New sequential and parallel algorithm for Dynamic Resource Constrained Project Scheduling Problem

André Renato Villela da Silva, Luiz Satoru Ochi
Computing Institute
Federal Fluminense University
Niterói, Brazil
avillela,satoru@ic.uff.br

Abstract

This paper proposes a new Evolutionary Algorithm for the Dynamic Resource Constrained Project Scheduling Problem. This algorithm has new features that get around some problems like premature convergence and other ones. The indirect representation approach was used because it allows the construction of a feasible solution from any input priorities. A parallel version is also proposed, making good use of multicore processors available nowadays. The results of sequential and parallel versions were very significant, improving in almost all ways the best results present in literature.

Keywords: scheduling algorithm; solution representation; evolutionary algorithms; parallel metaheuristics

1. Introduction

Project Scheduling Problems consists of executing a set of tasks over a planning horizon (time units), in order to achieve some objective usually related with the last time unit used, also called makespan.

Often, scheduling problems are resource constrained (RCPSP - Resource Constrained Project Scheduling Problem): to activate a task i, we need to pay a cost c_i (retrieved from an amount of available resources). Therefore, a new set of objective functions can be elaborated mixing the makespan with the amount of resources. There are a lot of modelings for the RCPSP that use both makespan and the available resources. The reader is referred to [3], [4], [5], [7], [8], [9], [10], [11].

Dynamic Resource Constrained Project Scheduling Problem (DRCPSP) was originally proposed in [12]. Its main difference among other scheduling problems is the task component called profit. After a task activation, it generates resources that can be used to activate other tasks.

DRCPSP can be described as follows. Let $G = (V, A)$ be a directed acyclic graph (DAG), where V is the set of vertices and A is the set of arcs that represent the tasks (activities) and their precedences, respectively. Associated to each task i there are an activation cost c_i and a profit p_i (positive integer values). There is also a planning horizon represented by a time interval composed of H time units.

This model has a potential application on industrial production and public service expansion projects which must be done in several stages. Some companies typically expand their services using the profits provided by their own businesses. Therefore, it is very important to select which projects are profitable or not.

There are some concepts of the DRCPSP modeling, such as *Activation* that is the entry of a task i into current partial solution, against payment of a cost c_i. After the activation, a *profit* p_i will be generated at each time unit.

- *Available task*: a task i is available, if all its predecessor tasks are activated or the task has no precedence.
- *Planning Horizon* is a set of time units, $[1..H]$, when the tasks can be activated.
- *Available Resources (Q_t)* are the amount of resources that can be used to activate tasks, at the end of time unit t.
- *Accumulated Profit (P_t)* is the sum of all profits that will be returned at the end of time unit t. These profits will become available resources at time unit $t + 1$.

The objective of the DRCPSP is to maximize the available resources at the end of the planning horizon. In order to earn resources, the only way is by activating some tasks, which generate these resources according to the respective profit of the task (p_i) and the time when the task is activated - a task activated earlier generates more resources. A task remains activated until the end of the planning horizon, thus, the earlier the activation, the greater the amount of generated resources.

Table 1 and Fig. 1 show an example of a scheduling algorithm that uses priority assigned to each task in order to define which one must be scheduled first. Supposing $H = 4$, $Q_0 = 4$, $P_0 = 0$ (simplified notation of Q and P are used in example) and priorities computated just for this example, we can see how a scheduling is generated from a priority list.

At start of time $t = 1$ we have two available tasks 1 and 2 (in gray). Other tasks are unavailable (in black) due to precedence constraints. Task 2 has higher priority and, so,

Table 1. Task's informations used in scheduling example

Task	Priority	Cost	Profit
2	2.23	3	2
3	1.78	4	4
4	1.56	1	2
1	0.98	2	1
6	0.45	4	5
5	0.21	2	3

