A GENETIC ALGORITHM C CODE FOR SCHEDULING PROBLEMS

Bryan A. Norman
and
James C. Bean
Dept of Industrial and Operations Engineering
The University of Michigan
1205 Beal Avenue
Ann Arbor, Michigan 48109-2117

Technical Report 94-22

September 1994
Revised October 1994

A Genetic Algorithm Code for Scheduling Problems: Serial Computing Version *

Bryan A. Norman
James C. Bean

Department of Industrial and Operations Engineering
University of Michigan, Ann Arbor, MI 48109-2117

Version 1.2

October 17, 1994

*This work was supported in part by the National Science Foundation under Grant DDM-9018515 and DDM-9308432 to the University of Michigan.

Contents

1 Introduction

This report is a documentation and a user manual for the C code, **genjssp**, developed for finding solutions to job shop scheduling problems (JSSP). The code can be applied to JSSPs with the following complexity:

1. Non-zero ready times.
2. Due dates.
3. Job shop or open shop structure.
4. Multiple, non-identical machines.
5. Routing flexibility for jobs.
6. Sequence dependent setup times.
7. Tooling constraints. Different jobs may compete for the same tool and/or the changeover from one tool to another may induce sequence dependent setup times.
8. The objective function is a combination of regular measures.

The code presented here does not consider precedence constraints among the jobs. However, it can be modified in the manner described in Norman[1994] to handle precedence constraints. It is also assumed that setups only occur when there is a tooling change. The code could be readily modified to accommodate other types of setups by making changes to the **chromosome_evaluation** function. The objective function considered in the program is the basic tardiness measure. Weighted tardiness and more complicated objective function measures can be considered by simply modifying the **chromosome_evaluation** function. It is assumed that the problem has a minimization objective. If the objective is maximization the following changes should be made: change the **chromosome_evaluation** function to reflect the correct objective function, sort each generation's objective function values in descending order rather than ascending order,

modify the code sections that collect information about the best solution found and the first solution found within 5% of the lower bound.

The code implements a genetic algorithm (GA) to search the problem space. The GA utilizes the random keys encoding described in Bean[1994] and a coarse-grained parallel population structure. For this particular implementation, alleles are represented by integers where the 3 least significant digits contain job number information and the fourth least significant digit contains machine assignment information. Computational tests show that **genjssp_par** finds good solutions, within 5% of provable lower bounds, to 300 job problems containing the previously listed complexities. See Norman and Bean[1994] and Norman[1994] for more information.

2 User Manual

2.1 Code Location

The source code and other related files are placed on the anonymous FTP machine called *freebie.engin.umich.edu* under the directory

/pub/misc/ga_sched

The files have been compressed and stored in one file, named **genjssp.zip**, using the Unix zip utility. To retrieve these files. the user should use the ftp program on a machine with tcp/ip connectivity to the Internet. One should connect to *freebie* with the ftp command, and log in with account name *anonymous* and his/her electronic mail address as the password.

In the following ftp session, the file **genjssp.zip** is retrieved.

```
dexter% ftp freebie.engin.umich.edu
Connected to knob2.engin.umich.edu.
220 knob2.engin.umich.edu FTP server (Version 5.60) ready.
Name (freebie.engin.umich.edu:user_id): anonymous
```

```
331 Guest login ok, send ident as password.
Password:  <<< Your Email Address Here >>>
230 Guest login ok, access restrictions apply.
ftp> cd pub/misc/ga_sched
250 CWD command successful.
ftp> get genjssp.zip
200 PORT command successful.
150 Opening ASCII mode data connection for genjssp.zip (29708 bytes).
226 Transfer complete.
local: genjssp.zip remote: genjssp.zip
127515 bytes sent in 1 seconds (1.2e+02 Kbytes/s)
ftp> quit
221 Goodbye.
dexter%
```

Once the file **genjssp.zip** is retreived the file should be unzipped. This will place several files in the current directory including: example problem data and the source code, example parameter file, sample output, and documentation for both the serial and parallel versions of the genetic algorithm. There is also a file titled **FILE.INFO** that contains a brief description of each of the files.

2.2 Compiling and Running

The code is compiled under the Ultrix Operating System (on DEC 5000) using the cc compiler. To produce an executable program using the optimizer option (O), type

cc genjssp.c -O -lm -o genjssp

The executable **genjssp** is invoked by the command

genjssp

There are two input files for the program genjssp. The first input file, named **genjssp_data**, contains the job data. The format for this file is as follows. The first line contains the values of the total number of jobs, N, the total number of machines, M, and the total number of tools, T. The next N lines contain specific data for each job. For each job $i = 1, \ldots N$, this data includes: job number, i, tool requirement, t_i, ready time, r_i, due time, d_i, processing time, p_i, number of machines that can process the job, nm_i, and the list of machine numbers that can process the job, $m_{i,j}$ for $j = 1, \ldots nm_i$. The remaining T lines contain the setup times, $s_{k,l}$, for switching from tool type k to tool type l for $k, l = 1, \ldots T$. The input file format is given on the next page.

$$
\begin{array}{cccccccc}
N & M & T & & & & & \\
1 & t_1 & r_1 & d_1 & p_1 & nm_1 & m_{1,1}, \ldots m_{1,nm_1} \\
2 & t_2 & r_2 & d_2 & p_2 & nm_2 & m_{2,1}, \ldots m_{2,nm_2} \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
N & t_N & r_N & d_N & p_N & nm_N & m_{N,1}, \ldots m_{N,nm_N} \\
s_{1,1} & s_{1,2} & \cdots & s_{1,T} & & & \\
s_{2,1} & s_{2,2} & \cdots & s_{2,T} & & & \\
\vdots & \vdots & \vdots & \vdots & & & \\
s_{T,1} & s_{T,2} & \cdots & s_{T,T} & & &
\end{array}
$$

The second input file, **genjssp_parameters**, contains problem specific parameter settings for the GA. These parameters are discussed in more detail in Section 2.3. The order of the parameters in the file **genjssp_parameters** should match the order found in the discussion in Section 2.3.

