Solving Resource-Constrained Project Scheduling Problems with CP Optimizer

Published on March 22, 2019 Edit article View stats

Philippe Laborie
Software Developer: Operations Research, Artificial Intelligence, Optimization, Planning & Scheduling

3 articles

The Resource-Constrained Project Scheduling Problem (**RCPSP**) is a classical problem among the myriad scheduling problems studied both in academia and in industry. After describing the problem, I show here how it can easily be modeled and efficiently solved using the **CP Optimizer** engine of IBM ILOG CPLEX Optimization Studio.

The problem

The RCPSP is undoubtedly one of the most widely studied scheduling problems in the literature.

Informally, an RCPSP considers a set of resources of limited availability and a set of tasks of known durations and known resource requirements, linked by precedence constraints. The problem consists of finding a schedule of minimal duration by assigning a start time to each task such that the precedence constraints and the resource limits are respected [AD-08].

More formally, the RCPSP can be stated as follows. Given:

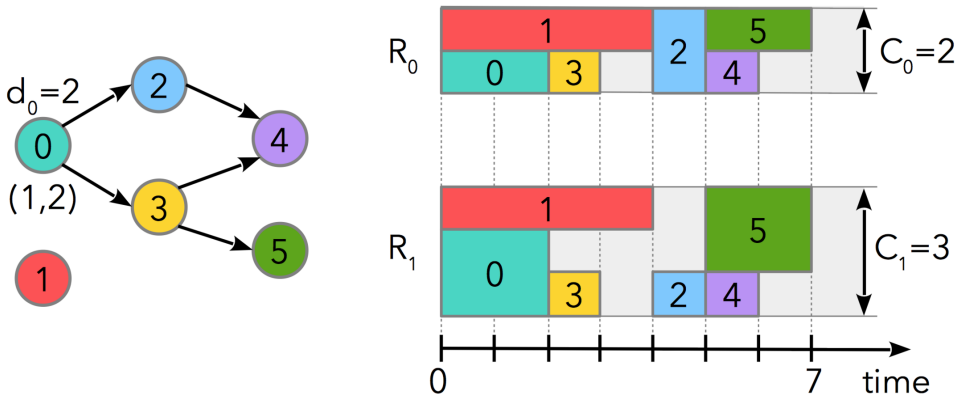
- A set of n tasks with given durations
- A set of m resources with given capacities,
- A network of precedence constraints between the tasks, and
- For each task and each resource the amount of the resource required by the task over its execution

The goal of the RCPSP is to find a schedule satisfying all the constraints (i.e. precedence and resource capacity constraints) whose makespan (i.e. the time at which all tasks are finished) is minimal.

An Example

Let's illustrate the problem on a small instance with 6 tasks and 2 resources in the figure below. For consistency with the data, we index the tasks and resources from 0.

- Resource R0 has capacity 2
- Resource R1 has capacity 3
- Task 0 has a duration of 2 and requires 1 unit of R0 and 2 units of R1
- Task 1 has a duration of 4 and requires 1 unit of R0 and 1 unit of R1
- Task 2 has a duration of 1 and requires 2 units of R0 and 1 unit of R1
- Task 3 has a duration of 1 and requires 1 unit of R0 and 1 unit of R1
- Task 4 has a duration of 1 and requires 1 unit of R0 and 1 unit of R1
- Task 5 has a duration of 2 and requires 1 unit of R0 and 2 units of R1
- Task 0 must execute before task 2 and task 3
- Task 2 must execute before task 4
- Task 3 must execute before task 4 and task 5



The left side of the figure shows the precedence constraints between the tasks. The right side illustrates a solution with a makespan of 7.

Given the small size of the problem we can easily prove that this solution is optimal, that is, there does not exist any feasible solution with a makespan strictly smaller than 7. I give a proof at the end of this post.

Complexity and Relevance of the Problem

The RCPSP is one of the most intractable combinatorial optimization problems and belongs to the class of problems that are **NP-hard** in the strong sense [GJ-79]. Some instances with only 60 tasks, generated more than 20 years ago [KS-96] and massively studied by the scheduling community, are still not solved to optimality as of today.

