problem. *IEEE Access*, 8:99380–99392, 2020.

[61] Chaoyong Zhang, Zhanpeng Xie, Xinyu Shao, and Guangtong Tian. An effective VNSSA algorithm for the blocking flowshop scheduling problem with makespan minimization. In *2015 International Conference on Advanced Mechatronic Systems (ICAMechS)*, pages 86–89, 2015.

[62] Antonio S Montemayor, Abraham Duarte, Juan José Pantrigo, Raúl Cabido, and J Carlos. High-performance VNS for the max-cut problem using commodity graphics hardware. In *Mini-Euro Conference on VNS (MECVNS 05), Tenerife (Spain)*, pages 1–11, 2005.

[63] Ai-fan Ling, Cheng-xian Xu, and Le Tang. A modified VNS metaheuristic for max-bisection problems. *Journal of Computational and Applied Mathematics*, 220(1):413–421, 2008.

[64] Pierre Hansen and Nenad Mladenović. Variable neighborhood search. In Fred Glover and Gary A. Kochenberger, editors, *Handbook of Metaheuristics*, International Series in Operations Research & Management Science, chapter 6, pages 145–184. Kluwer Academic, Dordrecht, 2003.

[65] Pierre Hansen, Nenad Mladenović, Jack Brimberg, and José A. Moreno Pérez. Variable neighborhood search. In M. Gendreau and J.-Y. Potvin, editors, *Handbook of Metaheuristics*, International Series in Operations Research & Management Science, chapter 3, pages 61–184. Springer, 2010.

[66] Pierre Hansen, Nenad Mladenović, and José A. Moreno Pérez. Variable neighbourhood search: methods and applications. *4OR*, 6(4):319–360, Dec 2008.

[67] Pierre Hansen and Nenad Mladenović. A tutorial on variable neighborhood search. Technical report, LES CAHIERS DU GERAD, HEC MONTREAL AND GERAD, 2003.

[68] P. Festa, P.M. Pardalos, M.G.C. Resende, and C.C. Ribeiro. Randomized heuristics for the max-cut problem. *Optimization Methods and Software*, 17(6):1033–1058, 2002.

[69] J. E. Beasley. OR-Library: Distributing test problems by electronic mail. *The Journal of the Operational Research Society*, 41(11):1069–1072, 1990.

[70] J. E. Beasley. OR-Library, 2004. `http://people.brunel.ac.uk/~mastjjb/jeb/orlib/files`.

[71] John E. Beasley. Heuristic algorithms for the unconstrained binary quadratic programming problem. Technical report, The Management School, Imperial College, London, UK, Dec 1998.

[72] Angelika Wiegele. Biq Mac library -a collection of max-cut and quadratic 0-1 programming instances of medium size. Technical report, Alpen-Adria-Universität Klagenfurt, Institut für Mathematik, Universitätsstr, Klagenfurt, Austria, 2007.

[73] Christoph Helmberg and Franz Rendl. A spectral bundle method for semidefinite programming. *SIAM J. Optim.*, 10:673–696, 2000.

[74] Yinyu Ye. Gset, 2003. `https://web.stanford.edu/~yyye/yyye/Gset`.

[75] Samuel Burer, Renato D. C. Monteiro, and Yin Zhang. Rank-two relaxation heuristics for MAX-CUT and other binary quadratic programs. *SIAM Journal on Optimization*, 12(2):503–521, 2002.

[76] Martí, Duarte, and Laguna. Maxcut problem, 2009. `http://grafo.etsii.urjc.es/optsicom/maxcut/set2.zip`.

[77] Gary A. Kochenberger, Jin-Kao Hao, Zhipeng Lü, Haibo Wang, and Fred W. Glover. Solving large scale max cut problems via tabu search. *Journal of Heuristics*, 19:565–571, 2013.

[78] Yang Wang, Zhipeng Lü, Fred Glover, and Jin-Kao Hao. Probabilistic GRASP-tabu search algorithms for the UBQP problem. *Computers & Operations Research*, 40(12):3100–3107, 2013.

[79] Gintaras Palubeckis and Vita Krivickienė. Application of multistart tabu search to the max-cut problem. *Information Technology and Control*, 31, 2004.

[80] Endre Boros, Peter L. Hammer, Richard Sun, and Gabriel Tavares. A max-flow approach to improved lower bounds for quadratic unconstrained binary optimization (QUBO). *Discrete Optimization*, 5(2):501–529, 2008. In Memory of George B. Dantzig.

[81] JASP Team. JASP (Version 0.16)[Computer software], 2021.

[82] Panos Kalatzantonakis, Angelo Sifaleras, and Nikolaos Samaras. Cooperative versus non-cooperative parallel variable neighborhood search strategies: a case study on the capacitated vehicle routing problem. *Journal of Global Optimization*, 78:327–348, 10 2020.