Implementation for the Project Scheduling Problem – Post GECCO Version

May 22, 2021

1 Algorithm

The implementation generally follows the report "Running Time Analysis for the Project Scheduling Problem" written by Dirk and was used in [1]. The present report outlines implementation details and the main differences between the implementation and Dirk's report.

1.1 Evolutionary Algorithm and Setup

Different algorithms can be run depending on the configuration used. Examples of configuration files are *OnePlusOneEA.sh*, *RLS.sh* and *GA.sh*.

For a (1+1) EA – *OnePlusOneEA.sh*:

- Modules / optimizer / EvolutionaryAlgorithm
  - generations = (e.g. 5064)
  - alpha = 1 (size of the population)
  - mu = 1 (number of parents per generation)
  - lambda = 1 (number of offspring per generation)
  - crossoverRate = 0
- Modules / optimizer / selector / ElitismSelector
- Modules / optimizer / operator / BasicMutate (basic mutation operators according to the genotype) – note that the integer mutation chooses a new value in U(lower bound, upper bound) without excluding the original value being mutated. As a consequence, the real mutationRate is actually mutationRate * k/(k+1) instead of mutationRate.
  - mutationRateType = CONSTANT
  - mutationRate = 1/(number of tasks * number of employees)
- Modules / default / random (choose the random seed)
- Modules / default / archive (choose population archive to keep the non-dominated individuals)
- Modules / problem / PSP (choose values)
- Modules / visualization / PSPLogger (log fitness, cost, duration, undt, reqsk for the archive individuals; choose e.g. log per 100 iteration step)

For an RLS – *RLS.sh*:

- Modules / optimizer / EvolutionaryAlgorithm
  - generations = (e.g. 5064)
  - alpha = 1 (size of the population)

  - mu = 1 (number of parents per generation)
  - lambda = 1 (number of offspring per generation)
  - crossoverRate = 0
- Modules / optimizer / selector / ElitismSelector
- Modules / optimizer / operator / PlusMutate (basic mutation operators according to the genotype)
  - integerMutateType = RLS
  - mutationRateType = CONSTANT
  - mutationRate = any value (this will be ignored by the RLS mutation)
- Modules / default / random (choose the random seed)
- Modules / default / archive (choose population archive to keep the non-dominated individuals)
- Modules / problem / PSP (choose values)
- Modules / visualization / PSPLogger (log fitness, cost, duration, undt, reqsk for the archive individuals; choose e.g. log per 100 iteration step)

For a (64+64) EA – *GA.sh*:

- Modules / optimizer / EvolutionaryAlgorithm
  - generations = (e.g. 79)
  - alpha = 64 (size of the population)
  - mu = 64 (number of parents per generation)
  - lambda = 64 (number of offspring per generation)
  - crossoverRate = (e.g. 0.75)
- Modules / optimizer / selector / Nsga2 (this will allow for tournament parents selection and can be used with one objective for an elitist approach)
  - tournament = 1 (this is actually binary tournament selection; 1 is the number of opponents *after* an initial individual is randomly selected for the tournament)
- Modules / optimizer / operator / BasicMutate (basic mutation operators according to the genotype) – note that the integer mutation chooses a new value in U(lower bound, upper bound) without excluding the original value being mutated. As a consequence, the real mutationRate is actually mutationRate * k/(k+1) instead of mutationRate.
  - mutationRateType = CONSTANT
  - mutationRate = 1/(number of tasks * number of employees)
- Modules / optimizer / operator / PlusCrossover
  - pspCrossoverType = MIXED_EMPLOYEE_TASK
  - integerType = RATE
  - integerRate = 0.0 (just to make sure that integer crossover type will not be applied)
- Modules / default / random (choose the random seed)
- Modules / default / archive (choose population archive to keep the non-dominated individuals)
- Modules / problem / PSP (choose values)
- Modules / visualization / PSPLogger (log fitness, cost, duration, undt, reqsk for the archive individuals; choose e.g. log per 100 fitness evaluations)

1.2 Command Line

The program can be run both with graphical interface or from the command line. In order to run with graphical interface, use org.opt4j.start.Opt4J as the main class. In order to run from the command line, use:

java -cp pspea.jar:opt4j-2.4.jar:junit.jar org.opt4j.start.Opt4JStarter <configFile.xml>

1.3 Skill Constraint Modes

The implementation allows us to choose between two skill constraint modes:

1. The mode explained in Dirk's report: an employee e assigned to a task t can only perform t if s/he has all the skills required by t:

$$\text{req}_t \subseteq \text{skill}_e. \tag{1}$$

2. [2]'s mode: the union of skills of all employees e assigned to a task t must contain all the skills necessary to perform t:

$$\text{req}_t \subseteq \bigcup_{\{e|phen_{e,t}>0\}} \text{skill}_e, \tag{2}$$

where $phen_{e,t}$ is the dedication of employee e to task t in the phenotype.

