

Implementation for the Project Scheduling Problem – Post GECCO Version

November 13, 2012

1 Algorithm

The implementation generally follows the report “Running Time Analysis for the Project Scheduling Problem” written by Dirk. 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 $\text{mutationRate} * k/(k+1)$ instead of mutationRate.
 - mutationRateType = CONSTANT
 - $\text{mutationRate} = 1/(\text{number of tasks} * \text{number of employees})$
- 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 $\text{mutationRate} * k/(k+1)$ instead of mutationRate.
 - mutationRateType = CONSTANT
 - mutationRate = $1/(\text{number of tasks} * \text{number of employees})$
- 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 :

$$\text{req}_t \subseteq \text{skill}_e. \quad (1)$$

2. Alba and Chicano (2007)’s mode: the union of skills of all employees e assigned to a task t must contain all the skills necessary to perform t :

$$\text{req}_t \subseteq \bigcup_{\{e | \text{phen}_{e,t} > 0\}} \text{skill}_e, \quad (2)$$

where $\text{phen}_{e,t}$ 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 $\{0, \dots, k\}$. The phenotype $phen$ is a matrix of employees’ dedications (in $\{0/k, 1/k, \dots, k/k\}$) 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:

$$\text{phen}_{e,t} = \frac{\text{gen}_{e,t}}{k},$$

where $\text{gen}_{e,t}$ 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 ,

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

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 $U(0, k) \setminus \{a\}$, 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*)

```

1: for each employee e do
2:   if rand < 0.5 then
3:     for each task t do
4:       gen3e,t ← gen1e,t
5:       gen4e,t ← gen2e,t
6:     end for
7:   else
8:     for each task t do
9:       gen3e,t ← gen2e,t
10:      gen4e,t ← gen1e,t
11:    end for
12:   end if
13: end for
14: 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*)

```

1: for each task t do
2:   if rand < 0.5 then
3:     for each employee e do
4:       gen3e,t ← gen1e,t
5:       gen4e,t ← gen2e,t
6:     end for
7:   else
8:     for each employee e do
9:       gen3e,t ← gen2e,t
10:      gen4e,t ← gen1e,t
11:    end for
12:   end if
13: end for
14: 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 2 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)

```

1: while TPG  $\neq \emptyset$  do
2:   Let  $V'$  be the set of all unfinished tasks without incoming edges in TPG.
3:   if  $V' = \emptyset$  then
4:     Output "Problem instance is not solvable!" and stop.
5:   end if
6:   for all tasks  $t_j$  in  $V'$  do
7:     for all employees  $e_i$  do
8:       Let  $d_{i,j} := \bar{x}_{i,j}$ .
9:     end for
10:    Compute the total dedication  $d_j := \sum_{i=1}^n d_{i,j}$ .
11:  end for
12:  Let  $t := \min_j (\text{eff}_j / d_j)$ .
13:  Let  $\text{cost} := \text{cost} + t \sum_{i=1}^n s_i \sum_{j=1}^m d_{i,j}$ .
14:  Let  $\text{completiontime} := \text{completiontime} + t$ .
15:  for all employees  $e_i$  do
16:    if  $\sum_{j=1}^m d_{i,j} > 1$  then
17:       $\text{overwork} := \text{overwork} + t \cdot (\sum_{j=1}^m d_{i,j} - 1)$ 
18:    end if
19:  end for
20:  for all tasks  $t_j$  in  $V'$  do
21:    Let  $\text{eff}_j := \text{eff}_j - t \cdot d_j$ .
22:    if  $\text{eff}_j = 0$  then
23:      Mark  $t_j$  as finished and remove it and its incident edges from TPG.
24:    end if
25:  end for
26: end while
27: Output (cost, completiontime, overwork) and stop.

