A comparative analysis of a genetic algorithm using priority rules and particle swarm optimization for the resource-constrained project scheduling problem

Evolutionary Computation
HIOF
ITI41222

Vemund Brynjulfsen

Table of Contents

Abstract

I will compare the two algorithms Particle Sawm Optimization (PSO) and a Genetic Algorithm (GA) for solving the Resource Constrained Project Scheduling Problem (RCPSP).

The GA uses priority rules, while the PSO dosen't. So it is intresting to look at how an algorithm using priority rules will compare to one that dosen't.

Introduction

RCPSP was introduced by (Kelley, 1961).

In RCPSP, there is a system with some tasks. The goal is to schedule these tasks optimally. The tasks need to respect predecense constraints, meaning that a task can't start before it's predecesors. The tasks have resource demands. There is a limited resource avalibility in the system, which can result in bottlenecks, if some task can't be scheduled at a point because of insuffecient resources in the system.

It's relevant for all kinds of operations that need scheduling. For example machines that schedule processes before executing them. Or for human-resource problems, where some tasks have to be done in a certain order by a team.

RCPSP was proven to be a NP-hard problem in (Blazewicz, Lenstra, & Kan, 1983). If the resource constraint is removed, and only the precedence constrained is maintained, then the problem is like topological sorting, which is solved in polynomial time. Because of the resource constraint, there are also bottlenecks which can be caused by insufficient resource avalibility at some scheduling point.

This article will go over the following: The problem is defined beneath in the **problem definition.** The two algorithms are explained in detail in **Method**. The results and running setup is shown the **Results**. Lastly the results are discusssed in **Discussion**.

Problem definition

RCPSP can be defined as the following. It has a R set of renewable resources. Each renewable resource $k \, c \, R$, has a avalibility of $a_k$at in all periodes.

The project has a set of N activity-nodes, from 0 to n+1. Each activity-node i has a duration of d_i which can't be interrupted. At each periode, it requires $r_{i,k}$ units of resource type $k \, c \, R$. Each activity-node has a set of predessecors P. A task can't be schduled before all it's predessecors P_i have finished.

The project is represented topologically as ordered an actvity-on-the-node (AoN) format, with an acyclic graph.

An activity can’t be scheduled at some point, if the total resource demand of all scheduled activities at that point will exeed the resource avalibility.

The objective is to find find a schedule which has the lowest makespan.

The input model for an algorithm that seeks to solve RCPSP will be a activity-network. The algorithm will then generate a priority list of the tasks, prioritizing which tasks will be scheduled before other tasks. The priority list will be used by a generation scheme to make a feasble schdule. The output is the generated schedule.

Example of an activity-network. Each node represents an activity. Each directed edge points to a successor.

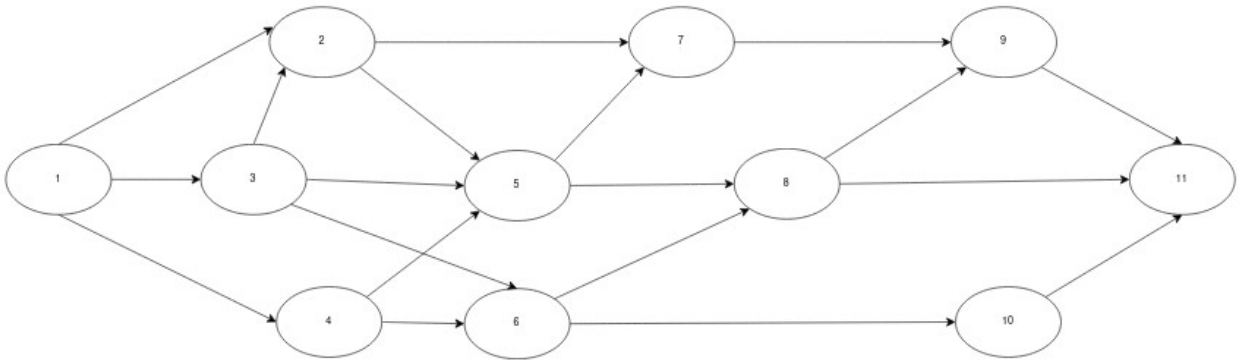


Figure 1: Example of an activity-network

Example of a schedule. Each activity has a resource utilization, and a duration.

(Chand, Rajesh, & Chandra, 2022, p. 2)

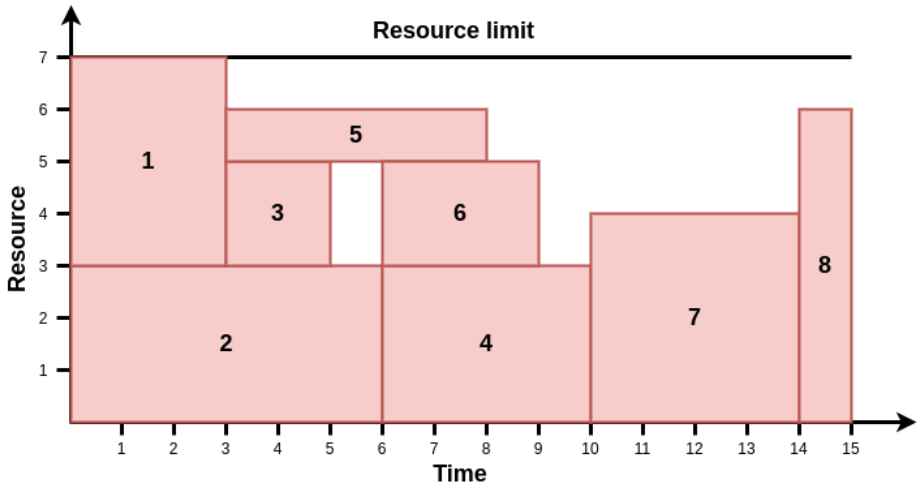


Figure 2: Example of a schedule

Method

Objective

The objective of this paper is to compare two implementations of GA and PSO for RCPSP.
This will be done by comparing them on different metrics and during their runtime.

Generation scheme

This paper uses serial schedule generation scheme (SSGS). SSGS is used to make a feaseble schedule from a priority list. A priority list is passed to SSGS, and then SSGS makes a schedule which respects predessecor and resource constrains.

Genetic algorithm

About algorithm

The algorithm utelizes an expression-tree to desisgn priority rules for scheduling. The function nodes are priority rules, which are combined through arithmetic nodes. The result will be a mathematical expression that is applied to each task, to assign priorites to the tasks.

Priority rules

This paper will solve RCPSP by an approach of generating priority rules.

A solution will consist of activity attributes, which makes up a priority rule. A task will recieve points for different attributes it has, which will be used to rank the tasks.

The attributes can tell different information about a task. Things like task's position in the network, it's resource needs, it's runtime etc. A priority can be applied to a task, and by doing so, a value can be extracted from the task, which can be used as a priority.

These rules are used for the GA

- TSC/TPC: The total count of the predecessors of an activity.
- ES/EF: The earliest start/finish time for an activity in the precedence feasible schedule, which is calculated by relaxing the resource constraints, where each activity is scheduled as early as possible.
- LS/LF: The latest start/finish time for an activity in the precedence feasible schedule, calculated by relaxing the resource constraints, where each activity is scheduled as late as possible.
- RR: The total count of resources required by an activity.
- AvgRReq: The average resource requirement of an activity.
- MaxRReq: The maximum resource requirment of an activity.
- MinRReq: The minimum resource requirment of an activity.

Their calculation and normalization is given by (Chand, Huynh, Singh, Ray, & Wagner, 2018).

Arithmetic operators

The arithmetics operators are standard mathematical functions that takes two arguments and performs a calulation.
The following arithmetic operators will be used

- Add(a, b): a + b
- Mul(a. b): a * b
- Sub(a, b): a – b
- Div(a, b): $\frac{a}{b}\ if\ b>0\ otherwise\ 0$
- Max(a, b): $a\ if\ a>b\ otherwise\ b$
- Min(a,b)*:* $a\ if\ a<b\ otherwise\ b$
- Neg1(a): $-1\cdot a$

Representation

A priority rule will be output by a priority function, wich will calculate priorities for different activities.
So the representation of priroity rules will be a priority function for calculating priority values.

The priority function will consist of mathematical operators and task attributes. This paper will use a binary tree as representation for a priority function, where the function set will consist of mathemtical operators, and the terminal set will consist of activity attributes.
To ensure that the attributes are on the same scale across different categories, the points from each of the attributes are first normelized. This is nessecary because they deal with different attributes that has to do with things related to the task, which can result in widely different outputs.

The expression-tree will be applied to each task. When the tree is traversed using in-order traversal, the expression-tree will be decoded into a mathematical expression, which will give each task a priority value. A schedual is made using these priority values, and then the fitness of the priority function is based off of the makespan the generated schedule

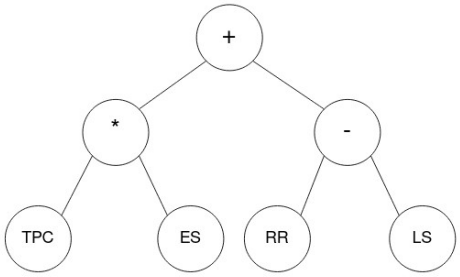


Figure 3: Example of a GA individual

Fitness

The output of the algorithm will be an optemized schedule for a project. The fitness will be the difference between the generated schedule's makespan, and the lower bound makespan for the project.

Parent selection

In the selection process, individuals are in some way weighted by their fitness, to be chosen to create offspring.

Selection is done by tournament selection. A number of individuals are chosen at random from the population. The tournament size equals the number of individuals chosen.
The individual with the best fitness is chosen.
The selection continues until the amount of selected individuals equals the initial population size.

Crossover

In the corssover process, the individuals selected in the selection process, create offspring by taking parts from each parent, and combining them to form new offspring.

A parent is chosen, then some crossover point is selected in the tree. The subtree from that point is swapped with a subtree in the other parent. This produces a child. Then this is done again to produce a 2nd child. Each child is like it's 1st parent, but different in that some subtree is swapped from the 2nd parent.

Context-perserved crossover is used. In this crossover, the crossover point is at the same position from the root node in both parents.

(Luo et al., 2022, Fig. 6)

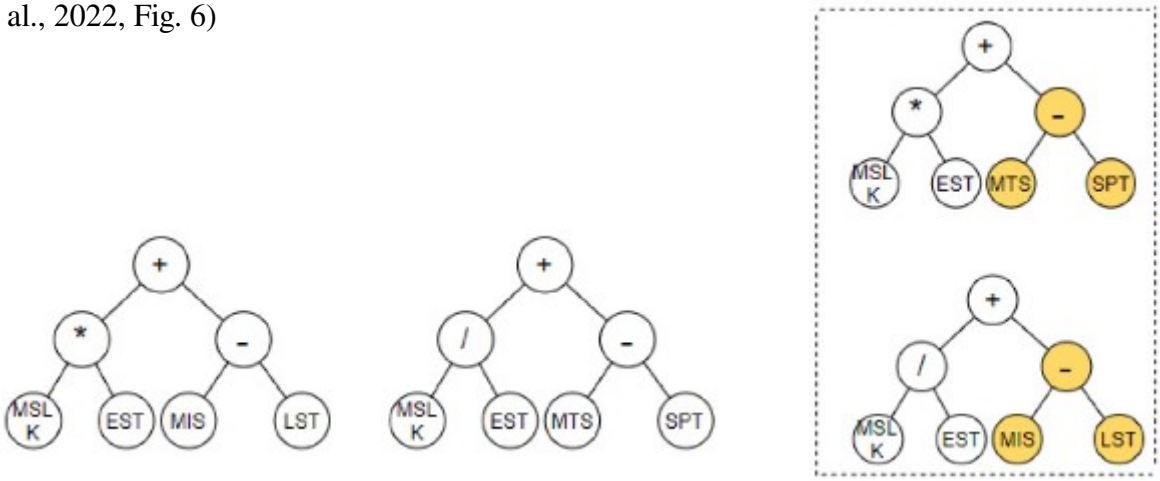


Figure 4: Crossover

Mutation

Mutation introduces randomness into the population, which might expand the search space for a solution. After the crossover process, some point is chosen in both offspring, and then changed to some other randomly generated subtree.

The mutation point is chosen at random, and can be both at a function node or terminal node. In this paper, point mutation is used.

(Luo et al., 2022, Fig. 7)

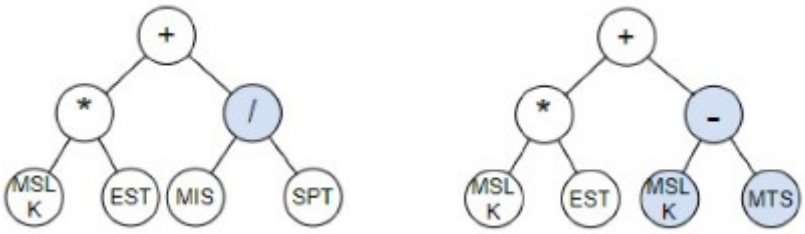


Figure 5: Mutation

Survivor selection

Tournament selection is used. Some members of the population are randomly picked, and then the fittest are carried over to the next generation.

