# A Framework for Parallel Ant Colony Optimization

Jin Wang


Kongens Lyngby 2013
IMM-M.Sc.-2013-66

Student Number: s111376
Supervisor: Carsten Witt

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-M.Sc.-2013-66

# Abstract

With this thesis, we implemented a parallel ant colony optimization framework in Java and studied its performance by a series of experiments. The thesis consists of five chapters.

The first chapter contains some background knowledge. Beginning with the origin of the evolutionary algorithm (EA), an existent island model, the parallel (1+1) EA with migration, was introduced. Then the ant colony optimization (ACO) and the MAX-MIN ant system (MMAS) which is an instance of ACO are covered.

In the second chapter, the detailed parallel ant colony optimization, namely the MMAS* island model, is defined after introducing the MMAS* algorithm and a general parallel ACO algorithm with migration. Then the migration topologies and benchmark problems which will be used in MMAS* island algorithm is presented.

The third chapter shows the implementation of the whole algorithm in Java. The implementation contains a graphical user interface (GUI) which allows the user to set common parameters including all topologies and problems mentioned in second chapter. The whole process of the algorithm can be observed in a graphical representation. In addition, it is also possible to set up and run experiments. The program will collect statistics from the result automatically.

The fourth chapter deals with the experiments regarding the performance of the parallel ACO. The first two experiments are on the LEADINGONESLEADINGZEROS problem which is a synthetic problem that designed to show the benefit of parallel evolution strategies. The experimental result shows that it is likely that low evaporation rate has a same effect with low migration interval on the success rate. The third experiment is an attempt of different evaporation rates on different islands on ONEMAX problem. It shows that the disadvantage on late generations ruins the advantage on early generations.

Finally, the conclusion chapter gives a summary of the whole study.

# Table of Contents

# 1. Introduction

In the first chapter, we first review some basic knowledge of Evolution Algorithms. Then an existent island model, parallel (1+1) EA with migration, was introduced. This algorithm brings an inspiration that applying similar method on Ant Colony Optimization (ACO) instead of EA. Therefore, we focused on the ACO, especially a class called MAX-MIN Ant System (MMAS).

## 1.1. Evolutionary Algorithms

Evolutionary algorithms (EAs) are stochastic optimization techniques based on the principles of natural evolution.[1] They are used in optimization problems and search problems. For such a problem, some individuals which represent candidate solutions of the original problem are stochastically generated as the population. Besides, an evaluation function that reflects the requirements of the problem is necessary to give scores to evaluate individuals. Then the evolution happens in generations. In each generation, some individuals are stochastically selected as parents, and then some children are generated based on parents' information. The two basic methods to generate children is mutation which uses one parent's information and recombination which uses two parents' information. Then some individuals are selected to be the new population in next generation. Due to the "survival of the fittest" concept, an individual with higher quality usually has higher probability to be chosen in the new population as well as the parents, though weaker individuals also have a small chance to be chosen. Note that in some algorithms, only the best individuals will be chosen.

In conclusion, a general flow of evolutionary algorithms can be represented in following steps:

1 Initialize a population.
2 Doing following sub-steps until the termination condition is satisfied.
  2.1 Select one or more individuals (parents).
  2.2 Generate one or more individuals (children, offspring) from parents.
  2.3 Select individuals to form a new population.

Most evolutionary algorithms can be classified in four paradigms. They differ by how the problem is encoded to individuals (the search spaces) and the operator used to generate new individuals.

- Evolution Strategies (ESs) were first presented by Rechenberg in paper [8] (Germany). ES is usually used to solve continuous optimization problems. For the operator, mutation is sometimes the unique reproductive operator used in ES, though it is not rare to also consider recombination.[1] The most important strategies are called (μ, λ)-ES and (μ+λ)-ES and differ from each other by the chosen selection method.[5] (μ, λ)-ES has a population of μ and generates λ children in each generation. Then μ individuals are selected from

λ children to form the next population. On the other hand, (μ+λ)-ES will select μ individuals from both original population and children to form the next population.

- Genetic algorithms (GAs) were first presented by Holland in paper [9]. GA is used to solve problems in discrete search spaces. In GA, bit-strings of length n are used to represent possible solutions. Different from ES, GA uses recombination (cross over) as its primary operator. The idea is the different part of the best solution can be found in different individuals and they can be combined to create better solutions. Additionally, mutation is also considered to introduce new variety to the population.
- Evolutionary programming (EP) was first presented by Fogel, Owens and Walsh in paper [10]. EP focuses in the adaption of individuals rather than in the evolution of their genetic information.[1] For the operator, EP usually use only mutation without recombination. Traditionally, each individual of a population of size μ produces a child by mutation in each generation. The new population consists of μ individuals from 2μ individuals of parents and children. The individuals with higher fitness value have more chance to be selected (e.g., fitness-proportional selection).
- Genetic Programming (GP) was first presented by Koza in paper [11]. For the search space, GP tries to search for an optimal computer program. Therefore the individuals of GP are candidate computer programs. Such programs are typically encoded by trees. The fitness of a program is determined by its performance under some test cases. For the operator, both mutation and recombination are used. Then the next generation is usually selected by fitness-proportional selection.

EAs have already been widely used in many domains.[1] For example, classical NP-hard problems such as the Travelling Salesman Problem[13], Max Independent Set [14], telecommunication problems such as digital data network design[15], predicting bandwidth demands in ATM networks[16], electronics and engineering problems such as power planning [17], circuit design [18].

## 1.2. Island Model

The island model is a parallel evolutionary algorithm. In paper [2],[3], the authors defined island model as the parallel (1+1) EA with migration. By contrast, the panmictic (μ+1) EA (Algorithm 1) is a simple algorithm that in each generation selects a parent uniformly at random and generates a child by mutation, where panmictic means that the population forms one group and all individuals could be chosen as parent. The child replaces one of the worst individuals in the population, unless it is inferior to all individuals in the population.

The following two algorithms' description is taken from paper [2]. $\boldsymbol{t}$ is the number of generation. $\boldsymbol{P_t}$ represents the population at generation $\boldsymbol{t}$. $\boldsymbol{\mu}$ is the number of individuals in the population. $\boldsymbol{x, y, z}$ are individuals. $\boldsymbol{f(x)}$ is the evaluation function.

**Algorithm 1** Panmictic (μ+1) EA

Let $\boldsymbol{t} \coloneqq \mathbf{0}$ and initialize $\boldsymbol{P_0}$ with $\boldsymbol{\mu}$ individuals chosen uniformly at random.
**repeat**
  Choose $\boldsymbol{x} \in \boldsymbol{P_t}$ uniformly at random.
  Create $\boldsymbol{y}$ by flipping each bit in $\boldsymbol{x}$ independently with probability $\boldsymbol{1/n}$.
  Choose $\boldsymbol{z} \in \boldsymbol{P_t}$ with worst fitness uniformly at random.
  **if** $\mathbf{f}(\mathbf{y}) \geq \mathbf{f}(\mathbf{z})$ **then** $\boldsymbol{P_{t+1}} = \boldsymbol{P_t} \setminus \{\boldsymbol{z}\} \cup \{\boldsymbol{y}\}$
    **else** $\boldsymbol{P_{t+1}} = \boldsymbol{P_t}$.
  Let $\boldsymbol{t} = \boldsymbol{t} + \mathbf{1}$.

Let $P = P^1 \dot{\cup} P^2 \dot{\cup} \ldots \dot{\cup} P^k$ be a partition of the whole population in multiple subpopulations, also referred to as islands. Further assume that there is a so-called migration topology, given by a directed graph. Islands represent vertices of the topology and directed edges indicate neighborhoods between the islands. Algorithm 2 represents a parallel EA, where $k$ subpopulations $P^i$, $i = 1, 2, \ldots, k$ evolve independently as in the (μ+1) EA from Algorithm 1, except for special migration generations. Every $\tau$ generations, migrants from one island, in this case copies of the island's best individual, are sent to all islands that are connected in the migration topology via a directed edge. The incoming migrants are included into the island using the same selection as in the panmictic (μ+1) EA: for each subpopulation, the received individual of highest fitness replaces a worst individual on the island, unless the former is inferior to all individuals on the island.

**Algorithm 2** Parallel EA with migration

Let $\boldsymbol{t} \coloneqq \mathbf{0}$.
For all $\boldsymbol{1 \leq i \leq k}$ initialize $\boldsymbol{P_0^i}$ uniformly at random.
**repeat**
  For all $\boldsymbol{1 \leq i \leq k}$ do in parallel
    **if** $\boldsymbol{t}$ **mod** $\boldsymbol{\tau = 0}$ and $\boldsymbol{t > 0}$ **then**
      Send an individual with maximum fitness in $\boldsymbol{P^i}$ to all neighbored populations.
      Let $\boldsymbol{y^i}$ be an individual with maximum fitness among all incoming migrants.
    **else**
      Choose $\boldsymbol{x^i} \in \boldsymbol{P_t^i}$ uniformly at random.
      Create $\boldsymbol{y^i}$ by flipping each bit in $\boldsymbol{x^i}$ independently with probability $\boldsymbol{1/n}$.
    Choose $\boldsymbol{z^i} \in \boldsymbol{P_t^i}$ with worst fitness uniformly at random.
    **if** $\boldsymbol{f(y^i) \geq f(z^i)}$ **then** $\boldsymbol{P_{t+1}^i = P_t^i \setminus \{z^i\} \cup \{y^i\}}$
      **else** $\boldsymbol{P_{t+1}^i = P_t^i}$.
  Let $\boldsymbol{t} = \boldsymbol{t} + \mathbf{1}$.

The value $\boldsymbol{\tau}$ is called migration interval. If $\boldsymbol{\tau} < \infty$ and all subpopulations have size 1, this is called the parallel (1+1) EA with migration or shortly island model.[3] It is shown in paper [2] (theoretical) and [3] (experimental) that for an example function that parallelism and just the right amount of migration between subpopulations can be

essential.

The island model can be extended by using an arbitrary basic evolutionary algorithm in sub-population, using different migration topology, migration rate (how many individuals are going to migrate in one migration), and the migration interval.

## 1.3. Ant Colony Optimization

Ant colony optimization (ACO) is another bio-inspired approach to solve optimization problems. In contrast to EAs, where solutions are constructed from the current set of solutions, solutions are in this case obtained by random walks on a so-called construction graph which is usually a directed graph.[5]

As the name suggests, the basic idea comes from observing the food hunting process among ants. From the beginning, ants perform a random walk, and then they will produce pheromone after finding the food while returning to the colony. Ants trend to choose paths with more pheromone in the random walk. Over time, the pheromone starts to evaporate, and the attraction of the path reduces. For a longer path to the food, it takes more time to complete a trip and the pheromone will evaporate more. On the other hand, a shorter path will be used more frequently and has more pheromone on it. Thus, when an ant finds a short path between colony and food, other ants will trend to follow that path, and then the pheromone will accumulate and at last the ant colony will trend to use the path.

Inspired by the behavior of the ants, a general scheme of the ant colony optimization program can be represented in following steps:

1 Initialize starting pheromone information.
2 Doing following sub-steps until the termination condition is satisfied.
  2.1 Generate a solution.
  2.2 Update the pheromone information depending on the quality of the solution.

The first ACO algorithm, called Ant System (AS) [12], was applied to the classical Traveling Salesman Problem (TSP). The problem gives a list of cities and their pairwise distances. The task is to find the shortest possible route that visits each city exactly once and returns to the origin city. AS initializes the problem by putting $m$ ants on $n$ cities and giving a small pheromone on all edges. Then for each step, ant $k$ moves from current city $i$ to another city $j$ ($j$ must be never visited in current tour or $P_{ij}^{k} = 0$) with probability:

$$P_{ij}^{k} = \frac{[\tau_{ij}]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum_{l \in \text{allowed}_l} [\tau_{il}]^{\alpha} \cdot [\eta_{il}]^{\beta}}$$

where $\tau_{ij}$ is the pheromone on edge $(i, j)$, $\eta_{ij}$ is the "visibility" of edge which equals the reciprocal of the length of edge $(\eta_{ij} = 1/d_{ij})$, $\text{allowed}_l$ is the set of the cities that could be visited in next step. $\alpha$ and $\beta$ are parameters that control the relative importance of pheromone versus visibility. In extreme cases, if $\alpha = 0$, then only visibility is used to determine the probability, which reduced to a greedy heuristic

algorithm. If $\beta = 0$, then only pheromone information is used.

Then after $n$ steps, each ant performs a tour of cities, which is called a generation. The pheromone is modified using following update rule:

$$\tau_{ij}^{'} = \rho\tau_{ij} + \Delta\tau_{ij}$$

where $\boldsymbol{\tau_{ij}^{'}}$ is the new pheromone, $\rho$ is the evaporation rate and the old pheromone is multiplied by $\rho$ to emulate the evaporation of the pheromone. The evaporation rate determines how fast the evaporation proceeds. $\Delta\tau_{ij}$ is the sum of the pheromones made by ants in current generation. For ant k, the pheromone on edge $(i,j)$ is:

$$\Delta\tau_{ij}^{k} = \begin{cases} \dfrac{Q}{L_k} & \text{if ant } k \text{ uses } (i,j) \text{ in current tour} \\ 0 & \text{otherwise} \end{cases}$$

where $Q$ is a constant and $L_k$ is the tour length of the $k^{\text{th}}$ ant.

Then the algorithm runs until a limit number of generations reached.

## 1.4. MAX–MIN Ant System

MAX-MIN ant system (MMAS) was first presented by Stützle and Hoos in 2000. In paper [6], the authors proposed three main differences from an ant system:

1. To exploit the best solutions found during a generation or during the run of the algorithm, after each generation only one single ant adds pheromone. This ant may be the one which found the best solution in the current generation (generation-best ant) or the one which found the best solution from the beginning of the trial (global-best ant).
2. To avoid stagnation of the search the range of possible pheromone trails on each solution component is limited to an interval $[\tau_{\min}, \tau_{\max}]$.
3. The pheromone trails are initialized to $\tau_{\max}$, which encourages a higher exploration of solutions at the start of the algorithm. In paper [7], the authors proposed another method to initialize pheromone which is to distribute uniformly.

More specifically, in each generation, only one ant is used to update the pheromone. The modified pheromone trail update rule is:

$$\tau_{ij}^{'} = \rho\tau_{ij} + \Delta\tau_{ij}^{\text{best}}$$

where $\Delta\tau_{ij}^{\text{best}} = 1/f(s^{\text{best}})$ and $f(s^{\text{best}})$ is the cost of either the generation-best solution or global-best solution. Thus the old pheromone with some evaporation plus a best solution pheromone forms the new pheromone.

# 2. Preliminaries

In this chapter, we will present the MMAS* algorithm and several topologies and problems that will be later implemented in Chapter 3.

## 2.1. The MMAS* Algorithm

The MMAS* algorithm is originally presented in paper [7]. The original algorithm is designed to deal with bit-string problem. Here, we extended it to a more general algorithm that compatible with more problems by replacing the construction graph and the pheromone update scheme.

The solution construction is achieved by a random walk on a construction graph $C = (V, E)$ that each edge has a pheromone value. In paper [7], the procedure is described as follows:

**Construct**$(\boldsymbol{C}, \boldsymbol{\tau})$
1 $v \coloneqq s$, mark $v$ as visited.
2 Let $N_v$ be the set of non-visited successors of $v$ in $C$.
If $N_v \neq \emptyset$ then
2.1 Choose successor $w \in N_v$ with probability $\tau_{(v,w)} / \sum_{(v,u)|u \in N_v} \tau_{(v,u)}$.
2.2 Mark $w$ as visited, set $v \coloneqq w$ and go to 2.
3 Return the solution $x$ and the path $P(x)$ constructed by this procedure.

where $s \in V$ is the starting vertex and $\tau$ is the pheromone values on the edges ($\tau: E \to \mathbb{R}$). The procedure **Construct**$(\boldsymbol{C}, \boldsymbol{\tau})$ is a general procedure that can be used to generate different kinds of solutions based on graph $C$ which will be introduced together with the problems in Section 2.4 - 2.6. We will also see how the solution construct procedure is modified to adjust different problems.

More specifically, the pheromone values are initialized to 1/2 and the global best solution is used to update the pheromone. Different to the MMAS algorithm which accepts solutions which have same fitness value as the current best solution in paper [7], the MMAS* algorithm only accepts solutions that are strictly better than the current best solution. The algorithm is shown as follows:

1 Set $\tau_{(u,v)} = 1/2$ for all $(u, v) \in E$.
2 Create $x^*$ using **Construct**$(\boldsymbol{C}, \boldsymbol{\tau})$.
3 Update pheromones with respect to $x^*$.
4 Repeat:
4.1 Create $x$ using **Construct**$(\boldsymbol{C}, \boldsymbol{\tau})$.
4.2 If $f(x) > f(x^*)$, then $x^* = x$.
4.3 Update pheromones with respect to $x^*$.

## 2.2. An Island Model for MMAS*

Based on the original island model defined in paper [2], [3], we first propose a parallel

ACO with migration. Then the MMAS* island model is obtained by specializing the parallel ACO with migration.

### 2.2.1. A Parallel ACO with Migration

Let $A = A^1 \dot{\cup} A^2 \dot{\cup} \ldots \dot{\cup} A^k$ be a partition of the whole ant colony in multiple sub ant colonies, also referred to as islands. Note that each sub ant colony $A^i$ has its own construction graph $C^i$. After the initialization, the solution generating and pheromone updating process of sub ant colonies are based on the construction graph as in the normal ACO except for the migration generations.

For the migration generations, a migration topology is given by a directed graph. Islands represent vertices of the topology and directed edges indicate neighborhoods between the islands. When the migration occurs, some information about good solutions are sent to all islands that are connected in the migration topology via a directed edge. The information can be the best solutions so far and/or the pheromone value in the construction graph. The incoming migrants are gathered to get the best solution or the best solutions depending on how many solutions are generated in a normal generation. Then the migrants are treated same as generated solutions in normal generations. In addition, if the migrants includes the pheromone value information and the best migrant is better than the best so far solutions, the pheromone value from the best migrant will replace the original pheromone value of the sub ant colonies.

The procedure of the algorithm is described as follows:

For all $1 \leq i \leq k$ do in parallel
1 Initialize starting pheromone information in $A^i$.
2 Doing following sub-steps until the termination condition is satisfied.
  2.1 If this is a normal generation,
    2.1.1 Generate $m$ solutions from construction graph $C^i$.
  2.2 If this is a migration generation,
    2.2.1 Send the information about best solution in $A^i$ to all neighbored ant colonies.
    2.2.2 Gather solutions and select $m$ best solutions among them.
  2.3 Update the pheromone information depending on the quality of the solution.

### 2.2.2. The MMAS* Island Model

The MMAS* island model is obtained by applying MMAS* on previous parallel ACO with migration plus some specified details.

First, as extended from MMAS*, the solution construction procedure **Construct**$(\boldsymbol{C}, \boldsymbol{\tau})$, the construction graph $\boldsymbol{C}$ and the pheromone update scheme are same with MMAS* for each sub ant colony. See section 2.1 for the solution construction procedure and section 2.4 - 2.6 for problem specific construction graphs and pheromone update schemes with the notations used here.

Note that the evaporation rate $\boldsymbol{\rho}$ can be initialized by different values in different

sub ant colonies, which brings more variance to the whole colony. It is possible to change the evaporation rate in migration generations if the incoming evaporation rate is "better". But considering introducing the different $\boldsymbol{\rho}$ in different sub ant colonies is to introduce more variance to the whole colony, but changing $\boldsymbol{\rho}$ will trend to unify the evaporation rate among sub ant colonies, which cancel out the benefit of variance in the whole colony. Therefore in this MMAS* island model, the evaporation rate will not be sent out in the migration generations, so it is constant for each sub ant colony.

Then the migration generation is set to occur every $\boldsymbol{\lambda}$ generations.

As the MMAS* only saves the global best solution, it is the only solution that can be sent in the migration generation. In addition, the pheromone value of the construct graph is also sent out. Then if the best migrant solution is better than the best so far solution, replace the pheromone value with the corresponding incoming pheromone value. As the incoming pheromone value might have frozen or updated with respect to the corresponding solution for several generations, it is more related to the corresponding best so far solution.

At last, Algorithm 3 describes the MMAS* island model. The notation with superscript $\boldsymbol{i}$ represents the item in $\boldsymbol{i}$-th sub ant colony. $x^*$ is the best so far solution. $\boldsymbol{t}$ is the number of generation. $\boldsymbol{k}$ is the number of sub ant colonies. $\boldsymbol{f}(\boldsymbol{x})$ is the evaluation function.

Algorithm 3 Parallel MMAS* with migration

Let $\boldsymbol{t} \coloneqq \mathbf{0}$.
**For all** $\mathbf{1} \leq \boldsymbol{i} \leq \boldsymbol{k}$
    Set $\boldsymbol{\tau}_{(u,v)} = \mathbf{1/2}$ for all $(\mathbf{u}, \mathbf{v}) \in \boldsymbol{C^i}$.
    Initialize $\boldsymbol{\rho^i}$
    Create $\boldsymbol{x^{*i}}$ using **Construct**$(\boldsymbol{C^i}, \boldsymbol{\tau^i})$.
    Update $\boldsymbol{\tau^i}$ with respect to $\boldsymbol{x^{*i}}$.
**repeat**
    **For all** $\mathbf{1} \leq \boldsymbol{i} \leq \boldsymbol{k}$ do in parallel
        **if** $\boldsymbol{t} \bmod \boldsymbol{\lambda} = \mathbf{0}$ and $\boldsymbol{t} > 0$ **then**
            Send pair $(\boldsymbol{x^{*i}}, \boldsymbol{\tau^i})$ to all neighbored populations.
            Let $(\boldsymbol{x^i}, \boldsymbol{\tau})$ be the pair that $\boldsymbol{x^i}$ has the maximum fitness among all incoming migrant pairs.
            **if** $\boldsymbol{f}(\boldsymbol{x^i}) > \boldsymbol{f}(\boldsymbol{x^{*i}})$ **then** $\boldsymbol{\tau^i} = \boldsymbol{\tau}$.
        **else**
            Create $\boldsymbol{x^i}$ using **Construct**$(\boldsymbol{C^i}, \boldsymbol{\tau^i})$.
        **if** $\boldsymbol{f}(\boldsymbol{x^i}) > \boldsymbol{f}(\boldsymbol{x^{*i}})$ **then** $\boldsymbol{x^{*i}} = \boldsymbol{x^i}$
        Update $\boldsymbol{\tau^i}$ with respect to $\boldsymbol{x^{*i}}$.
    Let $\boldsymbol{t} = \boldsymbol{t} + \mathbf{1}$.