```

1.9 Fitness Calculation

Five different cases are considered for the fitness calculation:

1. Feasible solution:

$$\text{fitness} = w\text{Cost} * \text{cost} + w\text{Duration} * \text{duration},$$

where $w\text{Cost}$ and $w\text{Duration}$ are pre-defined weights, and cost and duration are calculated as explained in section 1.7. Default values from Alba and Chicano (2007)'s work are $w\text{Cost} = 1E - 6$ and $w\text{Duration} = 0.1$.

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

$$\text{fitness} = -1.$$

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

$$fitness = wPessimistic * undt,$$

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

$$pessimisticDuration = k \sum_{t=1}^T eff_t,$$

$$pessimisticCost = \sum_{t=1}^T \sum_{e=1}^E s_e eff_t.$$

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:

$$fitness = wPessimistic * reqsk,$$

$$reqsk = \sum_{t=1}^T \left(|req_t| - \left| req_t \cap \bigcup_{\{e | phen_{e,t} > 0\}} skill_e \right| \right).$$

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 2.

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.

2 Experiments with Pop-EA – Convergence Plots

Obs.: the plots show the average $1/fitness$, instead of average fitness, in order to facilitate visualization and comparison against the results published by Alba and Chicano. As our algorithms with normalisation always attained hit rate of 100%, the fitness is comparable to the GA's fitness. The fitness of the (1+1) EA without normalisation is not comparable to the GA's when there are infeasible solutions, as the fitness of the infeasible solutions is calculated differently.

General comment: PopEA frequently starts off with solutions of considerably good fitness, whereas the algorithms without normalisation typically start with solutions of very bad fitness ($1/fitness$ close to zero). This is because normalisation eases the overwork constraint, allowing for feasible solutions to be found very quickly, whereas the algorithms without normalisation need more time to find feasible solutions.

General comment for benchmarks 1–3: The graphs show a quick convergence to near optimal solutions within 20 generations (1280 fitness evaluations) of the total of 79 generations for all the problem instances. We can see that all these instances were easy for PopEA to solve and find near optimum solutions, whereas

the algorithms without normalisation struggled and were in general more affected by different problem features.

Detailed comment for benchmarks 1–3: The fact that the gradient of the curves was in general similar for different problem instances reinforce our previous analysis (in terms of final solution quality) that shows that PopEA is less affected by changes in the number of employees, tasks and employee’s skills than the algorithms without normalisation. In benchmark2, tasks30, the gradient was slightly different from tasks10 and 20 in the sense that an almost optimum solution was found very quickly. So, almost no improvement is seen across generations. This is a somewhat surprising behaviour, as one would expect instances with more tasks to be more difficult. As the difference between the final and optimum solution quality from 10 to 20 and 30 tasks does not decrease monotonically, the differences in difficulty of these instances may be due to random factors involved in the instances generation, as the instances of this benchmark represent different projects with different TPGs. So, a different number of tasks affects convergence more than the number of employees and employee’s skills, but we cannot identify patterns of difficulty of instances due to the randomness in the generation of the instances. In benchmark3, skills2, the initial solutions were in general worse than for higher numbers of skills. However, PopEA still took similar time to convergence.

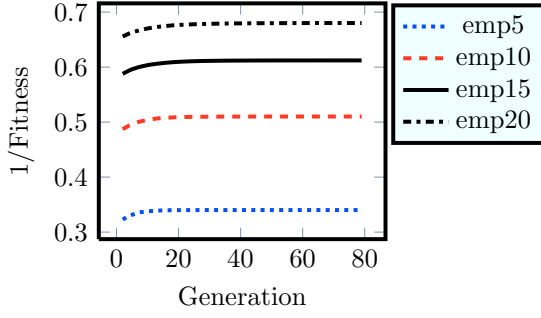
Obs.: As expected, a higher number of employees allowed for a better fitness to be achieved by PopEA. As the salary of all employees is the same and the skills constraint is ignored in benchmark 1, a higher number of employees allows for the optimal duration of the project to be shorter. Also as expected, a higher number of tasks is associated to worse PopEA fitness. As more tasks need to be performed, the optimal duration of the project was longer. The number of employee’s skills did not affect the final fitness achieved by the PopEA.