Even if the RCPSP is far from capturing all the complexity of real life scheduling problems

(more on this later), it is relevant in many scheduling domains. Here is an non-exhaustive list:

- In the construction industry for scheduling large projects
- In universities and training centers for scheduling courses on teachers, rooms, facilities, etc.
- In industrial assembly (assembly of aircraft, boats, etc.) for scheduling assembly tasks with resources like manpower, equipment, etc.
- In maintenance scheduling (aircraft, nuclear plants, etc.)
- In airports for scheduling the landing of aircrafts, the ground operations, the gates, etc.
- In ports for scheduling the movement of ships, the loading of containers onto ships, etc.
- In production scheduling to schedule production activities on machines
- In aerospace systems for scheduling the communication with satellites
- In facility management for scheduling the activity of staff
- In distributed computer systems and cloud computing for scheduling jobs on the different nodes / CPUs
- In hospitals for scheduling patient visits and surgeries

Existing Approaches and Benchmarks

Many approaches have been proposed for solving the RCPSP :

- Exact approaches using Mathematical Programming. Many formulations of the RCPSP as an **Integer Linear Program** (MIP) have been proposed in the literature. They are discussed in [Te-15]. These approaches are applied only to small problems (up to 30/60 tasks) and do not scale well to larger ones. This can be partially explained by the size of the formulation that usually grows as the cube of the number of tasks.
- Exact approaches using **Constraint Programming** and **SAT** hybrids [AB-11] [SF-13] [VL-15]
- Exact approaches using problem specific algorithms
- **Meta-heuristics** (Priority Rules, Genetic Algorithms, Tabu Search, Simulated Annealing, Particle Swarm Optimization, Ant Colony Optimization, etc.)

The table below gives a short description of the main existing benchmarks for the RCPSP.

Benchmark	Reference	#Instances	#Tasks	#Resources	Task dur.	Makespan
Patterson	[Pa-84]	110	[5-49]	[1-3]	[0-9]	[6-83]
AT	[AT-89]	144	[25-101]	6	[1-12]	[37-574]
KSD30	[KS-96]	480	30	4	[1-10]	[34-129]
KSD60	[KS-96]	480	60	4	[1-10]	[44-154]
KSD90	[KS-96]	480	90	4	[1-10]	[60-175]
KSD120	[KS-96]	600	120	4	[1-10]	[66-288]
BL	[BL-00]	39	[20-25]	3	[1-5]	[13-33]
PACK	[CN-01]	55	[15-33]	[2-5]	[1-19]	[20-138]
RG300	[DV-07]	300	300	4	[1-10]	[83-1774]
RG30	[VC-08]	1800	30	4	[1-10]	[48-173]
KSD15-D	[KA-11]	480	15	4	[1-250]	[187-999]
PACK-D	[KA-11]	55	[15-33]	[2-5]	[1-1138]	[644-3694]

For instance the ‘Patterson’ benchmark contains 110 instances with a number of tasks ranging from 5 to 49 and a number of resources ranging from 1 to 3. The duration of tasks ranges from 0 to 9 and the typical makespan from 6 to 83.

The existing benchmarks are interesting because they have driven the huge progress made on the RCPSP beginning more than 30 years ago and they have been cleverly designed so as to cover different topologies of precedence constraint graphs and combinations of resource requirements. Still, they suffer from an important limitation: the number of tasks ranges from 5 to 300 whereas real life scheduling problems are often much larger.

Due to these size limitations, we decided to complement the existing benchmarks with larger instances that, we think, are more representative of current RCPSPs found in industry. For example, one of our customers in the aircraft assembly domain need to solve a problem involving several hundred thousands of assembly tasks. This type of problem is typically 1000-10000x larger than the instances in the existing benchmarks! Here are the main features of the new benchmark we created (available **here**).

