Modelling and Solving Multi-mode Resource-Constrained Project Scheduling

Ria Szeredi[1] and Andreas Schutt[1,2(✉)]

[1] The University of Melbourne, Melbourne, Australia
ria.szeredi@student.unimelb.edu.au
[2] Decision Sciences, Data61, CSIRO, Canberra, Australia
andreas.schutt@data61.csiro.au

Abstract. The resource-constrained project scheduling problem is a fundamental scheduling problem which comprises activities, scarce resources required by activities for their execution, and precedence relations between activities. The goal is to find an optimal schedule satisfying the resource and precedence constraints. These scheduling problems have many applications, ranging from production planning to project management. One of them concerns multi-modes of activities, in which each mode represents a different time-resource or resource-resource trade-off option. In recent years, constraint programming technologies with nogood learning have pushed the boundaries for exact solution methods on various resource-constrained scheduling problems, but, surprisingly, have not been applied on multi-mode resource-constrained project scheduling. In this paper, we investigate different constraint programming models and searches and show the superiority of such technologies in comparison to the current state of the art. Our best approach solved all remaining open instances from a well-established benchmark library.

1 Introduction

The multi-mode resource-constrained project scheduling problem (MRCPSP) is an extension of the well-studied resource-constrained project scheduling problem (RCPSP), which comprises a set of non-preemptive activities, a set of resources with a constant capacity over time, and precedence relations between pairs of activities. For both problems, a start-time schedule for the activities is sought that respects the precedence relations, does not overload a resource at any point in time, while minimising the project duration (makespan). The differences between these problems are that activities can be executed in different modes and resources can be non-renewable in MRCPSP, while activities have a single mode and all resources are renewable in RCPSP. Different modes for an activity model time-resource and resource-resource trade-offs. These scheduling problems are NP-hard and have numerous applications, such as production planning, manufacturing, chemical processing, and project management [16].

An excellent overview of different and state of the art methods can be found in [5]. Most exact solution methods for solving MRCPSP are based on integer programming using branch-and-bound or branch-and-cut. The best methods were

M. Rueher (Ed.): CP 2016, LNCS 9892, pp. 483–492, 2016.
DOI: 10.1007/978-3-319-44953-1_31

published in [1,16]. Zhu et al. [16] present an exact branch-and-cut algorithm based on integer linear programming (ILP). They apply several pre-processing steps in order to reduce the number of variables in their models, pre-compute cuts from resource conflicts and precedence relations. In addition, a dedicated branching rule is developed for taking the different modes into account. To the best of our knowledge, it is the best exact method so far. Coelho et al. [1] propose a solution approach that decomposes the problem into two sub-problems. The first sub-problem consists of the assignment of the modes of execution solved by a Boolean Satisfiability (SAT) solver. The second sub-problem considers the fixed modes from the first sub-problem and solves the remaining problem using a local search method.

Closely related problems to MRCPSP are RCPSP and MRCPSP with generalised precedence relations (MRCPSP/max). In both cases, the best exact solution methods are based on constraint programming (CP) technologies incorporating nogood learning. For RCPSP, [12,13] present a branch-and-bound approach that is based on lazy clause generation (LCG) [2,7]. LCG is a CP solver that incorporates, amongst others, conflict analysis, conflict-driven search, and unit propagation on conjunction of clauses from SAT solvers. Exact solution approaches based on LCG are the best exact solution methods for various scheduling and packing problems [4,10,11,13–15].

For MRCPSP/max, [8] recently proposed a branch-and-bound approach formulated as a constraint integer program and implemented in the SCIP framework, which also has nogood learning facilities. In order to solve the problem, they implemented two new global constraints for generalised precedence relations and renewable resources, taking the multiple modes of an activity into account. The latter one is an extension of the global constraint `cumulative`. Their method is the best exact solution method for MRCPSP/max.

Surprisingly, no CP technology with nogood learning has been applied to MRCPSP. This paper addresses this gap and not only shows such a method outperforms the state of the art in [16], but also discusses different models and search strategies. In addition, we close all remaining open instances from the well-established benchmark library PSPLib.