General comment for benchmarks 4–5: The graphs show a quick convergence, specially for instances with 20 and 30 tasks. The convergence was a bit slower for 10 tasks, but PopEA still managed to find solutions with much better fitness than the algorithms without normalisation.

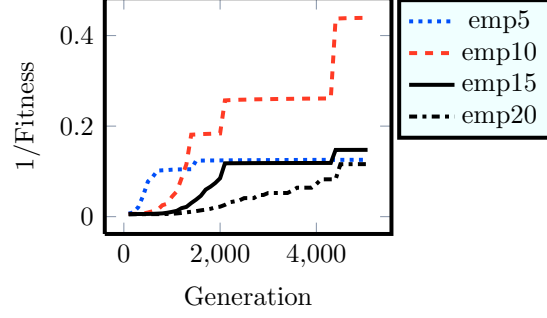
Detailed comment for benchmarks 4–5: we do not know what the best solution is and each problem instance corresponds to a different project. However, we can see that the behaviour of the PopEA’s fitness curves for these benchmarks was similar to the ones in benchmarks 1-3. For instance, a higher number of employees usually allowed for better fitnesses to be achieved, even though that does not seem to affect much the fast convergence of the algorithm. A higher number of tasks was also related to lower fitness, as expected. The problem instances with less tasks (10) surprisingly presented longer convergence time. The curves are also in general very flat, especially for the instances with 20 and 30 tasks. This might be seen as an indication that the algorithm is struggling to improve the solutions. However, considering the similarities of these results to the ones from benchmarks 1-3, in which the best solution is known, it is probable that the curves are flat because the solutions found are already near optimal. Moreover, the fitness obtained by the PopEA are much better than the ones obtained by the algorithms without normalisation.

We can see that our PopEA behaved fairly similarly in terms of its fast convergence time for different numbers of employee’s skills and project skills. A small number of tasks seems to have affected the convergence when varying the number of employee’s skills a bit more, even though it is not possible to distinguish any tendency in behaviour, which may be related to other random factors involved in the generation of the instances. A higher number of project skills associated to a small number of employees was in general associated to lower initial fitness. This is reasonable, as one would expect such instances to be more difficult. And yet, the time for convergence was not affected by that.

Summary: The plots show that PopEA frequently presents solutions with good fitness from the beginning, whereas (1+1) EA and GA without normalisation start off with much worse fitness (infeasible solutions). Benchmarks 1-3 show that PopEA manages to converge to near optimal solutions very quickly. Benchmarks 3-4 show that convergence is also achieved fast for these more difficult instances, even though we do not know what the best solutions are in this case. All the results show that PopEA is not much affected by different numbers of employees, employee’s skills and project skills, and attains solutions with much better fitness than the algorithms without normalisation. PopEA seems a bit affected by the number of tasks, which is reasonable considering that changing the number of tasks would considerably change the project. Nevertheless, even when varying the number of tasks, PopEA’s final solution quality was still much better than the algorithms without normalisation.

