The single-machine problem with earliness and tardiness penalties and a common due date

Miguel Díaz Iturry

1 Problem description

There are n jobs at time zero that have to be processed on a single machine and have to be delivered on a common due date. Depending on whether a job is completed before or after due date, it will have a penalty related with the earliness or tardiness of it's delivery.

For each job is given it's processing time p_i, earliness penalty α_i and tardiness penalty β_i. Having the completion time of the job i as C_i, the earliness of the job is calculated by $E_i = max(0, d - C_i)$. As well, the tardiness is calculated by $T_i = max(0, C_i - d)$.

Finally the total penalty of processing all jobs is given by Equation 1.

$$f(S) = \sum_{i=1}^{n} \alpha_i E_i + \sum_{i=1}^{n} \beta_i T_i \tag{1}$$

The objective is to give the schedule of jobs that minimize the total penalty. A schedule should contain the order in which the jobs are processed and the starting time of the first job.

It has been proven that the optimum schedule has three properties:

Property 1 In an optimal solution there are no idle times between the processing of consecutive jobs [2]

Property 2 The schedule is V-shaped, which means that all jobs that are completed before or at due date are sequenced in a decreasing order of ratio p_i/α_i. Jobs that are completed after due date are sequenced in an increasing order of ratio p_i/β_i [1].

Property 3 An optimal schedule exists in which either the processing of the first job starts at time zero or one job is completed at the due date. In first case, there will be a job that starts before due date and delivers after due date, this is named the *straddling job* [1].

2 Construction heuristic (CH)

The heuristic will try to separate jobs in two groups: jobs that end before due date (group B) and jobs that end after due date (group A).

1. Sort jobs as the inverse of the optimum for an earliness schedule:
   $S = \{job_0, job_1, ..., job_n\}, \quad where \quad p_0/\alpha_0 \leq p_1/\alpha_1 \leq ... \leq p_n/\alpha_n$
2. Score all jobs according to their position in S:
   $score1(job_i) = i$
3. Sort jobs as the optimum for a tardiness schedule:
   $S' = \{job_0, job_1, ..., job_n\}, \quad where \quad p_0/\beta_0 \leq p_1/\beta_1 \leq ... \leq p_n/\beta_n$
4. Update scores according to their position in S':
   $score2(job_i) = i$
5. Calculate final score for each job:
   $finalScore(job_i) = score1(job_i) - score2(job_i)$
6. Sort jobs decreasingly according to their final scores:
   $P = \{job_0, job_1, ..., job_n\} \quad where \quad score(job_0) \geq score(job_1) \geq ... \geq score(job_n)$.
7. For each job in P decide scheduling it before or after due date calculating how much it increases the penalty in each case and taking the lowest one. If a job cannot be completed before due date, schedule it after due date.

In the algorithm, the score tries to balance the *belonging* of a job to the groups A and B.

Considering insertion of jobs in group B, step 1 and 2 score jobs giving a greater score to the one that affects the less to penalty.

In step 3 and 4 the same idea is done, but considering insertion to group A.

Step 5 calculate the final score of each job by the difference $score1 - score2$ trying to balance the *belonging* of the jobs to the groups A and B. Jobs that have the greatest $finalScore$ are those that affect the less when being processed before due date.

Step 6 sort in decreasingly order of $finalScore$ to give priority to jobs that have major *belonging* to the group B.

Step 7 process each work and decides if inserting it in group A or B, maintaining the V-shape property.

3 Improvement heuristic (IH)

The IH proposed builds the neighborhood using two movements. The first movement changes the group of one job, that is if the job was scheduled before due date, now it is scheduled after, and vice-versa.

The second movement consists on interchanging one job scheduled before due date with other job scheduled after. For both movements the IH maintains the V-Shape arrangement of jobs.

When executing the two movements, the IH proposed selects the new optimum solution taking the best of all in the generated neighborhood. The IH algorithm stops when the selected solution does not improve the previous solution.

4 Genetic Algorithm (GA)

A Genetic Algorithm (GA) was implemented in order to explore a greater space of solutions.

The steps followed for the metaheristics are:

1. Generate $p1$ by CH and IH
2. Generate N_PAR-1 parents by a random construction and then applying IH ($N\_PAR = 10$)
3. Create N_CHLD selecting each time a pair of parents by a tournament with a geometric distribution ($P = 0.15$) and applying one of the 6 types of crossovers. For $n = [10-500]$, $N\_CHLD = 30$; for $n = 1000$, $N\_CHLD = 15$
4. Apply mutation operator: change a job's group with a probability $P\_MUT = 0.6$
5. Apply IH in the new generation
6. Select new parents taking the N_PAR best solutions among parents and children
7. Iterate until number of generations without improvement reaches the value $MAX\_ITER = 10$ or if reaching 30 minutes of running by problem

All solutions maintain the V-shape property and do not have *straddling* jobs. The performed crossovers are:

1. Process the job after due time if it was processed after due time by any of the parents.
2. Process the job after due time if it was processed before due time by any of the parents.
3. Process the job after due time if it was processed after due time by both of the parents.
4. Process the job after due time if it was processed before due time by both of the parents.

