Dynamic Fuzzy Logic Control of Genetic Algorithm Probabilities
Abstract

Genetic algorithms are commonly used to solve combinatorial optimization problems. The implementation evolves using genetic operators (crossover, mutation, selection, etc.). Anyway, genetic algorithms like some other methods have parameters (population size, probabilities of crossover and mutation) which need to be tune or chosen.

In this paper, our project is based on an existing hybrid genetic algorithm working on the multiprocessor scheduling problem. We propose a hybrid Fuzzy-Genetic Algorithm (FLGA) approach to solve the multiprocessor scheduling problem. The algorithm consists in adding a fuzzy logic controller to control and tune dynamically different parameters (probabilities of crossover and mutation), in an attempt to improve the algorithm performance. For this purpose, we will design a fuzzy logic controller based on fuzzy rules to control the probabilities of crossover and mutation. Compared with the Standard Genetic Algorithm (SGA), the results clearly demonstrate that the FLGA method performs significantly better.
Acknowledgement

Before accessing this thesis, I would like to express my gratitude to my supervisor, Dr. Pascal Rebreyend, for his constant and invaluable guidance, encouragement throughout all phases of my research work.

I also like to express my sincere thanks to Prof. Mark Dougherty, Dr. Hasan Fleyeh and Dr. Yella Siril. I learn a lot from your teaching. You are the best teacher for me in Sweden.

Thanks to ShengTong Zhong, Yang Shen, and Ping Zhao. You were, are, and always will be my best friends.

Special thanks to you, you give me the most help on my very hard time.
Dedicated...

To My Dear Parents!

我最敬爱的父母！

冯毅

2008 于瑞典
# Contents

1. Introduction ........................................................................................................................... 1

2. Problem Statement ................................................................................................................ 2
   2.1 Rudiments .................................................................................................................... 2
   2.2 Formulation ................................................................................................................. 3

3. Genetic Algorithm ............................................................................................................... 7
   3.1 Basal principle of Genetic algorithm ........................................................................... 7
   3.2 Coding of Individual ................................................................................................. 12
   3.3 Initial Population .................................................................................................... 14
   3.4 Evaluate Fitness ...................................................................................................... 15
   3.5 Selection .................................................................................................................. 15
   3.6 Crossover ................................................................................................................ 15
   3.7 Mutation .................................................................................................................. 17
   3.8 Termination condition............................................................................................. 18

4. Genetic algorithm with Fuzzy Logic ................................................................................... 19
   4.1 The basic concepts of Fuzzy Logic .......................................................................... 19
   4.2 Genetic Algorithm with fuzzy logic ........................................................................... 28
   4.3 The first strategy for fuzzy rules: ............................................................................... 30
   4.4 The second strategy for fuzzy rules: .......................................................................... 37

5. Experiment results, comparisons and discussion ................................................................39

6. Conclusion & Further research ............................................................................................ 50

References ............................................................................................................................... 52
1. Introduction

Genetic Algorithms have been widely used for classification, model selection, and other optimization tasks. However, the performance of genetic algorithms is largely affected by the values of their input parameters, such as the population size, probabilities of crossover and mutation. While poor parameter settings usually lead to several problems such as the premature convergence, local optimum, etc.

The multiprocessor scheduling problem consists of finding a task schedule that minimizes the execution time of the parallel program and the number of required processors. In this paper, we use a Fuzzy Logic Controller (FLC) to adapt dynamically the crossover and mutation probabilities in order to get a better performance.

Developing the fuzzy logic controller is based on some fuzzy rules. We design these rules according to the consideration of improving fitness.

The structure of this paper is as follows.

- As a starting point, introduce the multiprocessor scheduling problem in detail, show what problem we are facing in Section 2.


- In Section 4, we will introduce the basic concepts of Fuzzy Logic [1] and design our improved genetic algorithm by adding a fuzzy logic controller in the standard genetic algorithm.

- Introduce the benchmark for testing, comparing the experiment results and discussions are provided in Section 5.

- Finally, conclude our work and propose the further research.
2. Problem Statement

This project is based on an existing hybrid genetic algorithm working on the multiprocessor scheduling problem. Therefore, as a starting point, we first need to elaborate the multiprocessor scheduling problem itself.

2.1 Rudiments

Let a multiprocessor system be a set of m identical processors, m > 1. Each processor has its own memory, and each pair of processors communicate exclusively by message passing through an interconnection network. In addition, let a parallel program be a set of communicating tasks to be executed under a number of priority constraints.

Each task is associated a cost, representing its execution time. In order to be executed, each task of a given parallel program must be scheduled to some processor of a given multiprocessor system. In general, different schedules may need different numbers of processors; these communications can slow down the execution of the parallel program. So, different schedules of each task satisfying the priority constraints lead to different execution time of the parallel program. That is why we want to minimize the execution time of the parallel program in order to optimize this problem.

The multiprocessor scheduling problem is a kind of typical optimization problems, it has been extensively studied by a large number of researchers. Because an exhaustive search is often unrealistic, most of the work has been done on heuristic methods to find near-optimal solutions. There are two most studied heuristic methods for multiprocessor scheduling problem: list heuristic [5] and meta-heuristic. The meta-heuristic is known as genetic algorithm.

In this paper, we just use the genetic algorithm to solve the multiprocessor scheduling problem. A genetic algorithm is a guided random search method where
elements (called *individuals*) in a given set of solutions (called *population*) are randomly combined and modified (we call these combinations *crossover* and *mutation*, respectively) until some termination condition is achieved. The population evolves iteratively (in the genetic algorithm terminology, through *generations*) in order to improve the fitness of its individuals. The fitness of an individual $s_i$ is said to be better than the fitness of another individual $s_j$ if the solution corresponding to $s_i$ is closer to an optimal solution than $s_j$. In each iteration, the crossover generates a new population in which the individuals are supposed to keep the good characteristics of the individuals of the previous generation.

In the context of multiprocessor scheduling problem, Hou, Ansari and Ren [6], proposed a kind of pure genetic algorithm, in this algorithm, the individuals are coded based on strings. Actually, this method is an original pure genetic algorithm; it has some drawbacks. However, in this paper, we would prefer to concentrate on the genetic operators (*crossover* and *mutation*). That means the problem itself is not the main point of this project, we only need to study which operators should be used and how to control them to make a better performance.

### 2.2 Formulation

In order to formalize the multiprocessor scheduling problem, we first define a multiprocessor system and a parallel program. A multiprocessor system is composed of a set $p = \{ p_1, ..., p_m \}$ of $m$ identical processors. They are connected by a complete communication network, where all links are identical. Each processor can execute at most one task at a time and task preemption is not allowed. While computing, a processor can communicate through one or several of its links.

The parallel program is described by an acyclic digraph $D = (T, A)$. The vertices represent the set $T = \{ t_1, ..., t_n \}$ of tasks and each arc represents the priority relation between two tasks. An arc $(t_i, t_j) \in A$ represents the fact that at the end of its execution, $t_i$ sends a message to $t_j$ to start execution. In this case, $t_i$ is said to be
an immediate predecessor of \( t_{i_2} \), and \( t_{i_2} \) itself is said to be an immediate successor of \( t_{i_1} \). We suppose that \( t_1 \) is the only task without any immediate predecessor. A path is a sequence of nodes \(<t_{i_1}, \ldots, t_{i_n}>\), \((1 < k \leq n)\), such that \( t_{i_k} \) is an immediate predecessor of \( t_{i_{k+1}} \), \((1 \leq l < k)\). More general, a task \( t_{i_l} \) is a predecessor of another task \( t_{i_k} \) if there is a path \(<t_{i_1}, \ldots, t_{i_k}>\) in \( D \).

To every task \( t_{i_l} \), there is an associated value representing its duration, and we assume that these values are known before the execution of the program. In addition, the duration of all the communications as also known at compile-time. Thus, to every arc \((t_{i_1}, t_{i_2}) \in A\), there is an associated cost representing the transfer time of the message from \( t_{i_1} \) to \( t_{i_2} \). If both message source and destination are scheduled to the same processor, then the cost associated to this arc becomes null.