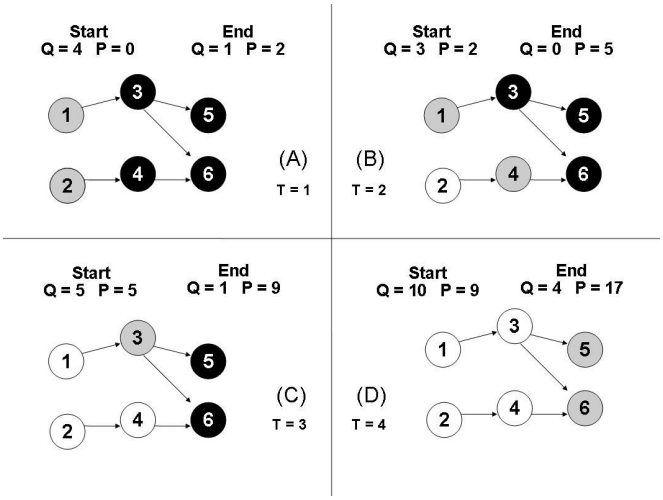

Figure 1. Example of new scheduling algorithm

it is scheduled first. We need to retrieve 3 units of resource from Q (available resources) and add 2 units to P (profits earned so far). Task 1 is the only available task, but its cost is greater than $Q = 1$. Task 1 cannot be scheduled now. Moving to next time $t = 2$, profit P is added to available resources Q, and at this point we have 3 units of resources to spend among tasks. Task 2 is already scheduled (in white) and tasks 1 and 4 are available. Task 4 has higher priority and, for this reason, it is scheduled first. We retrieve its cost from Q and add its profit to P. Task 1 can also be scheduled, because we still have 2 units of available resources. Task 1 is scheduled and we finish this time unit having no resources ($Q = 0$), but profit $P = 5$. Moving to next time, we have $Q = 5$, $P = 5$ and only task 3 is available. We schedule it and finish time unit $t = 3$ having $Q = 1$ and $P = 9$. At the last time unit $t = 4$, tasks 5 and 6 are available. We first schedule task 6 and then task 5. Our scheduling finishes and final solution quality is given by the sum of $Q = 4$ and $P = 17$ (21 units of resource).

The remainder of this paper is organized as follows. Section 2 provides an analysis of literature algorithms and their benefits. Section 3 presents the algorithms proposed to solve DRCPSP. The instances used are outlined in Section 4. Section 5 shows the computational results. The conclusion remarks and a brief discussion about further research are present in Section 6.

2. Literature Review of DRCPSP

In the first work about DRCPSP ([12]), two Evolutionary Algorithms (EA1 and EA2) were proposed. Both used a greedy randomized constructive algorithm, called ADDR. This algorithm makes a list of available tasks for each time unit. Using an α parameter, like in Greedy Randomized Adaptive Search Procedure (GRASPs) algorithms [6], ADDR chooses some tasks to be activated and update the amount of accumulated profit and available resources for this time unit.

A set of Enhancement Techniques (ETs) to be used with ADDR proved to be very important to improve final results. They try to help ADDR when it has to choose which tasks will be scheduled. Basically, these ETs stop ADDR from activating tasks considered non-profitable. Two local searches (LS1 and LS2) were also proposed, but their improvements are tiny and occur at a low frequency in some instances because they do a similar work as the ETs do. If ETs are well adjusted, LS1 and LS2 will hardly do anything promising.

Each individual of the proposed EAs uses a direct representation, which means that each task is represented by a non-negative integer. A value on this array indicates the activation time of the respective task. All evolutionary operators make use of this representation, trying to evolve the individuals.

The second paper about DRCPSP [13] worked with a different metaheuristic approach. Two GRASP versions were proposed: GR1 and GR2. The main reason to change the metaheuristic is the restart provided by GRASP algorithm. If a poor solution is generated and it is hard to improve, the GRASP just discard it at the next iteration. In the other hand, EA1 and EA2 should keep that solution for many generations if no better solutions are found. Actually, GR1 and GR2 use almost all algorithm proposed by [12], but the way they were used by GRASP makes difference. The constructive phase is composed of ADDR and the search phase is composed of LS1, LS2 or LS3 (proposed in this paper). LS3 reconstructs solutions using a good one as a seed. It chooses a time unit and remove all tasks activated after that. Then, it reconstruct a different solution with a more random criterion than used in ADDR. Again, a direct representation was used in this paper.