5. Process the job after due time if it was processed before due time by the first parent and after due time by the second parent.

6. Process the job after due time if it was processed after due time by the first parent and before due time by the second parent.

In any of the crossovers if a job cannot be processed before due time because of time constraints, it will be processed after.

5 Results

280 problems generated in [1] were used to test the proposed heuristics. There are 10 different problems for different number of jobs (n = 10, 20, 50, 100, 200 ,500 and 1000), and each of those problems use four restrictive factors (h = 0.2, 0.4, 0.6 and 0.8) to calculate the due date using the following expression: $d = \lfloor h \sum p_i \rfloor$.

5.1 Running time

The algorithm for the CH has a complexity $O(n^2)$, the average running time for different number of jobs are given in Table 1.

Table 1: Running time of the CH

Jobs	10	20	50	100	200	500	1000
Time (μs)	9.8	21	186	437	769	2390	6673

The neighborhood creation algorithm for the IH has a complexity $O(n^3)$, the average running time for different number of jobs are given in Table 2.

Table 2: Running time of the IH

Jobs	10	20	50	100	200	500	1000
Time (s)	$105 * 10^{-6}$	$183 * 10^{-6}$	$3.97 * 10^{-3}$	$46.48 * 10^{-3}$	0.6	21.95	336

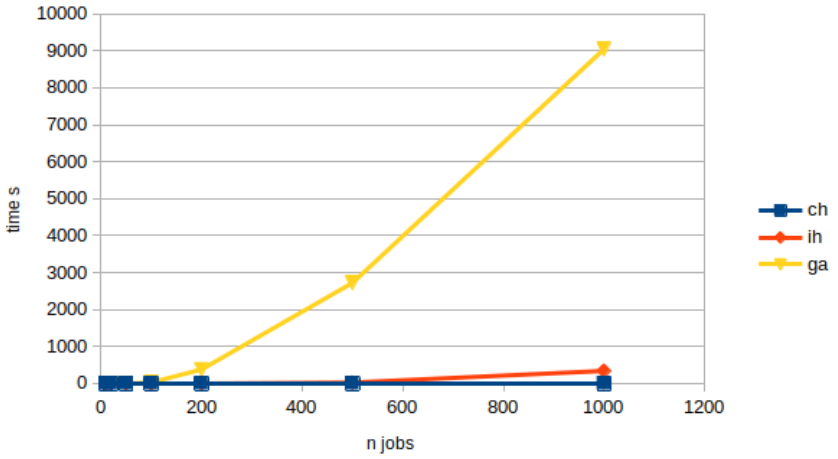
The GA increased the running time as shown in Table 3. For problems with 1000 jobs, the GA were stopped after first generation due the time limit.

Table 3: Running time of the GA

Jobs	10	20	50	100	200	500	1000
Time (s)	$6.54 * 10^{-3}$	$59.82 * 10^{-3}$	1.74	23.88	375	2719	9047

Figure 1 shows the difference of running times between the three proposed heuristics.

Figure 1: Running times of heuristics by number of jobs



5.2 Penalties

After scheduling the jobs using the CH, the resulting penalty for each problem was calculated and compared with benchmarks exposed in [1]. The average percentage difference with these benchmarks for each number of jobs and restrictive factor (h) are exposed in Table 4.

Table 4: Percentage difference of the CH vs. Biskup's benchmarks

Jobs	h=0.2	h=0.4	h=0.6	h=0.8	**Mean**
10	10.28	28.3	22.21	14.79	18.89
20	5.6	14.7	18.06	18.48	14.21
50	6.09	13.6	22.18	20.87	15.68
100	6.91	15.3	20.09	17.93	15.06
200	6.3	17.44	19.91	20.45	16.03
500	-0.67	8.43	27.43	27.69	15.72
1000	-0.52	7.05	27.46	27.12	15.27
Mean	4.85	14.97	22.48	21.05	15.84

It can be seen that the CH performs worse that Biskup heuristic and by the distribution of results, it cannot be concluded any correlation between the number of jobs or h against the penalty obtained.

The resulting penalties of the IH were as well compared against the benchmarks as shown in Table 5.