1.4 Genotype and Phenotype

The genotype *gen* is a vector of integers in $\{0, ..., k\}$. The phenotype *phen* is a matrix of employees' dedications (in $\{0/k, 1/k, ..., k/k\}$) to tasks. For decoding from genotype to phenotype, we divide each genotype integer value by k and assign it to the corresponding position in the phenotype matrix:

$$phen_{e,t} = \frac{gen_{e,t}}{k},$$

where $gen_{e,t}$ is the integer value for employee e to task t in the genotype.

If the skill constraint mode is 1, then there is an additional step to "fix" the phenotype in the following way. For each employee e and task t,

$$phen_{e,t} := \begin{cases} phen_{e,t} & \text{if } \text{req}_t \subseteq \text{skill}_e \\ 0 & \text{otherwise.} \end{cases}$$

This fixing step is **only** performed when the skill constraint mode is 1.

1.5 Mutation

If the basic mutation operation with constant mutation rate is chosen in the program's setup (section 1.1), each element of the genotype is mutated with probability defined by the mutation rate. The mutation takes a new integer value from $U(0, k)$. Note that the new value is taken from $U(0, k)$ without excluding the current value. So, there is a probability of 1/(k+1) that the new value is the same as the current value.

If the RLS mutation is chosen from the PlusMutation module, exactly one element of the genotype is mutation. This element is chosen uniformly at random and the mutation takes a new integer value from $U(0, k)\backslash\{a\}$, where a is the current value. The mutation rate should be set as constant mutation rate and is ignored.

1.6 Crossover

If pspCrossoverType = EMPLOYEE is chosen from PlusCrossover in the program's setup (section 1.1), for each pair of parents $gen1$ and $gen2$, with a defined probability of crossover, generate two offspring $gen3$ and $gen4$ as follows:

Algorithm 1 Crossover $(gen1, gen2)$

for each employee e **do**
  if $rand < 0.5$ **then**
    for each task t **do**
      $gen3_{e,t} \leftarrow gen1_{e,t}$
      $gen4_{e,t} \leftarrow gen2_{e,t}$
    end for
  else
    for each task t **do**
      $gen3_{e,t} \leftarrow gen2_{e,t}$
      $gen4_{e,t} \leftarrow gen1_{e,t}$
    end for
  end if
end for
Output $(gen3, gen4)$

If pspCrossoverType = TASK is chosen from PlusCrossover in the program's setup (section 1.1), for each pair of parents $gen1$ and $gen2$, with a defined probability of crossover, generate two offspring $gen3$ and $gen4$ as follows:

Algorithm 2 Crossover $(gen1, gen2)$

for each task t **do**
  if $rand < 0.5$ **then**
    for each employee e **do**
      $gen3_{e,t} \leftarrow gen1_{e,t}$
      $gen4_{e,t} \leftarrow gen2_{e,t}$
    end for
  else
    for each employee e **do**
      $gen3_{e,t} \leftarrow gen2_{e,t}$
      $gen4_{e,t} \leftarrow gen1_{e,t}$
    end for
  end if
end for
Output $(gen3, gen4)$

If pspCrossoverType = MIXED_EMPLOYEE_TASK is chosen from PlusCrossover in the program's setup (section 1.1), for each pair of parents $gen1$ and $gen2$, with a defined probability of crossover, generate two offspring $gen3$ and $gen4$ by randomly applying either crossover of the type EMPLOYEE (50% probability) or TASK (50% probability).

1.7 Cost and Duration

The algorithm to calculate cost and duration (completion time) for a certain phenotype is the same as algorithm 1 from Dirk's report **when the solution is feasible**. The following infeasibility cases are considered:

1. Problem instance is not solvable: $cost = -1$ and $duration = -1$.
2. Skill constraint mode is 1 and there are tasks to which equation 1 is not satisfied: cost and duration are calculated as if these tasks were instantly completed.

3. Skill constraint mode is 2 and there are tasks to which equation 2 is not satisfied: cost and duration are calculated as if these tasks were instantly completed.

The calculation of cost and duration for infeasible solutions is done in this way so that the fitness calculation can be easily extended to use these values if necessary.

1.8 Overwork

The implementation offers the option not to use normalisation. In that case, the fitness evaluation algorithm is different and also calculates the project's overwork. Algorithm 3 presents the algorithm (apart from the calculation of undt and reqsk), with the main differences in comparison to our original algorithm in red.