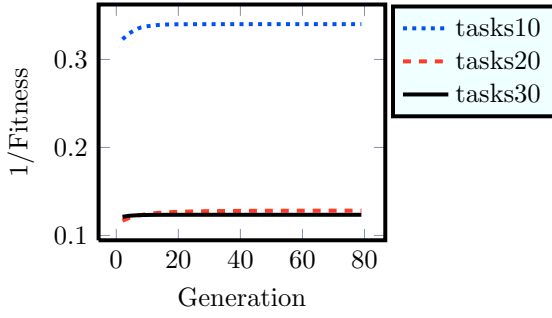


(a) PopEA

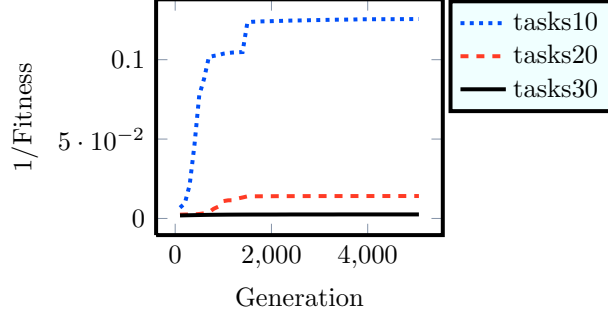


(b) (1+1) EA nonorm

Figure 1: Average best fitness evolution for benchmark 1.

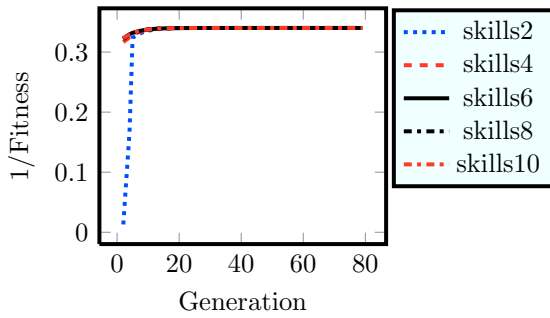


(a) PopEA

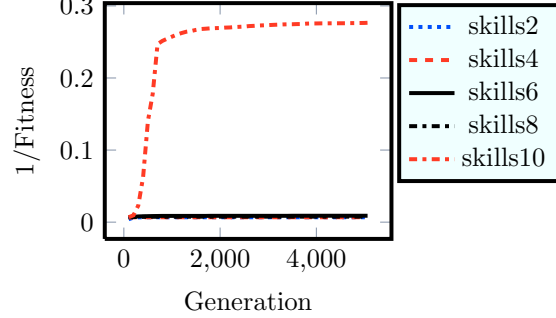


(b) (1+1) EA nonorm

Figure 2: Average best fitness evolution for benchmark 2.



(a) PopEA



(b) (1+1) EA nonorm

Figure 3: Average best fitness evolution for benchmark 3.