Table 5: Percentage difference of the IH vs. Biskup's benchmarks

Jobs	h=0.2	h=0.4	h=0.6	h=0.8	**Mean**
10	3.18	5.9	1.14	0.0	2.55
20	-1.05	0.67	-0.42	-0.41	-0.30
50	-3.39	-3.01	-0.31	-0.23	-1.73
100	-3.75	-3.37	-0.12	-0.17	-1.85
200	-3.09	-2.52	-0.15	-0.15	-1.48
500	-3.95	-2.82	-0.11	-0.11	-1.75
1000	-4.0	-3.22	-0.06	-0.06	-1.84
Mean	-2.29	-1.20	-0.01	-0.16	-0.91

In the table can be seen that the proposed heuristic tends to perform worst when the problem is bigger and the factor h increases. Nevertheless, the IH got to improve to the benchmarks by a 0.91 %

Table 6 shows the performance of the IH against the CH.

Table 6: Percentage difference of the IH vs CH

Jobs	h=0.2	h=0.4	h=0.6	h=0.8	**Mean**
10	-5.41	-14.85	-16.66	-12.19	-12.28
20	-6.1	-11.81	-14.79	-15.55	-12.06
50	-8.82	-14.15	-18.19	-17.23	-14.60
100	-9.79	-16.04	-16.77	-15.25	-14.46
200	-8.77	-16.81	-16.7	-17.03	-14.83
500	-3.3	-10.36	-21.61	-21.76	-14.26
1000	-3.5	-9.58	-21.58	-21.37	-14.01
Mean	-6.53	-13.37	-18.04	-17.20	-13.78

Table 7 shows the comparison between proposed GA and Biskup's benchmarks.

Similar with the IH, GA results are worst for bigger problems with greater factor h; however, they improve in a greater percentage (1.89%) to the benchmarks.

Table 7: Percentage difference of the GA vs. Biskup's benchmarks

Jobs	h=0.2	h=0.4	h=0.6	h=0.8	**Mean**
10	1.99	0.54	0.02	0.0	0.64
20	-3.74	-1.46	-0.61	-0.41	-1.55
50	-5.54	-4.62	-0.34	-0.24	-2.68
100	-5.73	-4.84	-0.15	-0.18	-2.72
200	-5.24	-3.62	-0.15	-0.15	-2.29
500	-5.58	-3.4	-0.11	-0.11	-2.30
1000	-5.14	-4.06	-0.06	-0.06	-2.33
Mean	-4.14	-3.06	-0.20	-0.16	-1.89

Table 8 shows the difference between GA and IH results and finally Figure 2 shows the percentage difference of the three heuristics against benchmarks.

Table 8: Percentage difference of the GA vs. IH

Jobs	h=0.2	h=0.4	h=0.6	h=0.8	**Mean**
10	-1.12	-4.37	-1.0	0.0	-1.62
20	-2.69	-2.05	-0.19	0.0	-1.23
50	-2.22	-1.66	-0.02	-0.01	-0.98
100	-2.05	-1.53	-0.02	-0.0	-0.90
200	-2.21	-1.11	-0.0	-0.0	-0.83
500	-1.7	-0.6	-0.0	-0.0	-0.57
1000	-1.19	-0.86	-0.0	-0.0	-0.51
Mean	-1.88	-1.74	-0.18	-0.00	-0.95

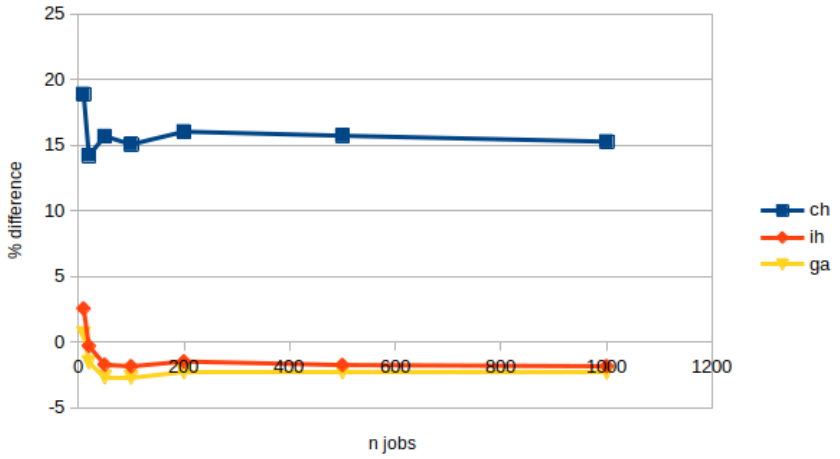
6 Conclusions

The CH didn't improve the results of benchmarks established by Biskup. The average difference between the present construction algorithm and the benchmarks published by Biskup [1] was 15.84%.

It can be seen in Table 4 that the results of the CH don't show any tendency by number of problems.