A schedule is a vector \( s = (s_1, \ldots, s_n) \), each \( s_k = \{t_{i_1}, \ldots, t_{i_{n_k}}\} \) is the set of the \( n_k \) tasks scheduled to \( p_k \) (the \( k \)th processor). Furthermore, for each task \( t_{i_l} \), \( p(t_{i_l}, s) \) and \( r(t_{i_l}, s) \), respectively, is the processor and the rank in this process of \( t_{i_l} \) under the schedule \( s \). The principle of the algorithm is to schedule each task \( t_{i_l} \) to \( p(t_{i_l}, s) \) according to its rank \( r(t_{i_l}, s) \).

Each task has an associated execution time, and every pair of two tasks has an associated transfer time. The total execution time yielded by a schedule is called makespan. It is clear that our job is to minimize the makespan.

\[
\text{makespan} = \text{execution time} + \text{transfer time}
\]

As an example, let’s first illustrates a schedule and the introduction data computed by a list heuristic. Figure 1 shows the acyclic digraph \( D \) is composed of six tasks, while Figure 2 shows the multiprocessor system is composed of three processors fully interconnected. Each \( d_{i_l} \) indicates the cost of \( t_{i_l} \) and each \( c(t_{i_{l_1}}, t_{i_{l_2}}) \)
represents the communication time associated to the arc \((t_i, t_j)\). A diagram representing the details of a schedule \(s\) is also shown by Figure 3. For instance, in this diagram, we can see that \(t_1\) is scheduled on processor \(p_1\) at the time interval \([0...2]\), \(t_2\) is scheduled on processor \(p_2\) at the time interval \([4...6]\), \(t_3\) is scheduled on processor \(p_3\) at the time interval \([4...5]\), and so on.

Figure 1: Acyclic Diagraph \(D = (T, A)\)

Figure 2: Multiprocessor system
Figure 3: Details of a schedule $S$

Where,

Arc costs,

$$c(i_1, i_2) = 2, \forall t_{i_1}, t_{i_2} \in T$$

Task durations

$$d(i) = \begin{cases} 
1, & i = 3, 5 \\
2, & i = 1, 2, 4, 6
\end{cases}$$

Schedule $s$:

$$\{\{t_1, t_4, t_5, t_6\}, \{t_2\}, \{t_3\}\}$$
3. Genetic Algorithm

We will use genetic algorithm inside our approach to solve the problem, so introducing the genetic algorithm firstly is necessary.

3.1 Basal principle of Genetic algorithm

Genetic algorithms are a particular class of evolutionary algorithms that use techniques inspired by evolutionary biology such as inheritance, mutation, selection, and crossover. It has been widely used in computing to find exact or approximate solutions to optimization and search problems.

Genetic algorithms are implemented as a computer simulation in which a population of abstract representations (called chromosomes or the genotype or the genome) of candidate solutions (called individuals, creatures, or phenotypes) to an optimization problem evolves toward better solutions. Traditionally, solutions are represented in binary as strings of 0 and 1, but other encodings are also possible. The evolution usually starts from a population of randomly generated individuals and happens in generations. In each generation, the fitness of every individual in the population is evaluated, multiple individuals are stochastically selected from the current population (based on their fitness), and modified (recombined and possibly randomly mutated) to form a new population. The new population is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. If the algorithm has terminated due to a maximum number of generations, a satisfactory solution may or may not have been reached.

A typical genetic algorithm requires two things to be defined:

1. A genetic representation of the solution domain.
2. A fitness function to evaluate the solution domain.
A standard representation of the solution is as an array of bits. Arrays of other types and structures can be used in essentially the same way. In our case, we use the method that is proposed by Hou, Ansari and Ren [6].

The fitness function is defined over the genetic representation and measures the quality of the represented solution. The fitness function is always problem dependent. In our case, the fitness of an individual is defined as the difference between its makespan and the one of the individuals having the largest makespan in the population. So the best individual correspond to the one having the smallest makespan and the largest fitness.

Once we have the genetic representation and the fitness function defined, GA proceeds to initialize a population of solutions randomly, and then improve it through repetitive application of mutation, crossover, and selection operators.

**Initialization**

Initially many individual solutions are randomly generated to form an initial population. The population size depends on the nature of the problem, but typically contains several hundreds or thousands of possible solutions. Traditionally, the population is generated randomly, covering the entire range of possible solutions (the search space). Occasionally, the solutions may be "seeded" in areas where optimal solutions are likely to be found.

**Selection**

During each successive generation, a proportion of the existing population is selected to breed a new generation. Individual solutions are selected through a fitness-based process, where fitter solutions (as measured by a fitness function) are typically more likely to be selected. Certain selection methods rate the fitness of each solution and preferentially select the best solutions. Other methods rate only a random sample
of the population, as this process may be very time-consuming.

Most functions are stochastic and designed so that a small proportion of less fit solutions are selected. This helps keep the diversity of the population large, preventing premature convergence on poor solutions. Popular and well-studied selection methods include roulette wheel selection and tournament selection. In our case, we will use roulette wheel selection with a certain selection rate.

Reproduction

The next step is to generate a second-generation population of solutions from those selected through genetic operators: crossover (also called recombination), and mutation.

Crossover is usually applied to selected pairs of "parents" with a probability equal to a given crossover rate, and generate two new individuals by crossing the parents characteristics. Hence, the "Childs" share many of the characteristics of its "parents". Normally, there are several ways to go in crossover procedure like one-point crossover, two-point crossover and order crossover.

Mutation is used to modify an individual with a certain mutation rate.

After crossover and mutation, we desire that some "good" individuals will survive and reproduce, while some "bad" individuals will be eliminated. Generally, the average fitness will have increased by this procedure for the population, since only the best organisms from the first generation are selected for breeding, along with a small proportion of less fit solutions, for reasons already mentioned above.
Termination

Repeat the steps mentioned above, we will get some individuals with better and better fitness from generation to generation. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population. In our case, we set a maximum number of generations to stop terminate the algorithm even if there is not a satisfactory solution.
The general structure of the standard genetic algorithm is as follows:

According to the basic principle of our approach to solve the multiprocessor scheduling problem, we will design our pure genetic algorithm in the following.
3.2 Coding of Individual

The coding of an individual $s$ (a schedule) is composed of $k$ strings \{$s_1, s_2, \ldots, s_k$\}. Each string $s_j$ represents the tasks scheduled to processor $p_j$ in $s$. So, the number of strings in $s$ should be equal to the number of processor in $s$. From Figure 3 we can see an example of a coding for three processors, the corresponding number of strings is also three.

However, the coding scheme has a weak point; the tasks represented by the strings may be not satisfying the priority constraints. For this reason, [6] proposed a method that make sure all strings in the initial population or reproduced by crossover and mutation will correspond to feasible schedules. This method is based on the concept of height of tasks. Let $t_i$ be a task, $h_p(t_i)$ be the maximum length of a path between $t_i$ and an immediate predecessor of $t_i$, and $h_s(t_i)$ be the maximum length of a path between $t_i$ and an immediate successor of $t_i$. Each task $t_i$ is then assigned a random height whose value is such that $h_p(t_i) < \text{height}(t_i) < h_s(t_i)$.

According to the concepts of heights, the tasks whose height is smaller should be ordered before the tasks whose height is larger. The heights of tasks can help to keep the priority relation, so, we can guarantee the feasibility of a given schedule.

As an another example, we use the program described in Figure 4 below, where tasks 1 to 6 and 8 to 10 have execution time 1, and task 7 has execution time 10. Suppose that we have two processors ($p_1, p_2$), and the communication times are null.
Figure 4: Another example of Acyclic Diagraph $D = (T, A)$

From this picture, we can clearly see the priority constrains. Task 10 is the last one that should be scheduled to processor.