Benchmark	#Instances	#Tasks	#Resources	Task dur.	Makespan
LARGE	30	[500-500,000]	[8-79]	[1-1000]	[50,000-8,000,000]

The largest instance contains 500,000 tasks, 79 resources, 4,740,783 precedence constraints and 4,433,550 resource requirements.

About CP Optimizer

CP Optimizer is available in **CPLEX Optimization Studio**. It provides a modeling

language for Combinatorial Optimization Problems that extends Integer Linear Programming (and classical Constraint Programming) with some algebra on *intervals* and *functions* allowing compact and maintainable formulations for complex scheduling problems.

CP Optimizer has an automatic solution search process. This search of CP Optimizer is :

- *Complete* : it provides optimality proofs and lower bounds
- *Anytime* : feasible solutions are in general produced quickly
- *Efficient* : it is usually competitive with problem-specific algorithms on classical problems, and the performance improves from version to version
- *Scalable*: the engine can handle problems involving up to several hundreds of thousands of tasks
- *Parallel*: it can exploit all the compute cores available on the machine
- *Deterministic*: solving the same problem twice on the same machine will produce the same result

You can get a free and unlimited version of IBM ILOG CPLEX Optimization Studio, including CP Optimizer, through **Academic Initiative** if you are in academia. CP Optimizer is available in **Python**, Java, C++ (native code) and OPL. I will use Python here for illustration.

An overview of CP Optimizer (modeling concepts, applications, examples, tools, performance, etc.) is available **here**.

The CP Optimizer RCPSP Formulation

The formulation here is designed to read data in a JSON format. For instance, the data of the illustrative example above reads:

```
{
  "ntasks"       : 6,
  "nresources"   : 2,
  "capacities"   : [ 2,3 ],
  "durations"    : [ 2,4,1,1,1,2 ],
  "requirements" : [ [ [0,1],[1,1],[2,2],[3,1],[4,1],[5,1] ],
                     [ [0,2],[1,1],[2,1],[3,1],[4,1],[5,2] ] ],
  "successors"   : [ [0,2],[0,3],[2,4],[3,4],[3,5] ]
}
```

Here is all the Python code you need to write in order to (1) read the data, (2) formulate the RCPSP in CP Optimizer and (3) solve the problem using the automatic search. The explanation of the formulation is given after the code.

```
# 1. READING THE DATA

import json
with open("instance.json") as file:
    data  = json.load(file)
n = data["ntasks"]
m = data["nresources"]
C = data["capacities"]
D = data["durations"]
R = data["requirements"]
S = data["successors"]
N = range(n)
M = range(m)

# 2. MODELING THE PROBLEM WITH CP-OPTIMIZER

from docplex.cp.model import *
model = CpoModel()

# Decision variables: tasks
task = [interval_var(size = D[i]) for i in N]

# Objective: minimize project makespan
model.add(minimize(max(end_of(task[i]) for i in N)))

# Constraints: precedence between tasks
for [i,j] in S: model.add(end_before_start(task[i],task[j]))

# Constraints: resource capacity
for j in M: model.add(sum(pulse(task[i],q) for [i,q] in R[j]) <= C[j])

# 3. SOLVING THE PROBLEM

sol = model.solve(TimeLimit=300,trace_log=True,LogPeriod=1000000)
```

Yes, that's all!

As stated before, the goal of the RCPSP is to find a schedule satisfying all the constraints (i.e. precedence and resource capacity constraints) whose makespan (i.e. the time at which all tasks are finished) is minimal.

The decision variables of the problem are the start and end time of the tasks and, as often in scheduling problems, a task represents an interval of time. CP Optimizer introduces the concept of *interval variable* for representing any interval of time during which a particular property holds (for example here, the property is that a particular task is executing) and whose position in time is part of the decisions of the problem. Like any decision variable in a Combinatorial Optimization Problem, an *interval variable* is an unknown of the problem that will be assigned a particular value in a solution. In case of an *integer variable* (say *x*), the value in a solution is an *integer* v (say: *x=3*). Similarly, in case of *interval variable*, the value in a solution is an *interval [s,e)* of integers (for example *x=[5,7)* if x represents Task 5 in the solution to the illustrative instance on the right side of the figure) where *s* is the start value of the interval and *e* its end value (by convention, intervals are always closed on the left and open on the right). The value *e-s* is called the *length* of the interval.

