3

Guide for Development with PISALib

This chapter shortly describes the two example problems- LOTZ on variator side and FEMO on selector side- which were implemented to demonstrate the use of PISALib. Then an overview on how to use the functions that are provided is given. It should serves as guide on how to implement a program with the help of PISALib. First the variator part is looked at, followed by the selector part. Finally a runtime analysis for some functions is given to give an idea of how efficient the implemented functions are.

3.1 LOTZ- Leading Ones, Trailing Zeros

As an example implementation on the variator side an easy two objective, discrete problem was chosen. The objective in a bit string of fixed length is to optimize leading ones as a first goal and trailing zeros as a second goal. The problem's solution is obvious. If we have a bit string of length five for example, the optimal solutions are: 00000, 10000, 11000, 11100, 11110 and 11111. They are all equally optimal (also called Pareto optimal). Since we know the problem's solution, we can test easily later if the variator and selector give us a good solution. We chose a few standard variation techniques to demonstrate the use of the local parameter file. Two different recombination methods- one point crossover and uniform crossover were chosen- and two different mutation methods- one bit mutation and independent bit mutation- were chosen.
In this section we will give a brief overview of the way the example works. Part 3.3 gives an overview of how implementations are made in general.
The variation, which is the main action executed in every loop, includes the following main steps:

1. Copying the newly added individuals
2. Doing a recombination of the copies

3. Doing a mutation of the offspring

Step 1 we do, so the following steps have a copy to work on and change. In step 2 we first decide if we should to a recombination at all with a certain probability and then decide which recombination type to make depending on the parameters read from the local parameter file. Both recombination types take pairs of the copies and mix them. One point crossover cuts them at a random position and switches two parts of the individuals. Example:

Copy 1: 11010 10111
Copy 2: 01100 11011
After recombination:
Offspring 1: 11010 11011
Offspring 2: 01100 10111

Uniform crossover considers each bit of both individuals to be swapped. If a random number is one for example they are swapped and if itŠs zero they arenŠt swapped. Example:

Copy 1: 1101010111
Copy 2: 0110011010
Swap mask: 1010100111
After recombination:
Offspring 1: 0111010010
Offspring 2: 1100011111

Step 3 involves deciding whether to make a mutation of the offspring at all, again with a certain probability and then choosing between independent bit mutation and one bit mutation. Independent bit mutation flips every bit with a certain probability, while one bit mutation flips only one bit.
Once these variation steps have been made, the new individuals have to be evaluated. PISA specifies that the objective values should always be minimized. So the first objective value (leading ones) is calculated by counting the leading ones and then subtracting that value from the bit sting length. Trailing zeros is calculated similarly, by counting the zeros at the end and then subtracting the count from the bit string length. Once these main steps are done the newly created offspring is ready to enter a selection process on the selector side...

3.1.1 Usage

LOTZ can be called with the following arguments:
variator paramfile filenamebase poll
paramfile: specifies the name of the file containing the local parameters (e.g. var_param.txt)
filenamebase: specifies the name (and optionally the directory) of the communication files. The filenames of the communication files and the configuration file are built by appending 'sta', 'var', 'sel','ini', 'arc' and 'cfg' to the filenamebase. (Example for the filenamebase: sample.) Caution: the filenamebase must be consistent with the name of the configuration file and the filenamebase specified for the selector

module.
poll: gives the value for the polling time in seconds (e.g. 0.5).
It can also just be called with no arguments in which case the default arguments are used:
paramfile: var_param.txt
filenamebase: sample.
poll: 1

3.2 FEMO

On the selector side FEMO (Fair Evolutionary Multi-objective Algorithm) was chosen as example. The algorithm keeps an archive of all non-dominated individuals. Each individual has a counter which is increased when the individual is chosen as parent. The individual with the lowest counter is always chosen as parent. If there are several individuals with an equally low counter one amongst them is chosen randomly. A new individual is just added to the global population if it is not dominated by any other individual and if it is not equal in all objective values to any individual. We will look at the practical implementation of FEMO now. From read_var or read_ini we get an array of identities of new individuals which was already added to the global population. We first test if any of those new individuals dominates any of the remaining new ones or any old ones, so any individuals which are in the global population at the time. If yes, we remove them. We test for the remaining new individuals if they are dominated or equal in all objective values to any other individual in the global population and if yes, we remove them.
To select individuals, we fill an array with identities of individuals with the lowest counter. Then we choose one of these identities and increase the selection counter for the corresponding individual. This process we repeat mu times. So it is possible and likely to choose the same individual several times. An individual dominated by another individual if all its objective values are worse than that of the other individual.