According to our algorithm proposed by [6], we have:

\[
\begin{align*}
\text{height}(1) &= \text{height}(2) = \text{height}(3) = 0 \\
\text{height}(4) &= \text{height}(5) = \text{height}(6) = 1 \\
\text{height}(7) &= \text{height}(8) = \text{height}(9) = 2 \\
\text{height}(10) &= 3.
\end{align*}
\]

However, one thing we must know: there is a severe drawback in the algorithm proposed by [6], some feasible solutions could not be produced. The problem itself is not the main point of this paper, but the results will influence the
comparison between pure algorithm and our improved algorithm. Therefore, we need
discuss about the drawback of the pure algorithm.

For instance, let’s use the program described in Figure 4 again. As will be
easily seen, there is an optimal schedule: \( s = \{ \{1,4,7,10\}, \{2,5,8,3,6,9\} \} \). The makespan
of this schedule is 13. However, by respecting the priority constraint over the heights
as in [6], task 3 (of height 0) is necessarily scheduled before the task of height 1. In
our example, task 3 should be scheduled before task 4 and task 5. In this case, task 7
has to be delayed since task 4 also will be delayed. Hence, the makespan can never
equal 13 time units. That means the method in [6] may not miss some or all optimal
solutions for the multiprocessor scheduling problem.

3.3 Initial Population

As mentioned before, the initial population is randomly generated, the tasks
being scheduled to the processor according to their heights. For instance, let \( T(h) \) be
the set of tasks with height \( h \) in \( D \). For each height \( h \), the following steps are
performed. Choose at random \( r \) tasks, \( 0 \leq r \leq |T(h)| \), from \( T(h) \) to be assigned to \( p_1 \).
Then, remove these \( r \) tasks from \( T(h) \) and assign them to \( p_1 \). Repeat this step for all
processors \( p_2, \ldots, p_{m-1} \). Finally, schedule all remaining tasks from \( T(h) \) to \( p_m \).

As will be readily seen, in the initial population of [6], a processor \( p_i \) has on
average more tasks than \( p_{i+1} \), \( 1 \leq i < m \). This is because the task distribution over
the processors is not uniform due to the initial population generation scheme.
3.4 Evaluate Fitness

In our case, the fitness of an individual is defined as the difference between its makespan and the one of the individuals having the largest makespan in the population. Our objective is to minimize the makespan and maximize the fitness of an individual. The fitness function in our algorithm is defined as:

\[ f(x) = |\text{Max}_{\text{makespan}} - \text{Self}_{\text{makespan}}| \]

3.5 Selection

We use the “roulette-wheel” method to implement the Selection, this method is also known as Fitness proportionate selection. If \( f_i \) is the fitness of individual \( i \) in the population, its probability of being selected is \( p_i = \frac{f_i}{\sum_{j=1}^{N} f_j} \), where \( N \) is the number of individuals in the population. It is clear that, the candidate solutions with a higher fitness will be more likely to be selected. The Figure bellow can simulate the process of selection.

Form this picture, there are 6 candidates solutions, the area of B (the fitness value of B) is greater than C, and smaller than A. So the value of selected probability is like that: \( P_C < P_B < P_A \).

3.6 Crossover

As mentioned before, there are several ways to go in crossover procedure like one-point crossover, two-point crossover and order crossover. In our case, the
crossover in [6] consists of cutting each string of each of the two parents in two parts - left and right. Firstly, choose a height \( h \) randomly, and separating the tasks.

- Whose height is smaller than \( h \) - left part.
- Whose height is no smaller than \( h \) - right part.

The left parts of each string remain the same, while the right parts of the string are exchanged. To ensure consistency, a partition \( V_1, V_2 \) of the tasks is defined as that the left parts contain only tasks in \( V_1 \) and the right parts contain only tasks in \( V_2 \). Consistency is ensured since there is no dependency from a task in \( V_2 \) to a task in \( V_1 \).

There is a crossover rate in the process of selecting pair of parents. In our case, in order to compare different situations when the crossover rate is fixed or can be tuned dynamically, we first set two different fixed crossover rates: 0.3 and 0.8.

In order to denote the process of crossover, we use the example described in Figure 4 again.

Suppose there are two schedules in \( i_{th} \) generation:

\[
s_1 = \{\{1,4,7,10\},\{2,3,5,6,8,9\}\}
\]
\[
s_2 = \{\{1,2,5,8\},\{3,4,6,7,9,10\}\}
\]

There are only two processors, so each schedule has two strings. Firstly, suppose we choose \( \text{height} = 2 \), for each string of each parent, the part whose \( \text{height} \) is smaller than 2 will remain the same, and exchange the part whose \( \text{height} \) is larger than 2. After crossover according to our method, we can get two new schedules as follows:

\[
s_1^{\text{new}} = \{\{1,4,8\},\{2,3,5,6,7,9,10\}\}
\]
\[
s_2^{\text{new}} = \{\{1,2,5,7,10\},\{3,4,6,8,9\}\}
\]
3.7 Mutation

In order to implement mutation of this algorithm, we first choose a task $t_i$ randomly, then, among all the tasks in the schedule, randomly choose another task $t_j$ whose height is the same as $t_i$ (this can make sure the priority relation). Finally, exchange the positions of task $t_i$ and $t_j$, then generate a new individual.

Follow the example above, suppose we have a schedule in $i_{th}$ generation:

$$ s = \{ \{1,2,4,7\}, \{3,5,6,8,9,10\} \} $$

Firstly, suppose we choose task 2 (of height 0), then, we choose another task 3 whose height is equal to task 2, and then exchange the positions of them. After mutation, we can get a new schedule as follows:

$$ s = \{ \{1,3,4,7\}, \{2,5,6,8,9,10\} \} $$

Just like the crossover, we first set two different fixed mutation rates: 0.3 and 0.02
3.8 Termination condition

Common terminating conditions are:

- A solution is found that satisfies minimum criteria
- Fixed number of generations reached
- successive iterations no longer produce better results
- Implement time is too long

Genetic algorithm can find exact or approximate solutions for most of optimization problems, but not always. Actually, in most of case, we cannot find a good solution because the algorithm itself will cost a lot of time. We must make a choice between costs and results.

In our case, firstly, we will stop the algorithm if the improvement on the best solution of the initial population is very small. Secondly, we set a fixed number of generations to 50; that means after 50 generation, we will stop the implementation of algorithm even if the result is not so good.
4. Genetic algorithm with Fuzzy Logic

4.1 The basic concepts of Fuzzy Logic

What is Fuzzy Logic?

From Wikipedia webpage, I got some basic principles as following:

Fuzzy logic is derived from fuzzy set theory dealing with reasoning that is approximate rather than precisely deduced from classical predicate logic. It can be thought of as the application side of fuzzy set theory dealing with well thought out real world expert values for a complex problem (Klir 1997).

Fuzzy logic allows for set membership values to range (inclusively) between 0 and 1, and in its linguistic form, imprecise concepts like "slightly", "quite" and "very". Specifically, it allows partial membership in a set. It is related to fuzzy sets and possibility theory. It was introduced in 1965 by Lotfi Zadeh at the University of California, Berkeley.

Why use Fuzzy Logic?

Fuzzy Logic offers several unique features that make it a particularly good choice for many control problems.

- It does not require precise, noise-free inputs. The output control is a smooth control function despite a wide range of input variations.
- In the target control system, FL controller uses user-defined rules, it can be modified easily to improve system performance. New rules can easily be incorporated into the system.
- Because of the rule-based operation, any reasonable number of inputs can be
processed (1-8 or more) and numerous outputs (1-4 or more) generated.

- FL can control nonlinear systems that would be difficult or impossible to model mathematically.

Fuzzy logic is widely used in machine control. The term itself inspires a certain skepticism, sounding equivalent to "half-baked logic" or "bogus logic", but the "fuzzy" part does not refer to a lack of rigour in the method, rather to the fact that the logic involved can deal with fuzzy concepts - concepts that cannot be expressed as "true" or "false" but rather as "partially true". Although genetic algorithms can perform just as well as fuzzy logic in many cases, fuzzy logic has the advantage that the solution to the problem can be cast in terms that human operators can understand, so that their experience can be used in the design of the controller. This makes it easier to mechanize tasks that are already successfully performed by humans.

