Combining Monte-Carlo and hyper-heuristic methods for the multi-mode resource-constrained multi-project scheduling problem

Shahriar Asta [a], Daniel Karapetyan [a,b,*], Ahmed Kheiri [a,c], Ender Özcan [a], Andrew J. Parkes [a]

[a] University of Nottingham, School of Computer Science, Jubilee Campus, Wollaton Road, Nottingham, NG8 1BB, UK
[b] University of Essex, Institute for Analytics and Data Science, Wivenhoe Park, Colchester, CO4 3SQ, UK
[c] Cardiff University, School of Mathematics, Senghennydd Road, Cardiff, CF24 4AG, UK

ARTICLE INFO

ABSTRACT

Multi-mode resource and precedence-constrained project scheduling is a well-known challenging real-world optimisation problem. An important variant of the problem requires scheduling of activities for multiple projects considering availability of local and global resources while respecting a range of constraints. A critical aspect of the benchmarks addressed in this paper is that the primary objective is to minimise the sum of the project completion times, with the usual makespan minimisation as a secondary objective. We observe that this leads to an expected different overall structure of good solutions and discuss the effects this has on the algorithm design. This paper presents a carefully-designed hybrid of Monte-Carlo tree search, novel neighbourhood moves, memetic algorithms, and hyper-heuristic methods. The implementation is also engineered to increase the speed with which iterations are performed, and to exploit the computing power of multicore machines. Empirical evaluation shows that the resulting information-sharing multi-component algorithm significantly outperforms other solvers on a set of "hidden" instances, i.e. instances not available at the algorithm design phase.

© 2016 Elsevier Inc. All rights reserved.

1. Introduction

Project scheduling has been of long-standing interest to academics as well as practitioners. Solving such a problem requires scheduling of interrelated activities (jobs), potentially each using or sharing scarce resources, subject to a set of constraints, and with one or several of a variety of objective functions. There are various project scheduling problems and many relevant surveys in the literature, e.g. see [4,20–22,38,47,64]. The best-known problem class is the Resource Constrained Project Scheduling Problem (RCPSP) in which activities have fixed usages of the resources, there are fixed precedence constraints between them, and often the objective is simple minimisation of the makespan (completion time of last activity). These problems have been proven to be NP-hard [2], and a well-known benchmark suite, PSPLIB, is provided in [28].

---

* Corresponding author.
E-mail addresses: sba@cs.nott.ac.uk (S. Asta), daniel.karapetyan@gmail.com (D. Karapetyan), kheiria@cardiff.ac.uk (A. Kheiri), exo@cs.nott.ac.uk (E. Özcan), ajp@cs.nott.ac.uk (A.J. Parkes).

A generalisation of the RCPSP is to also consider 'Multi-mode RCPSP' (MRCPSP) in which activities can be undertaken in one of a set of modes, with each mode potentially using different sets of resources. Furthermore, there are many options besides makespan for the objective function(s); a typical one is that a weighted sum of completion times is minimised. As common in optimisation problems, exact methods perform best on smaller instances and on larger instances heuristics and metaheuristics become necessary. Recent works on the MRCPSP range from exact approaches, such as, MILP [32], and branch-and-bound [58], to metaheuristics, such as, differential evolution [11], estimation of distribution algorithms [60], evolutionary algorithms [14,17,54], swarm intelligence methods [30], and others [8,59].

This paper presents our winning approach submitted to MISTA 2013 challenge[1] on a further extension called 'multi-mode resource-constrained multi-project scheduling' (MRCMPSP) and the results on the associated benchmark/competition instances. The full description of this problem domain can be found on the competition website and in [61]; however, for completeness we also summarise it in Section 2. The broad aim is to schedule a set of different and partially interacting projects, with each project consisting of a set of activities. There are no precedence constraints between the activities of different projects however they can compete for resources. Also, the objective function is extended to be a mix of a kind of weighted completion time and makespan. The MRCMPSP is hence interesting in that it has a mix of structures and requirements that are a step towards modelling the complexity of real-world scheduling problems. The high real-world relevance of the multi-project version of scheduling is well-known, e.g. a survey [34] found that "84% of the companies which responded to the survey indicated that they worked with multiple projects". However, the majority of scheduling work is on the single project version, though there is some existing work on the multi-project case, e.g. see [18,31,34,35].

Our approach searches the space of sequences of activities, from which schedules are constructed and then the quality of each schedule is evaluated using the objective function. The search process on the set of sequences operates in two phases in a "construct and improve" fashion. In the first phase, a heuristic constructor creates initial sequences of activities. A novel proposal in this paper is to investigate the overall global structure of the solutions and use this to motivate constructing the initial sequences using a Monte-Carlo Tree Search (MCTS) method, e.g. see [3]. This construction phase is followed by an improvement phase which makes use of a large and diverse set of heuristic neighbourhood moves. The search process during the improvement phase is carefully controlled by a combination of methods arising from a standard metaheuristic, namely memetic algorithm, and also an extension of existing hyper-heuristic components [27,45].

There is an interesting potential for dual views of the overall problem. It is defined as a multi-project problem, but it can be also viewed as a single project (multi-mode) RCPSP, in which the precedence graph has a particular structure, consisting of disjoint clusters. There is a sense in which we work with both views together. Some neighbourhood moves treat the problem in a single-project fashion and work on the constituent activities; other neighbourhood operators explicitly consider the multi-project nature of the problem, and focus on moves of projects. Both views, and kinds of operators, are used and work together to improve the overall project-level structure as well as the detailed activity level structure. A discussion and a computational study on both approaches can be found in [34].

The primary contributions of this paper are:

- Observation and investigation of how the primary objective function being essentially a "sum of project completion times" leads to good solutions having inherently different structure to those with makespan as the primary objective. In particular, minimisation of project completion times subject to limited global resource results in partial ordering of projects; this does somewhat reduce the effective size of the search space, but also may lead to good solutions being more widely separated. Understanding of this significantly affected our algorithm design, including an MCTS construction method aiming to create solutions having such structure.
- Novel neighbourhood moves, including those that are designed specifically for smoother navigation through the search space of the multi-project extension of MRCPSP – reflecting our observation of the effect that the main objective function has on the solution structures.
- An adaptive hybrid hyper-heuristic system to effectively control the usage of the rich set of neighbourhood moves.
- Evidence of the effectiveness based on successful results on a range of benchmark problems. This includes winning a competition, in which some problems were hidden at the algorithm design/tuning phase. We also tested our algorithm on single-project instances from PSPLIB. Although our algorithm was not designed to work on single-project instances, it demonstrated good performance in these tests, and was competitive with the state-of-the-art methods tailored to the single-project case. Furthermore, it improved 3 best solutions on these PSPLIB instances during these experiments.

These contributions are directed towards a system that is both robust and flexible; with the potential to be effective at handling a wide variety of problem requirements and instances. Arguably, one of the lessons of this paper is that greater complexity and richness of such scheduling problems needs to be matched with a greater complexity and richness of the associated algorithms; especially when not all instances are known in advance, and so algorithms should not over-specialise to a particular data set.

Regarding the structure of the paper; in Section 2 we describe the problem to be solved. In Section 3 we discuss how we have carefully chosen the appropriate data structures and implemented algorithms operating with those data structures efficiently in order to construct the schedule from a given sequence as fast as possible. (To build an effective system, one has

---

to pay attention to all of its components.) However, most of the contribution of this paper arises from choosing and combining the effective algorithmic components including the search control algorithm and then (partially) tuning the relevant parameters within the overall approach. These consist of the MCTS-based constructor given in Section 4, the neighbourhoods given in Section 5, and the improvement phase given in Section 6. The computational experiments and competition results are presented and analysed in Section 7; including some reports of performance on a multi-mode, though single project, benchmark set from PSPLIB. Section 8 concludes the paper.

2. Problem description

The problem consists of a set P of *projects*, where each project $p \in P$ is composed of a set of *activities*, denoted as A_p, a partition from all activities A. Each project $p \in P$ has a *release time* e_p, which is the earliest start time for the activities A_p.