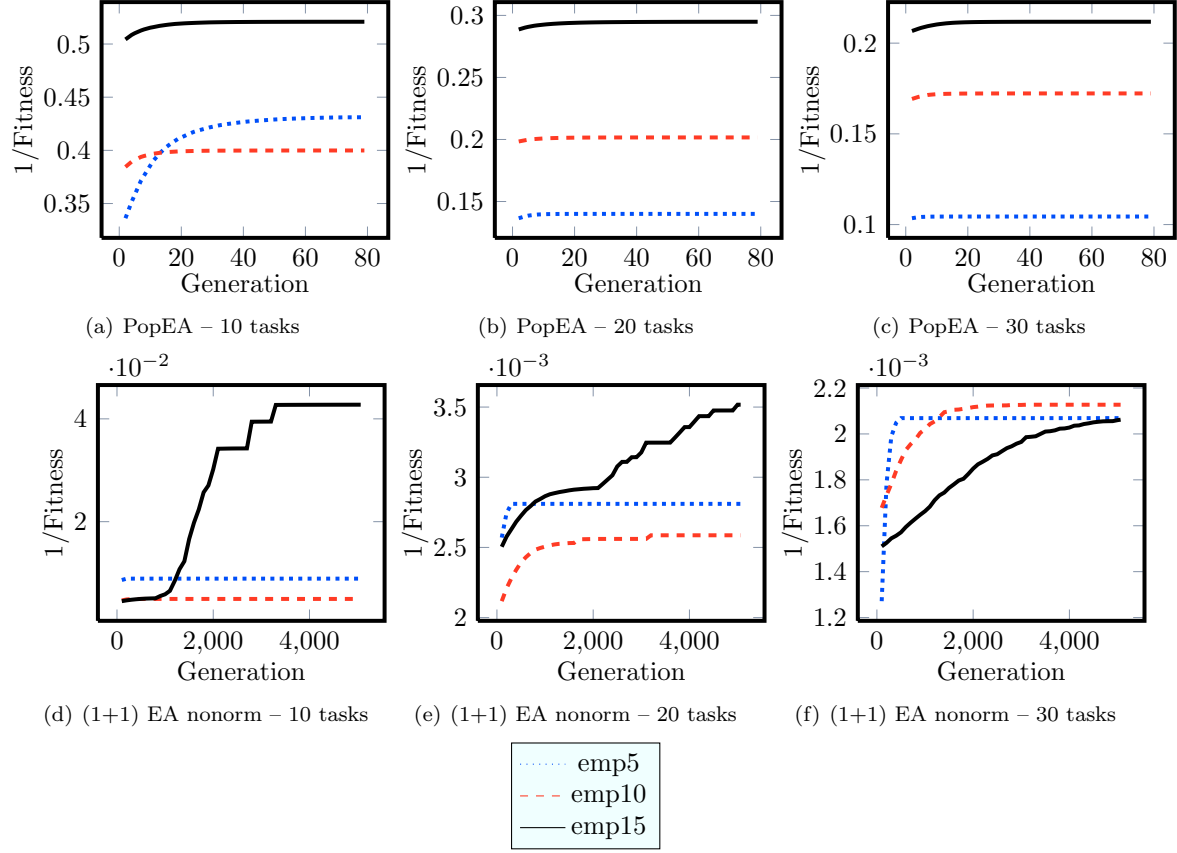


Figure 4: Average best fitness evolution for benchmark 4, 4–5 employee skills.