With respect to the variations for different h, the only one that performed

Figure 2: Percentage difference against benchmarks



differently was for $h = 0.2$ being the lower difference. Nevertheless, it cannot be concluded that the CH performs better for lower h, because for $h = 0.4, h = 0.6, h = 0.8$ the results don't show any tendency.

The IH as expected, improved the results obtained by the CH in a 13.78%. Also, it improved the results presented in [1] by a 0.91%.

The GA improved in a 1.89% the results of Biskup's benchmarks and in a 0.95% to the IH.

In regards to the Figure 1, it can be seen a considerable increment in the run time of the IH against the CH and the GA against the IH.

One great problem with the meta heuristic proposed is that, as it runs the local search for each children, it runs out of time in each generation. A possible improvement for running time was tested, in which the local search was not performed for each children; nevertheless, the GAs could not improve the IH's results.

7 Appendix

Source code:

```
#include <bits/stdc++.h>
#include <math.h>
#include <chrono>

#include <iostream>
#include <fstream>
```

```
using namespace std;

double hs[] = {0.2, 0.4, 0.6, 0.8};

struct job
{
  int p, a, b;
} jobs[1005];

const bool VALIDATE = 0;
const bool verbose = 0;

// Constants for GA

const int N_PAR = 10;          // Number of parents
const int N_CHD = 30;          // Number of children multiple of qty. of crossovers
const double P = 0.15;         // Probability for tournament (geometric distribution)
const double IMP_EPS = 0;      // EPS to use in improvement function
const int MAX_GEN = 1000;      // Maximum generations without improvement
const int TIME_PP = 30 * 60; // Limit time (s) by problem
const double MUTATION = 0.2; // Probability for a gen to mutate

const int CROSS_OVERS[] = {0, 1, 2, 3, 4, 5}; // Crossovers to apply
int n_crosses = sizeof(CROSS_OVERS) / sizeof(CROSS_OVERS[0]);

struct gen
{
  bool distribution[1005];
  int cost;

  bool operator<(const gen &g)
  {
    return cost < g.cost;
  }
} generation[N_PAR + N_CHD];

int ps,
    n;

int prev_job[1005], next_job[1005];
int group_a, group_b;
int expected_a[1005];
int expected_b[1005];
int ab_score[1005];
int process_order[1005];
int order_b[1005];
bool in_group_a[1005];
int best_cost;

ifstream in;
ofstream c_out;
ofstream i_out;
ofstream m_out;
ofstream log_out;

void random_code(char *s, int sz)
{
```

```
  for (int i = 0; i < sz - 1; i++)
  {
    s[i] = 'A' + (rand() % 26);
  }
  s[sz - 1] = 0;
  return;
}

void print_config()
{
  log_out << "CONFIG" << '\n';
  log_out << "\tN_PAR: " << N_PAR << '\n';
  log_out << "\tN_CHD: " << N_CHD << '\n';
  log_out << "\tP: " << P << '\n';
  log_out << "\tMUTATION: " << MUTATION << '\n';
  log_out << "\tCROSS_OVERS: ";
  for (int i = 0; i < n_crosses; i++)
  {
    if (i)
    {
      log_out << ", ";
    }
    log_out << CROSS_OVERS[i];
  }
  log_out << endl;
}

void print_group(int ini)
{
  for (int j = ini; j != -1;)
  {
    log_out << j << " ";
    j = next_job[j];
  }
  log_out << '\n';
}

int insert_a(int job)
{
  int cost_a = 0;
  int last_j = -1, lt = 0;
  bool inserted = false;
  for (int j = group_a; j != -1; j = next_job[j])
  {
    if (!inserted && jobs[job].p * jobs[j].b < jobs[j].p * jobs[job].b)
    {
      inserted = true;

      next_job[job] = j;
      prev_job[job] = prev_job[j];
      prev_job[j] = job;
      if (prev_job[job] == -1)
      {
        group_a = job;
      }
      else
      {
```

```
        next_job[prev_job[job]] = job;
      }

      lt += jobs[job].p;
      cost_a += lt * jobs[job].b;
    }
    lt += jobs[j].p;
    cost_a += lt * jobs[j].b;
    last_j = j;
  }
  if (!inserted)
  {
    if (group_a == -1)
    {
      group_a = job;
      prev_job[job] = -1;
      next_job[job] = -1;
    }
    else
    {