In [14], a new Evolutionary Algorithm (EA3) was proposed. It fixed the premature convergence problem, present in EA1 and EA2, through a complete removing of the stagnant population and its subsequent recreation by LS3. However, the reconstruction works well only if a reasonable good solution is used. To provide that, a partial solution is generated by CPLEX [1] and passed to EA3 to be improved. This partial solution is a scheduling constrained to half of planning horizon; EA3 continues the scheduling activating more tasks in order to improve the solution value. The

main drawback of that hybrid approach is the utilization of an exact method to solve the partial scheduling. The hybrid algorithm performed very well in instances having large number of tasks, but few time units. In hard instances, CPLEX may find many problems because it has to handle a lot of binary variables.

The most recent paper about DRCPSP [15] used a different solution representation: each task is represented by a real number which indicates the task priority. The scheduling algorithm uses the priority list to choose the tasks that will be activated. This representation scheme always produces a feasible solution for any set of priorities. There is no more need to check the feasibility of solutions generated by crossover or mutation algorithms.

Therefore, EA_priority proposed in [15] is the fastest algorithm proposed so far. The premature convergence present in older EAs is not a serious problem for EA_priority, as is showed in [15], but the crossover operator (average between two priorities from parents) does not allow a wider exploration on the space of solutions.

3. New Algorithms to Solve DRCPSP

We propose a new Evolutionary Algorithm starting from the best EA proposed so far: EA_priority. All features of that version are maintained, but new operators are added to improve the overall performance. This new EA will be called EA_ages.

Solution refinement is a very important operation because, at scheduling time, the priorities of tasks are not concerned about saving resources. After the scheduling, however, it is very simple to check if there are waste resources. Starting from the last time unit, we check if a task i and all its successors generate more resources than the ones used to activate them. If this is true, we keep task i and its successors into analyzed solution, otherwise, the tasks are removed. Every time we remove one or more tasks, we go back to the last time unit and restart the analysis. This refinement is used after each individual creation (initial population), but to put those tasks i down we set their priorities to zero.

The above refinement and previously proposed local searches (LS1 and LS2) hardly use the concept of neighborhood of a solution. They are deterministic improvement methods that are concerned about improving specific parts of solution. This way of improving is objective and fast, but does not explore the space of solutions. The indirect representation allows any priority list to generate a feasible solution. So, we define a neighbor solution S' as a solution similar to S but with two exchanged priorities. Local Search Swap makes a list of all pairs of tasks and randomly sorts it. Then, LS_Swap exchanges the priority of two tasks from the list and schedules using these priorities. If a generated solution S' is better than original solution S, the exchange is maintained, otherwise, it is undone. Moreover, the exchange will only occur with some pair of tasks: two tasks that could be activated in the same time unit (according to graph topology) but were activated in different time units. This criterion was defined to investigate the reason why those tasks were not activated at the same time. It probably occurs because those tasks have very different priorities. In this case, exchanging them might result in a different individual. After the last priority exchange, we always have a solution value equal to or better than the original solution.

LS_Swap works very well on small and medium instances. For large instances, we have to use it at some specific times, otherwise the computational time will be very high. We propose to use LS_Swap especially when a new best solution is found in order to explore the neighborhood of this new solution.

Crossover and mutation are the same proposed by EA_priority: crossover generates a priority tasks list computing the average priority for each priority from parent tasks; mutation randomly changes the priority of a task.

Generated offspring are often similar to their parents (due to crossover computation) and mutation cannot improve them because randomly changes are not enough to substantially modify the stagnant population.

This convergence is very common but undesirable in metaheuristic approaches. To avoid it two mechanisms are proposed. If thirty offspring populations are generated without improving the best solution, we explore the neighborhood of all solutions in subpopulation class A, composed of 20% best solutions, using LS_Swap on them. These solutions have very good fitness, but they may be in different places of the solution space. If this is true, exploring these solutions might cover a great area of this solution space and we have good probability to find a better solution. This Large Exploration may take some time and it is used only when population is stagnated.