Algorithm overview

Below is an overview of the algorithm.
At the start, a initial population of priority rules are generated.
Then the fitness of the priority rules are evaluated. The fitness is the runtime of scheduals in a training set. Then the fitness is used to determine which indiviuals will be favoured in selection. Then cross-over and mutation will be applied. This will lead to a new generation, and the cycle continues all over again.
The schedule made by the fittest individual of the final population will represent the run.
The algorithm goes as the following:

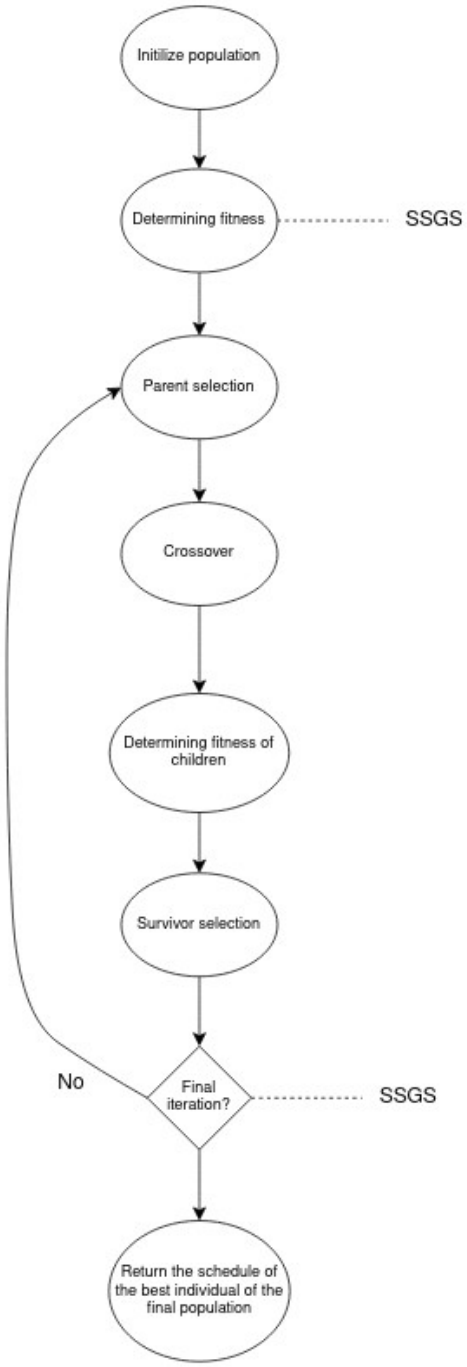


Figure 6: GA overview

Parameters

These parameters will be used:

The iteration amount T

Population size N

The crossover rate c_p

The mutation rate m_p

The max deptm $depth_{max}$

The min depth $depth_{min}$

Summary

Representation	Binary tree
Recombination	Context-perserved crossover
Mutation	Point mutation
Parent selection	Tournament selection
Survivor selection	Fitness based selection

Particle swarm optimization

About algorithm

The Particle swarm optimization (PSO) was first proposed by (Kennedy & Eberhart, 1995).
The idea is that there's a swarm of particles (individuals). The particles have a position, and a velocity towards another position. The goal is to move the particles to the optimal position (optimal solution).

The position of a particle is randomly initilized. The value of the velocity depends on the best recorded global position in the sawm, and the best recorded position the particle itself has had. A random factor also influences the velocity.

(Singh et al., 2022, Fig. 6)

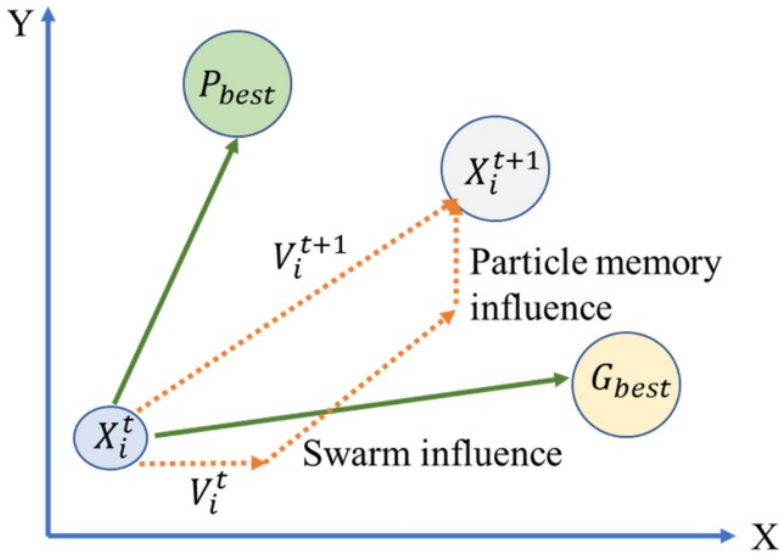


Figure 7: Illustration of PSO

Representation

The particle position and velocity is represented by a vector. The position elements are values between the range of [0,1], while the velocity elements are positions in the range of [-1,1].

The values are clamped to be in their perimtted ranges. If the position is outside of the it's permitted range, it's clamped to it's max/min value of the range.

The position elements represents a priority related to a task. Each task has to have a unique priority value. Because of this, each position element has be to unique. If several positions are the same, then they are randomized within a range.

Movment

The particles move in a N dimensional space. In this case, the amount of dimensions are the amount of tasks in a project.

The particles move according to a global best recorded position G, and it's own best recorded position lbest.

The position is annotated as $X_i=[X_{i1},\dots,X_{iN}]$, where X_{ij} is the *j*th element in the position.

Each particle i has a velocity of $V_i=[V_{i1},\dots,V_{iN}]$. The velocity will be incluended by the particle's best recorded position $L_i=[L_{i1},\dots,L_{iN}]$, and some recorded best global position from the swarm $G=[G_1,\dots,G_N]$.

A particle's velocity is updated according to the following formula:

$$V_{ij}^{new}=w\cdot V_{ij}+c_1\cdot r_1\cdot(L_{ij}-X_{ij})+c_2\cdot r_2\cdot(G_j-X_{ij})$$

A particle's position is updated according to the following formula:

$$X_{ij}^{new}=X_{ij}+V_{ij}^{new}$$

w is a inertia weight used to determine the influence of the particle's previous velocity. c_1 and c_2 are learning factors used to influence how much the global best position, and it's own best position will incluence the velocity. r_1 and r_2 are random factors in the range of [0,1].

Topology

The best recorded position from either the particles neighbourhood, or among the swarm is called gbest. The particle's own best recorded position is called lbest.

The particles need to communicate to determine which particle has the best recorded global position gbest. A particle dosen't nessecarly need to have the a best recorded position of the whole swarm. Using that position, might lead to a too early convergence. Instead it can limit gbest to come from it's neighbourhood. The advantage of this is a slower convergence, which might lead to a better solution.

There are different typologies that can be used to incluence where gbest comes from.

A star-topology draws the gbest from the whole swarm, while a ring-topology draws the gbest position from one of the particle's neighbours.

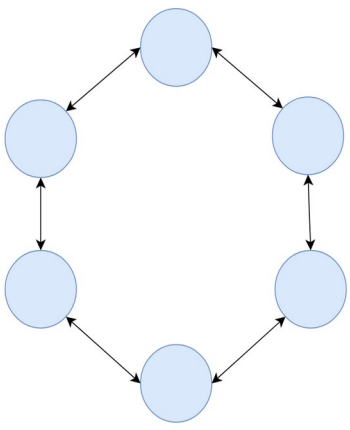


Figure 8: Ring-topology

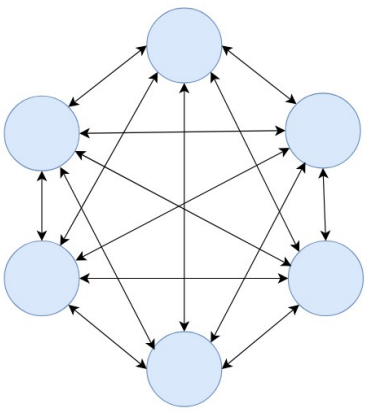


Figure 9: Star-topology

This algorithm includes a probability parameter *GR* for determining which topology to use at a *i*th iteration. A random value is generated to compare against the *GR* parameter. If the *GR* value is higher than the randomly generated value in the range of [0,1], then the star-topology is used, otherwise the ring-topology is used.

Algorithm overview

The particles in the swarm are initilized at the start. Then the initial fitness is determined, by applying the positions to the tasks to extract a priority, and then SSGS is used to create a schedule with a makespan, which will be used for the fitness.

The topology is determined by the GR parameter. Then the velocity and position is calculated.

After having determined the fitness of all particles, the global best position G is potentially updated (if a better position is found), and lbest (if the particle found a better position)

After all the iterations are done, schedule generated by the best global position G is returned.

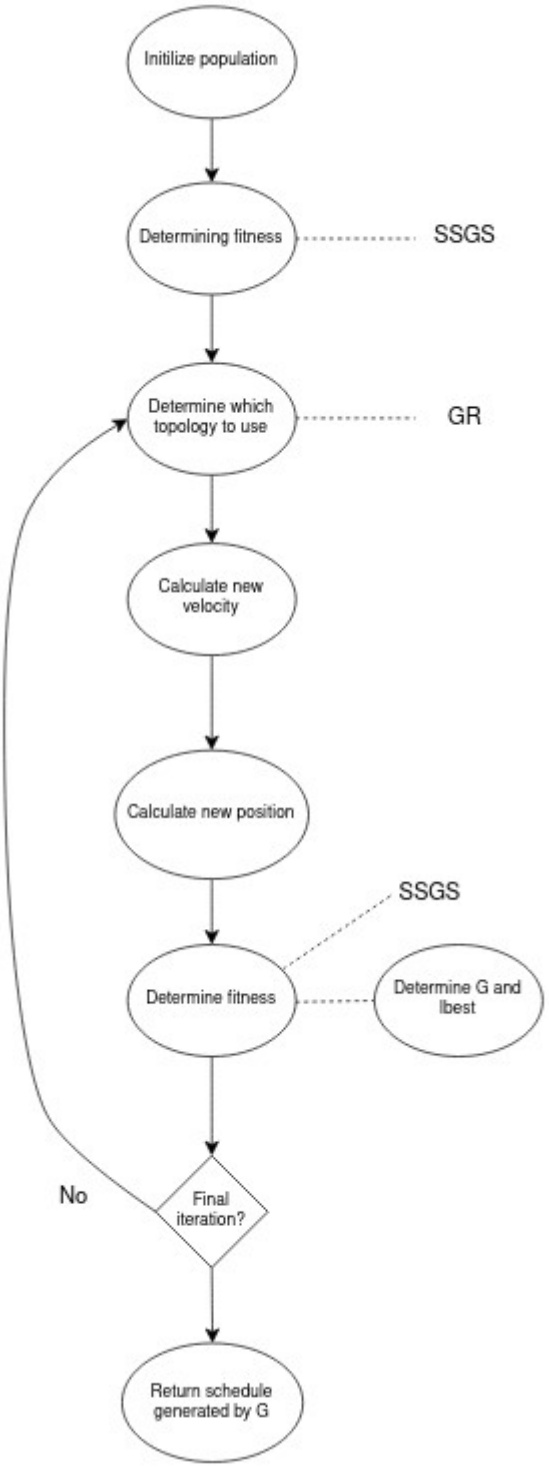


Figure 10: PSO overview

Parameters

Intertia weight w

learning factor lr_1

learning factor lr_2

random factor r_1

random factor r_2

star-topology probability GR

population size N

amount of iterations T

Results

Tools and data

Java was used to implement the two algorithms.

PSLIB's library is used for data. PSLIB was generated by (Kolisch & Sprecher, 1997). A software ProGen was made that generated artificial scheduling problems. This dataset is a common benchmark for solving RCPSP.

The datasets J30, J60, J90 and J120 was used.

Each dataset has instances of projects. Each project has a set amount of resources, tasks, and the tasks have predecessors. The lower bound is also known for these projects. In these projects, the first and last tasks are dummy tasks with zero resource requirments and duration. J30 has 30 non-dummy activites, J60 has 60 non-dummy activities etc.

The algorithms have quite a large running time, so it's difficuilt to get results from a lot of projects, especially as the task amount becomes larger. There's quite a difference in running time when using projects from the J30 set and the J120 set.

The results will be limited to the following projects:

Dataset	Parameter	Instance	Project File
J30	11	10	j3011_10.sm
J30	18	7	j3018_7.sm
J30	25	2	j3025_2.sm
J30	38	5	j3038_5.sm
J30	5	3	j305_3.sm
J60	11	10	j6011_10.sm
J60	18	7	j6018_7.sm
J60	25	2	j6025_2.sm
J60	38	5	j6038_5.sm
J60	5	3	j605_3.sm
J90	25	2	j9025_2.sm
J90	38	5	j9038_5.sm
J120	25	2	j12025_2.sm
J120	38	5	j12038_5.sm

Parameter optimization

The projects come with various amounts of tasks. Projects that have the same number of tasks, will use the same parameters. The parameters are in large part to increase or decrease the search space, which is more relevant depending on the amount of tasks in the project. A project with more tasks, have more possible solutions, and so a larger search might space is nessecary.

Grid-search is used to optemize the parameters for the algorithms. The grid-search is applied to a the projects stated **Tools and data**.

These parameters were found to be the best:

PSO

	J30	J60	J90	J120
w	1	0.5	0.5	0.5
lr_1	0.5	1	1	0.5
lr_2	0.5	1	1	0.5
r_1	0.5	0.3	0.5	1
r_2	0.5	0.3	0.5	1
GR	0.5	0.5	0.5	0.5

GA

c_p	0.3	0.7	0.3	0.3
m_p	0.3	0.7	0.3	0.3
$depth_{min}$	2	2	2	2
$depth_{max}$	5	8	8	8

The population is at 1000 for J30 and J60, and decreased to 500 in J90 and J120, as the running time becomes large. The iteration amount is 30 for all datasets.

Metrics

The running times of the algorithms will be looked at. This is intresting to look at, to see which algorithm has the largest runtime, and to later speculate on what might influence the algorithm runtimes.