2.3 Setting Parameters

The maximum problem and population sizes are given. They are used for dimensioning arrays.

Max_jobs	The maximum number of jobs.
Max_machines	The maximum number of machines.
Max_pop	The maximum population size.
Max_tools	The maximum number of tools.

They are specified at the beginning of the code with **#define** statements (see section 3.1). If these limits are violated, **genjssp** terminates with a descriptive error message. In this case, the user should increase the limit(s) and recompile the code.

The following are more problem-specific parameters, which are read from the input file **genjssp_parameters**. A brief description of each parameter is provided, for more detailed information see Norman[1994].

total_pop_size	The total population size. Empirically 1 to 4 times the number of jobs works well. For small problems use a minimum size of 100. Larger population sizes produce better results but require longer running times to converge to a solution.
max_generations	The maximum number of generations that the GA will run. Longer runs produce better solutions but require longer running times.
clone_frac	The elitest strategy is implemented by copying the **number_clones** best solution from each subpopulation into the next generation. Subpopulation size multiplied by **clone_frac** yields the value for **number_clones**. Empirically a value of .04 - .05 for **clone_frac** works well.
crossover_prob	The crossover probability used in the Bernoulli crossover operator. A value of 0.7 typically performs well.

immig_frac The fraction of each subpopulation that are immigrants. Empirically values in the range .02 - .03 work well.

immig_type_rate The fraction of the immigrants that are not strongly biased. Empirically a value of 0.6 works well.

ready_weight The biasing weight assigned to ready time. Based on empirical results, the weights for the ready time and the due time should be approximately equal. Empirically a value of 1.0 works well.

due_weight The biasing weight assigned to due time.

strong_bias_frac The fraction of each subpopulation that will be strongly biased initially. Empirically a value of .7 works well.

num_subpops The number of subpopulations. Empirically 4 works well. If the **total_pop_size** is very large (>1024) this value could be increased. There is a trade-off between the benefit of having more subpopulations and the liability of the subpopulations becoming too small to search effectively.

commun_frac The fraction of each subpopulation to communicate between subpopulations. Empirically values in the range .03 - .04 work well. In general, the value should be less than the value of **clone_frac** and enough less that **number_clones - number_to_commun** > 0.

gen_to_commun The number of generations between occurrences of communication between subpopulations. Empirically a value of 20 works well.

startseed The initial random number seed.

endseed The ending random number seed. The GA will run endseed-startseed times.

lb_value The value of a lower bound (assuming the objective is minimization) for the problem.

2.4 An Example

Consider the following example with 10 jobs, 2 machines, and 3 tools. The job data is provided below.

$$N = 10 \quad M = 2 \quad T = 3$$

i	t_i	r_i	d_i	p_i	nm_i	$m_{i,1}$	$m_{i,2}$
1	1	4.68	2.00	1.12	2	1	2
2	1	1.25	2.50	0.61	2	1	2
3	2	1.50	2.28	1.91	2	1	2
4	2	1.75	4.12	0.43	2	1	2
5	2	1.99	5.13	0.77	2	1	2
6	3	2.24	3.89	1.05	2	1	2
7	3	2.49	4.27	0.49	2	1	2
8	3	2.74	6.74	2.31	2	1	2
9	2	3.12	5.54	1.10	2	1	2
10	1	2.78	4.11	0.41	2	1	2

$$s_{1,1} = 0.00 \quad s_{1,2} = 0.68 \quad s_{1,3} = 1.42$$
$$s_{2,1} = 0.75 \quad s_{2,2} = 0.00 \quad s_{2,3} = 0.99$$
$$s_{3,1} = 1.81 \quad s_{3,2} = 1.12 \quad s_{3,3} = 0.00$$

The corresponding input file for the job data, here called **genjssp_data**, is set up for this problem as follows:

```
10   2   3
 1   1   4.68   2.00   1.12   2   1   2
 2   1   1.25   2.50   0.61   2   1   2
 3   2   1.50   2.28   1.91   2   1   2
 4   2   1.75   4.12   0.43   2   1   2
 5   2   1.99   5.13   0.77   2   1   2
 6   3   2.24   3.89   1.05   2   1   2
 7   3   2.49   4.27   0.49   2   1   2
```

```
  8   3   2.74   6.74  2.31  2  1  2
  9   2   3.12   5.54  1.10  2  1  2
 10   1   2.78   4.11  0.41  2  1  2
0.00 0.68 1.42
0.75 0.00 0.99
1.81 1.12 0.00
```

Note that the format of the data file may be such that each data element is in a separate line as long the elements are in the right order. An example input file for the GA parameters, **genjssp_parameters**, is as follows:

```
100 250 .06 .70 .04 .6 1.0 0.9 .7 2 .04 20 1 11 11.36
```

Here is a sample session and the corresponding output. For this example the variable **print_freq** was set equal to 5 and only the results for the first random seed are shown. For this random seed, the optimum value of 11.37 was found.

Example Session.

```
dexter% genjssp

genjssp

The number of jobs is 10.
The total number of machines is 2
The total number of tools is 3.

-----------------GA-PARAMETER VALUES----------------------

total_pop_size =        100
max_generations =       250
clone_frac =          0.060
crossover_prob =       0.70
immig_frac =          0.040
immig_type_rate =      0.60
ready_weight =         1.00
due_weight =           0.90
strong_bias_frac =     0.70
num_subpops =             2
```

```
commun_frac =          0.04
gen_to_commun =          20
startseed =               1
endseed =                11
lb_value =            11.36

------------------GENERATION RESULTS----------------------

Generation Number  Best Solution Found
         5              15.82
        10              14.36
        15              14.36
        20              14.36
        25              14.36
        30              13.65
        35              11.84
        40              11.43
        45              11.43
        50              11.37
        55              11.37
        60              11.37

--------------------SUMMARY DATA--------------------------


This seed's total time = 3.10 seconds.

The best solution found is 11.37

The best sol. of 11.37 found in gen 49 required 2.42 seconds.

Lb+5 percent solution 11.84 found in gen 32 required 1.62 seconds.

--------------------FINAL SOLUTON-------------------------
```

