Comparative Analysis on approximation algorithms for the Resource Constrained Project Scheduling Problem

José Silva Coelho, Luis Valadares Tavares
Dpto. de Engenharia Civil, Instituto Superior Tecnico, Portugal.
jcoelho@civil.ist.utl.pt, lavt@civil.ist.utl.pt

Abstract

In Resource Constrained Project Scheduling Problem (RCPSP) exact algorithms can only solve small instances, and several approximation algorithms have been proposed that can solve large instances with a quasi-optimal solution. So far there is no comparative analysis of several algorithms running on the same machine, with equal time limit and large instances; this is our main goal. A new performance indicator is proposed that is not affected by the presence of simple problems in the instance set. The instance set was built by assembling the main sets in the literature and also JobShop and FlowShop instances transformed to RCPSP. In this work 30 meta-heuristics were implemented, obtaining a total of 159.364.665 solutions evaluated of RCPSP.

1. Introduction

In this work several approximation algorithms for the RCPSP are compared in the same machines, with equal time limits and an instance set including large instances. A new performance indicator, that is not affected by the presence of simple problems in the instance set, and a simple test check routine are also proposed. In the rest of this section we describe the complete instance set, witch gathers all main instance sets of RCPSP found in the literature, and also JobShop and FlowShop instances, and we define the six priority rules used and a simple and fast lower bound, needed for the simplicity test that is presented in section 2. In section 3 the operators implemented and the meta-heuristics used and its parameters are defined. A new performance indicator is proposed in section 4 and in section 5 the computational results are shown.

The instance set was built by assembling the main instances sets in the literature. We do not lose anything including easy instances, because our new performance indicator is not affected for the presence of this type of instances. The oldest instances sets included are from Patterson (110 problems, 6 to 49 activities) and Alvarez (144 problems, 25 to 101 activities), and a more recent instance set is from Kolish, Sprecher and Drexl (KSD) (2040 problems, 30 to 120 activities). Instances transformed from JobShop and FlowShop (113 problems, 36 to 1500 activities) are included in the set, from several authors. There are also two instance sets generated, by Tavares generator (72 problems, 25 to 4000 activities) and an Elmaghraby generator (36 problems, 6 to 120 activities). The instance set totals 2515 problems, with a number of activities ranging from 6 to 4000.

A priority rule is based on ranking the activities according to a criterion, and then building a schedule by giving priority to activities with a higher ranking over the activities with a lower ranking, but without violating precedence and resource restrictions. It is a simple technique for obtaining solutions, since the only thing required is to calculate a value for each activity (see [1]).

The best result of the priority rules will be the first Upper Bound and the start point for the approximation algorithms, and only the improve made over this value will count for the performance of the algorithm in our new performance indicator. The priority rules used are: Latest Start Time (LST); Latest Finish Time (LFT); Shortest Processing Time (SPT); Greatest Rank Positional Weight (GRPW); Most Total Successors (MTS); Most Total Successors Processing Time (MTSPT). The first two rules use the latest start and finish time calculated in the CPM, and the logic is the same, schedule first the activities that must start immediately or the project will be

delayed. The SPT schedule first activities with high processing time, because these are the activities that in a bad place will cause the worst effect. The GRPW is the same, but also the processing times of the direct successors are sum. Finally, the MTS schedule first the activities with more successors (direct or indirect), because these activities cause more restrictions, and the rule MTSPT have the same logic, but is sum all the processing times of all successors (direct or indirect).

The use of Lower Bound is also important, because an approximated algorithm only know that it have an optimal solution when it obtains a solution with equal value of the Lower Bound. Having a good Lower Bound avoid losing time searching for a better solution when the optimal solution was already found, but we thing that an heavy Lower Bound does make any sense, because if takes too long, that time can be used for the approximate algorithm. Also, heavy Lower Bounds tend not to work in large instances. The Lower Bound implemented is divided in two parts. First we ignore the resources and build the scheduling with the CPM. Second, we use that scheduling for each resource, and ignoring activities, we correct the resource level violations, obtaining a new value for the lower project duration. The Lower Bound will be the high value obtained. We finalize this section mentioning that we present a fast Lower and Upper Bound, even for large instances (see [3] and [11]).

2. Simplicity Test

A simple instance is an instance that the optimal value can be obtained with only a simple priority rule. The implementation of the test is simple, calculate the Lower and Upper Bounds, defined in the section 1, and if they are equal, than the instance is simple.

A simplicity test is important. Without it we can be including problems that all algorithms solve and obtain the optimal value, and you could control the final results by adding or subtracting simple and/or no simple problems to the instance set. We cannot accept that, and we remove all the simple problems found in the instance set defined in section 1.