Deviation is the difference between a generated schdule's makespan for a project, and the lower bound for a project. A lower deviation is better, as it means the schedule's makespan is closer to the lower bound.

The makespans of the algorithms will be compared over iterations, to see how the fitness of the individuals changes.

The GA uses priority rules. It will be looked into which priority rules were used the most.

Wins refers to the amount of projects where the generated schedule got the same makespan as the lower bound. The algorithms will also be compared on this metric.

The development of some particle's position towards the optimal position G will be looked at briefly.

Observations

Running times

The running times of the algortimgs in seconds:

Running time (sec)

Project file	PSO	GA
J305_3.sm	5	140
J3011_10.sm	6	142
J3018_7.sm	6	223
J3025_2.sm	6	375
J3038_5.sm	7	332
J605_3.sm	17	869
J6011_10.sm	15	564
J6018_7.sm	19	1767
J6025_2.sm	20	1994
J6038_5.sm	21	3475
J9025_2.sm	24	3718
J9038_5.sm	26	2557
J12025_2.sm	40	3646
J12038_5.sm	41	4247

GA has a far higher running time than PSO.

Makespan

The makespans can be seen in **Appendix: Makespans**. The makespan for the fittest individual in each iteration can be seen for both algorithms. There is also a table showing how far each schedule was from the lower bound. For most projects, the two algorithms manage to make the optimal schedule. In some cases the optimal schedule is found in the first iteration.

Mean values for makespans:

Running time (sec)

Project file	PSO	GA
J30	59.4	58.6
J60	88.6	88
J90	118.5	118
J120	113.5	113

The mean values are similair. However, both alrogithms managed to find the optimal solution a lot of times. If we just take into account the projects where the algorithms had an unqeual makespan, then the difference in the mean makespan looks a bit clearer:

Mean makespans of projects where the alorithms didn't have the same makespan.
Running time (sec)

Project file	PSO	GA
J30	72	70
J60	86	83
J90	146	145
J120	119	118

It seems that the GA at times has a slower convergence towards a optimal solution, but at the end it manages to find a better solution than the PSO.

Wins

Here it can be seen the amount of projects where the algorithms managed to find the optimum schedule.

GA scores the highest.

Wins

PSO	GA	Total amount of projects
8	9	14

Priority rules

Appendix: Priority rules and trees shows different rules that were used. Along with some examples of trees.

While **Appendix: Ratio of priority rules** shows the percentage usage of each priority rule in the fittiest individuals for different runs on the datasets.

It's intersting to see the usage of the different rules, but there dosen't seem be a pattern. One thing that can also make it hard to analyze, is that the influence of a certain priority rule dosen't just depend on the amount of the rule. It also depends on the arithmetic operators it's used with, and what other rules it's combined with during the arithmetic operations.

Particle movment

Appendix: Particle movment shows an examples of particle movment.

The histograms shows the values for the different elements in the position vector.
The final G best position is shown, and then a random particle's best recorded position lbest at different iterations.
I can't exaclty see a pattern. If one looks closley, then perhaps it might be seen that the histogram for the random particle becomes more similair to that of G.

Discussion

Difficulities in analyzing

One difficulity with analzying the algorithms is the running time. GA has a lot of running time, so it's difficuilt to run the algorithm on a lot of projects.

In addition there is some difficulity in analyzing the importance of different priority rules to a project, since different arithmetic operators will «weight» the function nodes differently. A function node that has multiplication applied to it, is going to have more influence in the priority calculation, than a function node that has addition applied to it.

Theoretical complexity

The GA had far higher running time than the PSO.
If we analyse the theoretical time complexity of the two algorithms, then it's not suprising.

Both algortihms running time can be analyzed thorugh only their fittnes evaluation time, as the fitness evaluation takes the most amount of running time.

The minimum time complexity of evaluating fitness for RCPSP

Both algoritms's fitness evaluation time depends on:

- The amount of tasks in the project **N**, as the datastrcutures representing a solution in each algorithm is applied over all tasks, in order to extract a priority for the task.
- The time taken to apply SSGS, which will be notated as S. After a priority list has been made, SSGS is used to make a schedule from the priority list. The shcedule can then be used to get a makespan for the solution.

So the theoretical time complexity of fittess evaluation for an algorithm for RCPSP, will at least be O(S+N)

The time complexity for PSO

it depends on:

- The amount of elements in the position vector, which is the same as the amount of tasks in the project **P.** This is because each element in a particle's position is applied once to only one task.

So the time taken to evaluate the fitness of a particle is O((N=P)+S)

The time complexity for GA

It depends on:

- The amount on nodes in the expression-tree **T.** This will depend on the specified parameters for the tree depth, but then there is also some randomness in that all branches won't have the same depth. So it's a bit uncertain exactly how many nodes are in a tree. An approximation can be made from the parameters for the tree depth.

  A binary tree with a height h will have 2^{h+1} amount of tasks. If $depth_{max}$ is the maximum specified height, and $depth_{min}$ is the minimum specified height, then the amount of nodes in the tree will be in the range of $2^{depth_{min}+1}$ to $2^{depth_{max}+1}$

  Each node in the tree is applied over all tasks.

- The time taken to apply a function node over a node. This also plays a role in why the running time as higher than in PSO, because time taken to apply a function node adds a lot of runtime to the GA.

  Different function nodes will vary in their complexity. Some will have to look throught the unfinished schedule, some will look through the activity-network to analyze it for different attributes, etc. There will likely be an uneven amount of different function node types. It might be that a tree only has function nodes with a higher time complexity.

  The function nodes are the leaf nodes of the expression-tree. The amount of leaf nodes will be in the range of $2^{depth_{min}}$ to $2^{depth_{max}}$.

  We can assume that there are F amount of function nodes in the expression tree, which will be a subset of all nodes N.

  We can call the aritmentic nodes for A, which will also be a subset of N.

The time complexity can be theorized to be:

$O(N \cdot (A \cup F) + S)$

The evaluation time of the PSO is close to some optimal time. Each element in the PSO is applied once to a single activity node.

While for GA, each element in the expression-tree is applied over all activity nodes. On top of that there is some complexity added in calculating the value of the function nodes.

Analyzing performance

From the results in **Appendix: Makespans**, it can be seen that GA generally produces shcdules with lower makespans.

The GA and PSO seems to have similair makespans for some projects, but the GA often ends up breaking into a lower makespan.

I think the priority rules guides the GA to more optimal solutions.

The priority rules makes it so the GA is guided towards the global optima.
When finding a solution with GA in search space, the solution is guided by priority rules, which extracts features from the project to make a schedule. Meanwhile the PSO will start with priorities that are more random, and then each individual is guided towards a optimal individual. The individuals in GA aren't only guided with it's best randomly initilized individuals, but also the priority rules.

The priority rules might also make it less likely for the GA to fall into a local optima.
Some priority rule might constraint all individuals to always prioritize some task before another task. This makes the search space constrained. Since priority rules are known in the literature for being usefull for making optimal schedules, it can be that the constrained search space, is more in the global optimal search space. PSO isn't constrained in this way. If some best individual is at a local optimum top, then it might draw all individuals towards that local optimum top, while with GA all individuals would've been constraint to the global optimal search space.

On optemizing PSO using priority rules

The PSO can potentially also be optemized using priority rules. It would've been intresting to see how priority rules would've changed the performance.

I haven't seen that this has been discussed a lot in the literature. One way to do it could be that each element in a position-vector is a weight for some priority rule. Then all priority rules are applied over each task in a project to extract a priority. With such a solution the time complexity would've been closer to GA too.

A strength of the PSO is it's quick running time. And even if priority rules were added, the arithmetic operations on the rules would've been more limited than in GA. The position and velocity in PSO is calculated using some basic arithmetic operatos, wheras the GA combines a larger variety of these, and at a larger amount through an expression tree.

So it might not be fruitfull to optemize PSO with priority rules.

Comparison of implementation

One strength the PSO has over GA is that PSO is easier to implement. The PSO is relativly easy to implement, wheras the GA requiers dealing a binary trees as data structures, and having to implement various priority rules.

Use cases

The GA has quite a lot of running time, but produces better schedules.

The PSO can produce a solution in a few seconds. The GA will might generally produce a solution in an ~hour, but this also depends on the amount of tasks.

If the quality of the schedule is important, and the waiting time to get the schdule is less important. In that case GA can be better.

If some schedule is needed immedialty, while the quality of the schedule is of less concern, then PSO will be better.

For example with machinary that might need some tasks scheduled immediatly, then PSO will be better, but if it's a schedule that might be used for a project with a lot of cost related to it, or a project that might be carried out later and waiting an hour for the schedule is feasble, then GA will be better.

Conclusion

It was found that the GA produces better solutions than PSO. This is theorized to be because GA has priority rules that guides the solution to a optimum search space. But GA also has a larger running time because the priority rules will have some running time related to them, while they also have to applied to every task. PSO only applies one element to a one task.

PSO can be better in situations where a schedule is needed quickly, while GA can be better if the time taken to calculate the schedule is not of concern.

The algorithms can potentially be optemized further. In the literature there is a lot of variation in the usage of which priority rules and arithmetic operators. Some rules might yield better results.

It might be possible to decrease the running time of the GA using threads. This wasn't done in this implementation. When calculating function nodes, perhaps threads can be used to calculate all these values in a thread, and then apply the arithmetic operations when all function node priorities have been calculated.

Appendix

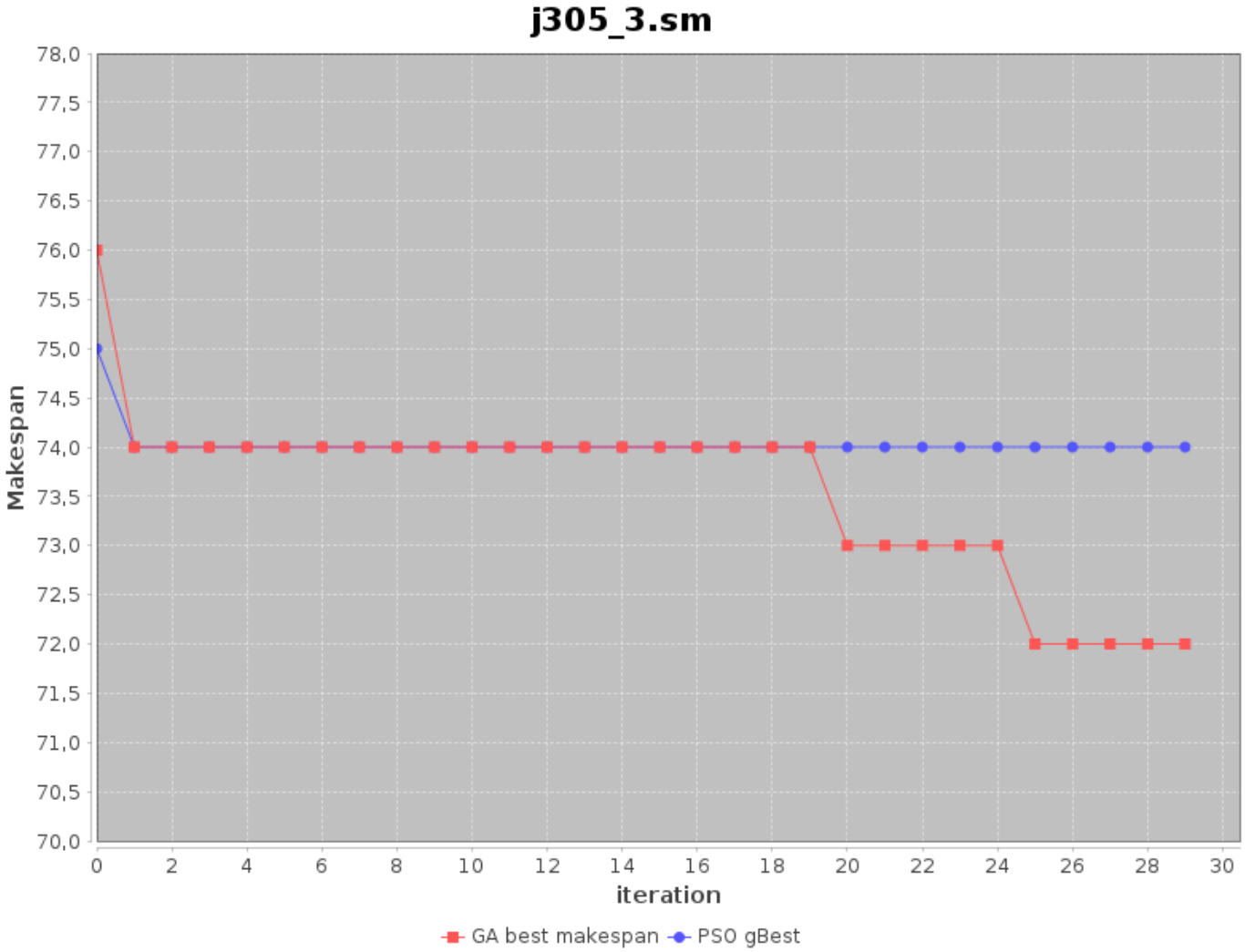
Makespans

The graphs shows the lowest makespan of the fittest individual at each iteration for GA and PSO. There is also a table showing how far each algorithm was from the lower bound, at the final iteration.