Job Number	Machine Assignment	Completion Time	Tardiness For This Job	Total Tardiness
2	2	1.86	0.00	0.00
7	2	3.77	0.00	0.00
3	1	3.41	1.13	1.13
4	1	3.84	0.00	1.13
6	2	4.82	0.93	2.06

```
    5          1          4.61          0.00          2.06
    9          1          5.71          0.17          2.23
    8          2          7.13          0.39          2.62
   10          1          6.87          2.76          5.38
    1          1          7.99          5.99         11.37
The objective value is =  11.37
```

3 Code Documentation

3.1 Basic Definitions and Data Structure

```
#include <stdio.h>
#include <math.h>
#include <sys/time.h>
#include <sys/types.h>

#define Max_jobs 500            /* Max number of jobs.     */
#define Max_machines 20         /* Max number of machines. */
#define Max_pop 1025            /* Max population size.    */
#define Max_tools 50            /* Maximum number of tools.*/

#define maxint 2147483647       /* Max integer value.      */

/* Timing variables. */
float time_passed;
struct tms before,after;

/* General GA variables. */
int total_pop_size;             /* Total population size.  */
int max_generations;            /* Maximum number of gens  */
int number_clones;              /* No. clones in a subpop. */
float clone_frac;               /* Clone frac. for each subpop.   */
int number_immig;               /* No. immigrants in a subpop.    */
float immig_frac;               /* Immig. frac. for each subpop.  */
float crossover_prob;           /* Crossover probability.  */
int number_jobs;                /* Total number of jobs.   */
int number_tools;               /* Total number of tools.  */
int total_number_machines;      /* Total number of machs.  */
int startseed,endseed;          /* Random seed values.     */
int min_target = 0;             /* Minimum target value.   */
float lb_value;                 /* Lower bound value.      */
float immig_type_rate;          /* Prob. for a type of immigrant.*/
int rank1[Max_pop+2];           /* Rank within each subpop.*/
int rank2[Max_pop+2];           /* Rank within total pop.  */
double data_to_rank[Max_pop+1]; /* Array which is ranked.  */

/* Subpopulation variables. */
int num_subpops;                /* Number of subpops.         */
int subpop_size;                /* Size of each subpop.       */
```

```
int subpop_number[Max_pop+1];   /* Subpop no. for each chrom. */
int num_to_commun;              /* No. of chrom. to commun.   */
float commun_frac;              /* Frac. of subpop to commun. */
int gen_to_commun;              /* No. of gens. until commun. */

/* Variables for the chromosome_evaluation() and                */
/* print_final_solution() functions.                            */
int   temp_alleles[Max_jobs+1];    /* Temp. array to sort RKs.    */
float machendtime[Max_machines+1]; /* When machine next available.*/
float toolavail[Max_tools];        /* When tool next available.   */
int   toollastmach[Max_tools];     /* Where a tool last used.     */
float setupvalue[Max_tools+1][Max_tools+1]; /* Setup times.       */

/* Biasing variables */
float ready_weight;             /* Ready time bias weight.        */
float due_weight;               /* Due time bias weight.          */
float raw_bias=0;               /* Raw bias.                      */
double adj_bias=0;              /* Adjusted bias.                 */
float true_bias[Max_jobs+1];    /* Array of adjusted biases.      */
int strong_bias;                /* No. per subpop strongly biased.*/
float strong_bias_frac;         /* Frac. of each subpop strongly biased.  */

/* Other variables. */
int job[Max_jobs+2];            /* Indices for machine array.     */
int machine[2000];              /* Alternative machines for a job.*/
int best_solution_gen=0;        /* Gen. that best solution found. */
int lb_solution_gen=0;          /* Gen. that Lb+5% solution found.*/
float best_solution;            /* Best solution found.           */
float lb_solution;              /* Lb+5% solution found.          */
float lblatetem;                /* Temp. var. to find lb_solution.*/
float total_time;               /* Total program time.            */
float total_time_to_lb;         /* Time until Lb+5% sol. found.   */
float total_time_to_best;       /* Time until best solution found.*/

/* Opens the data files. */
FILE *fpl;

/* Defining the structures for the data file. */
struct job_data
{
int index;          /* Job number.                 */
int tool;           /* Tool needed.                */
```

```
int num_machines;  /* No. machs that can make job. */
float ready;       /* Job ready time.              */
float due;         /* Job due time.                */
float proctime;    /* Job processing time.         */
};

struct job_data data[Max_jobs];

/* Defining the structures for the chromosomes. */
struct chromosome
{
int alleles[Max_jobs+1];     /* Array of allele values.    */
double fitness;              /* Fitness of chromosome.     */
double pseudo_fitness;       /* Fitness for within subpop. */
};

struct chromosome newchrom[Max_pop+2],oldchrom[Max_pop+2];
```

A given generation consists of **total_pop_size** solutions (or individuals) split into **num_subpops** subpopulations. Each individual is defined as a structure whose members are the solution itself, its objective function value, and its pseudo objective function value (used to rank within the different subpopulations.)