3.2.1 Usage

FEMO can be called with the following arguments:
selector paramfile filenamebase poll
paramfile: specifies the name of the file containing the local parameters (e.g. sel_param.txt)
filenamebase: specifies the name (and optionally the directory) of the communication files. The filenames of the communication files and the configuration file are built by appending 'sta', 'var', 'sel','ini', 'arc' and 'cfg' to the filenamebase. (Example for the filenamebase: sample.) Caution: the filenamebase must be consistent with the name of the configuration file and the filenamebase specified for the selector module.
poll: gives the value for the polling time in seconds (e.g. 0.5).
It can also just be called with no arguments in which case the default arguments are used:
paramfile: sel_param.txt

filenamebase: sample.
poll: 1

3.3 Implementation Guide

The implementation guide describes the basic steps that are needed to code an actual implementation with the help of the provided PISALib implementation in C. Some explanations are illustrated with help of the C code examples LOTZ and FEMO.

3.3.1 Variator

As mentioned before, all a user needs to look at to make a working implementation is the variator.h file to see what functions are at his disposal, the variator_user.h file to see which functions he has to implement and the variator_user.c file where he actually needs to write the code. In Appendix A an overview is given what is supplied and what needs to be implemented.
Dividing the implementation into four parts, following steps are required in general:

1. Design the individual structure or class and implement the functions for it
2. Implement additional functions to determine how to read local parameters at initialization and when the algorithms should terminate
3. Implement some variation function which produces new individuals out of the ones meant for variation and evaluates them
4. Define what happens in each state

In step one it has to be decided what information should be included in an individual. The individual should on the one hand normally represent a candidate solution to the practical problem and on the other hand have some sort of fitness. Example from LOTZ:

```
struct individual_t {
  int *bit_string; /* the genes */
  int length;      /* length of the bit_string */
  double lo;       /* objective value (length - leading ones) */
  double tz;       /* objective value (length - trailing zeros) */
};
```

The only restrictions that are imposed are that the `get_objective` function should return an objective value for the individual and a way to free the memory consumed by the individual should be implemented in `free_individual`. The `free_individual` function has to free all the memory allocated for an individual so no memory leaks occur. Example from LOTZ:

```
int free_individual(individual *ind)
  if(ind == NULL) return(1);
```

```
  free(ind->bit_string);
  free(ind);
  return(0);
}
double get_objective_value(int identity, int index)
{
}                                                          10
  double objective_value=-1.0;
  individual *temp_ind;
  if(index < 0 || index > (dimension - 1)) return(-1);

  temp_ind = get_individual(identity);
  if(temp_ind == NULL) return(-1);

  if(index == 0) objective_value = temp_ind->lo;
  else objective_value = temp_ind->tz;
  return(objective_value);                                 20
}
```

In step two `read_local_parameter` should be implemented. It is a convenient mean to define global variables that user does not want to hardcode into the program. The parameter file should be read and the variables stored in such a way, that they are accessible when needed later. As example one could imagine a parameter which determines what recombination type should be used. The easiest way to keep the parameter throughout the program is to us global variables to store them (see LOTZ source B.1.2 on page 35). The `is_finished` function is used in each cycle of the main program to see if the variator should terminate. If a termination condition is met it should return 1. As an example one can use a global counter which is initialized in state 0 and return 1 in `is_finished` if the counter reaches a certain value. Example from LOTZ:

```
int is_finished()
{
    return (gen >= maxgen);
}
```

In step three it is not really mandatory to write a special variation function. The user can do the variation of the individuals within the different states if he wants to. To simplify matters it is assumed that the user put all his variation steps in one function which does all the required. Convenient is for example if this function takes as argument the array of identities that should be variated and returns the identities of the newly generated individuals. Then it can be interposed between the `read_sel` function and the `write_var` function in state 2. Additional steps which should also be included are to add the newly created individuals to the global population with the `add_individual` function. Don't forget to evaluate the new or changed individuals, so their fitness values are up to date. An example on how to implement the variation function can be seen in the LOTZ source B.1.2 on page 40.