J30

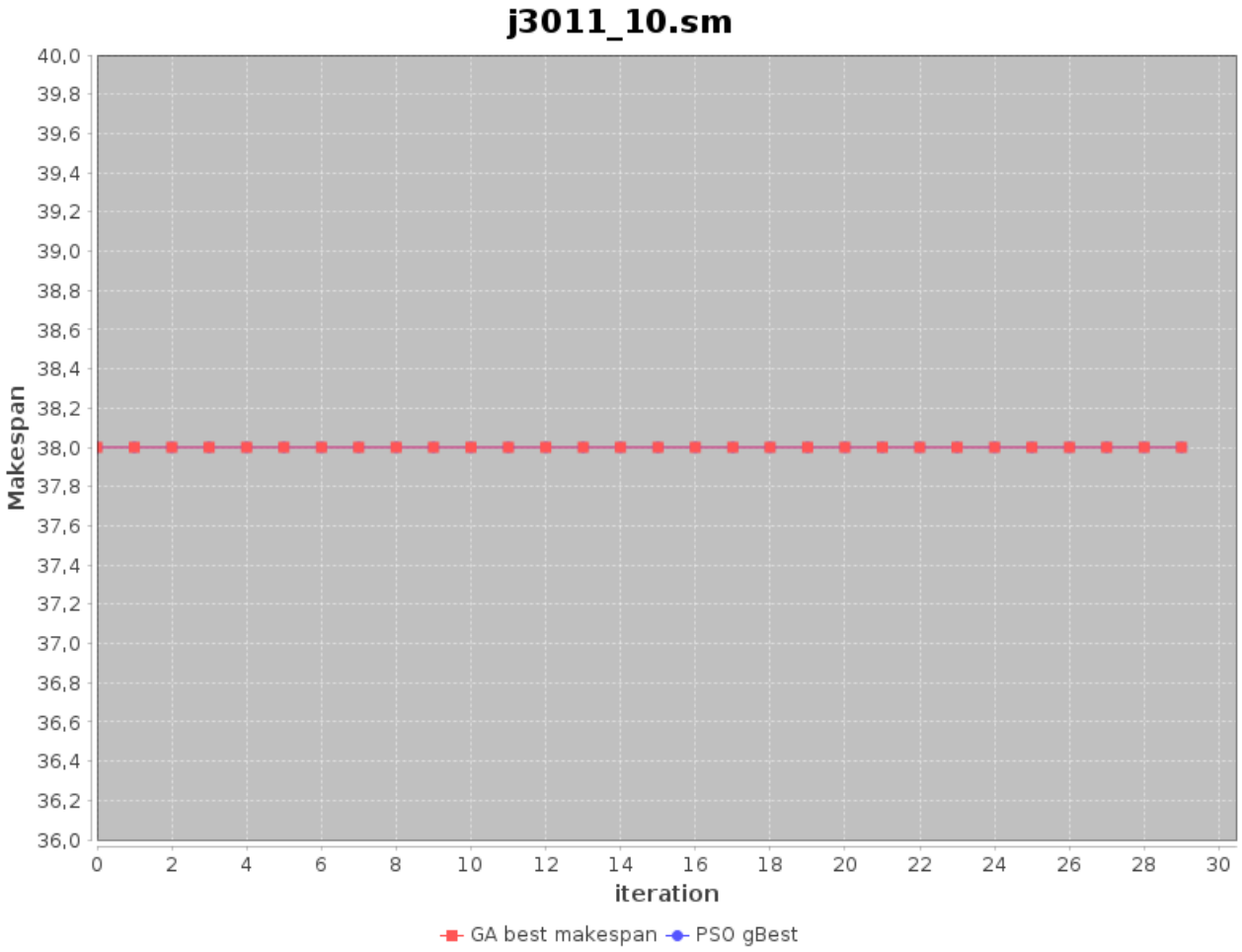
Deviation

Project file	PSO	GA
J305_3.sm	2	0



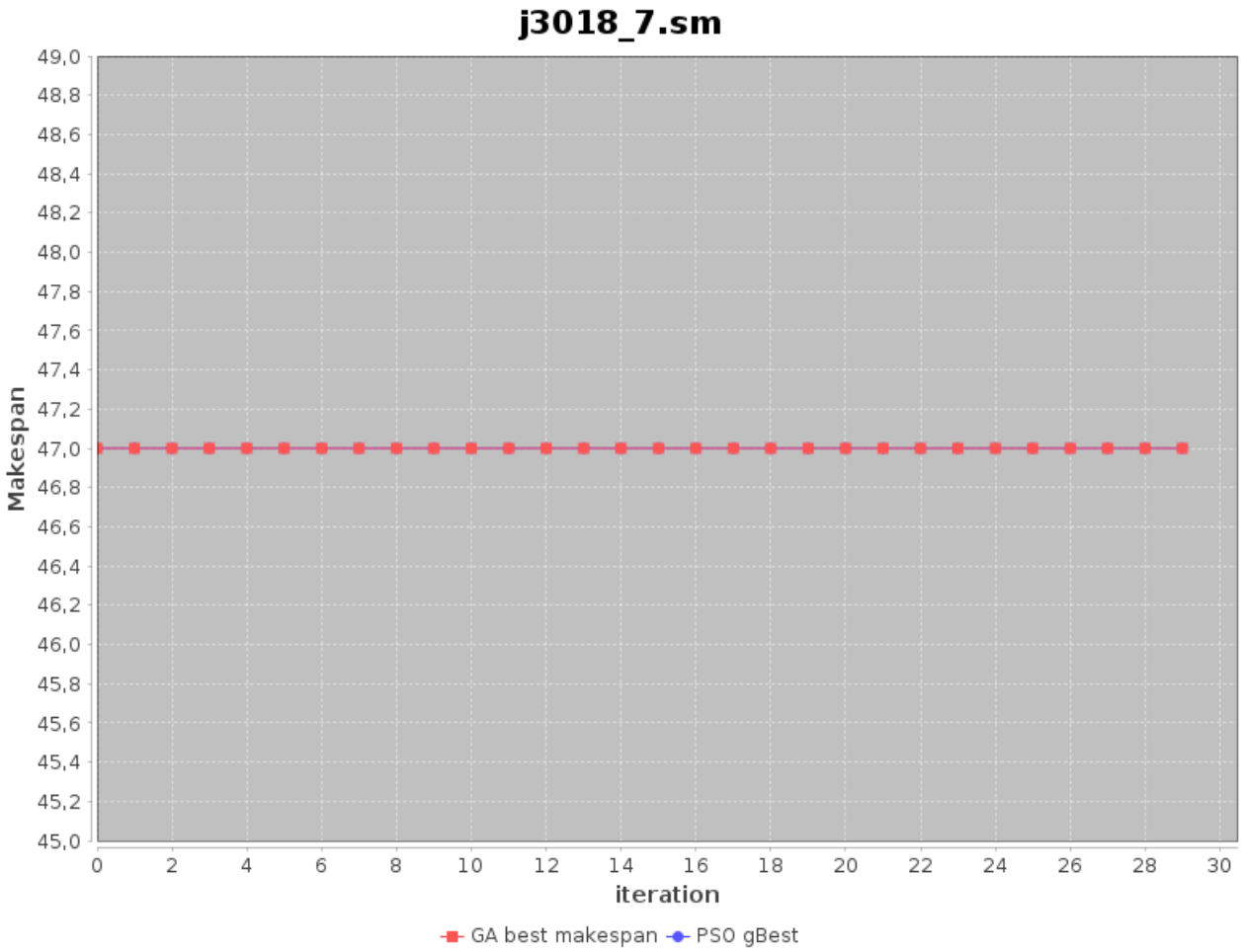
Deviation

Project file	PSO	GA
J3011_10.sm	0	0



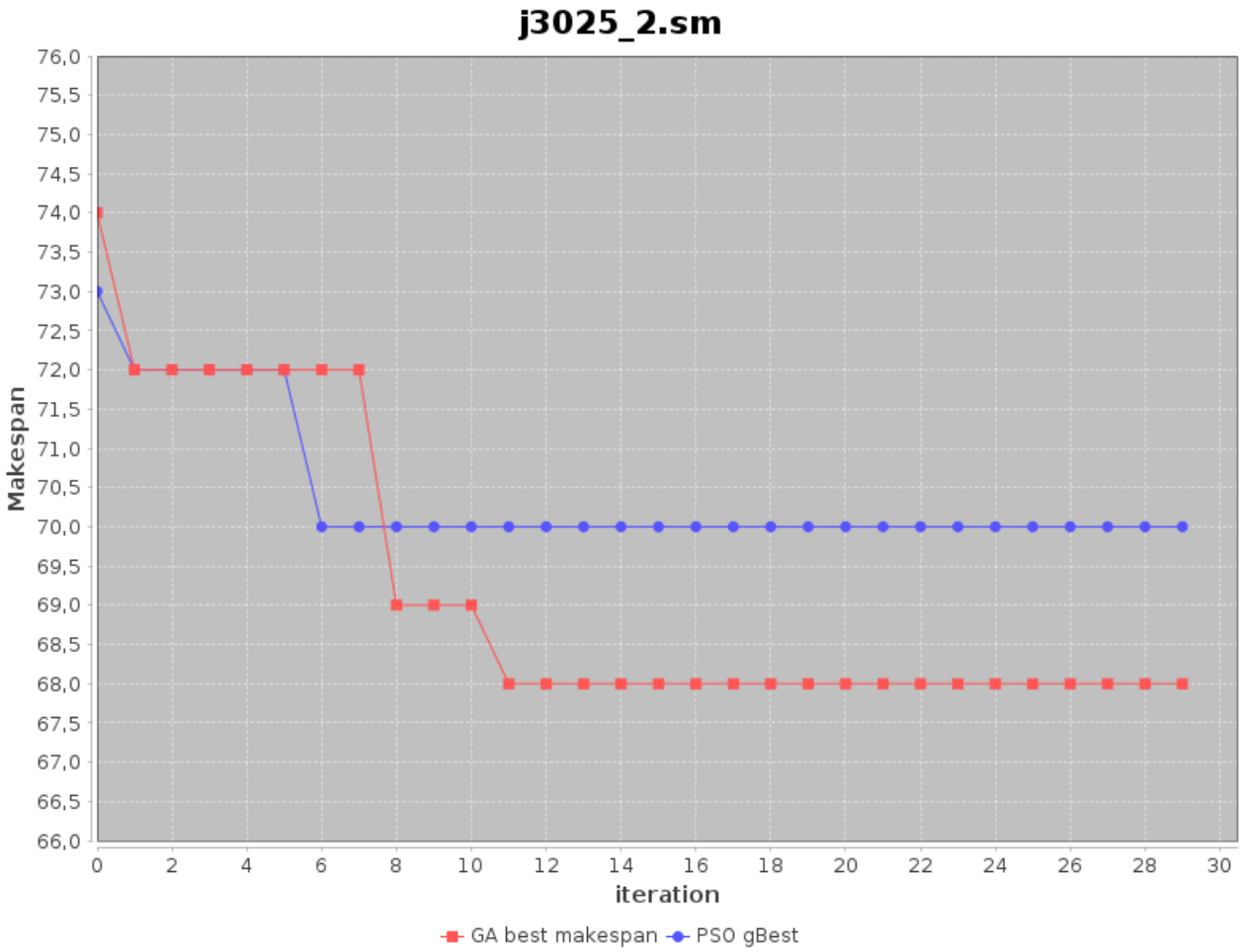
Deviation

Project file	PSO	GA
J3018_7.sm	0	0



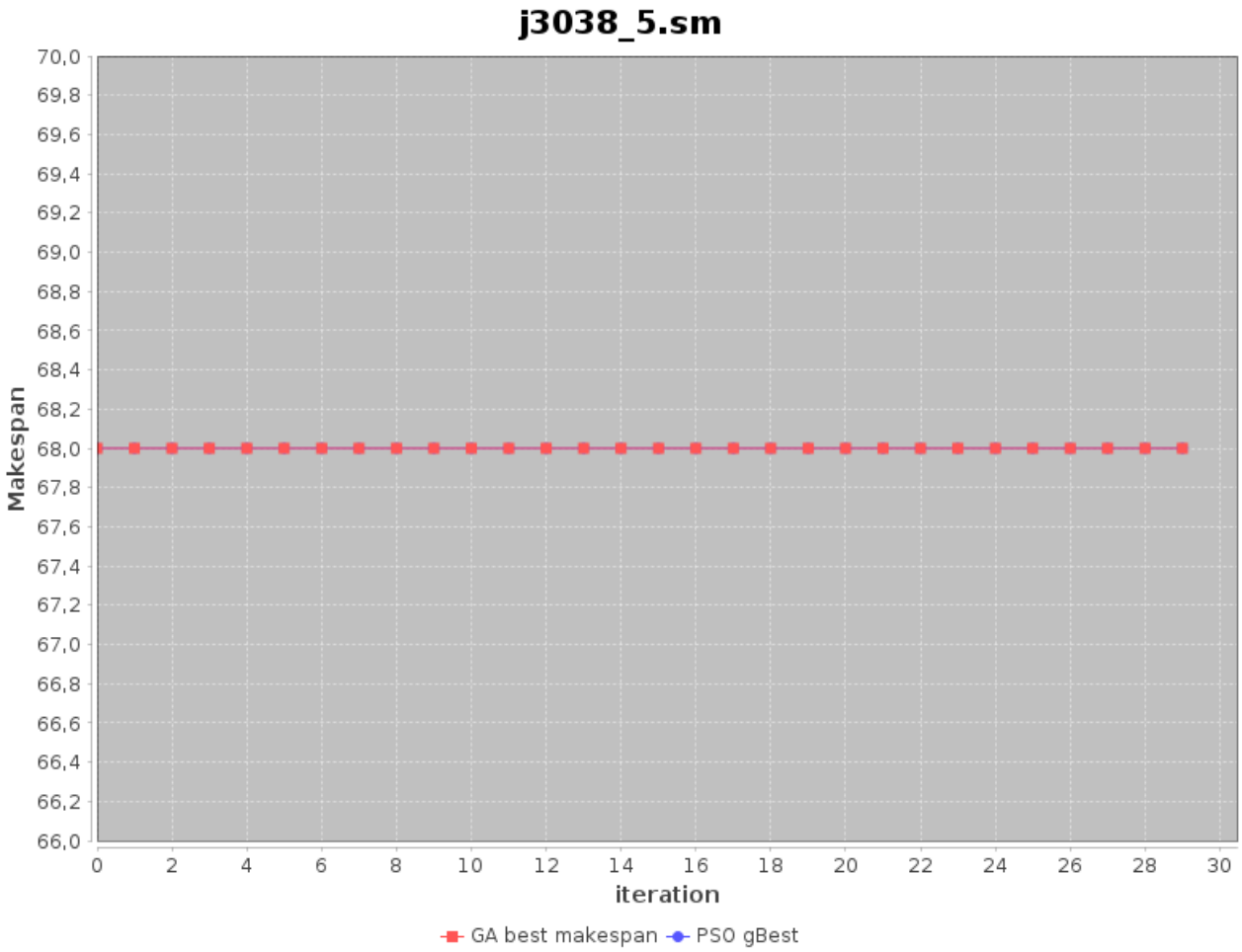
Deviation

Project file	PSO	GA
J3025_2.sm	4	2



Deviation

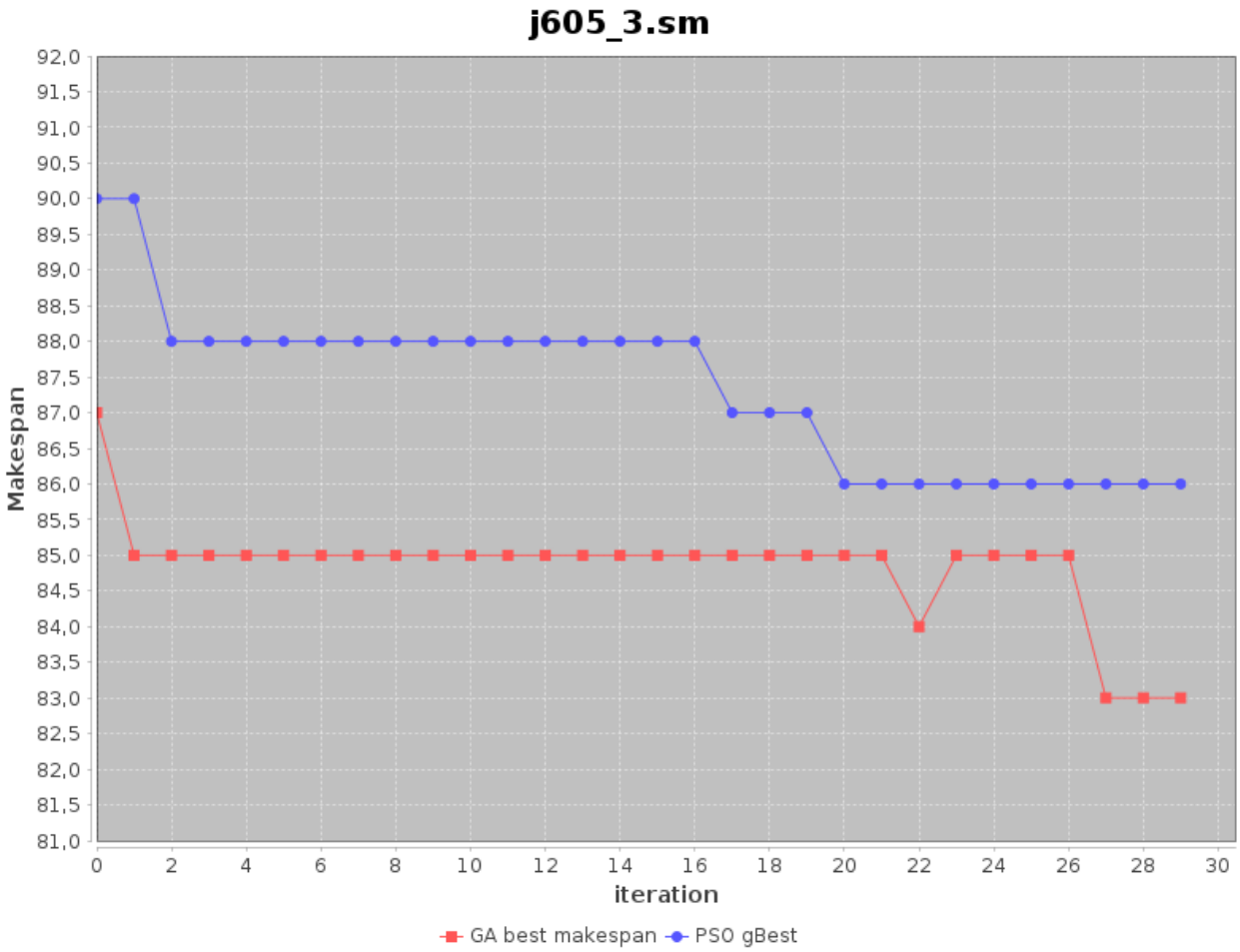
Project file	PSO	GA
J3038_5.sm	0	0



J60

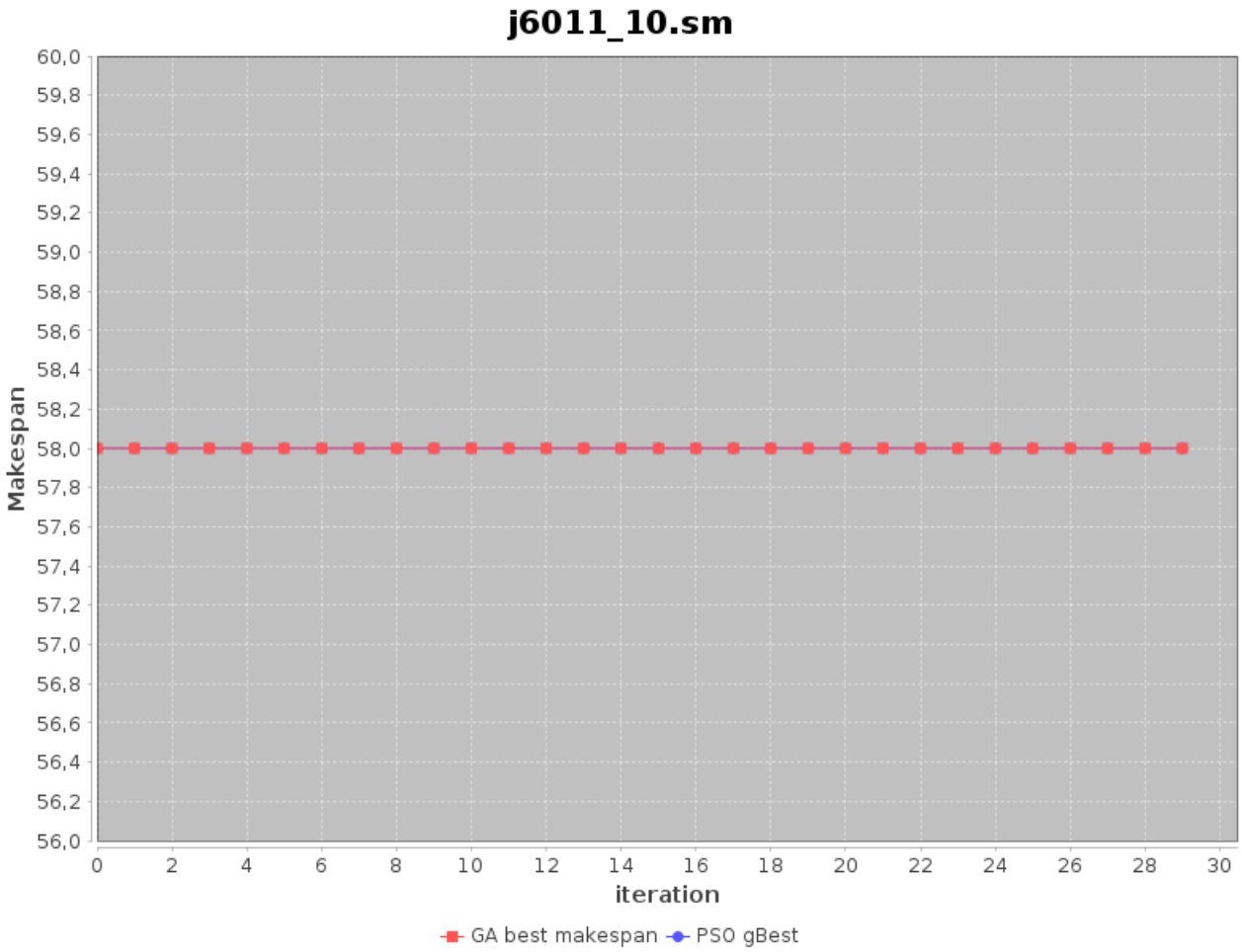
Deviation

Project file	PSO	GA
J605_3.sm	7	4



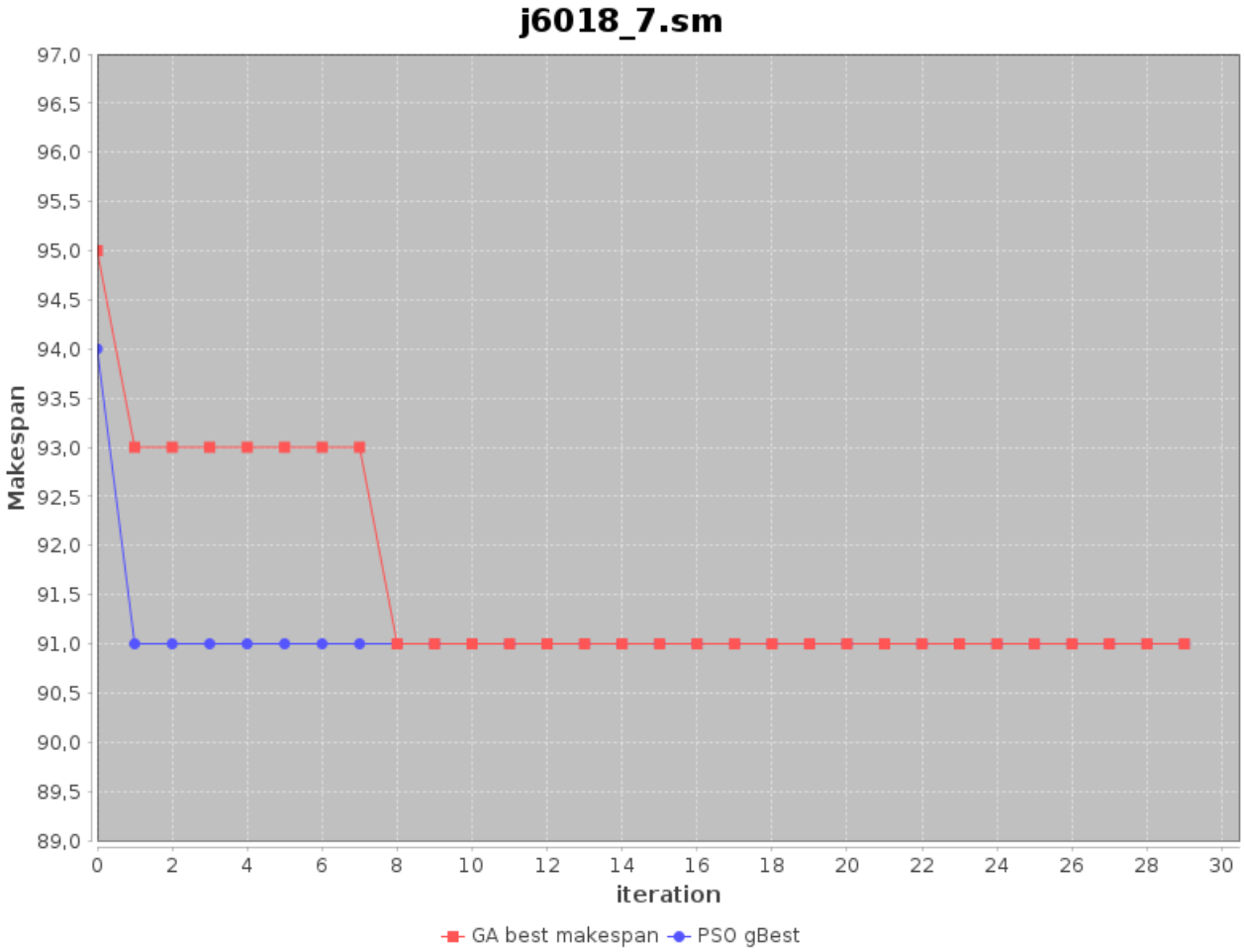
Deviation

Project file	PSO	GA
J6011_10.sm	0	0



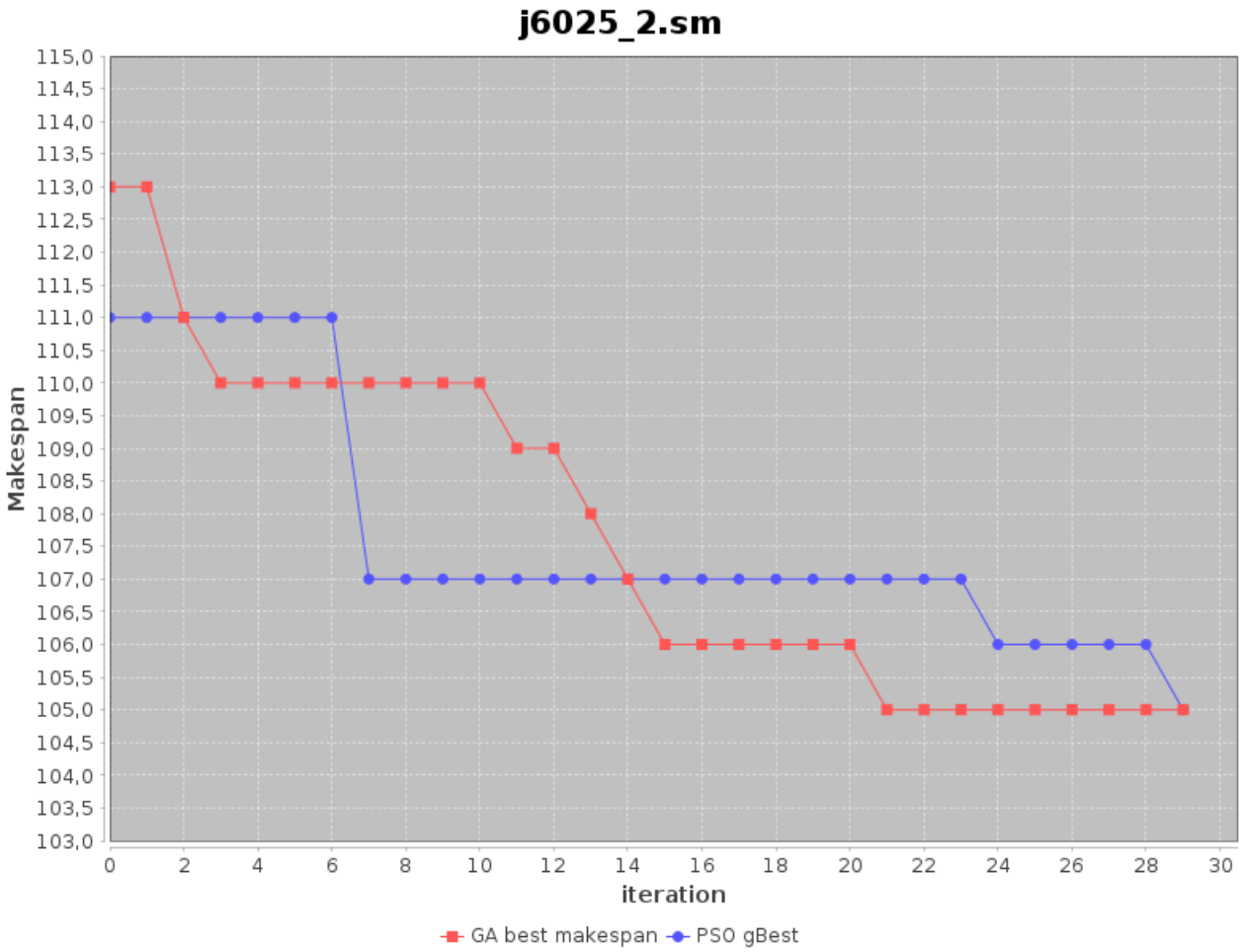
Deviation

Project file	PSO	GA
J6018_7.sm	0	0



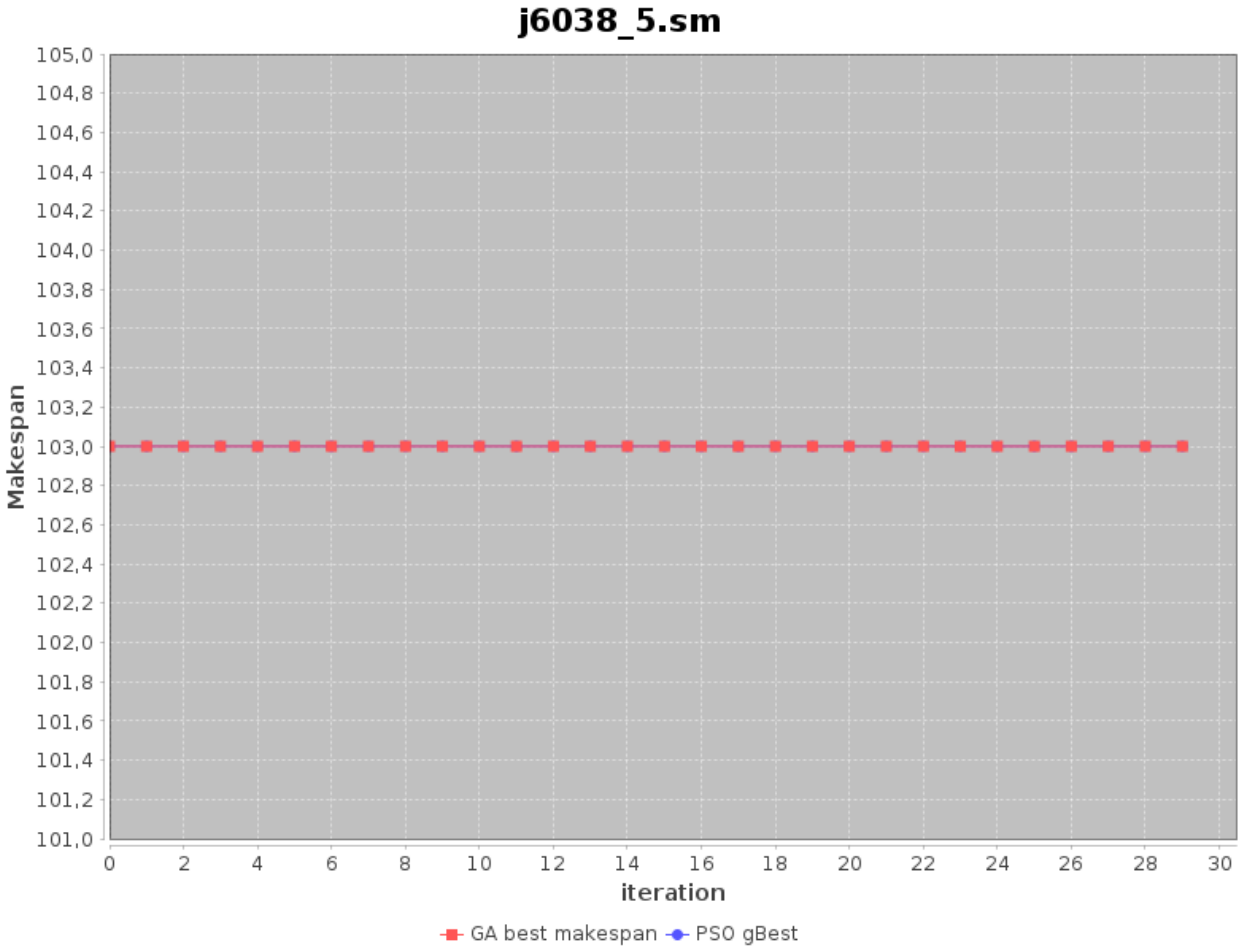
Deviation

Project file	PSO	GA
J6025_2.sm	7	7



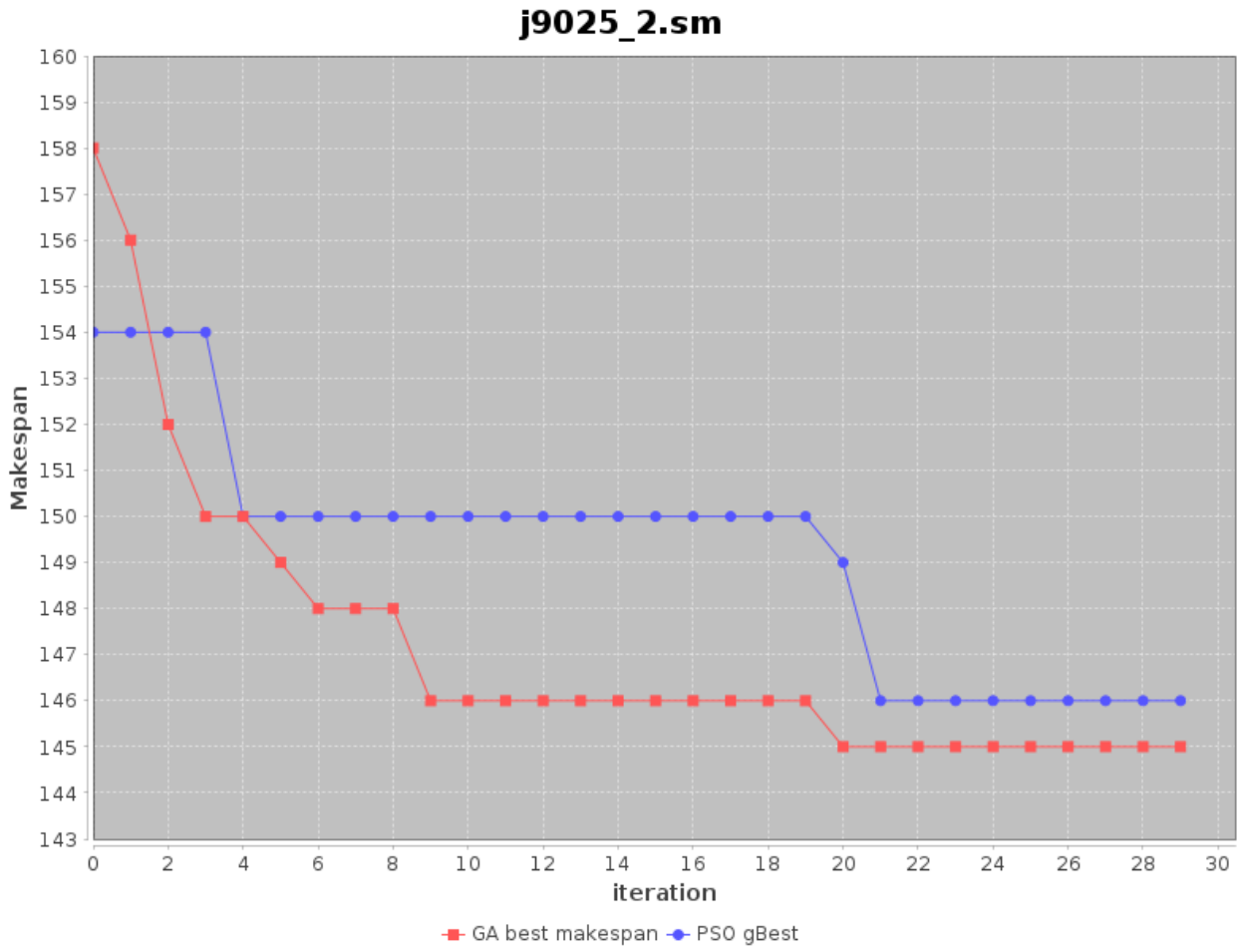
Deviation

Project file	PSO	GA
J6038_5.sm	0	0



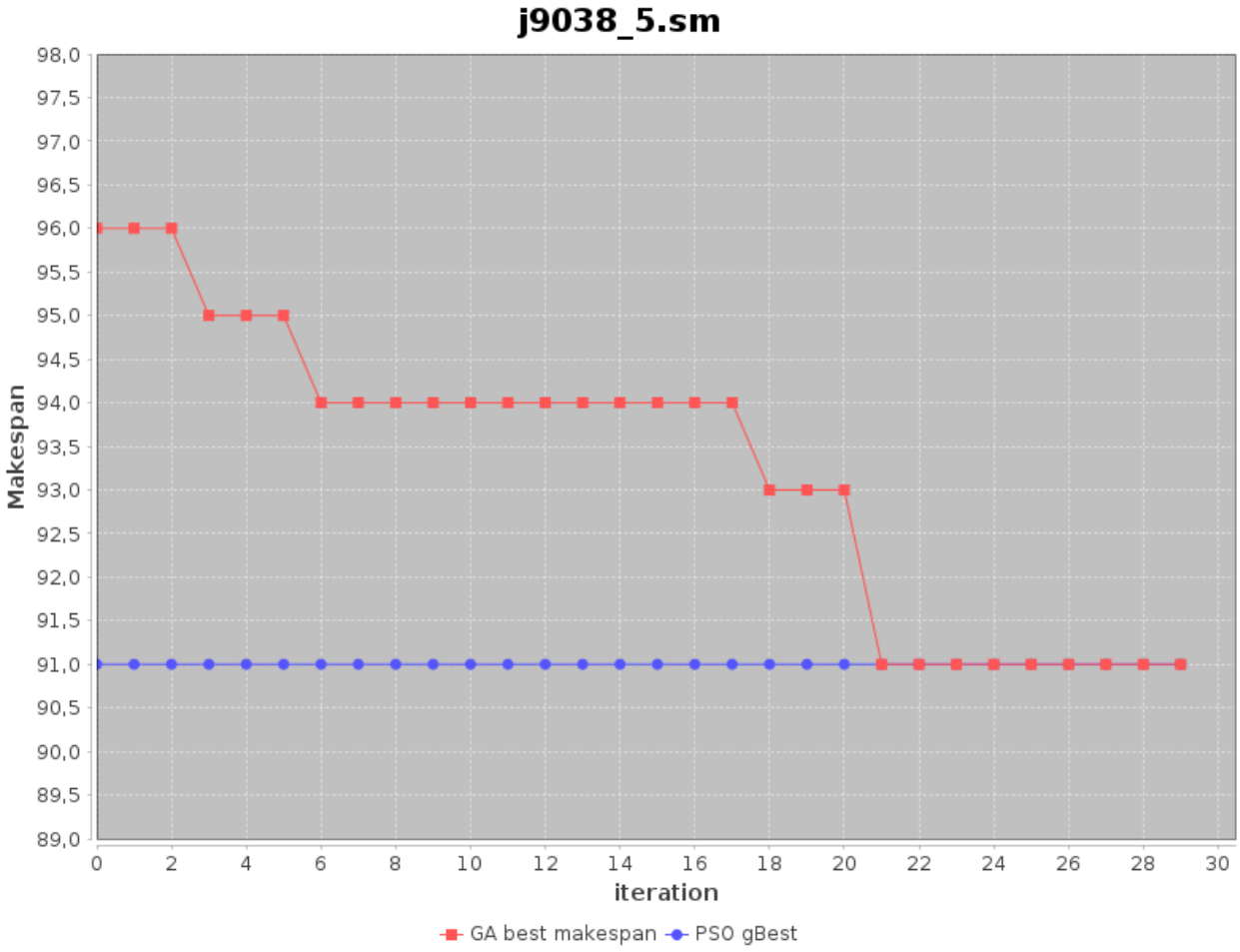
Deviation

Project file	PSO	GA
J9025_2.sm	3	2



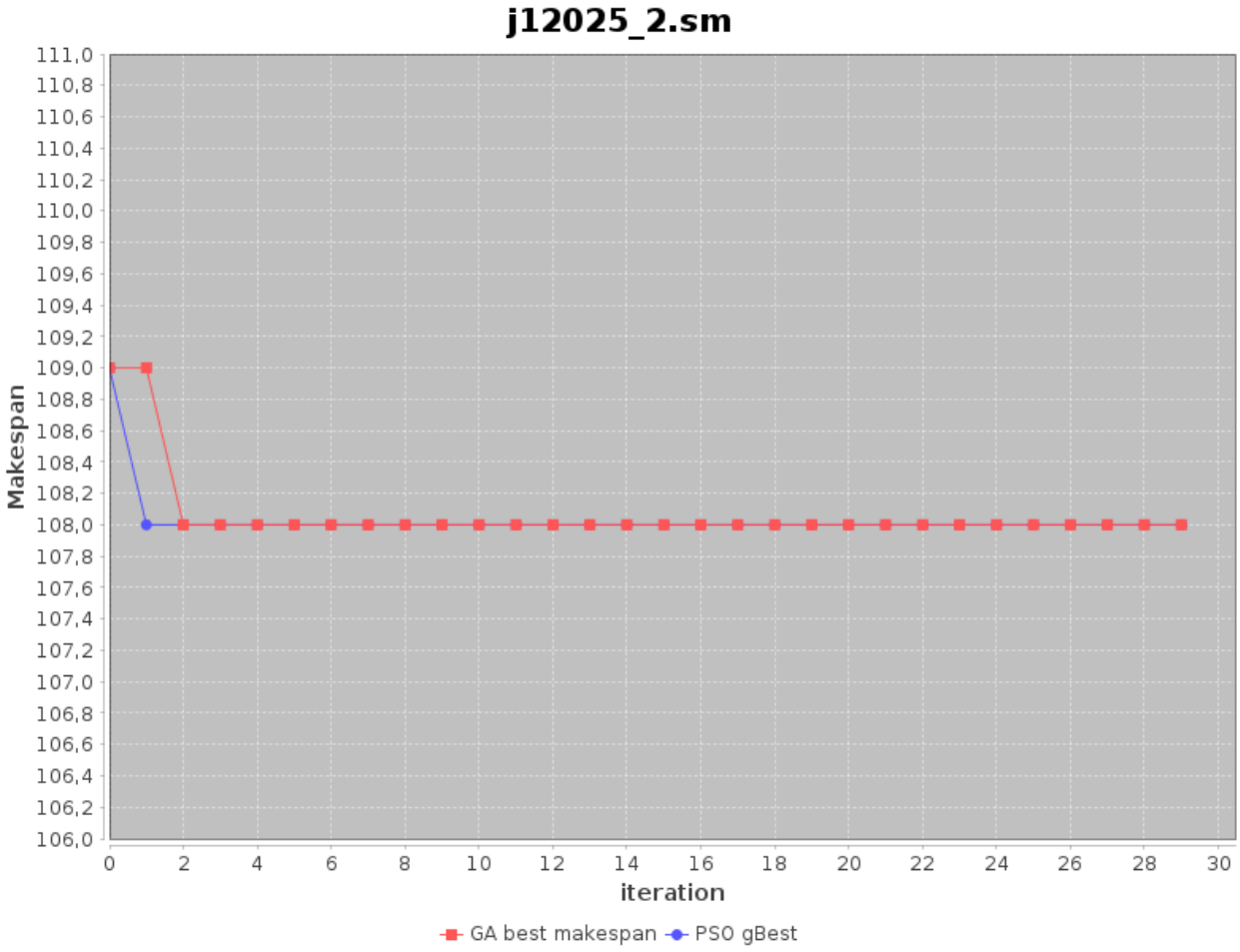
Deviation

Project file	PSO	GA
J9038_5.sm	0	0



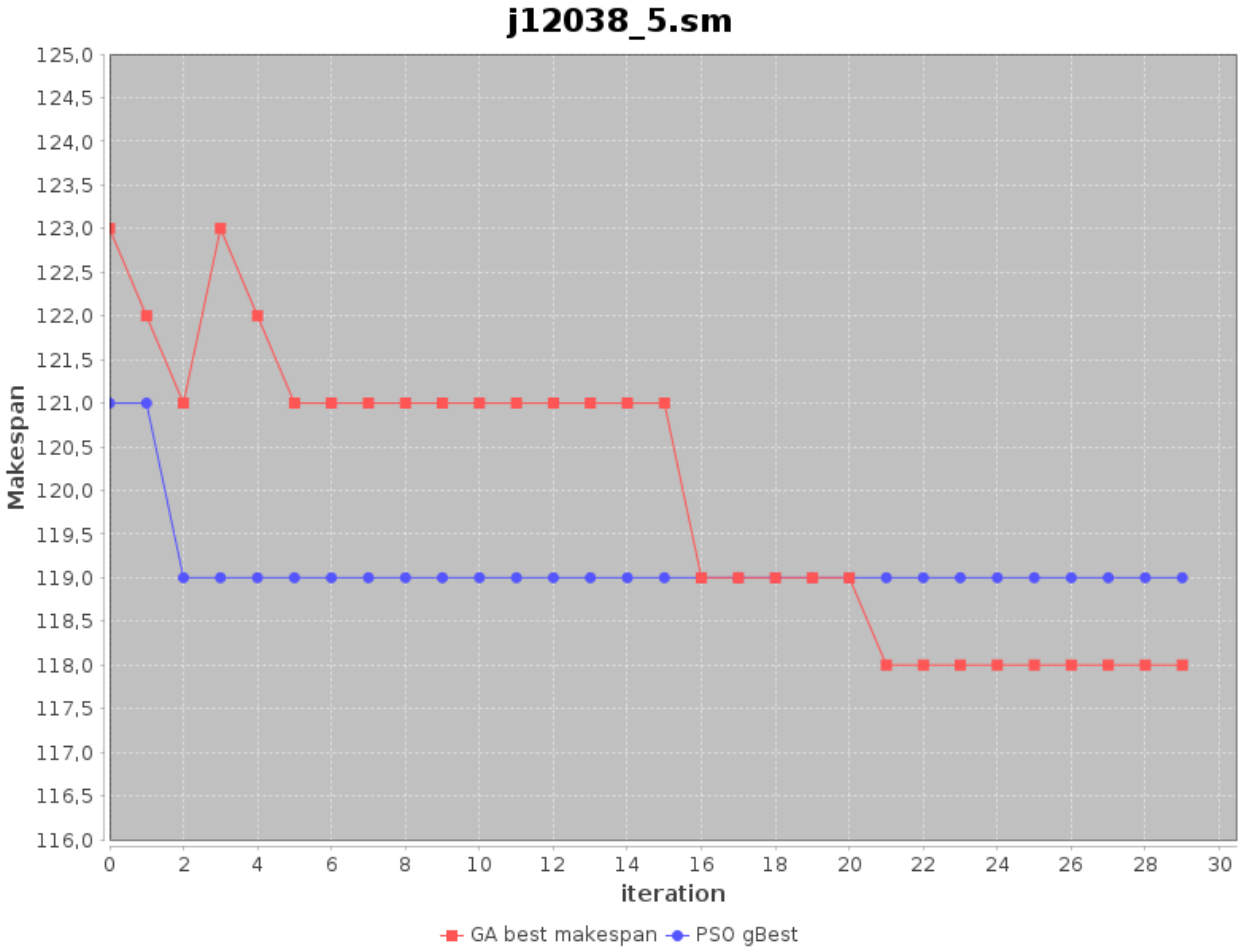
Deviation

Project file	PSO	GA