In some special cases, Algorithm 3 is equivalent to some other parallel algorithms. For example, if all $\boldsymbol{\rho}$ are big enough that the pheromone value can only be maximum or minimum value, then the pheromone of edges on the current best solution will be

$1 - 1/\boldsymbol{n}$ and pheromone of other edges will be $1/\boldsymbol{n}$. Therefore for each generation, all bits have a probability of $1/\boldsymbol{n}$ to be different from the current best solution, which is equivalent to the Algorithm 2 Parallel EA with migration that flipping bits with probability $1/\boldsymbol{n}$. Furthermore, if all $\boldsymbol{\rho}$ are big enough and the migration interval $\boldsymbol{\lambda}$ is infinity, then the migration will never happen and the algorithm becomes $k$ Algorithm 1 Panmictic (μ+1) EAs running at same time, which is called independent parallel (μ+1) EA.

## 2.3. Migration Topologies

There are five special topologies that will be mentioned in the implementation and experiments. They were used in the experiments in paper [2]. The topologies are described in the following. Note that all edges in the graph are bidirectional.

- Empty: If the topology is an empty graph, no migration will happen in the migration generation. Thus the island model becomes parallel independent MMAS*. Here the empty graph is used to compare with other topologies to show the necessity of the migration.
- Ring: A ring is a topology that all islands are connected to be a circle. A sample ring of 8 islands is shown in **Figure 1**.

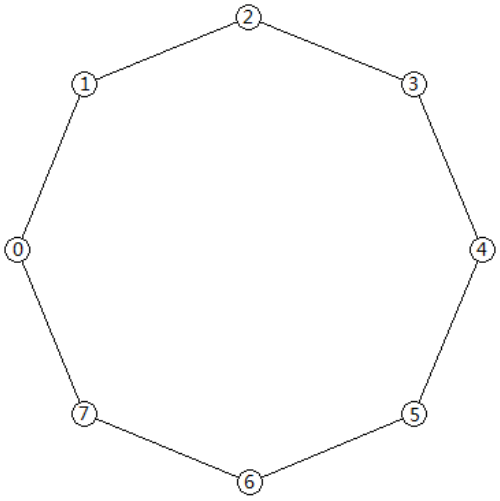



**Figure 1** Ring of 8 islands.

- Torus: In this paper, “torus” is a short notation for two-dimensional $(n, d)$-torus. A two-dimensional $(n, d)$-torus is a $n$-rows and $d$-columns grid with edges wrapping around on all sides. A sample $(3,4)$-torus of 12 islands is shown in **Figure 2**.
- Hypercube: A hypercube topology has a limitation that the number of islands $m = 2^k, k \in \mathbb{N}$. Numbering islands from $0$ to $m - 1$, then two islands $u$ and $v$ are connected if and only if $u\ XOR\ v = 2^l, l \in \mathbb{N}$. In other words, writing the island number in binary, then two islands are connected if and only if there is exactly one digits of the island number is different. A sample hypercube of 8 island is shown in **Figure 3**.

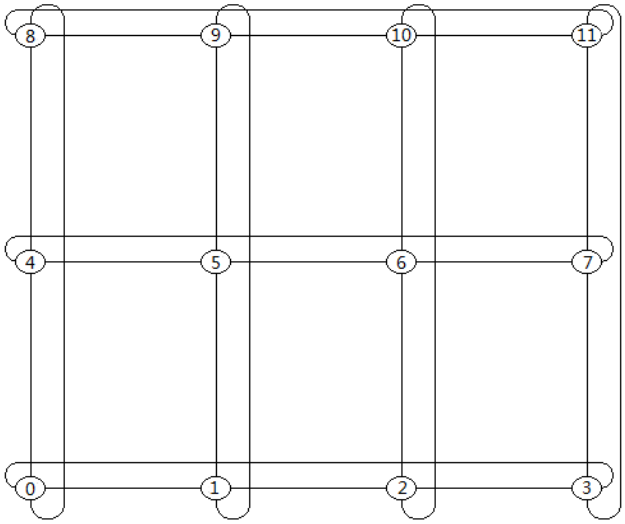



**Figure 2** (3,4)-torus of 12 islands.

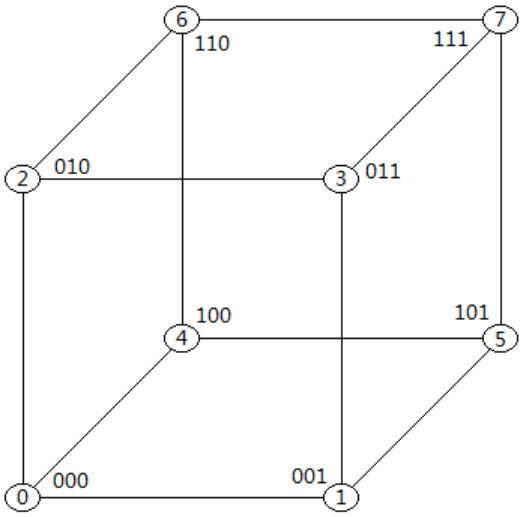



**Figure 3** Hypercube of 8 islands.

- Complete: If the topology is a complete graph, the best solution will take over all islands in every migration generation. Thus, the best solution is spread in the fastest way but the diversity is decreased. Furthermore, if the best solution is a local optimum, all islands are trend to be stuck at same local optimum. Here the complete graph is an extreme sample to compare with other migration topologies.

## 2.4. $\{0,1\}^n$ Optimization Problems

A $\{0,1\}^n$ optimization problem is to find a bit-string $x \in \{0,1\}^n$ that has the maximum or minimum fitness value in a given fitness function $f: \{0,1\}^n \to \mathbb{R}$. Two benchmark problems ONEMAX, LEADINGONES and a synthetic problem LEADINGONESLEADINGZEROS are mentioned.

## 2.4.1. Problem Description

The notation $x_i$ is used to represent the $i$-th bit of the bit-string $x$, i.e. $x = x_1 x_2 \dots x_n$ where $n$ is the length the string. All following problems are finding the solution with the maximum fitness value.

### 2.4.1.1. OneMax

OneMax (OM) is a theoretical benchmark problem that counts ones in the string. The fitness function is defined as:

$$\mathrm{OM}(x) = x_1 + x_2 + \cdots + x_n$$

The bit-string may fix an arbitrary zero bit to make a progress.

### 2.4.1.2. LeadingOnes

LeadingOnes (LO) is another theoretical benchmark problem that counts leading ones in the string. The fitness function is defined as:

$$\mathrm{LO}(x) = \sum_{i=1}^{n} \prod_{j=1}^{i} x_j$$

In contrast to OneMax, the bit-string $x$ must keep the leading ones and fix the first zero bit in the meanwhile to make a progress during the process of optimizing LeadingOne.

### 2.4.1.3. LeadingOnesLeadingZeros

LeadingOnesLeadingZeros (LOLZ) is a synthetic problem that first proposed in paper [2]. It is designed to show the benefit of parallel evolution strategies.

The bit-string $x$ of length $n$ is divided into $b$ blocks of length $l$. For simplicity, we assume $bl = n$ so there is no suffix outside the last block. In the first block, either leading ones or leading zeros contribute to the fitness value and both sides lead to an equal gain in fitness for each leading bit. But after $z$ leading bits has been reached, only leading ones can lead to a larger fitness. In other blocks, the leading ones and zeros contribute to the fitness value in the same way as the first block but only if all previous blocks are all-ones string.

Formally, from the definition in paper [2], let $z, b, l \in \mathbb{N}$ such that $bl = n$ and $z < l$. For a bit string $x = x_1 x_2 \dots x_n$ we abbreviate $x_{il+1} x_{il+2} \dots x_{(i+1)l}$ by $x^{(i)}$. Let $LO(x)$ represents the fitness function of LeadingOnes same as previous subsection and $LZ(x)$ represents the fitness function of LeadingZeros that $\mathrm{LZ}(x) = \sum_{i=1}^{n} \prod_{j=1}^{i} (1 - x_j)$. Then the fitness function is defined as:

$$LOLZ_{n,z,b,l}(x) = \sum_{i=1}^{b} \left( \left[ LO\left(x^{(i)}\right) + \min\left(z, LZ\left(x^{(i)}\right)\right) \right] \cdot \prod_{j=1}^{(i-1)l} x_j \right)$$

Thus the blocks must be optimized one by one, from left to right. A string with $z$ leading zeros in the current optimizing block represents a local optimum. If $z$ is large enough, it becomes nearly impossible to escape from this local optimum.

For such a fitness function, a population has a 1/2 probability to gather leading

ones and a 1/2 probability to gather leading zeros at the beginning of each block. It is obvious that a panmictic population only has a possibility of about $2^{-b}$ to select correct bits to collect at every block and finally get a global optimum. The possibility will decrease exponentially as the number of block increases. The separate subpopulations without migrations also need exponential time to find the optimal solution since the subpopulations cannot compliment to an exponentially low success possibility. This intuitive idea was formally proved in paper [2].

In contrast, an island model with properly configured migration can contribute to the optimization. As the islands choose to collect leading ones or zeros independently, some of the islands will stuck at leading zeros while other islands have overcome the threshold with leading ones and have a higher fitness value. If a migration happens in such situations, the solutions with higher fitness value that collect leading ones will replace the local optimum solutions that collect leading zeros. Thus the stuck islands once again have chance to be optimized. In paper [2], the authors proved that it only needs polynomial time to find the optimal solution with overwhelming probability under a properly configured island model. Furthermore, the theoretical result has been experimentally confirmed in paper [3] by the same authors.

## 2.4.2. Solution Construction

In the case of bit-string, paper [19] presents a directed graph called “chain” which is shown in **Figure 4**.

For a bit-string of length $n$ to be generated, the corresponding graph will consist of $3n+1$ nodes and $4n$ edges. Then for each bit $x_i (1 \le i \le n)$, $x_i = 0$ if the edge $(v_{3(i-1)}, v_{3(i-1)+1})$ is chosen and $x_i = 1$ if the edge $(v_{3(i-1)}, v_{3(i-1)+2})$ is chosen. In addition, we ensure that $\sum_{(u,\cdot)\in E} \tau_{(u,\cdot)} = 1 \; for \; u = v_{3i}, 0 \le i \le n-1$ . Let $p_i = Prob(x_i = 1)$ be the probability of setting the bit $x_i$ to 1 in the next constructed solution. Thus, $p_i = \tau_{(v_{3(i-1)}, v_{3(i-1)+1})}$ and $1 - p_i = \tau_{(v_{3(i-1)}, v_{3(i-1)+2})}$. After the branch at vertex $v_{3(i-1)}$ for $1 \le i \le n$, the following edges $(v_{3(i-1)+1}, v_{3i})$ or $(v_{3(i-1)+2}, v_{3i})$ has no effect to the solution.

In this paper, the pheromone on the edge $(v_{3(i-1)}, v_{3(i-1)+2})$ which will make bit $x_i = 1$ if chosen is called “the pheromone of $x_i$” for short and clear.

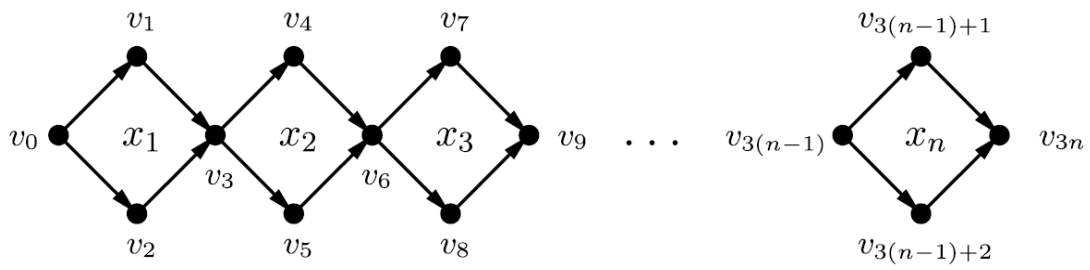



**Figure 4** Construction graph "Chain"

### 2.4.3. Pheromone Update Scheme

By the second feature of MMAS, the pheromone value $\tau_{(u,v)}$ is restricted in interval $[1/n, 1-1/n]$ so all candidate solutions still have a positive probability to be generated. The choice of these values is inspired by standard mutation operators in evolutionary computation where an incorrect bit has a probability of $1/n$ of being corrected.[7]

The pheromone values are updated differently depending on whether the edge is contained in the path $P(x)$. The update formula is shown as follows:

$$\tau'_{(u,v)} = \begin{cases} min\{(1-\rho)\cdot\tau_{(u,v)} + \rho, 1-1/n\}, & \text{if}(u,v) \in P(x), \\ max\{(1-\rho)\cdot\tau_{(u,v)}, 1/n\}, & \text{otherwise.} \end{cases}$$

Note that this update formula keeps the property that $\sum_{(u,\cdot)\in E}\tau_{(u,\cdot)} = 1\ for\ u = v_{3i}, 0 \le i \le n-1$.

## 2.5. Travelling Salesman Problem

### 2.5.1. Problem Description

As mentioned in section 1.3, Travelling salesman problem (TSP) is an important realistic optimization problem that as often used as a practical benchmark problem. Given a list of cities and the distances between each pair of cities, it asks for the shortest possible route that visits each city exactly once and returns to the origin city. In other words, given a weighted complete graph $G = (V, E, w)$, find a Hamilton cycle that has the minimum total weight.

For a solution $x = x_1x_2 \ldots x_nx_1$ where $x_i$ represents the $i$-th visited vertex (city) and $n = |V|$. Let $w(x_i, x_j)$ represent the weight (distance) between two vertices. The fitness function is defined as:

$$\mathrm{TSP(x)} = w(x_n, x_1) + \sum_{\mathrm{i}=1}^{\mathrm{n}-1} w(x_i, x_{i+1})$$

### 2.5.2. Solution Construction

The solution construction procedure of MMAS is same with Ant System in Section 1.3, but here we reform it to the **Construct**$(\boldsymbol{C}, \boldsymbol{\tau})$ plus construction graph scheme.

The construction graph of TSP is rather intuitive. It is simply the copy of original graph, with pheromone value In addition. The solution path generated by **Construct**$(\boldsymbol{C}, \boldsymbol{\tau})$ is a path instead of a cycle. It can be fixed by adding an edge from the ending vertex to the starting vertex.

In contrast to the construction graph “chain” for bit-string problems, the construction graph for TSP has weight information on the edge which should be considered during the solution construction procedure. Therefore in the step 2.1 of

**Construct**$(\boldsymbol{C}, \boldsymbol{\tau})$, the probability of choosing a successor $w \in N_v$ is updated to:

$$P_{ij}^{k} = \frac{[\tau_{(v,w)}]^{\alpha} \cdot [\eta_{(v,w)}]^{\beta}}{\sum_{(v,u)|u \in N_v} [\tau_{(v,u)}]^{\alpha} \cdot [\eta_{(v,u)}]^{\beta}}$$

where $\eta_{(v,w)} = 1/w(v,w)$ is the heuristic affection on the successor selection. $\alpha$ and $\beta$ are constant parameters to balance the weight between pheromone information and heuristic information.

### 2.5.3. Pheromone Update Scheme

The pheromone update scheme for TSP is different from the scheme for bit-string problems. The positive reflection for the edges on the current global best solution is $Q/f(x)$ instead of $\rho$ where $Q$ is a constant and $f(x)$ is the fitness value of the best solution. This update scheme is extended from the original Ant System algorithm from paper [12] with maximum and minimum value limit.

The update formula is shown as follows:

$$\tau'_{(u,v)} = \begin{cases} min\{(1-\rho) \cdot \tau_{(u,v)} + Q/f(x), 1 - 1/n\}, & \text{if}(u,v) \in P(x), \\ max\{(1-\rho) \cdot \tau_{(u,v)}, 1/n\}, & \text{otherwise.} \end{cases}$$

Note that in paper [23], another scheme is available. The positive reflection is $\rho$ and the maximum and minimum pheromone value is set to $1 - 1/n$ and $1/n^2$.

## 2.6. Minimum Spanning Tree

### 2.6.1. Problem Description

Given a connected undirected graph, a spanning tree is a subgraph that is a tree and connects all the vertices together. If the graph is also weighted, the weight of a spanning tree is the sum of the weight of all edges in the tree. Thus, given a connected undirected weighted graph $G = (V, E, w)$, the Minimum Spanning Tree (MST) problem asks for a spanning tree that has the minimum weight.

For a solution $x = \{x_1, x_2, \dots, x_{n-1}\}$ where $x_i$ represents the $i$-th edge in the tree and $n = |V|$. Let $w(x_i)$ represent the weight of edge $x_i$. The fitness function is defined as:

$$\mathrm{MST(x)} = \sum_{\mathrm{i=1}}^{\mathrm{n-1}} w(x_i)$$

### 2.6.2. Solution Construction

There are two candidate construct methods. The first one is using a construction graph same with the given graph $G$, then let an ant perform random walk on it until a valid solution is obtained. It is an intuitive idea but it is proved to be very slow in paper [22]. For a better performance, paper [22] provides another method called Kruskal-based construction procedure. We extended it with additional weight

information to the construction graph. Let the given graph $G = (V, E, w)$ where $|E| = m$ and the construction graph $C = (V', E', w')$. The edges from $G$ (numbering from 1 to m) with a designated starting node $s$ (numbering 0) form the nodes set $V'$ of the construction graph. So $|V'| = m + 1$. The edge set $E'$ is defined as:

$$E' = \{(i, j) | \ 0 \leq i \leq m \text{ and } 1 \leq j \leq m \text{ and } i \neq j\}$$

In the other word, it is a complete graph removing all edges to the starting node $s$ and all self loop edges. The weight of edges in construction graph is equal to the weight of the edge in the original graph that corresponds to the ending vertex. Formally, the weight function $w'$ is defined as:

$$w'(i, j) = w(E_j)$$

In such a construction graph, visiting a node means to select the corresponding edge on the graph $G$.

Then the solution construction procedure **Construct** $(\boldsymbol{C}, \boldsymbol{\tau})$ also need to be modified to adjust the construction graph. In step 2, not only visited nodes are prohibited, the nodes that will form a cycle in the original graph $G$ after selected are also not allowed.

## 2.6.3. Pheromone Update Scheme

We did not use the pheromone update scheme provided by paper [22] directly, but with a change to keep the coordination with other schemes.

In the scheme in paper [22], all edges pointing to nodes from solution $x$ except $s$ are rewarded. In other words:

$$\tau'_{(u,v)} = \begin{cases} h, & \text{if } v \in x \text{ and } v \neq s, \\ l, & \text{otherwise.} \end{cases}$$

where $h$ and $l$ are two constant to represent the high pheromone and low pheromone. The reason to award all edges to nodes in $x$ is to allow the ant to rediscover the edges of the previous spanning tree, i.e. the nodes in construction graph, with high and equal probability from any node.

Here, to keep the coordination with other update schemes, the new pheromone is still increased step by step instead of fixed to $h$ and $l$. The update formula is shown as follows:

$$\tau'_{(u,v)} = \begin{cases} min\{(1-\rho) \cdot \tau_{(u,v)} + Q/f(x), 1 - 1/n\}, & \text{if } v \in x \text{ and } v \neq s, \\ max\{(1-\rho) \cdot \tau_{(u,v)}, 1/n\}, & \text{otherwise.} \end{cases}$$

# 3. Parallel ACO Framework Implementation

In this Chapter, we implemented a program to illustrate the parallel MMAS* algorithm. The implementation is based on Java, JDK version 1.6 update 27. The class diagram in this paper is drawn by Microsoft Visio 2007.

## 3.1. Function Requirements

The following features are expected in the implementation:

- A graphical user interface (GUI) which allows the user to set common parameters such as topology and migration interval.
- Able to solve all problems introduced in Section 2.4 - 2.6.
- A visualization of the optimization procedure including the topology and status of the particular islands such as best-so-far solution.
- Able to run experiments and collect statistics.（Todo: the requirement analysis）

The requirements can be classified into two parts: the MMAS* island model itself and a GUI to control and observe it.

For the MMAS* island model, we first considered a single island. It is simply a new solution generation procedure with a pheromone updating procedure in a generation. So we first designed the classes of solutions and pheromones. Due to the concept of Object-Oriented programming, the solution generation procedure and the pheromone update procedure are included in the pheromone class and the fitness function is included in the solution class. Then we considered the island model. It is intuitive to use thread to simulate the island, with an extra thread to coordinate the islands. Furthermore, a topology class is implemented to support the migration. We selected the shared memory as the communication method between threads as it is the most simple and fast. Therefore, a data class including all solutions, pheromones and public parameters are implemented and shared in all islands. Extra attention is paid to prevent reading or writing dirty data.