Table 1. Number of simple problems by test set

Instance Set	Problems	Not Simple	Simple	% of Simple Instances
Elmaghraby	36	32	4	11%
KSD	2040	1223	817	40%
Tavares	72	72	0	0%
Patterson	144	128	16	11%
JobShop	110	88	22	20%
FlowShop	82	82	0	0%

In this table is the number of problems in each instance set that was found simple. It is a significant reduction of the instance set KSD, and as expected there is no simple problems in JobShop and FlowShop instances. In global terms the 2515 problems are reduced to 1656 problems.

3. Meta-Heuristics

We divide the meta-heuristic implementation into two parts. Firstly the problem specific operators are presented, and secondly the meta-heuristic specific parameters are described. With the technology that we use (COM objects), allow us even change the implementation of the problem and configures the algorithm in run time.

The algorithms need operators in the problem so they can change the solution. We divide the operators in the number of arguments they have. Operators with zero arguments (random

generators), generate a solution from scratch, with one argument (neighbour function), a solution is generated from another solution, and with two arguments (crossover operator), a solution is generated from two solutions.

For the first operator, that takes zero arguments there is a pure random generator, called "Sampling Method" (SM), and a more elaborated operator called "Regret Biased Random Sampling" (RBRS) that is a random method based in a priority rule that selects an activity for scheduling giving more chance for selecting a good activity. To implement this operator a list of all activities that can be scheduled has to be kept in memory (see [8]). We propose a new operator, called "Global Biased Random Sampling" (GBRS) that is also based in a priority rule, with the following formula:

$$\Phi(j) = \rho_j + \varpi \times \varepsilon$$

This formula returns a value for all activities, and ρj is the value of the priority rule, ϖ is a random value between 0 and 1, and ε is positive value in this work we adopted 10. In this method, activities are sorted by the formula result, and there is a need to correct the precedence violation. Sorting and correcting the precedence violations is significantly faster than calculating the list of scheduling activities in each step.

With these operators, normally we are not interest in obtaining a solution if it is not the best one. In that case we can stop calculating a solution if it will not be the best, and that gives space for optimisations. The best optimisation is the "Serial time window bound" (STWB), that marks the starting deadline that each activity must start before, if the final solution is to be improved. When a deadline is violated, the solution construction is aborted saving time. This optimisation can be applied for all the random generators, making six random generators in total.

The neighbour functions operators receive a solution and build another from it. This is the most used operator, that allows moving from one solution to another. All local search algorithms use this function intensively. Several neighbours are found in literature, the "Adjacent Pairwise Interchange" (API), the "Pairwise Interchange" (PI) and the SHIFT operator. We used also the "Right-Shift" (RS) operator, performing a total of four neighbour functions. The API operator generates a new solution for every two consecutive activities that do not have precedence, and can be swapped. The PI operator is a generalization of the API, which allows the swap between every pair of activities if the swap does not violate the precedence. The SHIFT operator shifts every activity to the left and to the right if the precedence restrictions are verified. The RS is the same operator but the shifts are only to the right.

Finally the crossover operator requires two solutions and builds another. This is the base operator of the genetic algorithm, and allows the crossing of a mother and a father, producing a child. The one-point crossover operator is one of the most popular crossover operators. It generates a random number q between 1 and N, and constructs the child with the first q activities from de mother and the rest from the father. The two-point crossover operator is similar, and in the study of Hartmann (see [10]) is concluded to be the best, and so is the one we used (HART). We propose also a crossover operator that we call "Late Join Function" (LJF), and consist in adopting the father solution, and swapping each adjacent pair that is in reverse order in the mother.

Now we specify all meta-heuristics and parameters used. The meta-heuristics will be applied to the different operators defined. There are three types of algorithms analysed: the Sampling Method, the Local Search algorithms, and the Genetic Algorithms.

The Sampling Method algorithm, use the random generator operators. All the time is used to generate random solutions, and the best solution is returned. The names of the algorithms are identical to the operator names, because they are used without search strategy. Note that an "S" is added in the name if the STWB is used in the generator.

For local search algorithms we used four configurations: local search with restart (LS), simulated annealing (SA) and two configurations of tabu-search (TS1,TS2). The local search with restart has the number of neighbours limited to 30, in random order. The solution changes when a

better neighbour is found, and if there is no such neighbour in the 30 analysed, a random restart is made.

The simulating annealing starts at temperature 1, with the neighbours also in random order. The temperature drops 10% if K consecutives changes occur, and increases 50% if K no changes occur. The value adopted for K is 2. The solution changes either if the neighbour is better or equal to the current solution or with some probability depending on the temperature and neighbour value.