The activities are interrelated by two different types of constraints: the *precedence constraints*, which force each activity $j \in A$ to be scheduled to start no earlier than all the immediate predecessor activities in set *Pred*(j) are completed; and the *resource constraints*, in which the processing of the activities is subject to the availability of resources with limited capacities. There are three different types of the resources: local renewable, local non-renewable and global renewable. Renewable resources (denoted using the superscript ρ) have a fixed capacity per time unit. Non-renewable resources (denoted using the superscript ν) have a fixed capacity for the whole project duration. Global renewable resources are shared between all the projects while local resources are specified independently for each project.

Renewable and non-renewable resources are denoted using the superscript ρ and ν, respectively. $\mathscr{R}_p^\rho$ is the set of local renewable resources associated with a project $p \in P$, and R_{pk}^ρ is the *capacity* of $k \in \mathscr{R}_p^\rho$, i.e. the amount of the resource k available at each time unit. $\mathscr{R}_p^\nu$ is the set of local non-renewable resources associated with a project $p \in P$, and R_{pk}^ν is the capacity of $k \in \mathscr{R}_p^\nu$, i.e. the amount of the resource k available for the whole duration of the project. $\mathscr{G}^\rho$ is the set of the global renewable resources, and G_k^ρ is the capacity of the resource $k \in \mathscr{G}^\rho$.

Each activity $j \in A_p$, $p \in P$, has a set of execution modes $\mathscr{M}_j$. Each mode $m \in \mathscr{M}_j$ determines the duration of the activity d_{jm} and the activity resource consumptions. For a local renewable resource $k \in \mathscr{R}_p^\rho$, the resource consumption is r_{jkm}^ρ; for a local non-renewable resource $k \in \mathscr{R}_p^\nu$, the resource consumption is r_{jkm}^ν; for a global renewable resources $k \in \mathscr{G}^\rho$, the resource consumption is g_{jkm}^ρ.

Schedule $D = (T, M)$ is a pair of time and mode vectors, each of size n. For an activity $j \in A$, values T_j and M_j indicate the start time and the execution mode of j, respectively. Schedule $D = (T, M)$ is feasible if:

- For each $p \in P$ and each $j \in A_p$, the project release time is respected: $T_j \geq e_p$;
- For each project $p \in P$ and each local non-renewable resource $k \in \mathscr{R}_p^\nu$, the total resource consumption does not exceed its capacity R_{pk}^ν.
- For each project $p \in P$, each time unit t and each local renewable resource $k \in \mathscr{R}_p^\rho$, the total resource consumption at t does not exceed the resource capacity R_{pk}^ρ.
- For each time unit t and each global renewable resource $k \in \mathscr{G}_p^\rho$, the total resource consumption at t does not exceed the resource capacity G_k^ρ.
- For each $j \in A$, the precedence constraints hold: $T_j \geq \max_{j' \in Prec(j)} T_{j'} + d_{j'M_{j'}}$.

The objective of the problem is to find a feasible schedule $D = (T, M)$ such that it minimises the so-called *total project delay* (TPD), defined by using the time for *total project completion* (TPC)

$$\text{TPC} = \sum_{p \in P} C_p \tag{1}$$

and

$$\text{TPD} \equiv f_\text{d}(D) = \text{TPC} - L = \left(\sum_{p \in P} C_p \right) - L, \tag{2}$$

where C_p is the completion time of project p

$$C_p = \max_{j \in A_p} \left(T_j + d_{jM_j} \right). \tag{3}$$

The constant L is a lower bound calculated as

$$L = \sum_{p \in P} (\text{CPD}_p + e_p), \tag{4}$$

with CPD_p being a given pre-calculated value. Since L is a constant, then it does not affect the optimisation (it was presumably introduced in the competition just to make the output numbers smaller and easier to interpret). Specifically, since L is the lower bound (though not necessarily a tight bound), $f_\text{d}(D) \geq 0$ for any feasible solution D.

Note that this primary objective is an instance of the standard "weighted completion time", usually denoted by "$\sum_j w_j C_j$", but specialised to the case, "$\sum_p w_p C_p$", in which only the completion times of the projects are used[2] (in the case of TPD all the weights are assigned to be one).

The tie-breaking secondary objective is to minimise the *total makespan*, (TMS), which is the finishing time of the last activity (or equivalently of the last project):

$$\text{TMS} \equiv f_\text{m}(D) = \max_{j \in A} \left(T_j + d_{jM_j} \right) = \max_{p \in P} C_p . \tag{5}$$

In our implementation, we combine the objective functions $f_\text{d}(D)$ and $f_\text{m}(D)$ into one function $f(D)$ that gives the necessary ranking to the solutions:

$$f(D) = f_\text{d}(D) + \gamma f_\text{m}(D) , \tag{6}$$

where $0 < \gamma \ll 1$ is a constant selected so that $\gamma f_\text{m}(D) < 1$ for any solution D produced by the algorithm. In fact, we sometimes use $\gamma = 0$ to disable the second objective. For details, see Section 6.1.

Under the conventional "$\alpha|\beta|\gamma$" labelling, it could perhaps be described as "MP$^+$S | *prec* | ($\sum_p w_p C_p$, C_max)" using 'MP$^+$S' to denote 'Multi-mode Multi-Project Scheduling'.

3. Schedule generator

Designing an algorithm for solving the multi-mode resource-constrained multi-project scheduling problem requires an appropriate solution representation. There are two 'natural' solution representations in the scientific literature:

Schedule-based: A direct representation using the assignment times, and also modes, of activities, i.e. vectors T and M.
Sequence-based: This is based on selecting a total order on all the activities. Given such a sequence, a time schedule is constructed by taking the activities one at a time in the order of the sequence and placing each one at the earliest time slot such that feasibility of the solution would be preserved. This approach is called *serial schedule generation scheme* (e.g., see [4]).

The schedule-based representation is perhaps the most natural one for a mathematical programming approach, but we believe that it could make the search process difficult for a metaheuristic method, in particular, generating a feasible solution at each step could become more challenging. As is common for heuristic approaches [61], we preferred the sequence-based representation, since it provides the ease of producing schedules that are both feasible and for which no activity can be moved to an earlier time without moving some other activities (the schedule is then said to be 'active').

The sequence-based representation is a pair $S = (\pi, M)$, where π is a permutation of all the activities A, and M is a modes vector, same as in the direct representation. The permutation π has to obey all the precedence relations, i.e., $\pi(j) > \pi(j')$ for each $j \in A$ and $j' \in Pred(j)$. The modes vector is feasible if $M_j \in \mathscr{M}_j$ for each $j \in A$ and the local non-renewable resource constraints are satisfied for each project $p \in P$.

In order to evaluate a solution S, it has to be converted into the direct representation D. By definition, the sequence-based representation $S = (\pi, M)$ corresponds to a schedule produced by consecutive allocation of activities $\pi(1), \pi(2),..., \pi(n)$ to the earliest available slot. The corresponding procedure, which we call the *schedule generator*, is formalised in Algorithms 1–3. Note that the procedure guarantees feasibility of the resulting schedule as it schedules every activity in such a way that feasibility of the whole schedule is preserved.

Algorithm 1: Serial schedule generation scheme.

1 Let $S = (\pi, M)$ be the sequence-based solution;
2 **for** $i \leftarrow 1, 2, \ldots, n$ **do**
3 $\quad$ Let $j \leftarrow \pi(i)$;
4 $\quad$ Schedule j in mode M_j to the earliest available slot such that feasibility of the schedule is preserved;
5 **end**

The worst case time complexity of this implementation is $O(n(\zeta + T\rho d))$, where $\zeta = \max_{j \in A} |Prec(j)|$ is the maximum length of the precedence relation list, T is the makespan, $\rho = |\mathscr{G}^\rho| + \max_{p \in P} |\mathscr{R}_p^\rho|$ is the maximum number of local and global renewable resources and $d = \max_{j \in A} d_{jM_j}$ is the maximum activity duration. The first term of the sum corresponds to handling precedence relations, and the second term corresponds to scanning slots and testing resource availability. Note that $\zeta < n$, and the maximum number ρ of resources is a constant in our benchmark instances. Also, T is typically linear in n, and, hence, the time complexity is quadratic. The schedule generator is the performance bottleneck of our solver (note that most of the local search moves described in this paper are no worse than linear time complexity). In our experiments,

---

[2] If there is no unique activity marking the end of a project, then a dummy empty end activity can always be added, without changing the problem.