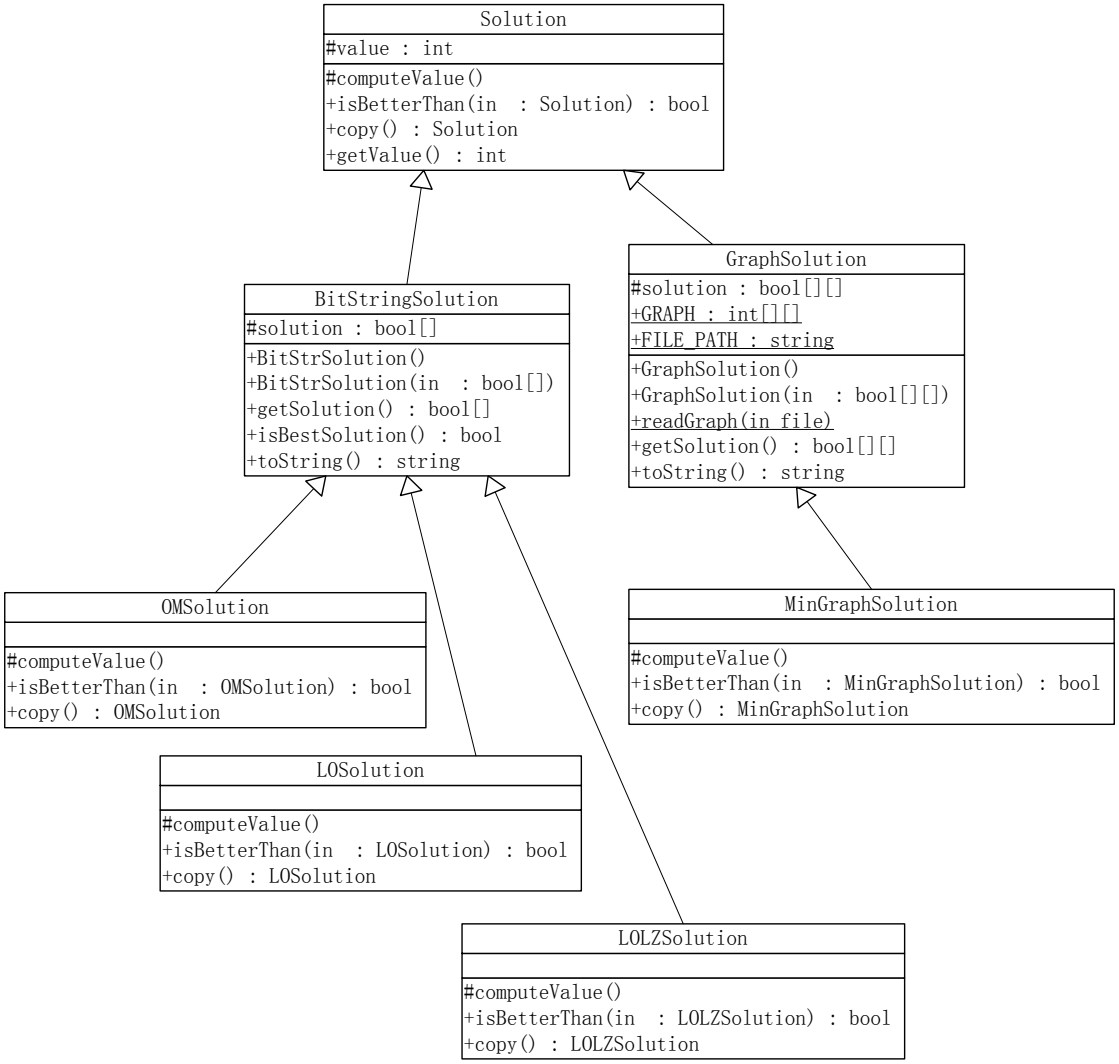
For the GUI, two functions are required. The first function is to allow the user to set parameters of the optimization, which is solved directly by adding some drop-down lists and input text areas. The second function is to visualize the optimization procedure, which is the core function of this software. The third function is to allow user to run experiments, which may contain multiple settings of the parameters and multiple runs of the algorithm. We designed the GUI to be similar to a video-player with four control buttons and a few parameters, which contributes to an intuitive user experience.

The detailed design is introduced in following three sections.

## 3.2. Basic Data Structures

### 3.2.1. Solutions

All solution classes have a same super abstract class *Solution*. It first derived two abstract subclasses *BitStringSolution* and *GraphSolution* for different solution structures. Then they derived four subclasses with fitness function realized. The class diagram is shown as follows:




The base class *Solution* have a instance variable “value” with getter method to represent the fitness value of the solution. There are three abstract methods left for implementation in the derived classes:

- computeValue: Call this method to update the fitness value of the solution. It is used after a new solution is constructed or the solution is updated.
- isBetterThan: Call this method to compare to another solution. It is used in the MMAS* algorithm to determine whether a new solution will replace the best-so-far solution.
- copy: Call this method to make a deep copy of the solution. It is usually used to copy the solution at migrations

The class *BitStringSolution* inherit the base class and add a Boolean array with getter method to represent the bit-string solution. Here a “true” value in the array represents a “one” bit in the string and a “false” represents a “zero”. This class is still

an abstract class as the abstract methods inherited from the base class are not implemented. In addition, the following methods are added in class *BitStringSolution*:

- BitStringSolution: Two constructors are available for the class. The one without any parameter will generate a random bit-string that each bit have 50% possibility to be “one” or “zero”. The one with a given Boolean array will use that array as the solution. Both of the constructors will update the fitness value after the solution is determined.
- isBestSolution: Call this method to determine whether the solution is already the best solution. It is used to check the termination of the optimization in the later experiments.
- toString: Override the default toString method. It is used to output the solution.

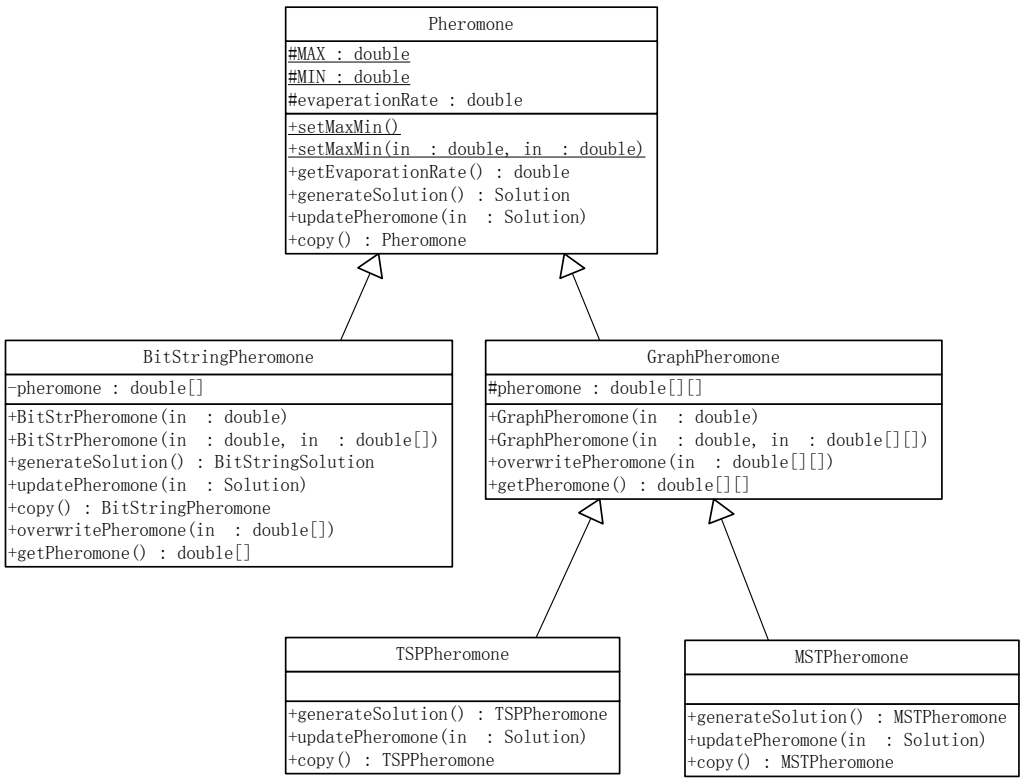
The class *OMSolution*, *LOSolution* and *LOLZSolution* all inherit the *BitStringSolution* and implemented all abstract methods depending on the problem.

Similar to the *BitStringSolution*, the class *GraphSolution* also inherit the base class and add a two-dimension Boolean array with getter method to represent the graph solution. Here the array is an adjacent matrix to represent whether an edge is included in the solution. The added constructors and toString method are implemented in a way that works similar to the methods with same name in class *BitStringSolution*. The method isBestSolution is not added in the *GraphSolution* because the graph problems’ best solution is sometimes not known. There is also a static method to read the problem graph from a given file to the adjacent matrix in a static two-dimension integer array while the file path is saved in another static variable. The given file should be in MSTLIB[24] format. TSPLIB is a library of sample instances for the TSP and related problems from various sources and of various types. The detailed file description can be found at its website. Note that only the most commonly used edge weight type EUC_2D was supported due to time limitations. The problem graph is designed to be saved in the class *GraphSolution* because it is basically used to calculate the fitness value.

The class MinGraphSolution inherit the GraphSolution class and implemented all abstract methods. This class is used for both TSP and MST problem as the optimization target of these problems are minimize the sum of weights of a subset of edges.

### 3.2.2. Pheromone

Similar to the solutions, all pheromone classes have a same super abstract class *Pheromone*. It first derived a subclass *BitStringPheromone* and an abstract subclass *GraphPheromone* for different solution structures. Then class *GraphPheromone* derived two subclasses for TSP and MST problem separately. Because the pheromone structures and the solution construction procedures are complete same for bit-string problems (OM, LO, LOLZ), the class *BitStringPheromone* is used for these problems without further derivation. The class diagram is shown as follows:




The base class *Pheromone* have a instance variable “evaporationRate” with getter method to represent the evaporation rate. It is not a static variable because later we want to run an experiment that set different evaporation rates on different islands. Then the two static variable MAX and MIN are global variables used in the MMAS* algorithm to update the pheromone on all pheromone classes. The two setter methods are designed to be called before the algorithm begins to initialize the maximum and minimum value of the pheromone. The setter method with two parameters simply set the MAX and MIN to the parameters’ value and the method without a parameter set the MAX and MIN to a default value which is $1 - 1/n$ and $1/n$, where $n$ is the solution size. Finally, there are three abstract methods left for implementation in the derived classes:

- generateSolution: Call this method to generate a solution based on the pheromone. It is used on each non-migration generation.
- updatePheromone: Call this method to update the pheromone regards to the given solution. It is used on all generations
- copy: Call this method to make a deep copy of the pheromone. It is usually used to copy the pheromone at migrations

The class *BitStringPheromone* inherit the base class and add a Double array with getter method to represent the pheromone that generate bit-string solutions. As the sum of the possibility to generate a “one” and the possibility to generate a “zero” is always 1, we only need to save any part of it. Here a value in the array represents the possibility to generate a “one” bit in the string. In addition, the following methods are added or implemented in class *BitStringPheromone*:

- BitStringPheromone: Two constructors are available for the class. The one without any parameter will initialize the pheromone array all to 0.5. The one with a given Double array will use that array as the initial pheromone.
- generateSolution: Call this method to generate a BitStringSolution. The algorithm is simply generate a random number in the zone [0, 1], if it is smaller or equal to the pheromone value, then the corresponding bit is set "one". Otherwise it is set to "zero". The time complexity of this operation is $O(n)$.
- updatePheromone: Call this method to update the pheromone array with regards to the given bit-string solution. See Section 2.4.3 for the detailed update scheme. The time complexity of this operation is $O(n)$.
- copy: Call this method to construct a deep copy of the class. Note that the evaporation rate is also copied. If we do not want to migrate the evaporation rate, the next method overwritePheromone is called instead of copy.
- overwritePheromone: Call this method to set the pheromone array with the given pheromone array. Note that this method will make a deep copy of the given pheromone array then set the current pheromone array to the copy. So it is safe to pass an arbitrary valid pheromone to the method.

The abstract class *GraphPheromone* also inherit the base class and add a two-dimension Double array with getter method to represent the graph pheromone. Though the theoretical solution construction graph differs a lot for the TSP and MST problem, they can be implemented in a same data structure. This leads to a different meaning of the values in the array, which will be described in detail separately in each class. Also, the method overwritePheromone works similar to the method with a same name in the *BitStringPheromone* class. Finally, the positive reflection of the all graph pheromone update is fixed to the evaporation rate $\rho$.

The class *TSPPheromone* is derived from class *GraphPheromone* and all abstract methods are implemented. As the construction graph is same with the problem graph in TSP, here the two-dimension pheromone array simply means the adjacent matrix of the pheromone. Finally, see Section 2.5.3 for the detailed update scheme.

- generateSolution: The algorithm to generate a *TSPSolution* is completely same as the construction procedure given in section 2.5.2. The construction procedure is done by a $n-1$ step random walk (the last step is going back to the starting city). In each step, we first compute the sum of the combination of the pheromone value and the heuristic value, which costs $O(n)$. Then a random number in the zone [0, sum] is generated to determine which edge is selected. Therefore the time complexity of generating a *TSPSolution* is $O(n^2)$.

Then class *MSTPreromone* is also derived from class *GraphPheromone* and all abstract methods are implemented. As described in Section 2.6.2, the construction graph for MST problem has $m+1$ vertices where $m$ is the number of edges in the problem graph. Though the construction graph is much larger than the problem graph, we can still use an adjacent matrix whose size is same as the problem graph to save

the pheromone for the construction graph thanks to the pheromone update scheme. The pheromone of all edges to a specified vertex will stay the same during the update because the initial value, the update formula and the update conditions are same. So we just need to save one copy of the pheromone value for all edges to a specified vertex, in the other word, a specified edge in the problem graph. Furthermore, as there are no edges to the starting vertex of the construction graph, no pheromone value is available for that point. Thus, the pheromone information of the construction graph is compressed to a much smaller adjacent matrix.

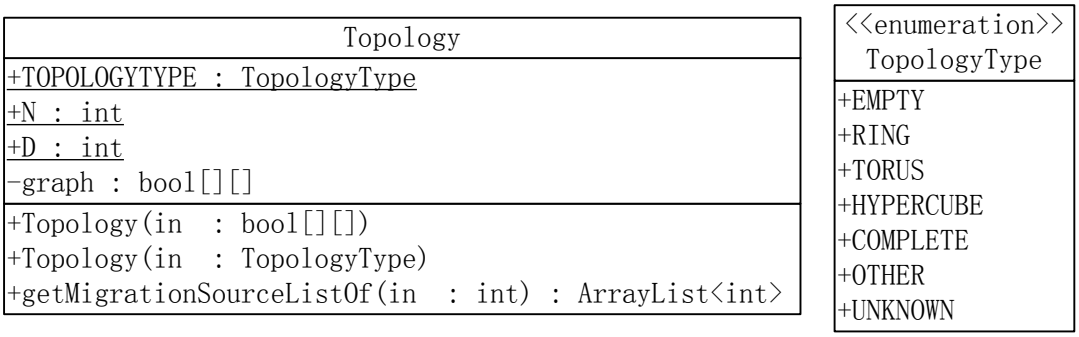
- generateSolution: The construction procedure of MST solution is given in section 2.6.2, but the implementation uses a simplified algorithm to generate solutions without changing the original procedure. First, as described above, the pheromone value of all edges to a specified vertex in the construction graph are same, so we actually do not concern which vertex is the current vertex while performing random walk. Thus, the random walk becomes equivalent to selecting $n-1$ vertices with same restriction. Similar to a step in the TSP solution construction, we also need to sum of the combination of the pheromone value and the heuristic value. But here we have $n^2$ vertices and the validity of a vertex is more complicated. An intuitive implementation to validate whether an unvisited vertex is valid in next step is making a temporary solution corresponding to the selected vertices plus the testing vertex and then applying a depth first search (DFS) on the temporary solution to find whether there is a cycle. If a cycle is found, then the testing vertex is not valid. If not, then the testing vertex is valid for next selection. This intuitive implementation cost $O(n)$ on each vertex test and then $O(n^3)$ on each vertex selection and a total of $O(n^4)$ time to generate a solution. The performance can be improved by using union-find set algorithm on the vertex validation procedure. Considering a situation in a MST solution construction procedure that some vertices in the construction graph is selected, the corresponding solution has some edges connecting vertices without making a cycle. When adding an edge to the solution, the only way to form a cycle is to add an edge whose two vertices have already connected by other edges. Conversely, if two vertices have not been connected, then add an edge between them will not make a cycle, and then the corresponding vertex in construction graph is valid. Thus, the algorithm is straight forward. At the beginning, all vertices in the solution are in the separated sets. During the solution construction procedure, testing whether a vertex in the construction graph is valid for selection can be solved by finding out whether two corresponding vertices in the solution are in the same set. Also, after selecting a vertex in the construction graph, we need to union two sets that contain corresponding vertices. Thus, after $n-1$ vertex selections and set union operations, a valid solution is constructed and all vertices are in the same set. Thanks to the union-find set algorithm with union by rank and path compression strategy, a single operation of union or find only cost $O(1)$ in practice. Thus it costs $O(1)$ on each vertex test and then $O(n^2)$ on each

vertex selection. Therefore, the time complexity of generating a *MSTSolution* is $O(n^3)$.

The class *MSTPreromone* has an inner class *UnionFindSet* that implements the union-find set algorithm with union by rank and path compression strategy. It is used for MST solution generation.

### 3.2.3. Topology

The class *Topology* is used to save the topology of the migration. An additional enumeration type *TopologyType* describes the special topology shape. The class diagram is shown as follows:




There are three static variables in *Topology* class. TOPOLOGYTYPE describes the shape of the graph. N and D are parameters exclusively for Torus topology. They should be initialized before construct an instance of *Topology* class. The instance variable graph saves the topology graph in an adjacent matrix.

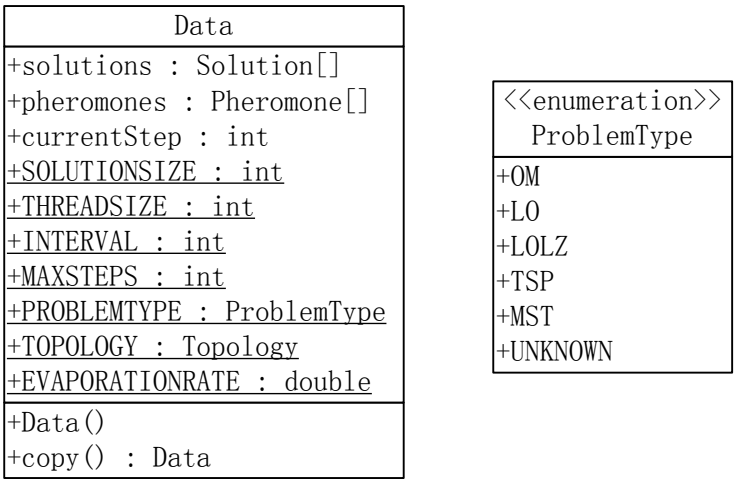
- Topology: Two constructors are available for the class. The one that takes a two-dimensional Boolean array will use it as its own adjacent matrix and set the TOPOLOGYTYPE to OTHER. The one that takes a topology type as the parameter will initialize the matrix and TOPOLOGYTYPE to the specified topology.
- getMigrationSourceListOf: Call this method to obtain a list of island numbers that directly connected to the given island. This method is called once from each island thread to know which islands they need to communicate with in the migration generation.

The enumeration type *TopologyType* has seven predefined values. The former five are introduced in section 2.3. OTHER is used when the topology is other graphs that given as an adjacent matrix. UNKNOWN is used when the topology is not initialized or not properly initialized to make an error message.

### 3.2.4. Data

The class *Data* is used to save all data and parameters of the MMAS* island algorithm. As all island threads will have a reference of a *Data* instance, so it becomes a shared memory that can be accessed from all island threads. Thus the communication is simply reading data from memory. An additional enumeration type *ProblemType*

describes the type of the problem to be solved. The class diagram is shown as follows:




The static variables contain SOLUTIONSIZE (the length of the solution of bit-string problems or the number of vertices of TSP/MST problem graph), THREADSIZE (the number of islands), INTERVAL (the migration interval), MAXSTEPS (the number of generations a single island can proceed), PROBLEMTYPE (the type of the problem), TOPOLOGY (the migration topology) and EVAPORATIONRATE (the evaporation rate). These variables should be initialized before the MMAS* algorithm starts. The instance variables include an array of solutions, an array of pheromones and an integer to indicate the number of generations.

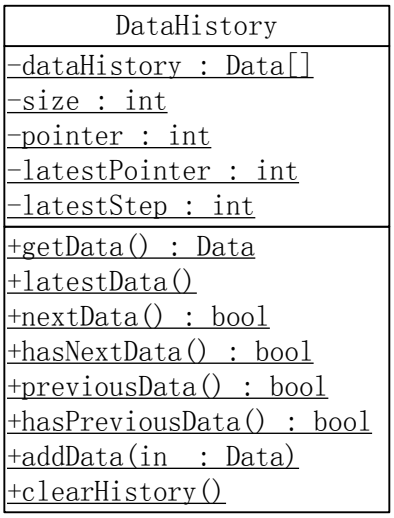
- Data: The constructor will allocate the space of the solutions array and pheromones array to fit the number of islands. But the solutions and pheromones are not initialized in this method.
- copy: Call this method to make a deep copy of the data. It is used to save history data.

### 3.2.5. DataHistory

The class *DataHistory* is used to save history *Data* classes. It is designed to implement the history review function in the GUI. As the *Data* class contains all information of the current status, copying and saving the *Data* class likes to take a snapshot of the system. The class diagram is shown in the following.

All variables and methods are designed to be static because we only need one data history recorder. We used an array with fixed length instead of an ArrayList to save the history *Data* classes because it is too space costly to save all snapshots. We only managed to save a limited number of history *Data* classes. After the maximum number of data is reached, a new data will replace the oldest data in the array. The remaining four variables are used to support the procedure of reviewing history data. The variable “size” indicates the number of saved history data in the array. The variable “pointer” indicates the data that currently viewing. The variable “latestPointer”

and “latestStep” indicates the latest data and its number of generations.




- getData: Call this method to get the history data pointed by variable pointer. Usually the pointer will point to the latest data.
- latestData: Call this method to move the pointer to the latest data.
- nextData: Call this method to move the pointer to a later data.
- hasNextData: Call this method to check whether there is a later data.
- previousData: Call this method to move the pointer to a older data.
- hasPreviousData: Call this method check whether there is a older data.
- addData: Call this method to save the given data in the *DataHistory* class, then the pointer will point to this latest data.
- clearHistory: Call this method to delete all data in the *DataHistory* class. It is used before a new MMAS* island algorithm starts to delete data from last run.

## 3.3. The MMAS* Island Algorithm

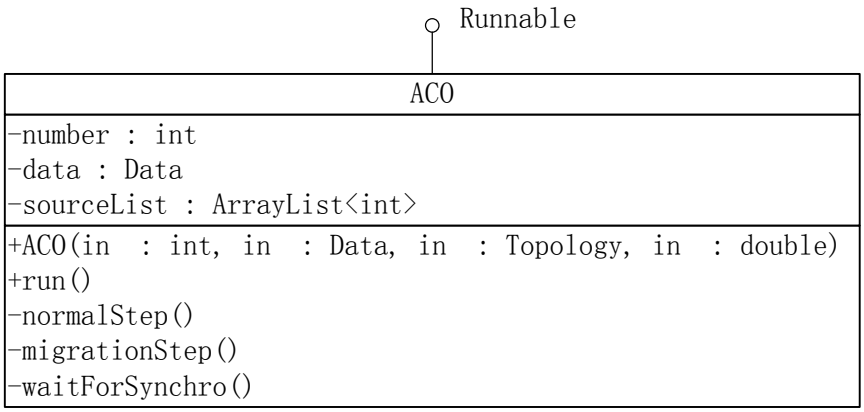
The core of the MMAS* island algorithm consists of two classes: *ACO* and *PACO*, where class *ACO* simulates single island and class *PACO* coordinates the islands. In addition, a *SerialPACO* class is implemented for experiments.

### 3.3.1. ACO

The class *ACO* implemented the interface *Runnable* as its instances are intended to be executed by a thread. The class diagram is shown in the following.