The tabu-search has a tabu list of 10 last explored solutions, and the neighbours are still in random order. Only the first 30 neighbours are considered, and the solution is replaced by the best of the neighbours that is not in the tabu list. The second version of tabu-search has a tabu list of size 1, and only the first 10 neighbours are used.

The genetic algorithms use a population of solutions. A generation is built based on the previous generation, and in the end the best solution in the population is returned. We assume the following algorithm to construct of the next generation: the k1 **elite** elements (top elements) are maintained from the previous population; the k2 **survival** elements (no top elements) are randomly selected from the previous population with a higher probability of selecting the better solutions; the k3 **child** elements are selected from random pairs, with a higher probability for better solutions, and constructed with a crossover operator; the k4 **mutants** elements are randomly selected from the previous population with a higher probability of selecting better solutions, and after that they are mutated by applying a random neighbour; the rest of the population is randomly generated. If the population limit is exceeded, then only the better elements of the new generation survive. In some implementations the child solutions are built by using all the population elements instead of a random rule. We think that this method is more flexible, and brings no significant disadvantages. The following table presents the four configurations tested by using crossover operators HART and LJF.

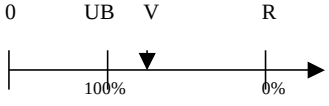
Table 2. Genetic Algorithm configuration

Population	Elite	Survival	Child	Mutants	Strangers
10	2	2	2	2	2
25	4	7	7	7	0
100	5	25	30	20	20
40	40	0	40	2	0

4. Performance Indicator

A good performance indicator is essential to interpret the results. Normally the mean percentage over the Upper or Lower Bound is used. If the optimal value is available, it is normally used, but for large instances normally there is no known optimal value. The problem with these indicators is that a meta-heuristic that does nothing, is classified, and there is no notion of the worst value that can be archived. Suppose that all meta-heuristics results are between 10 and 11 for an instance problem, and in other instance the meta-heuristics are all between 10 and 15. The result will penalise for the worst meta-heuristic in the first instance 10% ((11-10)/10), and in the second instance 50% ((15-10)/10). We think that all instances must value the same. The proposed performance indicator assigns the same weight to each problem, and does not consider problems where all algorithms archive the same result. The indicator formula is the following:

$$Performance = \frac{1}{N}\sum_{i=1}^{N}\frac{R_i - V_i}{R_i - UB_i}$$



In this formula Ri is the value of the starting solution (initial Upper Bound), Vi is the value of the solution obtained for the meta-heuristic, and UBi is the best value obtained for this problem with all meta-heuristics. The figure shows a graphic interpretation of the equation for each instance, the value obtained for the meta-heuristic will be between the starting solution R, and the best solution UB, receiving 0 and 100% respectively.

We think that this indicator is more adequate for comparing algorithms. The values are dependent of the algorithms tested, and as independent as possible of the instance set.

5. Computational Results

Tests were done over all meta-heuristics specified in section 3, with time limits of 0.1 seconds, 1 second, 5 seconds and 20 seconds. Three identical computers were used, with Windows 95, Pentium/200Mhz with physical memory of 32Mb and virtual memory limited to 64Mb. In total 159.364.665 solutions of RCPSP were processed in 1.296.856 seconds (discarding the time needed to load the instances and calculate Upper and Lower Bounds).

Table 3. Performance indicator results of algorithms, by time limit

Algorithm	**0.1 s**	**1 s**	**5 s**	**20 s**	**Algorithm**	**0.1 s**	**1 s**	**5 s**	**20 s**
HARTG4	34%	58%	68%	79%	PISA	17%	38%	52%	60%
HARTG1	35%	59%	70%	78%	SHIFTTS2	22%	44%	56%	60%
LJFG1	35%	58%	70%	77%	APILS	26%	43%	54%	59%
SHIFTLS	34%	56%	71%	76%	SHIFTTS	21%	44%	56%	59%
RSLS	34%	57%	70%	75%	PITS	22%	41%	53%	56%
PILS	35%	57%	70%	75%	PITS2	21%	41%	54%	56%
LJFG4	35%	57%	67%	74%	RBRS	18%	35%	48%	54%
LJFG2	34%	53%	61%	71%	RSTS	22%	41%	52%	53%
HARTG2	34%	52%	61%	70%	SM	17%	35%	46%	52%
GBRS	44%	60%	66%	68%	RSTS2	22%	41%	52%	52%
HARTG3	34%	56%	63%	68%	APITS	9%	18%	24%	51%
LJFG3	34%	56%	63%	68%	APITS2	9%	18%	23%	48%
RSSA	19%	42%	57%	64%	SRBRS	14%	29%	40%	47%
SGBRS	34%	52%	58%	60%	APISA	8%	21%	35%	46%
SHIFTSA	19%	39%	53%	60%	SSM	13%	29%	39%	46%

The order of the algorithms is not the same for the several time limits, and there is no algorithm with a performance more than 90%, although performance values increase with the time limit.