The variation function is called `variate` and the evaluation function are `eval_lo` and `eval_tz`.
In step four the state functions have to be completed correctly to guarantee a smooth run of the program (you can find an example of the implemented state functions in LOTZ source B.1.2 on page 37):
The `state0` function should take care of reading and setting the local parameters and initialization of the population. The alpha newly created individuals should be stored in an array so it can be passed to the `write_ini` function, which is finally called to communicate the new population to the selector side.
`state2` function is the main loop function that is called in every regular cycle between variator and selector. The following steps should be executed in the given sequence. The `read_sel` and the `read_arc` function should be called and the output of `read_sel` stored in an array of size mu. If either one of the two function fails, the state function should return 2, which signalizes that the reading operations failed and on the next polling interval a new try is started in the main loop automatically. If both of the functions succeed the next step would be to variate the individuals gotten from `read_sel`. Assuming a function variate was written in step three beforehand, we pass it the array and it should create lambda individuals with the old ones. After doing additional steps the final step in the state is to pass this array of identities to the `write_var` function to communicate the new ones to the selector side.
The `state4` function is responsible for the termination step. That can involve writing some sort of result output to a file or screen for example.
Normally no action needs to be taken in the `state7` function. In the example implementation (in variator.c) if state 7 is reached the variator terminates as well (goes to state 4). Of course the user can change this in variator.c if this is not the behavior he wishes.
In the `state8` function a reset of all the variables should be done. The cleaning of the global population is handled automatically in the state machine.
Normally nothing needs to be done in the `state11` function. In the example implementation the variator cleans the global population and sets state 0 after execution of the `state11` function.
These were the steps required to get a working variator. In the next section we will look at which steps are needed on the selector side.

3.4 Selector

Implementing the selector involves very similar steps as on the variator side. So a lot of descriptions will be alike. The main differences to the variator will be *emphasized*.
As mentioned before, all a user needs to look at to make a working implementation is the selector.h file to see what functions are at his disposal, the selector_user.h file to see which functions he has to implement and the selector_user.c file where he actually needs to write the code. In an overview is given what is supplied and what needs to be implemented.
Dividing the implementation into four parts, following steps are required in general:

1. Design the individual structure or class and implement the functions for it

2. Implement additional functions to determine how to read local parameters at initialization and when the algorithms should terminate
3. *Implement a selection function* which determines what individuals are selected for the offspring production and which removes the individuals that aren't needed anymore from the global population
4. Define what happens in each state

In step one it has to be decided what information should be included in an individual. On the selector side not a whole lot has to be changed normally. The regular case is that the individual has fitness values. An example for additional members can be seen in the example from FEMO:

```
struct individual_t {
  double *objective_value;
  int counter;
};
```

Functions that need to be implemented by the user are `set_objective_value`*,* `create_individual` and like on the variator side `free_individual`. `set_objective_value` and `create_individual` are needed by the `read_ini` and `read_var` functions when they add new individuals to the global population. `free_individual` is used in the `remove_individual` function which removes an individual from the global population. `create_individual` should allocate memory for an individual and initialize its members meaningful, `free_individual` should free the memory consumed by the individual so no memory leaks occur and `set_objective_value` should set an objective value to the given argument. All three functions are pretty straight forward and you can see best how they are used in the FEMO example:

```
int set_objective_value(individual *ind, int index, double obj_value)
{
  if(ind == NULL || index < 0 || index >= dimension)
    return(1);

  ind->objective_value[index] = obj_value;
  return(0);
}

individual *create_individual()
{

  individual *return_ind;
  double *obj_value;

  return_ind = malloc(sizeof(individual));
  if (return_ind == NULL) {
    log_to_file(log_file, __FILE__, __LINE__, "selector out of memory");
```

```
    return(NULL);
  }

  obj_value = malloc(sizeof(double) * dimension);
  if (obj_value == NULL) {
    log_to_file(log_file, __FILE__, __LINE__, "selector out of memory");
    return(NULL);
  }

  return_ind->objective_value = obj_value;
  return_ind->counter = 0;
  return(return_ind);
}

int free_individual(individual* ind)
{
  free(ind->objective_value);
  free(ind);
  return(0);
}
```

In step two `read_local_parameter` should be implemented. It is a convenient mean to define global variables that user does not want to hardcode into the program. The parameter file should be read and the variables stored in such a way, that they are accessible when needed later. As example one could imagine a parameter which determines what seed the random generator gets . The easiest way to keep the parameter throughout the program is to us global variables to store them (see FEMO source B.2.2 on page 50).
The `is_finished` function is used in each cycle of the main program to see if the selector should terminate. If a termination condition is met it should return 1. As an example one can use a global counter which is initialized in state 0 and return 1 in `is_finished` if the counter reaches a certain value. If it is assumed that the variator terminates, nothing needs to be done in `is_finished`.
In step three it is not really mandatory to write a special selection function. The user can do the selection of the individuals within the different states if he wants to. To simplify matters it is assumed that the user put all his selection and population cleaning steps in one function which does all the required, like in the example. Normally the user will want to select some individuals, and remove some from the population which are worse for example. For this matter the `remove_individual` function should be used. You can see a practical example in the FEMO code B.2.2 on page 56. The selection function in FEMO is called `select_ind`.
In step four the state functions have to be completed correctly to guarantee a smooth run of the program (you can find an example of the implemented state functions in LOTZ source B.2.2 on page 52):
The `state1` function should take care of reading and setting the local parameters.