In some cases, the best solutions might be very similar and applying LS_Swap on them might not find any better solutions. In this case, there is nothing to be done. We need to discard the whole stagnant population and restart using a new one. This new population must use the best priority list generated so far. Therefore, the best solution is used as a seed in order to generate different and good solutions. We propose to add a random factor λ to each priority of best solution. Generated solutions are near the best one in space of solutions and future operators may intensify the search of solutions in this subspace. The proposed Evolutionary Algorithm tries to keep alive strong individuals and make them continue their genotype in the following generations. The generations between two consecutive reconstructions compose an *Age*. Population Reconstructions can be understood as natural disasters that extinguish almost all living beings. Next, one of them starts the repopulation of environment. Every *Age* will only begin after some generations without improvements.

The last difference between EA_ages and EA_priority is the stop criterion. EA_priority has a fixed number of generations to provide a final solution. EA_ages also has a number of generations, but it is not fixed. Anytime it generate a new best solution the number of remaining generations is added to β, where β is computated by Eq. 1.

$$\beta = \left\lceil \frac{generation}{20} \right\rceil + 4 \quad (1)$$

$Generation$ corresponds to the generation number when the new best solution is found. This formula was defined based on preliminary tests made with many instances. These extra generations are very useful for the new EA because it can work harder on instances where it is easy to find better solutions. Moreover, if EA_ages finds a better solution in later generations it must need more generations to keep improving than if it find the best solution in earlier generations. So, the Eq. 1 provides more generations to new EA if it demonstrate to be working well. Fig. 3 shows the evolutionary scheme.

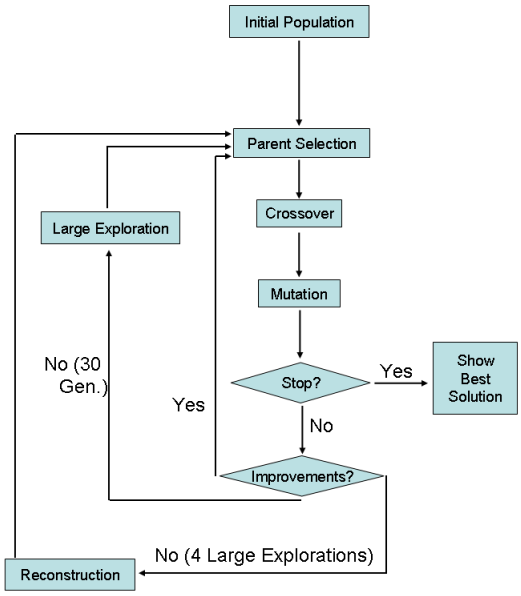

Figure 2. Evolutionary scheme of new proposed EA_ages

In Fig. 2, we can see that EA_ages starts with an Initial Population (100 individuals) randomly generated. Refinement is applied on each individual of this population which is sorted by individual fitness. Current population is divided into three classes: class A - composed of 20% best individuals; class C - composed of 20% worst individuals; class B - remaining individuals. Parent Selection operator takes one random parent from class A and other random parent from class B.

Crossover uses these two parents to generate two new individuals as follows. For each task priority, we compute the average parent priority and add a random number to provide two slightly different offspring. After generating 100 offspring individuals, we use the elitist criterion to define which individuals will be discarded and which ones will compose the next generation.

Mutation operator consists of changing the tasks priority of an individual. Each one has 5% chance of being mutated. If it is chosen to be mutated, we add a random number into each task priority just as the second part of crossover operator. Mutated individual is kept in the population even it is worse than the original one. After mutation, the remaining population is sorted.

When these evolutionary operators end, we need to know if the stop criterion was met. If this is true, evolutionary algorithm stops and shows the best solution found so far. Otherwise, we analyze improvements obtained at this generation as follows.