Table 4. Top five algorithms indicators of the 20 seconds limit

Indicators	**HARTG4**	**HARTG1**	**LJFG1**	**SHIFTLS**	**RSLS**
Performance	81%	81%	79%	77%	76%
UBs	45%	36%	32%	31%	27%
States	2.686	2.537	2.421	2.780	3.093
Steps	63	421	402	162	187
Evaluation	85%	82%	80%	91%	93%
Generation	13%	15%	18%	8%	6%
Strategy	2%	2%	2%	1%	1%

We can notice that the performance indicator grows when fewer algorithms are compared, and is interesting that no algorithm archive 50% in Upper Bound. That could indicate that there are classes of instance problems that work better with some algorithms and other problems that work better in the other algorithms, or simply that the algorithms need more time to run.

Another interesting analysis is the number of problems that count to the indicator. A problem could be non-simple and yet not to count because all algorithms obtained the same result. That

means that the instance problem is either very simple or very complicated, but in either case the instance is not good for comparing algorithms.

Table 5. Number of relevant problems

Instance Set	Problems	Counting	%
Elmaghraby	36	14	39%
KSD	2040	849	42%
Tavares	72	66	92%
Alvarez	144	60	42%
Patterson	110	20	18%
JobShop/FlowShop	113	108	96%

The Patterson problems are not very useful for comparing algorithms, and the Tavares problems and JobShop/FlowShop problems are the most useful instance sets.

We finalize the article mentioning the most important conclusions. Was tested a large instance set of RCPSP, that brings the major instance sets together, with large instances added from JobShop and FlowShop and from two generators that have no instance set generated. We believe that in the future larger instances will be needed as well as special cases like totally parallel instances, or almost parallel, that we believe to be the most hard to optimise. A new performance indicator was proposed, with some benefits over standard indicators, allowing a large number of instances that do not contribute to distinguish algorithms to be found. We implemented the main meta-heuristics, and the main neighbour and crossover operators, summing up to a total of 30 meta-heuristics tested. The random generator proposed GBRS was the best sampling method and having only 10% lower than the best meta-heuristic. For short time limits, it was even the best meta-heuristic. The local search with restart archive great results also, but the best meta-heuristic was the genetic algorithms, with the two-point crossover and population configuration used by Hartman.

References

[1] Peter Brucker, Andreas Drexl, Rolf Möhring, Klaus Neumann, Erwin Pesch, “Resource-constrained project scheduling: Notation, classification, models, and methods”, European Journal Of Operational Research (112)1 (1999) pp. 3-41

[2] Sprecher, A. (1996): “Solving the RCPSP efficiently at modest memory requirements”; Manuskripte aus den Instituten für Betriebswirtschaftslehre der Universität Kiel, Nr. 426

[3] Schirmer, A., Riesenberg, S. (1998): “Class-based control schemes for parameterized project scheduling heuristics”, Manuskripte aus den Instituten für Betriebswirtschaftslehre der Universität Kiel, Nr. 471

[4] Rainer Kolisch, “Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation”, European Journal Of Operational Research (90)2 (1996) pp. 320-333

[5] R. Kolisch, A. Sprecher, A.Drexl, “Characterization and generalization of a general class of resource-constrained project scheduling problems”, Management Science 41 (1995) 1693-1703

[6] L. Tavares, J. Ferreira, J. Coelho, “The risk of delay of a project in terms of the morfology of its network”, European Journal Of Operational Research (119)2 (1999) pp. 510-537

[7] F. Della Croce. “Generalized pairwise interchanges and machine scheduling”, European Journal Of Operational Research (83) (1995) pp. 310-319

[8] S. Hartmann and R. Kolish. “Experimental Evaluation of State-of-the-Art Heuristics for the Resource-Constrained Project Scheduling Problem”, 1999 INFORMS spring conference, Cincinatti, 1998

[9] R. Kolish and S. Hartman, “Heuristic Algorithms for Solving the Resource-Constrained Project Scheduling Problem: Classification and Computational Analysis”, Kluwer 1998

[10] S. Hartman, “A Competitive Genetic Algorithm for Resource-Constrained Project Scheduling”, Naval Research Logistics 45:733-750, 1998

[11] Tonius Baar, Peter Brucker, Sigrid Knust, “Tabu Search Algorithms and Lower Bounds for the Resource-Constrained Project Scheduling Problem”, Universitat Osnabruck