References

E. Alba and J. F. Chicano. Software project management with GAs. *Information Sciences*, 177:2380–2401, 2007.

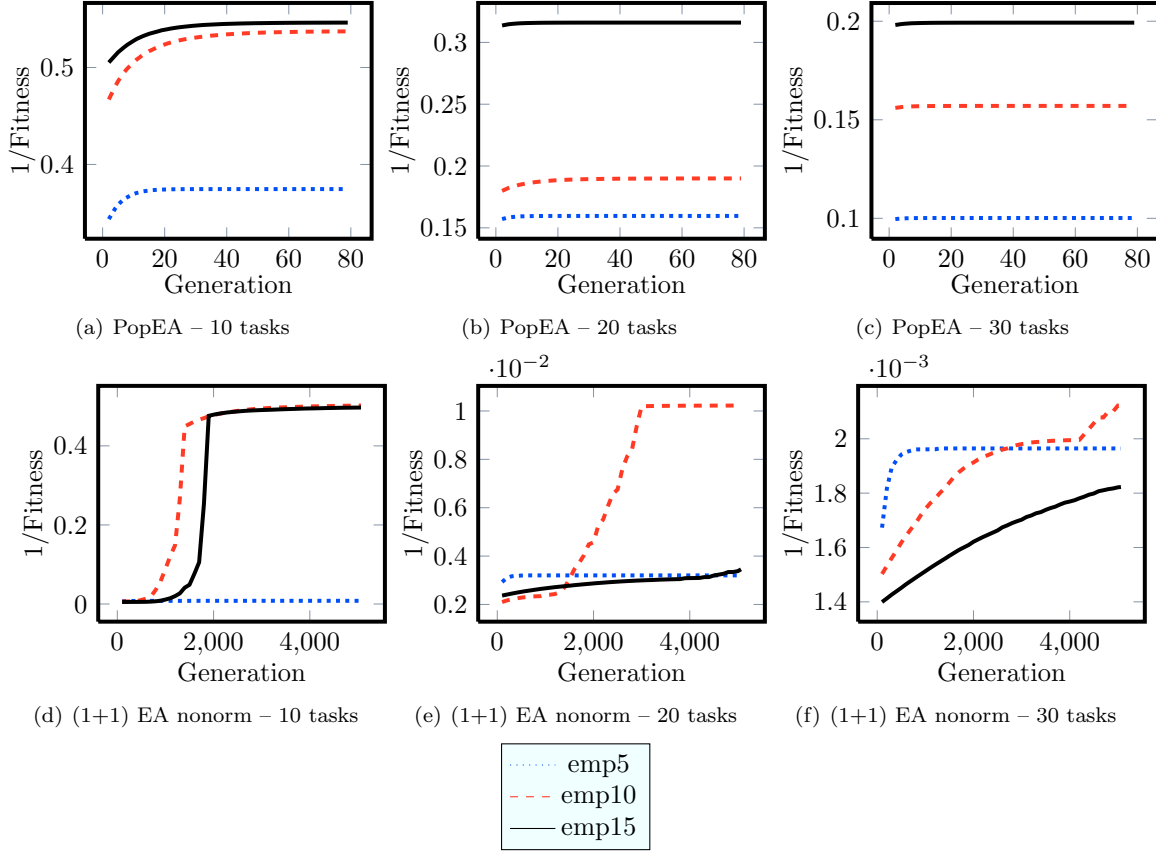


Figure 5: Average best fitness evolution for benchmark 4, 6-7 employee skills.