J12025_2.sm	0	0



Deviation

Project file	PSO	GA
J12038_5.sm	5	4



Priority rules and trees

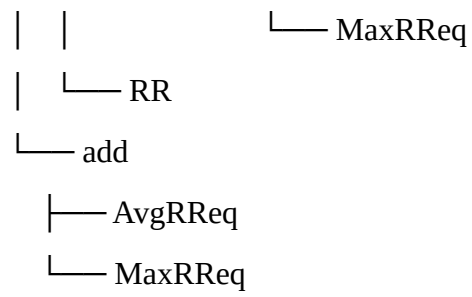
This section shows some examples of generated expression-trees, and the different priority rules used. Only the trees for J305_3.sm and J6025_2.sm will be shown as examples.

J305_3.sm

Tree Structure:

```
max
├── add
│   ├── div
│   │   ├── max
│   │   │   ├── div
│   │   │   │   ├── div
│   │   │   │   │   ├── TPC
│   │   │   │   │   └── MinRReq
│   │   │   │   └── add
│   │   │   │       ├── neg1
│   │   │   │       │   ├── MaxRReq
│   │   │   │       │   └── TPC
│   │   │   │       └── RR
│   │   │   └── max
│   │   │       ├── neg1
│   │   │       │   ├── TPC
│   │   │       │   └── TPC
│   │   │       └── min
│   │   │           ├── TPC
│   │   │           └── TSC
│   │   └── sub
│   │       ├── min
│   │       │   ├── add
│   │       │   │   ├── neg1
│   │       │   │   │   ├── TPC
```

```
│   │   │   │   │   └── MaxRReq
│   │   │   │   └── add
│   │   │   │       ├── add
│   │   │   │       │   ├── TSC
│   │   │   │       │   └── ES
│   │   │   │       └── TSC
│   │   │   └── div
│   │   │       ├── max