**In our case**, if we want to solve the multiprocessor scheduling problem by using genetic algorithm with fuzzy logic, we should know about what problem we are facing and what should be improved to make the performance better. For this reason, we recall the basal idea of genetic algorithm.

In the initial part, we use the method proposed by [6], this is not important to our research, because there are many methods can do it, we just use one of them. The following, selection, crossover, and mutation, there are some parameters in them. Former experience has taught us that different parameters setting can lead different performance, such as population size, especially crossover probability and mutation probability.

The probability of crossover may be too large sometime, while it may be too small sometime. Here “too large” and “too small” is based on the performance of algorithm. In general, if we can get an optimal or near-optimal solution, then we can say the parameters setting is good, otherwise, we say the parameters setting is bad.

For this consideration, if the parameter is too large, we should decrease it, if the parameter is too small, we should increase it. But here comes a question, how
much should we decrease or increase when it should be tune? How to define “too large” and “too small”? In fact, it is hard to model our problem mathematically, we can’t get the exact definition of “too large” and “too small”.

In the implementation of pure algorithm, the parameters cannot be changed. In this case, after we get the final solution, we can know the parameter setting is good or bad. But it’s too late, it’s very time-consuming. Can we change them at each generation or certain generation? That is for sure. We can define some fuzzy rules according to improve the fitness of each individual. There is not a precise model to state how these parameters can influence the result, but we can tune them at each generation or certain generation according to our defined fuzzy rules. That is why we use fuzzy logic to control the parameters of GA.

**What is membership function?**

The membership function is a graphical representation of the magnitude of participation of each input. It associates a weighting with each of the inputs that are processed, define functional overlap between inputs, and ultimately determines an output response. The rules use the input membership values as weighting factors to determine their influence on the fuzzy output sets of the final output conclusion. Once the functions are inferred, scaled, and combined, they are defuzzified into a crisp output which drives the system.

**What is Defuzzification?**

Defuzzification is the process of producing a quantifiable result in fuzzy logic. Typically, a fuzzy system will have a number of rules that transform a number of variables into a “fuzzy” result, that is, the result is described in terms of membership in fuzzy sets. For example, rules designed to decide how much pressure to apply might result in "Decrease Pressure (15%), Maintain Pressure (34%), Increase Pressure (72%)". Defuzzification would transform this result into a single number indicating the change in pressure. The simplest but least useful defuzzification method is to
choose the set with the highest membership, in this case, "Increase Pressure" since it has a 72% membership, and ignore the others, and convert this 72% to some number. The problem with this approach is that it loses information. The rules that called for decreasing or maintaining pressure might as well have not been there in this case.

A useful defuzzification technique must first add the results of the rules together in some way. The most typical fuzzy set membership function has the graph of a triangle. Now, if this triangle were to be cut in a straight horizontal line somewhere between the top and the bottom, and the top portion were to be removed, the remaining portion forms a trapezoid. The first step of defuzzification typically "chops off" parts of the graphs to form trapezoids (or other shapes if the initial shapes were not triangles). For example, if the output has "Decrease Pressure (15%)", then this triangle will be cut 15% the way up from the bottom. In the most common technique, all of these trapezoids are then superimposed one upon another, forming a single geometric shape. Then, the centroid of this shape, called the fuzzy centroid, is calculated. The x coordinate of the centroid is the defuzzified value.

**What is Fuzzy sets?**

The input variables in a fuzzy control system are in general mapped into by sets of membership functions similar to this, known as "fuzzy sets". The process of converting a crisp input value to a fuzzy value is called "fuzzification".

As an example, let’s consider a system of Microcontroller. Given "mappings" of input variables into membership functions and truth values, the microcontroller then makes decisions for what action to take based on a set of "rules", each of the form:

\[
\text{IF brake temperature IS warm AND speed IS not very fast} \\
\text{THEN brake pressure IS slightly decreased.}
\]

In this example, the two input variables are "brake temperature" and "speed"
that have values defined as fuzzy sets. The output variable, "brake pressure", is also defined by a fuzzy set that can have values like "static", "slightly increased", "slightly decreased", and so on.

In our case, we let probability of crossover and mutation as input variables that have values defined as fuzzy sets. The output variable will be the change of $p_c$ and $p_m$, they are also defined by a fuzzy set. According to the former experience, $p_c(t)$ should be normalized into range [-0.1, 0.1], and $p_m(t)$ should be normalized into range [-0.01, 0.01].

The General Structure of an FLC

An FLC is composed of the following:

- **Knowledge base** that includes the information given by the expert in the form of linguistic control rules.
- **Fuzzification interface**, which has the effect of transforming crisp data into fuzzy sets.
- **Decision making unit**, make the decision according to the knowledge base by using a reasoning method.
- **Defuzzification interface**, which produces a quantifiable result in fuzzy logic.
The general structure of an FLC is shown in Figure 5.

![Figure 5: General structure of an FLC](image)

**How to use Fuzzy Logic?**

1) Define the control objectives and criteria: What am I trying to control? What do I have to do to control the system? What kind of response do I need?

   In this paper, we are trying to control the probability of crossover and mutation, we need design a fuzzy controller based on two fuzzy rules to control them.

2) Determine the relationships between inputs and outputs.

   In our case, we may consider both change of $p_c$ and $p_m$, an AND relationship may be used, and then get the outputs.

3) Using the rule-based structure of Fuzzy Logic, break the control problem down into a series of IF X AND Y THEN Z rules that define the desired system output response for given system input conditions.
4) Create membership functions that define the meaning (values) of Input / Output terms used in the rules.

6) Test the system, evaluate the results, tune the rules and membership functions, if need, retest until satisfactory results are obtained.

In the following, we can see a general example, consider the design of a fuzzy controller for a steam turbine. The block diagram of this control system appears as follows:
The input and output variables map into the following fuzzy set:

Where,

N3: Large negative,
N2: Medium negative,
N1: Small negative,
Z: Zero,
P1: Small positive,
P2: Medium positive,
P3: Large positive.
By using “IF-THEN” rules, we can get the following:

**Rule 1:** IF temperature IS cool AND pressure IS weak, 
THEN throttle is P3.

**Rule 2:** IF temperature IS cool AND pressure IS low, 
THEN throttle is P2.

**Rule 3:** IF temperature IS cool AND pressure IS ok, 
THEN throttle is Z.

**Rule 4:** IF temperature IS cool AND pressure IS strong, 
THEN throttle is N2.

In practice, the controller accepts the inputs and maps them into their membership functions and truth-values. For an example, assume the temperature is in the "cool" state, and the pressure is in the "low" state. The pressure values will use Rule 2. Then we can get the defuzzified value of output, the output will adjust the throttle.

As a summary, **in my opinion**, Fuzzy Logic is a problem-solving control system methodology; it provides a simple way to arrive at a definite conclusion based upon vague, ambiguous, imprecise, noisy, or missing input information. FL's approach to control problems mimics how a person would make decisions. That means it uses an imprecise but very descriptive language to deal with input data more like a human operator.

Fuzzy Logic focuses on what the system should do rather than trying to understand how it works. **In our case**, we don’t need to pay much attention to how the scheduling problem works, we only need to concentrate on controlling the parameters to make them are adaptive. Obviously, in most cases, this will lead quicker and cheaper solutions.
As mentioned previously, before we use Fuzzy Logic in this project, we must clear that what we are trying to control. What do we have to do to control the system? Therefore, we discuss these in detail as follows.

4.2 Genetic Algorithm with fuzzy logic

In this part, we will introduce our improved genetic algorithm with fuzzy logic.

Standard genetic algorithm (SGA) has been known to offer significant approximate solution for a optimization or searching problem. However, despite of the successful application of SGA to these problems, the identification of the correct setting of genetic parameters (probabilities of crossover and mutation, etc.) for the problem is not an easy task. The performance of the genetic algorithm is directly affected by the careful selection of parameters.