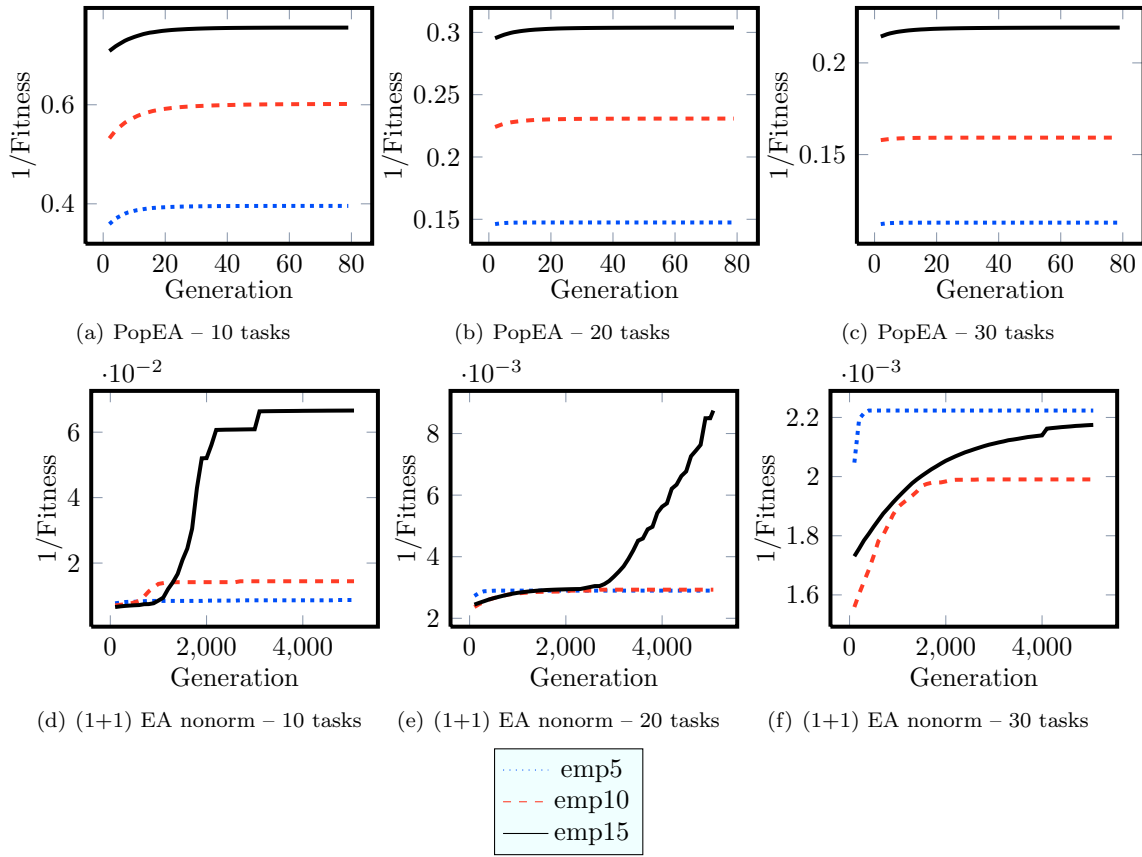


Figure 6: Average best fitness evolution for benchmark 5, 5 project skills.

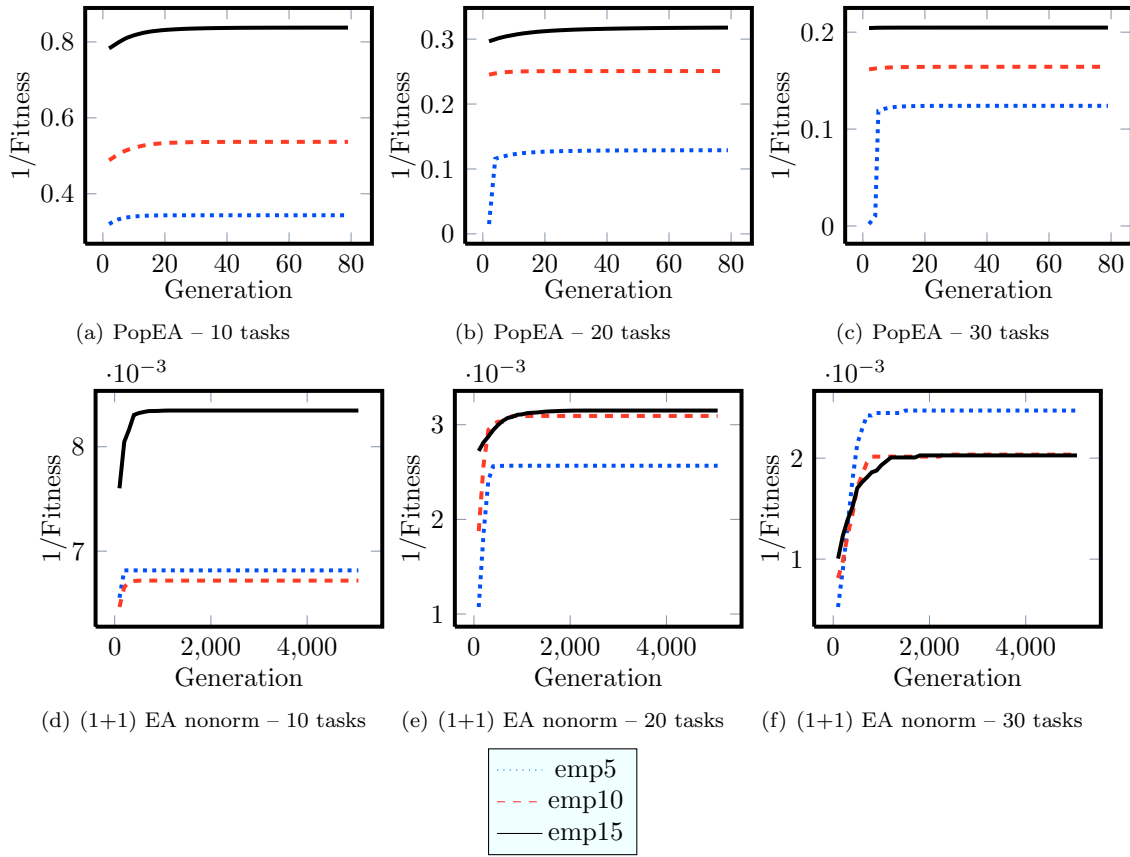


Figure 7: Average best fitness evolution for benchmark 5, 10 project skills.