- There is an improvement: extra generations are given to generation limit (stop criterion) and Parent Selection takes place for the next generation;
- There is no improvement for less than thirty generations: no extra generations are given and Parent Selection also takes place;
- The last improvement occurred thirty or more generations ago: Large Exploration tries to improve any of class A individuals by applying LS_Swap on them. Next, Parent Selection starts a new generation process. The next Large Exploration will only happen if there are more thirty generations without improvements.
- Four Large Explorations were done and there were no improvements: it is time to focus on the best solution neighborhood. Current population is discarded and a new one is created based on the best individual. Reconstruction makes previous *Age* finishes and starts a new one. It is very similar to restarting the evolutionary algorithm, but instead of using an initial random population, we use a population composed of best solution neighbors. However, the generation limit is not changed. For example: if we finish an *Age* having only five remaining generations to meet the stop criterion, the next *Age* will have only five generations to work. If this *Age* not improves its best individual in five generations, the algorithm finishes.

4. Instances

There are some instances being used since the first paper regarding the DRCPSP. They are compiled in a project, called LABIC Project from Universidade Federal Fluminense [2]. Basically, instances have data about cost and profit of each task and precedence among them. Initial resources and the size of planning horizon are also present. No other instances are known for this problem.

Hardness of a instance may be defined based on the number of tasks and the size of planning horizon. For example, an instance u with 50 tasks is easier than an instance v with 100 tasks. An instance w with 4 time units is easier than an instance y with 7 time units. This last situation occurs because few time units will probably allow the activation of fewer tasks than an instance with more time units: the larger the planning horizon, the greater the accumulated profit and the greater the number of activated tasks.

Instances with few time units are very suited for exact and hybrid methods because they can find good solutions in small amount of time. The hybrid metaheuristic CPLEX+EA3 [14] worked well on these instances and achieved good results. But, the most challenging instances are used in other papers and have up to 1000 tasks. Planning horizon size is computated as square root of number of tasks. In these instances, the optimal value is not known and both heuristics and exact methods are still ineffective because they spend much computational time.

To validate the new proposed EA a set of twenty instances were taken from LABIC Project. These instances have up to 1000 tasks and some of them were used in the paper which presented the most effective algorithm proposed so far: EA_priority.

5. Computational Results

Computational environment used in this work is the same used in [15]: Pentium 4 HT 3.0 GHz, 1GB RAM, Windows XP Professional SP2, source code written in C.

In the first test, we executed the EA_ages 30 times for each instance. Average results and computational times are shown in Table 2.

Table 2 shows significant improvements obtained by EA_ages. Its main strong points are LS_Swap, Extra Generations and *Ages* scheme. The first one is very complicated to measure because at the beginning of EA it improves the individual by 10% but as generations passes, it cannot keep that performance. The potential of Extra Generations can be seen in Table 3. This Table shows the improvement got after the generation 50 (initial stop criterion) as well as the average number of total generations.

Table 3 shows that stopping EA at generation 50 is the same as finishing it prematurely. Extra Generations improve the solution about 3.6%, in average. The Eq. 1 used to give Extra Generations was defined according to preliminary tests. Certainly it would be better if the Eq. 1 was adjusted according to each instance characteristics, but it is a very hard work and would demand a large amount of computational time.

Table 2. Average Results of Evolutionary Algorithms

Instance	EA_age	EA_prior.	Improv.	Time(s)
100a	304	303	0.3%	0.9
200a	632	636	-0.6%	5.0
300a	2004	1958	2.3%	56.4
400a	6908	5962	13.7%	226.0
500a	13523	11954	11.6%	223.0
600a	9358	8616	7.9%	578.9
700a	28500	26217	8.0%	527.5
800a	35001	32354	7.6%	859.7
900a	34355	29912	12.9%	673.1
1000a	66399	63512	4.3%	1135.0

Table 3. Improvement after 50 generations

Instance	Improvement	Total Gens.
100a	0.0%	50
200a	0.0%	66
300a	0.9%	174
400a	2.7%	218
500a	1.9%	176
600a	7.2%	206
700a	4.4%	169
800a	2.6%	133
900a	5.6%	157
1000a	3.2%	143

The last strong point of proposed EA is the *Age* scheme. In this scheme, when EA realize that it stagnate, the whole population is discarded and a new one is created using the best individual so far as a seed individual. The objective is to intensify the operations into the best individual neighborhood. We executed EA_ages in some instances with 3600 seconds as stop criterion. Table 4 shows the fitness of the best individual at each *Age* ending. Three instances (400a, 500a and 600a) were used as example.