Note that interval variables are more expressive than what we illustrate here on the RCPSP, in particular there is a very important notion of *optionality* and also a notion of interval *size* that is usually different from the notion of interval *length*. For the RCPSP, we do not need the notion of *optionality* and we can assume that the notion of *size* is the same as the one of *length*.

The first thing the RCPSP formulation does is create an array of n interval variables *task[i]* for i in N, with interval variable *task[i]* being of (known) size *D[i]* that represents the duration of the task. These interval variables *task[i]* are the only decision variables of the problem.

One very important point I would like to stress here: you should **not** consider an interval variable as being the juxtaposition of two integer variables (one for the start, another for the end). First, because it is not implemented this way under the hood in the engine but also, more importantly, because if you consider it this way, you will not have the right mindset for designing efficient scheduling models with CP Optimizer. An interval variable is an **atomic decision variable** in the problem. CP Optimizer provides a rich set of constraints that you can directly impose on the interval variables of the problem and even if CP Optimizer allows mixing interval variables and integer variables in the same model, most of the time, a scheduling problem can be modeled by using interval variables only (this is true for the RCPSP but also, typically, for all its extensions I will mention later).

Of course, it is sometimes necessary to access some endpoints of the intervals in the model as for instance here for representing the makespan of the schedule which is the maximum of all the end values of the intervals. Some integer expressions are available for doing that, like *expr=end_of(x)* on an interval variable x that returns the end value of the interval. But here, *expr* is an integer expression that will have its value entirely determined by the value of the decision variable in its scope (here interval variable x), just like you could use a square expression *expr=x^2* that functionally depends on an integer decision variable x.

This explains the formulation of the objective function to be minimized that reads: *max([end_of(task[0]), end_of(task[1]), ..., end_of(task[n-1])])*.

Precedence constraints are formulated using *end_before_start* constraints between interval variables.

Last, but not least, resource capacity constraints are formulated using the CP Optimizer concept of *cumul function expression*. A cumul function expression is an expression built on interval variables. Like with any expression, the cumul function expression will be fixed when all the interval variables in its scope are fixed. And, as the name suggests, the value of a cumul function expression is a *function*. For example, the value of a cumul function expression *f=pulse(x,2)* on interval variable *x=[s,e)* is a stepwise function whose value *f(t)* is *2* for t in *[s,e)* and *0* everywhere else. CP Optimizer provides a simple linear algebra on cumul functions (so you can write *f=pulse(x,2)+pulse(y,1)+...*) and a set of constraints that operate on cumul functions. One of these constraints limits the maximal value of a cumul function (for instance *f<=3*). So you now understand how resource capacity is modeled in the RCPSP formulation: for each resource j, a constraint *f <= C[j]* is imposed on a cumul function *f* that represents the evolution over time of the cumulated usage of resource j by the tasks *task[i]*, that is: *f=pulse(task[0],q0)+ pulse(task[1],q1)+...+pulse(task[n-1],qn-1)*.

Two comments for MIP fanatics:

1. Unlike in time-indexed MIP formulations, we never need to compute a horizon of the schedule for the problem. In fact when an interval variable is defined in CP Optimizer, by default, its minimal start value is 0 and its maximal end value is IntervalMax=4503599627370494 (on 64-bit platforms). Such a large horizon is not a problem because the size of the model does not depend on the time-unit or on the horizon: the number of objects in the model grows linearly with the size of the data. And, under the hood, the engine does not need to enumerate time values which explains why CP Optimizer is not afraid of RCPSP instances with minimal task length 1 and makespan value in the order of 8,000,000 like in some of the new instances we created.