Algorithm 2: Scheduling an activity to the earliest available slot.

---
1 Let j be the activity to be scheduled;
2 Let m be the mode associated with j;
3 Let p be an index such that $j \in A_p$;
4 Calculate the earliest start time of j as $t_0 \leftarrow \max\left\{e_p, \max\limits_{j' \in Prec(j)} \left(T_{j'} + d_{j'm}\right)\right\}$;
5 $t \leftarrow TestSlot(j, t_0)$;
6 Allocate activity j at t in mode m and update the remaining capacities;

Algorithm 3: A naive implementation of the *TestSlot*(j, t) function. The function returns the earliest time slot at or after t feasible for scheduling activity j.

---
1 Let m be the mode associated with j;
2 Let p be an index such that $j \in A_p$;
3 **for** $t' \leftarrow t,\ t+1,\ \ldots,\ t + d_{jm} - 1$ **do**
4   **for** $k \in \mathscr{R}_p^\rho$ **do**
5   Let a be the remaining capacity of k at t';
6   **if** $r_{jkm}^\rho > a$ **then return** $TestSlot(j, t+1)$;
7   **end**
8   **for** $k \in \mathscr{G}^\rho$ **do**
9   Let a be the remaining capacity of k at t';
10   **if** $g_{jkm}^\rho > a$ **then return** $TestSlot(j, t+1)$;
11   **end**
12 **end**
13 **return** t;

---

schedule generation was usually taking over 98% of the CPU time. By introducing several improvements (based on information sharing and caching of partial solutions) described below we reduced the running times of the schedule generator by a factor of around ten compared to our initial routine implementation. That significantly increased the number of iterations the higher level algorithm was able to run within a given time.

3.1. Issues in efficient implementation

In this section, we briefly discuss algorithmic and implementational issues that do not directly affect the number of sequence evaluations, but that are designed to increase the rate that evaluations are performed. This is of importance for application of the methods to real-world problems. However, it can also be of potential importance to choices between different heuristics or other algorithmic components.

The methods for evaluation and comparison of algorithms are not necessarily clear. In particular, two aspects that arise with respect to this work, and scheduling in general, are "hidden instances", and "termination criteria". Hidden instances (meaning ones that are not available until after the implementation is finished) act against the danger that occurs with open instances of the techniques becoming tailored to the specific instances. For reliability and verification of results this generally means the implementations must be finalised before the release of the instances. In practice, this seems to be rarely applied outside of the context of a competition. In such cases, one might regard competitions are a way to enforce the scientific good practice of fully finalising the algorithm and implementation before the testing.

The other aspect is evaluating algorithms' performance purely in terms of their (wall-clock) runtimes, or instead using some attempt at implementing independent "counting" measure of steps taken. The advantages of the former "runtime" method is that it relates to what real world users would usually care about, and also might be the only real option when no sensible pure counting methods are available. The advantage of the counting is that it hides the implementation efficiency and hence allows to compare "pure algorithm designs". In some research situations the classes of algorithms are sufficiently similar for a counting based comparison being viable, and then may well be standard e.g. in genetic algorithms the number of fitness evaluations is commonly used.

Hence, the counting-based approach encourages/supports rapid explorations of ideas for new algorithms, however exclusive usage would discourage developing new practical methods of improving algorithm performance. For example, counting fitness evaluations can miss the advantages of the incremental evaluation techniques routinely used in metaheuristics, and that are vital for their effectiveness. Runtime-based approach on contrary encourages the researchers to exploit methods that are practical in real circumstances, taking into account incremental evaluation, parallelism and other considerations crucial for real-world systems. We note here that the associated added complexity of algorithm engineering could potentially be

partly addressed by hyper-heuristics, as they could provide feedback to the programmer regarding the extra value of low-level heuristics if their implementation were improved. This is particularly relevant in the context of a solver employing multiple neighbourhoods, such as the one presented in this paper.

We believe that there is no simple answer to which of these two algorithm evaluation approaches is best in general and so both of them have their place. However in the context of the very well studied project scheduling we do believe that engineering questions need to be accounted for. As an example of the importance of such 'engineering issues' we refer to another well-studied area of solving propositional satisfiability (SAT) problems and that has been active for many decades.[3] An important part of development of SAT solvers [41] was the development of 'watched literals'. The technique only directly affected the standard and routine 'unit propagation' procedure in SAT solvers (which is the CPU-intensive portion, analogous to the schedule generator), but it did so in a fashion that meant new heuristics were then practical, leading to new algorithm designs.

In real-world usages of scheduling, an important aspect is the software engineering aspect of the time and cost of implementing and maintaining the software. An initial implementation of a neighbourhood is often relatively easy; however, the practical problems can arise when effective use requires that it is implemented in a fashion that uses incremental or delta evaluation (so that the objective function does not require a full re-evaluation). To support the incremental evaluation it is often necessary to implement appropriate data structures that are more sophisticated, and so harder to implement and maintain. With serial generation the majority of the CPU time is spent in the generation of the schedule from the sequence. Hence, we naturally found that significant improvements were achieved by modifying the generator algorithm. Good 'engineering' of the serial generation also has the important advantage that it helps all of the neighbourhoods. (If all the neighbourhoods were to rely on entirely separate implementations, then there would be much more pressure, for practical software engineering issues, to reduce to a smaller set.)

Observe (see Algorithm 2) that the schedule generation algorithm spends most of the time finding the first available slot for an activity. To speed up this phase, we use a modification of the Knuth-Morris-Pratt substring search algorithm. By testing resource availability in the reversed order, we can use early exploration of insufficient resources to skip several values of t, see Algorithm 4.

Algorithm 4: An improved implementation of the *TestSlot(j, t)* function. The function returns the earliest time slot at or after t feasible for scheduling activity j.

---

1 **for** $t' \leftarrow t + d_{jm} - 1, \ t + d_{jm} - 2, \ \dots, \ t$ **do**
2 $\quad$ **for** $k \in \mathcal{R}_p^{\rho}$ **do**
3 $\quad\quad$ Let a be the remaining capacity of k at t';
4 $\quad\quad$ **if** $r_{jkm}^{\rho} > a$ **then return** $TestSlot(j, t' + 1)$;
5 $\quad$ **end**
6 $\quad$ **for** $k \in \mathcal{G}^{\rho}$ **do**
7 $\quad\quad$ Let a be the remaining capacity of k at t';
8 $\quad\quad$ **if** $g_{jkm}^{\rho} > a$ **then return** $TestSlot(j, t' + 1)$;
9 $\quad$ **end**
10 **end**
11 **return** t

Another speed-up heuristic is exploiting the nature of the neighbourhood moves; we noted that any two solutions tested consequently are likely to share a prefix. Let $S^1 = (\pi^1, M^1)$ be some solution, $S^2 = (\pi^2, M^2)$ be its neighbour, and $D^1 = (T^1, M^1)$ and $D^2 = (T^2, M^2)$ be their direct representations. According to our assumption, $\pi^1(i) = \pi^2(i) = j$ and $M_j^1 = M_j^2$ for $i = 1, 2, \dots, x$, where x is the prefix length (which is likely to be significant). Then, by construction, $T_j^1 = T_j^2$ for each $j = \pi^2(1), \pi^2(2), \dots, \pi^2(x)$. Hence, knowing D^1, we do not need to calculate the values T_j^2 for $j = \pi^2(1), \pi^2(2), \dots, \pi^2(x)$. For details, see Algorithm 5.

This gives a form of incremental evaluation that has the advantage that it applies to all the neighbourhoods used. Potentially, this engineering optimisation could have impact on the design of the improvement algorithm in that the selection of neighbourhood moves could benefit from exploiting changes at the end of the schedule being faster to evaluate than those at the beginning. This is something that can potentially be captured by a hyper-heuristic, as one of the long-term intentions of hyper-heuristics is that they should monitor the CPU times taken by different moves, and combine this with monitoring of their effects, in order to give better adaptive control of the improvement phase.