2 MRCPSP Model

MRCPSP consists of a *set of non-preemptive activities* $V = \{1, 2, \ldots, n\}$, a *set of precedence relations* $E \subseteq V \times V$, and a *set of resources* $\mathcal{R}$. The set of resources is partitioned into the *set of renewable resources* $\mathcal{R}^R$ and the *set of non-renewable resources* $\mathcal{R}^N$. A resource $k \in \mathcal{R}$ has a discrete *resource capacity* R_k. An activity i has a fixed *set of modes* $\mathcal{M}_i$. For each mode $m \in \mathcal{M}_i$, the activity has a discrete non-negative *duration* (processing time) p_i^m and a discrete non-negative *resource requirement* r_{ik}^m for each resource $k \in \mathcal{R}$ over the planning horizon. The discrete non-negative *start time* S_i and the *mode of execution* M_i must be determined by the solution approach. The planning horizon starts at time period 0.

Definition 1 (Solution of MRCPSP). *A solution of MRCPSP is an assignment of start times S_i and modes of executions M_i for each activity i such that the following constraints hold*

$$\forall i \in V : 0 \leq S_i \quad \wedge \quad M_i \in \mathcal{M}_i$$

$$\forall (i,j) \in E : S_i + p_i^{M_i} \leq S_j \tag{1}$$

$$\forall k \in \mathcal{R}^R, \forall \tau \geq 0 : \sum\nolimits_{i \in V : S_i \leq \tau < S_i + p_i^{M_i}} r_{ik}^{M_i} \leq R_k \tag{2}$$

$$\forall k \in \mathcal{R}^N : \sum\nolimits_{i \in V} r_{ik}^{M_i} \leq R_k \tag{3}$$

where constraint (1) ensures the satisfaction of the precedence constraints and constraints (2–3) respectively guarantee a non-overload of renewable and non-renewable resources. An optimal solution additionally minimises the project duration (makespan), i.e., $\min \max_{i \in V}(S_i + p_i^{M_i})$.

2.1 Solver Independent Model

For the sake of readability, we present a "simplified" model and the "user-defined" searches using the solver-independent modelling language MiniZinc [6].

An MRCPSP instance is represented by the following parameters, whose meaning is given in the comment next to them where the arrays `mact` and `mode` respectively map a mode to its activity and an activity to its set of modes.

```
set of int: Res;     % Set of resources
set of int: Act;     % Set of activities
set of int: Mod;     % Set of modes
array[Res] of int: rcap;      % Resource capacity
array[Res] of int: rtype;     % Resource type (1: renewable; 2: non-renewable)
set of int: RRes = {k | k in Res where rtype[k] = 1};
set of int: NRes = {k | k in Res where rtype[k] = 2};
array[Mod] of Act: mact;          % Corresponding activity of a mode
array[Mod] of int: mdur;          % Duration of modes
array[Res,Mod] of int: mrreq;     % Resource requirements of modes
array[Act] of set of Mod: mode = [{m | m in Mod where mact[m] = i} | i in Act
    ]; % Set of modes for each activity
array[Act] of set of Act: succ; % Set of successors
```

Variables. Three variables are created for each activity `i` reflecting its start time `start[i]`, its duration `adur[i]`, and its resource requirements `arreq[k,i]` for each resource `k`. The duration and resource requirements are determined by the mode of execution. A Boolean variable `mrun[m]` models whether the mode `m` is executed in the final schedule. Lastly, the objective variable is defined as `makespan`.

```
array[Mod] of var bool: mrun;
array[Act] of var 0..UB: start;
array[Act] of var int: adur = [let {var {mdur[m] | m in mode[i]}: x} in x | i
     in Act];
array[Res,Act] of var int: arreq = array2d(Res,Act,[let {var {mrreq[k,m] | m
    in mode[i]}: x} in x | k in Res, i in Act]);
var 0..UB: makespan;
```

The variables in `start` and the variable `makespan` have an initial domain `0..UB` where `UB` is the initial upper bound on the objective. Unless otherwise stated, `UB` is initialised by `sum(i in Act)(max([mdur[m] | m in mode[i]]));`.