2. Alternative MIP formulations for the RCPSP usually grow cubically with the number of tasks. This becomes unwieldy if you want to address RCPSPs with even 100 tasks.

Once the problem has been formulated in CP Optimizer, the easiest part (for you as a user) is to run the automatic search by calling model.solve(). Here we will solve the problem using a time limit of 5 minutes.

Here is a search log for an easy instance of the KSD120 benchmark for which we verify on the first line of the log that indeed, only 120 decision variables are used in the formulation of this problem.

```
! ----------------------------------------------------------------------------
! Minimization problem - 120 variables, 181 constraints
! TimeLimit            = 300
! LogPeriod            = 1000000
! Initial process time : 0.00s (0.00s extraction + 0.00s propagation)
!  . Log search space  : 828.8 (before), 828.8 (after)
!  . Memory usage      : 838.2 kB (before), 838.2 kB (after)
! Using parallel search with 8 workers.
! ----------------------------------------------------------------------------
!          Best Branches  Non-fixed    W       Branch decision
                       0         120                 -
+ New bound is 114
! Using iterative diving.
*           141      309  0.02s        1      (gap is 19.15%)
*           139      549  0.02s        1      (gap is 17.99%)
*           114      789  0.02s        1      (gap is 0.00%)
! ----------------------------------------------------------------------------
! Search completed, 3 solutions found.
! Best objective         : 114 (optimal - effective tol. is 0)
! Best bound             : 114
! ----------------------------------------------------------------------------
! Number of branches     : 80435
! Number of fails        : 794
! Total memory usage     : 6.5 MB (6.4 MB CP Optimizer + 0.1 MB Concert)
! Time spent in solve    : 0.05s (0.04s engine + 0.00s extraction)
! Search speed (br. / s) : 1608701.5
! ----------------------------------------------------------------------------
```

The Results

The best known solutions for the different benchmarks are available from different sources:

- The **PSP-LIB** (Technische Universität München) that compiles more than 20 years of results on the KSD benchmarks

- The page of the **OR&S Research Group** (Ghent University) with results on the RG benchmarks
- The **RCPSP page** (University of Melbourne) that provides results reported in [SF-13] on most of the RCPSP benchmarks

We compiled for each instance, the best known results from these 3 sources. We call *Virtual Best Solver* a (theoretical) solver that can run in parallel (at no extra cost) all the approaches previously experimented on a given RCPSP instance and return the answer from the best.

We first report the results using the above 4 lines long CP Optimizer formulation with a time limit of 5 minutes on all 5203 instances of the existing benchmarks. CP Optimizer immediately finds a first feasible solution to all the instances (in less than 0.08s).

Unless stated otherwise, the results were computed using the version 12.9 of CP Optimizer on a MacBook Pro, 2.5 GHz Intel Core i7, 16GB RAM.

Benchmark	Size	#Instances	Average dist. to best known solution	Proportion of optimality proofs
Patterson	[5-49]	110	0.00%	100.00%
AT	[27-103]	144	-0.42%	74.31%
KSD30	30	480	0.00%	100.00%
KSD60	60	480	0.11%	88.33%
KSD90	90	480	0.33%	81.46%
KSD120	120	600	1.09%	46.00%
BL	[20-25]	39	0.00%	100.00%
PACK	[15-33]	55	-0.02%	67.27%
RG300	300	300	1.35%	6.04%
RG30	30	1800	-0.34%	93.44%
KSD15-D	15	480	0.00%	100.00%
PACK-D	[15-33]	55	0.00%	65.45%

The column *Average dist. to best known solution* averages over all the instances i of the benchmark the relative distance between the makespan obtained by CP Optimizer ci and the best known makespan bi (these best known solutions have usually been obtained using many different RCPSP-specific resolution algorithms): $1/n * sum((ci-bi)/min(ci,bi))$. Thus, a negative value means that on average, CP Optimizer finds better solutions than the best known solutions of the benchmark. We see that after 5mn of computation, CP

Optimizer is on par with, and sometimes outperforms, the *Virtual Best Solver*. The last column shows the proportion of instances on which CP Optimizer was able to find and prove optimality in less than 5mn.

