Solving Graph Coloring Problems Using Discrete Artificial Bee Colony

September 2019

CHEN KUI

Solving Graph Coloring Problems Using Discrete Artificial Bee Colony

Graduate School of Systems and Information Engineering

University of Tsukuba

September 2019

CHEN KUI

Abstract

In this thesis, we proposed two discrete artificial bee colony algorithms to solve graph coloring problems.

Graph coloring problem (GCP) is a famous combinatorial optimization problem. It is defined as coloring an undirect graph such that no adjacent nodes share the same color. Because GCP can be used to model many real-world problems, it has been studied by many researchers over a hundred year. However, even though the definition is simple, GCP is a NP-complete problem and has some aspects that are surprisingly difficult to solve.

Traditionally, there are two main strategies to solve GCP: heuristic and meta-heuristic methods. The typical heuristic method is DSatur, which colors each node according to saturation degree. In meta-heuristic family, many ideas based on local search are proposed to improve candidate locally in order to find better solutions. Such methods include simulated annealing, Tabu search, and iterated local search. On the other hand, global search meta-heuristic is usually population-based and tries to find better solutions in the whole search space. Famous global search meta-heuristic methods include genetic algorithm, evolution strategy and swarm intelligence.

Swarm intelligence (SI) is a population-based global search meta-heuristic. It is inspired from the biological systems that deals with nature. Typically, an SI system consists of several agents and a series of simple moving rules. Controlled by these rules, agents interact with each other and with the environment. Artificial bee colony (ABC) is one of SI algorithms. It is constructed by three main phases: (1) employed bee phase, which explores the whole search space; (2) onlooker bee phase, which exploits locally; and (3) scout bee phase, which is used to avoid being trapped in a local-optimal solution. ABC is proved to be an efficient and robust algorithm and is now widely used to solve optimization problems in continuous domain.

In this research, we try to apply ABC to solve GCPs. The difficulty of this work is adapting original ABC from continuous domain to discrete domain. First, we proposed S-ABC. Using hamming distance, S-ABC measures the distance between two candidates by similarity and updates a candidate by its similar neighbor. Compared with HDPSO on Minton random graphs, S-ABC obtains higher success rate and lower evaluation times. However, S-ABC does not use the graph information such as graph size, constraints, and the current fitness during its search process. To improve the performance, we then proposed adaptive ABC (A-ABC), which updates a candidate according to the above graph information. Experiments on three classes of random graphs show that A-ABC is efficient and robust and outperforms its competitors S-ABC, HDPSO, and D-FA. We also studied the scout bee phase and report that the scout bee phase is not required in solving GCPs.

Contents

Chapter 1 Introduction

1.1 Background of Research

Recently, swarm intelligence (SI) is widely used to solve optimization problems. Swarm intelligence was first introduced by Gerardo Beni and Jing Wang in 1989 [Beni 93]. It is a variety of meta-heuristics and is inspired from the biological systems such as the behavior of birds, ants, or fish for foraging and defending. Typically, a swarm intelligence system consists of several agents and a series of moving rules. Controlled by these rules, the agents interact with each other and with the environment. Based on this discipline, many swarm intelligence algorithms, for example, particle swarm optimization (PSO) [Kennedy 95], artificial bee colony (ABC) [Karaboga 07], cuckoo search (CS) [Yang 09], and firefly algorithm (FA) [Yang 09], have been developed and successfully applied to solve optimization problems in continuous domain.

For example, PSO is applied to electronic engineering [Peksen 14, Yang 14], automatic control [Kolomvatsos 14, Nedic 14], and communication [Yousefi 12, Minasian 13]; ABC is used to train neural network [Shah 11, Yeh 12], discover rule and data [Hsieh 11, Karaboga 11], and process image [Ma 11, Cuevas 12]; CS shows superior performance when solving spring design and welded beam design problems [Gandomi 13, Yang 13]; and FA obtains good performance in areas of the engineering design [Azad 11] and antenna design [Basu 11].

Even though most swarm intelligence algorithms are designed to solve continuous optimization problems, some of them have been extended to the areas of discrete optimization problems such as time-tabling problems [Irene 09, Tassopoulos 12, Kanoh 13], travelling-salesman problems [Kanoh 12, Kanoh 14], and flow-shop problems [Pan 11, Sayadi 10].

In this work, we focus on solving graph coloring problems (GCPs). GCP was first introduced by Frances Guthrie in 1800s [Lewis 16] and is a famous combinatorial optimization problem in the field of graph theory. It is defined as coloring an undirected graph such that no adjacent nodes share the same color. GCP can be used to model many real-world problems such as team building exercise, constructing timetables, scheduling taxis, and compiler register allocation. Because of its simple definition and practicability, GCP has been studied carefully over one century. However, even though the definition is easy to state, the difficulty of GCP is affected by many factors such as the size of graph, the number of edges, and the topology of graph.

Heuristics and meta-heuristics are two main strategies to solve GCP. The simplest heuristic method may be greedy algorithm [Dunstan 75]. Greedy algorithm first sorts the nodes randomly and then takes nodes successively according to the order and tries to assign the first feasible

color to each node. Greedy algorithm cannot guarantee to find the optimum. However, it has been shown that greedy algorithm can produce an optimal solution if given a correct order of nodes. As an improvement of greedy algorithm, DSatur [Brelaz 79] has been proposed to find the best order of coloring nodes. DSatur is similar to greedy algorithm but uses different ways to decide the node to be colored. With greedy algorithm, the order is decided before coloring, on the other hand, DSatur heuristically select the node to be colored next according to the saturation degree (i.e. the number of different colors assigned to the adjacent nodes) of the nodes.

In the meta-heuristic's family, single solution and population-based searches are two main ideas. Single solution approaches modify and improve a single candidate solution. The typical single solution meta-heuristics used to solve GCPs include simulated annealing [Chams 87] and iterated local search [Caramia 08]. On the other hand, global search meta-heuristics are usually population-based. Such methods include evolutionary algorithm [Eiben 98, Galinier 99], genetic algorithm [Davis 91, Dorne 98, Fleurent 96], and swarm intelligence.

The difficulty of applying swarm intelligence algorithms to solve GCPs is how to convert the continuous search space to the discrete search space. In general, there are three methods to do so:

(1) Using sigmoid function. By using sigmoid function, Binary PSO (BPSO) [Kennedy 97] is first developed to solve binary optimization. In BPSO, sigmoid function maps a real number from 0.0 to 1.0 on binary number 1 or 0. Experiments on five benchmark problems show that BPSO is flexible and robust, but it is easy to be trapped in the local optimal. This idea is then used by Modified PSO (MPSO) [Cui 08], Modified Turbulent PSO (MTPSO) [Hsu 11] and Improved CS (ICS) [Zhou 13] to solve 4-colorable GCPs. In these two algorithms, sigmoid function separates the real space into four partitions that identify four colors. Experiments show that they obtain good results on small graphs.

(2) Hybridizing with local search. This is a nature idea to improve the performance. By hybridizing with DSatur, Hybrid ABC (HABC) [Fister 12] and Memetic FA (MFA) [Fister 12] obtain high performance on GCP. In these two methods, ABC and FA are used to tune weights in continuous domain and each weight represents priority of the corresponding node. Then, DSatur colors nodes one by one according to the weights: the larger the weight is, the earlier the node should be colored. Experiments on three kinds of random graphs show that they match the competitive results of the best graph coloring algorithms such as Tabucol [Hertz 87] and Hybrid Evolutionary Algorithm [Galinier 99].

(3) Introducing discrete distance. Even though (1) and (2) obtain good results, swarm intelligence algorithms still work in continuous domain but use other tools (such as sigmoid function or DSatur) to map the real values on the target discrete domain. By introducing discrete distance, swarm intelligence algorithms can work in discrete domain directly. For example, HDPSO [Aoki 15] uses hamming distance to calculate the difference between two candidates and outperforms TPPSO [Kanoh 13] and GA on three colorable random graphs. Hamming

distance is a good tool to compare two candidates but is not appropriate to describe the distance a single candidate should be moved. To solve this problem, discrete cuckoo search (DCS) [Aranha 2017] uses discrete levy flight distribution to determine the number of nodes whose color should be changed and obtains good performance on 3-colorable random graphs.

There are some problems in above proposed methods. Using sigmoid function is simple but the related works are not efficient. All experiments are performed on small graphs with the number of nodes is less than 100. When testing them on larger graphs, the performance turns to worse. On the other hand, hybridizing with local search such as DSatur obtains high performance but these algorithms are designed for solving GCPs only and hard to be applied to other problems because of lack of flexibility. Finally, HDPSO and DCS are tested on 3-colorable Minton random graph only so the generality cannot be shown.

1.2 Purpose of Research

There are very few studies on solving GCPs using ABC. So, the purposes of our research are designing simple and high performance discrete artificial bee colony algorithms without hybridizing with local search and testing them on various classes of random graphs to show the generality.

We first introduce similarity to discretize original ABC. Using HDPSO as a baseline, we show that SABC is an efficient method to solve 3-colorable GCP by performing experiments on Minton random graphs.

Next, we propose adaptive ABC (A-ABC) and show its performance. In A-ABC, graph information (such as the number of nodes and edges, the fitness of current candidate) is considered. According to the information, A-ABC can adjust the number of replaced nodes automatically during the evolution. To show the performance and generality of A-ABC, we compare it with other four algorithms on three kinds of random graphs with various topology. The results show that adaptive ABC is a fast, robust, and general algorithm to solving GCPs.

1.3 Structure of Thesis

There are five chapters in this thesis, and they are organized as follows. Chapter 1 is introduction and chapter 5 is conclusions. Chapter 2 gives abstract of research theme. We first introduce the original ABC, PSO, FA, and CS, then we define the graph coloring problems and finally, we describe some related works. In chapter 3, S-ABC and D-FA are proposed and tested on Minton random graphs by comparing with HDPSO. In chapter 4, A-ABC is proposed and compared with other five algorithms on three classes of random graphs to show its performance and generality. We also study scout bee phase can report that scout bee phase is not required in solving GCPs.

Chapter 2 Overview of Research Area

2.1 Swarm Intelligence Algorithms

2.1.1 Basic Framework of Swarm Intelligence

Swarm intelligence is a population-based system that consists of some agents and a series of rules that define how to move these agents. In the context of optimization problems, agents are usually *candidate solutions* and the rules try to fine better solutions in the search space by updating these candidates. Assuming we have an *objective function*

$$\mathrm{y} = f(\mathbf{x}) \tag{2.1}$$

where $\mathbf{x} \in \Omega$ $(\Omega \subset \mathcal{R}^n)$ and $\mathrm{y} \in \mathcal{R}$. Assuming we want to find the *maximum* of $f(\mathbf{x})$, the framework of swarm intelligence is given in Figure 1.

Swarm intelligence usually begins with *randomly* generating N candidates $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$ in set Ω and each candidate is a vector with n elements because Ω is the proper subset of $\mathcal{R}^n$. The N candidates compose a *swarm* and N is the *swarm size*. Then, users should define the strategy of updating candidate: $u(\mathbf{x})$. After that, an iterator is used to improve the swarm and tries to find better candidates. Each iteration is known as a *generation*. For one generation, the N candidates are updated one by one. When a candidate $\mathbf{x}_i$ is updated, it is first saved in a

```
Randomly initialize N candidates x_1, x_2, ..., x_N in set Ω
Define update rules: x^new = u(x)
while (termination condition is not met)
    for i = 1 to N
        v_i = x_i
        v_i^new = u(v_i)
        if (f(v_i^new) > f(x_i))
            replace x_i by v_i^new
        end if
    end for
end while
```

Fig. 2.1. Framework of swarm intelligence

temporary variable $\mathbf{v}_i$ and then, by invoking the update rule $u(\mathbf{v}_i)$, a new candidate $\mathbf{v}_i^{new}$ is generated. Next, because we want to find the maximum of $f(\mathbf{x})$, we *evaluate* $\mathbf{v}_i^{new}$ and check whether $f(\mathbf{v}_i^{new})$ is larger than $f(\mathbf{x}_i)$. If so, that means a better candidate has been found, we remove $\mathbf{x}_i$ from the swarm and replace it by $\mathbf{v}_i^{new}$, otherwise $\mathbf{x}_i$ is kept without change. The iterator stops when *termination condition* is met. Common termination conditions are: (1) an optimum is found; (2) the maximum generation is met; or (3) the maximum evaluation times of objective function is met.

Obviously, the updating rule exerts a strong influence on the performance of swarm intelligence. By defining various updating rules, researchers have developed many swarm intelligence algorithms.

2.1.2 Artificial Bee Colony

Artificial bee colony (ABC) is a swarm intelligence algorithm that was proposed by Karaboga in 2007 [Karaboga 07]. It is an efficient and robust algorithm and has been applied to solve many continuous optimization problems [Karaboga 14].

Using the same objective function (2.1) in 2.1.1, the updating rules of original ABC is described as follows. When a candidate $\mathbf{x}_i$ will be updated, it is first saved to a temporary variable $\mathbf{v}_i$ as usual. Then, another candidate $\mathbf{x}_j$ in the swarm is selected randomly and $\mathbf{v}_i$ is updated by equation (2.2):

$$v_{il} = v_{il} + \phi \times (v_{il} - x_{jl}) \tag{2.2}$$

where $i, j \in \{1, 2, \dots, N\} \wedge i \neq j$, $l \in \{1, 2, \dots, n\}$, and ϕ is a uniformly distributed random number in $[-1, 1]$. Note that v_{il} and x_{jl} are l^{th} elements of $\mathbf{v}_i$ and $\mathbf{x}_j$ respectively and l is selected randomly. Equation (2.2) states that a new candidate $\mathbf{v}_i^{new}$ is generated by adding a small turbulent to a random element of $\mathbf{v}_i$. For convenience, we organize this process as a function $abc\_upate(\mathbf{x}_i)$ in Figure 2.2.

Interestingly, original ABC separates updating scheme into three phases: employed bee phase,

```
abc_update(x_i)
    v_i = x_i
    randomly select a candidate x_j (i ≠ j) in the swarm
    randomly select an element l and φ
    update v_i using equation 2.2
    return v_i
```

Fig. 2.2 Updating rule of original ABC

onlooker bee phase, and scout bee phase. They are given in Figure 2.3.

From Figure 2.3, we can see that (1) in employed bee phase, ABC searches the neighborhood of candidates one by one and tries to improve the swarm. Note that each candidate is updated only once; (2) onlooker bee phase is similar to employed bee phase but choose candidates that will be updated by roulette selection. In the context of finding maximum of objective function, this means that the larger $f(\mathbf{x})$ is, the higher probability candidate $\mathbf{x}$ is selected. For convenience, we call the value of objective function (i.e. $f(\mathbf{x})$) the *fitness* of $\mathbf{x}$. Using roulette selection, the candidates with high fitness may be selected and be updated more than one time but the ones with low fitness may never be selected; (3) if a candidate cannot be improved further in a predetermined number of generations, which known as *limit*, scout bee phase replace it by a randomly generated new candidate.

In short summary, employed bee phase explores the whole search space while onlooker bee phase exploits the places with high fitness. To avoid being trapped in a local optimum, scout bee phase removes candidates that cannot be improve further in *limit* generations. Note that *limit* is the only parameter in original ABC, this makes original ABC simple to fine-tune.

```
Employed bee phase(swarm)
    for i = 1 to N
        v_i^new = abc_update(x_i)
        if (f(v_i^new) > f(x_i))
            replace x_i by v_i^new
        end if
    end for

Onlooker bee phase(swarm)
    for i = 1 to N
        using roulette selection to select a target candidate x
        v^new = abc_update(x)
        if (f(v^new) > f(x))
            replace x by v^new
        end if
    end for

Scout bee phase(swarm)
    given a predetermined cycle limit
    for i = 1 to N
        if x_i is not improved in limit generations
            replace x_i by a random generated new candidate v_i
        end if
    end for
```

Fig. 2.3 Employed bee phase, onlooker bee phase, and scout bee phase

```
Randomly initialize a swarm with N candidates
Give the value of hyper-parameter limit

while (termination condition is not met)
    employed bee phase (swarm)
    onlooker bee phase (swarm)
    scout bee phase (swarm)
end while
```

Fig. 2.4 Framework of original ABC

Combing the three phases together, the framework of original ABC is given in Figure 2.4.

2.1.3 Particle Swarm Optimization

Particle swarm optimization (PSO) is one of the most famous swarm intelligence algorithms. It is well studied and has been widely applied in many research fields [Zhang 15].

In original PSO, each candidate $\mathbf{x}_i$ is assigned a *velocity vector* $\mathbf{v}_i$, which has the identical dimension with $\mathbf{x}_i$. In the context of the objective function (2.1), $\mathbf{x}_i$ and $\mathbf{v}_i$ are both n-dimensional vector. For the t^{th} generation, a candidate $\mathbf{x}_i^t$ is updated according to three factors: (1) its velocity $\mathbf{v}_i^t$, (2) the best previous candidate (i.e. the previous $\mathbf{x}_i$ that obtains the highest fitness) $\boldsymbol{pbest}_i$, and (3) the best previous candidate in the swarm $\boldsymbol{gbest}$. The next candidate $\mathbf{x}_i^{t+1}$ is determined by the following equations.

$$\mathbf{v}_i^{t+1} = w_0\mathbf{v}_i^t + c_1r_1(\boldsymbol{pbest}_i - \mathbf{x}_i^t) + c_2r_2(\boldsymbol{gbest} - \mathbf{x}_i^t) \tag{2.3}$$

$$\mathbf{x}_i^{t+1} = \mathbf{x}_i^t + \mathbf{v}_i^{t+1} \tag{2.4}$$

where w_0, c_1, and c_2 are parameters predetermined by users, r_1 and r_2 are uniformly distributed random number in [0, 1].

There are three parts in equation (2.3): the first part $w_0\mathbf{v}_i^t$ is known as inertia and represents the scaled velocity. It gives the necessary momentum of candidates to move in the search space. The second part $c_1r_1(\boldsymbol{pbest}_i - \mathbf{x}_i^t)$ forces the candidates to move toward their own best position so far. Finally, the third part $c_2r_2(\boldsymbol{gbest} - \mathbf{x}_i^t)$ guides all candidates to search in the direction of the global best. When a new velocity is obtained, it is added to the current candidate and a new candidate is generated by equation (2.4). With this update process, the framework of PSO is summarized in Figure 2.5.

Randomly initialize a swarm with *N* candidates
Give the values of hyper-parameters w_0, c_1, and c_2

for *i* = 1 to *N*
    randomly initialize velocity vector $\mathbf{v}_i^0$
end for

for *i* = 1 to *N*
    $\boldsymbol{pbest}_i = \mathbf{x}_i^0$
end for
$\boldsymbol{gbest}$ = Argmax $(f(\mathbf{x}_1^0), f(\mathbf{x}_2^0), \ldots, f(\mathbf{x}_N^0))$

t = 0
while (termination condition is not met)
    for *i* = 1 to *N*
        calculate the new velocity vector $\mathbf{v}_i^{t+1}$ by equation (2.3)
        calculate the new particle $\mathbf{x}_i^{t+1}$ by equation (2.4)

        if $(f(\mathbf{x}_i^{t+1}) > f(\boldsymbol{pbest}_i))$
            $\boldsymbol{pbest}_i = \mathbf{x}_i^{t+1}$

            if $(f(\boldsymbol{pbest}_i) > f(\boldsymbol{gbest}))$
                $\boldsymbol{gbest} = \boldsymbol{pbest}_i$
            end if
        end if
    end for

    t = *t* + 1
end while

Fig. 2.5 Framework of original PSO

2.1.4 Firefly Algorithm

Firefly algorithm (FA) is inspired by the flashing patterns of fireflies. In the original FA, a candidate $\mathbf{x}_i$ is also known as a '*firefly*' and any two fireflies $\mathbf{x}_i$ and $\mathbf{x}_j$ attract each other according to their '*brightness*'. When solving optimization problems, the 'brightness' of $\mathbf{x}_i$ is an alias of the fitness of $\mathbf{x}_i$.

A firefly's attractiveness is proportional to its intensity seen by other fireflies but is inversely proportional to the distance between two fireflies. Let β_0 be the attractiveness at the source, γ be a fixed light absorption coefficient, and r be the distance between two fireflies, the attractiveness is defined as:

$$\beta(r) = \beta_0 e^{-\gamma r^2} \tag{2.5}$$

When a firefly $\mathbf{x}_i^t$ for the t^{th} generation is updated, it is attracted by another brighter firefly $\mathbf{x}_j^t$ (i.e. the fitness of $\mathbf{x}_j^t$ is higher than that of $\mathbf{x}_i^t$). The updating rule is given below:

$$\mathbf{x}_i^{t+1} = \mathbf{x}_i^t + \beta_0 e^{-\gamma r_{ij}^2}\left(\mathbf{x}_i^t - \mathbf{x}_j^t\right) + \alpha\epsilon_t \tag{2.6}$$

where r_{ij} is the distance between $\mathbf{x}_i^t$ and $\mathbf{x}_j^t$, α is a user determined scale factor, ϵ_t is a random vector drawn from Gaussian or uniform distribution. Equation (2.6) states that the new firefly $\mathbf{x}_i^{t+1}$ is determined by the attractiveness define in equation (2.5) and a small turbulent vector.

It should be noted that FA uses a 'double-loop' structure to update the whole swarm. The framework of original FA is given in Figure 2.6. Because of its 'double-loop' structure, the time complexity of FA is $O(N^2\text{MaxGen})$ at the extreme case, where N is the swarm size and MaxGen is the maximum generations [Yange 13]. This disadvantage makes FA spend more evaluation times than PSO and ABC to find a solution.

```
Initialize the swarm randomly
Give the values of hyper-parameters β0, γ, and α

while (termination condition is not met)
    for i = 1 to N:
        for j = 1 to N:
            if (f(x_i) < f(x_j))
                Update x_i using equation (2.6)
                (i.e. moving x_i to another candidate with higher fitness)
            end if
        end for
    end for

    sort candidates and find the current best
end while
```

Fig. 2.6 Framework of original FA

2.1.5 Cuckoo Search Algorithm

Cuckoo search (CS) algorithm is also a swarm intelligence algorithm. A noticeable feature of CS is that it models the flight and feeding behavior of cuckoo using *Levy distribution* [Brown 07]. A *Levy Flight* (LF) is a random walk that uses Levy distribution. Figure 2.7 shows that compared with the standard random walk, LF makes some 'long jumps' among many small steps. This feature of FL is very useful and effective in global numerical optimization [Ali 15].

After initializing a swarm with *N* candidates, original CS updates a candidate $\mathbf{x}_i$ by two operators: *LF operator* and *parasitism operator*. In LF operator, a new candidate $\mathbf{x}_i^{new}$ is generated using equation (2.7):

$$\mathbf{x}_i^{new} = \mathbf{x}_i + \alpha \times Levy(\beta) \tag{2.7}$$

where $\alpha > 0$ is the step size that depends on the given problem and $\beta \in (1, 3]$ is the parameter of Levy distribution. Equation (2.7) implies that $\mathbf{x}_i$ is tuned by small steps most of the time but occasionally 'jumps' to another far place. In parasitism operator, with probability p_a, some 'bad' candidates (i.e. the candidates with low fitness) are replaced by some 'better' candidates generated randomly. The framework of original CS is given in Figure 2.8.

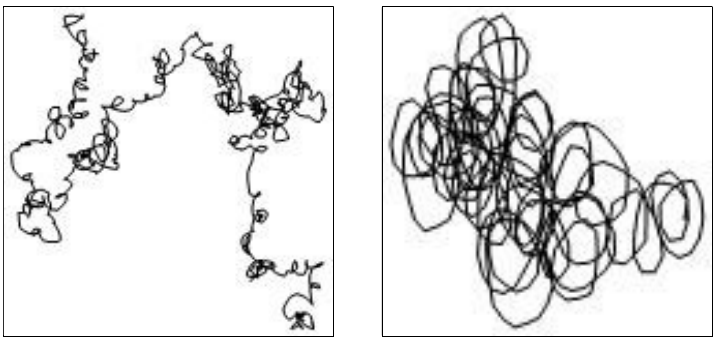


Fig..2.7 Levy Flight (left) and standard random walk (right)

Initialize a swarm with *N* candidates
Give the values of hyper-parameters α, β, and p_a

while (termination condition is not met)
  [**LF operator**]
  for *i* = 1 to *N*
    $\mathbf{x}_i^{new} = \mathbf{x}_i + \alpha \times Levy(\beta)$
    if $(f(\mathbf{x}_i^{new}) > f(\mathbf{x}_i))$
      replace $\mathbf{x}_i$ by $\mathbf{x}_i^{new}$
    end if
  end for

  [**parasitism operator**]
  for *i* = 1 to *N*
    if (random number $< p_a$)
      generate a new candidate $\mathbf{x}_i^{new}$ randomly
      if $(f(\mathbf{x}_i^{new}) > f(\mathbf{x}_i))$
        replace $\mathbf{x}_i$ by $\mathbf{x}_i^{new}$
      end if
    end if
  end for
end while

Fig. 2.8 Framework of original CS

2.1.6 Short Summary

We have introduced four swarm intelligence algorithms and we list their basic information in Table 2.1 as a short summary. From Table 2.1, we find that ABC has the smallest number of control parameters.

Table 2.1 Short summary of ABC, PSO, FA, and CS

Algorithms	Author	Parameters
ABC	[Karaboga, 07]	*limit*
PSO	[Kennedy 95]	w_0, c_1, c_2
FA	[Yang 09]	γ, β_0, α
CS	[Yang 09]	α, β, p_a

2.2 Graph Coloring Problems

2.2.1 Definition of Graph Coloring Problems

Graph coloring problem (GCP) is a famous combinatorial optimization problem in the field of graph theory. Here we define GCP more formally.

Definition. Let $G = (V, E)$ be an undirect graph with a set of n nodes V and a set of m edges E. Given such a graph and k colors, GCP asks whether this graph can be colored such that no nodes that connected by an edge share the same color.

For convenience, we call the number of nodes (i.e. n) *graph size* and two nodes connected by an edge *adjacent nodes*. The purpose of solving GCP is to color a graph G with k colors and the adjacent nodes should be given different colors. Commonly, the m edges are known as m *constraints*. If G can be colored by exactly k colors, we call G a *k-colorable graph* and the corresponding problem *k-GCP*. For example, if G can be colored by three colors, G is a 3-colorable graph and the problem is 3-GCP.

GCP is an *NP-complete* problem and is considered intractable because (1) its solution space grows exponentially in relation to the graph size and (2) there is no known polynomial bounded algorithm for solving it [Cook 71]. This means that an algorithm for locating a solution to GCP must resort to searching a significant portion of the solution space. In this work, we focus on solving GCP by discrete ABC.

As mentioned in 2.1.1, to solve such an optimization problem, we should first convert the problem to an *objective function*. We define the objective function for GCP as follows. Let the n nodes be labeled by natural number 1, 2, …, n successively and the color set be $C = \{1, 2, \dots, k\}$. Note that instead of using real colors such as red, blue, or green, we use integers to represent the colors. A candidate solution $\mathbf{x}$ is a *color vector* with n elements and each element is from the color set C, or more formally, $\mathbf{x} = \{x_1, x_2, \dots, x_n | \forall x_i \in C, i \in \{1, 2, \dots, n\}\}$. For example, $x_2 = 3$ implies that the second node is assigned a color 3. The total conflict of $\mathbf{x}$ is:

$$con(\mathbf{x}) = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} conflict(i,j) \tag{2.8}$$

where

$$conflict(i,j) = \begin{cases} 1, & i,j \text{ are adjacent nodes and have same color} \\ 0, & otherwise \end{cases} \tag{2.9}$$

Then, using equation (2.8), we define the objective function in equation (2.10):

$$fitness(\mathbf{x}) = 1.0 - \frac{con(\mathbf{x})}{m} \tag{2.10}$$

Obviously, a candidate $\mathbf{x}$ is a solution *if and only if* $fitness(\mathbf{x}) = 1.0$ because there are no adjacent nodes assigned the same color and $con(\mathbf{x}) = 0$.

2.2.2 Topology of Graph

There are many graph topologies, some of which have already been solved. Here, we concisely introduce complete graph, bipartite graph, cycle graph, wheel graph, and planar graph.

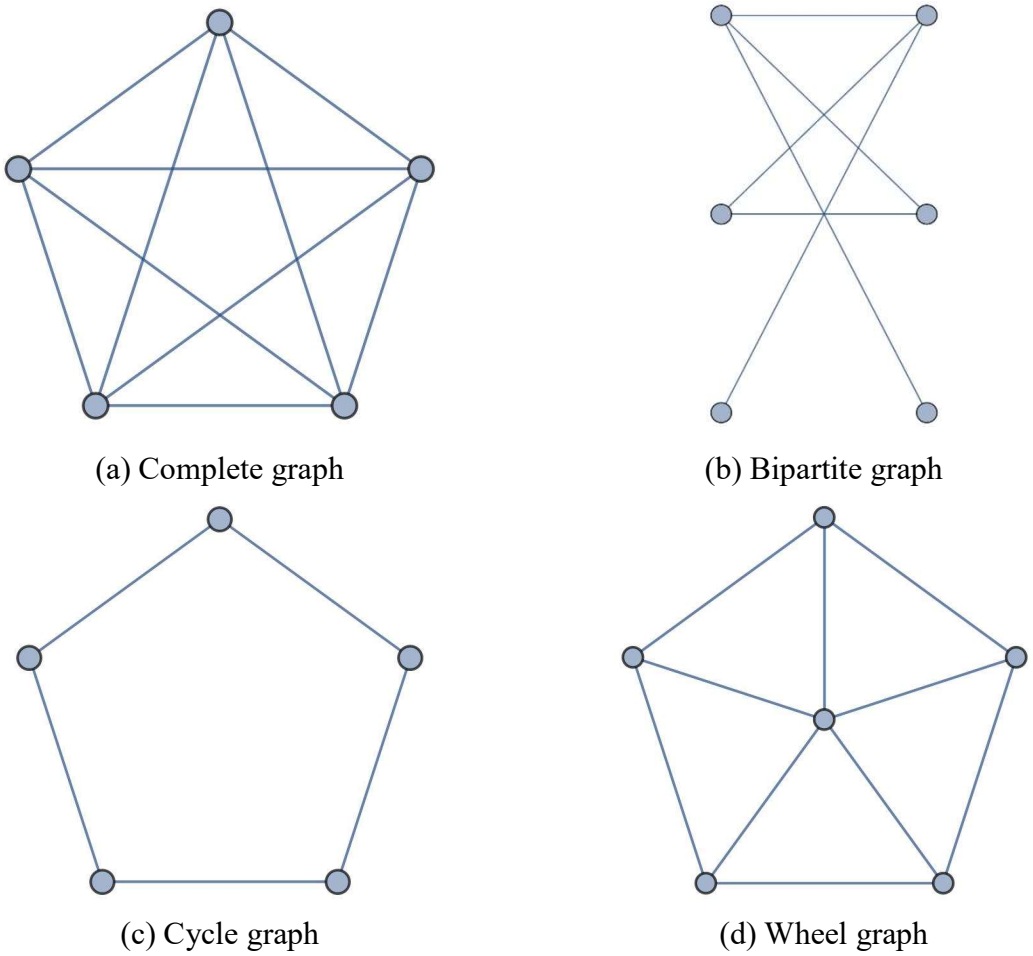
A *complete graph* with *n* nodes, which is denoted by K_n, is a graph whose any pair of nodes are connected by an edge, or in other words, any two nodes are adjacent nodes. Thus, a complete graph must have $m = \frac{n(n-1)}{2}$ edges. Obviously, because all nodes in complete graph are adjacent, the colors used to solve GCP on a complete graph must be *n*, i.e., all nodes should be assigned to their own individual color. An example of complete graph is given in Figure 2.9 (a).