The instance variable “number” indicates the island land number of the *ACO* instance. Note that the islands are numbered from 0 to Data.THREADSIZE-1 to make the island number equivalent to the index of the solution and pheromone array in Data. On other words, this makes data.solutions[number] be the solution of the island and data.pheromones[number] be the pheromone of the island. Later when the island number is outputted, it is added by 1 to make its range from 1 to Data.THREADSIZE.

Then the variable data provides the access to the solutions and pheromones. The sourceList indicates the adjacent island numbers in the topology graph.



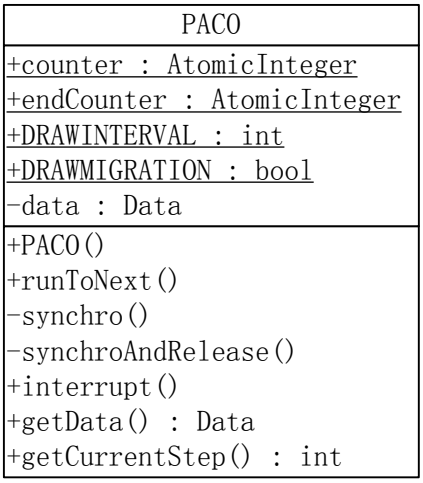

- ACO: The constructor takes four parameters, which are island number (integer), data reference (Data), topology reference (Topology) and the evaporation rate (double). The first two parameters are saved as instance variables while the sourceList are extracted from topology and the evaporation rate is used to initialize the pheromone of this island in the data. Then a solution is generated from the pheromone to initialize the solution. And at last, the pheromone is updated depending on the solution of this island.
- run: The method that inherit from the *Runnable* interface. It will be called automatically when the instance of the class is executed by a thread. The basic island algorithm is rather simple: loop Data.MAXSTEPS generations, perform normal generation or migration generation by calling methods described below depending on the number of generations. We only need to synchronize the islands to same generations before and during the migration generation. But due to the introduction of GUI, we need to be able to observe the island status after each specified generations and/or each migration generations. This is solved by adding extra synchronizations to specified generations. See section 3.3.3 for the procedure with interaction with *PACO* class in detail.
- normalStep: Call this method to perform a normal generation. First, a new solution is generated from the pheromone. Then, the current best solution is replaced with the new solution if it is better. Last, update the pheromone value according to the best solution.
- migrationStep: Call this method to perform a migration generation. First, it waits all islands to the same situation, i.e., all previous generations on all islands are finished. Second, solutions are collected from adjacent islands. If the best among them is better than current best solution, a copy of the best incoming solution and pheromone is made. Third, an additional synchronization is performed here to avoid reading or writing dirty data. Fourth, the current best solution and pheromone are replaced with the copy if applicable. Last, update the pheromone value according to the best solution.

Note in the fourth step, a copy prior to the second synchronization is necessary to avoid data corruption.

- waitForSynchro: Call this method to wait for all islands proceed to the same place. See function synchro and synchroAndRelease in next section (3.3.2) for its implementation.

### 3.3.2. PACO

The class *PACO* is used to perform a complete MMAS* island algorithm, which includes creating and coordinating island threads. The class diagram is shown as follows:




The static variable counter is used to count how many islands have proceeded to the next synchronization point. As it is a Java *AtomicInteger* instance, all operation on it is guaranteed to be atomic, so it can be modified from ACO without complicated synchronize code. Similarly, the static variable endCounter is used to count how many islands have finished all their generations. The next two variables are related to the observation of the islands' status. First, we need to synchronize and pause the islands to perform an observation every DRAWINTERVAL generations, the result is updated at GUI and recorded in *DataHistory*. Then, if DRAWMIGRATION is true, the statuses before and after each migration are also observed, updated and recorded. The instance variable data is simply the one that shared among all island threads.

- PACO: The constructor will initialize an instance of data, then initialize and start the island threads. After the construction, all islands will pause before the first generation waiting for the synchronization.
- synchro: Call this method to wait for all islands proceed to next synchronize point. It is implemented in a way that when an island calls waitForSynchro, it will get and increase the counter by 1, and then call data.wait() method to causes the thread to wait until another thread invokes the notify() method or the notifyAll() method. Method synchro will wait until the counter raise up to the number of islands.

- synchroAndRelease: Call this method to synchronize all islands by calling synchro method and then data.notifyAll() is called to resume all islands.
- runToNext: This is the core control method of the *PACO* class. Call this method to proceed the algorithm to next status that needs to be observed.
- interrupt: Call this method to stop the whole algorithm.

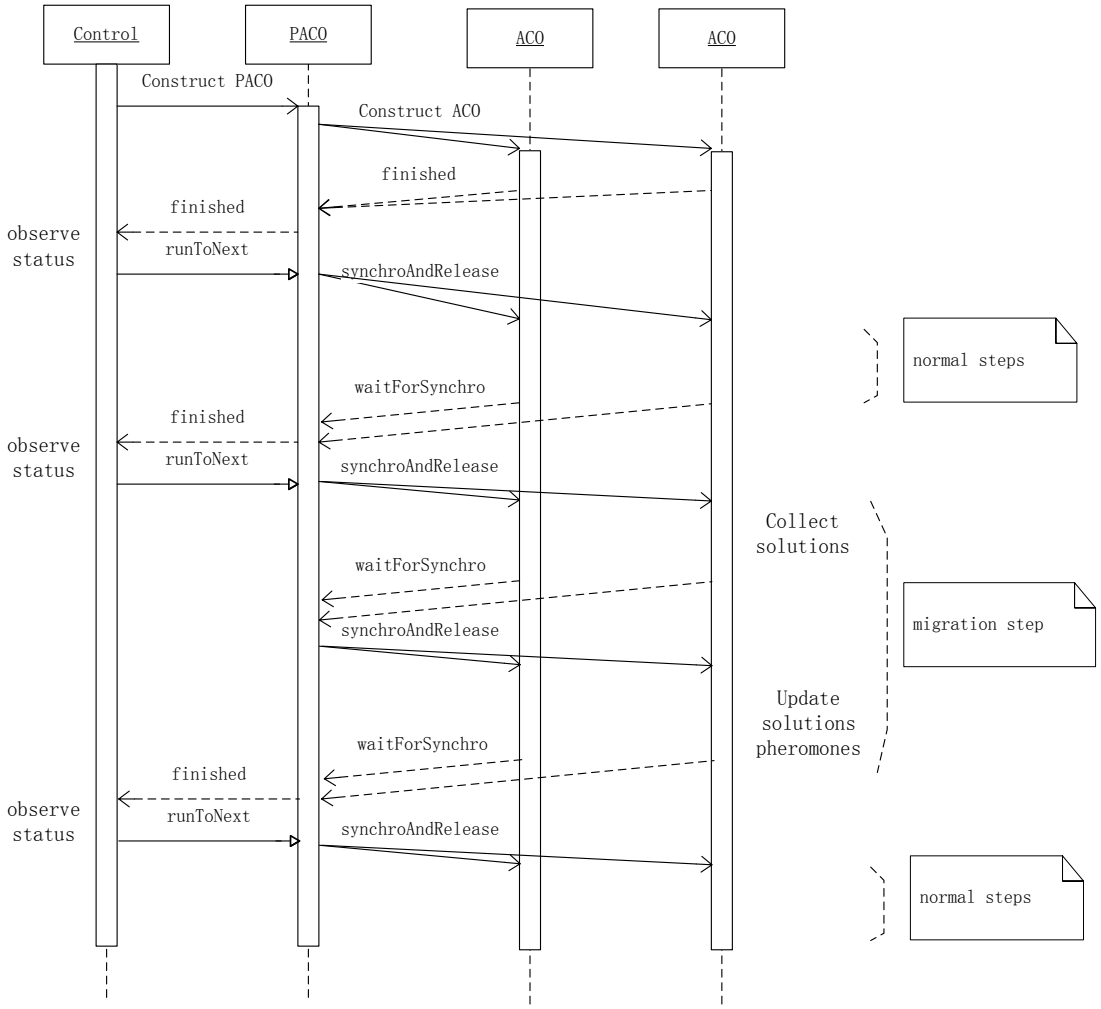



**Figure 5** Sequence Diagram of Two Islands

## 3.3.3. Sequence Diagram

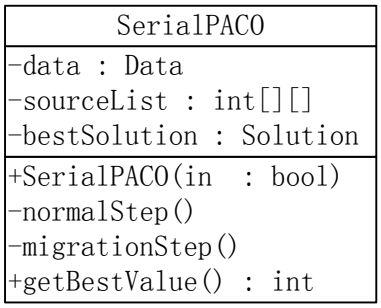
**Figure 5** is a sequence diagram of two islands that illustrates the relationship between PACO and ACO as well as how they are synchronized in different situations.

In this case, we will observe the statuses before and after the migration generation. First, the control thread creates an instance of PACO and the PACO instance constructs two ACO threads. After the construction, we have a chance to observe initial solutions and pheromones. Then the runToNext method of PACO is called. It releases ACO threads and many normal generations passed. Before the next migration generation, ACO threads runs to a synchronization point. When PACO

confirms all ACO threads have finished their work, it finished a call of runToNext. Now, the control thread once again obtains a chance to observe the status before the migration. Then the second runToNext method call resumes ACO threads. The islands collect neighbor island solutions and pheromones. After another synchronization, solutions and pheromones are updated to latest. Then the second runToNext call returns after all islands finished their work. Now the control thread may observe the status after the migration and perform the next runToNext call.

### 3.3.4. SerialPACO

Same as the literal meaning, The class *SerialPACO* is a serial implementation of the island MMAS* algorithm. The class diagram is shown as follows:




The reason why a serial version of the algorithm was implemented is the experiments. In some experiments, we need to collect data from sequences of generations. For example, to implement the stop criterion introduced in section 4.1.1, the information of whether the global best solution is found and whether all islands are stuck are necessary. If we use the parallel version of the algorithm, it will cost a lot of time to switch between threads and synchronize them in each generation. Beside the advantage of time cost for experiments that need to synchronize frequently, the serial version of the algorithm is also very simple to be implemented as there is no multi-thread control problem, so it can be easily verified. Then we can use the serial version of the algorithm to verify the correctness of the parallel version of the algorithm. The disadvantage of the *serialPACO* is that it does not make full use of the CPU power and result to be slower for runs that have less synchronization.

The instance variable “data” and “sourceList” have same function as them in class *ACO*. Note that the variable type of sourceList is changed to a two-dimensional integer array from an ArrayList of integer. This is because we need to save source lists of all islands, but java does not support a generic array. Though we can implement it in an ArrayList of ArrayList of integer, but it is less efficient than a two-dimensional integer array as we do not change it once it is created and the only operation is reading. The variable bestSolution is a copy of the best solution ever found.

- SerialPACO: The constructor will initialize the algorithm based on the static variables of *Data* and other classes and then runs to the end of the algorithm. The stop criterion is determined by the parameter earlyStop (Boolean). If it is true and the problem has a bit-string solution, then the algorithm will be

stopped when the global best solution is found or the process is stuck, which will be further explained in section 4.1.1. Otherwise the algorithm will be stopped when the given number of generations is reached. Then we can use the getter methods to get the result of the algorithm.

- normalStep and migrationStep: This two methods performs a generation of the algorithm. The code is modified from the *ACO* class.
- getBestValue: Call this method to get the fitness value of the best solution found.

## 3.4. The User Interface

The principle for the user interface (UI) design is to allow users to have an intuitive access to the MMAS* island algorithm as well as the experiments. We will show the graphical user interface (GUI) design first, and then an overview of the implementation.

**Figure 6** shows the main frame of the program:

①: The drop down list allows the user to select problems to be optimized. If the LOLZ problem is selected, then the program will pop up dialog boxes to ask for additional block size and threshold input. If the TSP or MST problem is selected, then the program will pop up a file chooser to ask user to select a .tsp file that contains a supported problem.

②: The input text area allows the user to set the maximum number of generations before the algorithm is terminated.

③: The input text area allows the user to set the evaporation rate for all islands.

④: The input text area allows the user to set the size of the problem. If the TSP or MST problem is selected, then the problem size is fixed to be the number of vertices read from the selected file and the user will be not able to edit the input text area.

⑤: The drop down list allows the user to select the migration topology. If the torus topology is selected, then the program will pop up dialog boxes to ask for the number of rows and columns.

⑥: The input text area allows the user to set the number of islands. If the torus topology is selected, then the number of islands is fixed to the number of rows multiplied by the number of columns and the user will be not able to edit the input text area. If the hypercube topology is selected, then there will be an additional check after input to make sure the number of islands satisfies the request.

⑦: The input text area allows the user to set the migration interval. If the input is 0, then there will be no migration generations.

⑧: The button to start a new experiment.

⑨: The input text area allows the user to set the frequency (number of generations) to update the algorithm status to the canvas panel. If the input is 0, then there will be no status update on canvas panel other than migration generations.

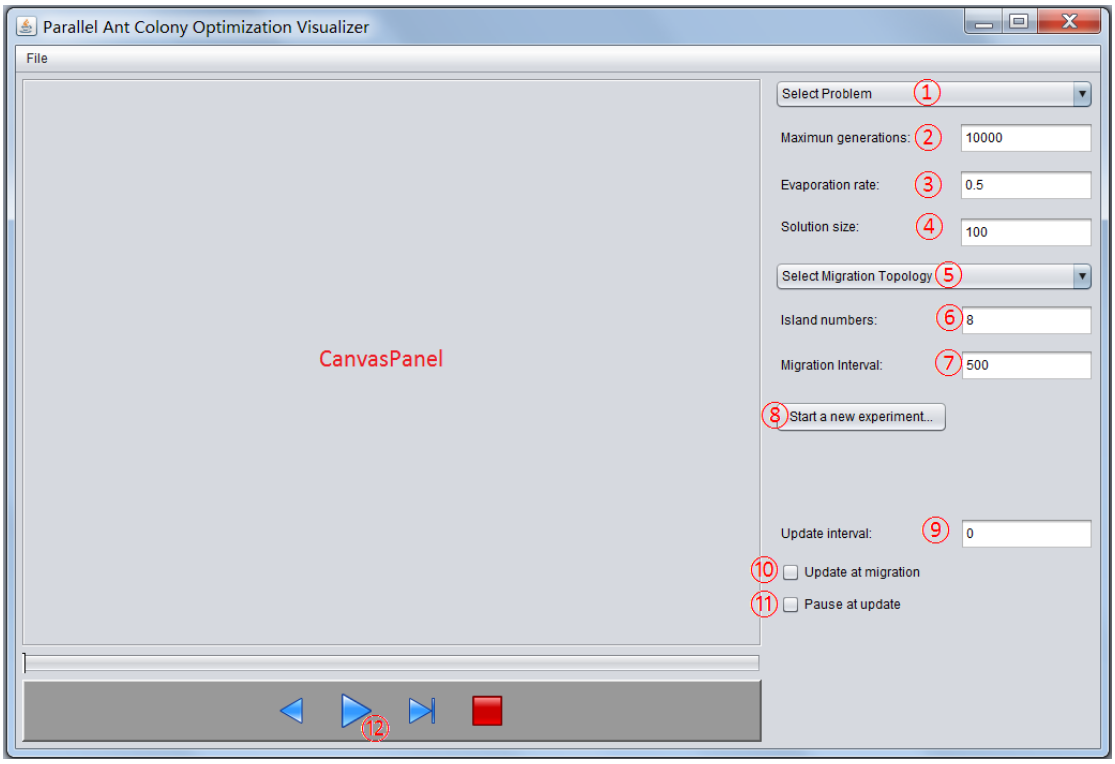



**Figure 6** The main frame of the program.

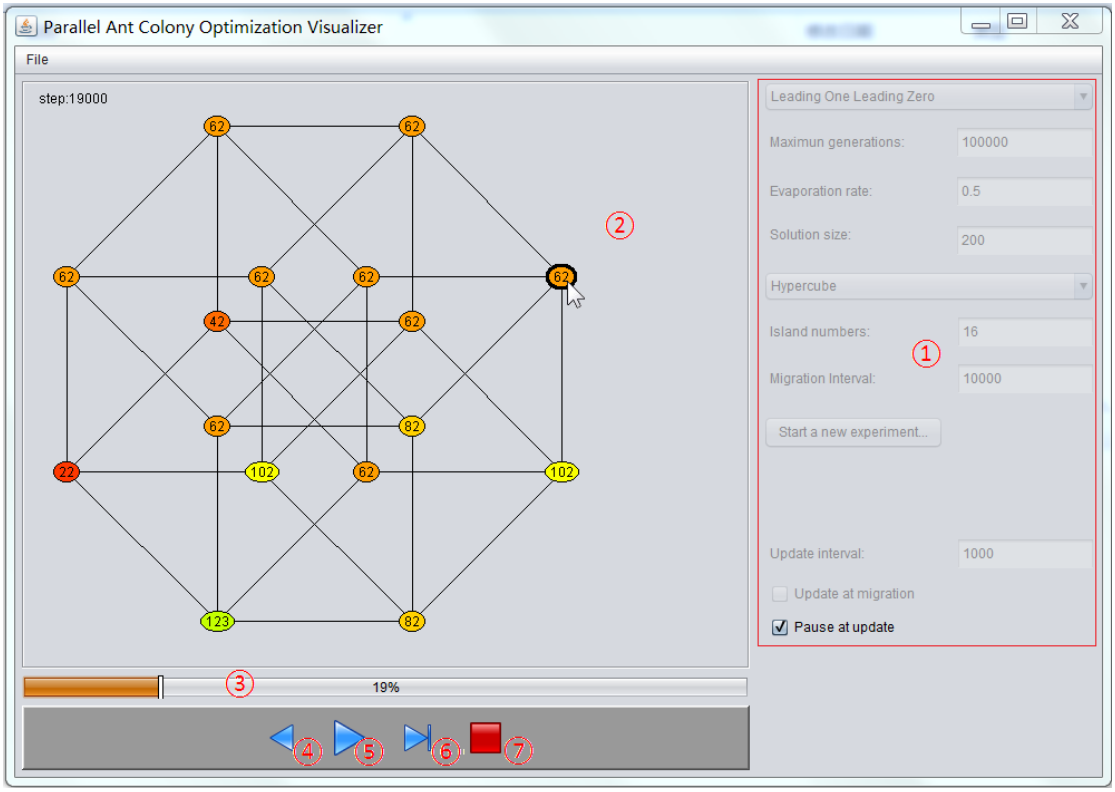



**Figure 7** The main frame of the program when the algorithm is running.

⑩: The check box allows the user to set whether to update the algorithm status to the canvas panel before and after migration generations.

⑪: The check box allows the user to set whether to pause the algorithm when a canvas panel update happens.

⑫: The button to start the algorithm.

Note that any invalid input will result in a prompt presenting correct input format and the value being edited will be set to a default value.

**Figure 7** shows the main frame when the algorithm is running:

①: All parameter setters are disabled to avoid further change except for the check box that allow user to pause the algorithm.

②: The canvas panel. The upper-left corner shows the generation number of the presenting status. Each circle represents an island and the lines between them means they are connected. The number in the circle is the fitness value of solution in that island. If the problem is OM, LO or LOLZ, the circle will be colored from red (worst possible solution) to yellow (a medium solution) to green (best possible solution), which provides a more direct observation. When the algorithm is pausing, it is also possible to see the detail of islands by clicking the circle. In the pop-up frame like **Figure 8**, it describes all information in an island, including the island number, the generation number, the evaporation rate, the pheromone, the solution and its value.

③: The dual progress bar shows the progress of the algorithm currently.

④: The backtrack button. This can only be used when the algorithm is pausing. Then we can backtrack up to 100 previous statuses that have ever been updated to the canvas panel.

⑤: The “resume” or “pause” button. It resumes the algorithm if it is pausing, or pause the algorithm if it is running. The icon of this button will be a pause icon when it is used to pause the algorithm.

⑥: The “run to the end” button. It will resume the algorithm and runs to the end without any further pause.

⑦: The stop button allow user to stop the algorithm at anytime.