In addition to the algorithmic improvements, we used several standard programming techniques to optimise the implementation performance, with the primary ones being:

---

[3] E.g. competitions have been held for many years, see http://www.satcompetition.org/.

Algorithm 5: Serial schedule generation scheme with prefix detection.

---

1 Let $S^2 = (\pi^2, M^2)$ be the new solution;
2 Let $S^1 = (\pi^1, M^1)$ be the previous solution and $D^1 = (T^1, M^1)$ be the corresponding direct representation;
3 Let $prefix \leftarrow true$;
4 for $i \leftarrow 1, 2, \ldots, n$ **do**
5   Let $j \leftarrow \pi^2(i)$;
6   **if** $prefix = true$ **and** $\pi^1(i) = j$ **and** $M_j^1 = M_j^2$ **then**
7   Allocate activity j to T_j^1 in mode M_j^2 and update the remaining capacities;
8   **else**
9   Schedule j in mode M_j^2 to the earliest available slot;
10 $prefix \leftarrow false$;
11   **end**
12 end

- **Incremental maintenance of auxiliary data:** Note that the schedule generator has to maintain the amount of remaining resource for each renewable resource and each time unit, i.e. $T \cdot \left(|\mathscr{G}^\rho| + \sum_{p \in P} |\mathscr{R}_p^\rho|\right)$ values. Since T is not known in advance, the corresponding auxiliary data structure has to be large enough to fit a schedule of size $T_{\max}$, where $T_{\max}$ is the upper bound of T. Considering the initialisation of this auxiliary data structure, the real time complexity of the algorithm is $O(n(\zeta + T\rho d) + T_{\max}\rho')$, where $\rho' = |\mathscr{G}^\rho| + \sum_{p \in P} |\mathscr{R}_p^\rho| = O(\rho|P|)$. Usually $T_{\max} \gg T$ and $\rho' \gg 1$ and, hence, the last term of the generator complexity has a major impact on the real performance of the procedure. By reusing the same auxiliary data structures and only re-initialising the portion that was altered in the previous run of the generator, we speed up the initialisation phase by a factor of about $T_{\max}/T$.
- **Local memory access:** Most of the solution evaluations happen in our metaheuristic within local search procedures. Each instance of the local search algorithm is assigned a CPU core, and a dedicated copy of the schedule generator is maintained for it. This is particularly important for the prefix reuse heuristic and CPU cache efficiency.

4. Solution structures

This section primarily explains why having "Total Project Delay" (TPD) as the main objective can be expected to lead to the general structure of solutions being very different to the structure obtained with the more standard makespan (TMS) objective. Our observations of the TPD-driven structure had an important influence on our design of the overall algorithm. The structure motivates many of the "project level" moves that are considered in Section 5. In this section, we also propose a heuristic method to construct the initial solutions which feed into the improvement phase later. As standard in such "construct and improve" optimisation, the concern was that the desirable structures might be difficult to achieve by the improvement unless the initial constructor heuristically tried to get close, by doing an appropriate "heuristically-guided global sampling" of the space of approximate structures. Accordingly, with the intent to increase the robustness of the solver, to handle such structures in unseen instances, a specialised MCTS-based constructor is proposed and developed.

4.1. TPD-driven solution structures

In this subsection, we report on the investigation of the natural question of what constitutes a 'good' approximate structure. We firstly look at the structure of a high quality schedule for a benchmark instance, and then we elucidate the observed structures using some small examples. The critical message is that optimising TPD leads to different structures of the solutions than when optimising TMS; we believe that algorithm design needs to take account of this difference.[4]

In particular, we have observed that in many cases, dominance by the TPD objective frequently leads to approximate ordering of the projects. A typical example of this is given in Fig. 1 using a good solution to instance B-1. Fig. 1a shows the structure obtained when only the standard makespan is minimised, but the structure is very different in Fig. 1b with the required TPD-dominated objective. That is, the pattern of completion times of each project, the C_p of (3), depends on whether they are driven by TPD, effectively minimising the average of the C_p, or instead driven by the makespan, reducing the maximum of the C_p. The example shows how, with TPD dominating, there are time periods in the schedule when the general focus is on relatively few projects and during the schedule this focus changes between projects.

Hence, the evidence from Fig. 1, and other multi-project cases we have looked at, suggests that the approximate ordering is common. Overall, these structures naturally arises from the combination of the TPD (2) objective with the limited global resources. Suppose that some P is the final finishing project, and so its last activity determines the TMS and P' is an earlier finishing project. It may well be that P' can move some of its activities earlier by delaying activities of P, though without changing the last activity of P. Such moves will improve the TPD without worsening the TMS.

---

[4] Similar effects are also observed in [62].
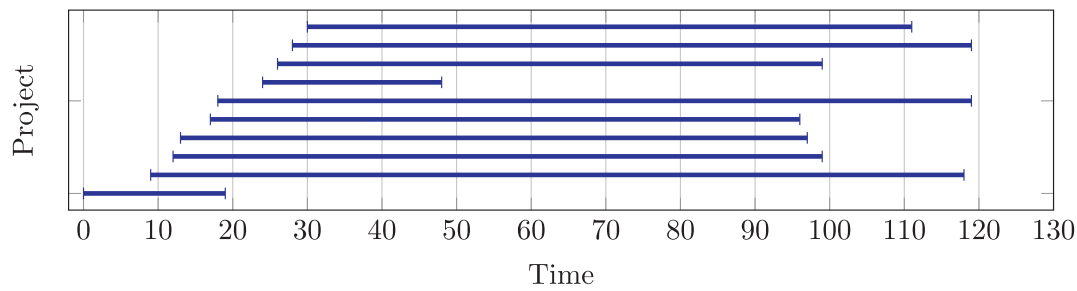
(a) Only makespan is minimised (the TPD objective is disabled), that is minimising the latest completion times of projects. TPC = 925, TMS = 119.
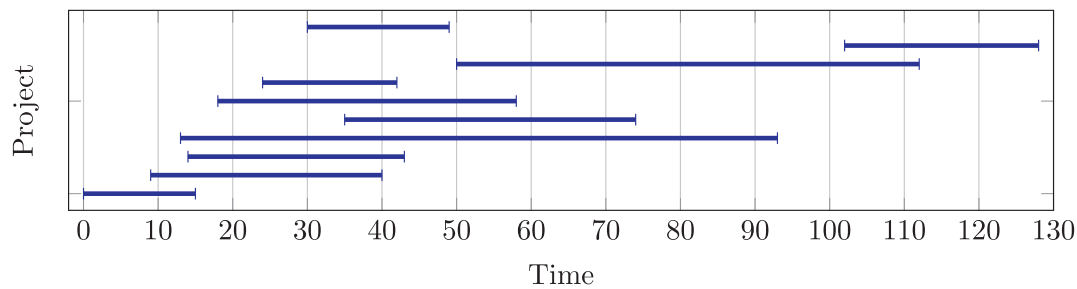
(b) Both objectives are enabled, that is, the primary objective is the TPD – minimising the sum of completion times of projects. TPC = 654, TMS = 128.

Fig. 1. An example of overall project-level structure of good solutions (using instance B-1). Each horizontal line shows the duration allocated to each project in the schedule.

That is, in contrast to the makespan objective, the TPD objective, together with limited shared resources, has a natural side-effect of encouraging unfairness between the finish times of projects, and will drive some projects to finish as early as possible.

To illustrate the way in which an approximate ordering might arise, and TPD-driven and TMS-driven solutions can be quite different, we give a set of small example instances. We denote these example instances by "2[x–y–x]", consisting of two projects. Each project consists of a precedence-constrained chain of just 3 activities of durations x, y and x, respectively. There are two (renewable) shared resources with activities 1 and 3 using the resource 2, and activity 2 using resource 1. We focus on values of x and y that lead to resource 2 being the bottleneck, and so the main driving force of the makespan – corresponding to the shared global resource in the MISTA benchmarks. Fig. 2a shows four schedules of this parameterised instance, along with their corresponding values of C_1 and C_2. Solutions S_4 and S_3 arise from simply swapping the order of the projects in solutions S_1 and S_2 respectively.