3.2 Input Function

The function **readdata** reads the problem data and GA parameters.

```
void readdata

/*********************************************************/
/* The function readdata() reads in the problem data     */
/* from the file genjssp_data and then creates two       */
/* vectors that relate machine compatibility with each   */
/* job. Then the GA parameters are read in.              */
/*********************************************************/
{
int i=1,j,k=0;

/* Open the job data file.                 */
fpl= fopen("genjssp_data", "r");

/* If the data file does not exist, print a message.     */
if (fpl == NULL)
   {
   printf("Unable to open genjssp_data \n");
   return;
   }

/* Read in data file size variables and check that they   */
/* do not exceed the defined limits.                      */
fscanf(fpl,"%d %d %d",&number_jobs,&total_number_machines,&number_tools);
printf("genjssp \n\n");
printf("The number of jobs is %d.\n",number_jobs);
printf("The total number of machines is %d.\n",total_number_machines);
printf("The total number of tools is %d.\n\n",number_tools);

if(number_jobs>Max_jobs)
   {
   printf("Error: Job Limit Exceeded.\n");
   printf("Data = %d and Limit = %d.\n",number_jobs,Max_jobs);
   exit(1);
   }
if(total_number_machines>Max_machines)
   {
   printf("Error: Machine Limit Exceeded.\n");
   printf("Data = %d and Limit = %d.\n",total_number_machines,Max_machines);
```

```
   exit(1);
   }
if(number_tools>Max_tools)
   {
   printf("Error: Tool Limit Exceeded.\n");
   printf("Data = %d and Limit = %d.\n",number_tools,Max_tools);
   exit(1);
   }

/* The job data information is read in as a structure. */
/* As the data is read in, two single-column arrays,   */
/* job[] and machine[] that relate machine             */
/* compatibility with jobs are created.                */
job[0]=0;
while (i<=number_jobs)
   {
   fscanf(fpl,"%d%d%f%f%f%d",&data[i].index,&data[i].tool,
      &data[i].ready,&data[i].due,&data[i].proctime,&data[i].num_machines);
   for(j=1;j<=data[i].num_machines;j++)
      {
      fscanf(fpl,"%d",&machine[k]);
      k++;
      }
   job[i]=k;
   i++;
   } /* end of while statement */
for(j=1;j<=number_tools;j++)
   for(k=1;k<=number_tools;k++)
      fscanf(fpl,"%f",&setupvalue[j][k]);
fclose(fpl);

/* Open the GA parameters file.            */
fpl= fopen("genjssp_parameters", "r");

/* If the data file does not exist, print a message.    */
if (fpl == NULL)
   {
   printf("Unable to open genjssp_parameters \n");
   return;
   }

/* Read in the GA parameter data.         */
```

```
fscanf(fpl,"%d%d%f%f%f%f%f%f%f%d%f%d%d%d%f",&total_pop_size,
   &max_generations,&clone_frac,&crossover_prob,&immig_frac,
   &immig_type_rate,&ready_weight,&due_weight,&strong_bias_frac,&num_s ubpops,
   &commun_frac,&gen_to_commun,&startseed, &endseed,&lb_value);

printf("-----------------GA-PARAMETER VALUES---------------------\n\n");
printf("total_pop_size = %10d \n",total_pop_size);
printf("max_generations = %9d \n",max_generations);
printf("clone_frac = %14.3f \n",clone_frac);
printf("crossover_prob = %10.2f \n",crossover_prob);
printf("immig_frac = %14.3f \n",immig_frac);
printf("immig_type_rate =  %8.2f \n",immig_type_rate);
printf("ready_weight = %12.2f \n",ready_weight);
printf("due_weight = %14.2f \n",due_weight);
printf("strong_bias_frac = %8.2f \n",strong_bias_frac);
printf("num_subpops = %13d \n",num_subpops);
printf("commun_frac = %13.2f \n",commun_frac);
printf("gen_to_commun = %11d \n",gen_to_commun);
printf("startseed = %15d \n",startseed);
printf("endseed = %17d \n",endseed);
printf("lb_value = %16.2f \n\n",lb_value);

/* Check that the population size does not exceed its limit. */
if(total_pop_size>Max_pop)
   {
   printf("Error: Population Size Limit Exceeded.\n");
   printf("Data = %d and Limit = %d.\n",
      total_pop_size,Max_pop);
   exit(1);
   }

/* Convert percentages to integer values. The .01 is to
   catch round-off errors.  */
subpop_size=total_pop_size/num_subpops;
number_clones= clone_frac*subpop_size+.01;
number_immig = immig_frac*subpop_size+.01;
strong_bias = strong_bias_frac*subpop_size+.01;
num_to_commun= commun_frac*subpop_size+.01;
}
```

3.3 Genetic Algorithm Functions

There is one main GA function called **genetic** and four other functions referred to as **initialize_population**, **reproduction**, **communication** and **chromosome_evaluation**. **Genetic** first calls **initialize_population** which initializes some parameters and randomly generates an initial population of solutions. Then, it calls iteratively **reproduction** which, given an initial generation, reproduces a new one using the process of elitist reproduction, Bernoulli crossover, and the immigration operator described in Norman and Bean[1994]. **Communication** is called every **gen_to_communicate** generations to share chromosomes between the different subpopulations. The function **chromosome_evaluation** evaluates a solution given its index in the current population, which is defined by the array of structures **newchrom**. The function **chromosome_evaluation** is called by **initialize_population** and **reproduction**.

```
void chromosome_evaluation(m)

/*********************************************************/
/* The function chromosome_evaluation(m) evaluates       */
/* the fitness of a chromosome.                          */
/*********************************************************/
int m;
{
int job_to_sched;
int oldtooltype[Max_machines+1],newtype[Max_machines+1];
int i,mach,temp;
float setuptime;

/* Initialize.                                           */
newchrom[m].fitness=0;
for(i=0;i<=total_number_machines;i++) machendtime[i] = 0;
for(i=0;i<=total_number_machines;i++) oldtooltype[i] = 0;
for(i=1;i<=number_tools;i++)
   {
   toolavail[i]=0;
   toollastmach[i]=0;
   }

/* Sort the random key values for the jobs.              */
```

```
for (i=1;i<= number_jobs;i++)
   temp_alleles[i] = newchrom[m].alleles[i]+i;
heapsort(number_jobs);

/* Construct a semi-active schedule based on the sorted */
/* order of the random keys.                            */
for (i=1;i<=number_jobs;i++)
   {
/* Determine the job number and machine assignment.              */
/* Note that the job number information                         */
/* is contained in the 3 least significant digits of the allele */
/* (this needs to be increased if the job number exceeds 1000)  */
/* and the machine assignment is contained in the fourth least  */
/* significant digit of the allele (this could be extended if the */
/* machine number exceeds 10).                                  */
   temp=temp_alleles[i]%10000;
   mach=(temp)/1000;
   job_to_sched = temp-(mach)*1000;

/* Calculate the appropriate setup time.      */
   if(data[job_to_sched].tool==oldtooltype[mach] &&
       toollastmach[data[job_to_sched].tool]==mach)
      setuptime=0;
   else
      setuptime=setupvalue[oldtooltype[mach]][data[job_to_sched].tool];

/* Calculate the starting time and subsequent    */
/* completion time for each job.         */
   if(machendtime[mach]<toolavail[data[job_to_sched].tool])
      machendtime[mach]=toolavail[data[job_to_sched].tool];
   machendtime[mach]+=setuptime;
   if (machendtime[mach]<data[job_to_sched].ready)
       machendtime[mach]=data[job_to_sched].ready;
   machendtime[mach] = toolavail[data[job_to_sched].tool] =
       machendtime[mach] + data[job_to_sched].proctime;

/* Update the tool indicators.         */
   toollastmach[data[job_to_sched].tool]=mach;
        oldtooltype[mach]=data[job_to_sched].tool;

/* Determine the tardiness.           */
   if (machendtime[mach] - data[job_to_sched].due>0)
```

```
      newchrom[m].fitness+= machendtime[mach]-data[job_to_sched].due;
   }
}
```