      next_job[last_j] = job;
      prev_job[job] = last_j;
      next_job[job] = -1;
    }

    lt += jobs[job].p;
    cost_a += lt * jobs[job].b;
  }
  in_group_a[job] = true;
  return cost_a;
}

int insert_b(int job)
{
  int cost_b = 0;
  int last_j = -1, lt = 0;
  bool inserted = false;
  for (int j = group_b; j != -1; j = next_job[j])
  {
    if (!inserted && jobs[job].p * jobs[j].a < jobs[j].p * jobs[job].a)
    {
      inserted = true;

      next_job[job] = j;
      prev_job[job] = prev_job[j];
      prev_job[j] = job;
      if (prev_job[job] == -1)
      {
        group_b = job;
      }
      else
      {
        next_job[prev_job[job]] = job;
      }

      cost_b += lt * jobs[job].a;
      lt += jobs[job].p;
```

```
    }
    cost_b += lt * jobs[j].a;
    lt += jobs[j].p;
    last_j = j;
  }
  if (!inserted)
  {
    if (group_b == -1)
    {
      group_b = job;
      prev_job[job] = -1;
      next_job[job] = -1;
    }
    else
    {
      next_job[last_j] = job;
      prev_job[job] = last_j;
      next_job[job] = -1;
    }

    cost_b += lt * jobs[job].a;
    lt += jobs[job].p;
  }
  in_group_a[job] = false;
  return cost_b;
}

void remove_a(int job)
{
  if (prev_job[job] == -1)
  {
    group_a = next_job[job];
  }
  else
  {
    next_job[prev_job[job]] = next_job[job];
  }
  if (next_job[job] != -1)
  {
    prev_job[next_job[job]] = prev_job[job];
  }
  in_group_a[job] = false;
}

void remove_b(int job)
{
  if (prev_job[job] == -1)
  {
    group_b = next_job[job];
  }
  else
  {
    next_job[prev_job[job]] = next_job[job];
  }
  if (next_job[job] != -1)
  {
    prev_job[next_job[job]] = prev_job[job];
```

```
  }
}

int cost_group_b()
{
  int cost = 0, lt = 0;
  for (int i = group_b; i != -1; i = next_job[i])
  {
    cost += lt * jobs[i].a;
    lt += jobs[i].p;
  }

  return cost;
}

int cost_group_a()
{
  int cost = 0, lt = 0;
  for (int i = group_a; i != -1; i = next_job[i])
  {
    lt += jobs[i].p;
    cost += lt * jobs[i].b;
  }

  return cost;
}

int calculate_cost()
{
  return cost_group_a() + cost_group_b();
}

void clean_groups()
{
  group_a = -1;
  group_b = -1;
  for (int i = 0; i < n; i++)
  {
    next_job[i] = -1;
    prev_job[i] = -1;
  }
}

void fill_groups(bool *arr)
{
  clean_groups();
  for (int i = 0; i < n; i++)
  {
    if (arr[i])
    {
      insert_a(i);
    }
    else
    {
      insert_b(i);
    }
  }
```

```
}

bool validate_group_b(int d)
{
  for (int i = group_b; i != -1; i = next_job[i])
  {
    d -= jobs[i].p;
  }
  return d >= 0;
}

bool validate_solution(int d)
{
  bool f_ans = true;
  bool ans[n];
  memset(ans, 0, n);
  for (int i = group_b; i != -1 && f_ans; i = next_job[i])
  {
    f_ans &= !ans[i] & !in_group_a[i];
    ans[i] = true;
  }
  if (!f_ans)
  {
    cout << "\tERROR_1" << endl;
    log_out << "\tERROR_1" << '\n';
  }
  for (int i = group_a; i != -1 && f_ans; i = next_job[i])
  {
    f_ans &= !ans[i] & in_group_a[i];
    ans[i] = true;
  }
  if (!f_ans)
  {
    cout << "\tERROR_2" << endl;
    log_out << "\tERROR_2" << '\n';
  }
  for (int i = 0; i < n && f_ans; i++)
  {
    f_ans &= ans[i];
  }
  if (!f_ans)
  {
    cout << "\tERROR_3" << endl;
    log_out << "\tERROR_3" << '\n';
  }
  return f_ans && validate_group_b(d);
}

bool long_validation(int d)
{
  bool ans = validate_solution(d);
  if (!ans)
  {
    cout << "\tERROR_4" << endl;
    log_out << "\tERROR_4" << '\n';
  }
```

```
  bool aux_group[n];
  for (int i = 0; i < n; i++)
  {
    aux_group[i] = in_group_a[i];
  }
  fill_groups(aux_group);
  for (int i = 0; i < n && ans; i++)
  {
    ans &= (aux_group[i] == in_group_a[i]);
  }
  if (!ans)
  {
    cout << "\tERROR_5" << endl;
    log_out << "\tERROR_5" << '\n';
  }

  return ans;
}

void shuffle_process_order()
{
  random_shuffle(&process_order[0], &process_order[n]);
}

/* Process order considering earliness vs tardiness score, take best