In SGA, when we give a fixed probability of parameter at the beginning of algorithm, we can’t change it in the process of implementation until it stops. In this situation, it is possible to destroy a high fitness individual when a large crossover probability is set. For a low crossover probability, sometimes it is hard to obtain better individuals and gets a slow convergence. High mutation probability will bring too much diversity and takes a longer time to get the optimal solution. Low mutation probability can’t prevent the premature convergence and miss some near-optimal solutions.

How to set the parameters correctly? Which value is the most adaptive? Can we change the parameters in the process of implementation? If change, which value of the variation is suitable? This is the motivation to introduce Fuzzy Logic to control the parameters.

In our project, the probabilities of crossover and mutation are just the objectives that we will control. So that we design two fuzzy logic controllers, one is for probability of crossover (\( p_c \)), the other one is for probability of mutation (\( p_m \)). A
diagram of FLC in our genetic algorithm is shown in Figure 6.

![Diagram of FLC in Genetic Algorithm](image)

**Figure 6: Fuzzy Logic Control Genetic Algorithm**

The dashed area above is just the FLC. The main idea is to use the current performance measures of GA as the inputs of FLC, and the outputs of the FLC are the new GA's parameters. Current performance measures measured by the fitness are sent to the FLC, and then, FLC computes the new control parameters through the certain fuzzy rules. When GA gets the new parameters from FLC, it will repeat the steps above until the termination condition is met. So the value of controller parameters may be renewed in every generation in our algorithm (FLGA).

In this paper, we design the fuzzy rules by using two different strategies. The general idea for updating the crossover and mutation probabilities is to consider the changes of the maximum fitness and average fitness in the GA population of two continuous generations.
4.3 The first strategy for fuzzy rules:

\[ p_c \] would be decreased and \[ p_m \] increased when \( f_{\text{ave}}(t) \) is near to \( f_{\text{max}}(t) \) or \( f_{\text{ave}}(t-1) \) is near to \( f_{\text{ave}}(t) \). This principle is based on the very simple idea that it encourages the good individuals produce more offspring, reduces the chance for poorly performing operators to destroy the potential individuals during the recombination process, faster the speed of convergence, and prevent the premature convergence.

1. When \( f_{\text{ave}}(t) \) is near to \( f_{\text{max}}(t) \), we decrease \( p_c \).

2. If there is a large or considerable difference between \( f_{\text{ave}}(t) \) and \( f_{\text{max}}(t) \), we increase \( p_c \).

3. When \( f_{\text{ave}}(t-1) \) is near to \( f_{\text{ave}}(t) \), that means the changes of fitness in two continuous generation are very small. Therefore, we should increase \( p_m \) to jump out the suboptimal, and faster the convergence.

4. If there is a large or considerable difference between \( f_{\text{ave}}(t-1) \) and \( f_{\text{ave}}(t) \), even \( f_{\text{ave}}(t-1) \) is larger than \( f_{\text{ave}}(t) \), that means the changes of fitness in two continuous generation are very large, and may produce a worse performance through generations. Therefore, we should properly decrease \( p_m \) to avoid some good individuals be destroyed (through mutation).

Here we introduce two parameters \( R_1 \) and \( R_2 \) to denote the rules above.

\[
R_1(t) = f_{\text{max}}(t) - f_{\text{ave}}(t), \quad R_1(t) \geq 0 \quad (2)
\]

\[
R_2(t) = f_{\text{ave}}(t) - f_{\text{ave}}(t-1)
\]
Where,

\( t \) is iteration,

\( f_{\text{max}} (t) \) is the best fitness at iteration \( t \),

\( f_{\text{ave}} (t) \) is the average fitness at iteration \( t \),

\( f_{\text{ave}} (t-1) \) is the average fitness at iteration \( (t-1) \).

So, we can make \( R_1 (t) \) and \( R_2 (t) \) as the inputs of FLC, and its outputs are the changes in crossover and mutation probabilities, \( \Delta P_c (t) \) and \( \Delta P_m (t) \). The membership functions of fuzzy inputs and outputs are shown in the following figures. \( R_1 (t) \) and \( R_2 (t) \) are normalized into the range \([-1.0, 1.0]\), \( \Delta P_c (t) \) is normalized into range \([-0.1, 0.1]\), and \( \Delta P_m (t) \) is normalized into range \([-0.01, 0.01]\).

Figure 7 & Figure 8 show the Membership Function for \( R_1 \) and \( R_2 \)

![Membership Function for R1](image-url)
Figure 8: Membership Function for $R_2(t)$

Where,

- $ZE^1 = \text{Zero}$,
- $NL^1 = -0.8$,
- $NR^1 = -0.6$,
- $NM^1 = -0.4$,
- $NS^1 = -0.2$,
- $PL^1 = 0.8$,
- $PR^1 = 0.6$,
- $PM^1 = 0.4$,
- $PS^1 = 0.2$.

The following two figures show the Membership Functions for $p_c$ and $p_m$.

Figure 9: Membership Function for $p_c$
Where,

\[ ZE^2 = \text{Zero}, \]
\[ NL^2 = -0.08, \quad PL^2 = 0.08, \]
\[ NR^2 = -0.06, \quad PR^2 = 0.06, \]
\[ NM^2 = -0.04, \quad PM^2 = 0.04, \]
\[ NS^2 = -0.02, \quad PS^2 = 0.02. \]

Figure 10: Membership Function for \( P_m \)

Where,

\[ ZE^3 = \text{Zero}, \]
\[ NL^3 = -0.008, \quad PL^3 = 0.008, \]
\[ NR^3 = -0.006, \quad PR^3 = 0.006, \]
\[ NM^3 = -0.004, \quad PM^3 = 0.004, \]
\[ NS^3 = -0.002, \quad PS^3 = 0.002. \]
According to the membership functions above, then we can get several decision rules as follows:

**Rule 1:** IF \( R_1 = ZE^1 \) AND \( R_2 = NL^1 \),

\[ \text{THEN } \Delta p_c(t) = PL^2, \Delta p_m(t) = PL^3 \]

**Rule 2:** IF \( R_1 = ZE^1 \) AND \( R_2 = NR^1 \),

\[ \text{THEN } \Delta p_c(t) = PR^2, \Delta p_m(t) = PR^3 \]

**Rule 3:** IF \( R_1 = PL^1 \) AND \( R_2 = NL^1 \),

\[ \text{THEN } \Delta p_c(t) = NS^2, \Delta p_m(t) = ZE^3 \]

**Rule 4:** IF \( R_1 = PL^1 \) AND \( R_2 = NR^1 \),

\[ \text{THEN } \Delta p_c(t) = NM^2, \Delta p_m(t) = NS^3 \]

Other Rules...

As mentioned before, all rules are based on the idea that it encourages the good individuals produce more offspring, reduces the chance for poorly performing operators to destroy the potential individuals during the recombination process. \( R_1 \) is the difference between \( f_{\text{max}}(t) \) and \( f_{\text{ave}}(t) \), \( R_2 \) is the difference between \( f_{\text{ave}}(t) \) and \( f_{\text{ave}}(t-1) \), both of them can state the performance of GA. We just use our experience before to make the decision rules.
For example,

- When $R_1$ remains the same, $\Delta p_c(t)$ will decrease if $R_2$ is changed in positive direction.

\[(R_1 \text{ remains the same}, \ R_2 \text{ is changed in positive direction.})\]

- When $R_2$ remains the same, $\Delta p_m(t)$ will decrease if $R_1$ is changed in positive direction.

\[\Delta p_m(t)\]

According to the above, we can easily get the other rules. Then we use them to get the outputs from FLC, and change the parameters $p_c$ and $p_m$:

\[p_c(t) = p_c(t-1) + \Delta p_c(t) \quad (4)\]

\[p_m(t) = p_m(t-1) + \Delta p_m(t) \quad (5)\]
Then we renew the procedure of our genetic algorithm as follows:

**Fuzzy Logic Control Genetic Algorithm**

Begin FLGA

- $t = 0$ Iteration counter
- Initialize population $P(t)$
- Compute the fitness values
- While (Termination condition is not met) do
  - $t = t + 1$
  - Select $P(t)$ from $P(t-1)$
  - Crossover $P(t)$
  - Mutate $P(t)$
  - Evaluate $P(t)$
  - Renew GA parameters
    - Call fuzzy logic controller
    - Update the parameters according equations (4) and (5)
- End while
End FLGA

The first strategy for fuzzy rules has some advantages and disadvantages. It considers both average and max fitness in each generation, so the parameters can be tune immediately. But if we analyze it carefully, the performance of algorithm can be influenced by both crossover and mutation. We do not know the performance becomes better is because of changes of $p_c$ or changes of $p_m$, we also do not know the performance becoming worse is affected by which one.
For an example, in the first strategy, at $i_{th}$ generation, suppose $f_{ave}(t)$ is near to $f_{max}(t)$, and $f_{ave}(t-1)$ is near to $f_{ave}(t)$. According to our fuzzy rules, we should decrease $p_c$ and increase $p_m$, and a better average fitness of next generation is assumed. **Here comes one question**, we get a better fitness of next generation is because of decreasing $p_c$ or increasing $p_m$?