The function **initialize_population** determines subpopulation data, calculates the biasing values for each job, and creates an initial generation of chromosomes.

```
void initialize_population()

/*********************************************************/
/* The function initialize_population() initializes     */
/* the chromosomes, evaluates their fitness and         */
/* performs the necessary ranking.                      */
/* It also determines the bias values from              */
/* their ready and due weights.                         */
/*********************************************************/
{
int i,j;

/* Indicate which subpop a given processor is in.    */

subpop_size=total_pop_size/num_subpops;
for(i=0;i<total_pop_size;i++)
   subpop_number[i]=i/subpop_size;

/* Calculate the biasing values for each job.      */

for (j=1;j<=number_jobs;j++)
   {
   raw_bias=(ready_weight*data[j].ready+720)+(due_weight*data[j].due);
   raw_bias /= 240.0;
   true_bias[j] = pow(10.0,raw_bias);
   }

/* Create the initial population of chromosomes.   */

for(i=0;i<total_pop_size;i++)
   {
   for (j=1;j<=number_jobs;j++)
      {
      newchrom[i].alleles[j] = triag(0.0,true_bias[j]);
      if((i-subpop_size*subpop_number[i])<strong_bias)
```

```
         newchrom[i].alleles[j] = (int) (0.5*(true_bias[j]));
      newchrom[i].alleles[j]*=10000;
      newchrom[i].alleles[j] += (machine[job[j-1]+
                (pick(job[j]-job[j-1])-1)])*1000;

/*   A check to see if the bias exceeds the feasible range */
/*   for the variable type or any other data problem       */
/*   that would lead to a negative allele value.           */

      if(newchrom[i].alleles[j]<0)
         {
         printf("raw_bias= %f true_bias[%d]= %f",raw_bias,j,true_bias[j]);
         printf(" newchrom[%d].alleles[%d]= %d \n", i,j,
                     newchrom[i].alleles[j]);
         }
      }
   chromosome_evaluation(i);
   }

/* Rank the current population. */
/* First, rank the total population. */
for(i=1;i<=total_pop_size;i++)
   data_to_rank[i]=newchrom[i-1].fitness;
ranking(total_pop_size,rank2);

/* Second, rank the subpopulations. */
for(i=0;i<total_pop_size;i++)
   newchrom[i].pseudo_fitness=
      1000000.0*subpop_number[i]+newchrom[i].fitness;
for(i=1;i<=total_pop_size;i++)
   data_to_rank[i]=newchrom[i-1].pseudo_fitness;
ranking(total_pop_size,rank1);
}
```