A *bipartite graph*, which is denoted by $G = (V_1, V_2, E)$, is a graph whose *n* nodes are separated into two sets V_1 and V_2 and edges are only added between nodes that belong to different sets. This means that there is no edge between any two nodes in the same set. Thus, obviously, only two colors are needed to solve GCP on a bipartite graph. An example of bipartite graph is given in Figure 2.9 (b).

A *cycle graph*, which is denoted by C_n where $n \geq 3$, is a graph whose nodes are connected one by one like a 'circle'. Or more precisely, a cycle graph has an edge set $E = \{\{v_1, v_2\}, \{v_2, v_3\}, \dots, \{v_{n-1}, v_n\}, \{v_n, v_1\}\}$. The number of colors used to solve GCP on a cycle graph is dependent on graph size *n*: two colors are needed when *n* is even while three colors are needed when *n* is odd. An example of cycle graph is given in Figure 2.9 (c).

A wheel graph with *n* nodes, which is denoted by W_n, is a graph derived from a cycle graph. To construct a wheel graph, we use the first *n* - 1 nodes to generate a cycle graph C_{n-1} and then, add edges between the last node (i.e. the n^{th} node) and other *n* – 1 nodes. That is adding edges $\{\{v_1, v_n\}, \{v_2, v_n\}, \dots, \{v_{n-1}, v_n\}\}$. Considering a cycle graph, we immediately know that three colors are needed to color a wheel graph when *n* is odd while four colors are needed when *n* is even. An example of wheel graph is given in Figure 2.9 (d).

Both cycle graph and wheel graph are cases of *planar graph*. A planar graph is a graph that can be drawn on a plane without any edges crossing. A famous conclusion on planar graph is *Four Color Theorem* [Appel 77, Appel 77], which states that *a planar graph can be feasibly colored by four or fewer colors*. Four Color Theorem is the first major theorem that is proved by using a computer and has been gained wide acceptance. One should remember that the converse theorem of Four Color Theorem does *not* hold. This means that even though a planar



(a) Complete graph (b) Bipartite graph

(c) Cycle graph (d) Wheel graph

Fig. 2.9 Four graph topologies

graph can be colored by four or fewer colors, *a 4-colorable graph may not be a planar graph*.

In this work, we test our algorithms on some more challenging *k-colorable random graphs*. A *k*-colorable random graph with *n* nodes is generated by first partitioning *n* nodes into *k* groups and then adding edges between nodes in different groups until some conditions are met. The topology of a *k*-colorable random graph is determined by the methods of partitioning nodes and adding edges. We focus on four classes of *k*-colorable random graphs: *Minton random graph*, *arbitrary-random graph*, *equi-partite graph*, and *flat graph*. Minton random graph is designed by S. Minton [Minton 92] and arbitrary-random graph, equi-partite graph, and flat graph are also known as *Culberson random graphs* that designed by J. Culberson [Culberson 96]

■ Minton random graph

Given a constraint density $d = \frac{m}{n}$ where *n* and *m* are the numbers of nodes and edges respectively, a *k*-colorable Minton random graph with *n* nodes is generated as follows:

① Calculate the number of edges by $m = n \times d$;
② Divide *n* nodes *equally* to *k* groups;
③ Add edges between nodes in different groups until the number of edges is *m*.

■ Arbitrary-random graph

Given an edge probability *p*, a *k*-colorable arbitrary-random graph with *n* nodes is generated as follows:

① Separate *n* nodes into *k* groups *randomly*, the numbers of nodes in various groups *may not be equal*;
② Add edges between nodes in different groups with edge probability *p*;
③ Accept the generated graph if it is a connected graph (i.e. a graph without 0 degree node).

■ Equi-partite graph

Given an edge probability *p*, a *k*-colorable equi-partite graph with *n* nodes is generated as follows:

① Separate *n* nodes into *k* groups *equally*;
② Add edges between nodes in different groups with edge probability *p*;
③ Accept the generated graph if it is a connected graph.

■ Flat graph

Given an edge probability *p*, a *k*-colorable flat graph with *n* nodes is generated as follows:

① Separate *n* nodes into *k* groups *equally*;
② Add edges between nodes in different groups with edge probability *p*;
③ Accept the generated graph if it is a connected graph *and the degrees of groups are equal*.

The differences of these graphs are as follows. Minton graph is determined by constraint density *d* and the number of edges *m* is fixed when *d* and *n* are given. Note that Minton random graph may not be connected graph. On the other hand, Culberson random graphs are generated according to edge probability *p*, which means the number of edges may not be equal between two instances even *n* and *p* are identical. In Culberson random graphs, the number of nodes of arbitrary random graph in various groups may not be equal because the nodes are divided randomly. On the contrary, equi-partite graphs and flat graphs keep the number of nodes in various groups as equal as possible. Flat graphs further keep the degrees of groups equal, in other words, the number of edges between any two groups is equal. Finally, Culberson random graphs are connected graphs.

2.2.3 Phase Transition

Interestingly, most *k*-GCPs are very easy to solve [Minton 90, Turner 88]. Thus, we should look for hard problem instances that pose challenges for candidate algorithms. Not only GCPs, all NP-complete problems have an *order parameter*, and the hard problems arise in a certain range of such parameter. The range of order parameter is known as a *phase transition* [Cheesman 91].

Phase transition can be found by experiment or some formula. [Cheesman 91] studied the phase transition of *k*-GCPs by experiments and [Hogg 94] derives some formula to calculate the phase transition. We introduce Hogg's formula in the context of *k*-GCPs concisely as follows.

Given a graph with *n* nodes and *k* colors, an average connectivity, i.e., the average number of edges connected with a node, is defined as γ. Then, the number of edges *m* in the graph is:

$$m = \frac{1}{2}\gamma n \tag{2.11}$$

Obviously, the constraint density *d*, which defined in 2.2.2, is:

$$d = \frac{m}{n} = \frac{1}{2}\gamma \tag{2.12}$$

Hogg defines β as a critical parameter used to find the hardest instances:

$$\beta = \frac{1}{2}\gamma k = d \times k \tag{2.13}$$

The purpose of Hogg's research is to find a critical value of β that generates the hardest problems. The basic idea to find such β is to calculate the average cost to find all solutions of *k*-GCP $\langle C \rangle$ and the average number of solutions $\langle N_{soln} \rangle$, then, the cost to find the first solution or to failure can be approximately calculated as:

$$\langle C_{ap} \rangle \approx \frac{\langle C \rangle}{\max(1, \langle N_{soln} \rangle)} \tag{2.14}$$

Based on this idea, Hogg reports that the critical value of β, which generates the hardest *k*-GCPs, is:

$$\beta_{crit} = -\frac{\ln(k)}{\ln(1 - k^{-2})} \tag{2.15}$$

Using equation (2.15), one can easily calculate the phase transition on constraint density *d* and

edge probability p (see 2.2.2).

Another interesting and subtle thing that should be noted is that *the theory value of phase transition is a little different from the experiment value*. [Williams 92] reports this phenomenon. We summarize the theory and experiment values of phase transitions given in [Williams 92] and [Cheesman 91] on 3-GCPs, 4-GCPs, and 5-GCPs in Table 2.2.

Table 2.2 shows that the experiment value of phase transition is smaller than theory value of phase transition. From the experiments performed later, we will find that the experiment value is more accurate than the theory value. For example, the 3-colorable Minton random graphs at $d = 2.5$ (experiment value) are harder than the ones at $d = 3.1$ (theory value).

The difficulty of a random graph does not only depend on phase transition, but also on its topology. Culberson [Culberson 96] finds that (1) the most difficult problems occur when the number of nodes in various node groups is equal, and (2) the larger the variation of degree between two node groups, the easier an algorithm finds a solution. Thus, he reports that the flat graphs are much harder than arbitrary-random graphs and equi-partite graphs, even all of them have same graph size n and edge probability p.

2.2.4 Short Summary

We have introduced several graph topologies, some of which are easy to solve. In this work, we select Minton random graph and Culberson random graph (include arbitrary-random graph, equi-partite graph, and flat graph) because they are more challenging.

There are two factors that affect the difficulty of a random graph: phase transition and topology. Phase transition is a range of a specific parameter (such as constraint density d and edge probability p) that generates hard problems, and topology is the method used to construct a graph. When graphs are constructed by the same process, for example, all graphs are flat graphs with 90 nodes, the phase transition dominates the difficulty of these graphs.

Table 2.2 The theory and experiment values of phase transitions on 3-GCP, 4-GCP, and 5-GCP

k	T/E	β_{crit}	γ_{crit}	d	p
3	Theory value	9.3	6.2	3.1	$8/n \sim 9/n$
	Experiment value	7.5	5.0	2.5	$7/n \sim 8/n$
4	Theory value	21.5	10.7	5.4	$13/n \sim 14/n$
	Experiment value	18.4	9.2	4.6	$12/n \sim 13/n$
5	Theory value	39.4	15.8	7.9	$19/n \sim 20/n$
	Experiment value	35.5	14.2	7.1	$17/n \sim 18/n$

On the other hand, when generating random graphs with identical graph size and phase transition, it is the topology that affects the difficulty of generated graphs.

Studies also show that the theory value and the experiment value of phase transition are not identical but have a small gap.

2.3 Related Works

2.3.1 DSatur [Brelaz 79]

As mentioned in 1.1, DSatur is a classical heuristic algorithm that is specially designed for solving GCPs. DSatur is an improved version of greedy algorithm. Unlike greedy algorithm, in which the order of coloring nodes is determined before coloring process, DSatur colors each node dynamically during the process. DSatur algorithm is given in Figure 2.10.

DSatur chooses the node that will be colored next according to *saturation degree* of nodes. The saturation degree of a node is the number of different colors assigned to its adjacent nodes. At the first iteration, the saturation degree of all nodes is 0, thus DSatur colors the node that has the largest degree. Then, the saturation degree of all adjacent nodes of the colored node will be

```
Let X = {1, 2, ..., n} and S = {}
when (X is not ∅)
    if (S is ∅)
        i = the node with largest degree
    else
        i = the node with largest saturation degree
    end if

    if (there is feasible color to color node i)
        node i is assigned to the feasible color
    else
        generate a new color
        assign node i the new color
    end if

    S = S ∪ i
    remove i from X
    update the saturation degree of all nodes in X
end while
```

Fig. 2.10 Framework of DSatur algorithm

updated. In the following iterations, DSatur selects the node that has the largest saturation degree and assigns it a feasible color. This process stops when all nodes have been colored.

Note that if there is no feasible color that can be assigned to a selected node, DSatur generates a new color and assign it to the selected node. Thus, *DSatur always colors a graph without conflict but the solution may not be optimal*. For example, even a random graph is 3-colorable, DSatur may color it with 4 or more colors.

2.3.2 Hybrid ABC [Fister 12]

Many original heuristic algorithms like DSatur can only color graphs of up to 100 nodes, thus some improvements are developed for solving large GCPs. Local search and hybrid algorithm are two main methods. For example, *Tabucol* [Hertz 87] and *hybrid evolutionary algorithm* (HEA) [Galinier 99] are representative local search algorithm and hybrid algorithm respectively.

In Fister's work, DSatur is hybridized with ABC to solve 3-GCPs. Here, ABC is used to tune real number weights for nodes: the larger the weight is, the higher priority the node will be selected by DSatur. There are three features in Hybrid ABC algorithm:

(1) Scout bee phase is replaced by *random walk with direction exploitation* (RWDE) [Rao 09] local search heuristic. If a candidate $\mathbf{x}_i$ cannot be improved for *limit* generations, instead of replacing it by a new randomly generated candidate, $\mathbf{x}_i$ is replaced by a new candidate that is generated by:

$$\mathbf{x}_i^{new} = \mathbf{x}_i + \lambda \times U_i \tag{2.16}$$

where λ is a predetermined scalar number and U_i is a unit random vector for $\mathbf{x}_i$.

(2) DSatur selects node to be colored by weights tuned with ABC. Unlike original DSatur, which decide the node to be colored by saturation degree, in hybrid ABC, DSatur selects the node with highest weight.

(3) DSatur does not generate new colors. In original DSatur, if there is no feasible color for a selected node, a new color is generated and assigned to the node. However, in hybrid ABC, DSatur is used to evaluate the candidates generated by ABC, thus, if a node cannot be colored because of conflict, the node is labeled '*uncolored*'. DSatur evaluates a candidate by the number of *uncolored nodes*. Obviously, if the number of uncolored nodes is 0, a solution is found.

The framework of hybrid ABC is given in Figure 2.11.

```
Initialize N candidates X = {x_1, x_2, ..., x_N} randomly
Let V = {v_1 = 0, v_2 = 0, ..., v_N = 0}
Where v_i is the number of uncolored nodes for x_i
Give the values of hyper-parameters λ and limit

while (termination condition is not met)

    [DSatur phase]
    for i = 1 to N
        evaluate x_i by DSatur
        update v_i
    end

    [ABC phase]
    employed bee phase (X, V)
    onlooker bee phase (X, V)
    for i = 1 to N
        if x_i cannot be improved in limit generations
            update x_i by equation (2.16)
        end if
    end for

end while
```

Fig. 2.11 Framework of hybrid ABC

2.3.3 HDPSO [Aoki 15]

HDPSO is a non-hybrid discrete PSO algorithm to solve 3-colorable Minton random graphs. At the beginning, HDPSO, like original PSO, randomly initializes a swarm with N candidates. However, a candidate is not a real value vector drawn from some distribution, but a color vector defined in 2.2.1, i.e., a candidate $\mathbf{x}_i = (x_{i1}, x_{i2}, \dots, x_{in})$, where $1 \leq i \leq N$ and n is graph size. The value of each element in $\mathbf{x}_i$ can only be drawn from discrete color domain $C = \{1, 2, 3\}$. The advantage of defining swarm in a discrete domain is that a candidate can be evaluated by objective function (2.10) directly.

The next task is to define the update rules for HDPSO. Because all candidates are in discrete domain, the update rule (equation (2.3) and (2.4)) used by original PSO cannot be applied directly. HDPSO separate equation (2.3) to three parts and uses transition probability to update a candidate $\mathbf{x}_i$ for the t^{th} generation.

First, the distance between two candidates is described by hamming distance $H(\mathbf{x}_i, \mathbf{x}_j)$,

which is the number of different elements between $\mathbf{x}_i$ and $\mathbf{x}_j$:

$$d(\mathbf{x}_i, \mathbf{x}_j) = 1 - \frac{H(\mathbf{x}_i, \mathbf{x}_j)}{n} \tag{2.17}$$

Then the velocity of a candidate for the t^{th} generation is defined as:

$$\mathbf{v}_i^t = d(\mathbf{x}_i^t, \mathbf{x}_i^{t-1}) \tag{2.18}$$

Next, the three terms of equation (2.3) are mapped on discrete domain:

$$w_0 \mathbf{v}_i^t \rightarrow V_{rand} = w_0 \mathbf{v}_i^t \tag{2.19}$$

$$c_1 r_1 (\boldsymbol{pbest}_i - \mathbf{x}_i^t) \rightarrow V_{pbest} = c_1 r_1 d(\boldsymbol{pbest}_i, \mathbf{x}_i^t) \tag{2.20}$$

$$c_2 r_2 (\boldsymbol{gbest} - \mathbf{x}_i^t) \rightarrow V_{gbest} = c_2 r_2 d(\boldsymbol{gbest}, \mathbf{x}_i^t) \tag{2.21}$$

Finally, the transition probabilities are:

$$V = V_{rand} + V_{pbest} + V_{gbest} \tag{2.22}$$

$$P_{rand} = \frac{V_{rand}}{V} \tag{2.23}$$

$$P_{pbest} = \frac{V_{pbest}}{V} \tag{2.24}$$

$$P_{gbest} = \frac{V_{gbest}}{V} \tag{2.25}$$

Using transition probabilities, HDPSO updates elements in a candidate one by one. The framework of HDPSO is given in Figure 2.11.

[Aoki 15] compares HDPSO with TPPSO [Kanoh 13] and GA on 3-colorable Minton random graphs with graph size 90, 120, and 150. Experiments show that HDPSO outperforms its competitors on small graph size (n = 90). Its performance gets worse when graph size increases.

```
Randomly initialize N candidates from discrete domain C = {1, 2, 3}
Give the values of hyper-parameters w_0, c_1, and c_2
Evaluate all candidates by equation (2.10)
Record gbest

while (termination condition is not met)

    for i = 1 to N
        calculate P_rand, P_pbest, and P_gbest for x_i

        for j = 1 to n
            generate a random number r ∈ (0, 1)

            if (r ≤ P_rand)
                x_ij = a random color from {1, 2, 3}
            else if (P_rand < r ≤ P_rand + P_pbest)
                x_ij = pbest_ij
            else
                x_ij = gbest_j
            end if
        end for

        evaluate new x_i by equation (2.10)
        update pbest_i
    end for

    update gbest

end while
```

Fig. 2.11 Framework of HDPSO

2.3.4 Discrete Cuckoo Search [Aranha 17]

Discrete Cuckoo Search (DCS) is also a non-hybrid algorithm that defines its swarm in discrete domain directly and extends the update rules of original CS to adapt the context of solving 3-GCPs.

Even though HDPSO has defined the discrete distance between two candidates by hamming distance, it is not appropriate to apply this idea in DCS because a candidate in CS is updated by Levy Flight (LF, see 2.1.5) without communicating with others.

To solve this problem, DCS modifies a candidate $\mathbf{x}_i$ by randomly selecting M elements of $\mathbf{x}_i$ and changing them to other colors. M is determined by discrete LF:

$$M = \lfloor \alpha \times Levy(\beta) \rfloor + 1 \tag{2.26}$$

Then, the two operators of original CS, LF operator and parasitism operator, are discretized by equation (2.26). The framework of DCS is given in Figure 2.12.

Initialize N candidates in discrete domain C = {1, 2, 3}
Give the values of hyper-parameters α, β, and p_a

while (termination condition is not met)

[**LF operator**]
for i = 1 to N
calculate M by equation (2.26)
randomly select M elements of $\mathbf{x}_i$
generate $\mathbf{x}_i^{new}$ by randomly changing the colors of the M elements

if $(f(\mathbf{x}_i^{new}) \geq f(\mathbf{x}_i))$
replace $\mathbf{x}_i$ with $\mathbf{x}_i^{new}$
end if
end for

[**Parasitism operator**]
for i = 1 to N
generate a random number $r \in (0, 1)$
if $(r < p_a)$
calculate M by equation (2.26)
randomly select M elements of $\mathbf{x}_i$
generate $\mathbf{x}_i^{new}$ by randomly changing the colors of the M elements
replace $\mathbf{x}_i$ with $\mathbf{x}_i^{new}$ without evaluation
end if
end for

end while

Fig. 2.12 Framework of DCS

The parasitism operator needs more explanations.

(1) In parasitism operator of original CS, $\mathbf{x}_i$ is not replaced by $\mathbf{x}_i^{new}$ unless the fitness of $\mathbf{x}_i^{new}$ is better. However, in DCS, the old candidate $\mathbf{x}_i$ is always replaced by $\mathbf{x}_i^{new}$ without evaluation. The author explains that always accepting the new candidate in parasitism improves the exploration power of the algorithm and avoids wasting evaluations. On the other hand, p_a is set to be a very small value so that parasitism operator is not called frequently.

(2) In [Aranha 17], *M* can be calculated by three methods: (a) using equation (2.26); (b) drawn randomly from a uniformly distribution; and (c) a fixed integer determined by some preliminary experiments. The author reports that method (a) obtains the best performance. Thus, for simplicity, *M* is determined by equation (2.26) in Figure 2.12.

Experiments are performed on 3-colorable Minton random graphs with graph size from 90 to 180. Results show that DCS is a powerful algorithm to solve 3-GCPs on Minton random graphs and its performance does not decay when graph size increases. The author states that determining *M* by equation (2.26) gives self-adaptation to some extent because it avoids the cost of finding the optimal *M* by some preliminary experiments.

Chapter 3 Similarity Artificial Bee Colony and Firefly Algorithm

In this chapter, we propose similarity artificial bee colony (S-ABC) algorithm to solve 3-GCPs. We first introduce similarity to discretize original ABC. Then with the same idea, we discretize original FA and propose D-FA algorithm as a comparison algorithm. Finally, using HDPSO as baseline, we compare S-ABC, D-FA, and HDPSO on 3-colorable Minton random graphs with various graph size to show the performance of S-ABC.

3.1 Encoding Scheme

Before giving the detail of the proposed methods, we define an encoding scheme for a swarm formally. *This encoding scheme will be used throughout the rest of this thesis*.

A swarm consists of N candidates, and in the context of k-GCPs, each candidate is a color vector that represents a candidate solution. Formally, given a graph with n nodes, we define color set C, swarm S, and a candidate $\mathbf{x}_i$ as follows:

$$C = \{1, 2, \dots, k\} \tag{3.1}$$

$$S = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\} \tag{3.2}$$

$$\mathbf{x}_i = \{x_{i1}, x_{i2}, \dots, x_{in} | \forall x_{ij} \in C \wedge j \in \{1, 2, \dots, n\}\} \tag{3.3}$$

Color set C includes k labels that denote k colors. For convenience, we use label 1, 2, …, k instead of using actual colors such as red, green, and blue (equation (3.1)). Swarm S is composed by N candidates, each of which is a color vector (equation (3.2)). A candidate $\mathbf{x}_i$ has n elements, each element can only be assigned a label from C and represents the 'color' assigned to the corresponding node (equation (3.3)).

At the beginning of a swarm intelligence algorithm, a swarm is initialized randomly. For example, let N = 4, k = 3, and n = 6, an example of a feasible initialized swarm is given in Figure 3.1.

$$C = \{1, 2, 3\}$$

$$S = \{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4\}$$

$$\mathbf{x}_1 = \{1, 2, 1, 3, 3, 2\}$$

$$\mathbf{x}_2 = \{3, 1, 3, 1, 3, 2\}$$

$$\mathbf{x}_3 = \{1, 2, 1, 3, 3, 2\}$$

$$\mathbf{x}_4 = \{1, 3, 2, 1, 2, 2\}$$

Fig. 3.1 An example of a randomly initialized swarm

As mentioned above, a candidate $\mathbf{x}_i$ is also known as a candidate solution. During a swarm intelligence algorithm, $\mathbf{x}_i$ is updated by predefined rules and evaluated by an *objective function*. Because we focus on *k*-GCPs, the objective function is defined as equation (2.10) in (2.2.1). For the convenience, we rewrite it below:

$$fitness(\mathbf{x}) = 1.0 - \frac{con(\mathbf{x})}{m} \tag{3.4}$$

where $con(\mathbf{x})$ is the total conflict of candidate $\mathbf{x}$ (see equation (2.8) in (2.2.1)) and *m* is the number of edges. Equation (3.4) calculates the *fitness* of a candidate $\mathbf{x}$. The purpose of any swarm intelligence algorithm in this work is to find a candidate $\mathbf{x}$ with $fitness(\mathbf{x}) = 1.0$, which means no adjacent nodes share the same color, i.e. $con(\mathbf{x}) = 0$.

3.2 S-ABC

3.2.1 Similarity

In HDPSO, the distance between two candidates are defined by hamming distance. We use this idea to construct *similarity* of two candidates $\mathbf{x}_i$ and $\mathbf{x}_j$:

$$s(\mathbf{x}_i, \mathbf{x}_j) = 1 - \frac{H(\mathbf{x}_i, \mathbf{x}_j)}{n} \tag{3.5}$$

where $s(\mathbf{x}_i, \mathbf{x}_j)$ is the similarity of $\mathbf{x}_i$ and $\mathbf{x}_j$, and *n* is the graph size.

Similarity describes the similar degree of $\mathbf{x}_i$ and $\mathbf{x}_j$, and we use it to discretize the original ABC.

3.2.2 Update Rules of S-ABC

In original ABC, employed bee phase and onlooker bee phase generates a new candidate $\mathbf{x}_i^{new}$ from $\mathbf{x}_i$ by randomly selecting a *neighbor candidate* $\mathbf{x}_j$ and then invoking equation (2.2). In the proposed method, instead of choosing a neighbor candidate randomly, we select a neighbor candidate that is *similar* to $\mathbf{x}_i$. Similarity defined in (3.2.1) is used to measure how similar two candidates are.

Once a neighbor candidate $\mathbf{x}_j$ is determined, we update $\mathbf{x}_i$ by randomly choose t elements of $\mathbf{x}_j$ and replace the corresponding elements of $\mathbf{x}_i$ by them. If $t = 1$, this update rule is identical to the original one, which generates a new candidate by adding a small turbulent on just one element of $\mathbf{x}_i$. The update process of S-ABC is given in Figure 3.2.

Give a candidate $\mathbf{x}_i$ to be updated

Update-candidate ($\mathbf{x}_i$)

[**Step 1. Select a neighbor by similarity**]

randomly select a candidate $\mathbf{x}_j$

calculate $s(\mathbf{x}_i, \mathbf{x}_j)$ by equation (3.5)

generate a uniformly distributed random number $r \in (0, 1)$

if $(r < s(\mathbf{x}_i, \mathbf{x}_j))$

accept $\mathbf{x}_j$ as a neighbor candidate

end if

[**Step 2. Update $\mathbf{x}_i$**]

randomly select t elements from $\mathbf{x}_j$
replace the corresponding elements of $\mathbf{x}_i$ by them

return $\mathbf{x}_i^{new}$

end process

Fig. 3.2 The update rule of S-ABC

3.2.3 Framework of S-ABC

The update rule defined in 3.2.2 is now applied in employed bee phase and onlooker bee phase. As original ABC does, employed bee phase of S-ABC updates every candidate once while onlooker bee phase uses roulette selection to exploit the candidates with high fitness. Employed bee phase and onlooker bee phase are given in Figure 3.3.

The scout bee phase of S-ABC is identical to that of original ABC: if a candidate cannot be improved for limit generations, it is abandoned and is replaced by a randomly generated new candidate. The framework of S-ABC is given in Figure 3.4.

```
Given a swarm S
employed bee phase (S)
    for i = 1 to N
        x_i^new = Update-candidate (x_i)

        if (fitness(x_i^new) ≥ fitness(x_i))
            replace x_i with x_i^new
        end if
    end for

onlooker bee phase (S)
    for i = 1 to N
        choose x by roulette selection
        x^new = Update-candidate (x)

        if (fitness(x^new) ≥ fitness(x))
            replace x with x^new
        end if
    end for
```

Fig. 3.3 Employed bee phase and onlooker bee phase of S-ABC

```
Initialize a swarm S
Set parameters t and limit

while (termination condition is not met)
    employed bee phase (S)
    onlooker bee phase (S)
    scout bee phase (S)
end while
```

Fig. 3.4 Framework of S-ABC

3.2.4 Short Summary

In this section, we proposed a non-hybrid swarm intelligence algorithm S-ABC. By defining similarity between two candidates, S-ABC discretizes original ABC directly.

There are two main difference between S-ABC and original ABC: (1) when updating a candidate $\mathbf{x}_i$, S-ABC only selects a neighbor candidate $\mathbf{x}_j$ that is similar to $\mathbf{x}_i$; (2) unlike the original ABC, S-ABC randomly selects t elements of $\mathbf{x}_i$ and replaces them by the corresponding elements of $\mathbf{x}_j$. Here $t \geq 1$. This means that one or more elements will be changed.

Finally, there are two parameters in S-ABC: t and *limit*, the former affects the update of a candidate and the latter removes candidates that cannot be improved.

3.3 Discrete Firefly Algorithm

For the sake of comparison, we apply similarity defined in S-ABC to discretize original firefly algorithm (FA) and propose a discrete firefly algorithm (D-FA).

As 2.1.4 shows, original FA updates a candidate $\mathbf{x}_i$ by equation (2.6), which has two main terms: (1) $\beta_0 e^{-\gamma r_{ij}^2}\left(\mathbf{x}_i^t - \mathbf{x}_j^t\right)$ and (2) $\alpha\epsilon_t$. The former moves $\mathbf{x}_i$ to a better candidate $\mathbf{x}_j$ and the latter adds some turbulent to $\mathbf{x}_i$. Recall that HDPSO separates the original update rule into three parts, we can also discretize (1) and (2) respectively. For the convenience, we call the first term 'β-step' and the second term 'α-step'.

3.3.1 Discretize β-step with Similarity

In D-FA, we use similarity $s\left(\mathbf{x}_i, \mathbf{x}_j\right)$ defined in 3.2.1 to measure the difference between two candidates. The attractiveness between $\mathbf{x}_i$ and $\mathbf{x}_j$ can be calculated as:

$$\beta\left(\mathbf{x}_i, \mathbf{x}_j\right) = \beta_0 e^{-\gamma s\left(\mathbf{x}_i, \mathbf{x}_j\right)^2} \tag{3.6}$$

where β_0 is the attractiveness at the source and γ is the light absorption coefficient.

Then, for each element of $\mathbf{x}_i$, a uniformly distributed random number $r \in (0, 1)$ is generated and compared with $\beta\left(\mathbf{x}_i, \mathbf{x}_j\right)$. If $r \leq \beta\left(\mathbf{x}_i, \mathbf{x}_j\right)$, the element of $\mathbf{x}_i$ is replaced by the corresponding element of $\mathbf{x}_j$. This process is given in Figure 3.5.

3.3.2 Discretize α-step

In original α-step, a small random turbulent is added to some elements of $\mathbf{x}_i$. Similarly, we randomly change α elements of $\mathbf{x}_i$ to other colors. This process is given in Figure 3.6.