between cost_a and cost_b */
int construction(int d)
{
  for (int i = 0; i < n; i++)
  {
    expected_a[i] = i;
    expected_b[i] = i;
    process_order[i] = i;
  }
  sort(expected_a, expected_a + n,
       [](int a, int b)
       { return jobs[a].p * jobs[b].b < jobs[b].p * jobs[a].b; });
  sort(expected_b, expected_b + n,
       [](int a, int b)
       { return jobs[a].p * jobs[b].a < jobs[b].p * jobs[a].a; });

  memset(ab_score, 0, n);

  for (int i = 0; i < n; i++)
  {
    ab_score[expected_b[i]] += i;
    ab_score[expected_a[i]] -= i;
  }

  for (int i = 0; i < n; i++)
  {
    order_b[expected_b[i]] = i;
  }

  sort(process_order, process_order + n,
       [](int a, int b)
       { return ab_score[a] > ab_score[b] || ab_score[a] == ab_score[b] &&
```

```
                                                    order_b[a] > order_b[b]; });
  clean_groups();
  int cost_a = 0;
  int cost_b = 0;
  for (int i = 0; i < n; i++)
  {
    int job = process_order[i];
    int ncost_a = insert_a(job);
    if (jobs[job].p <= d)
    {
      remove_a(job);
      int ncost_b = insert_b(job);
      if (ncost_b > ncost_a)
      {
        remove_b(job);
        insert_a(job);
      }
      else
      {
        cost_b = ncost_b;
        d -= jobs[job].p;
        ncost_a = cost_a;
      }
    }
    cost_a = ncost_a;
  }
  int cost = cost_a + cost_b;
  if (true && d && d < jobs[group_a].p)
  {
    int lt = d;
    cost_b = 0;
    for (int j = group_b; j != -1; j = next_job[j])
    {
      cost_b += lt * jobs[j].a;
      lt += jobs[j].p;
    }
    lt = -d;
    cost_a = 0;
    for (int j = group_a; j != -1; j = next_job[j])
    {
      lt += jobs[j].p;
      cost_a += lt * jobs[j].b;
    }
    cost = min(cost, cost_a + cost_b);
  }
  return cost;
}

int improvement(int cost, int d, double EPS)
{
  for (int j = group_b; j != -1; j = next_job[j])
  {
    d -= jobs[j].p;
  }
  while (true)
  {
    int n_cost = cost;
```

```
int b = -1, a = -1;

for (int i = 0; i < n; i++)
{
  bool started_in_a = in_group_a[i];
  if (started_in_a)
  {
    remove_a(i);
    insert_b(i);
    if (d - jobs[i].p >= 0)
    {
      int aux = calculate_cost();
      if (aux < n_cost)
      {
        n_cost = aux;
        a = i;
        b = -1;
      }
    }
  }
  else
  {
    remove_b(i);
    insert_a(i);
    int aux = calculate_cost();
    if (aux < n_cost)
    {
      n_cost = aux;
      b = i;
      a = -1;
    }
  }
  for (int j = i + 1; j < n; j++)
  {
    if (started_in_a && !in_group_a[j] && d + jobs[j].p - jobs[i].p >= 0)
    {
      remove_b(j);
      insert_a(j);
      int aux = calculate_cost();
      if (aux < n_cost)
      {
        n_cost = aux;
        b = j;
        a = i;
      }
      remove_a(j);
      insert_b(j);
    }
    else if (!started_in_a && in_group_a[j] && d + jobs[i].p - jobs[j].p >= 0)
    {
      remove_a(j);
      insert_b(j);
      int aux = calculate_cost();
      if (aux < n_cost)
      {
        n_cost = aux;
        a = j;
```

```
            b = i;
          }
          remove_b(j);
          insert_a(j);
        }
      }
      if (started_in_a)
      {
        remove_b(i);
        insert_a(i);
      }
      else
      {
        remove_a(i);
        insert_b(i);
      }
    }

    if ((double)(cost - n_cost) / cost * 100 <= EPS || (a == -1 && b == -1))
    {
      break;
    }
    if (b != -1)
    {
      remove_b(b);
      insert_a(b);
      d += jobs[b].p;
    }
    if (a != -1)
    {
      remove_a(a);
      insert_b(a);
      d -= jobs[a].p;
    }
    cost = n_cost;
  }
  return cost;
}

void generate_random_solution(bool *arr, int d, double c)
{
  for (int i = 0; i < n; i++)
  {
    int j = process_order[i];
    arr[j] = (rand() % 100 < 100.0 / N_PAR * c);

    // force going to group A for time constraint
    arr[j] |= jobs[j].p > d;

    if (!arr[j])
    {
      d -= jobs[j].p;
    }
  }