The function **reproduction** first employs the elitest strategy by leaving the **number_clones** best solutions in each subpopulation unchanged. Second, the Bernoulli crossover operation is performed. The solutions are ranked within each subpopulation and the **number_immigrants** worst solutions in each subpopulation are replaced using the immigration operator.

```
void reproduction()

/*************************************************************/
/*  The function reproduction() performs the generational    */
/*  changes for the GA. Both Bernoulli crossover and         */
/*  immigration are performed within this function.          */
/*************************************************************/
{
int i,j,k;
int crossover_partner1,crossover_partner2;
float immigration_variate;
int tempchrom_alleles;
float n;

for(i=0;i<total_pop_size;i++)
   oldchrom[i]=newchrom[i];

/* Perform crossover. Notice that the number_clones          */
/* chromosomes with the best objective function measure are */
/* cloned into the next generation (an elitest strategy).    */
for(i=0;i<num_subpops;i++)
   {
   for(j=number_clones;j<subpop_size;j++)
      {
/*  Select crossover partners.  */
      crossover_partner1=pick(subpop_size)-1;
      crossover_partner2=pick(subpop_size)-1;
      crossover_partner1+=i*subpop_size;
      crossover_partner2+=i*subpop_size;
      crossover_partner1=rank1[crossover_partner1];
      crossover_partner2=rank1[crossover_partner2];

/* Perform Bernoulli crossover with probability crossover_prob. */
      for(k=1;k<=number_jobs;k++)
         {
         n = urand();
         if(n>=crossover_prob)
            {
            newchrom[rank1[i*subpop_size+j]].alleles[k]=
            oldchrom[crossover_partner1].alleles[k];
            newchrom[total_pop_size+1].alleles[k]=
            oldchrom[crossover_partner2].alleles[k];
```

```
            }
         else
            {
            newchrom[rank1[i*subpop_size+j]].alleles[k]=
            oldchrom[crossover_partner2].alleles[k];
            newchrom[total_pop_size+1].alleles[k]=
            oldchrom[crossover_partner1].alleles[k];
            }
         }

   /* Evaluate the two solutions and retain the best one. */
      chromosome_evaluation(rank1[i*subpop_size+j]);
      chromosome_evaluation(total_pop_size+1);
      if (newchrom[total_pop_size+1].fitness <
         newchrom[rank1[i*subpop_size+j]].fitness)
            newchrom[rank1[i*subpop_size+j]]=newchrom[total_pop_size+1];

      }
   }

/* Rank the current population. */
/* First, rank the total population. */
for(i=1;i<=total_pop_size;i++)
   data_to_rank[i]=newchrom[i-1].fitness;
ranking(total_pop_size,rank2);

/* Second, rank the subpopulations. */
for(i=0;i<total_pop_size;i++)
   newchrom[i].pseudo_fitness=
      1000000.0*subpop_number[i]+newchrom[i].fitness;
for(i=1;i<=total_pop_size;i++)
   data_to_rank[i]=newchrom[i-1].pseudo_fitness;
ranking(total_pop_size,rank1);

/* Perform the immigration operation. */
for(i=0;i<num_subpops;i++)
   {
   for(j=subpop_size-number_immig;j<subpop_size;j++)
      {

/*   Determine whether the variate will be strongly  */
/*   biased or not.                                  */
```

```
      immigration_variate=urand();
      if(immigration_variate<immig_type_rate)
/*       Create a non-strongly biased immigrant.     */
         for(k=1;k<=number_jobs;k++)
            {
            newchrom[rank1[i*subpop_size+j]].alleles[k] =
                  triag(0.0,true_bias[k]);
            newchrom[rank1[i*subpop_size+j]].alleles[k]*=10000;
            newchrom[rank1[i*subpop_size+j]].alleles[k]+=
            (machine[job[k-1]+(pick(job[k]-job[k-1])-1)])*1000;
            }
       else
/*       Create a strongly biased immigrant.     */
          for(k=1;k<=number_jobs;k++)
             {
             newchrom[rank1[i*subpop_size+j]].alleles[k]=
                ((int)(0.5*true_bias[k]))*10000;
             newchrom[rank1[i*subpop_size+j]].alleles[k]+=
             (machine[job[k-1]+(pick(job[k]-job[k-1])-1)])*1000;
             }

        }
    }
}
```

The function **communicate** shares chromosomes between the different subpopulations after a fixed number of generations. Note that the sharing always occurs in the same direction between the same two subpopulations.

```
void communicate()

/*************************************************************/
/* The function communicate() shares the best solutions      */
/* from different subpopulations. The best                   */
/* num_to_commun solutions                                   */
/* from the receiving subpopulation are replaced by the best */
/* num_to_commun solutions from the                          */
/* sending subpopulation.                                    */
/*************************************************************/
{
```

```
int i,j;
struct chromosome tempchrom[Max_pop+1];

/* Store the data from the last subpopulation */
/* so that the first subpopulation can access */
/* it at the appropriate time.                */
for(j=0;j<num_to_commun;j++)
   tempchrom[j]=newchrom[rank1[total_pop_size -(subpop_size)+j]];

/* Starting with the last subpopulation, each subpopulation */
/* i (except the first one) replaces its best               */
/* num_to_commun chromosomes with the best                  */
/* num_to_commun chromosomes from population i-1.           */
for(i=num_subpops-1;i>0;i--)
   for(j=0;j<num_to_commun;j++)
      {
      newchrom[rank1[i*subpop_size+j]]=
      newchrom[rank1[(i-1)*subpop_size+j]];
      newchrom[rank1[i*subpop_size+j]].pseudo_fitness=1000000.0*
      subpop_number[rank1[i*subpop_size+j]]+
      newchrom[rank1[i*subpop_size+j]].fitness;
      }

/* Copy the chromosomes from temp to the first subpopulation. */
for(j=0;j<num_to_commun;j++)
   {
   newchrom[rank1[j]]=tempchrom[j];
   newchrom[rank1[j]].pseudo_fitness=1000000.0*
   subpop_number[rank1[j]]+newchrom[rank1[j]].fitness;
   }

for(i=1;i<=total_pop_size;i++)
   data_to_rank[i]=newchrom[i-1].pseudo_fitness;
ranking(total_pop_size,rank1);
for(i=1;i<=total_pop_size;i++)
   data_to_rank[i]=newchrom[i-1].fitness;
ranking(total_pop_size,rank2);
}
```

The function **genetic** runs while the stopping critera are not met (i.e., **stop**=0). There are three stopping criteria: the GA has run for **max_generations** number of gen-

erations, the best solution is equal to a lower bound, and the best solution has remained unchanged for 15 generations and the **number_of_clones** best solutions within the entire population have the same objective function value.

```
void genetic()

/*********************************************************/
/* The function genetic() runs the GA until a stopping  */
/* criterion is met.                                     */
/*********************************************************/
{
int generation_count,stop,print_freq = 1;

/* Initialize. */
stop = 0;
generation_count = 0;
printf("-----------------GENERATION RESULTS---------------------\n\n");
printf("Generation Number  Best Solution Found \n");

/* Continue with new generations until a stopping */
/* criterion is met.                              */
while (1-stop) {
   generation_count++;

/* Communicate between subpopulations when necessary.   */
   if(generation_count%gen_to_commun==0)
      communicate();

/* Perform reproduction to create a new generation.   */
   reproduction();

/* If it is time, print the current best solution.   */
   if(generation_count==print_freq*(generation_count/print_freq))
      {
      printf("%10d               %.2f \n",generation_count,
              newchrom[rank2[1]].fitness);
      }

/* Check to see if the best solution and lower + 5%   */
/* solution need to be updated.                       */
   if(newchrom[rank2[1]].fitness +.01<best_solution)
      {
```

```
      best_solution=newchrom[rank2[1]].fitness;
      best_solution_gen=generation_count;
      times(&after);
      total_time_to_best= ((after.tms_utime-before.tms_utime)+
                          (after.tms_stime-before.tms_stime))/60.0;
      }
   if(newchrom[rank2[1]].fitness +.01 < lblatetem*1.05)
      {
      lb_solution=newchrom[rank2[1]].fitness;
      lb_solution_gen=generation_count;
      lblatetem=0.0; /* reset lblate so only the first solution
                        within 5% is found */
      times(&after);
      total_time_to_lb=((after.tms_utime-before.tms_utime)+
                       (after.tms_stime-before.tms_stime))/60.0;
      }

/* Check the stopping criteria.     */
   if (newchrom[rank2[1]].fitness<=min_target) stop=1;
   if(generation_count >= max_generations) stop = 1;
   if(best_solution_gen+15<=generation_count)
      if(generation_count>50)
         if(newchrom[rank2[number_clones-1]].fitness-.01<best_solution)
            stop = 1;
   }
}
```