3.3.3 Framework of D-FA

Now, we establish the framework of D-FA. It is similar to the original one. For each generation, a candidate with low fitness moves toward other candidates with higher fitness by invoking β-step and α-step successively. Once all candidates have been updated, they are sorted according to the fitness in descending order and the current best candidate is saved. The framework of D-FA is given in Figure 3.7.

```
β-step (x_i, x_j)
    calculate s(x_i, x_j) by equation (3.5)
    calculate β(x_i, x_j) by equation (3.6)
    for q = 1 to n
        generate a uniform random number r ∈ (0, 1)
        if r ≤ β(x_i, x_j)
            replace x_iq with x_jq
        end if
    end for
```

Fig. 3.5 β-step in D-FA

```
α-step (x_i)
    randomly select α elements of x_i
    replace them with other colors
```

Fig. 3.6 α-step in D-FA

```
Initialize a swarm S randomly
Set parameters β0, γ, and α

while (termination condition is not met)

    for i = 1 to N
        for j = 1 to N
            if (fitness(x_i) ≤ fitness(x_j))
                β-step (x_i, x_j)
                α-step (x_i)
            end if
        end for
    end for

    sort S in descending order and record the current best

end while
```

Fig. 3.7 Framework of D-FA

3.4 Experiments

3.4.1 Experiment Design

We design some experiments to investigate the performance of S-ABC and D-FA. Using HDPSO as a baseline, we test S-ABC, D-FA, and HDPSO on 3-colorable Minton random graphs with various graph size. The experiment steps are given as follows.

■ Step 1. Generate Minton random graphs

Let $k = 3$, we first generate 3-colorable Minton random graphs with various graph size n and constraint density d using the steps given in (2.2.2). For each n and d, 50 instances are generated.

As described in (2.2.3), the difficulty of random graphs with same topology is determined by phase transition of the order parameter. For Minton random graphs, the order parameter is the constraint density d, and according to Table 2.2, the hardest problems occur when $d = 2.5$. The details of generated graphs are given in Table 3.1.

Table 3.1 The details of generated Minton random graphs

n	*d*	Phase transition	The number of graph instances for each *d*
90	1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 5.0, 6.0	2.5	50
120			
150			

■ Step 2. Parameter dependence

Before comparison, we should first find the optimal values of hyper-parameters for each algorithm. The details are given in next section.

■ Step 3. Comparison study

After the optimal hyper-parameters have been found, S-ABC, D-FA, and HDPSO are compared on Minton random graphs with various *n* and *d*.

Two measures are used to evaluate the performance of a given algorithm: *success rate* (SR) and *average evaluation times on success trials* (AES). SR is the percentage of graph instances in which a solution is found. AES represents the average number of evaluation times of objective function (i.e. equation (3.4)) in successful graph instances.

For example, we test S-ABC on Minton random graphs with $n = 90$ and $d = 2.5$. Let the total number of graph instances we test on be N_T and the number of success instances be N_S. SR is defined in equation (3.7):

$$SR = \frac{N_S}{N_T} \tag{3.7}$$

and AES is defined in equation (3.8):

$$AES = \frac{\sum_{i=1}^{N_S} e_i}{N_S} \tag{3.8}$$

where e_i is the evaluation times of objective function of the i^{th} success instance.

Note that in some literature, *time complexity* such as *CPU time in seconds* is also used to evaluate algorithms. However, such measures are hardware-dependent, and they are often affected by the spec of computers. Thus, in this work, we adopt SR and AES, which are hardware independent measure, to evaluate our algorithms.

3.4.2 Parameter Dependence

(1) S-ABC

There are two hyper-parameters in S-ABC: *t* and *limit*. To find their optimal values, we first fix *limit* to 100 and try different values of *t* on 30 graphs with $n = 90$ and $d = 2.5$. The value that obtains the highest SR is adopted to the optimum of *t*. Then, we fix *t* to its optimal value and try different values of *limit* on the same graph instances to find the optimum of *limit*. The experiment settings are given in Table 3.2. and the results are given in Figure 3.8. According to the results, we accept $t = 2$ and $limit = 80$ as their optimal values.

(2) D-FA

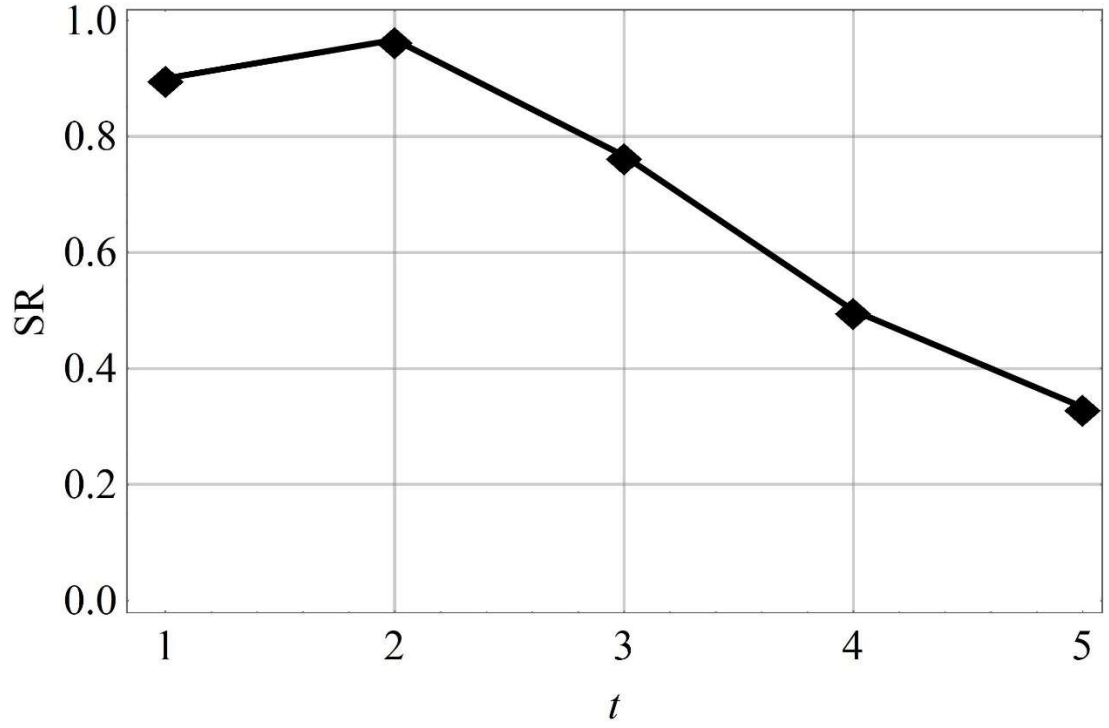
We optimize the hyper-parameters β_0, γ, and α with the same method. The experiment settings are given in Table 3.3 and the results are shown in Figure 3.9. We accept $\beta_0 = 1.0$, $\gamma = 0.2$, and $\alpha = 2$ as their optimal values. Note that if there are several values of objective parameter that obtain same SR, the one that consumes the smallest AES is adopted as the optimum. For example, in Figure 3.9(a), $\gamma = 0.2$, $\gamma = 0.3$, and $\gamma = 0.7$ obtain the same SR 0.97. We adopt $\gamma = 0.2$ as the optimal γ because it consumes the smallest AES ($AES_{\gamma=0.2} = 2.39 \times 10^7$) while $AES_{\gamma=0.3} = 3.20 \times 10^7$ and $AES_{\gamma=0.7} = 7.12 \times 10^7$ respectively.

Table 3.2 Experiment settings to find the optimal parameters of S-ABC

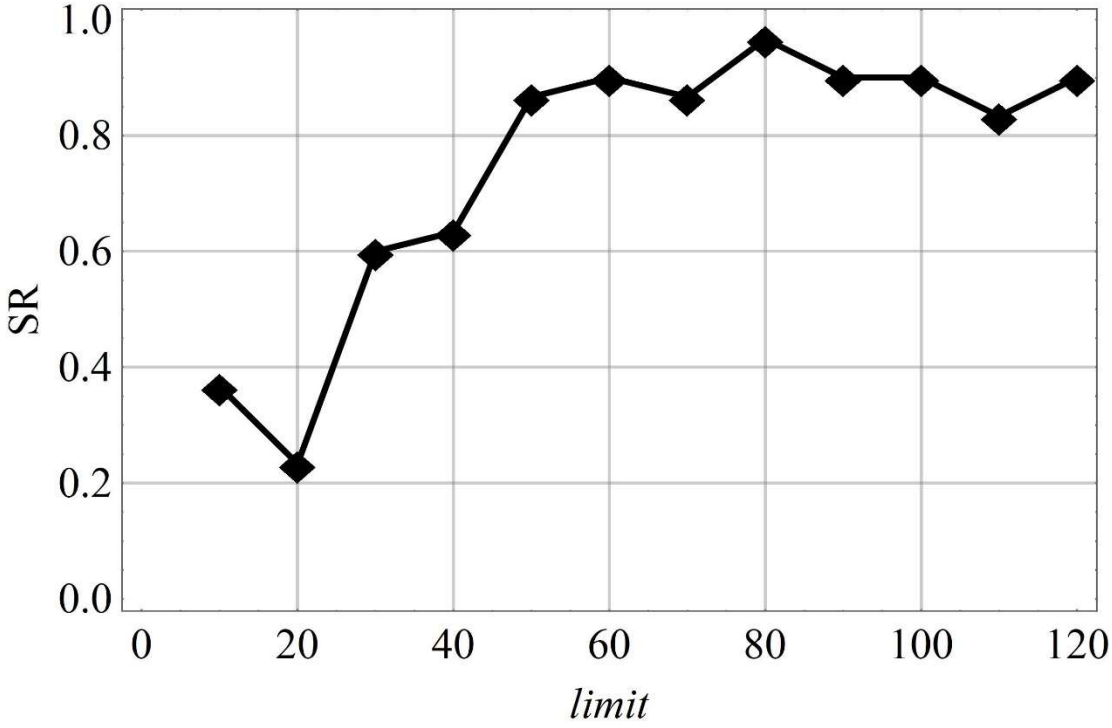
Experiment	*limit*	*t*	Swarm size (*N*)	Max generation	Graph settings
Find optimal *t*	100	1 ~ 5	200	10000	$n = 90$ $d = 2.5$ 30 instances
Find optimal *limit*	10 ~ 120	Optimal *t*			

Table 3.3 Experiment setting to find the optimal parameters of D-FA

Experiment	β_0	γ	α	Swarm size (*N*)	Max generation	Graph settings
Find optimal γ	1.0	0.1~1.3	3	200	10000	$n = 90$ $d = 2.5$ 30 instances
Find optimal β_0	0.1~1.2	Optimal γ	3			
Find optimal α	Optimal β_0	Optimal γ	1~5			

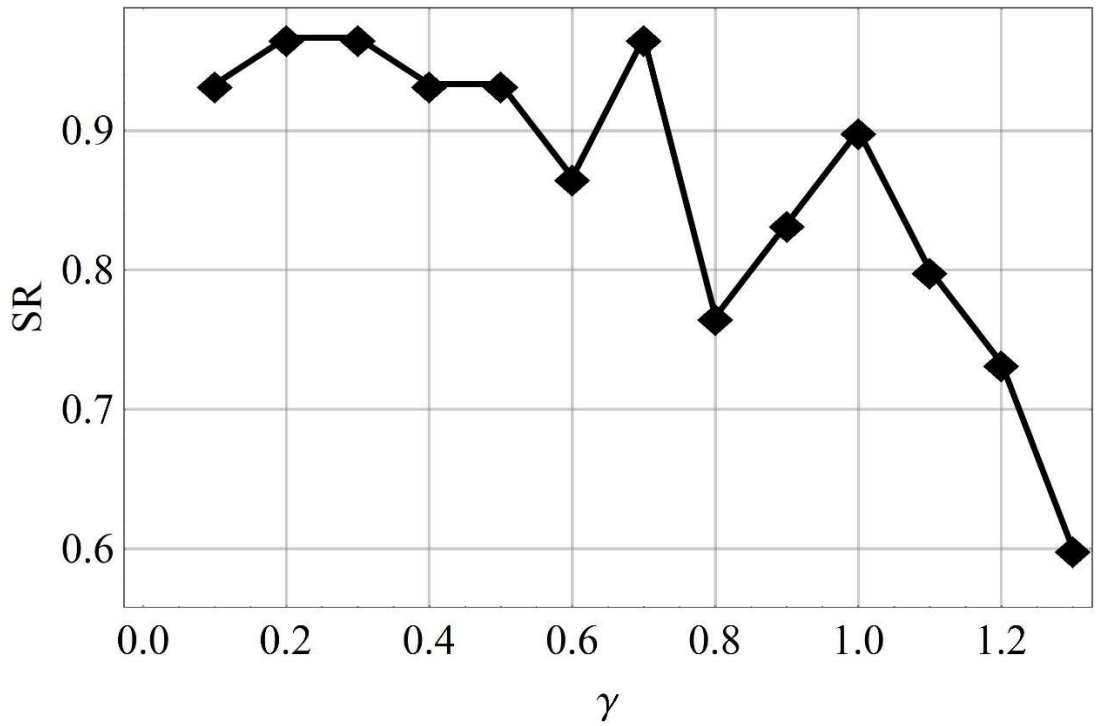


(a) Relationship between *t* and SR

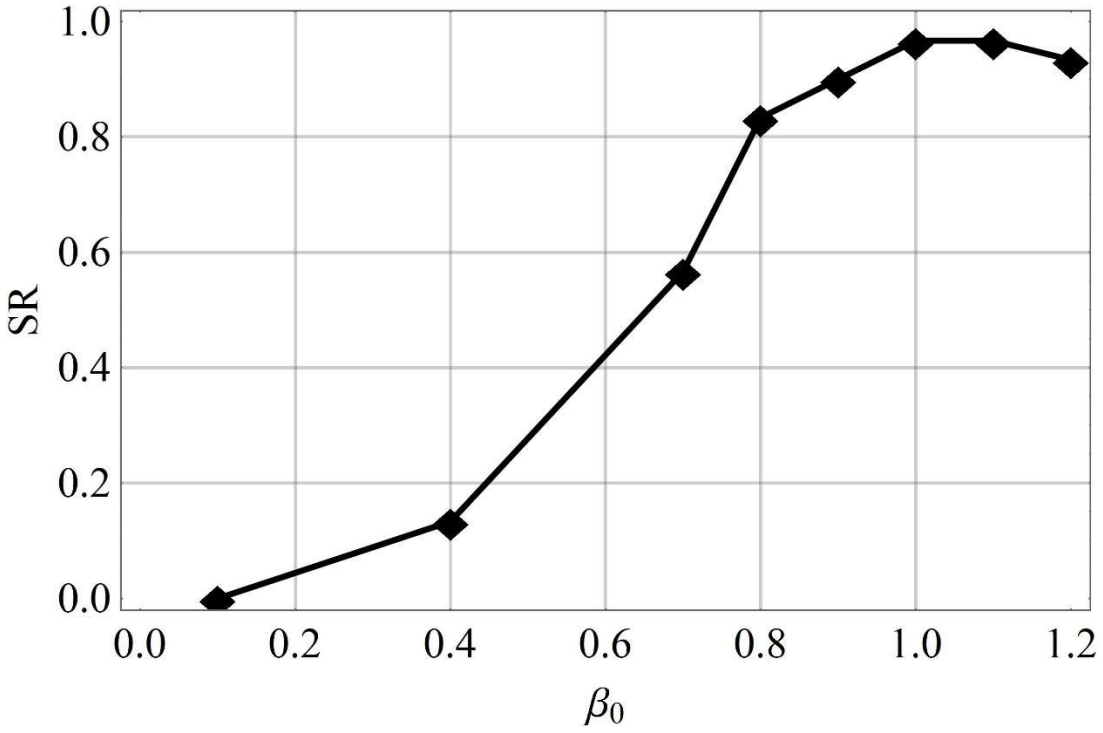


(b) Relationship between *limit* and SR

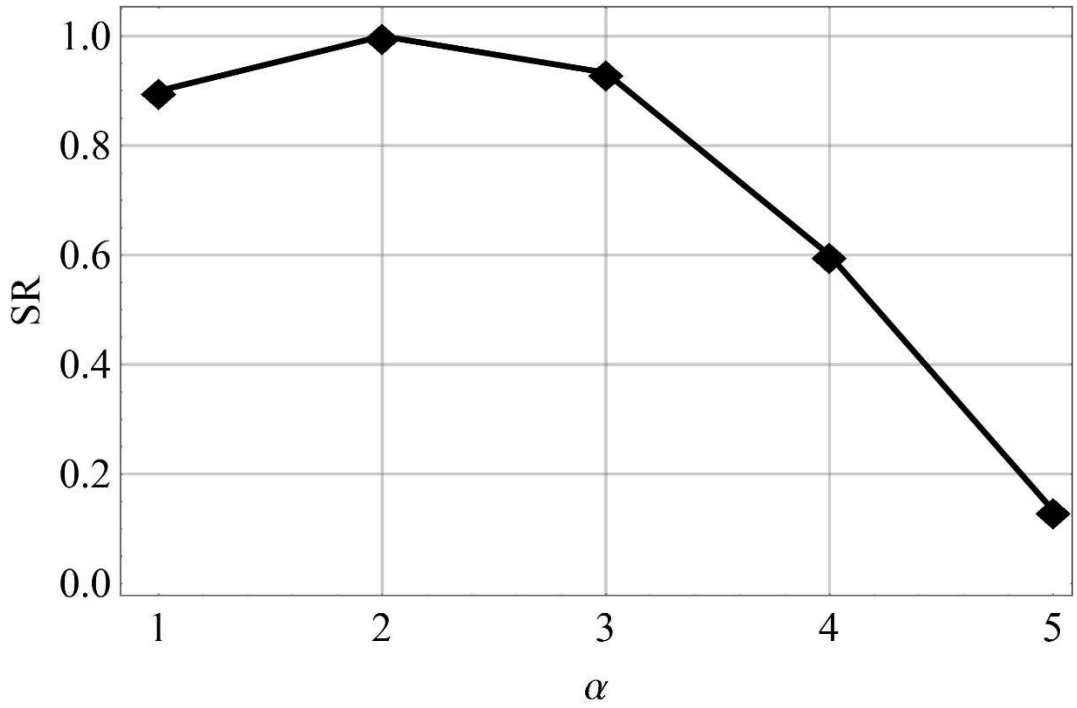
Fig. 3.8 Parameter optimization of S-ABC



(a) Relationship between γ and SR



(b) Relationship between β_0 and SR



(c) Relationship between α and SR

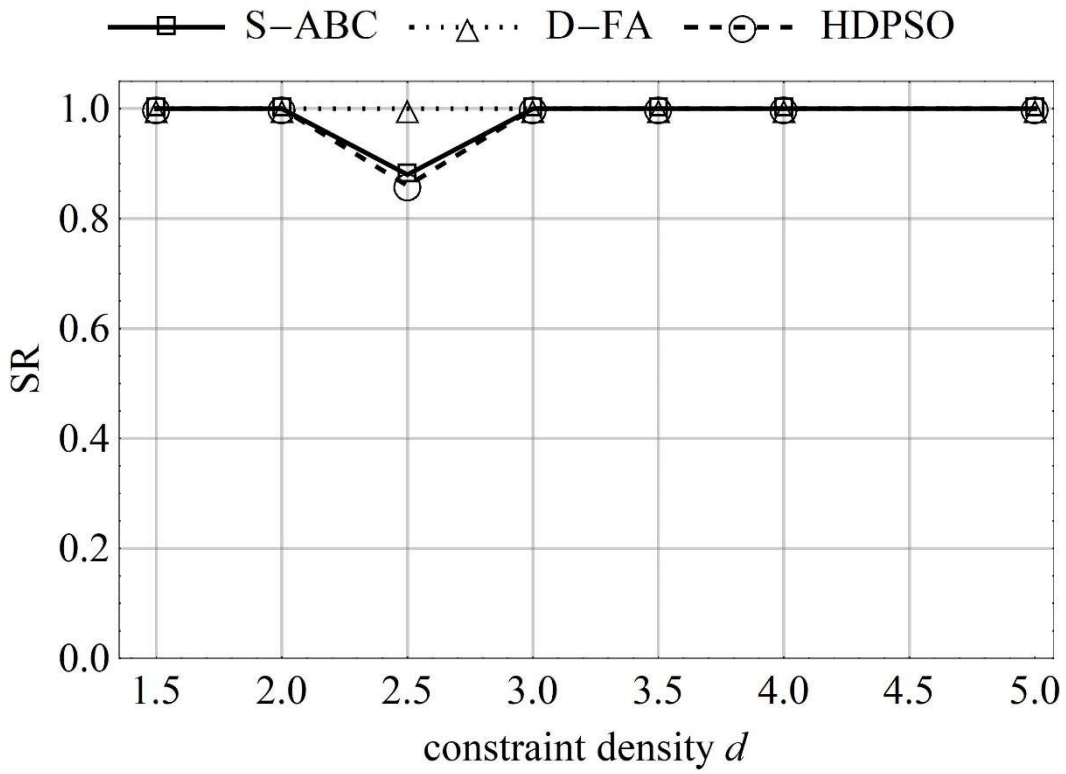
Fig. 3.9 Parameter optimization of D-FA

3.4.3 Comparison Study

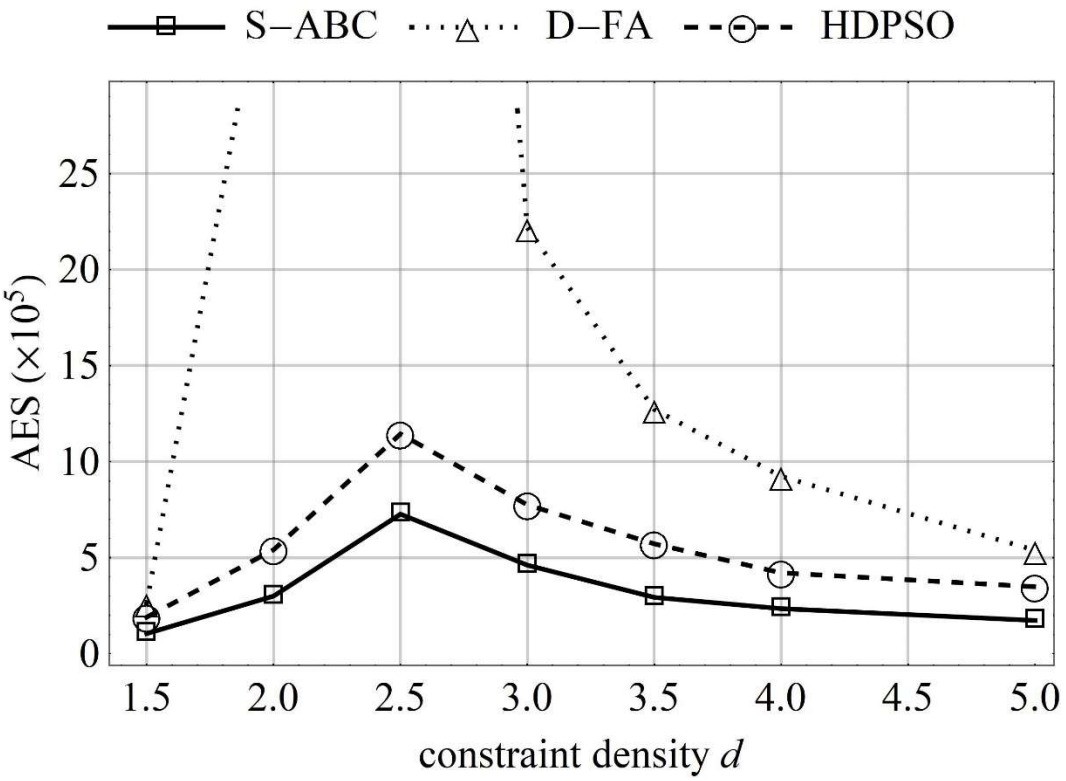
Using the optimal hyper-parameters, the proposed algorithms are compared with HDPSO on 3-colorable Minton random graphs with various *n* and *d*. The experiment settings are listed in Table 3.4, and the values of w_0, c_1, and c_2 of HDPSO are taken from [Aoki 15] directly. For the sake of fairness, we set the swarm size 200 and maximum generations 10000 for all three algorithms. Candidate algorithms are tested on 50 graph instances for each *d*. The results are shown from Figure 3.10 to Figure 3.12.

Table 3.4 Experiment settings for comparison study

	S-ABC	D-FA	HDPSO
Swarm size	200	200	200
Max generations	10000	10000	10000
Parameters	$t = 2$ *limit* = 80	$\beta_0 = 1.0,\ \gamma = 0.2,$ $\alpha = 2$	$w_0 = 0.05\ \ c_1 = 7.0$ $c_2 = 0.03$

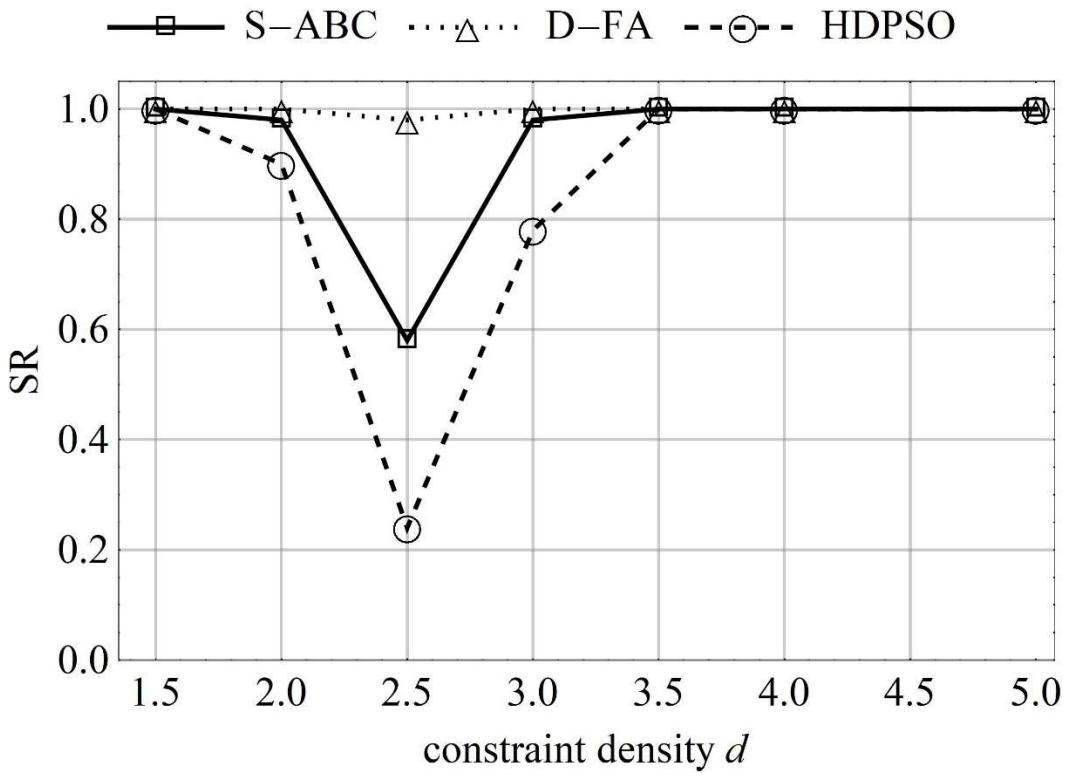


(a) SR on various d (n = 90)

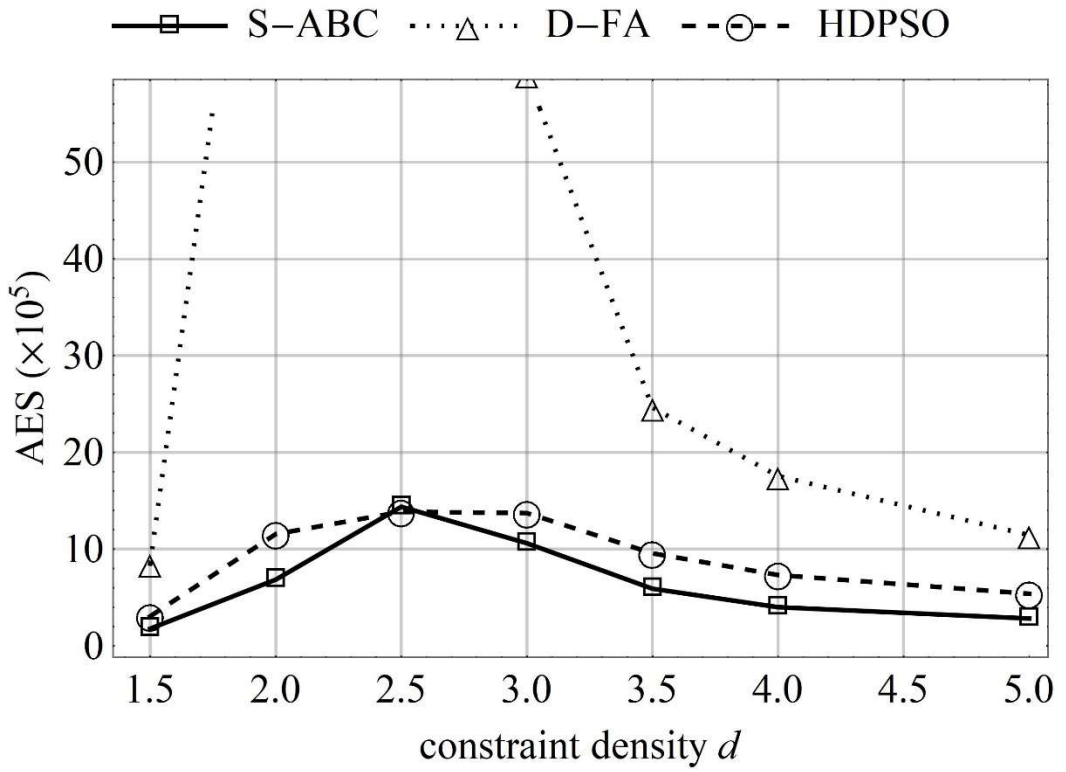


(a) AES on various d (n = 90)

Fig. 3.10 SR and AES on various d (n = 90)