The solutions S_1 and S_4 do not interleave the two projects, and so leave gaps in the bottleneck resource 2. In contrast, solutions S_2 and S_3 interleave the projects and so lead to full usage of the bottleneck resource, and are hence automatically the best solutions for the TMS objective. However, S_2 and S_3 are not always the best solutions for the TPD objective. Changing the solution leads to a tradeoff between C_1 and C_2, and this is illustrated, in Fig. 2b for various choices of x and y. In this figure, the values of x and y are selected so that we always have $2x + y = 100$; this makes the effect of changing the relative balance of x and y clearer, and also means that the values can be interpreted as percentages of the project time.

Before proceeding further, we observe that we can also interpret the project completion times, C_p, from the point of view of 'multi-objective optimisation', by regarding each of them as separate objectives. In particular, in standard fashion, one can say that a set of values for C_p dominates another set provided that none of the C_p values are worse (larger) and at least one is better (smaller). One can hence discuss the cases in Fig. 2b as being the C_p-Pareto Front, of non-dominated sets of C_p. We emphasise this view is for obtaining insight into the space of solutions, and is not used directly within our algorithm; we did not perform multi-objective optimisation over the C_p, though believe it would be worthy of future investigation. Also, note this view is different from taking the pair (TPD, TMS) as a bi-objective problem, and having a (TPD, TMS)-Pareto Front (this would correspond again to a different class of algorithms than we consider here, but is again worthy of future investigation). Incidentally, there are two distinct 'multi-objective views' on the space of solutions:

- C_p-Pareto Front, of dimension p, formed from the p completion times
- (TPD, TMS)-Pareto Front, of dimension 2, and formed from the aggregates (sum and max) of the C_p

However, it is important to realise that although the C_p and (TPD, TMS) views are different, they are tightly linked. Both the TPD, arising from $\Sigma_p C_p$, and the TMS, from $\max_p C_p$, are monotone (non-decreasing) with respect to the C_p and

(a) Case "2[x–y–x]". Showing four possible solutions, $S_1, \ldots, S_4$, and associated project completion times C_1 and C_2.

(b) Tradeoff between completion times of projects. We select a range of different values for x and y, but with the invariant that $2x + y = 100$; so that the effect of varying the balance between x and y is made clear.
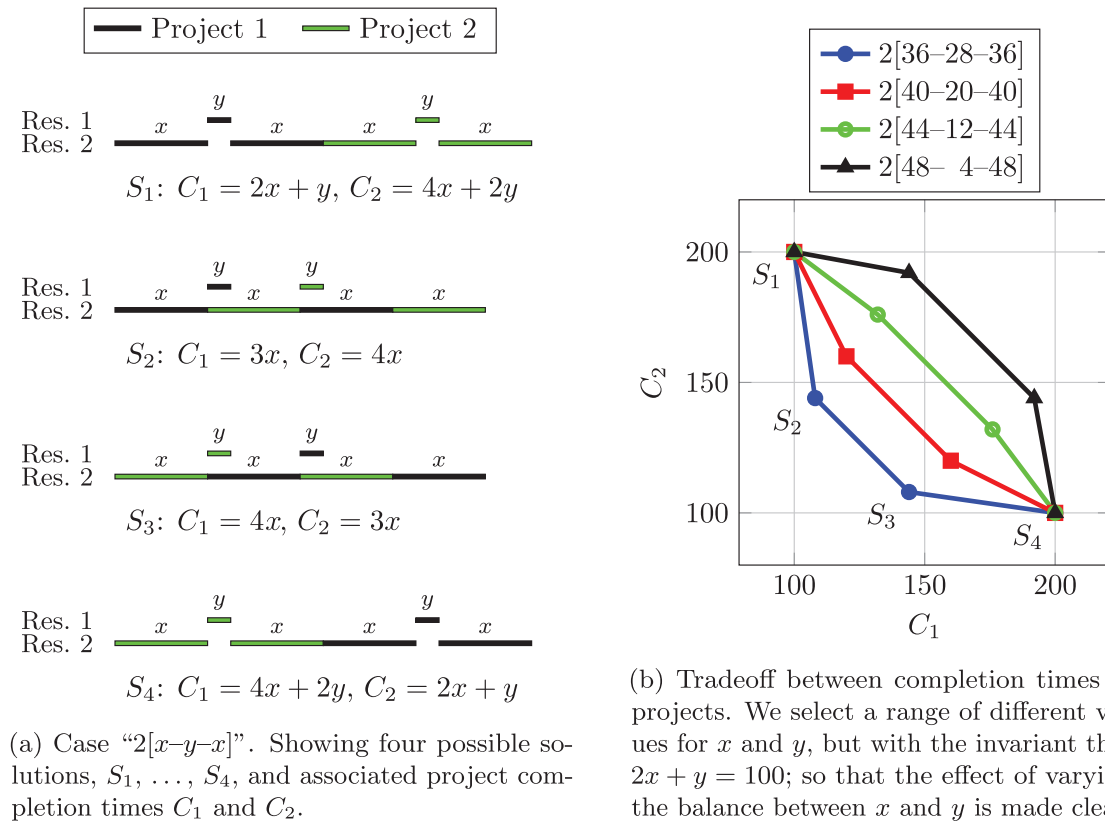
Fig. 2. Simple 2-project examples to illustrate concave and convex non-dominated sets. The notation "2[x–y–x]" means two projects each consisting of a chain of 3 tasks of durations [x, y, x] with shared resources, and the tasks 1 and 3 in each chain using resource 2, and task 2 using resource 1.

so preserve Pareto dominance. That is, if solution C_p dominates solution C'_p, then the resulting pair (TPD, TMS) dominates (TPD', TMS') – though note that the converse does not apply. Hence, the (TPD, TMS)-Front can be extracted from the C_p-Front. However, this 'projection' loses information; the C_p-Front gives more insight into the space of solutions.

Consequently, the "C_p-Pareto Front", or set of non-dominated C_p values, can give some insight into the effect of TPD compared to TMS on the structure of the solutions. Hence, we can regard Fig. 2b as showing the effect of the x–y balance on the C_p-Pareto Front. In particular, we see that for two of the cases, 2[44–12–44] and 2[48–4–48] the C_p-Pareto Front is concave in terms of the set of feasible C_p values (on the top right). When the Front is concave then the two solutions minimising the TPD are S_1 and S_4 and are at opposite ends of the C_p-Front. In contrast, the solutions S_2 and S_3 minimising the TMS are in the middle.

Generally, with a concave C_p-Pareto Front the solutions (locally) minimising $\Sigma_p C_p$ are naturally at the ends of the Front, and furthermore solutions at the ends of the Front will be more likely to have some C_p values small and others large. This matches with the way that the observed solutions minimising TPD do indeed sequence the projects. That is, when the projects are roughly equal size, and the C_p Pareto Front is concave, then it becomes reasonable that the TPD objective prefers the end points, and so the candidate good solutions are more likely to be widely separated in the search space.

Also, notice that as the fraction of activities using the bottleneck resource 2 (that is the value of 2x) increases, then the C_p-Pareto Front becomes more concave, and the TPD-driven solutions become different from the TMS-driven ones. This is consistent with the our general experience of the properties of MISTA instances; the shared global resource tends to be a bottleneck, and consequently they show the approximate ordering as seen in Fig. 1. Future work could well use such features related to the tightness of bottleneck resources to predict these effects, and so be used to select appropriate algorithm components. We also remark that the interesting structure of the tradeoff between the completion times does suggest that future work might well use multi-objective methods, e.g. see [10], and of course could study the effect of different weights for the completion times.