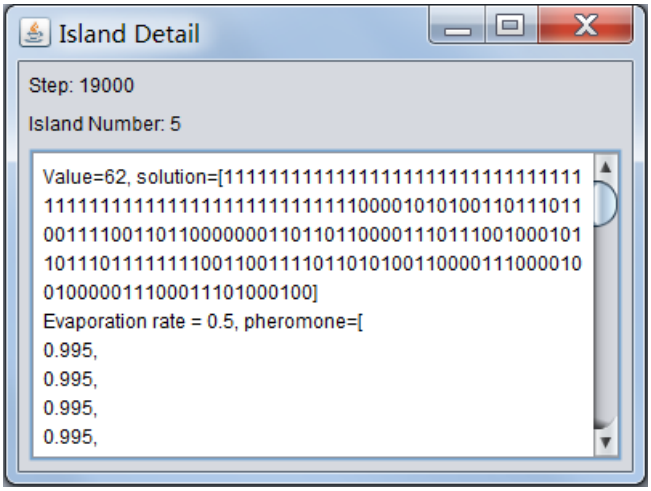



**Figure 8** The frame that shows the detailed island data.

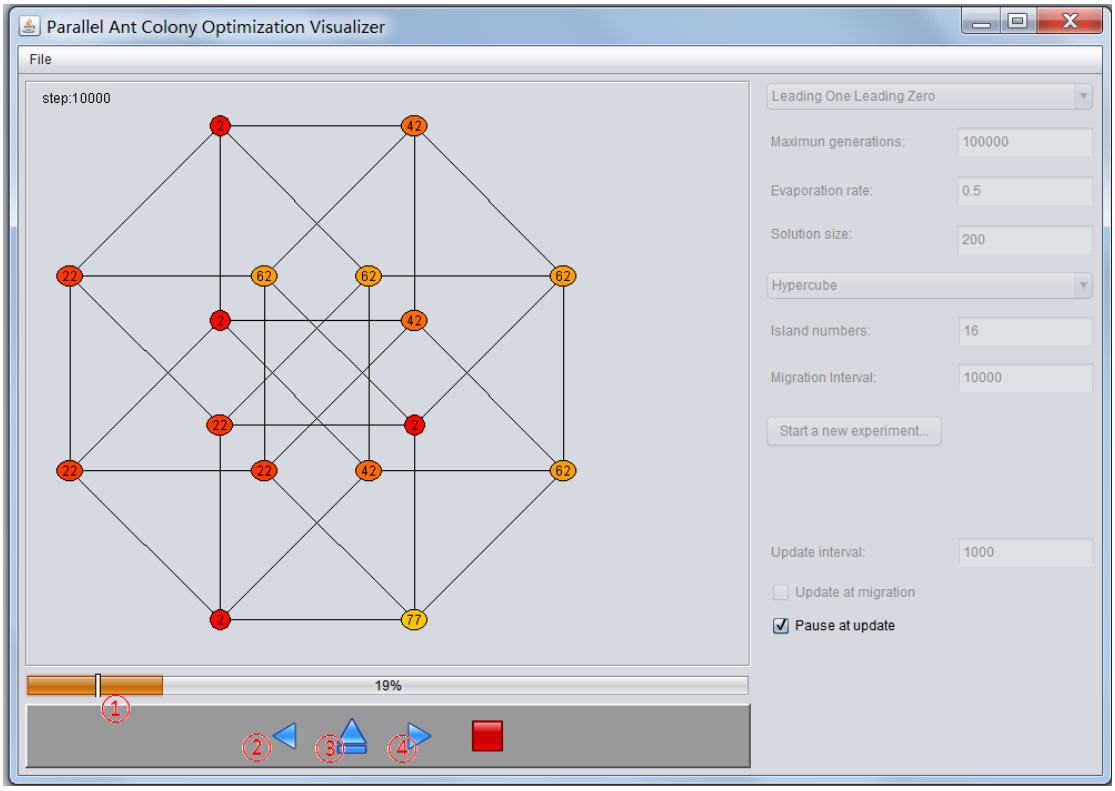



**Figure 9** The main frame of the program when backtracking to a previous status.

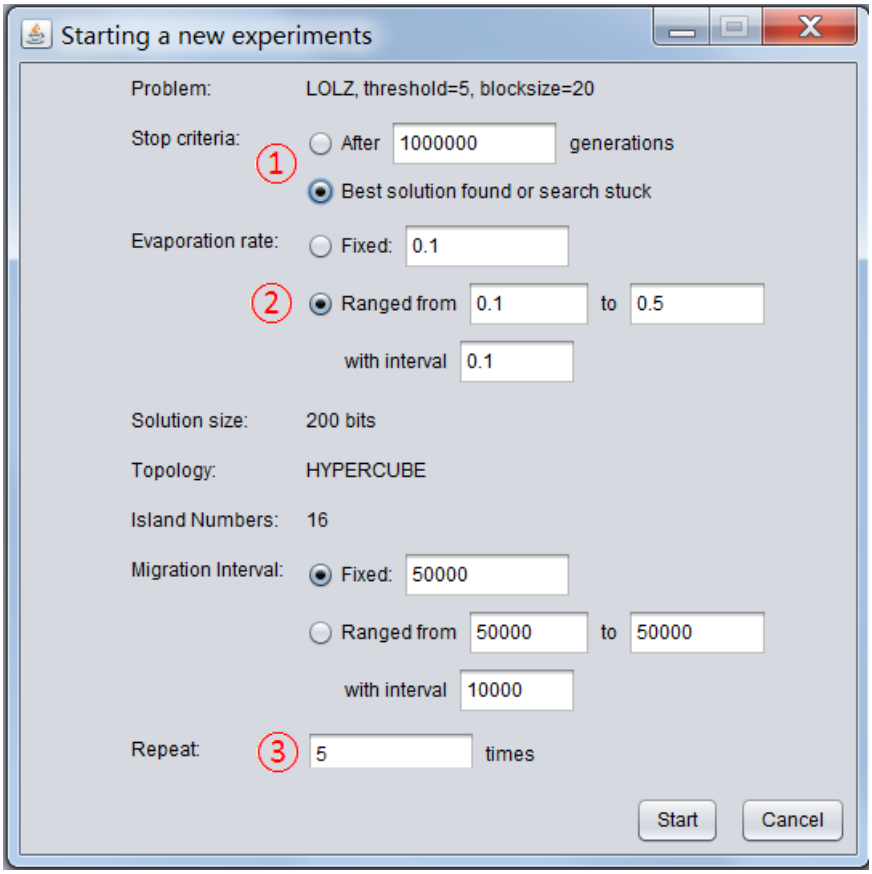



**Figure 10** The frame that allow user to set parameters for a new experiment.

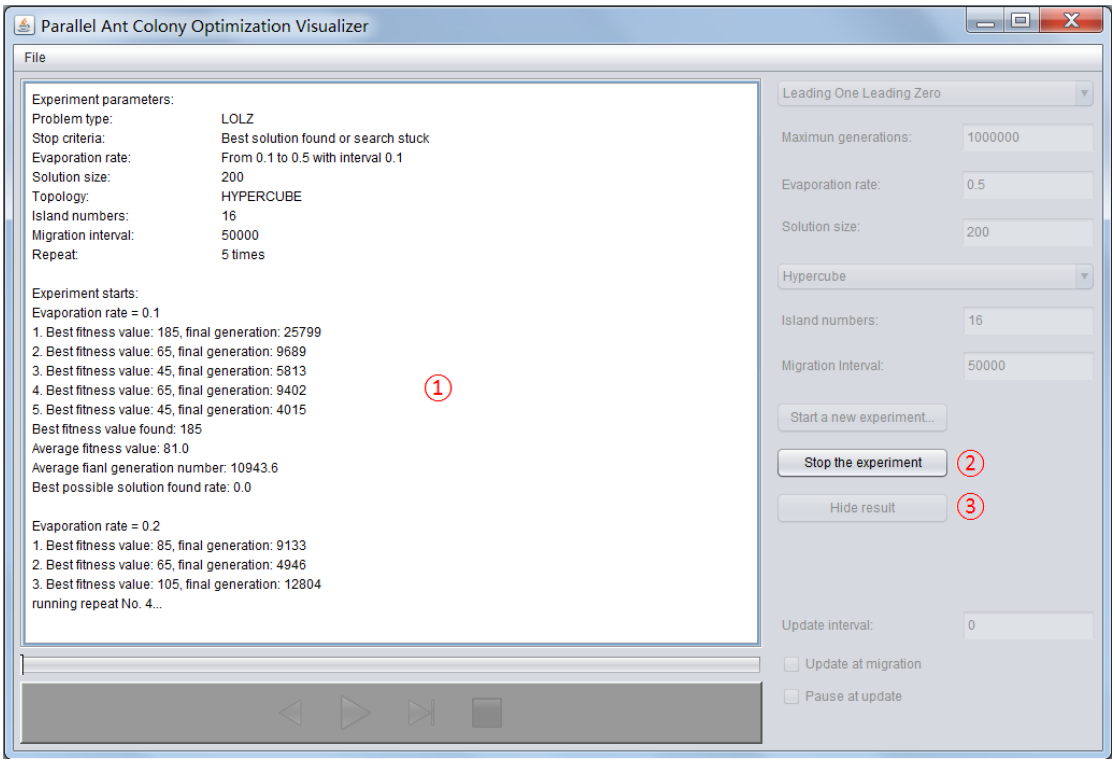



**Figure 11** The main frame of the program when an experiment is running.

**Figure 9** shows the main frame of when the algorithm is backtracking previous statuses.

①: The white indicator in the dual progress bar will indicate the progress of the status displayed in canvas panel, while the brown bar and the percentage is still the progress with regards to the latest generation of the algorithm.

②: The previous status button let the canvas panel display one previous status.

③: The quit backtrack button allows the user to stop backtracking and let the canvas panel display the latest status of the algorithm.

④: The next status button let the canvas panel display one later status.

**Figure 10** shows the frame that allow user to set parameters for a new experiment. The default setting is taken from the main frame.

①: Two stop criteria are selectable. The best solution found or search stuck criterion is only available for bit-string problems because the best solution for some TSP is not known and we do not have a similar criterion for TSP and MST though one could come up with suggestions. The search stuck is detailed defined in section 4.1.1 and it is only applied to LOLZ problem.

②: It is possible to set the evaporation rate and the migration interval to a range of values instead a fixed value. The statistics will be collected separately on different settings of the algorithm.

③: The input text area allows the user to set the number of repeats of the algorithm for all settings.

**Figure 11** shows the main frame when an experiment is running. When an experiment is running, all other operations are disabled.

①: The canvas panel in the main frame is replaced by a text area displaying the

experimental result dynamically and the statistical result after all repeats of a setting of the algorithm is finished.

②: The stop experiment button can terminate the experiment in advance.

③: After the experiment is finished, the hide result button can hide the text area and show the canvas panel again and enable all other functions.

**Figure 12** shows the relationship between the classes. The *MainFrame* class is the main class of the whole program. It will initialize the main frame (**Figure 6**, **Figure 7**, **Figure 9** and **Figure 11**) when the program starts. It can create a *Control* class in a separate thread to run the MMAS* algorithm. *DualProgressBar* and *CanvasPanel* are two classes that extended from java swing components to display the algorithm status in a more intuitive way. The *SolutionFrame* class is used to display the detailed information on a single island (**Figure 8**). The *ExperimentsFrame* class is created by the main frame to allow user to create experiments (**Figure 10**). The setting of the experiment is sent to the *RepeatExperimentTask* class, an inner class of the *MainFrame* class, which extends *SwingWorker* class. The *SwingWorker* class is an abstract class to perform lengthy GUI-interacting tasks in a dedicated thread. Here the *RepeatExperimentTask* class is used to run serial PACO in a separate thread to prevent the GUI from not responding.

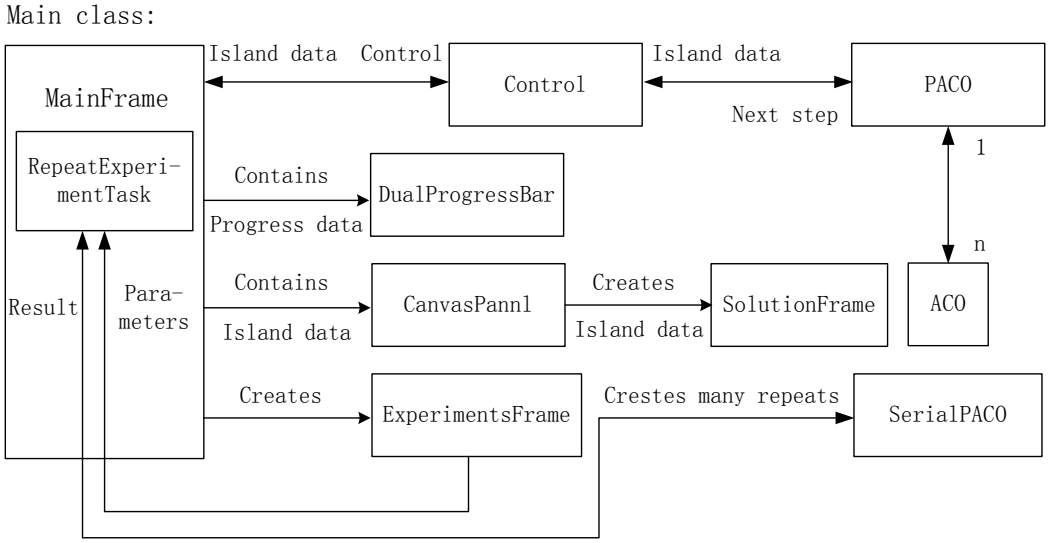



**Figure 12** The relationship between UI and algorithm classes.

## 3.5. Testing

As the MMAS* island algorithm is a random algorithm, it is not possible to test the whole algorithm by comparing input and output. But it is possible to do so in deterministic parts, including data initialization, pheromone update, migration and termination. They are covered with unit test. Due to time limitations, the unit tests are not described in this thesis. For the solution generating procedure that contains randomness, we ran a test based on the expectation. For example, in a bit-string problem, update the pheromone regarding a given solution until all pheromone values becomes the maximum value or minimum value. Then use this pheromone to

generate 1000 solutions and compare the solutions to the given solution, counting the number of bits flipped. As each bit has a probability of $1/n$ to be flipped, where $n$ is the size of the solution, then the expected number of bits flipped for 1000 solutions is 1000. It is likely that the solution generation procedure is correct if the counting result is around 1000. But if the counting result is something like 600, then it is more likely that there are bugs.

After tests for the algorithm, we start to test the graphical user interface. The interaction between GUI and core algorithm is focused, especially whether the actions that pause, resume and terminate the algorithm work properly. Besides that, several groups of data are tested to see whether they are presented as expected in the panel. Other tested functions include data transmitting and invalid operation prompt. Finally, the GUI's stability and reaction speed is tested by a large amount of using the interface.

## 3.6. Conclusion, Extensibility and Further Development

In this chapter, we covered the implementation of data structures, algorithms and the GUI with some tests.

During the design of this implementation, the extensibility is considered. Now it is very easy to extend the program to be able to solve a new kind of problem. Just extend a solution class and a pheromone class for the new problem, or even use a proper existent solution or pheromone class. Then make some small modifications on the initialization of the PACO class to adjust the new problem. There is no need to change anything related to the MMAS* algorithm.

Further development may focus on a more flexible way to pause and observe the status of the algorithm and a grid-computing compatible implementation.

# 4. Performance Experiments

In this chapter, we are using several experiments to examine the performance of the parallel ant colony optimization. The basic experimental setup is the optimization of LOLZ using 32 islands in parallel with solution size $n = 1000$, threshold $z = 10$ and 10 blocks ($b = 10$) of length $l = 100$ each. The topologies are defined in section 2.3 and torus takes (4, 8) as its row and column parameters. This setup is taken from paper [2] since it provided some existent experimental result to be referred. The main two changing parameters in paper [2] is topology and migration interval. In following experiments, we will have evaporation rate as the third changing parameter.

The experimental will repeat the algorithm 100 times and collect three results:

- Success rate: number of runs that successfully find a global best solution divides the total number of runs. This gives us a view how good the algorithm is.
- Average final fitness value: the average of the fitness value of best solution found when the algorithm stops. This shows the performance of the algorithm in search abilities.
- Average final number of generation: the average number of generation when the algorithms stops. Combining with the average final fitness value, this describes how fast the algorithm finds better solutions.

## 4.1. Special Settings for LOLZ Problem

### 4.1.1. An Alternative Stop Criterion

In paper [2], the authors introduced their stop criterion for LOLZ problem. All algorithms were stopped when either the global optimum had been found or each individual island had at least $z$ leading zeros in the block currently to be optimized. In the second case, if we want to generate a better solution, at least these $z$ leading zeros have to be flipped at same time. In EA, the probability to flip a bit is $1/n$. Under the setting of $n = 1000$ and $z = 10$, the probability of generating a better solution is less than or equal to $(1/n)^z = 10^{-30}$. Then the expected time until a local optimum is left is at least $10^{30}$. Therefore it makes sense to stop either the best solution is found or a better solution cannot be found in a reasonable generations.

In MMAS* algorithm, the probability to flip a zero bit to one bit is no longer always $1/n$. It can be just flipped and the pheromone is decreasing from $1 - 1/n$. So if the evaporation rate is not very large, the probability to generate a better solution from a local optimal could be large than EA. Therefore in addition to the condition that each individual island had at least $z$ leading zeros in the block currently to be optimized, the pheromone of these bits also has to be the minimum value $1/n$. In the following experiments, we will use this modified stop criterion for LOLZ problems.

### 4.1.2. Initial Generation

In paper [3], the authors proved a theorem that parallel EA with a proper setting can find global optimum of LOLZ problems in polynomial time. The basic idea of the theorem is to let the migration happen when the optimization proceeds to approximately half of each block. Thus the solutions on islands that make a wrong decision that collects zeros can be replaced by better solutions that collect leading ones. According to this idea, the first migration happens after $l/2$ bits have been optimized, but there are $l$ bits between migration generations. So the theorem set the initial generation number to half of the migration interval to adjust the algorithm to the theorem.

In the MMAS* algorithm, we decide to change the condition of a migration generation from $\boldsymbol{t}\ mod\ \boldsymbol{\tau} = 0$ to $\boldsymbol{t}\ mod\ \boldsymbol{\tau} = \boldsymbol{\tau}/2$ instead of changing initial generation number to achieve same function for LOLZ problem experiments. The reason to make this change is to eliminate the confusion of whether a generation number contains the extra $\boldsymbol{\tau}/2$ generations. This modified migration generation condition is used in following LOLZ problem experiments.

## 4.2. Reproduction of the Parallel EA Experiments

At beginning, we would like to reproduce the basic results made in paper [2]. As mentioned in section 2.2.2, the parallel MMAS* algorithm can be specialized to parallel EA used in paper [2] by setting evaporation rate to a large enough value. Thus, the pheromone value can only be maximum value $1 - 1/\boldsymbol{n}$ or minimum value $1/\boldsymbol{n}$. Then for each generation, all bits have a probability of $1/\boldsymbol{n}$ to be different from the current best solution, which is equivalent to the parallel EA that flipping bits with probability $1/\boldsymbol{n}$.

The result of the original experiment in paper [2] is shown in **Table 1**. All algorithms use $\mu = 32$ islands or population size for panmictic EA and the migration interval $\boldsymbol{\tau} = 50000$. All algorithms run 1000 times to collect data. See section 1.2 for detailed algorithm.

**Table 1** The result of the original experiment in paper [2].

| Algorithm | Success rate | Final fitness | Generation number |
|---|---|---|---|
| Panmictic $(\mu + 1)$ EA | 0.0 | 114 | 92367 |
| Independent subpopulations | 0.038 | 550 | 377472 |
| Island model on ring | 0.995 | 999 | 709704 |
| Island model on torus | 1.0 | 1000 | 655858 |
| Island model on hypercube | 0.651 | 907 | 647339 |
| Island model on $\boldsymbol{K}_{\mu}$ | 0.327 | 651 | 344759 |

The panmictic $(\mu + 1)$ EA has a population of 32 individuals and generate an offspring based on one of them. The independent subpopulations algorithm is running 32 independent $(1 + 1)$ EAs, so an offspring is generated based on each of the parent individuals. It is clear that the independent subpopulations algorithm performs better than panmictic $(\mu + 1)$ EA. The island model on ring and torus topology

performs best with success rate 0.995 and 1.0, while hypercube is 0.651 and the complete graph ($\boldsymbol{K}_\mu$) only have 327 of 1000 runs succeeded.

We calculated the 95% and 99% confidence interval of the success rate to statistically test our result. If the experimental result falls outside the confidence interval, then there is a probability of 95% or 99% that something goes wrong and the result should be examined. Let $\bar{p}$ be the estimated success rate, which equals to the empirical success rate. Then the confidence interval is $[\bar{p} - E, \bar{p} + E]$ where margin of error $E = z_{\alpha/2}\sqrt{\bar{p}(1-\bar{p})/n}$, $n$ is the number of repeats and $z_{\alpha/2}$ is 1.96 for 95% confidence interval and 2.576 for 99% confidence interval [25]. As this estimate method requires at least 5 successes and at least 5 failures, so the confidence intervals of Panmictic $(\mu + 1)$ EA and Island model on torus are not available here. But we can expect a very low success rate for Panmictic $(\mu + 1)$ EA and a very high success rate for Island model on torus. The result is shown in **Table 2**.

**Table 2** The confidence interval of the success rate of the experiment in paper [2].

| Algorithm | Success rate | 95% | 99% |
|---|---|---|---|
| Panmictic $(\mu + 1)$ EA | 0.0 | [ | [0.0, 0.0] |
| Independent subpopulations | 0.038 | [0.0261, 0.0499] | [0.0224, 0.0536] |
| Island model on ring | 0.995 | [0.9906, 0.9994] | [0.9893, 1.0] |
| Island model on torus | 1.0 | [1.0, 1.0] | [1.0, 1.0] |
| Island model on hypercube | 0.651 | [0.6215, 0.6805] | [0.6122, 0.6898] |
| Island model on $\boldsymbol{K}_\mu$ | 0.327 | [0.2979, 0.3561] | [0.2888, 0.3652] |

When we reproduce the experiment, the evaporation rate is set to 1 to make sure the pheromones of bits can only be maximum value $1 - 1/\boldsymbol{n}$ or minimum value $1/\boldsymbol{n}$, while all other parameters are selected exactly same as the original experiment. There are also four changes must be mentioned. First, the independent subpopulations algorithm is replaced by an almost equivalent setting of island model on empty graph. The minor difference is there is no new solution generated on the migration generation in island model algorithm. As the migration interval is 50000, the difference is only 0.002%, which can be ignored. Second, the notation of $\boldsymbol{K}_\mu$ is replaced by complete graph to make it clear and unity. Third, the panmictic $(\mu + 1)$ EA is not reproduced because we cannot simulate it using MMAS* algorithm. The MMAS* algorithm can simulate $(1 + 1)$ EA because the pheromone is updated regards to the best-so-far as well as the only solution and then used to generate new solution, while the $(1 + 1)$ EA also generates new solution according to the best-so-far solution. But the $(\mu + 1)$ EA has a big population of candidate parent solutions. It requires $\mu$ construction graphs for an ACO to simulate, which makes too much difference from a MMAS* algorithm. Note that increasing the number of ants in MMAS* algorithm will increase the number of new solutions in a generation, which is a wrong approach. In addition, the panmictic $(\mu + 1)$ EA always performs worse than independent subpopulations algorithm as discussed above. We finally decided not to reproduce the panmictic $(\mu + 1)$ EA. Fourth, we only repeat 100 runs due to time limitations.

The result of our experiment is shown in **Table 3**. From the table, we can see the results of empty and hypercube are consistent with the original result, but the results of ring, torus and complete have a huge difference with the original result.

**Table 3** The result of the experiment reproduced. (with Java provided PRNG)

| Algorithm | Success rate | Final fitness | Generation number |
|---|---|---|---|
| Island model on empty | 0.02 | 541 | 370116 |
| Island model on ring | 0.69 | 903 | 661711 |
| Island model on torus | 0.67 | 908 | 623809 |
| Island model on hypercube | 0.65 | 866 | 582449 |
| Island model on complete | 0.05 | 396 | 207616 |

After a careful examination of the implementation and several step by step debugging to check whether the program is running as we expected, we are confidence of our implementation being correct.

We come up with an idea that the pseudo random number generator (PRNG) may influence the result. The current PRNG we used is the default generator in *Random* class provided by Java. The internal algorithm is linear congruential generator which is one of the oldest and well known algorithms. See method “next” in *Random* class at [26] for more detailed implementation.

The first alternative generator we considered is Xorshift from [27]. It is an extremely fast algorithm and provides a changeable period of the numbers generated discovered by George Marsaglia. We take the triple [21, 35, 4] to make 64-bit random numbers with a period of $2^{64} - 1$ and the implementation is shown as follows:

```
@Override
protected int next(int nbits) {
    long x = this.seed;
    x ^= (x << 21);
    x ^= (x >>> 35);
    x ^= (x << 4);
    this.seed = x;
    x &= ((1L << nbits) - 1); // eliminate unnecessary leading bits
    return (int) x;
}
```

This method overrides the original method in *Random* class which is called by other methods to generate different formats of random numbers. The three Xor operations is the core of the Xorshift algorithm and later operations removes unnecessary leading bits and cut the number to an integer to adjust the return type.