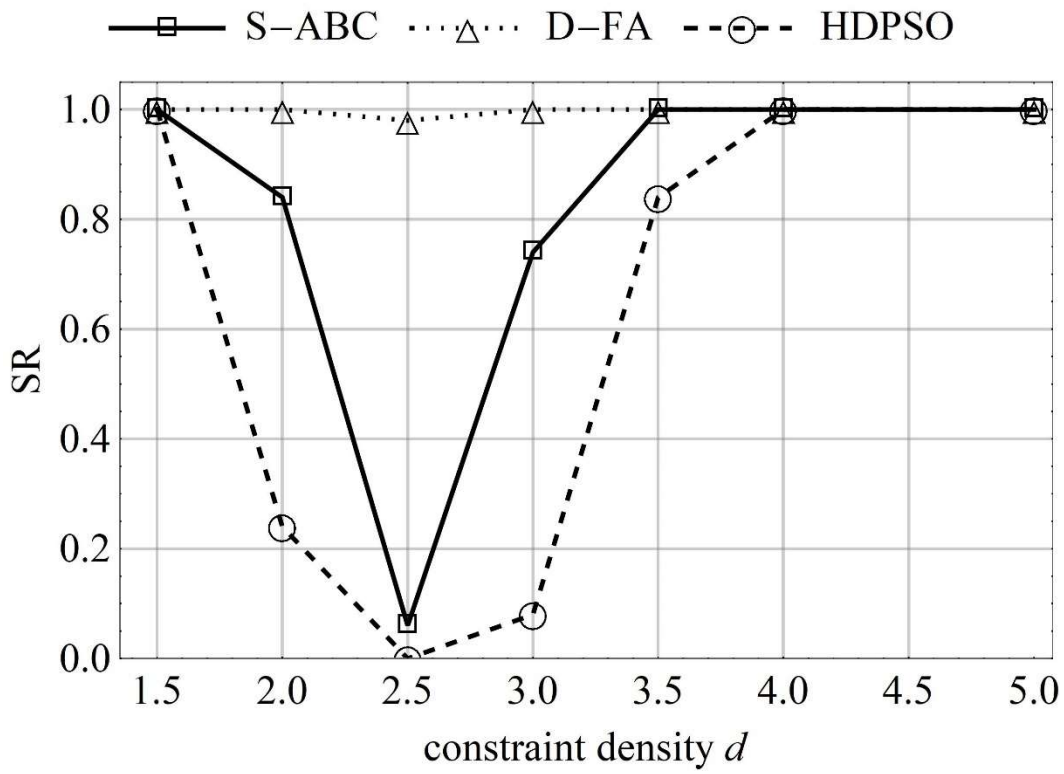


(a) SR on various d (n = 120)

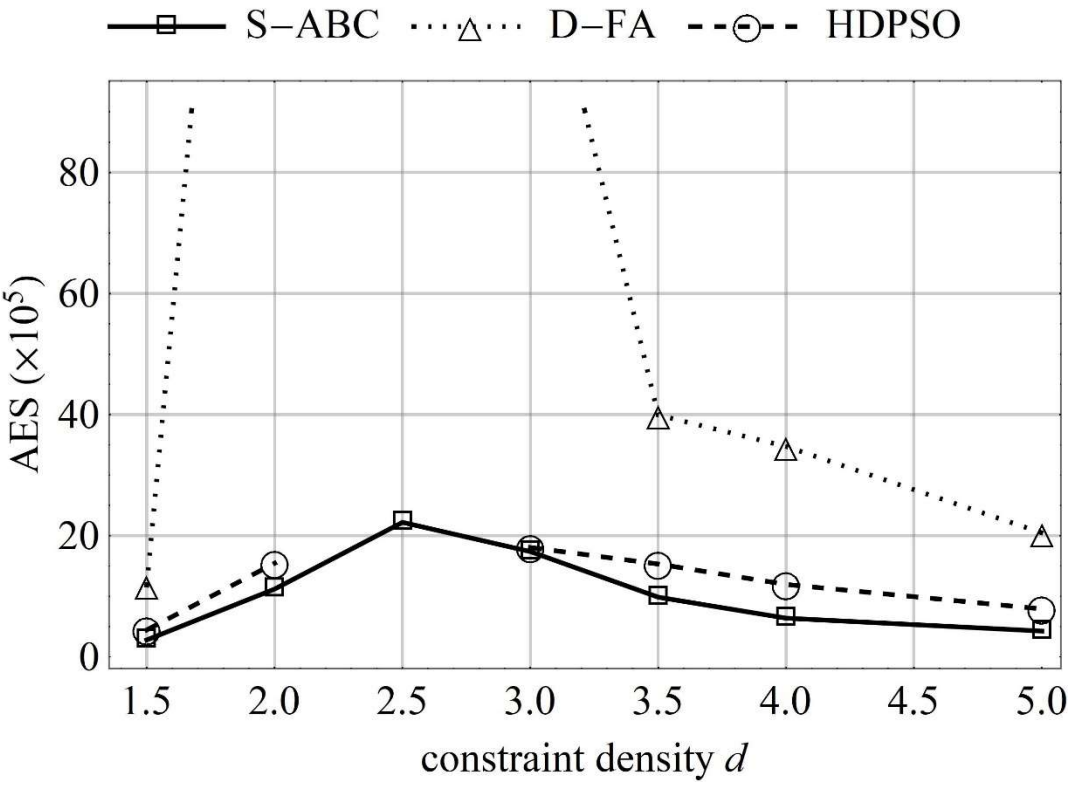


(a) AES on various d (n = 120)

Fig. 3.11 SR and AES on various d (n = 120)



(a) SR on various d (n = 150)



(a) AES on various d (n = 150)

Fig. 3.12 SR and AES on various d (n = 150)

■ Discussion

[**S-ABC**]

From Figure 3.10 (a), 3.11 (a) and 3.12 (a), we can see that SR of S-ABC is much higher than its competitor HDPSO when constraint density d is in the range of phase transition, i.e. when d = 2.5. With the graph size increasing, S-ABC retains relatively high SR while SR of HDPSO becomes poor.

On the other hand, as Figure 3.10(b), 3.11(b), and 3.12(b) show, AES of S-ABC is lower than HDPSO dramatically in most cases, which demonstrates S-ABC can find a solution in a relatively small evaluation time.

Taking into account of SR and AES both, we can say that S-ABC is effective and outperforms HDPSO.

[**D-FA**]

At the first glance, SR of D-FA on all graphs, especially when d = 2.5, is much higher than SABC and HDPSO (see Figure 3.10(a), 3.11(a), and 3.12(a)). However, it is not because D-FA is an efficient algorithm but because D-FA has a 'double-loop' structure so that it evaluates objective function much more times than its competitors in one generation.

By checking Figure 3.10(b), 3.11(b), and 3.12(b), one may find that even though D-FA performs much better than the other two on the aspect of SR when d = 2.5, its AES is much higher than S-ABC and HDPSO. This means that DFA consumes more evaluation times to find a solution than the other two methods.

We infer that if the termination condition is not 'maximum generation' but 'maximum evaluation times of objective function', D-FA may not be so efficient. This can be proved in next Chapter.

[**Phase transition**]

A very interesting thing we can find from the experiments is that the experiment value of phase transition is '*more accurate*' than the theory value for 3-colorable Minton random graphs. For example, the theory value of d that generates the hardest problem is 3.1 while the experiment value is 2.5 according to Table 2.2. From our experiments, we can see that the graphs generated by d = 2.5 is much harder than those generated by d = 3.

3.5 Summary

In this chapter, by defining similarity to describe the distance between two candidates, we proposed S-ABC and D-FA to solve 3-GCPs on 3-colorable Minton random graphs. Using HDPSO as a baseline, we compared the three algorithms on Minton random graphs with various graph size n and constraint density d. Considering SR and AES both, we found that S-ABC is an efficient algorithm because of its high SR and low AES.

Chapter 4 Adaptive Artificial Bee Colony Algorithm

In Chapter 3, we proposed an efficient algorithm S-ABC to solve 3-GCPs. However, S-ABC updates a candidate by a predetermined parameter *t*, which does not change during the evolution no matter how good or bad a candidate is. In this chapter, we propose a discrete adaptive artificial bee colony (A-ABC) algorithm that can adjust *t* automatically according to the graph size and the fitness of a candidate. We also study the scout bee phase and report that scout bee phase is not required in solving discrete optimization problems.

4.1 Proposed Method

4.1.1 Update Rule of A-ABC

We use the encoding scheme defined in section 3.1 to establish our algorithm. A-ABC updates a candidate by modeling equation (2.2). When a candidate $\mathbf{x}_i$ is selected, A-ABC first saves it to a temporary candidate $\mathbf{v}_i$ and then randomly selects a neighbor $\mathbf{x}_j$. Unlike S-ABC, which accepts a candidate to be a neighbor $\mathbf{x}_j$ only if $\mathbf{x}_j$ is similar to $\mathbf{x}_i$, in A-ABC, any candidate in the swarm has the equal chance to be selected, which is exactly what the original ABC does.

Once a candidate $\mathbf{x}_j$ is determined, *t* elements in $\mathbf{v}_i$ are replaced by the corresponding elements of $\mathbf{x}_j$. Let $\mathrm{T} = \{l_1, l_2, .., l_t\}$ be the index set of the selected elements. The replacement process is defined by equation (4.1)

$$v_{il} = \begin{cases} x_{jl}, & v_{il} \neq x_{jl} \\ r, & v_{il} = x_{jl} \end{cases} \tag{4.1}$$

where $l \in \mathrm{T}$.

Equation (4.1) states that the replacement arises only if the values (or more accurate, the colors) of the two elements are different. If the corresponding elements have the same color, the change is meaningless. In this case, v_{il} is replaced by a random color *r* from the color set *C* and $v_{il} \neq r$.

Then, $\mathbf{v}_i$ will be evaluated by the objective function. If its fitness is better than $\mathbf{x}_i$, $\mathbf{x}_i$ will be replaced by $\mathbf{v}_i$.

4.1.2 Introducing Adaptive Function

Obviously, the parameter *t* affects the performance of A-ABC significantly. It can be optimized by some preliminary experiments and keeps static during the searching process. However, using a fixed *t* does not consider the current environment (or state) such as the fitness of candidates, the graph size, and so on).

To solve the problem, we define an adaptive function to determine *t*:

$$t(\mathbf{x}_i) = \left\lceil n \left(\frac{con(\mathbf{x}_i)}{m}\right)^u \right\rceil \tag{4.2}$$

where $\lceil x \rceil$ is the least upper bound of *x* (for example, $\lceil 3.2 \rceil = 4$) and *u* is a positive integer scaling factor.

Equation (4.2) states that the value of *t* is proportional to the total conflict of a candidate $\mathbf{x}_i$. This is feasible because we update a candidate with high conflict by using large *t* so that it searches widely while using small *t* for the candidates with low conflict by tuning them locally.

4.1.3 Employed Bee Phase and Onlooker Bee Phase.

Employed bee phase and onlooker bee phase use equation (4.1) to improve candidates. Like original ABC does, employed bee phase tries to improve all candidates while onlooker bee phase exploits the candidates having high fitness. The former can be regarded as global search and the latter is local search respectively. The employed bee phase and the onlooker bee phase are shown in Figure 4.1.

4.1.4 Scout bee phase

In original scout bee phase, a candidate $\mathbf{x}_i$ is replaced by a randomly generated new candidate if $\mathbf{x}_i$ cannot be improved further in *limit* trials of evaluating the objective function. The limit in A-ABC is defined by equation (4.3).

$$limit = c\frac{MaxEval}{2N} \tag{4.3}$$

where *MaxEval* is the maximum evaluation times of objective function, *N* is the swarm size, and $\frac{MaxEval}{2N}$ is the maximum generation because the objective function is calculated $2N$ times for one generation. Parameter *c* is a real value factor used to control the value of *limit*. As original ABC, there is only one scout bee in S-ABC. Thus, only one candidate may be abandoned for each generation.

Given a swarm *S* with *N* candidates.

[**Employed bee phase**]

Employed bee phase (*S*)
    for *i* = 1 to *N*
        determine *t* for $\mathbf{x}_i$ using equation (4.2)
        update $\mathbf{x}_i$ using equation (4.1) and get $\mathbf{v}_i$
        calculate the fitness of the new $\mathbf{v}_i$
        if $(fitness(\mathbf{v}_i) \geq fitness(\mathbf{x}_i))$
            replace $\mathbf{x}_i$ by $\mathbf{v}_i$
        end if
    end for

[**Onlooker bee phase**]

Onlooker bee phase (*S*)
    for *i* = 1 to *N*
        choose $\mathbf{x}$ from *S* by roulette selection
        determine *t* for $\mathbf{x}$ using equation (4.1) and get $\mathbf{v}$
        calculate the fitness of $\mathbf{v}$
        if $(fitness(\mathbf{v}) \geq fitness(\mathbf{x}))$
            replace $\mathbf{x}$ by $\mathbf{v}$
        end if
    end for

Fig. 4.1 Frameworks of employed bee phase and onlooker bee phase

4.1.5 Framework of A-ABC

The framework of A-ABC is similar to original ABC. Combining the three phases defined above, the framework of A-ABC is given in Figure 4.2. Note that the scout bee phase is optional. We will study scout bee phase in next section.

```
Initialize randomly a swarm S with N candidates
Set parameters u and c

while (termination condition is not met)
    Employed bee phase (S)
    Onlooker bee phase (S)

    if (scout bee phase is used)
        Scout bee phase (S, c)
    end if
end while
```

Fig. 4.2 Framework of A-ABC

4.2 Experiments

4.2.1 Experiment Design

In this section, we design several experiments to show the effect, robustness, and generality of A-ABC. We first compare A-ABC with various u and with fixed t to show the effect of adaptive function (4.2). Then we study the performance of scout bee phase in A-ABC and show that the scout bee phase is not required when solving GCPs. Next, we test five algorithms: A-ABC, S-ABC, D-FA, HDPSO, and DCS using large evaluation times. Let the five algorithms evolve sufficiently, we study the convergence region of each algorithm and show that A-ABC is a fast and efficient algorithm. After that, we compare the five algorithms on 3-colorable Culberson random graphs with various graph size n and edge probability p to show the robustness of A-ABC. Finally, to show the generality of A-ABC, we compare the five algorithms on 4-colorable and 5-colorable Culberson random graphs.

We test algorithms on three classes of Culberson random graphs (i.e. arbitrary-random graph, equi-partite graph, and flat graph). As discussed in 2.2.3, flat graph is much harder than the other two when given the same graph size n and edge probability p.

We compare the performance of algorithms on three measures: (1) success rate (SR), (2) average evaluation times of objective function to find a solution (AES), and (3) the standard deviation of evaluation times of objective function to find a solution (SD). SR and AES have been defined in section 3.4.1 (see equation (3.7) and (3.8)). SD is defined as:

$$SD = \sqrt{\frac{1}{N_S - 1}\sum_{i=1}^{N_S}(e_i - AES)^2} \tag{4.4}$$

where N_S is the number of success instances and e_i is the evaluation times of objective function of the i^{th} success instance.

We list the scheme for performing the experiments in Table 4.1.

4.2.2 Effect of Adaptive Function

In 4.1.2, adaptive function (4.2) is defined to calculate the value of *t*, which represents the number of replaced nodes. Given a random graph with *n* nodes and *m* edges, $con(\mathbf{x}_i)$ can be determined by equation (2.8) directly. Thus, the value of *t* is affected by the integer scaling factor *u*. By adjusting *u*, *t* can adapt to graphs with various sizes and to candidates with different fitness.

To show the effect of the adaptive function, two algorithms are compared on 3-colorable Culberson random graphs: (1) A-ABC with various *u*, and (2) ABC with fixed *t*. The only difference is that the former uses adaptive function to tune *t*, but the latter uses fixed *t* during the evolution.

Table 4.1 The scheme of experiments

No	Experiment	GCPs	Purpose
1	Compare various *u* and *t*	3-colorable Culberson random graphs	Show the effect of adaptive function
2	Effect of scout bee phase		Show that the scout bee phase is not required
3	Compare the evoluti on scheme		Show the convergence region of A-ABC
4	Large comparison		Show the effect and robustness of A-ABC
5	Study the generality	4-colorable and 5-colorable Culberso n random graphs	Show the generality of A-ABC

We test the two algorithms on 3-colorable Culberson random graphs with n = 120 and p = 0.058. As discussed in Table 2.2, the most difficult 3-colorable graphs are generated when $\frac{7}{n} \leq p \leq \frac{8}{n}$. For a graph with n = 120, p = 0.058 generates the hard graph instances.

The experiment settings are given in Table 4.2. The swarm size is 200 and max evaluation times of objective function, defined as *MaxEval*, is 1,200,000. Parameters u and t are from 1 to 5. Scout bee phase is not used in this experiment and will be studied in the next section. The results are list from Table 4.3 to 4.5.

Table 4.2 The experiment settings of testing the effect of adaptive function

	Using adaptive function	Using fixed t
N	200	200
Parameters	u = 1 to 5	t = 1 to 5
MaxEval	1,200,000	1,200,000
Graphs	3-colorable Culberson random graphs (Arbitrary random graph, equi-partite graph, and flat graph) n = 120 and p = 0.058	
Number of graph instances	30	
Others	Scout bee phase is not used	

Table 4.3 Various *u* and *t* on 3-colorable arbitrary random graphs

		SR	AES ($\times 10^5$)	SD ($\times 10^5$)
Using adaptive function	$u = 1$	0.07	7.93	3.28
	$u = 2$	0.70	4.78	3.13
	$u = 3$	0.57	6.38	3.00
	$u = 4$	0.67	5.46	2.67
	$u = 5$	0.50	4.83	2.55
Using fixed *t*	$t = 1$	0.60	4.98	2.91
	$t = 2$	0.50	4.25	2.61
	$t = 3$	0.27	7.07	3.54
	$t = 4$	0.03	7.86	----
	$t = 5$	0.00	----	----

Table 4.4 Various *u* and *t* on 3-colorable equi-partite graphs

		SR	AES ($\times 10^5$)	SD ($\times 10^5$)
Using adaptive function	$u = 1$	0.00	----	----
	$u = 2$	0.67	4.67	2.96
	$u = 3$	0.63	6.51	3.03
	$u = 4$	0.60	5.59	2.87
	$u = 5$	0.53	4.99	1.79
Using fixed *t*	$t = 1$	0.60	6.18	3.43
	$t = 2$	0.63	6.74	3.50
	$t = 3$	0.23	5.83	2.77
	$t = 4$	0.03	10.93	----
	$t = 5$	0.00	----	----

Table 4.5 Various *u* and *t* on 3-colorable flat graphs

		SR	AES ($\times 10^5$)	SD ($\times 10^5$)
Using adaptive function	$u = 1$	0.00	----	----
	$u = 2$	0.26	4.78	2.21
	$u = 3$	0.23	7.11	2.45
	$u = 4$	0.26	5.88	2.94
	$u = 5$	0.10	4.03	4.71
Using fixed *t*	$t = 1$	0.23	6.63	1.99
	$t = 2$	0.10	4.57	2.79
	$t = 3$	0.03	6.71	----
	$t = 4$	0.00	----	----
	$t = 5$	0.00	----	----

■ Discussion

A well designed adaptive function should set *t* large for candidates with low fitness and, on the contrary, small for those with high fitness. From Table 4.3 to 4.5, we find that in most cases, SR of using adaptive function with $u = 2$, 3, or 4 is much higher than that of using fixed *t*. On the other hand, SRs of A-ABC are very poor when $u = 1$, i.e. the adaptive function is linear. This is because linear function gives large *t* even if a candidate has high fitness. We also find that if $u > 4$, most values of *t* given by adaptive function are 1 or 2, thus the SRs get worse and approach to SRs of using fixed $t = 1$ or $t = 2$. The experiment shows that A-ABC with quadratic adaptive function, i.e., $u = 2$ obtains the highest SR.

In summary, by adopting adaptive function, SR is improved remarkably because the value of *t* is tuned according to the candidates' fitness during the evolution. For a candidate that has low fitness, large *t* is used so that it searches globally while for a candidate with high fitness, small *t* is adopted so that it searches locally.

4.2.3 Performance of Scout Bee Phase

As described above, scout bee phase removes a candidate that cannot be improved in *limit* trials and replaces it with a new generated candidate. In A-ABC, as equation (4.3) shows, *limit* is controlled by swarm size *N* and a scaling factor *c*. By setting different values of *c* and *N*, we design some experiments to study the effect of scout bee phase in A-ABC.

To make our discussion clear, besides SR, AES, and SD, we introduce other two measures, AAS and AAF, in our experiments. AAS and AAF represent *A*verage number of candidates *A*bandoned by scout bee phase in *S*uccessful graph instances and in *F*ailed graph instances respectively. More formally, AAS and AAF are defined as follows:

$$AAS = \frac{\sum_{i=1}^{N_S} BS_i}{N_S} \tag{4.5}$$

$$AAF = \frac{\sum_{i=1}^{N_F} BF_i}{N_F} \tag{4.6}$$

where N_S and N_F are the numbers of successful and failed graph instances respectively, BS_i and BF_i are the numbers of abandoned candidates of the i^{th} successful and failed graph instances.

Because *limit* is dependent on two parameters *c* and *N*, we design two experiments to test the performance of scout bee phase by using various *c* and *N*. First, we try various *c* and observe SR, AAS, and AAF. The experiment settings are given in Table 4.6.

Table 4.6 The experiment settings of testing the effect of scout bee phase on various *c*

N	200
c	0.3 ~ 20.0
u	2
MaxEval	1,200,000
Graphs	3-colorable Culberson random graphs (Arbitrary random graph, equi-partite graph, and flat graph) n = 120 and p = 0.058
The number of graph instances	30

The results on three classes of Culberson random graphs are shown from Table 4.7 to 4.9.

Table 4.7 The effect of scout bee phase with various *c* on arbitrary-random graphs

c	SR	AES ($\times 10^5$)	SD ($\times 10^5$)	AAS	AAF
0.3	0.20	5.93	2.18	108	296
0.5	0.53	4.59	2.85	42	205
1.0	0.37	7.38	3.10	30	71
2.0	0.50	5.75	3.44	9	19
3.0	0.60	5.81	2.50	6	10
5.0	0.57	5.89	2.90	3	6
10.0	0.60	5.49	2.88	1	3
20.0	0.63	5.63	2.92	1	1
No scout	0.70	4.78	3.13	0	0

Table 4.8 The effect of scout bee phase with various *c* on equi-partite graphs

c	SR	AES ($\times 10^5$)	SD ($\times 10^5$)	AAS	AAF
0.3	0.23	6.50	3.68	136	298
0.5	0.27	5.92	2.11	50	207
1.0	0.27	4.29	2.30	14	68
2.0	0.53	6.06	3.73	9	18
3.0	0.60	4.57	2.80	5	11
5.0	0.60	5.21	3.07	3	6
10.0	0.60	5.56	2.03	2	3
20.0	0.67	4.18	2.74	0	1
No scout	0.67	4.67	2.96	0	0

Table 4.9 The effect of scout bee phase with various *c* on flat graphs

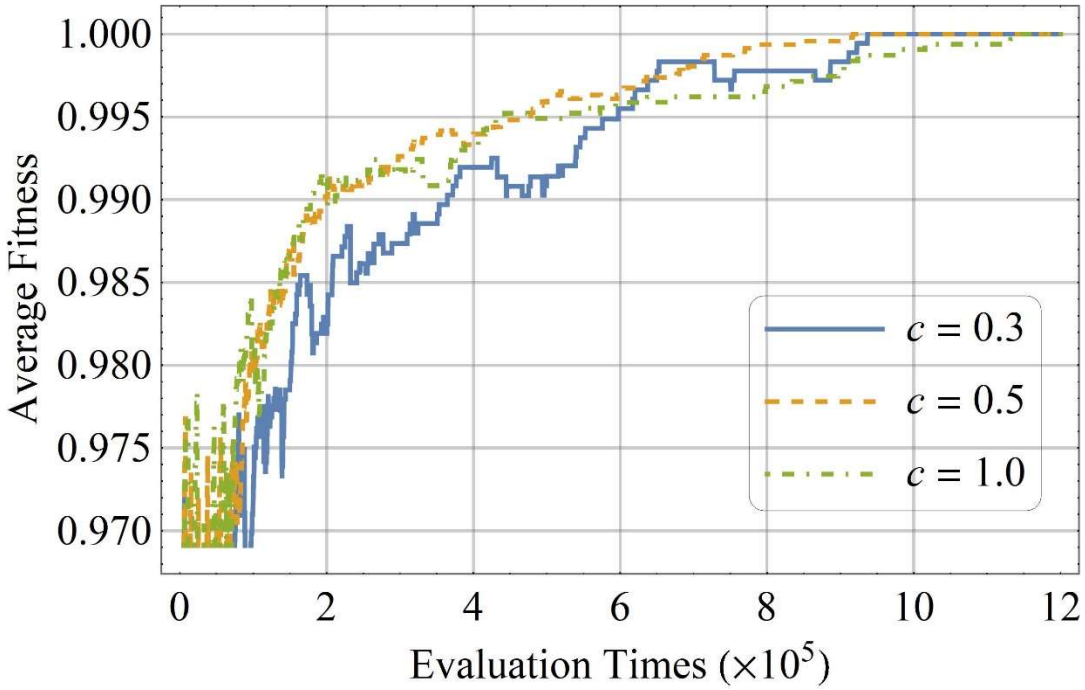
c	SR	AES ($\times 10^5$)	SD ($\times 10^5$)	AAS	AAF
0.3	0.23	7.57	3.25	159	298
0.5	0.10	6.94	2.43	69	209
1.0	0.30	7.53	2.03	26	70
2.0	0.23	4.70	2.97	8	17
3.0	0.33	5.04	1.88	6	10
5.0	0.17	5.22	2.01	3	6
10.0	0.17	7.36	3.35	2	3
20.0	0.20	4.60	1.40	0	1
No scout	0.27	4.78	2.21	0	0

■ Discussion

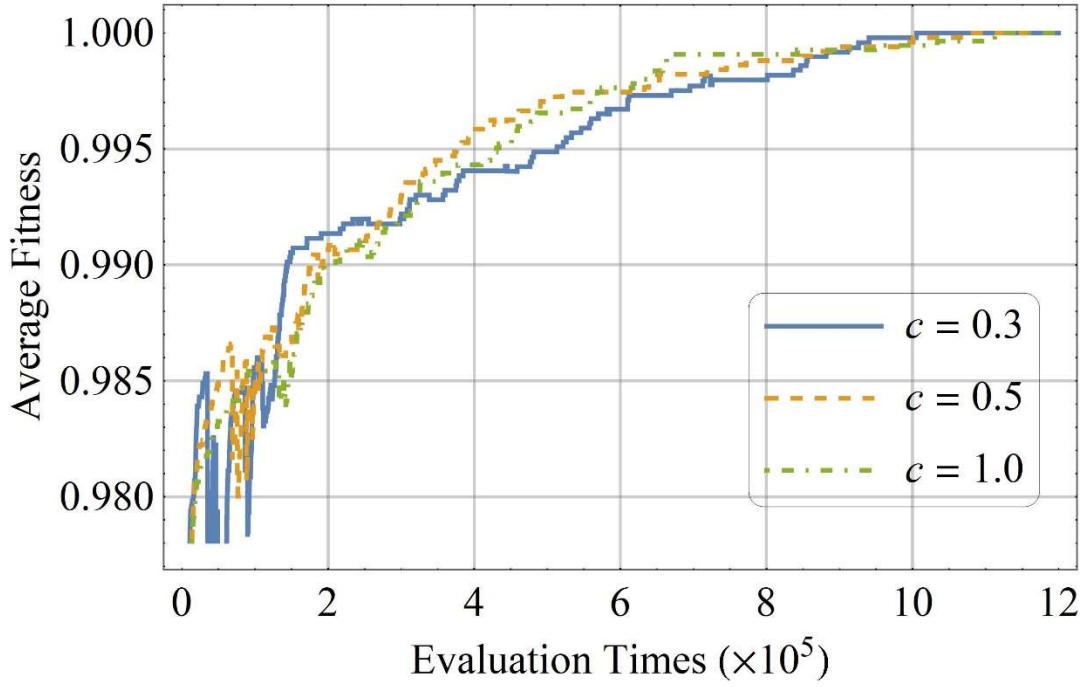
The number of candidates and fitness values of discrete optimization problems is quite different from that of continuous optimization problems. In continuous domain, the number of candidates is infinite. Correspondingly, there are infinite fitness values and frequently, a small fluctuation of a candidate may lead to a large modification of its fitness value. Thus, in the context of swarm intelligence, it is reasonable to claim that a candidate is trapped in the local optimum if its fitness value cannot be improved for several generations.

On the other hand, in discrete domain, fitness values are restricted to some finite values. As a result, many candidates may share the same fitness value. For example, given a random graph with 120 nodes and 266 edges, there are only 267 possible fitness values. In onlooker bee phase, roulette selection is used to select the candidates with high fitness, thus if *c* is small (i.e. *limit* is small), the selected candidates are often removed by scout bee phase and this leads to a sudden decline of the global best fitness value. This is the reason that in most cases, small *c* gives poor SR but large AAS and AAF in Table 4.7 to 4.9.

We can clearly observe this phenomenon by drawing the evolution Figures. Here, we plot the average fitness curves of successful arbitrary random graph instances to evaluation times for various *c*. Figure 4.3 shows that (1) small *c* results in fitness value fluctuating violently because the candidates with high fitness values are frequently removed from the swarm (Figure 4.3 (a)); (2) large *c* obtains relatively stable improvement of fitness value, but the scout bee phase is rarely used (Figure 4.3 (b)).



(a) The average fitness curves of successful graph instances with small *c*.



(b) The average fitness curves of successful graph instances with large *c*.

Fig. 4.3 The average fitness curves of successful arbitrary random graph instances with various *c*.

As mentioned above, *limit* is controlled by *c* and *N*. Thus, we use different *N* and various *c* to study the influence of *N* on scout bee phase. The experiment settings are given in Table 4.10 and the results are shown in Table 4.11 and 4.12.

Table 4.10 The experiment settings of testing the effect of scout bee phase on various *N* and *c*

N	100 and 300
c	0.3 ~ 20.0
u	2
MaxEval	1,200,000
Graphs	3-colorable equi-partite random graphs $n = 120$ and $p = 0.058$
The number of graph instances	30

Table 4.11 The effect of scout bee phase with various *c* on equi-partite graphs (*N* = 100)

c	SR	AES ($\times 10^5$)	SD ($\times 10^5$)	AAS	AAF
0.3	0.43	5.05	3.76	68	204
0.5	0.37	5.24	3.33	39	128
1.0	0.57	6.39	3.05	18	63
2.0	0.63	4.54	2.77	4	14
3.0	0.67	4.13	2.63	2	8
5.0	0.57	4.62	2.60	2	4
10.0	0.53	4.98	3.25	1	2
20.0	0.73	4.26	2.47	0	0
No scout	0.53	4.15	2.18	0	0

Table 4.12 The effect of scout bee phase with various *c* on equi-partite graphs (*N* = 300)

c	SR	AES ($\times 10^5$)	SD ($\times 10^5$)	AAS	AAF
0.3	0.27	6.37	2.84	165	383
0.5	0.33	5.62	2.72	66	249
1.0	0.40	5.85	4.16	28	74
2.0	0.47	6.19	3.92	14	25
3.0	0.43	6.76	3.16	10	15
5.0	0.50	6.94	3.27	6	9
10.0	0.43	5.89	3.30	2	4
20.0	0.5	7.01	2.84	1	2
No scout	0.73	7.68	2.65	0	0

■ Discussion

From Table 4.11 and 4.12, we observe that no matter *N* is mall or large, the small *c* gives poor SR but large AAS and AAF while the large *c* gives high SR but small AAS and AAF. This is identical to the phenomenon shown in Table 4.7, 4.8, and 4.9. We find that it is not *N* but *c* that has the crucial influence on SR because SRs get worse if scout bee phase in invoked frequently.