The simple example above is atypical in that the two projects are identical. However, if the projects are similar in size then it is reasonable to expect that the C_p-Pareto Front may be similar in overall structure but slightly distorted. When the front is concave but still roughly symmetric with respect to swapping the C_p, then the TPD-driven solutions are likely to still be towards the ends of the Front and so widely separated. However, because the front is not totally symmetric the ends are likely to have slightly different values of TPD. Hence, in such circumstances, it is reasonable to expect that there may be many widely separated local optima in TPD, and also that the optima correspond to different (approximate) orderings of the projects.

4.2. Monte-Carlo Tree Search

We have seen above that the TPD objective tends to favour a different solution structure than standard makespan minimisation, and in particular can drive a partial orderings of the projects, and there may be many good solutions that are widely separated in the solution space. Consequently, our construction method attempts to do a broad exploration of the solution space and to create initial sequences with a broadly similar partial order. The reason for including this was simply that we expected the partial ordering to lead to widely separated local minima in the search space, with large barriers between them – as swapping the order of two projects might mean going through interleaved intermediate states in which the TPD is worse. We included a method to sample the space of approximate project orderings, with the intent to avoid starting in a poor ordering and being trapped.

We expect that only a partial ordering is needed because we can assume that the subsequent improvement phase can make small or medium size adjustments to the overall project ordering structure. However, the improvement phase could have more difficulty, and take more iterations, if the general structure of the project ordering were not close to the structure expected in best solutions. Consequently, and for simplicity, we decided that a reasonable approximation would be to use a 3-way partition of the projects taken to correspond to 'start', 'middle' and 'end' parts of the overall project time. We required the numbers of projects in each part (of the partition) to be equal – or with a difference of at most one when the total number of projects is not a multiple of 3.

The problem then is how to quickly select good partition of the projects, and the method we selected is a version of Monte-Carlo Tree Search (MCTS) methods [3]. The general idea of MCTS is to search a tree of possibilities, but the evaluation of leaves is not done using a predefined heuristic, but instead by sampling the space of associated solutions. The sampling is performed using multiple invocations of a "rollout" which is designed to be fast and unbiased. It needs to be fast so that multiple samples can be taken; also rather than trying to produce "best solutions" it is usually designed to be unbiased – the (now standard) idea being that it should provide reliable branching decisions in the tree, but is not directly trying to find good solutions.

In our case, the tree search corresponds to decisions about which projects should be placed in which part of the partition. The rollout is a fast way to sample the feasible activity sequences consistent with the candidate choice for the partition of the projects. Specifically, the tree search works in two levels; firstly to select the projects to be placed in the end part and then to select the partition between the start and middle parts.

The first stage considers[5] 100 random choices for the partition of the projects, and then selects between these using 120 samples or the rollout.[6] The rollout consists of two main stages:

1. Randomly select a total ordering of the activities consistent with the precedences and with the candidate partitioning. Specifically, within each partition we effectively consider a dispatch policy that randomly selects between activities that are available to be scheduled because their preceding activities (if any) are already scheduled.
2. Randomly select modes for the activities. If the result is not feasible then this can only be because of the mode selection causing a shortfall in some non-renewable resources. Hence, it is repaired using a local search on the space of mode selections. We use moves that randomly flip one mode at a time, and an objective function that measures the degree of infeasibility by the shortfall in resources. Since the non-renewable resources are not shared between projects, this search turned out to be fast and reliable. As a measure of precaution, if the procedure fails to obtain a feasible selection of modes after a certain number of local search iterations, we restart it with a random selection of modes.

The first stage ends by making a selection of the best partitioning, using the quality of the 25th percentile of the final solution qualities (the best quartile) of the results of the rollouts. The 'end' part is then fixed to that of the best partition. The decision to fix the 'end' part also arose out of the observation that in good solutions the end projects are least interleaved. The MCTS proceeds to the second stage, and follows the same rollout procedure but this time to select the contents of the middle (and hence start) parts. This entire process usually completes within only a few seconds, and was used as the construction stage before the much longer improvement phase. We emphasise that the TPD-structure observed earlier had an important influence of the design of the neighbourhoods used in the key improvement phase – in particular, the motivation that some moves should affect the project level structure.

5. Neighbourhood operators

In this section, we describe our neighbourhood operators, used by several components during the improvement phase. The operators (also referred to as low-level heuristics or simply moves, depending on the algorithm which makes use of them) are categorised into three groups. This categorisation is mainly based on the common nature of the strategy the operators employ while manipulating the solution. Some of the moves are similar to those used by other submissions; see [61], and [16] and [56] for few examples. To make the paper self-contained, below we provide brief descriptions of all the moves used in our algorithm.

---

[5] The parameters for the MCTS were the result of some mild tuning, and used in the competition submission, but of course are adjustable.

[6] The number 120 was selected so that the rollouts could be evenly distributed between 2, 4, 6, 8 or 12 cores of the machine.

We guarantee that all of our operators preserve feasibility of the solution. Also, all of the moves are randomised so that they could be repeatedly used in "simulated annealing"-like improvement or applied as mutation operators. All the random selections are made at uniform unless specified otherwise.

5.1. Activity-level operators

Operators in this category involve basic operations, widely used in the literature, such as changing the mode of a single activity or swapping the positions of two activities. On top of that, we implemented a limited first improvement local search procedure for each of the basic moves.

To describe the moves in this category, we will need additional notations. Let $pos(j)$ be the position of activity $j \in A$ within a given solution. If activity j is shifted to a different position, we say that the feasible range of its new position is $[\ell(j), u(j)]$, where $\ell(j)$ and $u(j)$ can be computed as follows:

$$\ell(j_1) = \begin{cases} \max_{j \in Pred(j_1)} pos(j) + 1 & \text{if } Pred(j_1) \neq \emptyset, \\ 1 & \text{otherwise} \end{cases} \tag{7}$$

and

$$u(j_1) = \begin{cases} \min_{j \in Succ(j_1)} pos(j) - 1 & \text{if } Succ(j_1) \neq \emptyset, \\ n & \text{otherwise,} \end{cases} \tag{8}$$

where $Succ(j_1)$ is the set of successors of j_1.

- SWAP ACTIVITIES: swap two activities in the sequence. Select an activity $j_1 \in A$ randomly. Select another activity $j_2 \neq j_1 \in A$ randomly such that $\ell(j_1) \leq pos(j_2) \leq u(j_1)$. If $\ell(j_2) \leq pos(j_1) \leq u(j_2)$ then swap j_1 and j_2. Otherwise leave the solution intact.
- SHIFT: shift an activity to a new location in the sequence. Shift a randomly selected activity $j \in A$ to a new position randomly selected from $pos(j) \in [\ell(j), u(j)]$.
- CHANGE MODE: change the mode of a single activity. Select an activity j randomly at uniform, and if $|\mathcal{M}_j| = 1$ then leave the solution intact. Otherwise select a new mode $m \neq M_j \in \mathcal{M}_j$ and, if this does not lead to a violation of non-renewable resource constraints, update M_j to m.
- FILS SWAP ACTIVITIES: apply the first improvement local search (FILS) procedure based on the swap move. The operator has one parameter: the width $W > 1$ of the window to be scanned. Select an activity $j_1 \in A$ randomly. Define the window $[\ell', u']$ for activity j_2 as follows. If $u(j_1) - \ell(j_1) < W$, let $\ell' = \ell(j_1)$ and $u' = u(j_1)$. Otherwise select $\ell' \in [\ell(j_1), u(j_1) - W + 1]$ and set $u' = \ell' + W - 1$. Then, for every $j_2 \in A \setminus \{j_1\}$ such that $\ell' \leq pos(j_2) \leq u'$, attempt to swap j_1 and j_2 (see SWAPACTIVITIES). If the attempt is successful and it reduces the objective value of the solution then accept it and stop the search. Otherwise roll back the move and proceed to the next j_2 if any.
- FILS SHIFT: apply the first improvement local search (FILS) procedure based on the shift move. The operator is implemented very similarly to FILS SWAPACTIVITIES, i.e. it attempts to shift a randomly selected activity to an new position within a window of a given size.
- FILS CHANGE MODE: apply the first improvement local search (FILS) procedure based on the change mode move (the operator has no parameters). Select an activity $j \in A$ randomly. For $m \in \mathcal{M}_j \setminus \{M_j\}$, produce a new solution by setting $M_j = m$. If the resulting solution is feasible and provides an improvement over the original solution, accept the move and stop the local search. Otherwise proceed to the next $m \in \mathcal{M}_j \setminus \{M_j\}$ if any.