Table 4. Average Results of Evolutionary Algorithms

Age	400a	500a	600a
1	7010	13787	9488
2	7066	13870	9685
3	7066	13945	9704
4	7067	13978	9736
5	7068	14000	9773
6	7069	14030	9773
7	–	–	9799

It is important to remark that an *Age* finishes because the algorithm does not have any expectation to avoid the convergence (stagnate) state. In other words, we have no guarantee that keeping that population will provide anymore improvements. So, the *Age* scheme is an attempt to focus on an individual that worked well. We did not get important

improvements, but the algorithm could make a better use of time limit.

Other operators like crossover, mutation and parent selection are the same proposed in [15] and, therefore, their benefits were already known.

We also propose a parallel version of EA_ages. In this version, evolutionary operators have their workload divided into processing cores of a multicore processor. These processors have the ability of executing simultaneous threads sharing the same memory space without exchanging messages among them. When Crossover, Mutation, Large Exploration and Population Reconstruction take place, some threads work only with a part of individuals in question. For example: if we are using two cores in Population Reconstruction, the first thread reconstructs 50 individuals and the second one other 50 individuals. After that, the population is sequentially sorted and algorithm continues until one of mentioned operator. At that point, the workload is divided again among threads (cores). Each algorithm step apart from the four evolutionary operators is sequentially executed.

This parallelism is very interesting because almost all recent manufactured processors are multicore and operating systems have a load balance feature that allow each thread to execute in a different processing core. Obviously, operating system kernel and background applications compete for processor cores. So, using all cores may not provide a very good speedup, but it is better than using a single core.

The multicore environment used in the following test is a Intel Core2 Quad (Q6600) 2.4 GHz (each core), 4GB, Linux 2.6.24-19. We made two tests: the first one measured the average speedup of using two or four threads instead of using a single one. Table 5 shows the relative speedup of two and four threads version of EA_ages using the same sequential version stop criterion (50 plus extra generations).

Table 5. Relative Speedup of multi-thread version

Instance	2 threads	4 threads
300a	45.3%	113.1%
400a	80.2%	165.3%
500a	85.1%	127.3%
600a	79.9%	182.6%
700a	67.3%	117.6%
800a	92.6%	139.0%
900a	85.5%	107.3%
1000a	78.8%	130.2%

We can see in Table 5 that the two threads version got a reasonable speedup, but the same did not occur with four threads.

The reason may be the very fine granularity of EA_ages parallelism. The parallel operators end their execution very fast because they are responsible for few individuals. Sequential steps like sorting population and evaluation of a possible new best individual take almost the same time of parallel steps. So, there is only one core being used in almost half of all processing time.

However, there is a positive characteristic in this parallel approach: the parallel version generates good solution faster. Table 6 shows the best solution value found by each version at some specific times (10, 100, 1000 and 3600 seconds).

Table 6. Evolution of best solution using threads

Instance	Threads	10s	100s	1000s	3600s
400a	1	6519	6979	7025	7025
	2	6424	7000	**7100**	7167
	4	**6762**	**7009**	7094	**7174**
700a	1	25236	28124	29413	29796
	2	25916	28277	29661	**30108**
	4	**26214**	**28960**	**29674**	30070
1000a	1	46618	64885	67978	68765
	2	**59360**	64904	68332	69431
	4	58992	**65795**	**68710**	**69494**

In the three instances chosen for this test, a parallel version using two or four threads always had the best solution. Moreover, the four threads version was a little better than other version. Although it was an expected result, the poor speedup of this version could have disrupted it. Of course it is possible to use more than four cores (or processors) with the parallel version just spreading individuals as equitable as possible among processing units. If sequential parts of algorithm like individual sorting and natural selection become parallel, certainly it will be very interesting too.

6. Concluding Remarks

This paper presented a new Evolutionary Algorithm for the Dynamic resource Constrained Project Scheduling Problem (DRCPSP). This EA_ages uses a indirect representation that can generate a feasible schedule from any priority list.