In summary, scout bee phase plays a major role in avoiding the local optimum in continuous optimization problems but is not so efficient when solving discrete optimization problems because the fitness values are limited in a small range. Thus, we suggest either replacing scout bee phase with other methods or, as this work does, simply removing it from A-ABC.

4.2.4 Convergence Region of A-ABC

In above two experiments, we use a relatively small value of *MaxEval*, and it seems that A-ABC can find a solution quickly in such small evaluation times. It would be interesting to compare the performance of A-ABC with other algorithms when a large value of *MaxEval* is given so that all candidate algorithms can evolve sufficiently. In this experiment, we compare five algorithms: A-ABC, Discrete Cuckoo Search (DCS), S-ABC, HDPSO, and D-FA. The experiment settings are given in Table 4.13. Note that we set *MaxEval* to 10,000,000, which is a large value and the five candidate algorithms can evolve sufficiently.

Table 4.13 The experiment settings of testing the convergence region

A-ABC	$N = 200$, $u = 2$, scout bee phase is not used
DCS	$N = 10$, $\alpha = 1.0$, $\beta = 1.5$, $p_a = 0.0001$
S-ABC	$N = 200$, $t = 2$, $limit = 80$
HDPSO	$N = 10$, $w_0 = 0.05$, $c_1 = 7.0$, $c_2 = 0.03$
D-FA	$N = 200$, $\alpha = 2$, $\beta_0 = 1.0$, $\gamma = 0.2$
MaxEval	10,000,000
Graph	3-colorable Culberson random graphs (Arbitrary random graph, equi-partite graph, and flat graph) $n = 120$ and $p = 0.058$
The number of graph instances	30

As mentioned in section 2.3.4, [Aranha 17] gives three different methods to calculate *M* and proposes four discrete cuckoo search algorithms by using various combinations of the three methods. In this experiment, we accept the one obtaining the highest SR as the candidate algorithm. The framework of the best DCS is give in Figure 2.12.

The results are given in Table 4.14, 4.15, and 4.16. For convenience, the results are ranked in descending order of SR.

Table 4.14 Comparison with five algorithms on arbitrary random graphs

Algorithm	SR	AES ($\times 10^5$)	SD ($\times 10^5$)
DCS	1.00	16.11	18.70
A-ABC	0.80	11.35	12.60
S-ABC	0.73	18.46	11.69
HDPSO	0.47	27.02	29.44
D-FA	0.20	60.08	20.60

Table 4.15 Comparison with five algorithms on equi-partite graphs

Algorithm	SR	AES ($\times 10^5$)	SD ($\times 10^5$)
DCS	1.00	11.08	9.27
A-ABC	0.90	13.69	15.36
S-ABC	0.80	26.98	19.55
HDPSO	0.40	21.11	22.80
D-FA	0.37	53.94	25.12

Table 4.16 Comparison with five algorithms on flat graphs

Algorithm	SR	AES ($\times 10^5$)	SD ($\times 10^5$)
DCS	0.77	28.24	25.61
A-ABC	0.50	23.67	22.49
S-ABC	0.40	50.30	26.61
HDPSO	0.20	16.39	13.77
D-FA	0.07	51.77	0.17

■ Discussion

Table 4.14, 4.15, and 4.16 show that DCS obtain the highest SR in all candidate algorithms and A-ABC is the second-best one. Compared with S-ABC, HDPSO, and D-FA, A-ABC obtains better SR and lower AES and SD on most cases of graphs.

To make our discussion more persuasive, we perform χ^2 proportion test to show that the five algorithms have significant difference on SR. The steps of performing χ^2 proportion test on SR in Table 4.14 (i.e. arbitrary random graphs) are given as follows.

Target: to show whether the proportion of success times to failed times is equal or not.

Step 1. Target parameters and hypothesis.

Let p_{DCS}, p_{A-ABC}, p_{S-ABC}, p_{HDPSO}, and p_{DFA} be proportion of success times to failed times of each algorithm. For example, in Table 4.14, in which the experiment is performed on 30 instances of arbitrary random graphs,

$$p_{A-ABC} = \frac{30 \times 0.8}{30 \times (1 - 0.8)} = 4 \tag{4.7}$$

Based on this definition, null hypothesis $\mathcal{H}_0$ and alternative hypothesis $\mathcal{H}_a$ are:
$\mathcal{H}_0$: p_{DCS}, p_{A-ABC}, p_{S-ABC}, p_{HDPSO}, and p_{DFA} are equal;
$\mathcal{H}_a$: p_{DCS}, p_{A-AB} , p_{S-ABC}, p_{HDPSO}, and p_{DFA} are not equal.

Step 2. Observed values and expectation values of success times and failed times.

Observed values and expectation values of success times and failed times can be easily calculated from Table 4.14. They are summarized in *Contingency* Table 4.17.

Step 3. Degree of freedom and test statistic.

Because we use χ^2 proportion test, we must determine the degree of freedom (*df*) of χ^2 distribution. Obviously, *df* = 4.

The test statistic χ_*^2 can be calculated as follows:

$$\chi_*^2 = \sum_{i=1}^{5} \frac{(O_i^S - E_S)^2}{E_S} + \sum_{j=1}^{5} \frac{(O_j^F - E_F)^2}{E_F} = 50.46 \tag{4.8}$$

where E_S = 19.02 and E_F = 10.8 are expectation values of success and failed times respectively, O_i^S and O_j^F are observed success and failed times for the i^{th} and the j^{th} algorithms. For example, the observed success and failed times of A-ABC are 24 and 6, so O_2^S = 24 and O_2^F = 6.

Table 4.18 Contingency table of success and failed times on arbitrary random graph

Algorithms	Success Times (Observed/Expectation)	Failed Times (Observed/Expectation)	Total
DCS	30 (19.02)	0 (10.8)	30
A-ABC	24 (19.02)	6 (10.8)	30
S-ABC	22 (19.02)	8 (10.8)	30
HDPSO	14 (19.02)	16 (10.8)	30
D-FA	6 (19.02)	24 (10.8)	30
Total	96	54	150

Step 4. Calculate the *p*-Value.

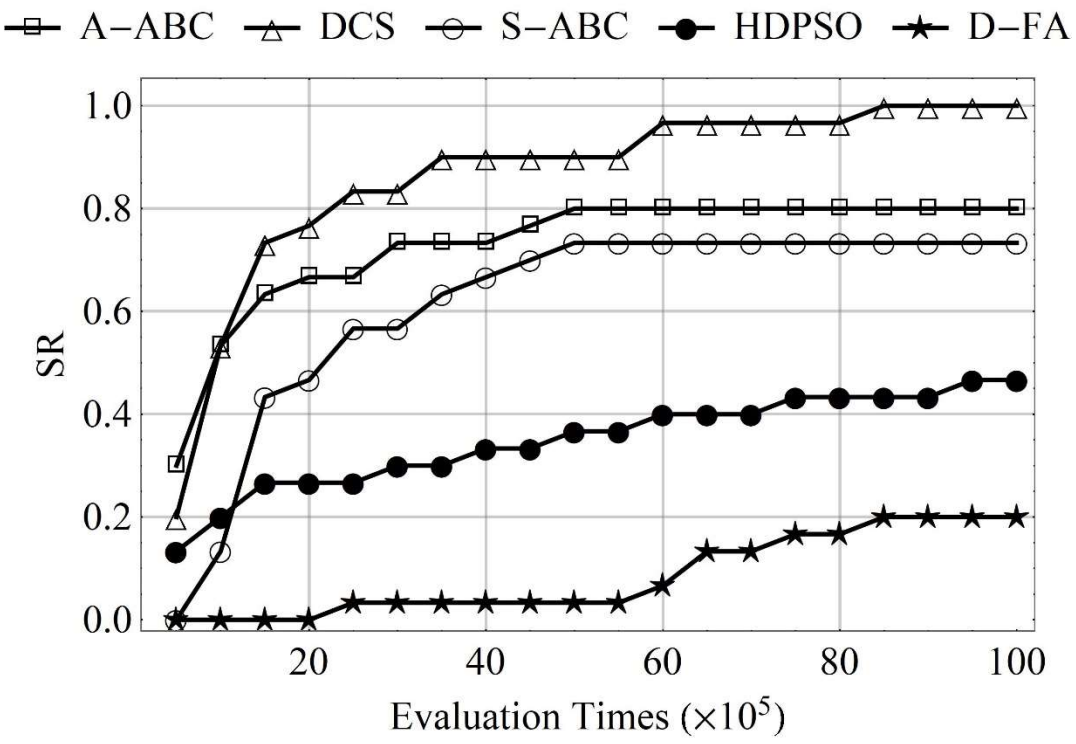
Using one tail χ^2 distribution, *p*-Value is given as follows:

$$p = P(\chi^2 > \chi_*^2 | df = 4) = 2.89 \times 10^{-10} \quad (4.9)$$

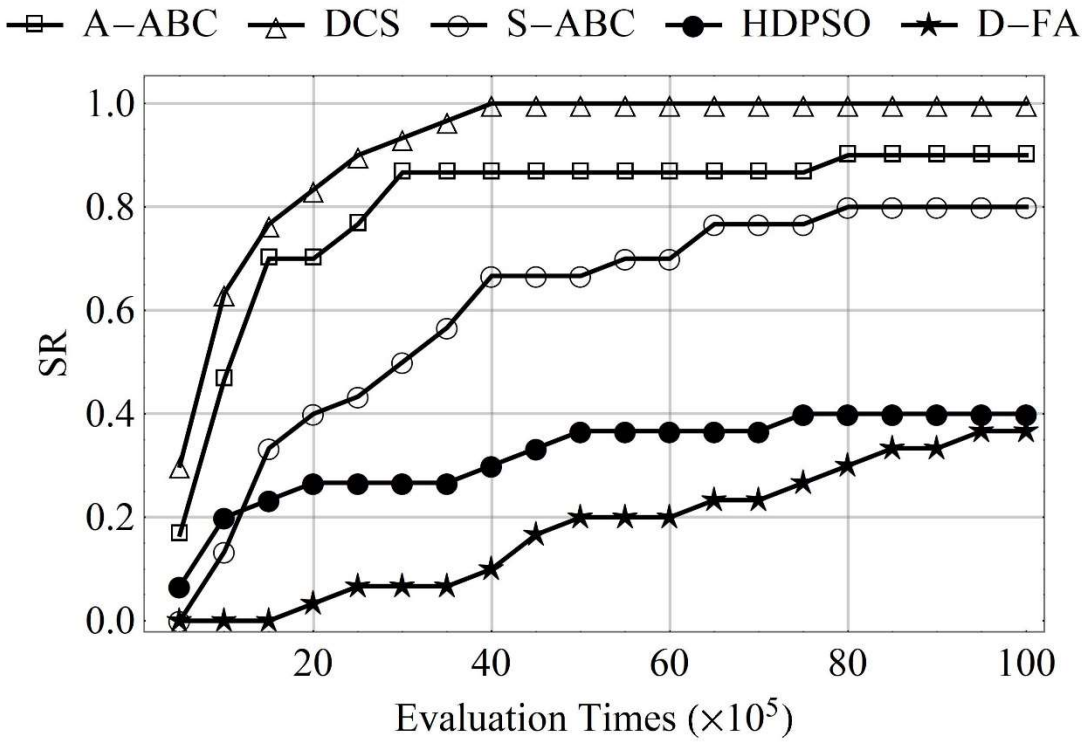
Step 5. Conclusion.

When level of significance α = 0.05, *p*-Value given in step 4 is obviously smaller than α. Thus, we *reject* null hypothesis $\mathcal{H}_0$ and state that the proportion of success times to failed times of the five algorithms is significant different.

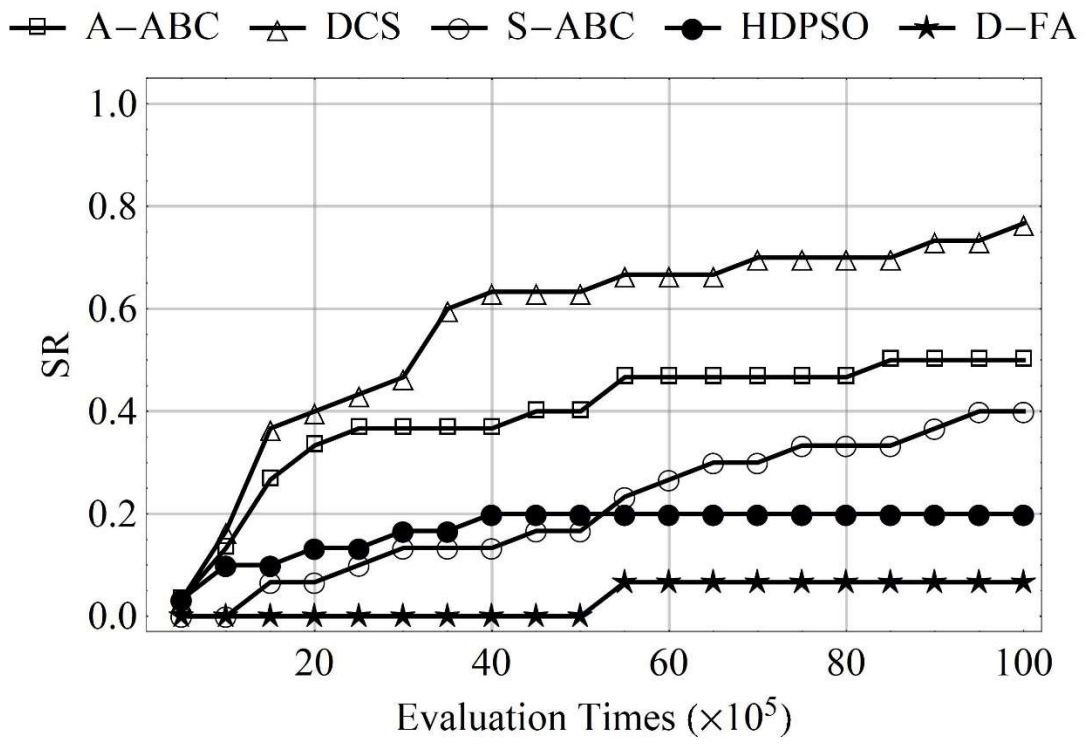
By plotting the accumulative SR, we can observe the convergence region of each algorithm. To do this, we separate *MaxEval*, which is 10,000,000 in this experiment, into 20 regions by an interval 500,000. Figure 4.4(a), (b), and (c) are corresponding to Table 4.14, 4.15, and 4.16 respectively, and show the accumulative SR on three classes of random graphs.



(a) The accumulative SR on arbitrary random graphs



(b) The accumulative SR on equi-partite graphs



(c) The accumulative SR on flat graphs

Fig. 4.4 The accumulative SR of each algorithm on three classes of random graphs

From Figure 4.4, we find that: (1) DCS obtains the highest performance among the five algorithms. When solving arbitrary random graphs and equi-partite graphs, DCS obtains 100% SR and finds most solutions in a small evaluation times (less than 4,000,000). (2) A-ABC is the second-best algorithm. In contrast to S-ABC, HDPSO, and D-FA, A-ABC obtains higher SR and convergence speed. By tuning *t* with adaptive function during evolution, A-ABC can find more solutions in a small evaluation times. For example, SRs of A-ABC, S-ABC, HDPSO, and DFA are 0.70, 0.40, 0.27, and 0.03 respectively when Evaluation Times = 2,000,000 in Figure 4.4 (b). This shows A-ABC can find more solutions than other algorithms even if given a strict search condition, such as a small *MaxEval*.

In summary, A-ABC is the second-best algorithm in this experiment. Not only it obtains high SR but also consumes small evaluation times. By studying the convergence region, we also observe that A-ABC can find more solutions if given a small *MaxEval*.

4.2.5 Large Comparison Study

In above experiment, we compared five algorithms on 3-colorable Culberson random graphs with $n = 120$ and $p = 0.058$. To show the robustness of A-ABC, in this section, we design some experiments that compare the candidate algorithms on graphs with various n and p. The graph settings and algorithm settings are given in Table 4.19 and Table 4.20 respectively.

In this section, the candidate algorithms are tested on three graph sizes: 100, 150, and 200. For each graph size, various graph instances are generated by using different edge probability p. Edge probability p, which is the order parameter, begins at p_{start}, increases by a small interval Δp, and ends with p_{end}. Using this setting, 15 different values of p are generated, and they can cover the experiment and theory phase transitions, which are written as *Experiment PT* and *Theory PT* respectively in Table 4.19. As mentioned in section 2.2.3, the most difficult graph instances arise when p is in the range of phase transition. However, there is a small gap between Experiment PT and Theory PT. By testing the candidate algorithms on various p, not only we can study the robustness of algorithms, but we can also observe which phase transition, Experiment PT or Theory PT, is harder than the other one.

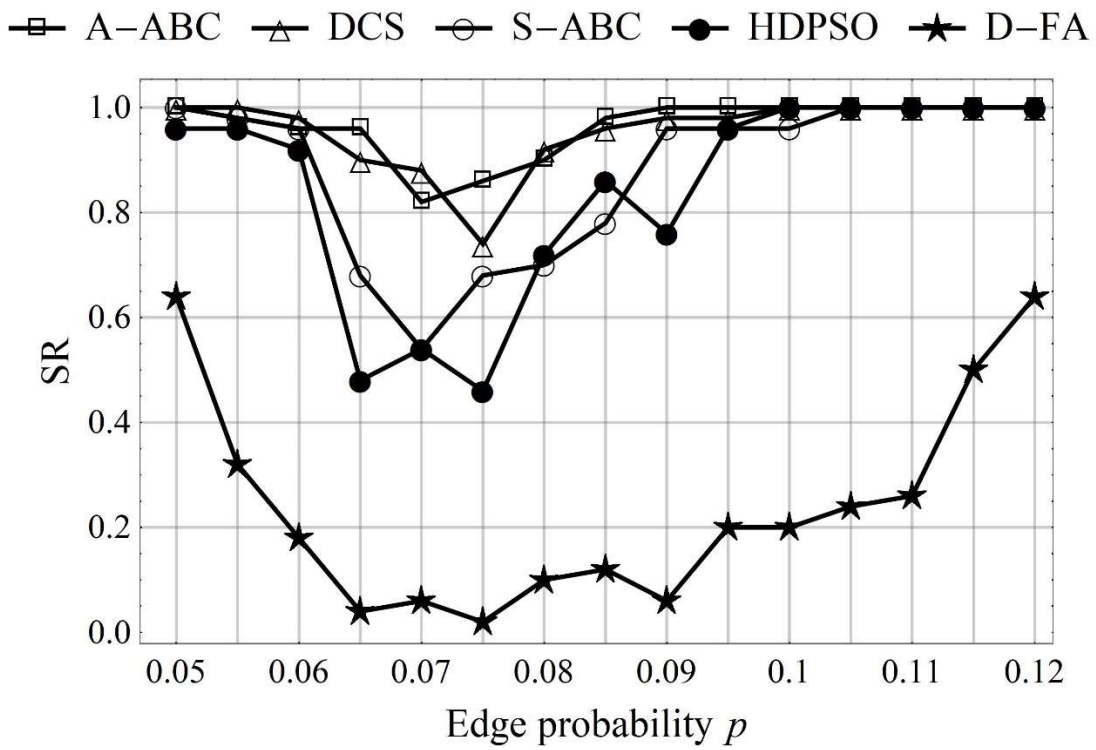
Table 4.19 The 3-colorable graph settings

n	p_{start}	p_{end}	Δp	Experiment PT	Theory PT
100	0.050	0.115	0.005	0.070, 0.075, 0.080	0.080, 0.085, 0.090
150	0.031	0.070	0.003	0.046, 0.049, 0.052	0.055, 0.058, 0.061
200	0.025	0.064	0.003	0.034, 0.037, 0.040	0.040, 0.043, 0.046

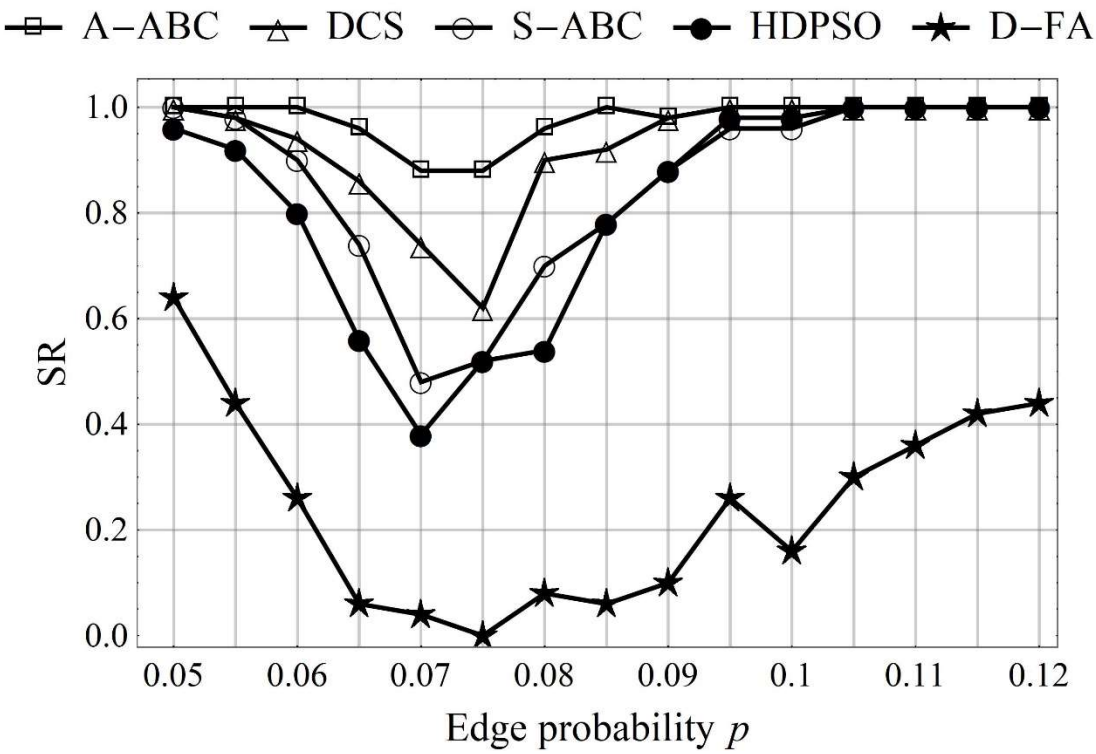
Table 4.20 Algorithms settings for large comparison study

A-ABC	$N = 200$, $u = 2$, scout bee phase is not used
DCS	$N = 10$, $\alpha = 1.0$, $\beta = 1.5$, $p_a = 0.0001$
S-ABC	$N = 200$, $t = 2$, $limit = 80$
HDPSO	$N = 10$, $w_0 = 0.05$, $c_1 = 7.0$, $c_2 = 0.03$
D-FA	$N = 200$, $\alpha = 2$, $\beta_0 = 1.0$, $\gamma = 0.2$
MaxEval	1,000,000 when $n = 100$ 7,000,000 when $n = 150$ 10,000,000 when $n = 200$
Graph	3-colorable Culberson random graphs (Arbitrary random graph, equi-partite graph, and flat graph) with various n and p
The number of graph instances	50

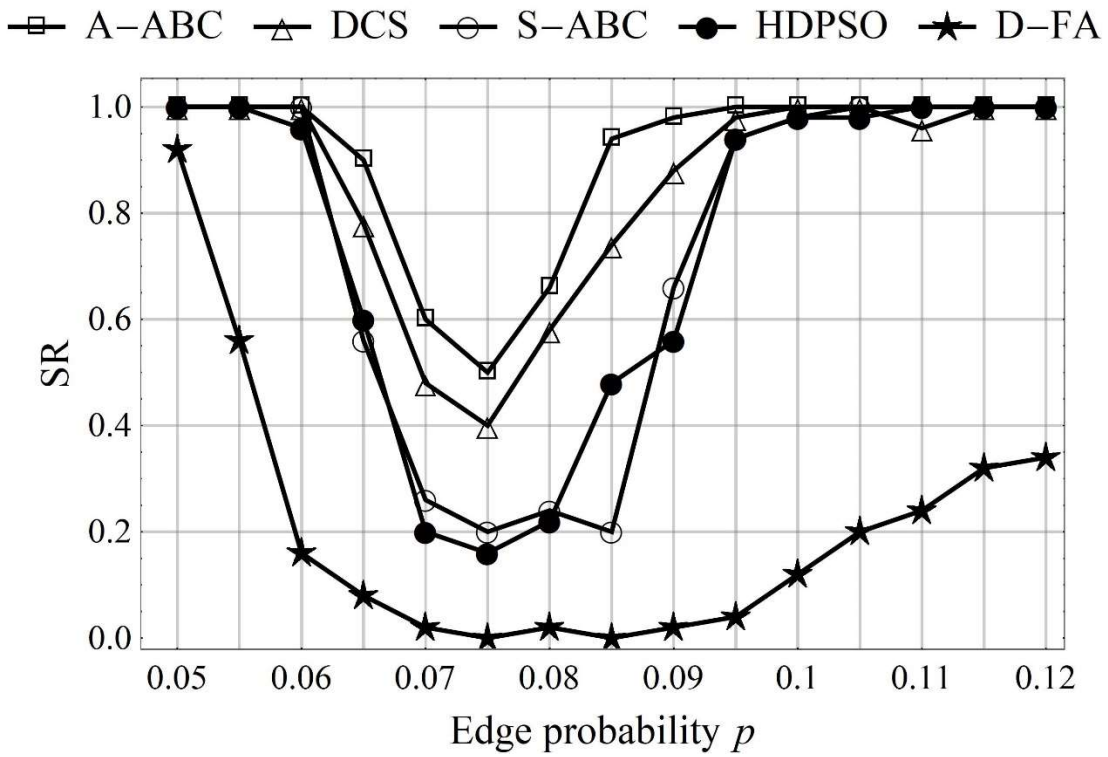
The candidate algorithms are compared on 3-colorable Culberson random graphs with various n and p given in Table 4.19. For each pair of n and p, the candidate algorithms are tested on 50 graph instances. *MaxEval* is various according to graph size. The bigger the graph size is, the larger *MaxEval* is. The results are given in Figure 4.5, 4.6, and 4.7.



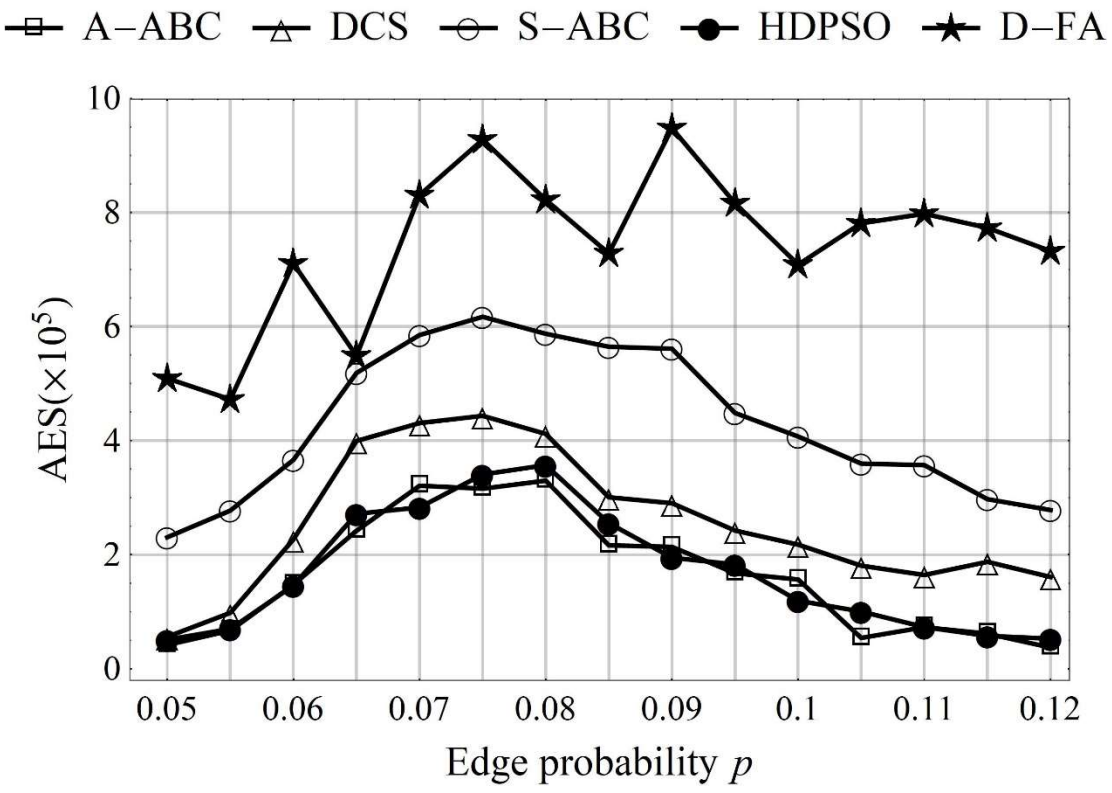
(a) SR of arbitrary random graphs (n = 100)



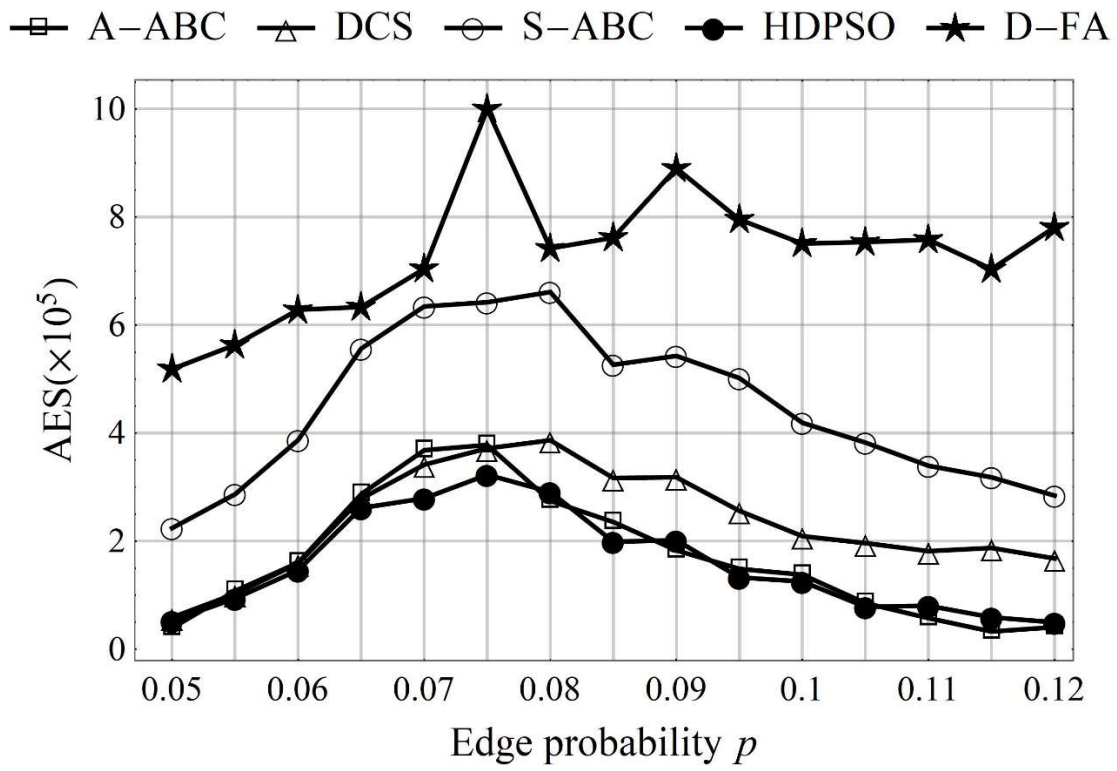
(b) SR of equi-partite graphs (n = 100)