- If it is only because of decreasing $p_c$, that means we should not increase $p_m$. Hence, the rule for updating the crossover probability should be questioned.

- If it is only because of increasing $p_m$, that means we should not decrease $p_c$. Hence, the rule for updating the mutation probability should be questioned.

For these considerations, we propose another strategy for fuzzy rules.

### 4.4 The second strategy for fuzzy rules:

As mentioned above, when $f_{ave}(t)$ is near to $f_{max}(t)$, it is hard to say we should decrease or increase $p_c$. In a similar reason, when $f_{ave}(t-1)$ is near to $f_{ave}(t)$, it is hard to say we should decrease or increase $p_m$. However, notice that, one principle is doubtless: the occurrence probability is based on the fitness of generation. Be differ from the first strategy, we would tune $p_c$ or $p_m$ separately instead of tuning both $p_c$ and $p_m$ at each generation. That means we would only tune $p_c$ at two continuous generations and only tune $p_m$ at next two continuous generations.
At the first two continuous generations, \( p_m \) remains the same, \( p_c \) is increased if the change in average fitness is positive otherwise it is decreased.

\[
p_m \text{ remains the same, IF } f_{ave}(t) \geq f_{ave}(t-1), \text{ THEN increase } p_c.
\]

\[
IF f_{ave}(t) < f_{ave}(t-1), \text{ THEN decrease } p_c.
\]

At the second two continuous generations, \( p_c \) remains the same, \( p_m \) is increased if the change in average fitness is positive otherwise it is decreased.

\[
p_c \text{ remains the same, IF } f_{ave}(t) \geq f_{ave}(t-1), \text{ THEN increase } p_m.
\]

\[
IF f_{ave}(t) < f_{ave}(t-1), \text{ THEN decrease } p_m.
\]

In the second strategy, we do not tune both \( p_c \) and \( p_m \) immediately at each generation, but change them alternately.
5. Experiment results, comparisons and discussion

In this section, we will analyze the results of the following algorithms:

- The pure genetic algorithm with a fixed crossover probability 0.3 and mutation probability 0.3.
- The pure genetic algorithm with a fixed crossover probability 0.8 and mutation probability 0.02.
- Fuzzy Logic genetic algorithm (both the first and second strategy for fuzzy rules) with an initial crossover probability 0.3 and mutation probability 0.3.

What is Benchmark?

In computing, a benchmark is the act of running a computer program, a set of programs, or other operations, in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it. The term, benchmark, is also mostly utilized for the purposes of elaborately designed benchmarking programs themselves. Benchmarking is usually associated with assessing performance characteristics of computer hardware, for example, the floating-point operation performance of a CPU, but there are circumstances when the technique is also applicable to software. Software benchmarks are, for example, run against compilers or database management systems.

Benchmarks provide a method of comparing the performance of various subsystems across different chip/system architectures. Benchmarking is helpful in understanding how the database manager responds under varying conditions. we can create scenarios that test deadlock handling, utility performance, different methods of loading data, transaction rate characteristics as more users are added, and even the effect on the application of using a new release of the product.

The Benchmarks in this project are based on a multiprocessor scheduling problem with DAG (directed acyclic graph). After a series of theoretical analysis, we will test the scheduling problem with benchmarks. As mentioned before, the
scheduling problem itself is not so important to us in this project, the point of this paper is how to use fuzzy logic controller to dynamically control the parameters of genetic algorithm. For the simple reason, we set the initial population size to 30, the multiprocessor system has 16 processors.

5.1 Benchmark

The benchmark instances we used in the experiment are provided by a tool called ANDES – Synth [11, 12]. It contains several well-known parallel programs as follows:

1. **Bellford**: it solves the shortest path problem from all nodes to a single destination in a weighted directed graph. The graph represents the algorithm called Bellman-Ford [13].
2. **Diamond1**: known as a space-time digraph representing a systolic computation [14].
3. **Diamond2**: this digraph is known as a systolic matrix multiplication.
4. **Diamond3**: the systolic computation of the transitive closure of a relation on a set of elements is this digraph [15].
5. **Diamond4**: it’s the systolic computation of the transitive closure of a relation on a set of elements [16].
6. **Divconq**: this stands for divide and conquer algorithm. The digraph has a tree shape.
7. **FFT**: that’s Unidimensional fast Fourier transform.
8. **Gauss**: the task digraph describes the execution of a Gaussian elimination used in the resolution of linear systems.
9. **Iterative**: it is a generic iterative algorithm with iteration being represented in the same level of the digraph. Concerning the next iteration, the immediate successors of a task Ti at Kth position are tasked at (k+1)th position.
10. **MS-Gauss**: represents a computation containing successive resolutions of linear systems by Gaussian elimination.
11. **Prolog**: the structure of this digraph is obtained at random and corresponds to
the resolution of a logic program.

12. **Qcd**: thus gradient method for linear systems [13].

Each parallel program has two kind of size: large (-l) and medium (-m). **Table 3** shows the different number of tasks.

**Table 3**: Characteristics of test graphs

<table>
<thead>
<tr>
<th>Parallel Program</th>
<th>Number of Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bellford</td>
<td>-m 365</td>
</tr>
<tr>
<td></td>
<td>-l 992</td>
</tr>
<tr>
<td>Diamond1</td>
<td>-m 258</td>
</tr>
<tr>
<td></td>
<td>-l 1026</td>
</tr>
<tr>
<td>Diamond2</td>
<td>-m 486</td>
</tr>
<tr>
<td></td>
<td>-l 1227</td>
</tr>
<tr>
<td>Diamond3</td>
<td>-m 731</td>
</tr>
<tr>
<td></td>
<td>-l 1002</td>
</tr>
<tr>
<td>Diamond4</td>
<td>-m 731</td>
</tr>
<tr>
<td></td>
<td>-l 1002</td>
</tr>
<tr>
<td>Divconq</td>
<td>-m 382</td>
</tr>
<tr>
<td></td>
<td>-l 766</td>
</tr>
<tr>
<td>FFT</td>
<td>-m 194</td>
</tr>
<tr>
<td></td>
<td>-l 1026</td>
</tr>
<tr>
<td>Gauss</td>
<td>-m 782</td>
</tr>
<tr>
<td></td>
<td>-l 1227</td>
</tr>
<tr>
<td>Iterative</td>
<td>-m 262</td>
</tr>
<tr>
<td></td>
<td>-l 938</td>
</tr>
<tr>
<td>MS-Gauss</td>
<td>-m 768</td>
</tr>
<tr>
<td></td>
<td>-l 1482</td>
</tr>
<tr>
<td>Prolog</td>
<td>-m 214</td>
</tr>
<tr>
<td></td>
<td>-l 1313</td>
</tr>
<tr>
<td>Qcd</td>
<td>-m 326</td>
</tr>
<tr>
<td></td>
<td>-l 1026</td>
</tr>
</tbody>
</table>
Notice that, different programs have different number of tasks, that means the best and average makespans of solutions are diverse. If we want to prove the general good effect of our algorithm, we must apply our algorithm to many enough situations.