3.4 Main Function and Utilities

The function **print_final_solution** prints the final solution to the screen. The output contains the machine assignment, completion time and tardiness for each job.

```
void print_final_solution()

/*********************************************************/
/* The function print_final_solution() prints out the   */
/* final GA solution.                                    */
/*********************************************************/
int m;
{
int job_to_sched;
```

```
int oldtooltype[Max_machines+1],newtype[Max_machines+1];
int i,mach,temp;
double tardiness;
float setuptime;

/* Initialize.                                              */
newchrom[m].fitness=0;
for(i=0;i<=total_number_machines;i++) machendtime[i] = 0;
for(i=0;i<=total_number_machines;i++) oldtooltype[i] = 0;
for(i=1;i<=number_tools;i++)
   {
   toolavail[i]=0;
   toollastmach[i]=0;
   }
printf("--------------------FINAL SOLUTION-------------------------\n\n");
printf("  Job     Machine    Completion  Tardiness For     Total  \n");
printf(" Number  Assignment     Time       This Job      Tardiness\n");

/* Sort the random key values for the jobs.             */
for (i=1;i<= number_jobs;i++)
   temp_alleles[i] = newchrom[m].alleles[i]+i;
heapsort(number_jobs);

/* Construct a semi-active schedule based on the sorted */
/* order of the random keys.            */
for (i=1;i<=number_jobs;i++)
   {
/* Determine the job number and machine assignment.              */
/* Note that the job number information                         */
/* is contained in the 3 least significant digits of the allele   */
/* (this needs to be increased if the job number exceeds 1000)    */
/* and the machine assignment is contained in the fourth least    */
/* significant digit of the allele (this could be extended if the */
/* machine number exceeds 10).                                  */
   temp=temp_alleles[i]%10000;
   mach=(temp)/1000;
   job_to_sched = temp-(mach)*1000;

/* Calculate the appropriate setup time.      */
   if(data[job_to_sched].tool==oldtooltype[mach] &&
       toollastmach[data[job_to_sched].tool]==mach)
      setuptime=0;
```

```
   else
      setuptime=setupvalue[oldtooltype[mach]][data[job_to_sched].tool];

/* Calculate the starting time and subsequent */
/* completion time for each job.              */
   if (machendtime[mach]<toolavail[data[job_to_sched].tool])
      machendtime[mach]=toolavail[data[job_to_sched].tool];
   machendtime[mach]+=setuptime;
   if (machendtime[mach]<data[job_to_sched].ready)
       machendtime[mach]=data[job_to_sched].ready;
   machendtime[mach] = toolavail[data[job_to_sched].tool] =
       machendtime[mach] + data[job_to_sched].proctime;

/* Update the tool indicators.         */
   toollastmach[data[job_to_sched].tool]=mach;
   oldtooltype[mach]=data[job_to_sched].tool;

/* Determine the tardiness.           */
   tardiness=0.0;
   if (machendtime[mach] - data[job_to_sched].due>0)
      newchrom[m].fitness += tardiness =
      machendtime[mach]-data[job_to_sched].due;

/* Print out the information for each job.       */

   printf("%5d %3d %6.2f %6.2f %6.2f \n",job_to_sched,mach,
              machendtime[mach],tardiness,newchrom[m].fitness);
   }
printf("The objective value is = %6.2f \n",newchrom[m].fitness);
}
```

The function **urand** randomly generates uniform $(0,1)$ variates using the standard function **random**.

```
float urand()

/*********************************************************/
/* The function urand() generates a                      */
/* uniform (0,1) variate.                                */
/*********************************************************/
{
float p;
```

```
p = random();
p = p/maxint;
return p;
}
```

The function **pick** randomly generates integer numbers using the standard function **random**.

```
int pick(n)

/*************************************************************/
/*  The function pick(n) returns an integer in the range    */
/*  1 to n.                                                  */
/*************************************************************/
int n;
{
int p1;
float p2;

p2 = random();
p1 = p2*n/maxint;
p1=p1+1;
if (p1<1) p1 = 1;
if (p1>n) p1 = n;
return p1;
}
```

The function **triag** randomly generates an integer that is triagonally distributed between lo_value and hi_value with mode equal to their average.

```
int traig(lo_value,hi_value)

/*************************************************************/
/*  The function triag(lo_value,hi_value) returns an        */
/*  integer that is triangularly distributed in the range   */
/*  [lo_value, hi_value].                                    */
/*************************************************************/
float lo_value;
float hi_value;
{
```

```
float tempgene1;
int tempgene2;
float midpoint;

midpoint=0.5*(hi_value+lo_value);
tempgene1=urand();
if(tempgene1<(float)(midpoint-lo_value)/(float)(hi_value-lo_value))
   tempgene2=lo_value+sqrt(midpoint-lo_value)*
             sqrt((hi_value-lo_value)*tempgene1);
else
   tempgene2=hi_value-sqrt(hi_value-midpoint)*
             sqrt((hi_value-lo_value)*(1.0-tempgene1));
return tempgene2;
}
```