Activities and mode constraints. The duration and resource requirements of an activity are linked via a set of linear constraints, encapsulated in the first two constraints below. The last constraint ensures that exactly one mode is executed for each activity.

```
constraint forall(i in Act)(adur[i] = sum(m in mode[i])(mdur[m] * mrun[m]));
constraint forall(i in Act, k in Res)(arreq[k,i] = sum(m in mode[i])(mrreq[k,
    m] * mrun[m]));
constraint forall(i in Act)(sum(m in mode[i])(mrun[m]) = 1);
```

Alternatively, we can create an auxiliary variable `mi` and replace the first two constraints above by element constraints (elem) for modelling `adur` and `arreq`.

```
constraint forall(i in Act)( let {var mode[i]: mi} in (mrun[mi] = 1 /\ adur[i
    ] = mdur[mi] /\ forall(k in Res)(arreq[k,i] = mrreq[k,mi])) );
```

Precedence constraints. The precedence constraints are modelled as usual using the start time and duration variables.

```
constraint forall(i in Act, j in succ[i])(start[i] + adur[i] <= start[j]);
```

Renewable resource constraints. There are two options for modelling renewable resources using the global constraint cumulative. The first option (ract) creates one activity in the cumulative constraint for each activity, in which the durations and resource requirements are variables.

```
constraint forall(k in RRes)(cumulative(start, adur, [arreq[k,i] | i in Act],
     rcap[k]));
```

The second option (rmode) creates one activity for each mode, resulting in a greater number of activities generated in the cumulative constraint, but having only the resource requirements as variables. These variables can be created as a variable view on the Boolean variables in `mrun`.

```
constraint forall(k in RRes)(cumulative([start[mact[m]] | m in Mod], mdur, [
    mrreq[k,m] * mrun[m] | m in Mod], rcap[k]));
```

On the one hand, a cumulative propagator can exploit the knowledge of the duration-resource-requirement pairing in rmode. On the other hand, it loses the knowledge that exactly one mode has to be executed. Note that [8] extended the cumulative propagator for taking multi-modes for activities into account. However, the considered solvers in this paper do not provide this extension.

Non-renewable resource constraints. Non-renewable resources are simply modelled by linear constraints. As for the renewable resource constraints, there are again two options. The first option (nact) models it via the activities using the variables for the resource requirements,

```
constraint forall(k in NRes)(sum(i in Act)(arreq[k,i]) <= rcap[k]);
```

whereas the second option (nmode) models via the modes using the Boolean variables mrun.

```
constraint forall(k in NRes)(sum(m in Mod)(mrreq[k,m] * mrun[m]) <= rcap[k]);
```

Pairwise non-overlapping constraints. Pairwise non-overlapping constraints for activities might speed up the solution process and be advantageous for learning solvers. These are redundant constraints with respect to the renewable resource constraints. Two activities i and j cannot be run concurrently and are disjunct iff $\forall m_i \in \mathcal{M}_i, \forall m_j \in \mathcal{M}_j, \exists k \in \mathcal{R}^R : r_{ik}^{m_i} + r_{jk}^{m_j} > R_k$. The next constraint (disj) ensures that one of such activities is run before the other one.

```
predicate post_noc_disj(int: i, int: j) = (start[i] + adur[i] <= start[j] \/
    start[j] + adur[j] <= start[i]);
```

Other pairs of activities that might be a disjunct in some modes are modelled by following half-reified constraints.

```
predicate post_noc_mode(int: i, int: j) = forall(mi in mode[i], mj in mode[j]
     where exists(k in RRes)(mrreq[k,mi] + mrreq[k,mj] > rcap[k]))( (mrun[mi]
     /\ mrun[mj]) -> (start[i] + mdur[mi] <= start[j] \/ start[j] + mdur[mj]
    <= start[i]) );
predicate post_noc_rres(int: i, int: j) = forall(k in RRes)( (arreq[k,i] +
    arreq[k,j] > rcap[k]) -> (start[i] + adur[i] <= start[j] \/ start[j] +
    adur[j] <= start[i]) );
```

The predicate post_noc_mode (nocm) models non-overlapping constraints for each mode pair, while post_noc_rres (nocr) only for each renewable resource.