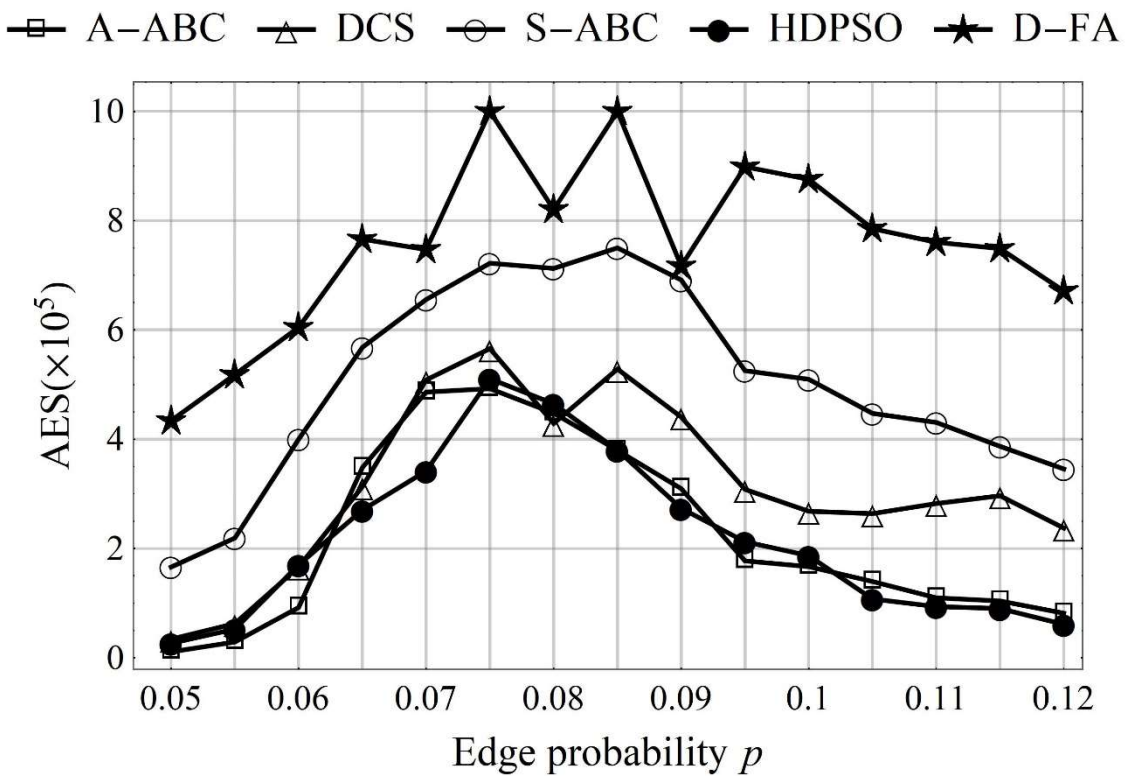
(c) SR of flat graphs (n = 100)



(d) AES of arbitrary random graphs (n = 100)

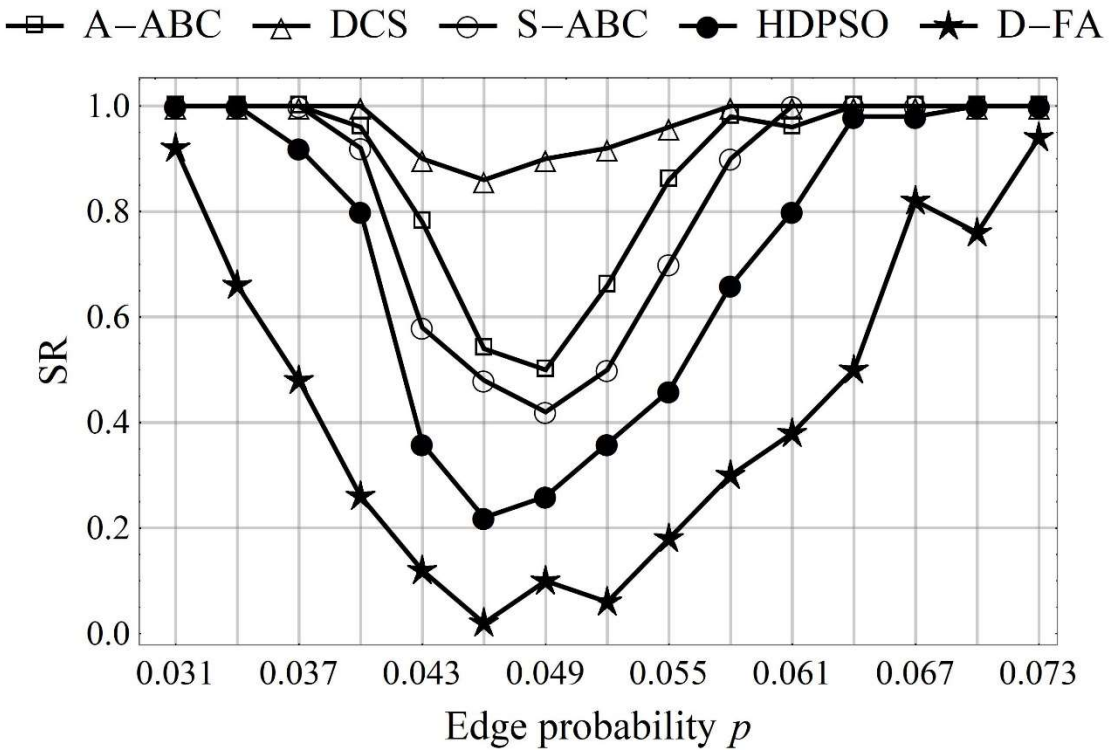


(e) AES of equi-partite graphs (n = 100)

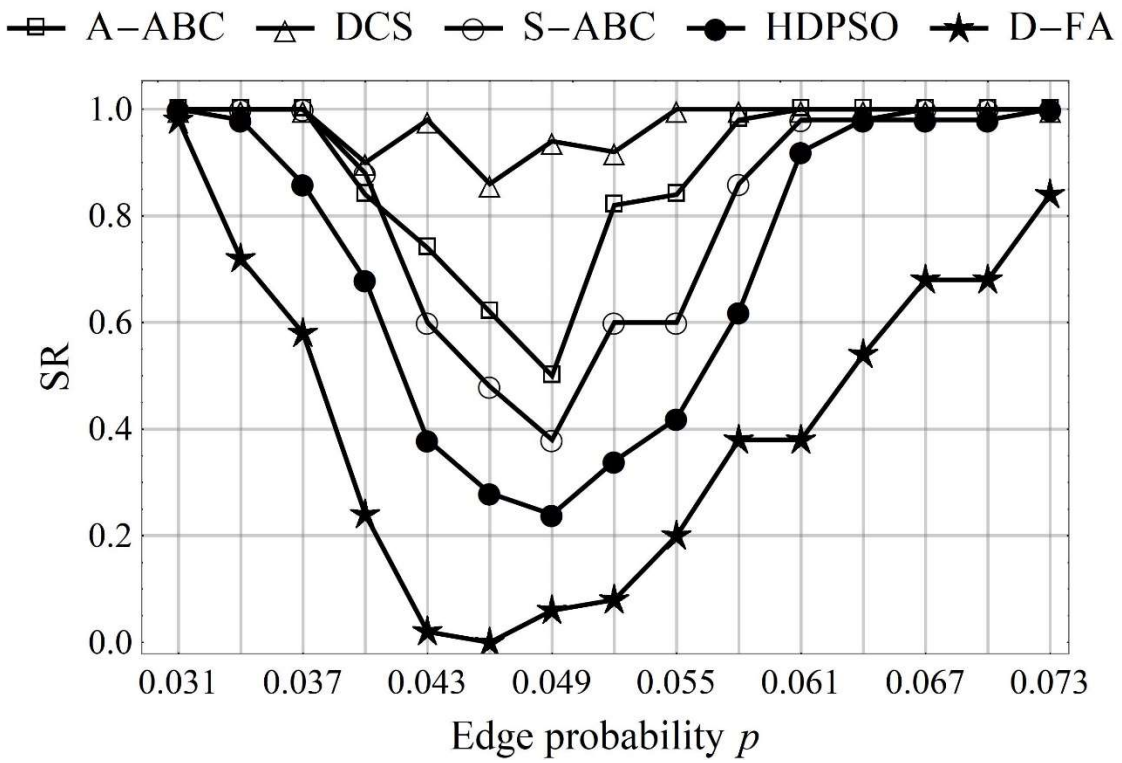


(f) AES of flat graphs (n = 100)

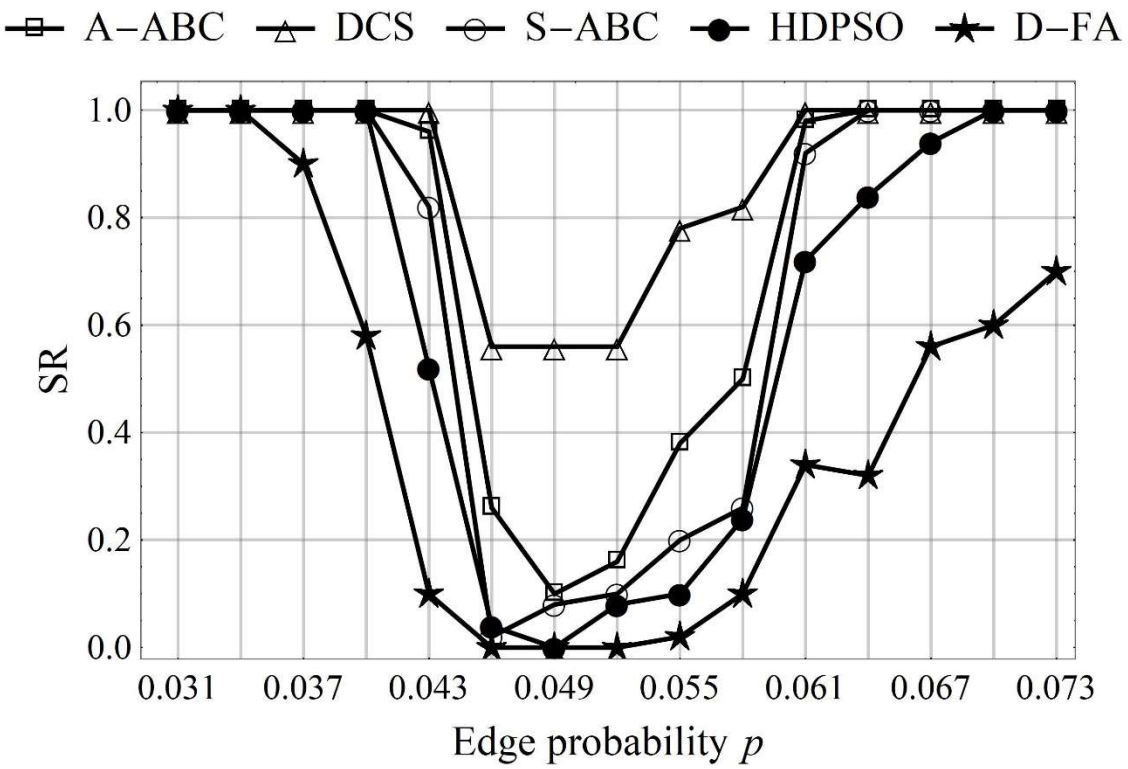
Fig. 4.5 Comparison study on random graphs with n = 100



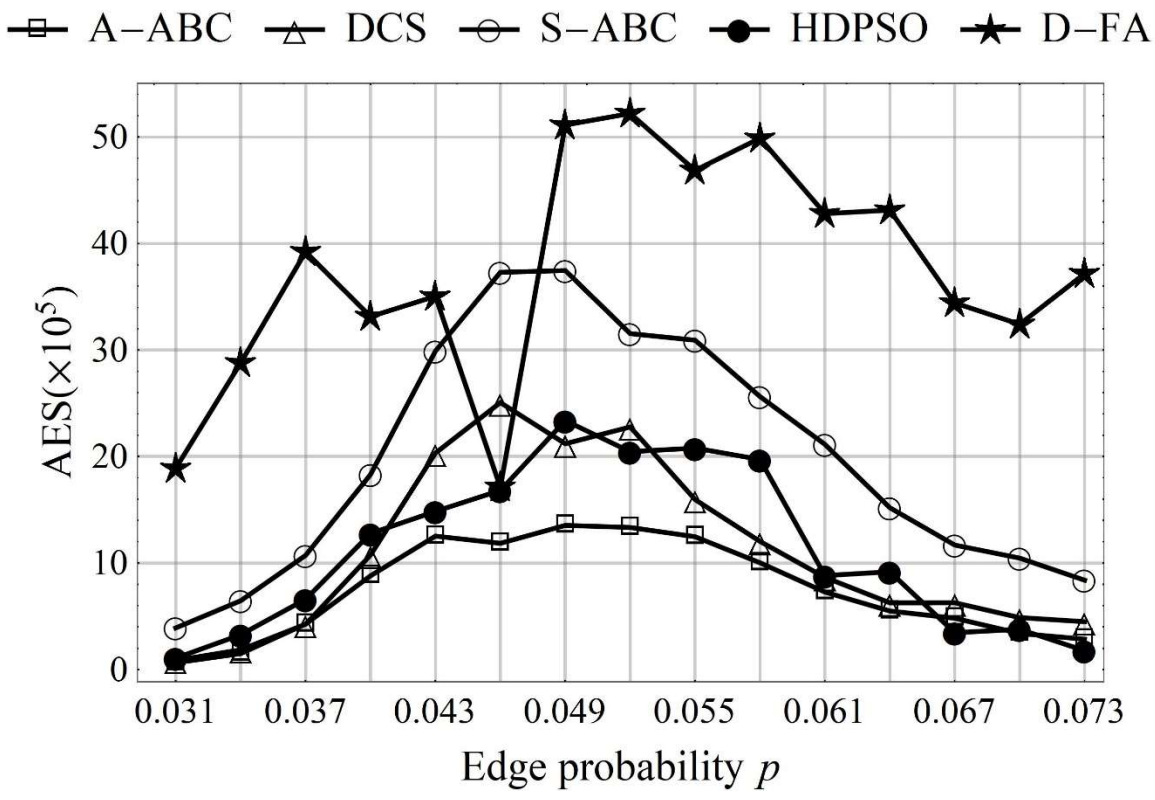
(a) SR of arbitrary random graphs (n = 150)



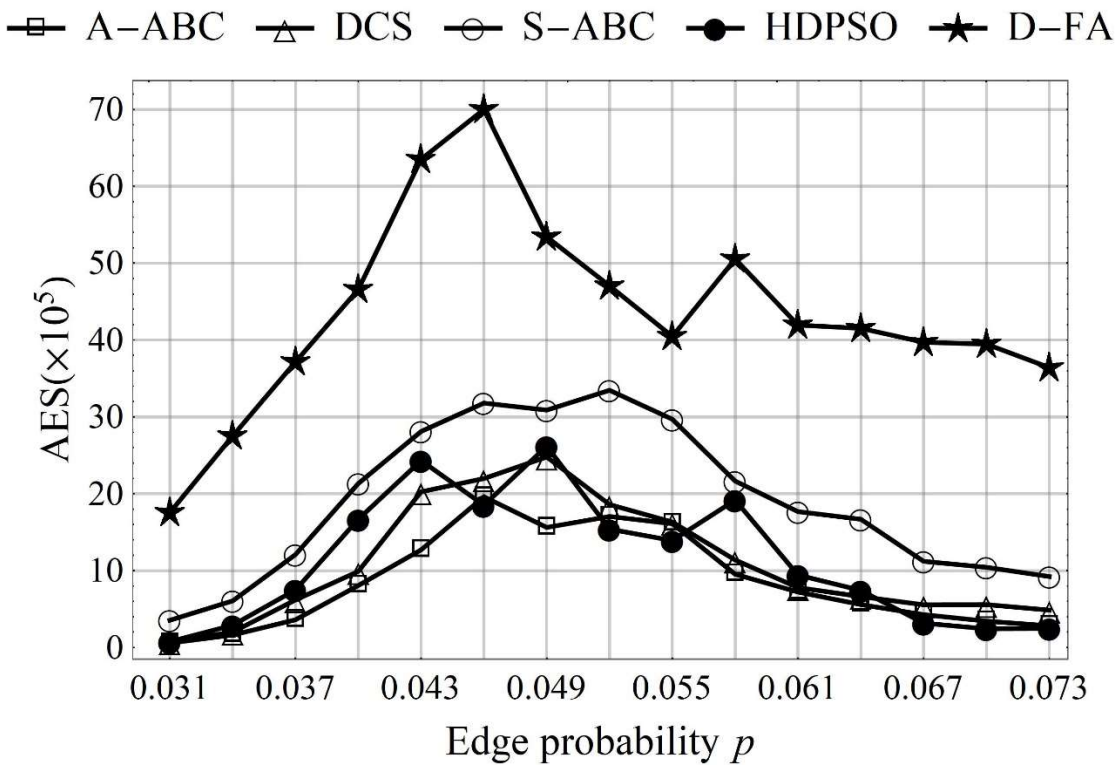
(b) SR of equi-partite graphs (n = 150)



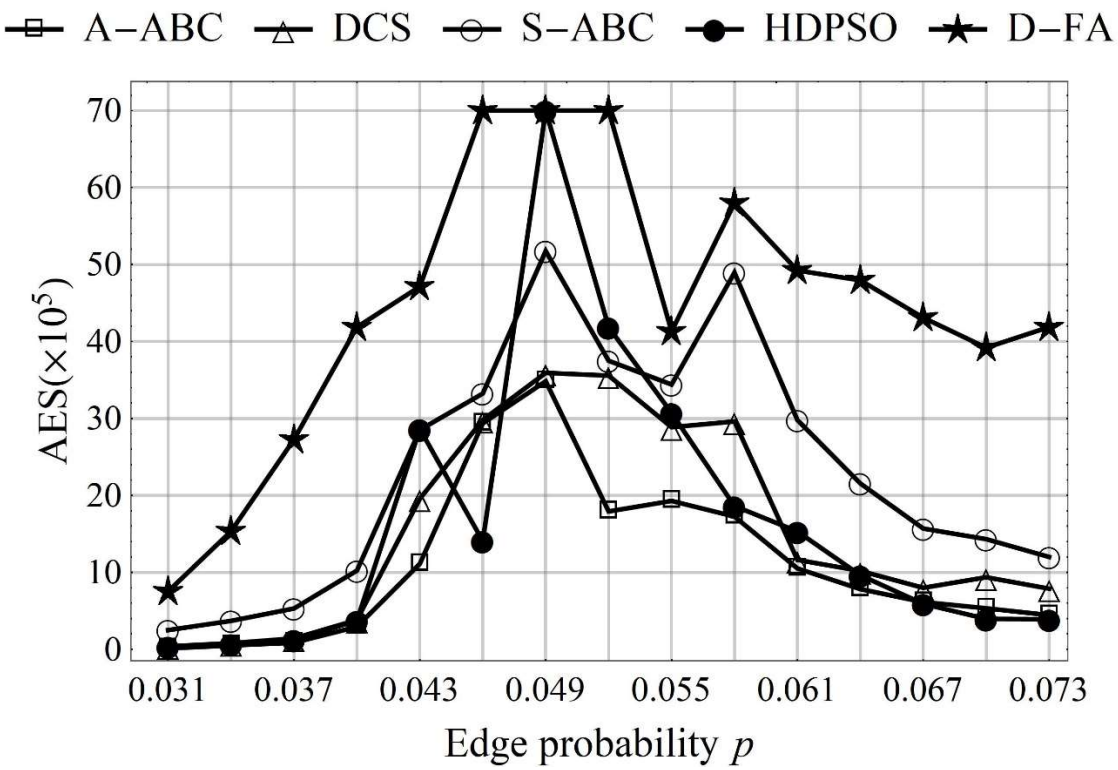
(c) SR of flat graphs (n = 150)



(d) AES of arbitrary random graphs (n = 150)

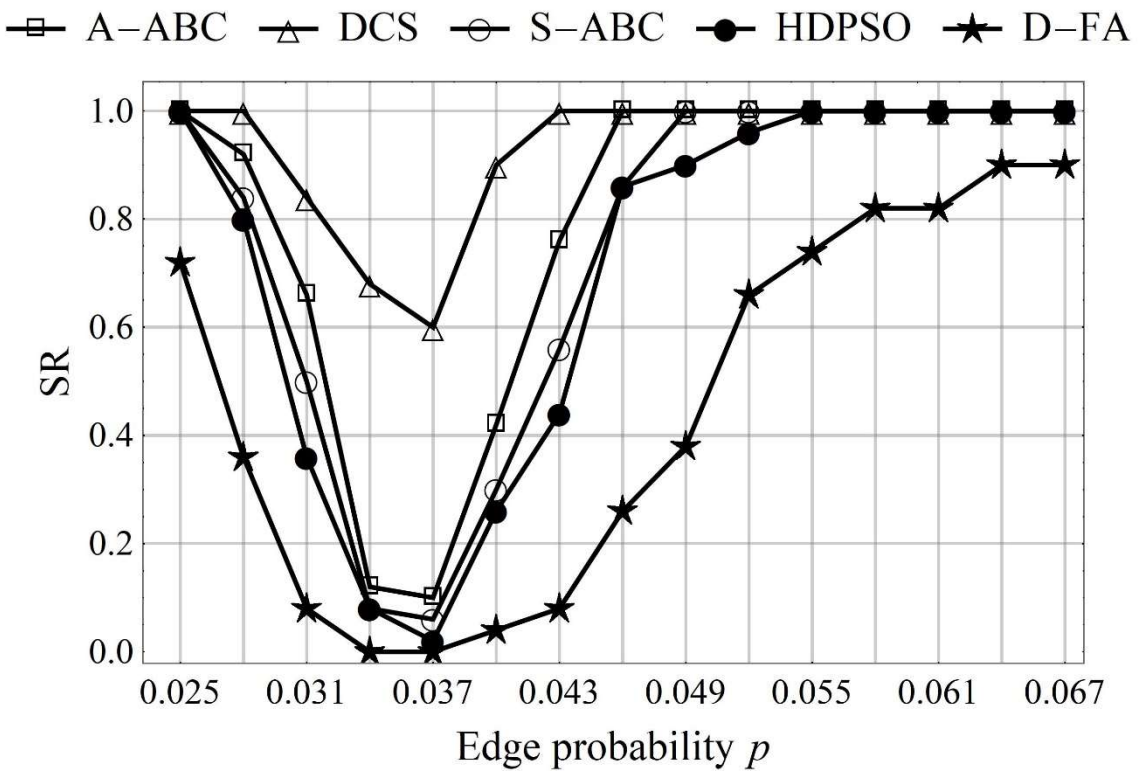


(e) AES of equi-partite graphs

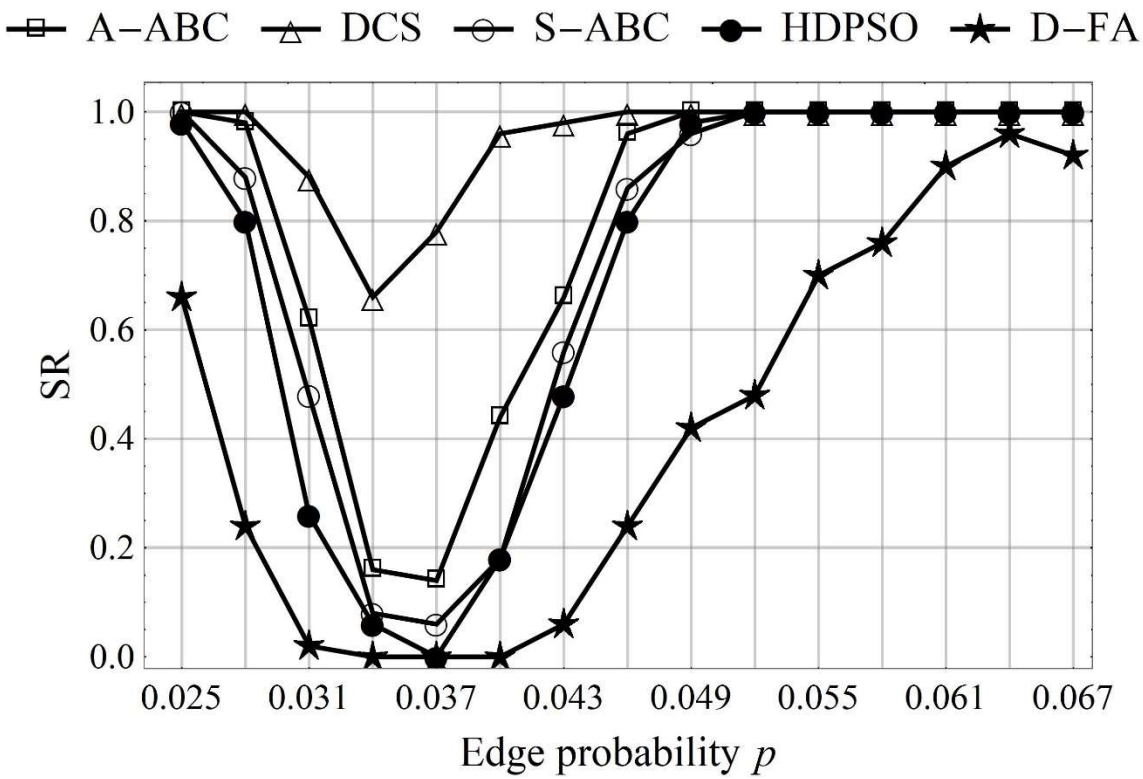


(f) AES of flat graphs

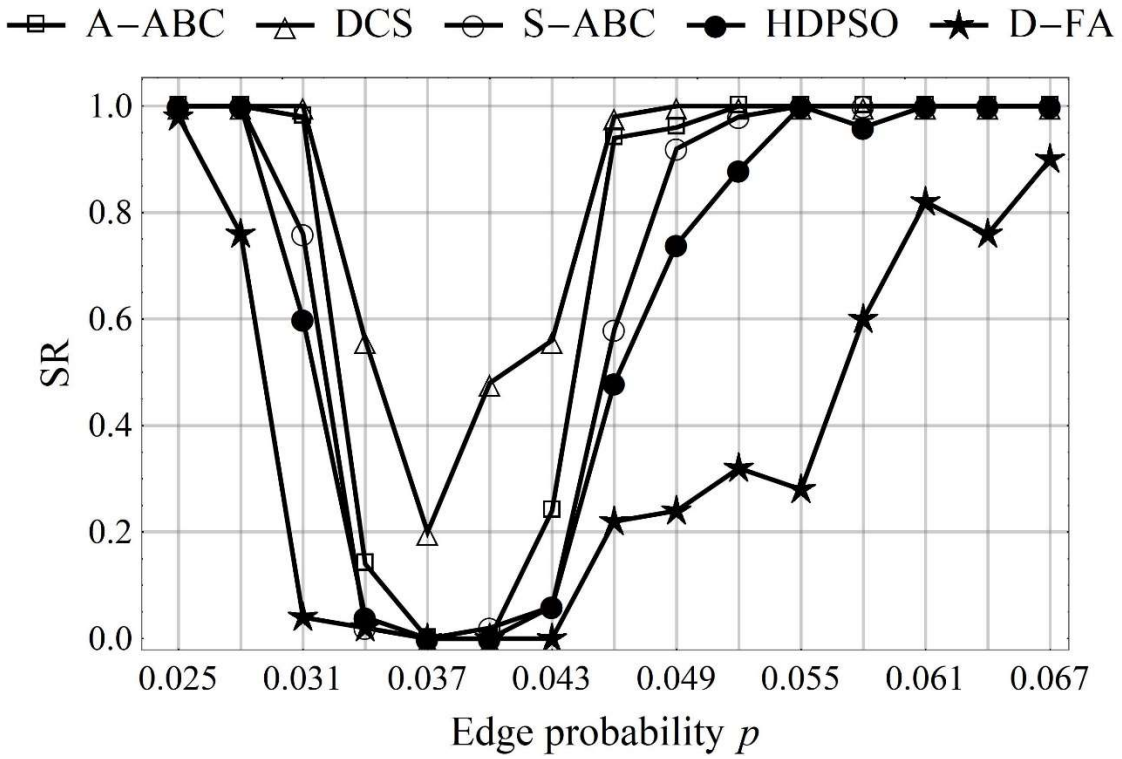
Fig. 4.6 Comparison study on random graphs with n = 150



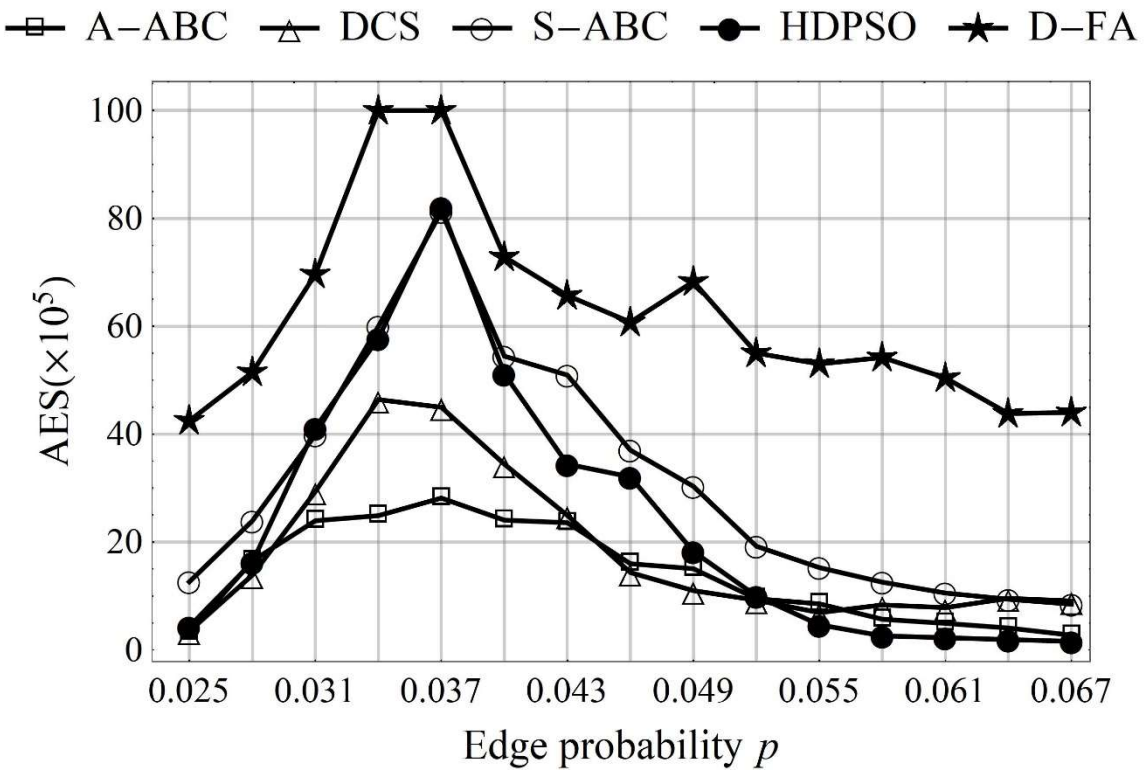
(a) SR of arbitrary random graphs (n = 200)