The function **heapsort** sorts the array temp_alleles in ascending order. It is called by **chromosome_evaluation** in order to sort the random key values.

```
void heapsort(num_to_sort)

/*****************************************************************/
/*  The function heapsort() heapsorts the array                  */
/*  temp_alleles[n] in increasing order.                         */
/*****************************************************************/
int num_to_sort;
{
int stop;
int l,ir,j,i;
int rra;

l=(num_to_sort >> 1) +1;
ir = num_to_sort;

for(;;)
    {
    if(l>1)
        rra=temp_alleles[--l];
    else
        {
        rra=temp_alleles[ir];
        temp_alleles[ir]=temp_alleles[1];
        if( --ir == 1)
```

```
            {
            temp_alleles[1]=rra;
            return;
            }
        }
    i=l;
    j=l << 1;
    while (j<=ir)
        {
        if (j < ir && temp_alleles[j] < temp_alleles[j+1])
            ++j;
        if (rra < temp_alleles[j])
            {
            temp_alleles[i]=temp_alleles[j];
            j += (i=j);
            }
        else j=ir+1;
        }
    temp_alleles[i]=rra;
    }
}
```

The function **ranking** sorts the array data_to_rank in ascending order. It is used to sort the chromosomes based on their fitness. It is used for sorting within subpopulations (rank1) and for sorting the entire population (rank2). In order to sort within the subpopulations, fitness is converted to pseudo-fitness by multiplying the subpopulation number of each member of a subpopulation by a large constant (1000000.0 in this code) and adding that to the actual fitness value. A sort based on pseudo-fitness orders the chromosomes first by subpopulation and then by fitness within each subpopulation. Note that for different problems a larger constant may be required depending on the range of objective function values.

```
void ranking(num_to_rank_by_fitness,rank)

/*************************************************************/
/*  The function ranking() heapsorts the array               */
/*  data_to_rank in increasing order to determine the rank   */
/*  ordering of the elements and returns                     */
```

```
/*  the ranks in the array rank.                                    */
/*******************************************************************/

int num_to_rank_by_fitness;
int rank[Max_pop+1];
{
int stop;
int l,ir,j,i;
double rra;
int temp_rank;

for(i=0;i<num_to_rank_by_fitness;i++)
   rank[i]=i;

l=(num_to_rank_by_fitness >> 1) +1;
ir = num_to_rank_by_fitness;

for(;;)
   {
   if(l>1)
      {
      rra=data_to_rank[--l]; temp_rank=rank[l-1];
      }
   else
      {
      rra=data_to_rank[ir]; temp_rank=rank[ir-1];
      data_to_rank[ir]=data_to_rank[1];
      rank[ir-1]=rank[1-1];
      if( --ir == 1)
         {
         data_to_rank[1]=rra; rank[1-1]=temp_rank;
         return;
         }
      }
   i=l;
   j=l << 1;
   while (j<=ir)
      {
      if (j < ir && data_to_rank[j] < data_to_rank[j+1])
         ++j;
      if (rra < data_to_rank[j])
         {
```

```
          data_to_rank[i]=data_to_rank[j];
          rank[i-1]=rank[j-1];
          j += (i=j);
          }
       else j=ir+1;
       }
   data_to_rank[i]=rra; rank[i-1]=temp_rank;
   }
}
```

The **main** function calls **readdata**, **genetic**, prints final solution information and calls **print_final_solution**.

```
main()

/*********************************************************/
/* Beginning of main program.                            */
/*                                                       */
/*********************************************************/
{
int i,j,generation_count,stop;
int loopseed,seed;

readdata();

/* Run the GA for a number of times based on the values */
/* of startseed and endseed.                            */
for(loopseed=startseed;loopseed<endseed;loopseed++)
        {
        seed=loopseed;
        srandom(seed);

/* Reset the timing values and the best and lower bound */
/* solution values.                                     */
        times(&before);
        time_passed=0.0;
        total_time_to_lb=0;
        total_time_to_best=0;
        best_solution=100000; lb_solution=100000;
        best_solution_gen=0; lb_solution_gen=0;
        lblatetem=lb_value;
```

```
        initialize_population();

        genetic();

/* Calculate and print out the total time that the GA   */
/* ran, the value of the best solution and the time     */
/* required to find it, the time required to get within */
/* 5% of the lower bound, and the final best solution.  */
printf("\n--------------------SUMMARY DATA-------------------------\n\n");
        times(&after);
        time_passed =  ((after.tms_utime-before.tms_utime)+
                        (after.tms_stime-before.tms_stime))/60.0;
        printf("\nThis seed's total time = %.2f seconds.\n",
                time_passed);
        printf("\nThe best solution found is %.2f\n ",
                newchrom[rank2[1]].fitness);
        printf("\nThe best sol. of %.2f found in gen %d ",
                best_solution,best_solution_gen);
        printf("required %.2f seconds.\n ",total_time_to_best);
        printf("\nLb+5 percent solution %.2f found in gen %d ",
                lb_solution,lb_solution_gen);
        printf("required %.2f seconds.\n ",total_time_to_lb);
        printf("\n");
        total_time+=time+passed;

        print_final_solution(rank2[1]);
        printf("\n");
        }/*end seed loop*/
printf("\nThe total program time = %.2f seconds.\n",total_time);
}
```

REFERENCES

Bean, J. C. [1994], "Genetics and Random Keys for Sequencing and Optimization," **ORSA Journal on Computing,** Vol. 6, Spring 1994, 154-160.

Norman, B. A. and J. C. Bean [1994], "Random Keys Genetic Algorithm for Job Shop Scheduling," Technical Report 94-5, Department of Industrial and Operations Engineering, University of Michigan.

Norman, B. A. [1994], Forthcoming Ph.D. Dissertation, Department of Industrial and Operations Engineering, University of Michigan.