After applying the new PRNG to the program, we run the experiment again and the result is shown in **Table 4**.

The result using Xorshift is similar to the result using linear congruential generator. To verify this result, we tried another PRNG: Mersenne twister. Mersenne twister, developed by Makoto Matsumoto and Takuji Nishimura, generates very high-quality pseudo random numbers [28]. It is also used as the default random number generator

in Python, PHP, Matlab, etc. We used the Mersenne twister (class *MersenneTwister*) in COLT [29], which provides a set of Open Source Libraries for High Performance Scientific and Technical Computing in Java. The result is shown in **Table 5**.

**Table 4** The result of the experiment reproduced. (with Xorshift PRNG)

| Algorithm | Success rate | Final fitness | Generation number |
|---|---|---|---|
| Island model on empty | 0.03 | 554 | 380555 |
| Island model on ring | 0.73 | 918 | 674934 |
| Island model on torus | 0.70 | 905 | 617033 |
| Island model on hypercube | 0.61 | 877 | 587727 |
| Island model on complete | 0.08 | 425 | 226816 |

**Table 5** The result of the experiment reproduced. (with Mersenne twister PRNG)

| Algorithm | Success rate | Final fitness | Generation number |
|---|---|---|---|
| Island model on empty | 0.02 | 526 | 355986 |
| Island model on ring | 0.66 | 888 | 650574 |
| Island model on torus | 0.76 | 932 | 645678 |
| Island model on hypercube | 0.59 | 866 | 577617 |
| Island model on complete | 0.06 | 416 | 219961 |

**Table 6** The success rate of the basic experiments (with different PRNG)

| Algorithm | Paper [2] | Java (linear congruential) | Xorshift | Mersenne twister |
|---|---|---|---|---|
| Panmictic $(\mu + 1)$ EA | 0.0 | N/A | N/A | N/A |
| Island model on empty | 0.038 | 0.02 | 0.03 | 0.02 |
| Island model on ring | 0.995 | 0.69 | 0.73 | 0.66 |
| Island model on torus | 1.0 | 0.67 | 0.70 | 0.76 |
| Island model on hypercube | 0.651 | 0.65 | 0.61 | 0.59 |
| Island model on complete | 0.327 | 0.05 | 0.08 | 0.06 |

**Table 6** collects the success rate of all above results to make it easier to compare. It is clear that the results from our implementation are similar while the result of ring, torus and complete from paper [2] cannot be reproduced since none of our result fall into any of the confidence intervals in **Table 2**.

We have contacted the authors of paper [2]. They were not willing to disclose the detail of their experiments, neither source code nor the compiled program. But they mentioned that the PRNG they used is not Mersenne twister and they had found a fact that the PRNG can have a big impact on the results of this kind of experiments. The authors also would like to repeat the experiments using Mersenne twister, but we didn't get any further information from them until this thesis is finished.

In the following experiments we made, we will use the PRNG provided by *Random* class in Java (linear congruential generator) for convenient and easy to be reproduced since the Xorshift and Mersenne twister can have a variance of implementations.

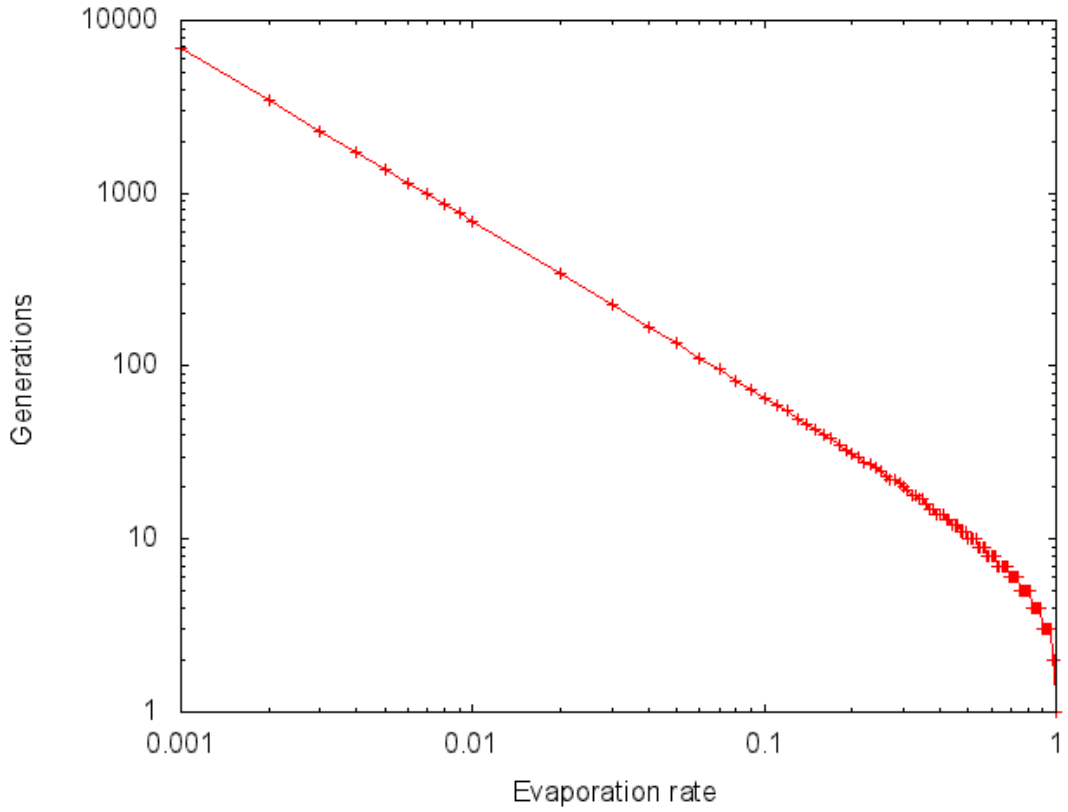



**Figure 13** The relationship between the evaporation rate and the number of generations needed to change the pheromone from one side to another. Note that both axes use a logarithm scale.

## 4.3. Comparison of Topologies and Evaporation Rates

After reproducing the result of the parallel EA algorithm, we would like to investigate the influence of the evaporation rates as a newly introduced parameter.

In primary experiments, the evaporation rates were set to 0.1 to 1 with an interval of 0.1. The result showed that there is no significant difference between them on success rate, final fitness value and final generation number. The reason why there is not much influence made by the evaporation rate is that the evaporation rate is still too large. Take the parallel EA on ring topology as an example, it get an average final fitness value 903 in 661711 generations. As the initial pheromone gives an even chance for every bit to be zero or one, so the solution has expected $n/2$ zeros and $n/2$ ones. It means that we only need to flip $n/2$ zeros in expectation to obtain the global best solution. In other word, flipping a zero to one will increase the total fitness value for 2 in expectation. Therefore the parallel EA on ring topology needs an average of $661711/(903/2) \approx 1466$ generations to flip a zero bit at correct place. Now consider the behavior of the pheromone value when the bit is flipped from zero to one. It costs less generations to flip the pheromone for a larger evaporation rate. **Figure 13** shows the relationship between the evaporation rate and the number of generations needed to change the pheromone from minimum to maximum or vice versa. It only costs several or tens of generations to change the pheromone from

minimum value to maximum value when the evaporation rate is under $10^{-1}$ scale. As the MMAS* algorithm behaves same as an EA after the pheromone has fixed (every bits have a probability of $1/n$ to be flipped), the number of generations that the pheromone is changing is too small to make a significant difference.

Therefore, we use a more wide range of evaporation rates of 1, 0.5, 0.1, 0.05, 0.01, 0.005 and 0.001 in the formal experiments. They need 1, 10, 66, 135, 688, 1378 and 6904 generations to flip the pheromone respectively. The reason why we did not apply ever smaller evaporation rates is the pheromone changes so slow that the algorithm is more like a random search in a majority of generations.

The migration interval is set to 50000 which is same as the migration interval in paper [2]. The result of different topologies is shown in **Table 7** (ring), **Table 8** (torus), **Table 9** (hypercube) and **Table 10** (complete). The last column shows the estimated number of generations to flip a zero bit at correct place. It equals the number of generations divides the final fitness, then multiplies 2. As mentioned in the beginning of this chapter, the torus topology takes (4, 8) as its two parameters.

**Table 7** The result of the ring topology with different evaporation rates.

| Evaporation rates | Success rate | Final fitness | Generation number | Flip |
|---|---|---|---|---|
| 0.001 | 0.50 | 845 | 914959 | 2166 |
| 0.005 | 0.60 | 863 | 694057 | 1608 |
| 0.01 | 0.70 | 906 | 695586 | 1536 |
| 0.05 | 0.65 | 888 | 659410 | 1485 |
| 0.1 | 0.64 | 900 | 667644 | 1484 |
| 0.5 | 0.62 | 886 | 649844 | 1467 |
| 1 | 0.72 | 899 | 663634 | 1476 |

**Table 8** The result of the torus topology with different evaporation rates.

| Evaporation rates | Success rate | Final fitness | Generation number | Flip |
|---|---|---|---|---|
| 0.001 | 0.57 | 832 | 835574 | 2009 |
| 0.005 | 0.67 | 874 | 653887 | 1496 |
| 0.01 | 0.69 | 907 | 649489 | 1432 |
| 0.05 | 0.69 | 912 | 627753 | 1377 |
| 0.1 | 0.76 | 926 | 635420 | 1372 |
| 0.5 | 0.70 | 911 | 626535 | 1375 |
| 1 | 0.70 | 908 | 623436 | 1373 |

**Table 9** The result of the hypercube topology with different evaporation rates.

| Evaporation rates | Success rate | Final fitness | Generation number | Flip |
|---|---|---|---|---|
| 0.001 | 0.42 | 777 | 758730 | 1953 |
| 0.005 | 0.54 | 848 | 617989 | 1458 |
| 0.01 | 0.58 | 847 | 586574 | 1385 |
| 0.05 | 0.67 | 884 | 594294 | 1345 |
| 0.1 | 0.61 | 881 | 595111 | 1351 |
| 0.5 | 0.61 | 879 | 588962 | 1340 |
| 1 | 0.72 | 903 | 604463 | 1339 |

**Table 10** The result of the complete topology with different evaporation rates.

| Evaporation rates | Success rate | Final fitness | Generation number | Flip |
|---|---|---|---|---|
| 0.001 | 0.04 | 360 | 288303 | 1602 |
| 0.005 | 0.08 | 400 | 234105 | 1171 |
| 0.01 | 0.06 | 423 | 233497 | 1104 |
| 0.05 | 0.10 | 433 | 233625 | 1079 |
| 0.1 | 0.07 | 435 | 227649 | 1047 |
| 0.5 | 0.05 | 442 | 235966 | 1068 |
| 1 | 0.08 | 450 | 240521 | 1069 |

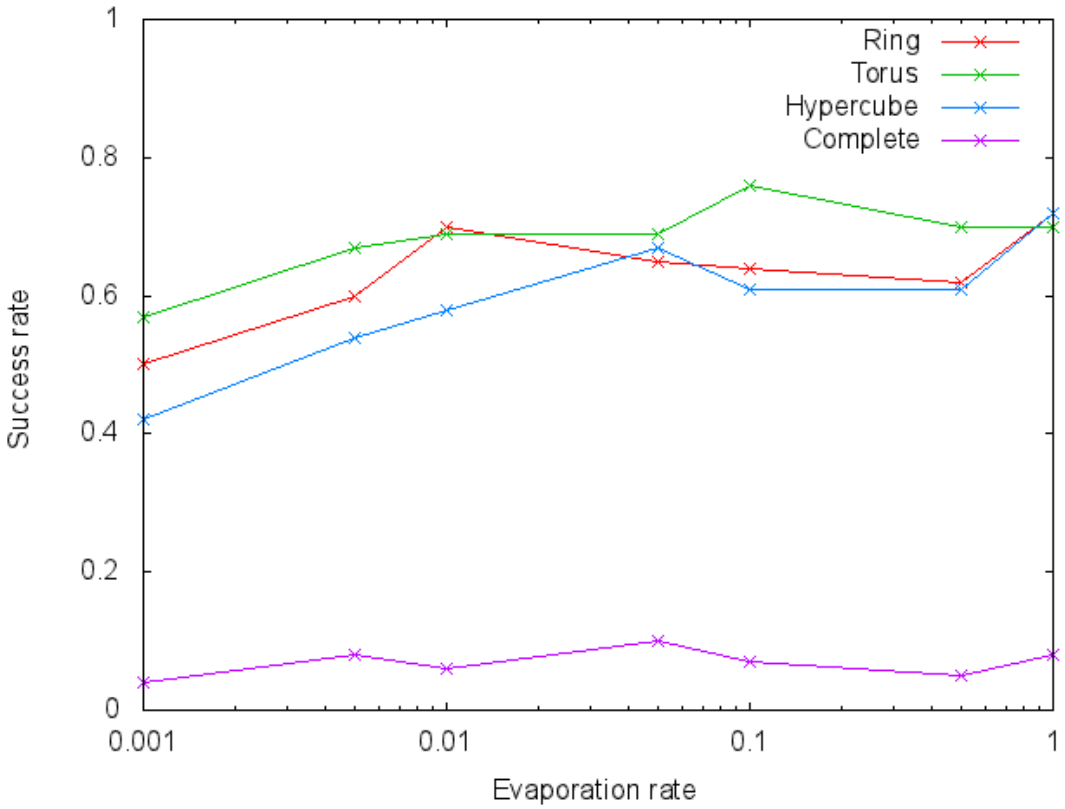



**Figure 14** The success rates on different evaporation rates. Note that the axis of Evaporation rate uses a logarithm scale.

**Figure 14** shows the success rates of all topologies in one graph. All topologies excluding complete graph have a trend that the success rate will decrease when the evaporation rate decreases under 0.01.

There is another significant trend that the average final generation number as well as the estimated number of generations to flip a zero bit at correct place will increase when the evaporation rate decreases. This can be explained with some reasonable assumptions. Consider a generation that a better solution $x$ is generated. Assume the bit-string is collecting leading ones in the current block to be optimized. The situation collecting leading zeros can be explained in a same method. Assume the first zero bit of $x$ is the $i$-th bit, then the probability to generate a better solution when the evaporation rate $\rho = 1$ is $\mathrm{P}_1 = (1 - 1/n)^{i-1}(1/n)$, which means the first $i - 1$ one-bits remain unchanged and the $i$-th bit is flipped. Note here the situation

that flip a sequence of ones to zeros within the threshold is ignored as the probability is too small compared to the probability to continue collecting ones.

When the evaporation rate is not large enough to update the pheromone from one side to another, the probability to generate a better solution becomes complicated. We start from easy cases and then extend to cover more cases. First, we assume all pheromones are fixed at the maximum or minimum value before $x$ is generated. $x$ at least flips the first zero bit ($j$-th bit) of the previous best found solution for a better fitness value. Thus the flipped bit ($j$-th bit) has a probability of $1/n(1-\rho)+\rho$ instead of $1-1/n$ to select one. Second, we assume all bits between $j$-th (excluding) bit and $i$-th (including) remain unchanged when $x$ is generated. Then the pheromone of these bits also remains unchanged and an intermediate estimate of the probability to generate a better solution is $(1-1/n)^{i-2}[1/n(1-\rho)+\rho](1/n)$.

Now we consider the situation that the $i$-th bit might be flipped from one to zero when $x$ is generated. In this case the pheromone of the $i$-th bit will be $(1-1/n)(1-\rho)$ after $x$ is generated if the pheromone is fixed before. Known the $i$-th bit in solution $x$ is zero, the probability that it is flipped from one to zero is $1/n$ and the probability that the it remains zero is $1-1/n$. Therefore, the expected pheromone of the $i$-th bit is $(1/\mathrm{n})(1-1/n)(1-\rho)+(1-1/n)(1/n)=(1/\mathrm{n})(1-1/n)(2-\rho)$ and a more precise estimate of the expected probability to generate a better solution is $\mathrm{P}_\rho{}' = (1-1/n)^{i-2}[1/n(1-\rho)+\rho](1/\mathrm{n})(1-1/n)(2-\rho) = (1-1/n)^{i-1}(1/\mathrm{n})[1/n(1-\rho)+\rho](2-\rho)$. Then we prove $\mathrm{P}_\rho^{'} < \mathrm{P}_1$:

$$
\begin{aligned}
& \mathrm{P}_\rho^{'} < \mathrm{P}_1 = (1-1/n)^{i-1}(1/n) \\
\text{iff}\quad & [1/n(1-\rho)+\rho](2-\rho) < 1 \\
\text{iff}\quad & (1-\rho+n\rho)(2-\rho) < n \\
\text{iff}\quad & (-1+2\rho-\rho^2)n + 2 - 3\rho + \rho^2 < 0 \\
\text{iff}\quad & (1-\rho)(2-\rho)/[(1+\rho)(1+\rho)] < n
\end{aligned}
$$

where $0 \le \rho \le 1, n = 1000$. Therefore $0 \le (1-\rho)(2-\rho)/[(1+\rho)(1+\rho)] \le 0.5 < n$. Therefore $\mathrm{P}_\rho^{'} < \mathrm{P}_1$.

Now we consider the possibility that the bits between $j$-th (excluding) bit and $i$-th (excluding) has changed from zero to one when $x$ is generated. In this case, the probability to select one is smaller than $1-1/n$ and then the expected probability to generate a better solution is smaller than $\mathrm{P}_\rho^{'}$.

For the situations that the pheromone is not fixed, if it is not the $j$-th or $i$-th bit, then the pheromone is less than the maximum value $1-1/n$ and the probability to generate a better solution is less than $\mathrm{P}_\rho{}'$. If it is the $j$-th or $i$-th bit, the pheromone value is larger and the probability to generate a better solution is more than $\mathrm{P}_\rho{}'$. But the bit must be flipped at a previous generation that generated a better solution, whose probability is $1/n$. Here the probability is under an assumption that all pheromones trend to either maximum value or minimum value, which is a phenomenon that empirically confirmed. Therefore we have a minor probability of $1/n$ to generate a better solution for a larger probability, while we have a major probability of $1-1/n$ to generate a better solution for a smaller probability in a same scale as the time and speed to flip the pheromone of a bit from either side to another

is completely same. Finally, the expected probability to generate a better solution is smaller than $\mathrm{P}_{\rho}^{'}$.

During the generations that does not generate a better solution, the pheromones will be updated according to the best found solution until the maximum or minimum value is reached. Then the probability to generate a better solution becomes $\mathrm{P}_1$. Therefore, for the evaporation rates that cannot flip the pheromone from one side to another in one generation, the expected probability to generate a better solution is smaller or equal to the probability when the evaporation rate is 1.

Then we will prove $\mathrm{P}_{\rho}^{'} = (1-1/n)^{i-1}(1/\mathrm{n})[1/n(1-\rho)+\rho](2-\rho)$ will decrease when $\rho$ decreases. $\mathrm{P}_{\rho}^{'}$ is only meaningful when $\rho \geq 0$ and $(1-1/n)(1-\rho) > 1/n$, which is $0 \leq \rho \leq 1-1/(\mathrm{n}-1)$. For larger $\rho$, the pheromone will flip from one side to another in one generation. Then the function $f(\rho) = \mathrm{P}_{\rho}^{'} = -(1-1/n)^{i-1}(1/\mathrm{n})((1-1/n)\rho+1/n)(\rho-2)$ is a quadratic function with maximum value at

$$\rho = \frac{\frac{-\frac{1}{n}}{1-\frac{1}{n}}+2}{2} = 1-\frac{1}{\mathrm{n}-1}$$

and it is monotonically increasing when $\rho < 1-1/(n-1)$. So $\mathrm{P}_{\rho}^{'}$ will decrease when $\rho$ decreases. Therefore, the expected number of generations to flip a zero bit at correct place will increase when the evaporation rate decreases.

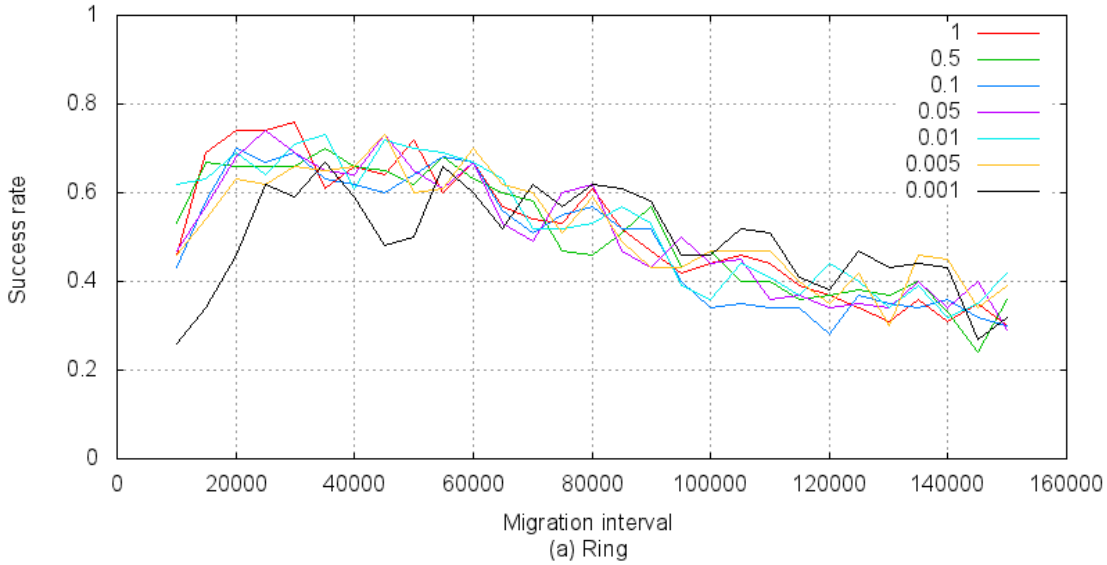
When the number of generations to flip a zero bit at correct place increases, the number of bits flipped in a given number of generations will decrease. We are wondering whether this has a similar influence of decreasing the migration interval. This idea will be covered by experiments in the next section.