All of the genetic algorithms stop when the improvement on the best solution of the initial population is very small. The main idea is that we should let the algorithm run as long as it is improving the solution in a reasonable way. But in our case, we set a fixed number of generation to 50. That means, if we can get a good result before the 50th generation, then stop it. But we also will stop the implementation of algorithm after 50 generation even if the result is not so good. The reason is the running time may be very high when the generation is greater than 50, we may cost a couple of hours or more than one day. We always make a choice between cost and running time when using genetic algorithm.

In order to compare the quality of the solutions provided by SGA and FLGA, we add a traceable procedure in our algorithms so that we can record the best and average solutions of each generation.

In this project, we mainly focus on the effect of fuzzy logic controller, compare the results which from the standard genetic algorithm without FLC with those from fuzzy logic genetic algorithm. Actually, many factors can influence the results of GA. Such as coding method, population size, different restrictions, different rules of crossover and mutation, etc. Our algorithm may get a better result if we change the these factors, but that is not the main point of this project, the aim of our research is to design an algorithm which can dynamically control the parameters of GA, so that the genetic algorithm can be self-controlled, and get a better performance.

Our job is to research and prove it.

We collected all of the best and average solutions obtained by SGA and FLGA, and then make it as Table 4. All of the makespans are given in seconds.
Table 4: Comparison of the best and average solutions obtained by SGA and FLGA

<table>
<thead>
<tr>
<th>Graph (DAG)</th>
<th>SGA ((p_c = 0.3, p_m = 0.3))</th>
<th>SGA ((p_c = 0.8, p_m = 0.02))</th>
<th>FLGA-1 (The First Strategy)</th>
<th>FLGA-2 (The Second Strategy)</th>
<th>Best Known</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Size</td>
<td>Best Schedule</td>
<td>Average Schedule</td>
<td>Best Schedule</td>
<td>Average Schedule</td>
</tr>
<tr>
<td>Bellford</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-m</td>
<td>327</td>
<td>341</td>
<td>392</td>
<td>112</td>
<td>137</td>
</tr>
<tr>
<td>-l</td>
<td>1206</td>
<td>1002</td>
<td>1065</td>
<td>413</td>
<td>465</td>
</tr>
<tr>
<td>Diamond1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-m</td>
<td>286</td>
<td>326</td>
<td>397</td>
<td>268</td>
<td>286</td>
</tr>
<tr>
<td>-l</td>
<td>1437</td>
<td>1323</td>
<td>1401</td>
<td>761</td>
<td>801</td>
</tr>
<tr>
<td>Diamond2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-m</td>
<td>520</td>
<td>536</td>
<td>604</td>
<td>253</td>
<td>260</td>
</tr>
<tr>
<td>-l</td>
<td>1508</td>
<td>1206</td>
<td>1264</td>
<td>892</td>
<td>915</td>
</tr>
<tr>
<td>Diamond3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-m</td>
<td>937</td>
<td>890</td>
<td>920</td>
<td>520</td>
<td>586</td>
</tr>
<tr>
<td>-l</td>
<td>1153</td>
<td>957</td>
<td>1103</td>
<td>664</td>
<td>677</td>
</tr>
<tr>
<td>Diamond4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-m</td>
<td>436</td>
<td>392</td>
<td>466</td>
<td>226</td>
<td>231</td>
</tr>
<tr>
<td>-l</td>
<td>735</td>
<td>742</td>
<td>968</td>
<td>335</td>
<td>372</td>
</tr>
<tr>
<td>Divconq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-m</td>
<td>326</td>
<td>354</td>
<td>397</td>
<td>217</td>
<td>220</td>
</tr>
<tr>
<td>-l</td>
<td>902</td>
<td>866</td>
<td>925</td>
<td>442</td>
<td>495</td>
</tr>
<tr>
<td>FFT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-m</td>
<td>137</td>
<td>115</td>
<td>134</td>
<td>94</td>
<td>109</td>
</tr>
<tr>
<td>-l</td>
<td>562</td>
<td>435</td>
<td>496</td>
<td>216</td>
<td>265</td>
</tr>
<tr>
<td>Gauss</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-m</td>
<td>726</td>
<td>771</td>
<td>886</td>
<td>386</td>
<td>412</td>
</tr>
<tr>
<td>-l</td>
<td>2012</td>
<td>1322</td>
<td>1435</td>
<td>715</td>
<td>886</td>
</tr>
<tr>
<td>Iterative</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-m</td>
<td>92</td>
<td>116</td>
<td>131</td>
<td>39</td>
<td>71</td>
</tr>
<tr>
<td>-l</td>
<td>253</td>
<td>286</td>
<td>299</td>
<td>168</td>
<td>224</td>
</tr>
<tr>
<td>MS-Gauss</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-m</td>
<td>14071</td>
<td>14965</td>
<td>15007</td>
<td>5113</td>
<td>5694</td>
</tr>
<tr>
<td>-l</td>
<td>15816</td>
<td>17963</td>
<td>13226</td>
<td>3029</td>
<td>3730</td>
</tr>
<tr>
<td>Prolog</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-m</td>
<td>204</td>
<td>265</td>
<td>272</td>
<td>86</td>
<td>143</td>
</tr>
<tr>
<td>-l</td>
<td>1524</td>
<td>1433</td>
<td>1505</td>
<td>311</td>
<td>396</td>
</tr>
<tr>
<td>Qcd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-m</td>
<td>4591</td>
<td>5022</td>
<td>5706</td>
<td>1426</td>
<td>1503</td>
</tr>
<tr>
<td>-l</td>
<td>16018</td>
<td>18026</td>
<td>17216</td>
<td>4102</td>
<td>4215</td>
</tr>
</tbody>
</table>
From this table, we can clearly see that the results of FLGA are much better than those of SGA. There are two SGA, one is with a fixed $p_c = 0.3$ and $p_m = 0.3$, and the other one is with a fixed $p_c = 0.8$ and $p_m = 0.02$. The algorithms of the two SGA are the absolute same, and they have a common ground that the probabilities of crossover and mutation will be a constant. However, in our proposed algorithm, the probabilities of crossover and mutation can be changed at each or certain generation according to a certain fuzzy rule.

The results are obvious, both FLGA-1 and FLGA-2 have the much better solutions, which means we can get a much better schedule of multiprocessor scheduling problem. The results obtained by two SGA look like near-same, but the results of the first SGA are worse than the second SGA at most situations. Because the $p_m$ of the first SGA is equal to 0.3, actually it is a high probability of mutation, although it can jump out from the local-optimal, however, it may break some good individuals of each generation, so it may destroy some near-optimal solutions. Notice that, the bold number indicate the results will become very bad if $p_m$ is too high. The results may change radically in the process of convergence.

By comparing the FLGA-1 and FLGA-2, we can see that FLGA-2 has a better performance than FLGA-1 in most cases. This indicates that the second strategy is better than the first strategy in our project. It is because of the principle of fuzzy rules. Although we cannot tune the parameters at each generation in FLGA-2, but we has a more effective method based on considering the change of fitness in two continuous generations. In FLGA-1, we can tune the parameters immediately at each generation, but it is hard to judge whether the change of $p_c$ and $p_m$ is correct or appropriate so that we may make an incorrect action. In the first strategy, the change of $p_c$ may lead the average fitness of next generation better, while the change of $p_m$ may lead the average fitness of next generation worse. However, the accumulative performance may be a little better all the same. Obviously, only according to the accumulative performance, we cannot make sure whether we should increase or decrease the parameters.
We also compared the experimental results with the best well-known results. We can found that the solutions obtained by FLGA is closer to the best well-known solutions than SGA. However, both of them cannot be very close to the best well-known results, this is because of the drawback of the coding method used in the pure algorithm [6]. The pure algorithm may miss some optimal solution. In spite of the fuzzy logic can make good efforts, it can’t get the optimal solution. As mentioned before, the coding method and the problem itself are not the main point of this paper, so the improving of performance is acceptable. Moreover, the most important thing is that the fuzzy controller makes effort, which indicates the genetic algorithms can perform better with fuzzy logic inside it.