  shuffle_process_order();
}
```

```
void cross1(bool *p1, bool *p2, bool *s, int d)
{
  for (int i = 0; i < n; i++)
  {
    int j = process_order[i];
    bool mutate = (rand() % 100) < (MUTATION * 100);
    s[j] = jobs[j].p > d || ((p1[j] || p2[j]) ^ mutate);
    if (!s[j])
    {
      d -= jobs[j].p;
    }
  }
  shuffle_process_order();
}

void cross2(bool *p1, bool *p2, bool *s, int d)
{
  for (int i = 0; i < n; i++)
  {
    int j = process_order[i];
    bool mutate = (rand() % 100) < (MUTATION * 100);
    s[j] = jobs[j].p > d || ((!p1[j] || !p2[j]) ^ mutate);
    if (!s[j])
    {
      d -= jobs[j].p;
    }
  }
  shuffle_process_order();
}

void cross3(bool *p1, bool *p2, bool *s, int d)
{
  for (int i = 0; i < n; i++)
  {
    int j = process_order[i];
    bool mutate = (rand() % 100) < (MUTATION * 100);
    s[j] = jobs[j].p > d || ((p1[j] && p2[j]) ^ mutate);
    if (!s[j])
    {
      d -= jobs[j].p;
    }
  }
  shuffle_process_order();
}

void cross4(bool *p1, bool *p2, bool *s, int d)
{
  for (int i = 0; i < n; i++)
  {
    int j = process_order[i];
    bool mutate = (rand() % 100) < (MUTATION * 100);
    s[j] = jobs[j].p > d || ((!p1[j] && !p2[j]) ^ mutate);
    if (!s[j])
    {
      d -= jobs[j].p;
    }
  }
```

```
  shuffle_process_order();
}

void cross5(bool *p1, bool *p2, bool *s, int d)
{
  for (int i = 0; i < n; i++)
  {
    int j = process_order[i];
    bool mutate = (rand() % 100) < (MUTATION * 100);
    s[j] = jobs[j].p > d || ((!p1[j] && p2[j]) ^ mutate);
    if (!s[j])
    {
      d -= jobs[j].p;
    }
  }
  shuffle_process_order();
}

void cross6(bool *p1, bool *p2, bool *s, int d)
{
  for (int i = 0; i < n; i++)
  {
    int j = process_order[i];
    bool mutate = (rand() % 100) < (MUTATION * 100);
    s[j] = jobs[j].p > d || ((p1[j] && !p2[j]) ^ mutate);
    if (!s[j])
    {
      d -= jobs[j].p;
    }
  }
  shuffle_process_order();
}

/* Create pointers to crosses functions to automatize calls */
void (*crosses[])(bool *, bool *, bool *, int) = {&cross1, &cross2,
    &cross3, &cross4, &cross5, &cross6};

default_random_engine generator;
geometric_distribution<int> distribution(P);

// geometric distribution
int run_tournament()
{
  int winner;
  do
  {
    winner = distribution(generator);
  } while (winner >= N_PAR);
  return winner;
}

void ga(int d)
{
  memcpy(generation[0].distribution, in_group_a, n);
  generation[0].cost = best_cost;

  if (verbose)
```

```
{
  log_out << "\nPARENT " << 0 << " cost: " << generation[0].cost << "\nGROUP A\n";
  print_group(group_a);
  log_out << "GROUP B\n";
  print_group(group_b);
}
for (int i = 1; i < N_PAR; i++)
{
  generate_random_solution(generation[i].distribution, d, i);
  fill_groups(generation[i].distribution);
  int aux_cost = calculate_cost();
  generation[i].cost = improvement(aux_cost, d, IMP_EPS);
  memcpy(generation[i].distribution, in_group_a, n);

  if (verbose)
  {
    log_out << "\nPARENT " << i << " cost: " << generation[i].cost << "\nGROUP A\n";
    print_group(group_a);
    log_out << "GROUP B\n";
    print_group(group_b);
  }
}

sort(generation, generation + N_PAR);
if (generation[0].cost < best_cost)
{
  cout << "\t\tIMPROVING " << best_cost << " > " << generation[0].cost << endl;
  log_out << "\t\tIMPROVING " << best_cost << " > " << generation[0].cost << '\n';
  best_cost = generation[0].cost;
}

int cycles = 0;
int n_gen = 0;

auto tini = std::chrono::high_resolution_clock::now();

while (cycles++ < MAX_GEN)
{
  n_gen++;
  if (verbose)
  {
    cout << "\tGENERATION: " << n_gen << endl;
    log_out << "\tGENERATION: " << n_gen << '\n';
  }

  int idx_crosses = 0;
  for (int i = N_PAR; i < N_PAR + N_CHD; idx_crosses++, i++)
  {
    int p1 = run_tournament();
    int p2 = run_tournament();

    if (verbose)
    {
      log_out << "\nSELECTED: p1 " << p1 << " cost: "
          << generation[p1].cost << " && p2 " << p2 << " cost: "