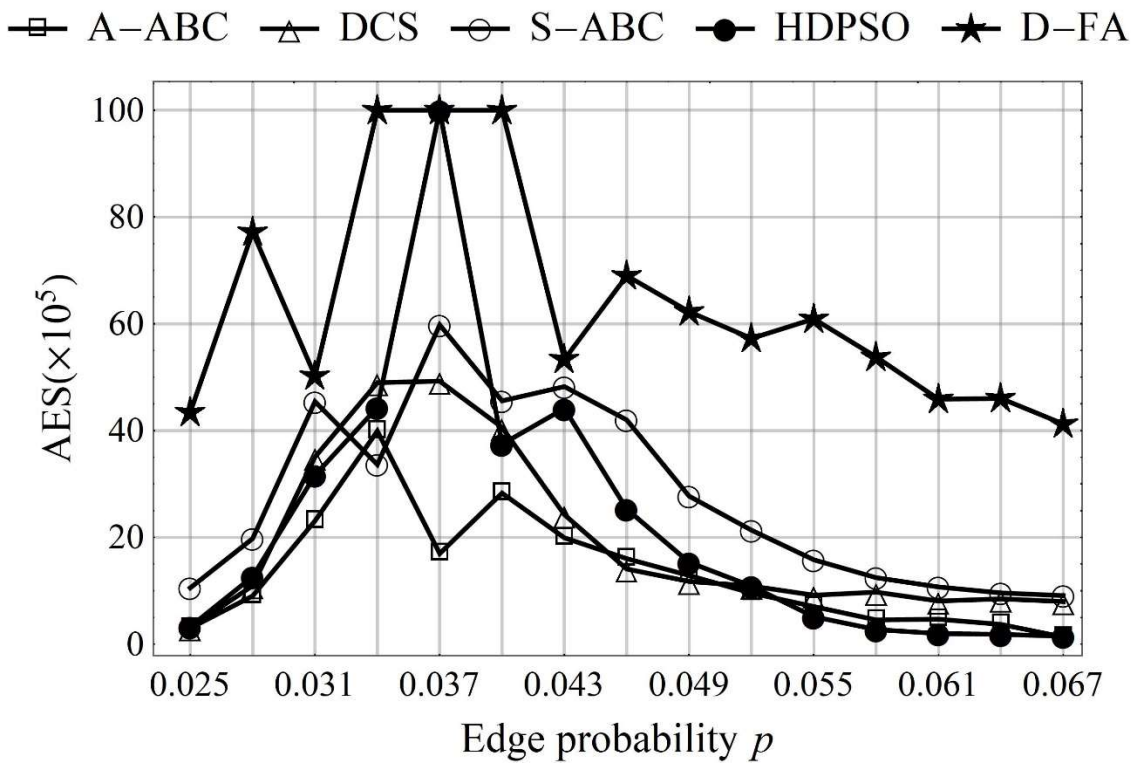
(b) SR of equi-partite graphs (n = 200)



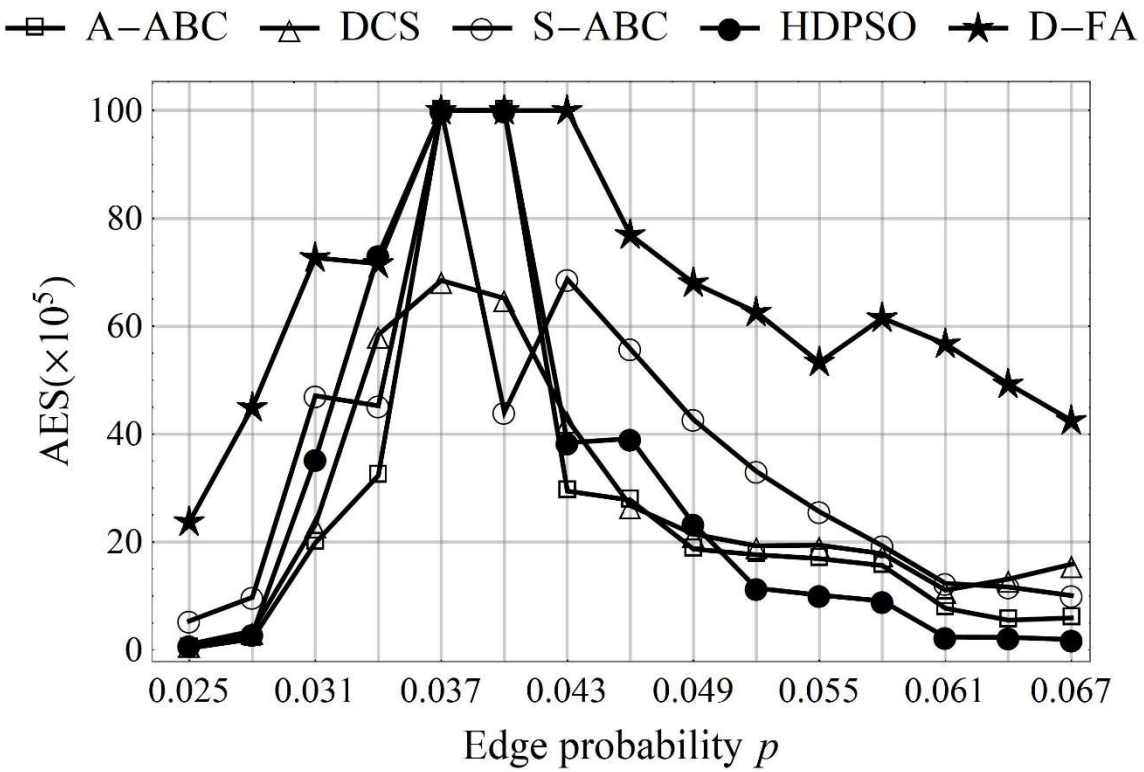
(c) SR of flat graphs (n = 200)



(d) AES of arbitrary random graphs (n = 200)



(e) AES of equi-partite graphs (n = 200)



(f) AES of flat graphs (n = 200)

Fig. 4.7 Comparison study on random graphs with n = 200

■ Discussion

Figure 4.5, 4.6, and 4.7 are grouped by graph instances with various n and each Figure is divided into 6 diagrams according to SR and AES. Note that if SR is 0, the corresponding AES is set to *MaxEval*. For example, DFA cannot find any solution in all 50 flat graph instances with $n = 100$ and $p = 0.075$, thus SR = 0 (Figure 4.5 (c)). In this case, the related AES in Figure 4.5 (f) is set to 1,000,000.

First, from these results, A-ABC outperforms S-ABC, HDPSO, and DFA but inferior to DCS on SR in most cases, especially in the range of phase transition. A-ABC is also a fast algorithm, its AES is similar to DCS and is obviously smaller than S-ABC, HDPSO, and DFA. For example, AES of A-ABC is half as much as that of S-ABC in the range of phase transition in nearly all experiments.

Next, the performance of DFA is very poor. This does not oppose the results given in section 3.4.3. Because of the double loop structure of DFA, extremely large AES is the cost of obtaining high SR. Thus, it is difficult for DFA to find a solution in a limited *MaxEval*.

Finally, by checking SR in Experiment PT and Theory PT, the graph instances in Experiment PT is much harder than those in Theory PT. We also find that SR on flat graph instances is commonly lower than that on instances of arbitrary random graph and equi-partite graph. The results in our experiments are consistent with those mentioned in section 2.2.3: Experiment PT is a little different from Theory PT and the difficulty of a random graph is affected not only by edge probability p, but also by its topology.

In summary, A-ABC is not only efficient but also very robust. By adjusting t by adaptive function during the evolution, it adapts more effectively to random graphs with various sizes and topology than S-ABC, HDPSO, and DFA, which use static parameters.

4.2.6 Generality of A-ABC

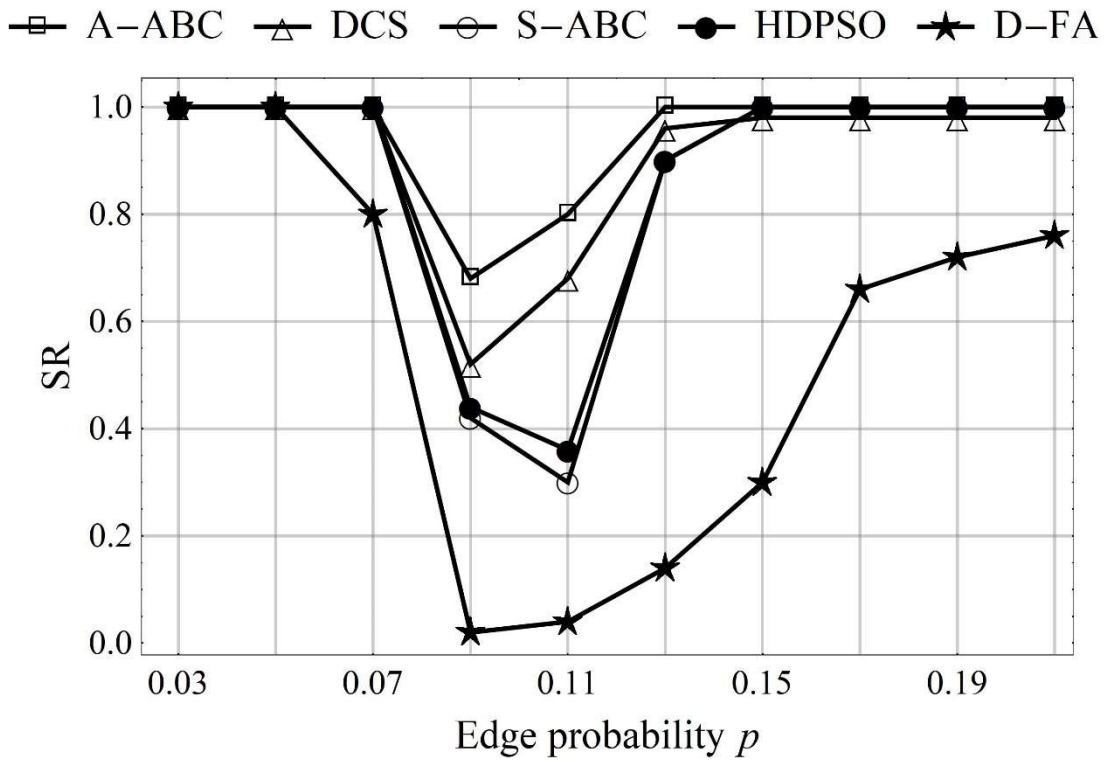
The above experiments are all performed on 3-colorable Culberson random graphs. However, many practical applications, which can be modelled on graph coloring problems, may not be limited to 3-colorable problems. Thus, in this section, we study the generality of A-ABC by comparing it with other algorithms on k-colorable graphs, where $k > 3$. The graph settings are given in Table 4.21.

Table 4.21 The 4-colorable and 5-colorable graph settings

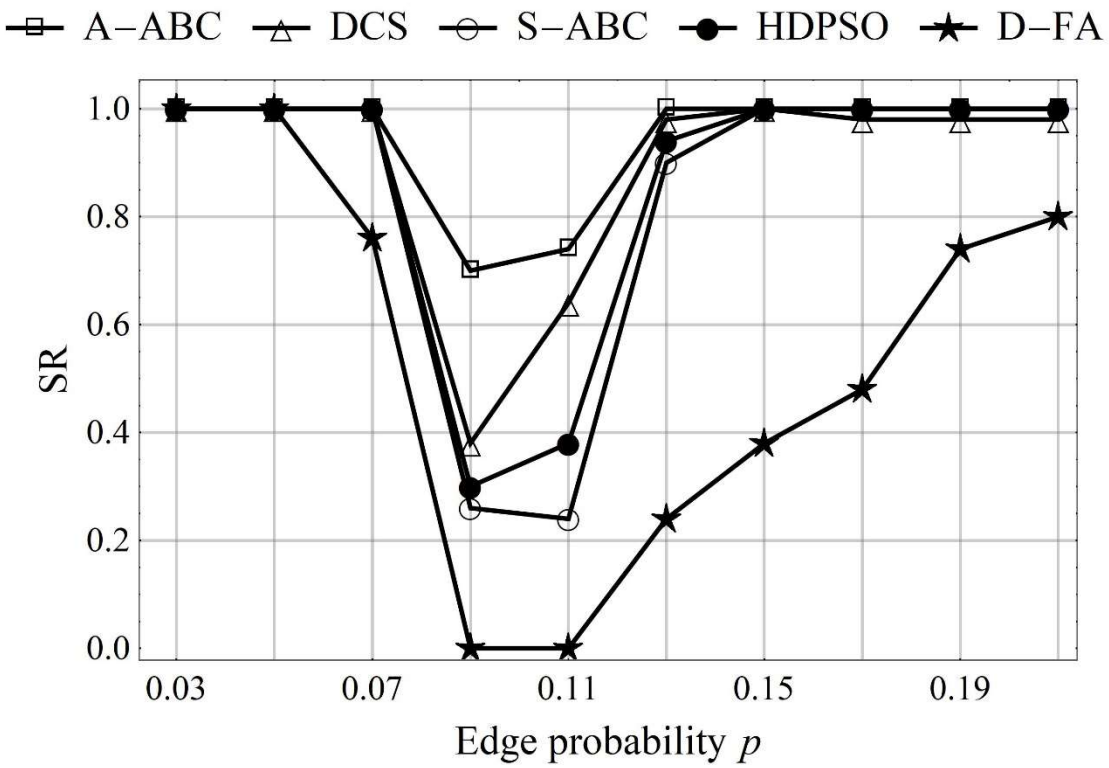
Type	n	p_{start}	p_{end}	Δp	Experiment PT	Theory PT
4-colorable	120	0.030	0.210	0.020	0.090, 0.110	0.130
5-colorable	150	0.080	0.170	0.010	0.110, 0.120	0.130,0.140

There are two classes of graphs in the test suite: 4-colorable Culberson random graphs with $n = 120$ and 5-colorable Culberson random graphs with $n = 150$. The corresponding Experiment PT and Theory PT can be calculated from Table 2.2.

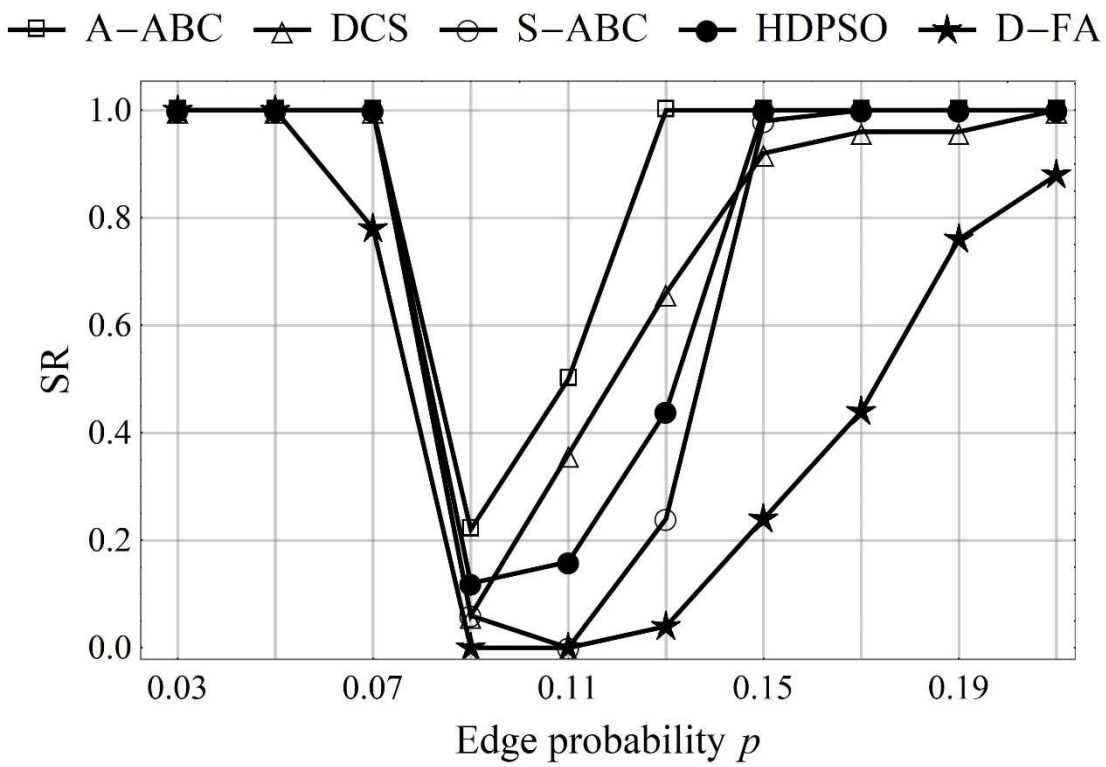
As we did in section 4.2.5, for each pair of n and p, the candidate algorithms (i.e. A-ABC, DCS, S-ABC, HDPSO, and D-FA) are tested on 50 graph instances. The parameter settings of each algorithm are identical to those in Table 4.20. The values of *MaxEval* are set to 3,000,000 and 7,000,000 for $n = 120$ and $n = 150$ respectively. The results are shown in Figure 4.8 and 4.9.



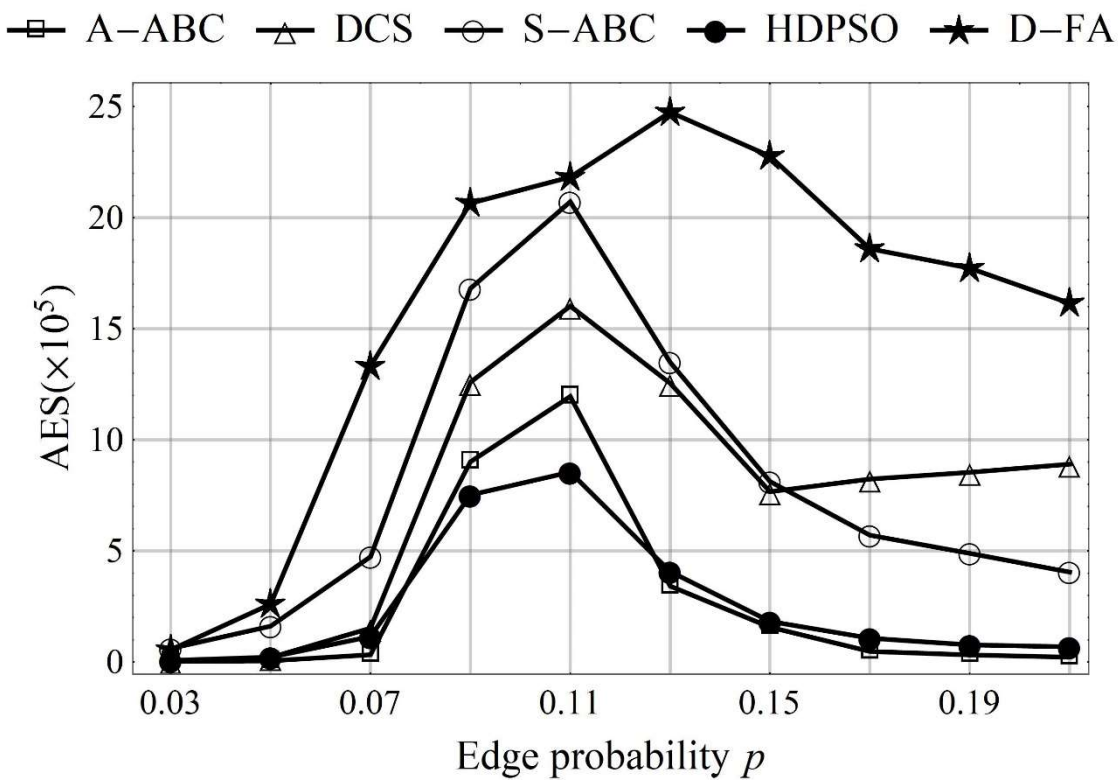
(a) SR of 4-colorable arbitrary random graphs (n = 120)



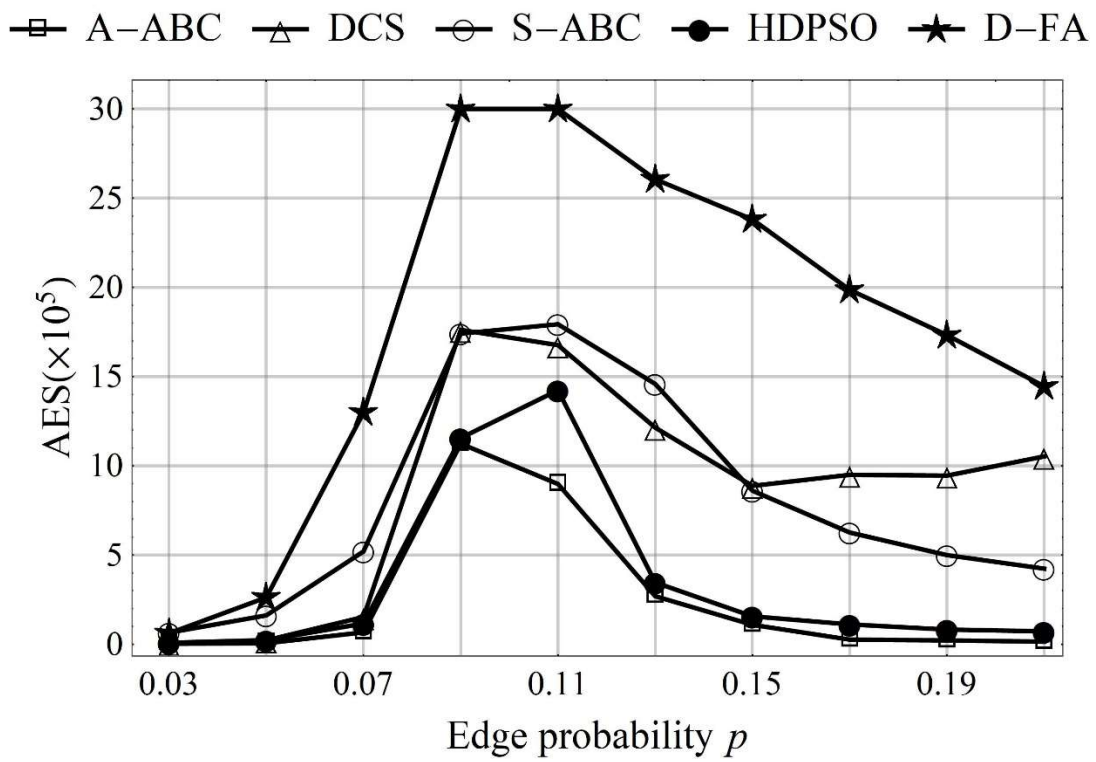
(b) SR of 4-colorable equi-partite graphs (n = 120)



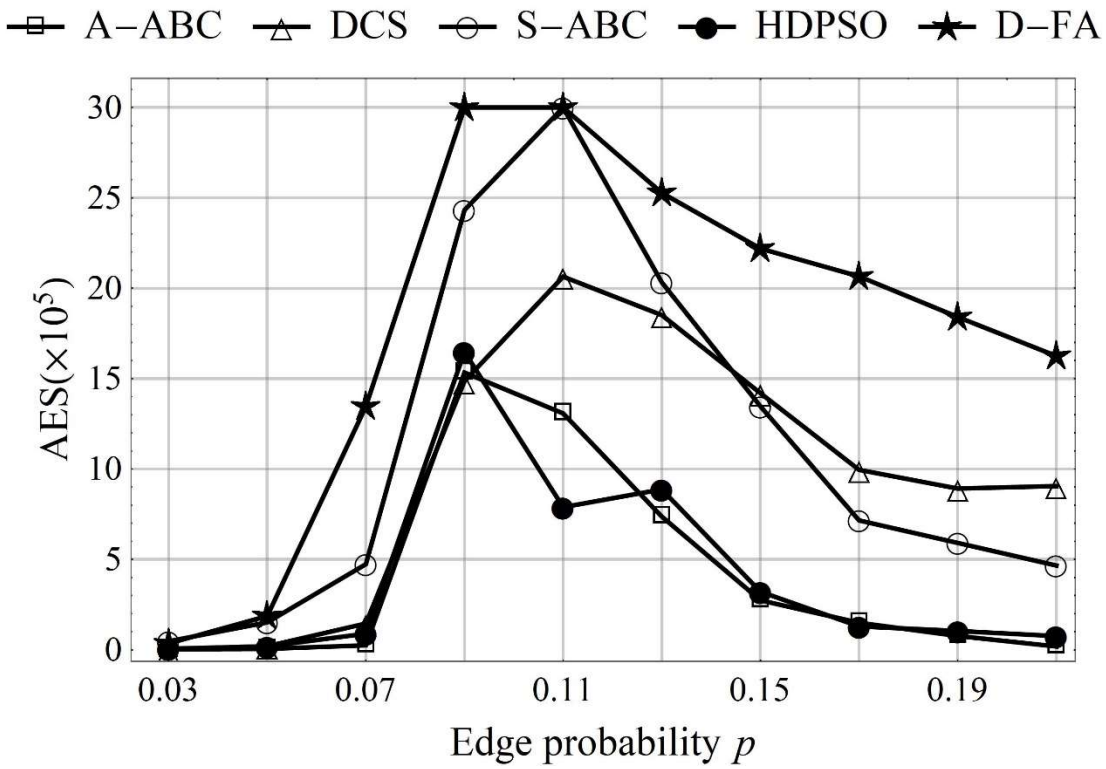
(c) SR of 4-colorable flat graphs (n = 120)



(d) AES of 4-colorable arbitrary random graphs (n = 120)

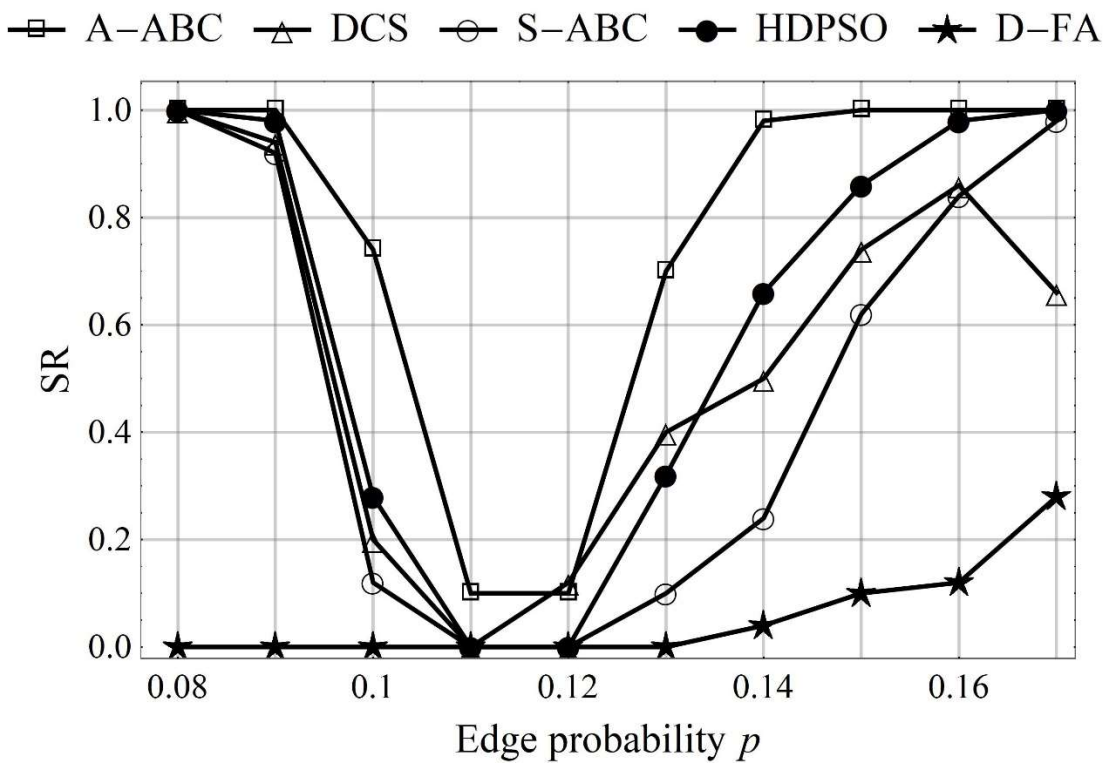


(e) AES of 4-colorable equi-partite graphs (n = 120)

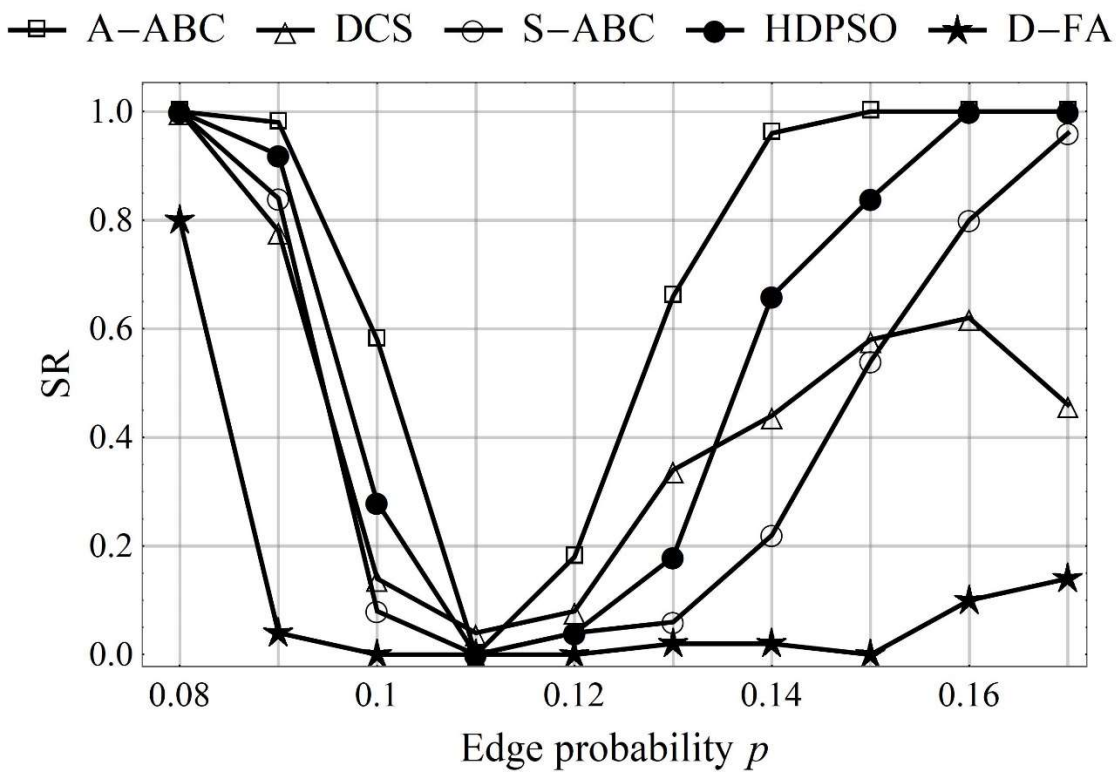


(f) AES of 4-colorable flat graphs (n = 120)