Both of our FLGA is with a initial $p_c=0.3$ and $p_m=0.3$, this is the same as the first SGA, here comes an interesting question: In FLGA, if we set a different initial $p_c$ and $p_m$, what’s the results will be? Will the different initial parameters produce different results? Can the $p_c$ and $p_m$ will be actually self-controlled? The fuzzy rules we defined can really make effect? Let’s prove it.

Table 5 Shows the comparison of the final solutions obtained by FLGA-1 with different initial $p_c$ and $p_m$. 


Table 5: Comparison of the final solutions obtained by FLGA-1 (the first strategy) with different initial $p_c$ and $p_m$

<table>
<thead>
<tr>
<th>Graph(DAG)</th>
<th>FLGA-1 ($p_c = 0.3$, $p_m = 0.08$)</th>
<th>FLGA-1 ($p_c = 0.8$, $p_m = 0.02$)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Name</td>
<td>Size</td>
</tr>
<tr>
<td>Bellford</td>
<td>-m</td>
<td>112</td>
</tr>
<tr>
<td></td>
<td>-l</td>
<td>413</td>
</tr>
<tr>
<td>Diamond1</td>
<td>-m</td>
<td>268</td>
</tr>
<tr>
<td></td>
<td>-l</td>
<td>761</td>
</tr>
<tr>
<td>Diamond2</td>
<td>-m</td>
<td>253</td>
</tr>
<tr>
<td></td>
<td>-l</td>
<td>892</td>
</tr>
<tr>
<td>Diamond3</td>
<td>-m</td>
<td>520</td>
</tr>
<tr>
<td></td>
<td>-l</td>
<td>664</td>
</tr>
<tr>
<td>Diamond4</td>
<td>-m</td>
<td>226</td>
</tr>
<tr>
<td></td>
<td>-l</td>
<td>335</td>
</tr>
<tr>
<td>Divconq</td>
<td>-m</td>
<td>217</td>
</tr>
<tr>
<td></td>
<td>-l</td>
<td>442</td>
</tr>
<tr>
<td>FFT</td>
<td>-m</td>
<td>94</td>
</tr>
<tr>
<td></td>
<td>-l</td>
<td>216</td>
</tr>
<tr>
<td>Gauss</td>
<td>-m</td>
<td>386</td>
</tr>
<tr>
<td></td>
<td>-l</td>
<td>715</td>
</tr>
<tr>
<td>Iterative</td>
<td>-m</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>-l</td>
<td>168</td>
</tr>
<tr>
<td>MS-Gauss</td>
<td>-m</td>
<td>5113</td>
</tr>
<tr>
<td></td>
<td>-l</td>
<td>3029</td>
</tr>
<tr>
<td>Prolog</td>
<td>-m</td>
<td>86</td>
</tr>
<tr>
<td></td>
<td>-l</td>
<td>311</td>
</tr>
<tr>
<td>Qcd</td>
<td>-m</td>
<td>1426</td>
</tr>
<tr>
<td></td>
<td>-l</td>
<td>4102</td>
</tr>
</tbody>
</table>
From the table above we notice that, in spite of the initial $p_c$ and $p_m$ are different, the results are very close. That means we need not pay much attention to the setting of initial parameters of FLGA, the fuzzy rules we defined can help to adjust the parameters dynamically.

We also draw the performance curves to trace the best solutions of SGA and FLGA-1 from generation to generation. Program *Diamond4 -m* is the test instance.

![Figure 11: The performance curves of SGA and FLGA-1](image)

As we can see from the Figure 11, SGA converge quickly, but they also have a larger probability to get trapped in local optimum, while FLGA-1 can spend less
generation to arrive the global optimum, then get the much better results. Moreover, the curve of the first SGA changes radically and may go up sometimes, this is because the \( p_m \) of the first SGA (\( p_c = 0.3, p_m = 0.3 \)) is very large, it may destroy some near-optimal solutions, and abruptly change them to bad solutions.

From the experimental results, it seems that we can get the following:

- For a high crossover probability, it can converge quickly, but easy to get trapped in local optimum.
- For a low crossover probability, it is hard to make some good individuals to reproduce new generation, and takes longer time to converge.
- For a high mutation probability, it introduces too much diversity and takes longer time to get the optimal solution.
- For a low mutation probability, it may miss some near-optimal solution.

One more interesting thing: Can we trace the change of parameters in FLGA? Yes we can, after each generation, we record the current \( p_c \) and \( p_m \), and finally plot them in the curves. Program **Diamond4 -m** is the test instance.

![Figure 12: the curves denote the change of crossover probability in FLGA](image-url)
From this picture, we can see that the crossover probability of SGA is constant, and the crossover probability of FLGA-1 changes at each generation, while crossover probability of FGLA-2 changes at each two generation. Because the second strategy tune $p_c$ and $p_m$ alternately, so there would be a same value between two continuous generations.

In the same way, we can get the curves denote the change of mutation probability in FLGA.

![Figure 13: the curves denotes the change of mutation probability in FLGA](image)

Like the previous picture, **Figures 13** shows that the mutation probability of SGA is constant; and the mutation probability of FLGA-1 changes at each generation, while mutation probability of FGLA-2 changes at each two generation. The reason is the similar to above.
6. Conclusion & Further research

In this paper, we introduced a fuzzy logic genetic algorithm based on an existing hybrid genetic algorithm working on the multiprocessor scheduling problem. In fact, the fuzzy rules we defined can be easily used in the other problems that will use genetic algorithm. We focus on the fuzzy logic controller but not the problem itself. The working point is to design appropriate fuzzy rules according to different problems. The multiprocessor scheduling problem is just as the starting point.

The standard genetic algorithm is a successful method to solve some optimization and search problems. Normally, these problems are NP-hard [3]. However, the pure genetic algorithm also has some disadvantages, such as premature-convergence, local optimum, etc. Different settings of probabilities of parameters can make different performances.

Before the implementation of genetic algorithm, it is impossible to set the parameters absolute right. We just can do it based on our experience, but cannot model it mathematically. Hence, we get help from fuzzy logic controller to adjust them dynamically.

The experimental results indicate that the fuzzy logic controller helps to improve the performance of a pure genetic algorithm. Although it will cost extra time to adjust the parameters in each generation, it converges to the global optimum in a smaller number of generations.

In this paper, we propose two different strategies for fuzzy rules, and the experimental results can prove both of them have a good effect. We also compare the performance between them. Different strategies for fuzzy rules can lead different results, but the general idea is changeless which is based on the consideration of fitness, the main attempt is to improve the fitness through tuning the parameters from generation to generation.
But here comes a question: how to design the exact rules? There are many principles to make it, how to choice? Actually, in most of case, we only can do it according to our experiences and theoretical analysis, and then prove it by the experimental results. However, it is not always a good method, and a lack of efficient. Therefore, in the further research, we should try to develop a systematic and intelligently automatic approach to design such control systems.

Can we use genetic algorithm itself to optimize rule bases from traditional FLCs? If we can, it will be a totally self-control system. That is very interesting.
References

(All web pages referenced are available on 19th May, 2008)


[4]: Multiprocessor Scheduling:
http://en.wikipedia.org/wiki/Multiprocessor_scheduling


evaluation of parallel programs”, in Environments and Tools for Parallel Scientific
Computing, J. J. Dongarra and B. Tourancheau, Eds., Amsterdam, The Netherlands,

[12]: J. Kitajima and B. Plateau, “Modeling parallel program behavior in
1994.

[13]: D. Bertsekas and J. Tsitsiklis, Parallel and Distributed Computation:

[14]: O. Ibarra and S. Sohn, “On mapping systolic algorithms onto the

part ii – design”, IEEE Transactions on Parallel and Distributed Systems,vol .1, no. 4,

[16]: C. Scheiman and P. Cappello, “A processor-time-minimal systolic array
for transitive closure”, IEEE Transactions on Parallel and Distributed Systems, vol. 3,

[17]: Gilles Brassard, Paul Bratley: “Fundamentals of Algorithmics”, Prentice-

[18]: Geogre F Luger, “Artificial intelligence: structures and strategiess for