5.2. Ruin & recreate operators

The Ruin & Recreate (R&R) operators are widely used in metaheuristics as mutations or strong local search moves but, to the best of our knowledge, they are relatively new to serial generation in scheduling. As the name suggests, such operators have two phases: the "ruin" phase removes some elements of the solution and the "create" phase reshuffles those elements (for example, randomly) and then inserts them back. The number of elements to remove and re-insert is a parameter of a R&R operator, which controls its strength (i.e. the average distance between the original and the resulting solutions).

We implemented three different types of R&R operators, and several strategies to select activities involved in the move. One can arbitrary combine any type of the R&R operator with any activity selection strategy.

The move types are RESHUFFLE POSITIONS, RESHUFFLE MODES and RESHUFFLE POSITIONS AND MODES. RESHUFFLE POSITIONS move removes all the selected activities $A' \subset A$ from the solution (leaving $|A'|$ gaps in the sequence) and then re-inserts them in a random order while respecting the precedence relations. To determine a feasible order, we compute which activities V^i can be placed in each gap $i = 1, 2, \ldots, |A'|$ (in terms of precedence relations between the activities in A' and activities in $A \setminus A'$), and also produce a precedence relations sub-graph induced by A'. Then we apply a backtracking algorithm. In each iteration, it fills one gap, starting from the earliest gaps in the sequence. For gap i it randomly selects an activity $j \in A'$ such that there are no incoming arcs to j in the precedence sub-graph. If such an activity exists, the algorithm removes j from A' and from the precedence sub-graph. Otherwise it rolls back to attempt another activity on the previous level of search. The

depth of roll backs is unlimited, i.e., in the worst case, the algorithm will perform depth first search of the whole search tree (observe that there is always at least one feasible arrangement of the activities).

RESHUFFLE MODES move changes the modes of selected activities while keeping their positions intact. Each of the selected activities $j \in A'$ is assigned a randomly chosen mode, either equal or not to the previous mode M_j. Observe that the new mode selection may cause infeasibility in terms of non-renewable resources. Thus, we use the multi-start metaheuristic, simply repeating the above step until a feasible mode selection is found.

RESHUFFLE POSITIONS AND MODES combines the above two moves, which is trivial to implement as modes feasibility is entirely independent of the sequence feasibility.

We implemented several activity selection strategies some of which exploit our knowledge of the problem structure.

- UNIFORM: as the name suggests, $A' \subset A$ is selected randomly at uniform. The number of elements in A' is a parameter of the move.
- PROJECT: the activities A' are selected within a single project, i.e. $A' \subset A_p$. The project $p \in P$ is selected randomly.
- LOCAL: the selection of activities is biased to those scheduled near a certain time slot in the direct representation D. We randomly sample the activities accepting an activity j with probability

$$probability(j) = \frac{1}{\frac{|T_j - \tau|}{width} + 1} , \tag{9}$$

where $0 \leq \tau \leq f_m(D)$ (see (5)) is the randomly selected time slot (the centre of the distribution) and $width$ is a parameter defining the spread of the distribution. The sampling stops when A' reaches the prescribed cardinality.

- GLOBAL RESOURCE DRIVEN: the selection of activities is biased to the ones scheduled to time slots that under-utilise the global resources. The rationale is that global resources usually present a bottleneck in minimising the project completion times, and inefficiencies in the global resources consumption should be addressed when polishing the solution. In this selection strategy, as in LOCAL SELECTION, we use random sampling, but the acceptance probability for an activity $j \in A$ is defined by

$$probability(j) = \frac{\sum_{k \in \mathscr{G}^\rho} remaining_k(T_j)}{\sum_{k \in \mathscr{G}^\rho} G_k^\rho} , \tag{10}$$

where $remaining_k(t)$ is the remaining (unutilised) capacity of global resource k at the time slot t.

- ENDING BIASED: the selection of activities is biased to the last activities within the projects. The rationale is that, in an unpolished solution, the completion of a project might be improved by careful packing of the last activities such that all the last activities end at roughly the same time, effectively maximising the utilisation of the local resources and redistributing the global resources between projects. As in LOCAL SELECTION and GLOBAL RESOURCE DRIVEN SELECTION, we use random sampling of activities. The probability of accepting an activity $j \in A$ is

$$probability(j) = \frac{project - pos(j)}{|A_p|} , \tag{11}$$

where p is the project of activity j and $project - pos(j)$ is the position of activity j among the activities of project p.

5.3. Project-level operators

It was shown in Section 4 that the TPD objective function tends to create a partial ordering of projects in high-quality solutions. The danger, however, is that the ordering to which the solution converges might be sub-optimal. Consider two well-polished solutions S_1 and S_2 having different project orderings. The distance between S_1 and S_2 is likely to be significant with respect to the activity-level and R&R operators as many activities need to be moved to convert S_1 into S_2. Moreover, the transitional solutions in such a conversion will have significantly poorer quality compared to that of S_1 and S_2. This indicates that, once the algorithm converges to a certain project ordering, it needs a lot of effort to leave the corresponding local minimum and change the project ordering.

The project-level operators are designed to overcome such barriers, allowing faster exploration of the rough landscape of the MRCMPSP. They perform on the project level and thus one move of any of operators from this category usually results in a change in the project ordering. That being said, the project-level moves are likely to corrupt the solution causing many activity-level inefficiencies that need to be treated with activity-level and R&R moves. Thus, they ought to be used rarely within local search, but they can serve well as mutation operators.

In some of the project-level moves we use the formal concept of project ordering $P(S)$. To extract $P(S)$ from a solution S, we compute the "centre of mass" $centre(p)$ for each project $p \in P$ as

$$centre(p) = \frac{1}{|A_p|} \sum_{j \in A_p} pos(j)$$

and define the ordered set $P(S)$ of projects according to their centres of mass.

- SWAP TWO PROJECTS: swap two randomly selected projects in the sequence. Select two projects $p_1 \neq p_2 \in P$ randomly. Remove all the activities belonging to projects p_1 and p_2 from the sequence. Fill the gaps with all the p_2 activities and then all the p_1 activities preserving the original order within each of the projects. If, in the original solution, p_1 was located earlier than p_2 then the move swaps the projects (for instance, (1, 1, 1, 2, 1, 2, 2, 2) turns into (2, 2, 2, 2, 1, 1, 1, 1)). Otherwise it simply separates them more clearly (for instance, (2, 2, 2, 1, 2, 1, 1, 1) turns into (2, 2, 2, 2, 1, 1, 1, 1)).
- SWAP NEIGHBOUR PROJECTS: swap two projects adjacent in the project ordering. Extract the project ordering $P(S)$, randomly select $1 \leq i < q$ and set $p_1 = P(S)_i$ and $p_2 = P(S)_{i+1}$. Then apply the SWAP TWO PROJECTS move.
- COMPRESS PROJECT: place all the activities of a project adjacently to a new location x in the sequence while preserving their original ordering. Randomly select a project $p \in P$ and remove all the activities $j \in P_p$ from the sequence, squeezing the gaps (the resulting sequence will contain $n - |P_p|$ activities). Insert all the activities $j \in P_p$ consecutively at position $\lceil x(n - |P_p|) \rceil$, where the parameter $0 \leq x \leq 1$ is the new relative location of P_p.
- SHIFT PROJECT: shift all the activities of a project by some offset. Randomly select a project $p \in P$ and calculate $pos_{\min} = \min_{j \in P_p} pos_j$ and $pos_{\max} = \max_{j \in P_p} pos_j$. Randomly select an offset $-pos_{\min} < \delta n - pos_{\max}$ and shift every activity $j \in P_p$ by δ positions toward the end of the sequence.
- FLUSH PROJECTS: flush the activities of one or several projects to the beginning or ending of the sequence. Compute the project ordering $P(S)$ and select x consecutive projects P' from $P(S)$, where $1 \leq x < |P|$ is a parameter. Flush all the activities in projects P to either the beginning or the ending of the sequence (defined by an additional parameter of the move).