│   │   │       │   ├── ES
│   │   │       │   └── EF
│   │   │       └── mul
│   │   │           ├── mul
│   │   │           │   ├── TSC
│   │   │           │   └── MinRReq
│   │   │           └── neg1
│   │   │               ├── RR
│   │   │               └── AvgRReq
│   │   └── neg1
│   │       ├── TSC
│   │       └── add
│   │           ├── sub
│   │           │   ├── add
│   │           │   │   ├── RR
│   │           │   │   └── RR
│   │           │   └── neg1
│   │           │       ├── ES
│   │           │       └── TSC
│   │           └── max
│   │               ├── MaxRReq
│   │               └── div
│   │                   ├── RR
```



Attributes of final fittest individual

Attribute	Count
EF	1
MaxRReq	5
ES	3
TSC	6
AvgRReq	2
TPC	6
MinRReq	2
RR	6

Attributes in all individuals in final population

Attribute	Count
EF	39892
MaxRReq	76125
ES	49157
TSC	78313
AvgRReq	66074
TPC	45944
MinRReq	36256
RR	77649

J3011_10.sm

Attributes of final fittest individual

Attribute	Count
EF	1
MaxRReq	3
ES	2
TSC	1
AvgRReq	2
TPC	3
MinRReq	1
RR	3

Attributes in all individuals in final population

Attribute	Count
EF	29531
MaxRReq	23418
ES	85381
TSC	23177
AvgRReq	54082