We can compare more specifically with two approaches that are considered to be the state-of-the art exact algorithms on RCPSP problems, namely Lazy Clause Generation [SF-13] and SAT Modulo Theories [AB-11] on the benchmarks for which results of these approaches are available. Note that for the proportion of optimality proofs, we run CP Optimizer with a larger 10mn time limit similar to [SF-13] and increased the focus of the automatic search on optimality proofs (parameter FailureDirectedSearchEmphasis). The comparison should of course be handled with care as the results were not computed on the same machines.

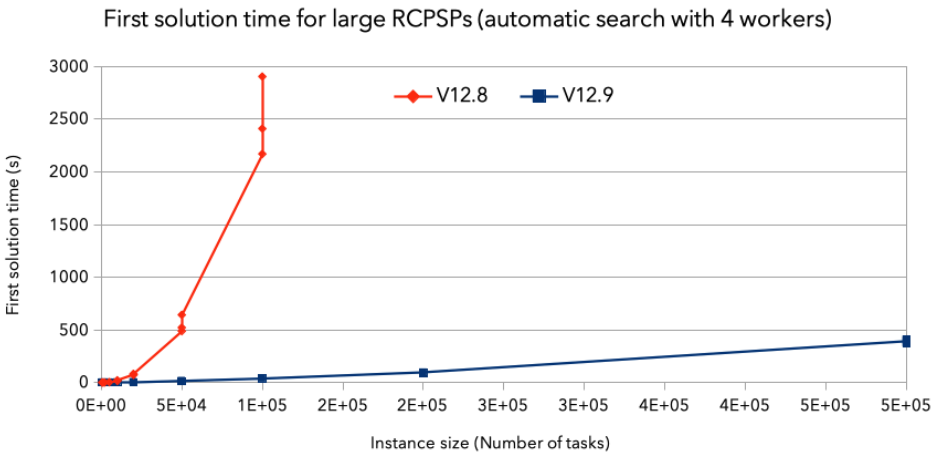
Benchmark	Average dist. to best solution of [SF-13] 300s	Proportion of optimality proofs CP Optimizer 12.9 600s	Proportion of optimality proofs LCG [SF-13] 600s	Proportion of optimality proofs SMT [AB-11] 500s
Patterson	**0.00%**	**100.00%**	**100.00%**	
AT	**-0.45%**	77.08%	**89.58%**	
KSD30	**0.00%**	**100.00%**	**100.00%**	**100.00%**
KSD60	**-0.18%**	89.17%	**90.00%**	
KSD90	**-0.48%**	**83.75%**	83.33%	
KSD120	**-1.74%**	**47.33%**	47.17%	
BL	**0.00%**	**100.00%**	**100.00%**	
PACK	**-0.02%**	**74.55%**	70.91%	65.00%
KSD15-D	**0.00%**	**100.00%**	**100.00%**	**100.00%**
PACK-D	**0.00%**	**70.91%**	67.26%	43.00%

Here also we see that the results for optimality proofs are on par with the best approaches. The first column shows the average distance of the CP Optimizer solutions (after 5mn) to the ones of [SF-13] (after 10mn). We see that CP Optimizer's solutions are always better on average, meaning that on the problems for which [SF-13] does not prove optimality, CP Optimizer usually finds in 5mn a better quality solution than [SF-13] does in 10mn.

When we focussed on the KSD benchmarks (which are by far the most studied ones), during this study we managed to close and improve the bounds (LB: lower bound, UB: upper bound) of a number of previously open instances. Here they are with the improved bounds in **bold**.