Algorithm 3 Evaluate(cost, completiontime, TPG)

while TPG $\neq \emptyset$ **do**
  Let V' be the set of all unfinished tasks without incoming edges in TPG.
  if $V' = \emptyset$ **then**
    Output "Problem instance is not solvable!" and stop.
  end if
  for all tasks t_j in V' **do**
    for all employees e_i **do**
      Let $d_{i,j} := \bar{x}_{i,j}$.
    end for
    Compute the total dedication $d_j := \sum_{i=1}^{n} d_{i,j}$.
  end for
  Let $t := \min_j(\text{eff}_j/d_j)$.
  Let cost := cost $+ t \sum_{i=1}^{n} s_i \sum_{j=1}^{m} d_{i,j}$.
  Let completiontime := completiontime $+ t$.
  for all employees e_i **do**
    if $\sum_{j=1}^{m} d_{i,j} > 1$ **then**
      $overwork := overwork + t \cdot (\sum_{j=1}^{m} d_{i,j} - 1)$
    end if
  end for
  for all tasks t_j in V' **do**
    Let $\text{eff}_j := \text{eff}_j - t \cdot d_j$.
    if $\text{eff}_j = 0$ **then**
      Mark t_j as finished and remove it and its incident edges from TPG.
    end if
  end for
end while
Output (cost, completiontime, $overwork$) and stop.

1.9 Fitness Calculation

Five different cases are considered for the fitness calculation:

1. Feasible solution:

$$fitness = wCost * cost + wDuration * duration,$$

   where $wCost$ and $wDuration$ are pre-defined weights, and $cost$ and $duration$ are calculated as explained in section 1.8. Default values from [2]'s work are $wCost = 1E-6$ and $wDuration = 0.1$.

2. The solution is infeasible because the problem instance is not solvable:

$$fitness = -1.$$

3. Skill constraint mode is 1 and there are tasks to which equation 1 is not satisfied:

$$fitness = wPessimistic * undt,$$

$$wPessimistic = 2 * (wCost * pessimisticCost + wDur * pessimisticDuration),$$

$$pessimisticDuration = k \sum_{t=1}^{T} \text{eff}_t,$$

$$pessimisticCost = \sum_{t=1}^{T} \sum_{e=1}^{E} s_e \text{eff}_t.$$

   where *undt* is the number of tasks that violate equation 1.

4. Skill constraint mode is 2 and there are tasks to which equation 2 is not satisfied:

$$fitness = wPessimistic * reqsk,$$

$$reqsk = \sum_{t=1}^{T} \left(|\text{req}_t| - \left| \text{req}_t \bigcap \bigcup_{\{e|phen_{e,t}>0\}} \text{skill}_e \right| \right).$$

5. We are not using normalization and there is overwork, but no violations to items 3 and 4:

$$fitness = wPessimistic + overwork,$$

   where overwork is the total amount of overwork calculated using algorithm 3.

It is worth to note the difference between the fitness calculation when the skill constraint mode is 1 and 2 and there are violations to equations 1 and 2, respectively. In the former case, only *undt* is considered, whereas *reqsk* is used in the latter case. The reason for that is that in the former case, the number of skills that are still necessary to perform an infeasible task is not relevant. Changing a solution to use an employee who contains more of the necessary skills for a certain infeasible task, but not all the necessary skills, would not improve the solution. However, in the latter case, such a change would make the solution closer to become a feasible solution. In this case, it is also worth noting that the use of *reqsk* already captures *undt*'s features, making the use of *undt* unnecessary.

1.10 Log File

The log file will save some information about the run of the algorithm. The number of iterations (generations) whose information is saved in the log file is specified by PSPLogger in the configuration file explained in Section 1.1. Each row of the log file contains the following information:

- Iteration – this is the number of the iteration whose information is being recorded.
- Evaluations – this is the number of fitness evaluations performed up to the point when the information is recorded.
- Runtime[s] – the runtime elapsed until now.
- COSTDUR[MIN] – the value of the fitness function for the best solution of this iteration.
- Cost – the cost of the project based on the best solution (with best fitness) of the population in this iteration.

- Duration – the duration of the project based on the best solution (with best fitness) of the population in this iteration.
- Undt – number of tasks for which each employee does not have *all* the skills required by the task. For example, if a given task t_j requires skills 0, 1 and 2, and at least one employee allocated to this task does not have all these three skills, this task will count as one of such tasks. **Only used with skills constraint option 1.** If the skills constraint is not option 1, this will always contain zero.
- Reqsk – sum of the number of skills missing by the team as a whole assigned to perform each task. **Only used with skills constraint option 2.** If the skills constraint is not option 2, this will always contain zero.
- Overwork –total amount of overwork that employees need to do.
- PhenotypeBeforeNormalization – dedication of employees to tasks. This is printed based on the code below, where n is the number of employees, m is the number of tasks and *allocation* is the matrix of dedications of employees by tasks.

```
for (e=0; e<n; ++e)
      for (t=0; t<m; ++t)
            print(allocation[e][t] + ",");
```

References

[1] L.L. Minku, D. Sudholt, and X. Yao. Improved evolutionary algorithm design for the project scheduling problem based on runtime analysis. *IEEE Transactions on Software Engineering*, 40(1):83–102, 2014.

[2] Enrique Alba and J. Francisco Chicano. Software project management with GAs. *Information Sciences*, 177:2380–2401, 2007.