## 4.4. Comparison of Migration Intervals on Different Settings

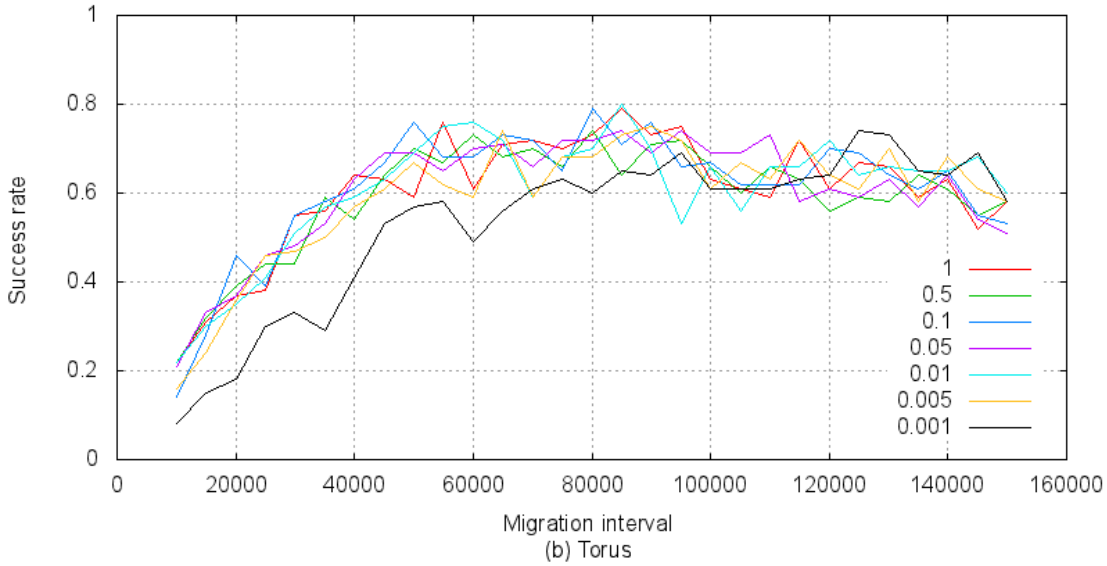
To study the performance of the MMAS* island model on different migration intervals, we ran a series of experiments covering all four nonempty topologies, migration intervals from 10000 to 150000 with an interval of 5000 and selected evaporation rates of 1, 0.5, 0.1, 0.05, 0.01, 0.005 and 0.001. All other parameters were same as experiments in previous section. The result is shown in **Figure 15**.

All topologies have a shared feature that the success rate will decrease when the migration interval decreases to a very small number. This is because the smaller migration interval, the faster a solution can spread to other islands. If the best found solution is stuck on collecting leading zeros (with a probability of 0.5 for each block), it is more likely to overtake other islands before a better solution that collection leading ones can be generated. In addition, a smaller migration interval makes a higher probability to have a migration generation happens when the problem is optimizing in a threshold of a block.

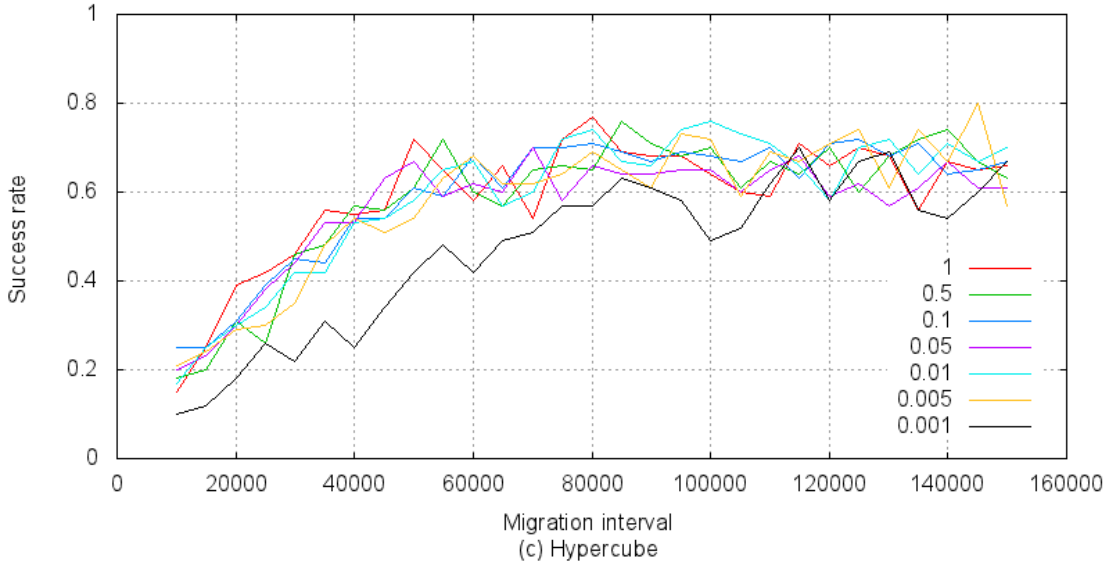
The most sparse topology ring has its best success rate when the migration interval is around 20000-30000, and it has the best success rate among all topologies on low evaporation rates. It is because the sparse topology slows down the solution spread speed, which gives more generations for slower islands to generate a better



(a) Ring



(b) Torus



(c) Hypercube

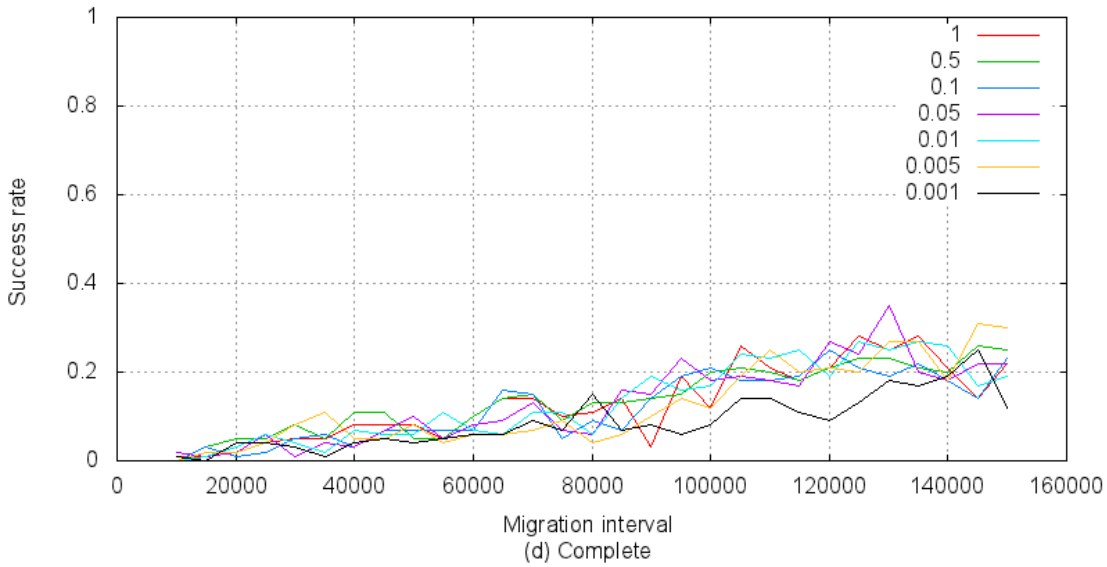



**Figure 15** The success rates on different migration intervals. The legends represent the evaporation rates used.

solution before being overtaken. For migration intervals larger than 40000, the success rate decreases significantly and becomes less than the success rate of torus and hypercube.

The torus and hypercube have a similar shape on the graph, while the success rate of torus topology has a slight decrease for migration intervals over 90000 and the hypercube topology remains a high success rate for migration intervals from 50000 to 150000.

The complete topology has a very low success rate for small migration interval and keeps increasing when the migration interval becomes larger. In paper [2], the authors mentioned that this may because increasing of the migration interval decreases the probability to migrate a stuck solution with higher fitness value to all other islands, which result in immediate stuck on all islands. In our opinion, this idea is correct, but there is a more important reason why the success rate for complete topology is very low and why it can be improved by increasing the migration interval.

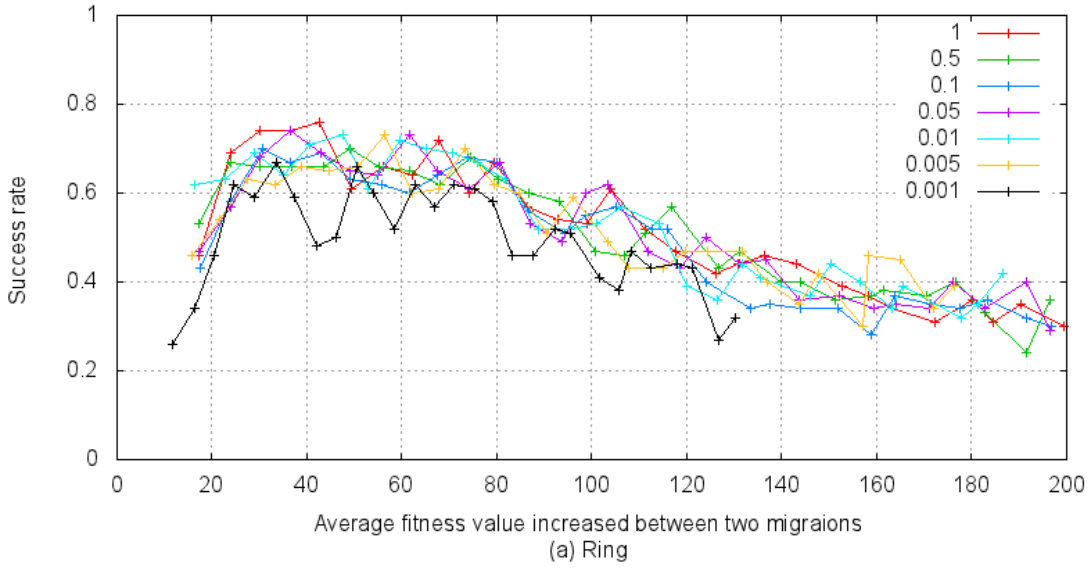
Consider the possible patterns of the first two bits of a block. It can be 00, 01, 10 and 11 with 25% probability each. Then we analyze the behavior of these patterns by assuming that all previous bits have optimized to ones and all pheromones are fixed with regards to the current best solution. For the patterns of 01 and 10, the solution trend to collect ones if the zero bit is flipped to one, or zeros if the one bit is flipped to zero with equal probability. For the pattern of 11, the probability that the bit-string is still collecting ones in the next better solution is at least $(n-1)$ times larger than the probability that the bit-string is collecting zeros in the next better solution. The reason is that it only needs to flip the first zero-bit to one-bit to continue collecting ones but it needs to flip at least two one-bits to zero-bits to start collecting zero. For the pattern of 00, the probability is reversed. This makes the patterns of 00 or 11 at the beginning of a block become very rare (with a probability at most $1/n$) to change its collecting bit. In the complete topology, all islands will have same solution and pheromone after a migration. Then all islands will trend to collect zeros and the algorithm will be stuck

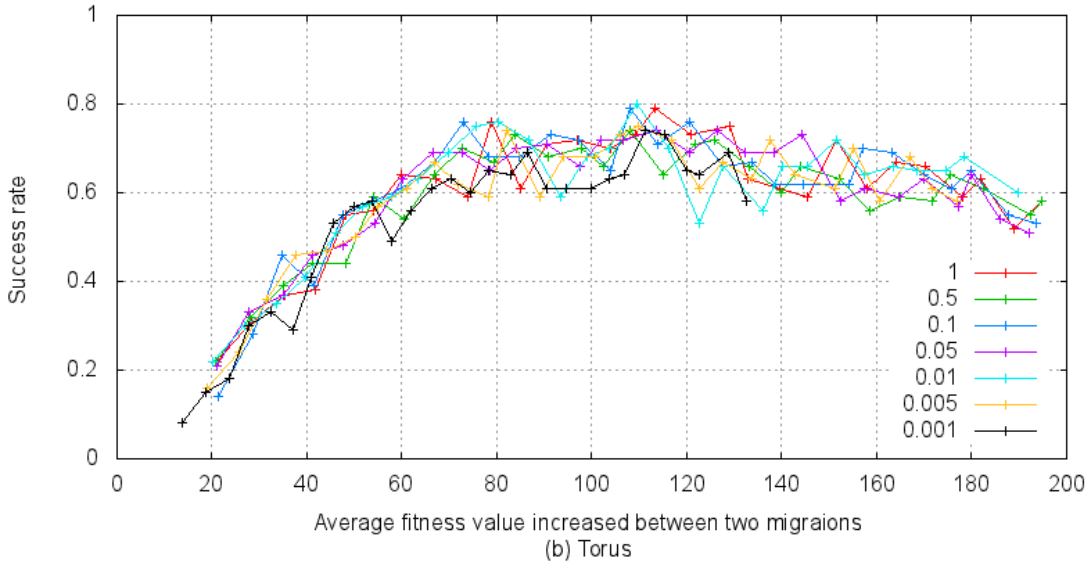
immediately at the block that begins with a 00 pattern. In the setting of solution size $n = 1000$, 10 blocks of 100 bits each and 32 islands, the probability that no island is collecting ones at the block that begins with a 00 pattern is at least $(1 - 1/1000)^{32} \approx 0.9685$. Then if the migration interval is not large enough, it is likely that the first migration will happen before the solution optimized to the second block. Assuming the patterns of 01, 10 and 11 would not cause stuck thank to the migration strategy, the expected success rate of the complete topology is at most $(1 - 25\% \times 0.9685)^9 \approx 0.0825$, which meets our empirical result in **Table 6**. When the migration interval increases, the probability that the first migration happens after the solution has optimized to the second block will increase, which result in a better expected success rate.

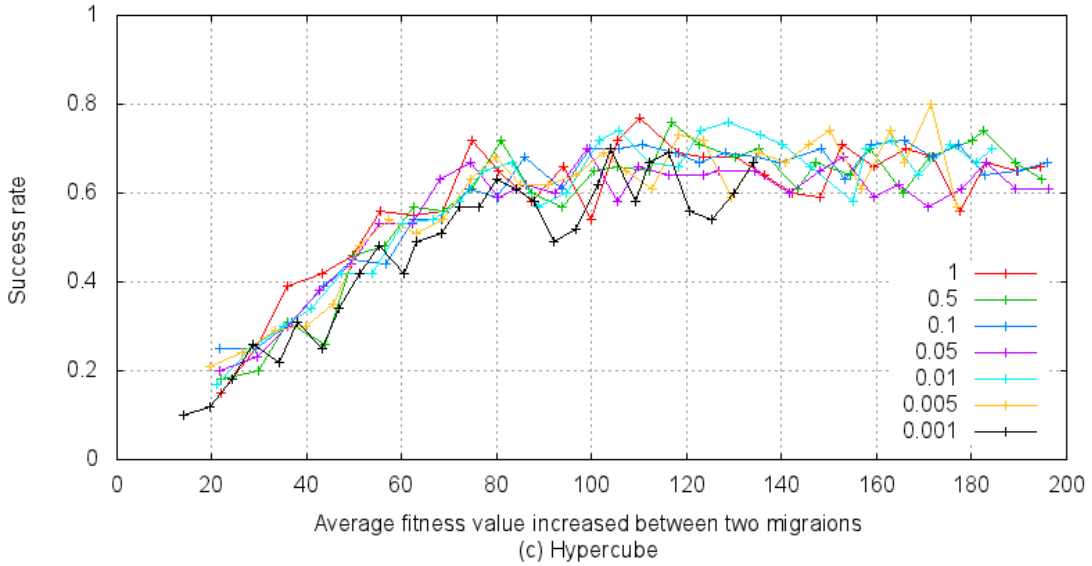
The influence by different evaporation rates is not very clear but the one with 0.001, especially on torus and hypercube topologies. On all topologies, the success rate of evaporation rate 0.001 is worse than other evaporation rates in low migration intervals. But in high migration intervals, it becomes similar on hypercube and complete topologies or even a little bit better on ring and torus topologies. If we left shift the line of evaporation rate 0.001 in **Figure 15**, it can fit in other lines with higher evaporation rates. This phenomenon recalls the assumption that decreasing the evaporation rate has a similar influence as decreasing the migration interval and both of them result in increasing the number of generations to flip a zero bit at correct place. Therefore we draw a series of graphs in **Figure 16** to show the relationship between them.

The x-axis value is the average fitness value increased between two migrations. It equals the migration interval divided by the average number of generations used to increase the fitness value by one, which is the average final generation number divided by the average final fitness value.

The experimental results show that there is no clear difference between different evaporation rates, especially the curve for evaporation rate 0.001 is not always away from other curves. Based on the analysis and experimental results above, we conjecture that in the LOLZ problem, the migration interval and the evaporation rate





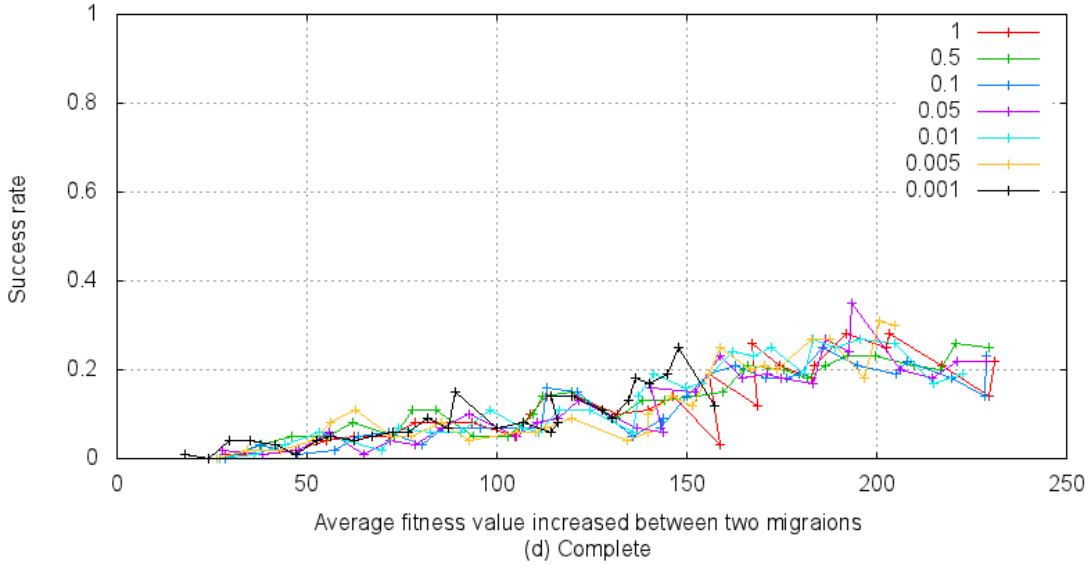




**Figure 16** The success rates on different settings that transformed to average fitness value increased between two migrations. The legends represent the evaporation rates used.

are two approaches to change the average fitness value increased between two migrations, which can determine the expected success rate in a given topology. In more detail, increasing the evaporation rate and/or increasing the migration interval will increase the average fitness value increased between two migrations, and vice versa.

## 4.5. Preliminary Experiment of Different Evaporation Rates on Different Islands

All above experiments have a same evaporation rate for all islands. It is general to consider whether a setting of different evaporation rates on different islands can help the optimization process.

There is an idea that different evaporation rates on different islands may increase the search speed of the ONEMAX problem. Unlike the LOLZ problem, OM always have a promising probability to generate a better solution and then the global best solution can be found in a reasonable generations. So we focus on the final generation number instead of the success rate which is always 1 for our algorithm. The idea needs two connected islands with high and low evaporation rate respectively. The island with low evaporation rate will generate solutions in a wide range in a more random way that easily get to a better solution at the beginning of the search process. Then the island with high evaporation rate will generate solutions more related to the best found solution. It makes more use of bits already set to one than island with small evaporation rate as few bits will different from the best found solution.

We ran a serial of experiments to try to confirm this idea. The problem is set to OM of $n = 1000$ and the algorithm will terminate when the global best solution is found. All experiments will run 100 times for an average result.

We start with comparative trials to get some basic data of the OM problem for later parameter selection and comparison. The first experiment is to run the algorithm on a single island, i.e. the classical MMAS* algorithm, for the evaporation rate from 0.01 to 1 with an interval of 0.01. The result is shown in **Figure 17**. We can see the final generation numbers are almost same for evaporation rate over 0.5. Then the algorithm becomes slower when the evaporation rate decreases. A similar result was also discovered in paper [30].

The second experiment is to run the algorithm on two connected islands for evaporation rate from 0.01 to 1 with an interval of 0.01. In addition, the interval of the migration becomes essential for the algorithm. A smaller migration interval can spread the better solution (as well as the pheromone) to another island faster. Then the islands can generate new solutions based on better information. But the migration generation does not generate any new solutions, so a larger migration interval strategy may benefits from generating more solutions than a smaller migration interval.

We ran a small test of migration intervals around a rough estimate to decide the migration interval for further experiments. As the expected initial solution value is 500

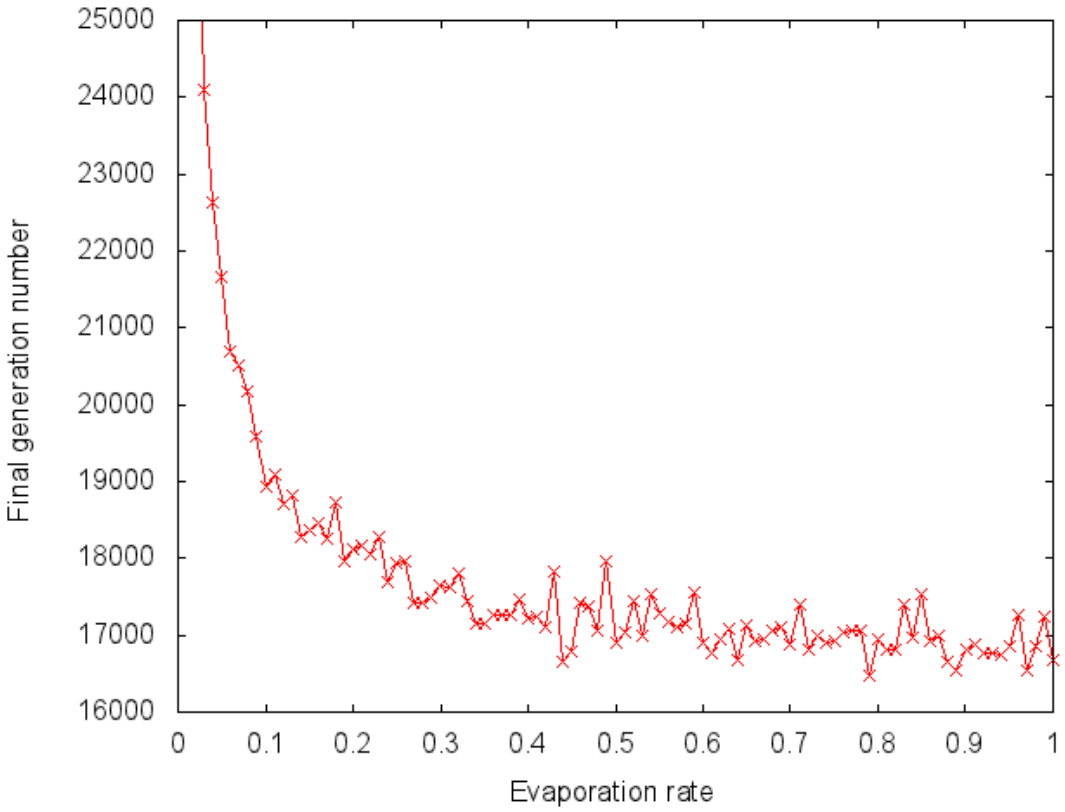



**Figure 17** The final generation number on different evaporation rates of single island OM. The final generation number when evaporation rate is 0.02 and 0.01 exceeds the range of this graph. They are 26965 and 34506 separately.