Instance	LB	UB	Proof	Instance	LB	UB	Proof
j60_9_3	**100**	100	*	j120_8_6	**85**	85	*
j60_9_8	**96**	96	*	j120_8_10	**92**	**92**	*
j60_9_9	**99**	99	*	j120_9_4	**86**	86	*
j60_9_10	**93**	93	*	j120_13_8	90	**93**	
j60_25_2	**98**	98	*	j120_14_5	93	**96**	
j60_25_4	**108**	108	*	j120_18_8	100	**105**	
j60_25_7	**90**	90	*	j120_19_9	**88**	88	*
j60_25_8	**99**	99	*	j120_26_10	**183**	183	*
j60_25_10	**108**	108	*	j120_28_7	**109**	109	*
j60_30_2	**70**	70	*	j120_29_3	**97**	**97**	*
j60_41_10	**111**	111	*	j120_29_6	**90**	**90**	*
j90_5_4	**102**	**102**	*	j120_33_2	103	**112**	
j90_5_5	**111**	111	*	j120_37_10	124	**131**	
j90_5_6	**86**	86	*	j120_38_5	108	**113**	
j90_5_7	**107**	107	*	j120_38_10	134	**139**	
j90_5_9	**115**	115	*	j120_42_1	**108**	108	*
j90_21_1	**110**	110	*	j120_47_5	**127**	127	*
j90_21_7	**109**	109	*	j120_48_2	110	**112**	
j90_21_8	**111**	111	*	j120_48_3	106	**111**	
j90_37_2	**115**	115	*	j120_48_7	**106**	**106**	*
j90_37_6	**131**	131	*	j120_49_2	**109**	109	*
j90_46_4	**93**	93	*	j120_53_7	116	**118**	
j120_6_4	**153**	**153**	*	j120_54_6	102	**108**	
j120_7_2	113	**113**	*	j120_54_10	**108**	108	*
j120_7_3	92	**99**		j120_55_1	**100**	100	*
j120_7_7	112	**117**		j120_59_9	115	**118**	
j120_8_2	**103**	103	*	j120_60_5	**104**	**104**	*
j120_8_5	98	**103**					

Finally, let's look at some results on the new benchmark with large instances. The figure below compares the time taken by CP Optimizer versions 12.8 and 12.9 for finding a first feasible solution to some instances of the large benchmark. Of course, we use the same 4 lines long CP Optimizer formulation as for the classical benchmarks. The x-axis represents the number of tasks of the instances.



We can see that it takes about 10mn for version 12.8 to find a first feasible solution to RCPSPs with 50,000 tasks (which is already remarkable for a generic exact solver), version 12.9 can find feasible solutions to problems that are 10 times larger (500,000 tasks) in less time (this is because of a new ingredient of the automatic search called *iterative diving* introduced in 12.9).

The Extensions

Many extensions of the basic RCPSP have been proposed like:

- Maximum delays
- Setup/transfer times
- Multi-mode
- Non-renewable resources
- Inventories
- Resource skills
- Preemptive tasks
- Hierarchical project decomposition (work-breakdown structure)
- Resource calendars

More realistic objective functions:

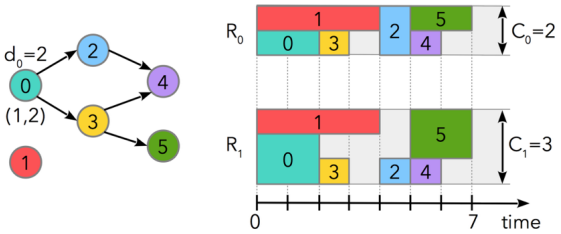
- Costs related with unperformed tasks
- Earliness / Tardiness costs
- Net Present Value

- Resource usage costs (Resource Availability Cost Problem)
- Cost related with task duration (Max-Quality RCPSP)

The CP Optimizer formulation I described above can easily be extended to handle these extensions. I will try to describe some of them in future posts.