Objective constraints. The objective variable is constrained by the latest end time of an activity as follows.

```
constraint makespan = max(i in Act where succ[i]={})(start[i] + adur[i]);
constraint forall(i in Act where succ[i]={})(start[i] + adur[i] <= makespan);
```

Search strategies. We investigated different search strategies including the default ones of the considered solvers.

```
ann: mode_s  = bool_search(mrun, input_order, indomain_max, complete);
ann: start_s = int_search(start, smallest, indomain_min, complete);
ann: adur_s = int_search(adur, smallest, indomain_min, complete);
ann: arreq_s = int_search([arreq[k,i] | k in NRes, i in Act], smallest,
    indomain_min, complete);
ann: modeThenStart = seq_search([ mode_s, start_s ]);
ann: arreqThenMode = seq_search([ arreq_s, modes_s ]);
ann: arreqThenStart = seq_search([ arreq_s, start_s ]);
ann: arreqThenModeThenStart = seq_search([ arreq_s, mode_s, start_s ]);
ann: durThenStart = seq_search([ adur_s, modes_s ]);
ann: durThenModeThenStart = seq_search([ adur_s, mode_s, start_s ]);
```

The search modeThenStart splits the search into two stages. First, it assigns the mode to each activity and then solves the remaining RCPSP by searching on the start times. The next three searches arreqThenMode, arreqThenStart and arreqThenModeThenStart assign the smallest resource requirements of activities for non-renewable resources first, before continuing the search on the modes and/or the start times. The last two searches assign the shortest duration of each activity before assigning the mode and/or the start times. Note that if a search does not assign all variables, then the solver uses its default search to assign the remaining variables.

3 Experiments

We conducted experiments on the well-studied MRCPSP benchmark set from the PSPLib available at www.om-db.wi.tum.de/psplib/. The benchmark set contains different test sets, which differ in their characteristics. Except for the test set `j30`, all instances are closed. In the remainder of this paper, we concentrate on `j20` and `j30`. Instances from these test sets are composed of 20 and 30 activities having between one and three modes, two renewable resources, two non-renewable resources, and a number of precedence relations.

All experiments were run on machines operating CentOS 6.5 with AMD 6-Core Opteron 4334 clocking 3.1 GHz, and 64 GB memory. A runtime limit of 10 min was imposed unless otherwise stated. For compiling the MiniZinc model into the solver-specific FlatZinc format, we used MiniZinc 2.0.13 downloaded from www.minizinc.org. The following CP solvers were investigated Gecode 4.4.1 (`gecode`), Opturion CPX 1.0.2 (`ocpx`), G12/LazyFD (`lazyfd`), and Chuffed rev. 885 (`chuffed`) where the last three are LCG solvers.

3.1 Comparison of Models

Section 2.1 presents two ways of modelling renewable resource, non-renewable resource and pairwise non-overlapping constraints. Since the behaviour of non-learning and learning solvers can be significantly different, we present the results for `gecode` and `chuffed` as representatives for each kind.

Table 1 shows the results for both solvers using the search `modeThenStart` in two parts. Part I shows the different combinations for the renewable (rres) and non-renewable (nres) resource constraints. For both solvers, the best combination in terms of number of optimal solutions (#opt), mean runtime in seconds (m.rt.), and mean number of explored nodes (m.#nodes) is to use the activity representation (act) for both constraints. Interestingly, `chuffed` performance drastically decays when using the mode representation (nmode) for non-renewable resource constraints, while `gecode` only worsens slightly. This could indicate that `chuffed` misses some propagation. Note that we also run the experiments with both activity and mode representations, but the runtime increased substantially while the number of explored nodes did not change significantly.

Part II in Table 1 lists the results when the redundant non-overlapping constraints are used. In each setting, it is advantageous to use the redundant constraints. The best option is to use constraints for non-overlapping (disj, nocr) and activity representations (ract, nact), which gives the lowest mean runtimes for both solvers and the highest number of optimally solved instances in the case for `gecode`. Part III shows that using element constraints (elem) for the activity and mode constraints performs similar to the best option in Part II. For the remainder, we consider the model in Part III, which performs at best on the test set `j30`.