Then it should read the initial population with `read_ini`. The returned array contains alpha new individuals which have been automatically added to the global population. Now the selection function should be used. The mu selected individuals are then passed to the `write_sel` function to be written to the sample file, the `write_arc` function should finally be called to write the remaining individuals in the global population to the archive file.
`state3` function is the main loop function that is called in every regular cycle between variator and selector. The following steps should be executed in the given sequence. The `read_var` function should be called and the output of `read_var` stored in an array of size lambda. If the function fails, the state function should return 2, which signalizes that the reading operations failed and on the next polling interval a new try is started in the main loop automatically. If the function succeeds the next step would be to select mu individuals and remove the ones not needed from the global population. In the FEMO example you can see how these steps are done in select. After doing additional steps the final steps in the state are to pass this array of identities to the `write_sel` function to communicate the new ones to the selector side and to call the `write_arc` function to write the contents of the global population to the archive file.
Normally no action needs to be taken in `state5`. In the example implementation (in selector.c) if state 5 is reached the selector terminates as well (goes to state 6). Of course the user can change this in variator.c if this is not the behavior he wishes.
The `state6` function is responsible for the termination step. That can involve writing some sort of result output to a file or screen for example.
Normally nothing needs to be done in `state9`. In the example implementation the selector sets state 11 after execution of the `state9` function.
In `state10` a reset of all the variables should be done. The cleaning of the global population is handled automatically in the state machine.

3.5 Running Time Analysis of some Functions

Since PISALib provides functions to manage the global pool of individuals; they should have optimal access time and not add additional overhead to every operation. This would lessen the value of PISALib as it would turn optimization problems slow. Some of the critical functions that will be used very often by the user to access individuals are analyzed here and most of them come close to optimal execution time.

3.5.1 Variator

First a quick overview of the global population class: Pointers to individual are kept in an array. The index of the array corresponds to the identity of an individual. If there is no individual at the index, there should be a NULL pointer. The structure also keeps record of the population size and the largest identity (last_identity) used at the moment.
Now to the running time of the different functions:

get_individual It runs in O(1) since all it has to do is access the array at the given index and return the pointer. So it runs in optimal time.

get_next (get_first) This is one of the most difficult functions to analyze with regards to running time since the access time depends on the filling level of the array. Worst case running time is O(last_identity), if there is just one individual at the first position and one at the last position. In general it should have a lot faster access time, since the freed ids are always reused from the free ids stack, thus preventing higher ids from being generated before the lower ones are used.

remove_individual It runs in O(1), since it just has to access the array by the index, set the pointer to NULL, diminish the global population size by one and eventually also the last_identity.

add_individual Worst case= O(current population size), if there is no more space left in global population and the array size needs to be doubled. Normal running time= O(1) to get a free id (pop operation off stack) + O(1) insertion time (access to array)= O(1) This is nearly optimal. Optimality is traded for having the advantage of a flexible population size.

read_sel Reading the selection file and writing it to an array is straight forward and runs in O(mu), which is optimal.

read_arc Read arc reads the individual from the arc file and updates the population. For that it reads the individuals into an array (to_keep), sorts this array with quicksort and then goes through the global population's individuals throwing out the ones not found in to_keep. Calling arc file size: n, O(n) is needed to read the file and fill the array, O(n * log(n)) for the quicksort algorithm and O(global population size) for removal of the individuals. So depending which is bigger it is either O(n *log(n)) or O(global population size). But it is not $O(n^2)$ which would result if comparing each of them without sorting.

write_var, write_ini The functions write the objective values of the individuals in the given array to file. This takes O((lambda or alpha) * dimension) since the get_objective function takes O(1). This is optimal.

3.5.2 Selector

Since the selector is very similar and symmetrical to the Variator only the functions with main differences are commented.

add_individual Here adding individual works slightly differently. Since the identity is given as an argument the individual can be directly inserted into the array, which saves the stack operation. But the array size also needs to be doubled it the identity is too large. The execution time is thus nearly the same as in the Variator part, namely O(1) approximated at average O(max array size) in the worst case.

read_var (read_ini) The functions add the individuals to the global population, which takes O(lambda or alpha) for reading into an array and O(lambda or alpha) for adding to the global population, which is optimal.

write_sel Just writes an array of integers to a file which takes O(mu) and is optimal.

write_arc This function writes all the identities remaining in the global population to file. The involved steps are, going through the population (with get_next) and writing the identity to file. The required execution time is in the order of O(global population size).