Optimality Proof for the Example

Suppose we have a solution with a **makespan of 6**, we will show that this is impossible. It is enough to consider resource R0:



- Tasks 2 and 1 cannot be executed in parallel because of the capacity of resource R0. If task 2 is executed before task 1, the makespan would be at least duration of task 0 (d=2, which is a predecessor of task 2) plus duration of task 2 (d=1) plus duration of task 1 (d=4) which is 7. So for a makespan of 6, **task 2 is necessarily executed after task 1**.
- Now, because of the precedence chain 1->2->4 of duration 6, the tasks on this chain are necessarily fixed: **task 1 starts at t=0, task 2 starts at t=4, task 4 starts at t=5**.
- Because task 5 has a duration of 2 and cannot be executed together with task 2, **task 5 must be schedule before task 2**, otherwise given that task 2 ends at 5, the makespan would be at least 7.
- Because tasks 0 and 3 are constrained to be executed before task 5, it means they must also be executed before task 2, but this is impossible as there is not enough capacity to execute all these 3 tasks plus task 1 before the start time t=4 of task 2.

References

[AB-11] C. Ansótegui, M. Bofill, M. Palahí, J. Suy and M. Villaret: Satisfiability Modulo Theories: An Efficient Approach for the Resource-Constrained Project Scheduling Problem. SARA 2011

[AD-08] C. Artigues, S. Demassey and E. Neron. Resource-Constrained Project Scheduling. Wiley. 2008.

[AT-89] R. Alvarez-Valdes and J.M. Tamarit. Advances in Project Scheduling, chap. Heuristic algorithms for resource-constrained project scheduling: A review and an empirical analysis. pp. 113–134. Elsevier (1989)

[BL-00] P. Baptiste and C. Le Pape. Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. Constraints, vol. 5 (1-2), pp. 119-139, 2000.

[CN-01] J. Carlier and E. Néron. On linear lower bounds for resource constrained project scheduling problem. European Journal of Operational Research. vol. 149, pp. 314-324, 2001.

[DV-07] D. Debels and M. Vanhoucke. A decomposition-based genetic algorithm for the resource-constrained project scheduling problem. Operations Research, vol. 55, pp. 457-469. 2007.

[GJ-79] M.R. Garey and D.S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman and Company. 1979.

[KA-11] O. Kone, C. Artigues, P. Lopez and M. Mongeau. Event-based MILP models for resource-constrained project scheduling problems. Computers & Operations Research, vol. 38(1), pp. 3–13, 2011.

[KS-96] R. Kolisch and A. Sprecher. PSPLIB - A project scheduling problem library. European Journal of Operational Research, vol 96, pp. 205-216. 1996.

[LR-18] P. Laborie and J. Rogerie and P. Shaw and P. Vilim. **IBM ILOG CP Optimizer for Scheduling**. Constraints journal, Volume 23, Issue 2, pp 210–250. 2018.

[Pa-84] J. Patterson. A Comparison of Exact Approaches for Solving the Multiple Constrained Resource, Project Scheduling Problem. Management Science, vol. 30, pp. 854-867. 1984.

[SF-13] A. Schutt and T. Feydy and P.J. Stuckey. Explaining Time-Table-Edge-Finding Propagation for the Cumulative Resource Constraint. Proc. 10th International Conference CPAIOR, pp. 234-250. 2013.

[Te-15] A. Tesch. Compact MIP Models for the Resource-Constrained Project Scheduling Problem. Master's Thesis. Technische Universität Berlin. 2015.

[VC-08] M. Vanhoucke and J. Coelho and D. Debels and B. Maenhout and L. V.Tavaresc. An evaluation of the adequacy of project network generators with systematically sampled networks. European Journal of Operational Research. vol. 187(2), pp. 511-524. 2008.

[VL-15] P. Vilim and P. Laborie and P. Shaw . Failure-directed Search for Constraint-based Scheduling. Proc. 12th International Conference CPAIOR, pp 437-453. 2015.

Published by

Philippe Laborie
Software Developer: Operations Research, Artificial Intelligence, Optimization, Planning & Scheduling
Published • 2y

3 articles

#CPOptimizer and the Resource-Constrained Project Scheduling Problem ...
#scheduling #cplex #constraintprogramming #ORMS #operationsresearch #artificialintelligence #mip #optimization