Table 1. Comparison of different models on 554 instances from test set `j20`.

part	rres	nres	disj	nocm	nocr	elem	chuffed			gecode		
							#opt	m.rt.	m.#nodes	#opt	m.rt.	m.#nodes
I	ract	nact					554	1.36s	21k	469	99s	6615k
	ract	nmode					525	56.7s	1544k	457	134s	9577k
	rmode	nact					554	2.41s	40k	464	104s	3509k
	rmode	nmode					516	61.2s	1421k	435	152s	5444k
II	ract	nact	✓				554	1.15s	19k	472	98s	6610k
	ract	nact	✓	✓			554	2.03s	18k	477	94s	4388k
	ract	nact	✓		✓		554	1.15s	13k	478	92s	5495k
III	ract	nact	✓		✓	✓	554	1.16s	13k	478	92s	5139k

Table 2. Comparison of different search strategies on test set `j30`.

search	chuffed					
	#opt	#feas	#unsat	#unkn	m.rt	m.#nodes
`modeThenStart`	495	57	88	0	60.3 s	615 k
`arreqThenMode`	488	59	88	5	64.0 s	549 k
`arreqThenStart`	491	61	88	0	61.7 s	580 k
`arreqThenModeThenStart`	490	62	88	0	62.4 s	614 k
`durThenStart`	506	46	88	0	56.3 s	628 k
`durThenModeThenStart`	506	46	88	0	58.3 s	682 k

3.2 Comparison of Search Strategies

Table 2 presents the outcome of the different searches when the best model (see previous sub-section) is used. Searches starting with the assignment of duration variables (`durThenStart` and `durThenModeThenStart`) are the quickest and optimally solve the greatest number of instances. The next best search is `modeThenStart`, while the remaining searches, all of which assign the resource requirement variables for non-renewable resources first, perform worst. Interestingly, searching over mode variables is slightly worse than leaving them out. For instance, the mean runtime of `durThenStart` is slightly less than that of `durThenModeThenStart`. Similar results for the search strategies were obtained on the models presented in Table 1 in preliminary experiments. For `chuffed`, we also ran each search in combination with `chuffed` activity based search, in which `chuffed` alternates between the two searches at each restart. The results are similar, but with the alternating searches more instances were solved to optimality and the mean runtime and number of nodes were lower.

3.3 Comparison of Solvers

Table 3 shows the results of the different solvers for the best model in combination with the solver's default search and the best search. Clearly, the default searches,

Table 3. Comparison of the different solvers on the test set `j30`.

solver	search	#opt	#feas	#unsat	#unkn	m.rt	m.#nodes
`gecode`	`durThenStart`	422	105	40	73	184 s	6056 k
`gecode`	`default`	385	87	0	168	246 s	6184 k
`ocpx`	`durThenStart`	468	84	44	44	151 s	>52 k
`ocpx`	`default`	492	58	88	2	113 s	>33 k
`lazyfd`	`durThenStart`	473	78	11	78	166 s	n/a
`lazyfd`	`default`	515	37	88	0	53.1 s	n/a
`chuffed`	`durThenStart`	506	46	88	0	56.3 s	628 k
`chuffed`	`default`	540	12	88	0	18.6 s	148 k

which are conflict driven, of the nogood learning solvers drastically outperform the user-defined searches. Note that `chuffed` default search alternates between a conflict driven and the user-defined search. As expected, nogood learning solvers outperform `gecode`, because their derived nogoods avoid the re-exploration of similar search sub-trees proven to be infeasible and information retrieved by the conflict analysis is used to guide the search. The clear winner is `chuffed`. The big difference in the performance of the LCG solvers may be surprising at first, but can be explained by the differences in the cumulative constraints. All three LCG solvers implement the cumulative constraint using the time-table propagation from [13], but only `lazyfd` and `chuffed` allow for variable durations and resource requirements as input as described in [9]. In addition, MiniZinc does not provide an interface for the cumulative constraint of `lazyfd`. Hence, the cumulative constraint is mapped into the time-indexed decomposition when compiling the model for `ocpx` and `lazyfd`.