TPC	41609
MinRReq	31065
RR	45838

J3018_7.sm

Attributes of final fittest individual

Attribute	Count
EF	0
MaxRReq	3
ES	1
TSC	3
AvgRReq	0
TPC	2
MinRReq	0
RR	4

Attributes in all individuals in final population

Attribute	Count
EF	23236
MaxRReq	92100
ES	57113
TSC	44521
AvgRReq	30609
TPC	29792
MinRReq	24706
RR	31153

J3025_2.sm

Attributes of final fittest individual

Attribute	Count
EF	2
MaxRReq	8
ES	3
TSC	2
AvgRReq	3
TPC	4
MinRReq	0
RR	1

Attributes in all individuals in final population

Attribute	Count
EF	69427
MaxRReq	156097
ES	130648
TSC	79017
AvgRReq	84868
TPC	114063
MinRReq	31878
RR	45035

J3038_5.sm

Attributes of final fittest individual

Attribute	Count
EF	2
MaxRReq	2
ES	2
TSC	2
AvgRReq	7
TPC	2
MinRReq	4
RR	0

Attributes in all individuals in final population

Attribute	Count
EF	66796
MaxRReq	58249
ES	43983
TSC	23842
AvgRReq	163688
TPC	71196
MinRReq	64305
RR	43987

J6011_10.sm

Attributes of final fittest individual

Attribute	Count
EF	2
MaxRReq	2
ES	1
TSC	3
AvgRReq	3
TPC	1
MinRReq	1
RR	5

Attributes in all individuals in final population

Attribute	Count
EF	81088
MaxRReq	105639
ES	50696
TSC	81498
AvgRReq	56039
TPC	46312
MinRReq	50367
RR	58084