**Table 11** The final generation number (Final gen.) and total number of solutions generated (# of Solutions) of different migration interval and evaporation rates.

| Evaporation rate | 0.8 | | 0.2 | |
|---|---|---|---|---|
| Migration interval | Final gen. | # of Solutions | Final gen. | # of Solutions |
| 5 | 10950 | 17520 | 11810 | 18896 |
| 10 | 10245 | 18442 | 10905 | 19630 |
| 15 | 9947 | 18568 | 10978 | 20494 |
| 20 | 9934 | 18876 | 11095 | 21082 |
| 25 | 10130 | 19450 | 11161 | 21430 |
| 30 | 9920 | 19180 | 11291 | 21830 |
| 35 | 10108 | 19640 | 11027 | 21424 |
| 40 | 10616 | 20702 | 11451 | 22330 |

and the best solution is 1000, we want to have 500 migrations during the optimization. Furthermore, we have two islands generating solutions at same time, so the expected generations to generate a better solution are half of the generations for a single island. So the rough estimate migration interval $\tau = 17000/500/2 = 17$, where 17000 is taken from the single island experiment result. Around 17, the test contains the migration interval from 5 to 40 with an interval of 5. The evaporation rate is set to 0.8 and 0.2 which are faster and slower settings from the single island experiment. The result is shown in **Table 11**. The table contains the final generation number as well as

the total number of solutions generated which is calculated by removing the migration generations from the final generation number and then multiplies the number of islands. The number of solutions generated decreases when the migration interval is smaller, which meets our expectation. But the communication cost should not be ignored, so a migration interval of 20 is selected based on the final generation number as a balance of migration and generating new solutions. **Figure 18** shows the result of the experiment of evaporation rates on two connected islands with migration interval $\tau = 20$. It has a similar shape with **Figure 17**, but the final generation number is much smaller due to the acceleration by an additional island and migration.

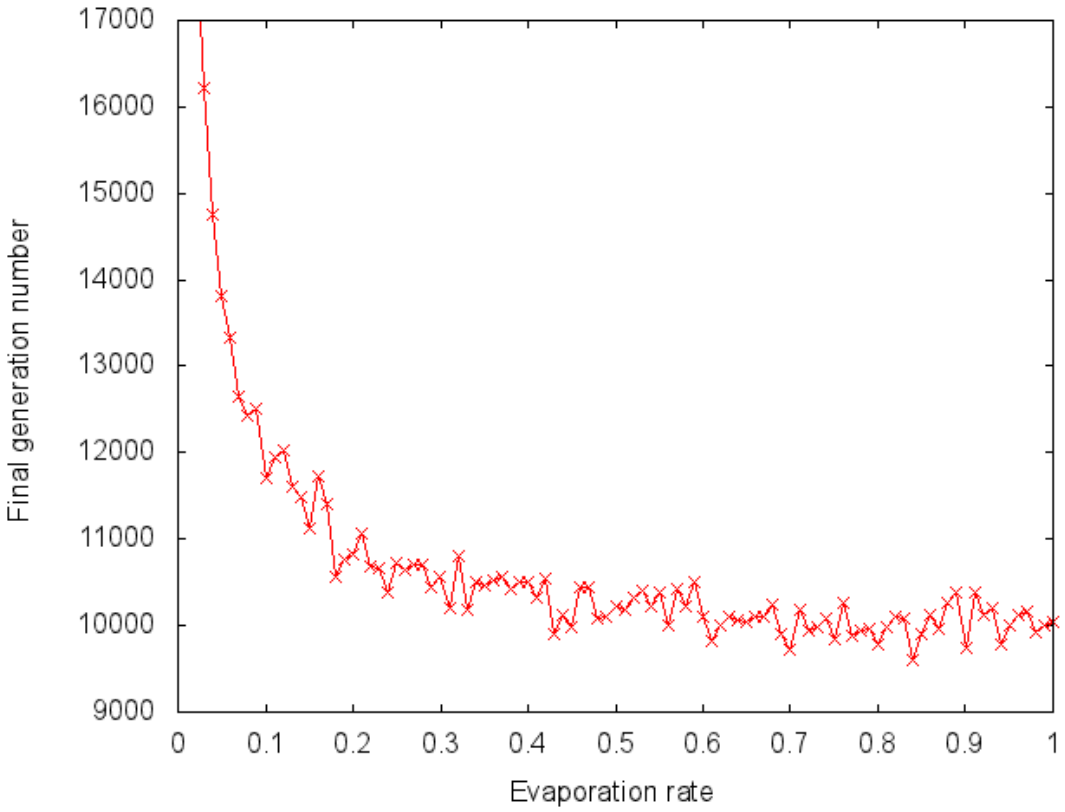



**Figure 18** The final generation number of different evaporation rates of 2 islands OM. The final generation number when evaporation rate is 0.02 and 0.01 exceeds the range of this graph. They are 18395 and 24833 separately.

With above results, we start the experiment of two islands with high and low evaporation rates separately. For the island with high evaporation rate, 0.5 and 1 is selected as they are the low end and the high end of the evaporation rate settings that outperform others in single and two island experiments. The island with low evaporation rate is traversed from 0.01 to 0.09 with interval of 0.01 and 0.1 to the high evaporation rate on another island with interval of 0.1. We first ran the experiments with migration interval $\tau = 20$, but the result did not meet our expectation that two island with high and low evaporation rate can perform better. Then we ran the experiments with migration interval $\tau = 10$ for confirmation. The result is shown in **Figure 19**. Both migration intervals have a similar shape that the final generation number decreases when the lower evaporation rate increases, while the group with high evaporation rate 1 performs better than the group with 0.5. None of the settings

we have tried can outperform the 2 island model with migration of 20 in a high evaporation rate setting (0.5~1). The best final generation number we have found in this experiment is 9928 and the best final generation number from two island with same evaporation rate is 9726. Though our expectation not confirmed in the experiments, there is a notable improvement for low evaporation rate settings after replacing a low evaporation rate island with a high evaporation rate island.

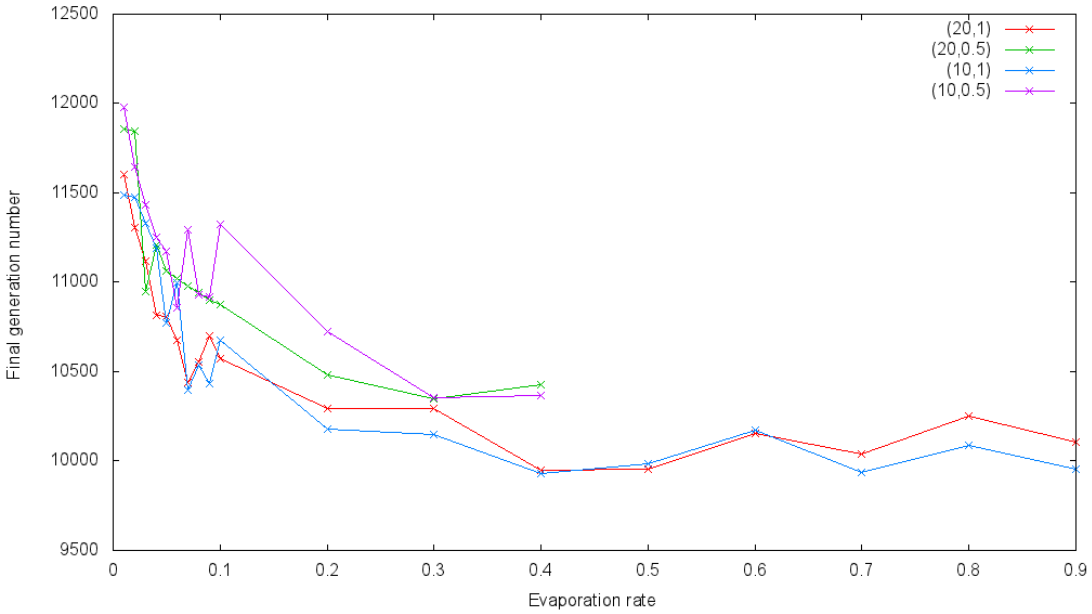



**Figure 19** The final generation number of different evaporations on 2 islands. The legend represents the migration interval and the higher evaporation rate while the x-axis is the lower evaporation rate.

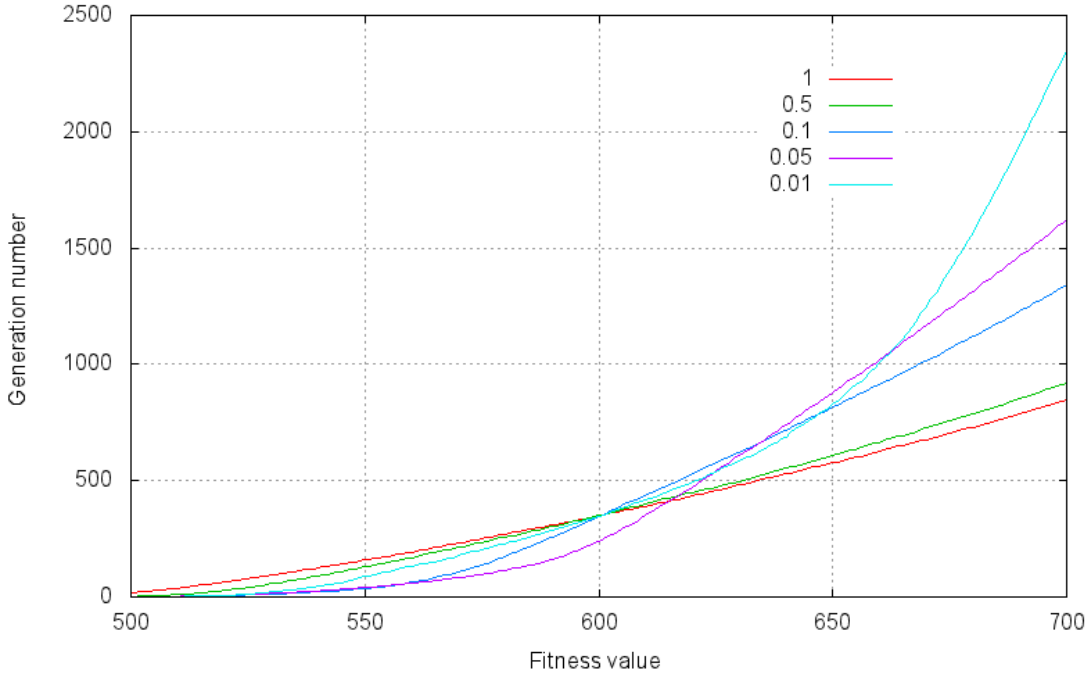



**Figure 20** The average generation number to reach fitness value between 500 and 700.

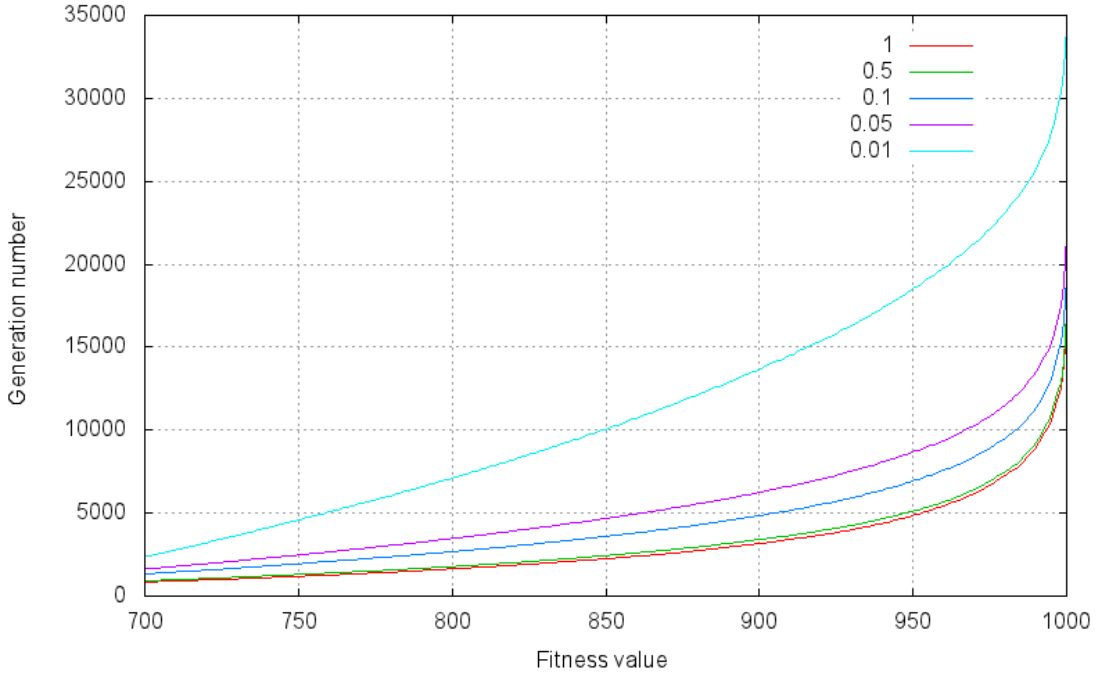



**Figure 21** The average generation number to reach fitness value between 700 and 1000.

The reason why the settings of different evaporation rates on different islands cannot outperform a good setting of two islands with same evaporation rate as well as the great improvement for low evaporation rates finally summarized to a worse performance of the island with low evaporation rate.

To explain this phenomenon, we redo some single island experiments and record the average generation number to reach new fitness values. The result is shown in **Figure 20** and **Figure 21**.

The graphs show that a low evaporation rate does search faster in the very early generations but later becomes much less efficient. It is apparently that a setting that can find better solutions faster after the best fitness value reaches 600 is more important as most of generations are used for solutions after 600. Therefore for two connected islands with low and high evaporation rates separately, the low evaporation rate island does helps to accelerate the search process in very early generations. But it soon becomes less efficient, i.e. the expected number of generations needed to generate a better solution for low evaporation rate island is more than the number for high evaporation island. Thus the setting of two connected islands with low and high evaporation rates separately loses its advantage obtained in the beginning generations, and finally costs more generations to reach the global best solution.

Although we could not confirm the advantage for the whole run of the algorithm, it provides an inspiration that different evaporation rates on different islands may helps the LOLZ problem to be optimized. First, the migration topology is directional only from lower evaporation islands to higher evaporation islands instead of bidirectional. As a lower evaporation rate islands are expected to be slower than the higher evaporation rate island, if a migration happened, it is likely that the higher evaporation rate island is stuck on a local optimal from some previous generations and get out of the local optimal after the migration, otherwise the higher evaporation rate island is

expected to have a better solution than the lower evaporation rate island. The advantage of this setting is preventing stuck but higher fitness value solutions migrated from higher evaporation rate islands to lower evaporation rate islands. The disadvantage of this setting is that few or even no islands can migrate to a low evaporation rate island, which makes them being stuck easily. Due to time limitations, the theoretical and experimental confirmation and improvement of this idea is left for further work.

# 5. Conclusion

By our empirical research, there are several interesting experimental results discovered regarding the parallel MMAS* model. First, it can be as good as the parallel (1+1) EA with migration on LOLZ problem. Though the success rate will decrease when the evaporation rate is under 0.01 scale when migration interval is 50000, it is still much better than algorithms without migration. Second, the parallel MMAS* model with low evaporation rate costs more generations to correct a bit on average compare to the parallel (1+1) EA with migration. This is discovered on both LOLZ and OM problem. The reason why this happens is that the expected probability to generate a better solution for parallel MMAS* model is smaller than parallel (1+1) EA with migration when the pheromone is changing. Third, in the LOLZ problem, the migration interval and the evaporation rate are two approaches to change the average fitness value increased between two migrations, which can determine the expected success rate in a given topology. Fourth, the setting of two connected island with low and high evaporation rates separately cannot be better than two connected island with both high evaporation rates on OM problem. This is caused by the low evaporation rate island performing worse than the high evaporation rate island in most of time.

Future work could deal with the theoretical analysis on the parallel MMAS* model that reveals the insight into how evaporation rate affect the optimization process. It would also be interesting to consider the setting of different evaporation rates on different islands or an extension of the algorithm that allow changing migration interval during the optimization and whether these modification can bring benefit to LOLZ or other problems.

# Acknowledgement

The author would like to thank my supervisor Carsten Witt for the many useful comments and suggestions through whole thesis. I also thank my parents for suggestions on the last draft.

# Reference

1. Alba, E., Cotta, C.: Evolutionary algorithms. In Olariu, S., Zomaya, A.Y., eds.: Handbook of Bioinspired Algorithms and Applications. Chapman & Hall/CRC (2006) 3{19}.
2. Lässig, J., Sudholt, D.: The benefit of migration in parallel evolutionary algorithms. In: Proceedings of GECCO 2010, pp. 1105–1112. ACM, New York (2010).
3. Lässig, J., Sudholt, D: Experimental Supplements to the Theoretical Analysis of Migration in the Island Model. In: PPSN XI, Part I, LNCS 6238, pp. 224–233. Springer (2010).
4. Cant'u-Paz, E., Goldberg, D.E.: On the scalability of parallel genetic algorithms. Evolutionary Computation 7(4), 429–449 (1999).
5. Frank Neumann, Carsten Witt: Bioinspired computation in combinatorial optimization : algorithms and their computational complexity. Natural Computing Series, Springer, ISBN 978-3-642-16543-6.
6. Thomas Stützle, Holger H. Hoos : MAX–MIN Ant System. In: Future Generation Computer Systems 16, 889–914 (2000).
7. Neumann, Frank • Sudholt, Dirk • Witt, Carsten: Analysis of different MMAS ACO algorithms on unimodal functions and plateaus. In: SWARM INTELLIGENCE — 2009, Volume 3, Issue 1, pp. 35-68.
8. I. Rechenberg. Evolutionsstrategie: Optimierung technischer Systemenach Prinzipien der biologischen Evolution. Frommann-Holzboog Verlag, Stuttgart, 1973.
9. J.H. Holland. Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Harbor, 1975.
10. L.J. Fogel, A.J. Owens, and M.J. Walsh. Artificial Intelligence Through Simulated Evolution. Wiley, New York, 1966.
11. J.R. Koza. Genetic Programming. MIT Press, Cambridge MA, 1992.
12. M. Dorigo, V. Maniezzo, A. Colorni: The ant system: optimization by a colony of cooperating agents, IEEE Trans. Systems Man Cybernet. B 26 (1996) 29–42.
13. S. Chatterjee, C. Carrera, and L. Lynch. Genetic algorithms and traveling salesman problems. European Journal of Operational Research, 93(3):490-510, 1996.
14. C.C. Aggarwal, J.B. Orlin, and R.P. Tai. Optimized crossover for the independent set problem. Operations Research, 45(2):226-234, 1997.
15. C.H. Chu, G. Premkumar, and H. Chou. Digital data networks design using genetic algorithms. European Journal of Operational Research, 127:1400158, 2000.
16. N. Swaminathan, J. Srinivasan, and S.V. Raghavan. Bandwidth-demand prediction in virtual path in atm networks using genetic algorithms. Computer Communications, 22(12):1127-1135, 1999.
17. A.J. Urdaneta, J.F. Gómez, E. Sorrentino, L. Flores, and R. Díaz. A. hybrid genetic algorithm for optimal reactive power planning based upon successive linear programming. IEEE Transactions on Power Systems, 14(4):1292-1298, 1999.
18. M. Guotian and L. Changhong. Optimal design of the broadband stepped impedance transformer based on the hybrid genetic algorithm. Journal of Xidian University, 26(1):8-12, 1999.
19. Gutjahr, W.J.: On the Finite-Time Dynamics of Ant Colony Optimization. Methodology and Computing in Applied Probability, 8(1), 105–133 (2006).
20. Gutjahr, W. J., & Sebastiani, G.: Runtime Analysis of Ant Colony Optimization with Best-So-Far Reinforcement. Methodology and Computing in Applied Probability, 10, 409–433 (2008).
21. Jörg Lässig, Dirk Sudholt: General Upper Bounds on the Running Time of Parallel

Evolutionary Algorithms. arXiv:1206.3522, 2012.

22. Frank Neumanna, Carsten Witt: Ant Colony Optimization and the minimum spanning tree problem. THEORETICAL COMPUTER SCIENCE, 2010, Volume 411, Issue 25, pp. 2406-2413.
23. Timo Kötzing, Frank Neumann, Heiko Röglin, Carsten Witt: Theoretical analysis of two ACO approaches for the traveling salesman problem. SWARM INTELLIGENCE — 2012, Volume 6, Issue 1, pp. 1-21.
24. TSPLIB. http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/.
25. Mario F. Triola: Elementary Statistics. 10th Edition. Pearson Education (2007), ISBN 9780321522917, 320-331.
26. Java Class Random. http://docs.oracle.com/javase/6/docs/api/java/util/Random.html.
27. George Marsaglia: Xorshift RNGs. Journal of Statistical Software. 2003, Vol. 8, Issue 14.
28. Matsumoto, M.; Nishimura, T.: Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation 8 (1): 3–30. doi:10.1145/272991.272995 (1998).
29. COLT. http://acs.lbl.gov/software/colt/.
30. Timo Koetzing, Frank Neumann, Dirk Sudholt, Markus Wagner: Simple Max-Min Ant Systems and the Optimization of Linear Pseudo-Boolean Functions. FOGA 11: PROCEEDINGS OF THE 2011 ACM/SIGEVO FOUNDATIONS OF GENETIC ALGORITHMS XI — 2011, pp. 209-218.