EA_ages got around the population stagnate state using an *Age* scheme that destroy all individuals and reconstruct new ones from a best individual. Other strong points like Extra Generations and the local search LS_Swap also contributed to improve the literature results. Extra Generations by themselves were responsible for a 3.6% improvement; LS_Swap made worthwhile explorations in the best solution neighborhood, mainly at initial generations.

A parallel version of EA_ages was also proposed using two or four threads in a multicore processor. The speedup would be better if a not so fine granularity scheme was used. However, there is a good benefit of using the parallel versions: they can find better solution in smaller amount of time. According to the presented results, the parallel EA_ages is the algorithm that best uses the computational power of present-day processors in order to find better solution for DRCPSP.

The literature of Evolutionary Algorithm provides a wide set of evolutionary operators. If we assume that each one

can be well suited for specific instances, we can imagine a parallel algorithm where each population uses a different configuration of operator. Each population can be improved in a different way, exploring better the solutions space. Each population can be executed in a different core, improving the granularity of Evolutionary algorithm and, consequentially, its speedup.

Another promising technique consists of using exact methods with the indirect representation. If it is well adjusted, the exact method might handle only a small part of the whole scheduling. The indirect representation has the ability to keep the priorities of a good partial schedule by crossover operator computation.

Acknowledgment

This work was partially supported by CNPq - grant 141074/2007-8

References

[1] Cplex software web page http://www.ilog.com/products/cplex.

[2] Labic project web page http://www.ic.uff.br/~labic.

[3] C. Boeres, E. Rios, and L. S. Ochi. Hybrid evolutionary static scheduling for heterogeneous systems. In *Proc. of the IEEE Conf. on Evolutionary Computation (CEC2005)*, pages 1929–1937, Edinburgh (Scotland), 2005.

[4] K. Bouleimen and H. Lecocq. A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem and its multiple mode version. *EJOR - European Journal of Operational Research*, 149:268–281, 2003.

[5] P. Brucker, A. Drexl, R. Mohring, K. Neumann, and E. Pesch. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research*, 12:3–41, 1999.

[6] T. A. Feo and M. G. C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization 6*, pages 109–133, 1995.

[7] J. Gonçalves, J. Mendes, and M. G. C. Resende. A hybrid genetic algorithm for the job shop scheduling problem. *European Journal of Operational Research*, 167:77–95, 2005.

[8] J. Gonçalves, J. Mendes, and M. G. C. Resende. A genetic algorithm for the resource constrained multi-project scheduling problem. *EJOR - European Journal of Operational Research*, 189:1171–1190, 2008.

[9] S. Liu and C. Wang. Resource-constrained construction project scheduling model for profit maximization considering cash flow. *Automation in Construction*, 17:966–974, 2008.

[10] J. Remy. Resource constrained scheduling on multiple machines. *Information Processing Letters*, 91:177–182, 2004.

[11] M. Rogalska, W. Bozejko, and Z. Hejducki. Time/cost optimization using hybrid evolutionary algorithm in construction. *Automation in Construction*, 18:24–31, 2008.

[12] A. R. V. Silva and L. S. Ochi. A dynamic resource constrained task scheduling problem. In *Proc. of Latin-Ibero-American Congress on Operations Research (CLAIO)*, Montevideo (Uruguay), November.

[13] A. R. V. Silva and L. S. Ochi. Effective grasp for the dynamic resource-constrained task scheduling problem. In *Proc. of International Network Optimization Conference (INOC)*, Spa (Belgium), April 2007.

[14] A. R. V. Silva and L. S. Ochi. A hybrid evolutionary algorithm for the dynamic resource constrained task scheduling problem. In *Proc. of the International Workshop on Nature Inspired Distributed Computing (NIDISC)*, LongBeach (USA), March 2007.

[15] A. R. V. Silva, L. S. Ochi, and H. G. Santos. New effective algorithm for dynamic resource constrained project scheduling problem. In *Proc. of International Conference on Engineering Optimization (ENGOPT)*, Rio de Janeiro (Brazil), June 2008.