J6018_7.sm

Attributes of final fittest individual

Attribute	Count
EF	6
MaxRReq	2
ES	13
TSC	9
AvgRReq	8
TPC	11
MinRReq	4
RR	8

Attributes in all individuals in final population

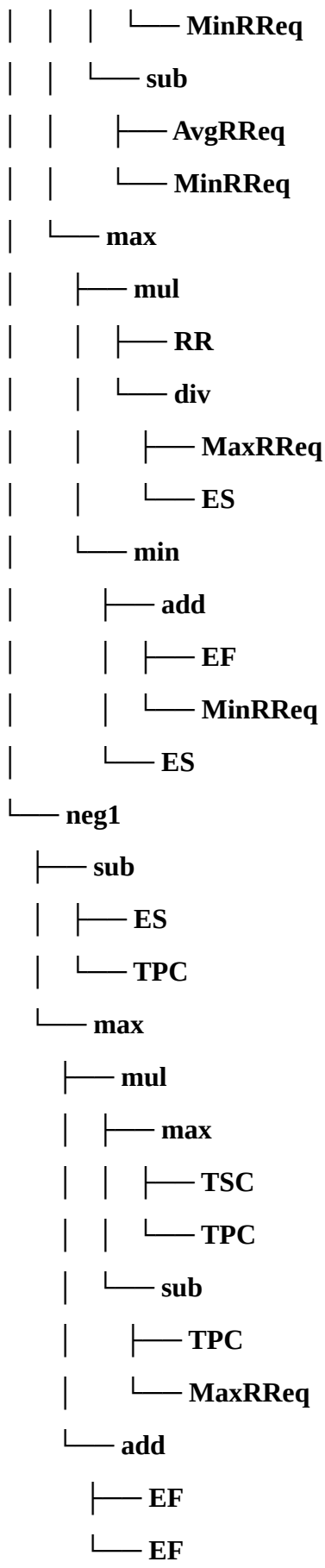
Attribute	Count
EF	101064
MaxRReq	52270
ES	121709
TSC	124243
AvgRReq	69849
TPC	112658
MinRReq	104190
RR	90257

J6025_2.sm

Tree Structure:

add

├── max
│ ├── max
│ │ ├── max
│ │ │ ├── mul
│ │ │ │ ├── EF
│ │ │ │ └── RR
│ │ │ └── max
│ │ │ ├── TPC
│ │ │ └── ES
│ │ └── max
│ │ ├── mul
│ │ │ ├── RR
│ │ │ └── div
│ │ │ ├── MaxRReq
│ │ │ └── ES
│ │ └── min
│ │ ├── add
│ │ │ ├── EF
│ │ │ └── MinRReq
│ │ └── ES
├── EF
└── div
├── max
│ ├── add
│ │ ├── div
│ │ │ ├── TPC


MinRReq
sub
AvgRReq
MinRReq
max
mul
RR
div
MaxRReq
ES
min
add
EF
MinRReq
ES
neg1
sub
ES
TPC
max
mul
max
TSC
TPC
sub
TPC
MaxRReq
add
EF
EF

Attributes of final fittest individual

Attribute	Count
EF	6
MaxRReq	3
ES	6
TSC	1
AvgRReq	1
TPC	5
MinRReq	4
RR	3

Attributes in all individuals in final population

Attribute	Count
EF	139797
MaxRReq	113930
ES	250146
TSC	90995
AvgRReq	159070
TPC	174596
MinRReq	172959
RR	103740

J6038_5.sm

Attributes of final fittest individual

Attribute	Count
EF	6
MaxRReq	6
ES	5
TSC	6
AvgRReq	2
TPC	5
MinRReq	8
RR	7

Attributes in all individuals in final population

Attribute	Count
EF	160558
MaxRReq	152697
ES	90609
TSC	172201
AvgRReq	130664
TPC	85475
MinRReq	119837
RR	183990

J9025_2.sm

Attributes of final fittest individual

Attribute	Count
EF	13
MaxRReq	15
ES	9
TSC	15
AvgRReq	7
TPC	7
MinRReq	6
RR	6

Attributes in all individuals in final population

Attribute	Count
EF	117366
MaxRReq	168015
ES	122032
TSC	186355
AvgRReq	75311
TPC	85134
MinRReq	99994
RR	67602

J9038_5.sm

Attributes of final fittest individual

Attribute	Count
EF	8
MaxRReq	10
ES	8
TSC	8
AvgRReq	5
TPC	8
MinRReq	8
RR	6

Attributes in all individuals in final population

Attribute	Count
EF	48441
MaxRReq	59981
ES	49930
TSC	51325
AvgRReq	42102
TPC	52386
MinRReq	41941
RR	65365

J12025_2.sm

Attributes of final fittest individual

Attribute	Count
EF	5
MaxRReq	2
ES	7
TSC	8
AvgRReq	3
TPC	4
MinRReq	2
RR	10

Attributes in all individuals in final population

Attribute	Count
EF	75953
MaxRReq	52753
ES	124390
TSC	64857
AvgRReq	73114
TPC	79586
MinRReq	53739
RR	110994

J2038_5.sm

Attributes of final fittest individual

Attribute	Count
EF	7
MaxRReq	5
ES	6
TSC	4
AvgRReq	7
TPC	11
MinRReq	13
RR	9

Attributes in all individuals in final population

Attribute	Count
EF	75484
MaxRReq	60192
ES	75015
TSC	57122
AvgRReq	74407
TPC	101064
MinRReq	113691
RR	99842

Ratio of priority rules

J30

Attribute	Percentage of rules
EF	0.05
MaxRReq	0.19
ES	0.13
TSC	0.13
AvgRReq	0.16
TPC	0.16
MinRReq	0.06
RR	0.12

J60

Attribute	Percentage of rules
EF	0.11
MaxRReq	0.10
ES	0.16
TSC	0.12
AvgRReq	0.12
TPC	0.14
MinRReq	0.10
RR	0.16

J90

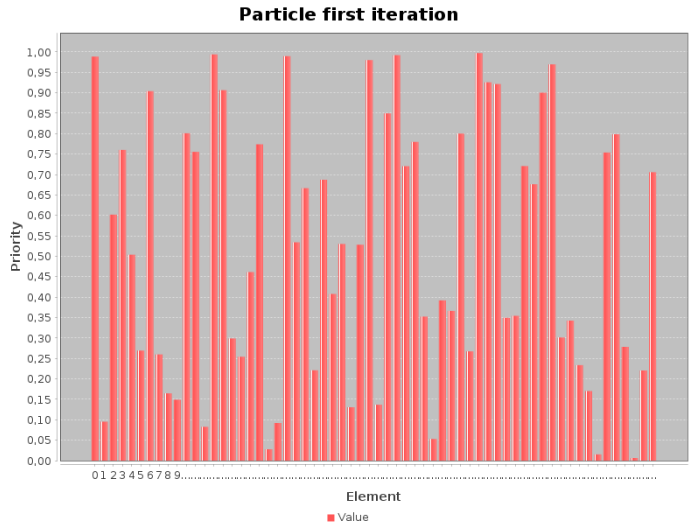
Attribute	Percentage of rules
EF	0.09
MaxRReq	0.19
ES	0.13
TSC	0.18
AvgRReq	0.09
TPC	0.12
MinRReq	0.11
RR	0.09

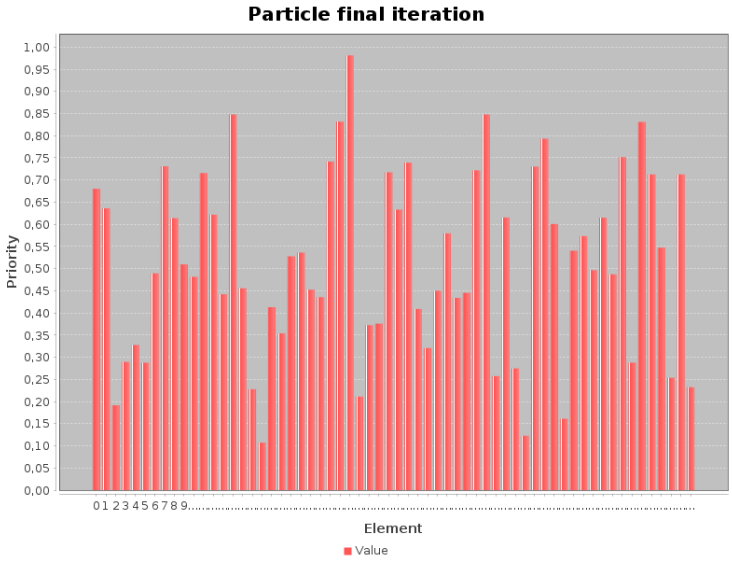
J120

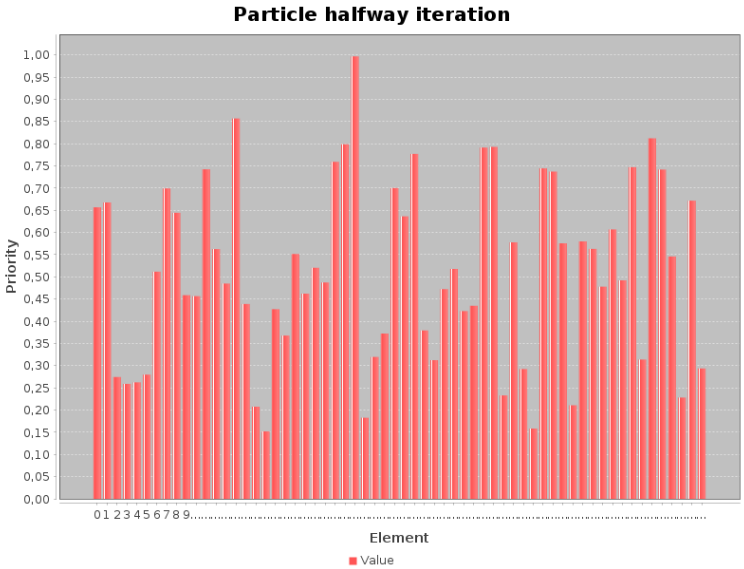
Attribute	Percentage of rules
EF	0.12
MaxRReq	0.07
ES	0.13
TSC	0.12
AvgRReq	0.10
TPC	0.15
MinRReq	0.15
RR	0.18

Particle movment

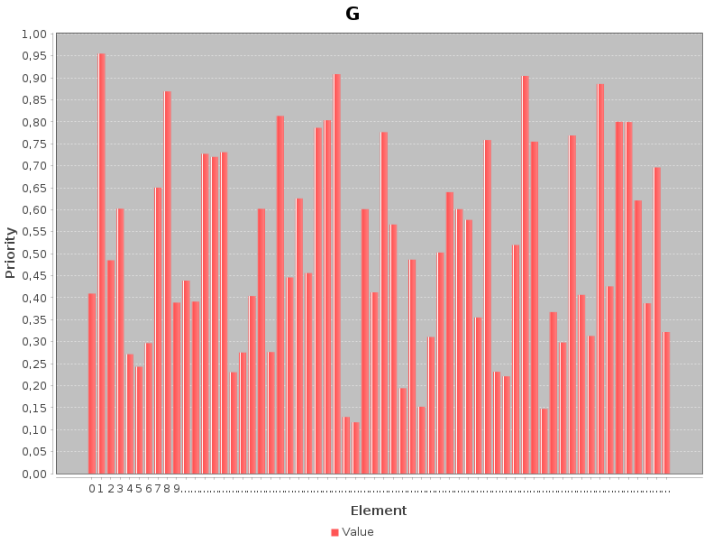
A random particle's position at different iterations.







Position of G (best recorded position in swarm)



Refrences

Luo, J., Vanhoucke, M., Coelho, J., & Guo, W. (2022). An efficient genetic programming approach to design priority rules for resource-constrained project scheduling problem. *Expert Systems with Applications, 198*, 116753. https://doi.org/10.1016/j.eswa.2022.116753

Chen, R.-M. (2011). Particle swarm optimization with justification and designed mechanisms for resource-constrained project scheduling problem. *Expert Systems with Applications, 38*(6), 7102–7111. https://doi.org/10.1016/j.eswa.2010.12.059

Jia, Q., & Guo, Y. (2016). Hybridization of ABC and PSO algorithms for improved solutions of RCPSP. *Journal of the Chinese Institute of Engineers*, *39*(6), 727–734. https://doi.org/10.1080/02533839.2016.1176866

Kennedy, J., & Eberhart, R. (1995). Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks* (Vol. 4, pp. 1942–1948). https://doi.org/10.1109/ICNN.1995.488968

Chand, S., Huynh, Q., Singh, H., Ray, T., & Wagner, M. (2018). On the use of genetic programming to evolve priority rules for resource-constrained project scheduling problems. *Information Sciences, 432*, 146–163. https://doi.org/10.1016/j.ins.2017.12.013

Chand, S., Rajesh, K., & Chandra, R. (2022). MAP-Elites based Hyper-Heuristic for the Resource Constrained Project Scheduling Problem. *arXiv*. https://doi.org/10.48550/arXiv.2204.11162

Kelley, J. E. (1961). Critical-path planning and scheduling: Mathematical basis. *Operations Research, 9*(3), 296–320. https://doi.org/10.1287/opre.9.3.296

Kolisch, R., & Sprecher, A. (1997). PSPLIB - A project scheduling problem library: OR Software - ORSEP Operations Research Software Exchange Program. *European Journal of Operational Research, 96*(1), 205–216. https://doi.org/10.1016/S0377-2217(96)00170-1