3.4 Comparison to the State of the Art

Zhu et al. [16] present—to the best of our knowledge—the best exact solution method for MRCPSP. This method is based on Integer Linear Programming using a branch-and-cut for minimizing the makespan. It is implemented in the Mixed Integer Programming solver CPLEX 7.5. They run their method on a Linux machine with 1.8 GHz Xeon processor. Within one hour, it could optimally solve 506 instances out of 552 feasible instances with a mean total runtime of 393.1 s. The average runtime was 125.25 s for finding the best solution. By constrast, the best set up of `chuffed` could optimally solve 540 instances (34 more) within 10 min. In addition, the runtimes of `chuffed` are drastically lower. Thus, `chuffed` outperforms the state of the art.

Closed instances. With respect to [5,16], there are 46 open instances in the test set `j30`. Within the 10 min runtime limit, the presented solver was able to close 34 of them. In preliminary experiments, we ran `chuffed` without a runtime

limit and were able to close all instances. The last instance was closed after 18 hours.

4 Conclusion

We investigated different CP models for solving MRCPSP. To our best knowledge, it is the first published CP model, on which an exact solution method with nogood learning was applied. The best model uses the activity representation for modelling the resource constraints via the constraint `cumulative` and pairwise non-overlapping constraints for activities that might be in disjunction. All the considered user-defined searches were inferior to the default search of the CP solvers. The LCG solver `chuffed` was the best performing solver, which also outperformed the state of the art ILP solver [16]. Within 10 min, all open instances were closed except 12. Relaxing the time limit, all remaining open instances were closed within 18 h.

References

1. Coelho, J., Vanhoucke, M.: The Multi-mode resource-constrained project scheduling problem. In: Schwindt, C., Zimmermann, J. (eds.) Handbook on Project Management and Scheduling. International Handbooks on Information Systems, vol. 1, pp. 491–511. Springer, Heidelberg (2015)
2. Feydy, T., Stuckey, P.J.: Lazy clause generation reengineered. In: Gent [3], pp. 352–366
3. Gent, I.P. (ed.): Principles and Practice of Constraint Programming - CP 2009. LNCS, vol. 5732. Springer, Heidelberg (2009)
4. Kreter, S., Schutt, A., Stuckey, P.J.: Modeling and solving project scheduling with calendars. In: Pesant, G. (ed.) CP 2015. LNCS, vol. 9255, pp. 262–278. Springer, Heidelberg (2015)
5. Mika, M., Waligóra, G., Węglarz, J.: Overview and state of the art. In: Schwindt, C., Zimmermann, J. (eds.) Handbook on Project Management and Scheduling. International Handbooks on Information Systems, vol. 1, pp. 445–490. Springer, Heidelberg (2015)
6. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.R.: MiniZinc: towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007)
7. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation via lazy clause generation. Constraints **14**(3), 357–391 (2009)
8. Schnell, A., Hartl, R.F.: On the efficient modeling and solution of the multi-mode resource-constrained project scheduling problem with generalized precedence relations. OR Spectrum **38**(2), 283–303 (2015)
9. Schutt, A.: Improving scheduling by learning. Ph.D. thesis, The University of Melbourne (2011). http://hdl.handle.net/11343/36701
10. Schutt, A., Chu, G., Stuckey, P.J., Wallace, M.G.: Maximising the net present value for resource-constrained project scheduling. In: Beldiceanu, N., Jussien, N., Pinson, É. (eds.) CPAIOR 2012. LNCS, vol. 7298, pp. 362–378. Springer, Heidelberg (2012)

11. Schutt, A., Feydy, T., Stuckey, P.J.: Scheduling optional tasks with explanation. In: Schulte, C. (ed.) CP 2013. LNCS, vol. 8124, pp. 628–644. Springer, Heidelberg (2013)
12. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Why cumulative decomposition is not as bad as it sounds. In: Gent [3], pp. 746–761
13. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Explaining the cumulative propagator. Constraints **16**(3), 250–282 (2011)
14. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Solving RCPSP/max by lazy clause generation. J. Sched. **16**(3), 273–289 (2012)
15. Schutt, A., Stuckey, P.J., Verden, A.R.: Optimal carpet cutting. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 69–84. Springer, Heidelberg (2011)
16. Zhu, G., Bard, J.F., Yu, G.: A branch-and-cut procedure for the multimode resource-constrained project-scheduling problem. INFORMS J. Comput. **18**(3), 377–390 (2006)