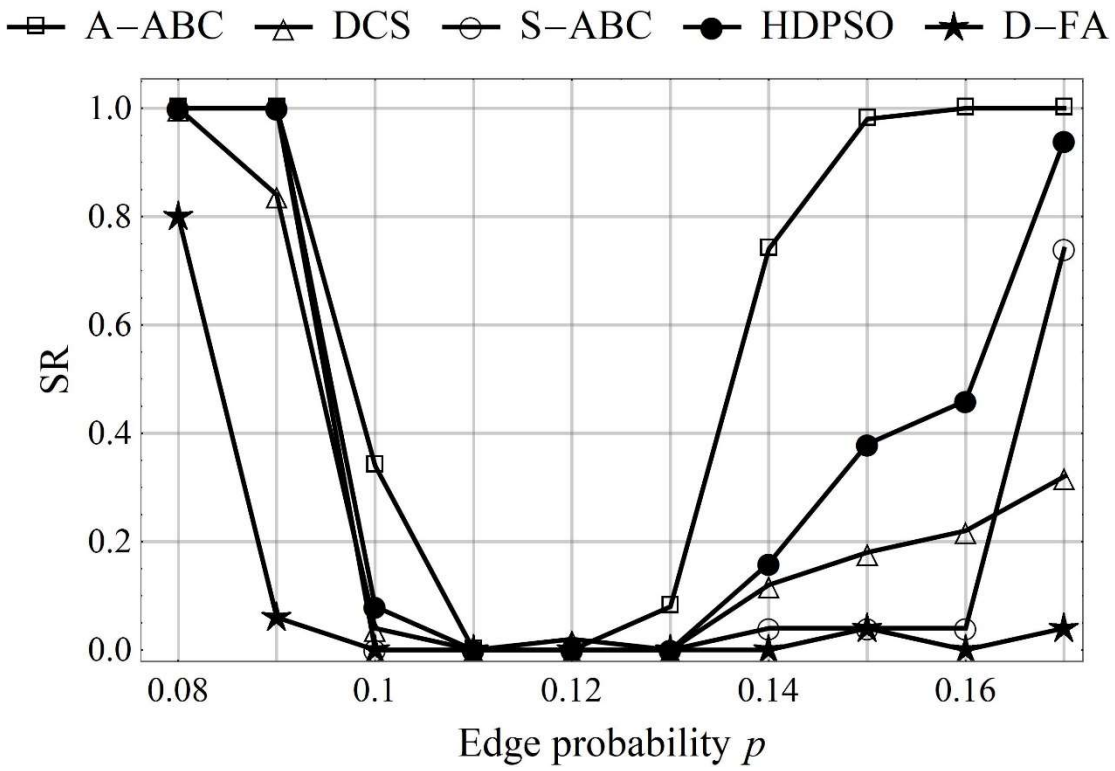
Fig. 4.8 Generality study on 4-colorable random graphs with n = 120



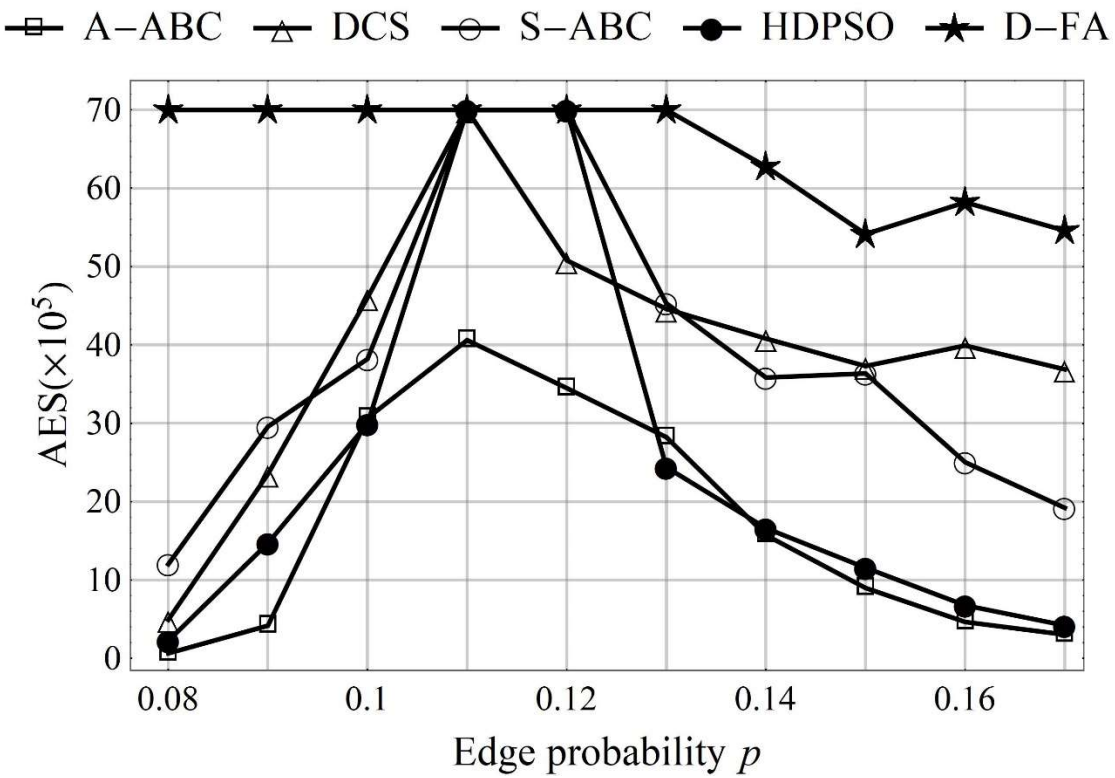
(a) SR of 5-colorable arbitrary random graphs (n = 150)



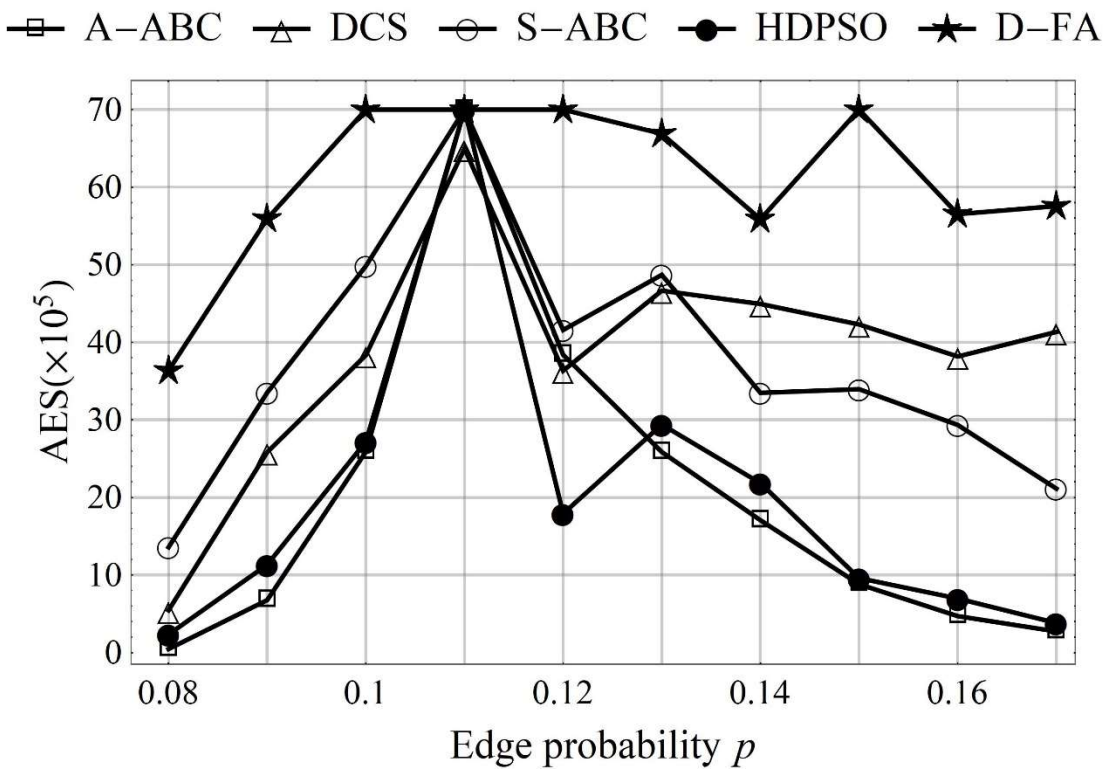
(b) SR of 5-colorable equi-partite graphs (n = 150)



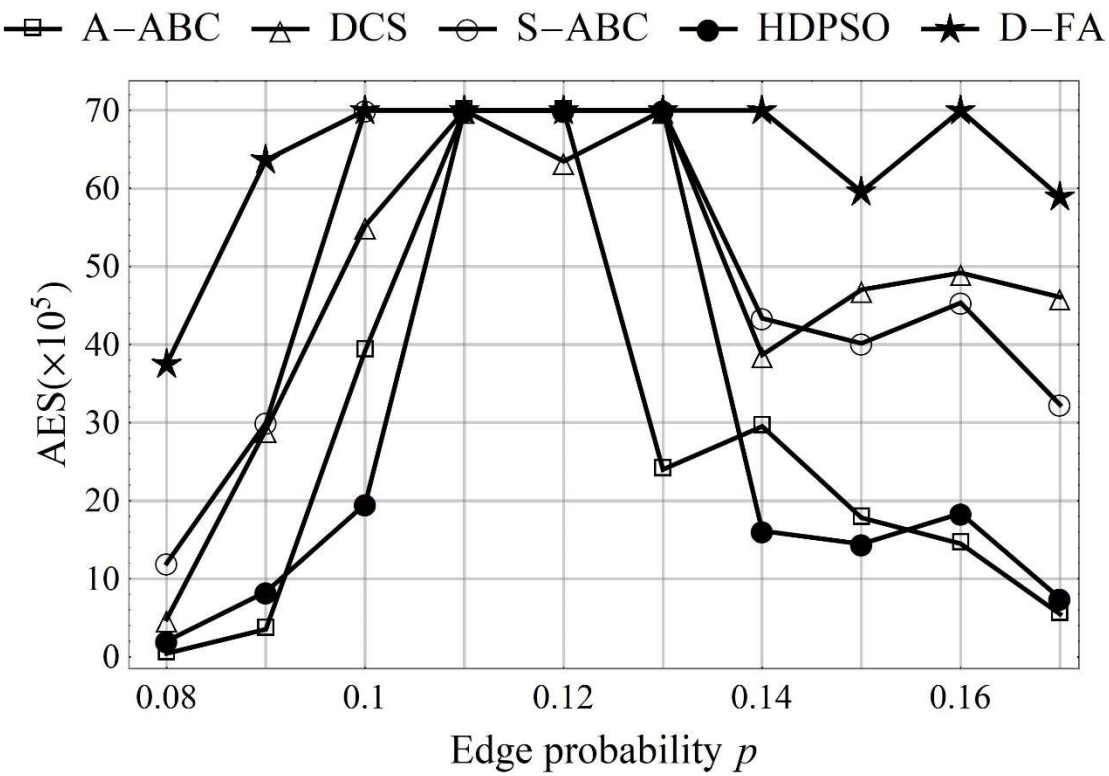
(c) SR of 5-colorable flat graphs (n = 150)



(d) AES of 5-colorable arbitrary random graphs (n = 150)



(e) AES of 5-colorable equi-partite graphs (n = 150)



(f) AES of 5-colorable flat graphs (n = 150)

Fig. 4.9 Generality study on 5-colorable random graphs with n = 150

■ Discussion

From Figure 4.8 and 4.9, we see that A-ABC outperforms the other four candidate algorithms on SR and AES in most case. In section 4.2.5, DCS is the best algorithm that obtains the highest SR and the lowest AES in most cases when solving 3-colorable random graphs. However, when solving 4-colorable and 5-colorable random graphs, DCS's performance gets worse on graph instances that are both in and out the range of phase transition. For example, when solving 4-colorable flat graphs, SR of A-ABC is obviously higher than that of DCS in the range of phase transition (Figure 4.8(c)) while AES of A-ABC is only half as much as that of DCS when $p >$ 0.11 (Figure 4.8(f)). It shows that even though both A-ABC and DCS are adaptive algorithms, A-ABC is more general and is applicable to vast topologies of random graphs.

We also observe that SR of S-ABC is lower than that of HDPSO both in and out the range of phase transition when solving 4-colorable and 5-colorable random graphs, which is just the opposite of the results in section 4.2.5. This shows that S-ABC, which uses the fixed parameters *t* and *limit* optimized on 3-colorable random graphs, can obtain good results on 3-colorable random graphs but is not applicable to more general problems.

The performance of D-FA is not improved and is still the worst in all five candidate algorithms. The double loop design consumes too much evaluation in one loop so that it is very difficult for D-FA to find a solution in a limited *MaxEval*.

Finally, as we have seen before, graph instances in Experiment PT are harder than those in Theory PT. This is a common phenomenon in 3-colorable, 4-colorable and 5-colorable random graphs.

4.3 Summary

In this chapter, we proposed a discrete adaptive ABC (A-ABC) algorithm and studied its performance from many aspects. We first compared A-ABC with ABC using fixed *t* to show the effect of adaptive function. Then, by using various *N* and *c*, we studied scout bee phase carefully and found that scout bee phase is not necessary when solving discrete optimization problems such as GCPs. After that, by observing the convergence regions of A-ABC and other four competitors, we found that A-ABC is a fast and efficient algorithm that can find a solution in a limited *MaxEval*. Finally, we compared the five candidate algorithms on 3-colorable, 4-colorable, and 5-colorable Culberson random graphs with various graph size *n* and edge probability *p*. The results showed that A-ABC is a robust and general algorithm, which can solve GCPs with various sizes and topologies.

We also studied the experiment values and theory values of phase transition and found that experiment values are smaller than theory values.

Chapter 5 Conclusions

5.1 Conclusions of Research

In this work, we proposed two discrete ABC algorithms, S-ABC and A-ABC, and used them to solve GCPs. From the experiments, we draw the following conclusions.

5.1.1 Conclusions of Chapter 3 "Similarity Artificial Bee Colony Algorithm"

In this chapter, we compared S-ABC, D-FA, and HDPSO on 3-colorable Minton random graphs with various graph size n and constraint density d. The conclusions of this chapter are as follows:

- By introducing Similarity, S-ABC obtains higher SR and lower AES than HDPSO. This shows that S-ABC is more effective than HDPSO when solving larger graph instances.
- D-FA obtains the highest SR in the three candidate algorithms, but it consumes much more evaluation times than the other two.
- Phase transition's experiment values are smaller than its theory values in the context of 3-colorable Minton random graphs.

5.1.2 Conclusions of Chapter 4 "Adaptive Artificial Bee Colony Algorithm"

In this chapter, we studied A-ABC from many aspects on three classes of Culberson random graphs (arbitrary-random graphs, equi-partite graphs, and flat graphs). The conclusions of this chapter are as follows:

- SR is improved when using adaptive function instead of using predetermined fixed t during the evolution.
- Scout bee phase is not necessary when solving discrete optimization problems.
- A-ABC is a faster and efficient algorithm and can find more solutions than S-ABC, HDPSO, and D-FA if given a relatively small *MaxEval*.
- A-ABC is the second best algorithm when solving 3-colorable Culberson random graphs with various graph size n and edge probability p. This shows that A-ABC is robust.
- A-ABC obtains the best results when solving 4-colorable and 5-colorable Culberson random graphs. This shows that A-ABC is a general algorithm that can be applied to a vast graphs with various topologies.
- Phase transition's experiment values are smaller than its theory values in the context of Culberson random graphs.

5.1.3 Engineering Significance of Research

- We extended Similarity introduced in HDPSO to ABC and developed S-ABC, which outperforms HDPSO on SR and AES both.
- We introduced adaptive function to ABC and developed A-ABC, which is a fast, robust, and general algorithm when solving GCPs.
- We found that scout bee phase is not necessary when solving discrete optimization problems.

5.2 Future Work

The future work is as follows:

- Applying the ideas of Similarity and adaptive function to other swarm intelligence algorithms.
- We removed scout bee phase from ABC in this work. We want to replace it with other local search methods to improve the performance.
- Comparing the proposed methods with other non-swarm intelligence algorithms.
- Try to solve discrete optimization problems such as time-tabling problems and flow-shop problems.

Acknowledge

First, I want to express my sincere gratitude to my advisor Prof. Kanoh Hitoshi (狩野　均) for the continuous support of my Ph.D. research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D. study.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. Yasunaga Moritoshi (安永　守利), Prof Takahashi Daisuke (高橋　大介), Associate Prof. Furukawa Hiroshi (古川　宏), and Dr. Claus Aranha, for their insightful comments and encouragement, but also for the hard question that incented me to widen my research from various perspectives.

My sincere thanks also go to the members of Knowledge System Laboratory. Without their precious support, it would not be possible to conduct this research.

Finally, I express my sincere thanks to my parents for their constant support.

References

[Ali 15] Ali, A. F.: A Hybrid Gravitational Search with Levy Flight for Global Numerical Optimization, Information Sciences Letters, vol.4(2), pp.71-83 (2015)

[Aoki 15] Aoki, T., Aranha, C., and Kanoh, H.: PSO Algorithm with Transition Probability Based on Hamming Distance for Graph Coloring Problem, IEEE International Conference on Systems, Man, and Cybernetics, pp1956-1961 (2015)

[Appel 77] Appel, K. and Haken, W.: Every Planar Map is Four Colorable. Part 1: Discharging, Illinois Journal of Mathematics, vol.21(3), pp.429-490 (1977)

[Appel 77] Appel, K. and Haken, W.: Every Planar Map is Four Colorable. Part 2: Reducibility, Illinois Journal of Mathematics, vol.21(3), pp.491-567 (1977)

[Aranha 17] Aranha, C., Toda, H., and Kanoh, H.: Solving the Graph Coloring Problem Using Cuckoo Search, Tan Y., Takagi H., Shi Y. (eds) Advances in Swarm Intelligence. Lecture Notes in Computer Science, vol 10385, pp.552-560 (2017)

[Azad 11] Azad, Sina. K. and Azad Saeid. K.: Optimum Design of Structures Using an Improved Firefly Algorithm, International Journal of Optimization in Civil Engineering, vol. 1(2), pp327-340 (2011)

[Brown 07] Brown C.T., Liebovitch, L.S., and Glendon, R.: Lévy Flights in Dobe Ju/'hoansi Foraging Patterns, Human Ecology, vol.35(1), pp.129-138 (2007)

[Basu 11] Basu, B. and Mahanti, G. K.: Firefly and Artificial Bees Colony Algorithm for Synthesis of Scanned and Broadside Linear Array Antenna, Progress in Electromagnetic Research B., vol. 32, pp169-190 (2011)

[Beni 93] Beni, G. and Wang, J.: Swarm Intelligence in Cellular Robotic Systems, Robots and Biological Systems: Towards a New Bionics, vol. 102, pp703-712 (1993)

[Brelaz 79] Brelaz, D.: New Methods to Color the Vertices of a Graph, Communications of the ACM, vol.22(4), pp.251-256 (1979)

[Caramia 08] Caramia, M. and Olmo, P.: Coloring Graphs by Iterated Local Search Traversing Feasible and Infeasible Solutions, Discrete Applied Mathematics, vol.156(2), pp.201-217 (2008)

[Cham 87] Chams, M., Hertz, A., and Werra, D.: Some Experiments with Simulated Annealing for Coloring Graphs, European Journal of Operational Research, vol.32(2), pp.260-266 (1987)

[Cheeseman 91] Cheeseman, P., Kanefsky, B., and Taylor, W.M.: Where the Really Hard Problems Are, The 12th International Joint Conference on Artificial Intelligence, vol.1, pp.331-337 (1991)

[Cook 71] Cook, S.: The Complexity of Theorem-Proving Procedures, In Proceedings of the Third Annual ACM Symposium on Theory of Computing, STOC, pp.151-158 (1971)

[Cuevas 12] Cuevas, E., Sencion-Echauri, F., Zaldivar, D., and Perez-Cisneros, M.: Multi-Circle Detection on Images Using Artificial Bee Colony (ABC) Optimization, Soft Computing, vol. 16(2), pp281-296 (2012)

[Cui 08] Cui, G. Z., Qin, L., Liu, S., Wang, Y. F., Zhang, X. C., and Cao, X. H.: Modified PSO Algorithm for Solving Planar Graph Coloring Problem, Progress in Natural Science, vol. 18(3), pp353-360 (2008)

[Culberson 96] Culberson, J. and F. Luo: Exploring the k-Colorable Landscape with Iterated Greedy, In Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, American Mathematical Society, pp. 245-284 (1996)

[Davis 91] Davis, L.: Order-Based Genetic Algorithms and the Graph Coloring Problem, Handbook of Genetic Algorithms, New York: Van Nostrand Reinhold, pp.72-90 (1991)

[Dorne 98] Dorne, R. and Hao, J.K.: A New Generic Local Search Algorithm for Graph Coloring, Parallel Problem Solving from Nature, LNCS, vol.1498, Springer, Berlin, Heidelberg (1998)

[Dunstan 75] Dunstan, F.: Greedy Algorithm for Optimization Problems, Presented at Euro I Meeting, Brussels, Jan, 1975.

[Eiben 98] Eiben, A.E., Hauw, J.K, and Hemert, J.I.: Graph Coloring with Adaptive Evolutionary Algorithms, Journal of Heuristics, 4, pp.25-46 (1998)

[Fleurent 96] Fleurent, C. and Ferland, J.A.: Genetic and Hybrid Algorithms for Graph Coloring, Annals of Operations Research, vol.63(3), pp.437-461 (1996)

[Fister 12] Fister, I. Jr, Fister, I., and Brest, J.: A Hybrid Artificial Bee Colony Algorithm for Graph 3-Coloring, Swarm and Evolutionary Computation, LNCS, vol. 7269, pp66-74 (2012)

[Fister 12] Fister, I. Jr, Yang, X. S., Fister, I., and Brest, J.: Memetic Firefly Algorithm for Combinatorial Optimization, Bioinspired Optimization Methods and Their Applications (2012)

[Galinier 99] Galinier, P. and Hao, J. K.: Hybrid Evolutionary Algorithms for Graph Coloring, Journal of Combinatorial Optimization 3, pp.379-397 (1999)

[Galinier 06] Galinier, P. and Hertz, A.: A Survey of Local Search Methods for Graph Coloring, Computers & Operations Research, vol.33, pp.2547-2562 (2006)

[Gandomi 13] Gandomi, A. H., Yang, X. S., and Alavi, A. H.: Cuckoo Search Algorithm: A Metaheuristic Approach to Solve Structural Optimization Problems, Engineering with Computers, vol. 29(1), pp.17-35 (2013)

[Hertz 87] Hertz, A. and Werra, D.: Using Tabu Search Techniques for Graph Coloring, Computing, vol.39(4), pp.345-351 (1987)

[Hsieh 11] Hsieh, T. J. and Yeh, W. C.: Knowledge Discovery Employing Grid Scheme Least Squares Support Vector Machines Based on Orthogonal Design Bee Colony Algorithm, IEEE Transactions on Systems, Man, and Cybernetics, Part B, vol. 41(5), pp.1198-1212 (2011)

[Hsu 11] Hsu, L. Y., Horng, S. J., Fan, P. Z., Khan, M. K., Wang, Y. R., Run, R. S., Lai, J. L.,

and Chen, R. J.: MTPSO Algorithm for Solving Planar Graph Coloring Problem, Expert Systems with Applications, vol. 38(5), pp.5525-5531 (2011)

[Irene 09] Irene, H. S. F., Deris, S., and Hashim, S. Z. M.: University Course Timetable Planning Using Hybrid Particle Swarm Optimization, Genetic and Evolutionary Computation, June 12-14, pp239-246 (2009)

[Kanoh 12] Kanoh, H. and Ochiai, J.: Solving Time-Dependent Traveling Salesman Problems Using Ant Colony Optimization Based on Predicted Traffic, Distributed Computing and Artificial Intelligence, vol. 151, pp25-32 (2012)

[Kanoh 13] Kanoh, H. and Chen, S.: Particle Swarm Optimization with Transition Probability for Timetabling Problems, Adaptive and Natural Computing Algorithms, LNCS, vol. 7824, pp256-265 (2013)

[Kanoh 14] Kanoh, H., Ochiai, J., and Kameda, Y.: Pheromone Trail Initialization with Local Optimal Solutions in Ant Colony Optimization, International Journal of Knowledge-based and intelligent Engineering Systems, vol. 18, no. 1, pp11-21 (2014)

[Karaboga 07] Karaboga, D., Basturk, B.: A Powerful and Efficient Algorithm for Numerical Function Optimization: Artificial Bee Colony (ABC) algorithm, Journal of Global Optimization, vol. 39, pp459-471 (2007)

[Karaboga 11] Karaboga, D. and Ozturk, C.: A Novel Clustering Approach: Artificial Bee Colony (abc) Algorithm, Applied Soft Computing, vol. 11, pp652-657 (2011)

[Karaboga 14] Karaboga, D., Gorkemli, B. and Ozturk, C.: A Comprehensive Survey: Artificial Bee Colony (ABC) Algorithm and Applications, Artificial Intelligence Review, vol.42(1), pp.21-57 (2014)

[Kennedy 95] Kennedy, J. and Eberhart, R.: Particle Swarm Optimization, Proceedings of the IEEE International Conference on Neural Networks, vol. 4, pp1943-1948 (1995)

[Kennedy 97] Kennedy, J. and Eberhart, R. C.: A Discrete Binary Version of the Particle Swarm Algorithm, IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation, vol. 5, pp4104-4108 (1997)

[Kolomvatsos 14] Kolomvatsos, K. and Hadjieftymiades, S.: On the Use of Particle Swarm Optimization and Kernel Density Estimator in Concurrent Negotiations, Information Sciences, vol. 262, pp99-116 (2014)

[Lewis 16] Lewis, R. M. R.: A Guide to Graph Coloring – Algorithms and Applications, eBook ISNB: 978-3-319-25730-3, Springer International Publishing (2016)

[Ma 11] Ma, M., Liang, J. H., Guo, M., Fan, Y., and Yin, Y. L.: SAR Image Segmentation Based on Artificial Bee Colony Algorithm, Applied Soft Computing, vol. 11(8), pp5205-5214 (2011)

[Minasian 13] Minasian, A. A. and Bird, T. S.: Particle Swarm Optimization of Microstrip Antennas for Wireless Communication Systems, IEEE Transactions on Antennas and Propagation, vol. 61, no. 12, pp6214-6217 (2013)

[Minton 90] Minton, S. Johnston, M.D., Philips, A.B., and Laird, P.: Solving Large-Scale

Constraint Satisfaction and Scheduling Problems Using a Heuristic Repair Method, Proc. 8th. Nat. Conf. on A.I., pp.17-24 (1990)

[Minton 92] Minton, S., Johnson, M.D., Philips, A.B., and Laird, P.: Minimizing Conflicts: A Heuristic Repair Method for Constraint-Satisfaction and Scheduling Problems, Artificial Intelligence, vol.58, pp.161-205 (1992)

[Nedic 14] Nedic, N., Prsic, D., Dubonjic, L., Stojanovic, V., and Djordjevic, V.: Optimal Cascade Hydraulic Control for a Parallel Robot Platform by PSO, International Journal of Advanced Manufacturing Technology, vol. 72, no. 5-8, pp1085-1098 (2014)

[Pan 11] Pan, Q. K., Tasgetiren, M. F., Suganthan, P. N., and Chua, T. J.: A Discrete Artificial Bee Colony Algorithm for the Lot-Streaming Flow Shop Scheduling Problem, Information Sciences, vol. 181(12), pp2455-2468 (2011)

[Peksen 14] Peksen, E., Yas, T., and Kiyak, A.: 1-D DC Resistivity Modeling and Interpretation in Anisotropic Media Using Particle Swarm Optimization, Pure and Applied Geophysics, vol. 171, no. 9, pp2371-2389 (2014)

[Rao 09] Rao, S.S.: Engineering Optimization: Theory and Practice, John Willey & Sons, New Jersey (2009)

[Sayadi 10] Sayadi, M., Ramezanian, R., and Ghaffari-Nasab,. N.: A Discrete Firefly Meta-Heuristic with Local Search for Make Span Minimization in Permutation Flow Shop Scheduling Problems, International Journal of Industrial Engineering Computations, vol. 1(1), pp1-10 (2010)

[Shah 11] Shah, H., Ghazali, R. and Nawi, N. M.: Using Artificial Bee Colony Algorithm for MLP Training on Earthquake Time Series Data Prediction, Journal of Computing, vol. 3, no. 6, pp135-142 (2011)

[Tassopoulos 12] Tassopoulos, I. X. and Beligiannis, G. N.: A Hybrid Particle Swarm Optimization Based Algorithm for High School Timetabling Problems, Applied Soft Computing, vol. 12(11), pp3472-3489 (2012)

[Turner 88] Turner, J.S.: Almost All k-colorable Graphs Are Easy to Color, Journal of Algorithms, vol.9(1), pp.63-82 (1988)

[Williams 92] Williams, C.P. and Hogg, T.: Using Deep Structure to Locate Hard Problems, AAAI, pp.472-477 (1992)

[Yang 09] Yang, X. S.: Firefly Algorithms for Multimodal Optimization, Stochastic Algorithms: Foundations and Applications, LNCS, 5792: 169-178 (2009)

[Yang 09] Yang, X. S. and Deb, S.: Cuckoo Search via Levy Flights, World Conference on Nature & Biologically Inspired Computing, no.37, 210-214 (2009)