          << generation[p2].cost << "\n";
    }
```

```
      if (idx_crosses == n_crosses)
      {
        idx_crosses = 0;
      }
      int idx_aux = CROSS_OVERS[idx_crosses];
      crosses[idx_aux](generation[p1].distribution,
        generation[p2].distribution, generation[i].distribution, d);
      fill_groups(generation[i].distribution);

      int aux_cost = calculate_cost();

      if (verbose)
      {
        log_out << "\nCHILD " << i << " cost: " << aux_cost << "\nGROUP A\n";
        print_group(group_a);
        log_out << "GROUP B\n";
        print_group(group_b);
      }

      generation[i].cost = improvement(aux_cost, d, IMP_EPS);
      memcpy(generation[i].distribution, in_group_a, n);

      if (verbose)
      {
        log_out << "\nIMPROVED " << i << " cost: " << generation[i].cost << "\nGROUP A\n";
        print_group(group_a);
        log_out << "GROUP B\n";
        print_group(group_b);
      }
    }

    sort(generation, generation + N_PAR + N_CHD);
    if (generation[0].cost < best_cost)
    {
      cout << "\t\tIMPROVING " << best_cost << " > " << generation[0].cost << endl;
      log_out << "\t\tIMPROVING " << best_cost << " > " << generation[0].cost << '\n';
      cycles = 0;
      best_cost = generation[0].cost;
    }
    auto tend = std::chrono::high_resolution_clock::now();

    if (chrono::duration_cast<chrono::seconds>(tend - tini).count() >= TIME_PP)
    {
      cout << "Timeout!" << endl;
      log_out << "Timeout!" << '\n';
      cycles = MAX_GEN;
    }
  }
  fill_groups(generation[0].distribution);
}

int main(int argv, char **argvs)
{
  srand(time(NULL));
  char inFile[30], outFile[30], logFile[30], code[5];
  random_code(code, 5);
  cout << code << endl;
```

```
sprintf(inFile, "dados/sch%s.txt", argvs[1]);
sprintf(outFile, "out%s_%s.txt", argvs[1], code);
sprintf(logFile, "log%s_%s.txt", argvs[1], code);

in.open(inFile);
c_out.open("c_" + string(outFile));
i_out.open("i_" + string(outFile));
m_out.open("m_" + string(outFile));
log_out.open(logFile);

cout << "INPUT " << inFile << endl;
log_out << "INPUT " << inFile << '\n';
cout << "OUTPUT " << outFile << endl;
log_out << "OUTPUT " << outFile << '\n';

print_config();

in >> ps;
auto ttini = std::chrono::high_resolution_clock::now();
//
while (ps--)
{
  in >> n;

  int total_time = 0;

  for (int i = 0; i < n; i++)
  {
    in >> jobs[i].p >> jobs[i].a >> jobs[i].b;
    total_time += jobs[i].p;
  }

  for (int i = 0; i < 4; i++)
  {
    cout << "Problem: " << ps << " " << hs[i] << endl;
    log_out << "Problem: " << ps << " " << hs[i] << '\n';

    int d = total_time * hs[i];
    int cost = construction(d);
    // print constructive result
    c_out << cost << endl;
    best_cost = INT_MAX;
    best_cost = improvement(cost, d, IMP_EPS);
    // print improvement result
    i_out << best_cost << endl;
    ga(d);

    if (VALIDATE && !long_validation(d))
    {
      cout << "\tERROR validation\n";
      log_out << "\tERROR validation\n";
      m_out << -1 << endl;
      continue;
    }
    // print metaheuristic result
    m_out << best_cost << endl;
```

```
    }
  }
  auto ttend = chrono::high_resolution_clock::now();

  cout << "Time: "
    << chrono::duration_cast<chrono::microseconds>(ttend - ttini).count() << '\n';
  log_out << "Time: "
    << chrono::duration_cast<chrono::microseconds>(ttend - ttini).count() << '\n';

  c_out.close();
  i_out.close();
  m_out.close();
  log_out.close();

  return 0;
}
```

References

[1] Dirk Biskup and Martin Feldmann. Benchmarks for scheduling on a single machine against restrictive and unrestrictive common due dates. *Computers and Operations Research*, 28(8):787–801, 2001.

[2] T. C. E. Cheng and H. G. Kahlbacher. A proof for the longest-job-first policy in one-machine scheduling. *Naval Research Logistics (NRL)*, 38(5):715–720, 1991.