6. Improvement phase

Most of the runtime of our algorithm is spent on improving the initial solutions. We use a multi-threaded implementation of a simple memetic algorithm with a powerful local search procedure based on a hyper-heuristic which controls moves discussed in the previous section.

6.1. Memetic algorithm

A genetic algorithm is a population based metaheuristic combining principles of natural evolution and genetics for problem solving [53]. A pool of candidate solutions (individuals) for a given problem is evolved to obtain a high quality solution at the end. A fitness function is used to measure the quality of each solution. Mate/parent selection, recombination, mutation and replacement are the main operators of an evolutionary algorithm. However, the usefulness of recombination is still under debate in the research community [13,37]. A recent study showed that recombination can be useful at a certain stage during the search process, if the mutations do not change the quality of resultant individuals leading to a population containing different individuals with the same fitness [55]. The choice of the recombination operator can influence the best setting for the rate of mutation depending on the problem dealt with. Although the study is rigorous, it is still limited considering that some benchmark functions, such as ONEMAX are used for the proofs. A memetic algorithm (MA) hybridises a genetic algorithm with local search which is commonly applied after mutation on the new individuals [39,40]. Many improvements for MAs have been suggested, for example the population sizing [24] and interleaved mode of operation [46]. MAs have been successfully applied to many different problems ranging from generalised travelling salesman [19] to nurse rostering [43].

The improvement phase of our algorithm is controlled by a simple multi-threaded MA which manages the solution pool and effectively utilises all the cores of the CPU. Our MA is based on quantitative adaptation at a local level according to the classification in [42].

Within the MA, we use a powerful local search procedure that takes a few seconds on a single core to converge to a good local minimum. (Running the local search for a longer time still improves the solution but the pace of improvements slows down.) As the local search procedure has to be applied to every solution of the population in each generation, and the local search is by far the most intensive time consumer in our algorithm, the total running time of the algorithm can be estimated as

$$\frac{num-gen \cdot pop-size \cdot ls-time}{cores},$$

where $num-gen$ is the number of generations, $pop-size$ is the size of the population, $ls-time$ is the time taken by the local search procedure on one core per solution and $cores$ is the number of available CPU cores. Our algorithm is designed to converge within a few minutes. This implies that, to have a sufficient number of generations, the size of the population ought to be of the same order as the number of cores. On the other hand, as the memetic algorithm requires a barrier synchronisation at each generation, and our local search procedure cannot utilise any more than one CPU core, the size of the population has to be $pop-size = cores \cdot i$ to fully utilise all the cores, where i is a positive integer. In our implementation, we fixed the running time of the local search procedure to five seconds, and the size of the population to the number of CPU cores available to the algorithm.

Because of the small population size and limited number of generations, we decided to use a simple version of the MA, see Algorithm 6. We denote the population as $\mathcal{S} = \{S_1, S_2, \ldots, S_{cores}\}$, and the subroutines used in the algorithm are as follows:

Algorithm 6: Improvement phase.

---

1 $\gamma \leftarrow 0$;
2 **for** $i \leftarrow 1, 2, \ldots, |\mathcal{S}|$ **do**
3 | $S_i \leftarrow Construct()$;
4 **end**
5 **while** $elapsed - time \leq given - time$ **do**
6 | **if** $elapsed - time \geq 0.7 given - time$ **then**
7 | | $\gamma \leftarrow 0.000001$ (enable secondary objective function);
8 | **end**
9 | **for** $i \leftarrow 1, 2, \ldots, |\mathcal{S}|$ *(multi-threaded)* **do**
10 | | $S_i \leftarrow LocalSearch(S_i)$;
11 | **end**
12 | **for** $i \leftarrow 1, 2, \ldots, |\mathcal{S}|$ **do**
13 | | **if** $Accept(S_i) = false$ **then**
14 | | | $X \leftarrow Select(\mathcal{S})$;
15 | | | $S_i \leftarrow Mutate(X)$;
16 | | **end**
17 | **end**
18 **end**

- *Construct*() returns a new random solution with the initial partial project sequence obtained during the construction phase (see Section 4).
- *Accept*(S_i) returns *true* if the solution S_i is considered 'promising' and *false* otherwise. The function returns *false* in two cases: (1) $f(S_i) > 1.05 f(S_{i'})$ for some $i' \in \{1, 2, \ldots, |\mathcal{S}|\}$ or (2) the solution was created at least three generations ago and S_i is among the worst three solutions. Ranking of solutions is performed according to $f_d(S_i) + idle$, where *idle* is the number of consecutive generations that did not improve the solution S_i.
- *Select*(S) returns a solution from the population chosen with the tournament selection based on two randomly picked individuals.
- *Mutate*(X) returns a new solution produced from solution X by applying a mutation operator. The mutation operator to be applied is selected randomly and uniformly among the available options:
  - Apply the RESHUFFLE POSITIONS AND MODES move with the LOCAL selection strategy (see Section 5.2). The cardinality of the selection is $|A'| = 3$. Repeat the procedure 20 times, each time randomly selecting the centre of distribution $1 \leq \tau \leq f_m(D)$, see (9). The *width* parameter is taken as $width = 0.1 f_m(D)$.
  - Apply the SWAP NEIGHBOUR PROJECTS operator once (see Section 5.3).
  - Apply the FLUSH PROJECTS operator with the number of selected projects being one and flushing to the end of the sequence (see Section 5.3).
  - Apply the FLUSH PROJECTS operator with the number of selected projects being two and flushing to the beginning of the sequence. It was noted that the project ordering is usually less explicit at the beginning of the sequence in good solutions, and for that reason we did not use flushing single projects to the beginning in our mutations.
  - Same as the last mutation except that the number of selected projects is three.

It was noted in Section 4 that the components $f_d(D)$ and $f_m(D)$ of the objective function $f(D)$ (see (6)) are competing. Indeed, minimisation of the total makespan favours solutions with projects running in parallel as such solutions are more likely to achieve higher utilisation of the global resources. At the same time, minimisation of the TPD favours solutions with the activities grouped by projects. Hence, the second objective creates a pressure for the local search that pushes the solutions away from the local minima with regards to the first (main) objective. To avoid this effect, we initially disable the second objective ($\gamma \leftarrow 0$, see Section 2) and re-enable it only after 70% of the given time is elapsed.

The parameters of the memetic algorithm (such as the ones used in the *Accept*(S_i) function, or the number of solutions in the tournament in *Select*(S)) have been chosen using a parameter tuning procedure. It should be noted, however, that the algorithm is not very sensitive to any of those parameters, which makes it efficient on a wide range of instances, and this conclusion was supported by our empirical tests, see Section 7.

6.2. A Dominance based hyper-heuristic using an adaptive threshold move acceptance

There is a growing interest towards self-configuring, self-tuning, adaptive and automated search methodologies. Hyper-heuristics are such high level approaches which explore the space of heuristics (i.e. move operators) rather than the solutions in problem solving [5]. There are two common types of hyper-heuristics in the scientific literature [7]: *selection* methodologies that choose/mix heuristics from a set of preset low-level heuristics (which can both improve or worsen the solution) and attempt to control those heuristics during the search process; and *generation* methodologies that aim to build

new heuristics from a set of preset components. The main constituents of an iterative selection hyper-heuristic are *heuristic selection* and *move acceptance* methods. At each step, an input solution is modified using a selected heuristic from a set of low-level heuristics. Then the move acceptance method is used to decide whether to accept or reject the new solution. More on different types of hyper-heuristics, their components and application domains can be found in [5,6,44,52].

In this study, we combine two selection hyper-heuristics under a single point based iterated local search framework by employing them adaptively in a staged manner. The approach extends the heuristic selection and move acceptance methods introduced in [45] and [27], respectively. The pseudocode of our adaptive iterated multi-stage hyper-heuristic is given in Algorithm 7. Lines 5–21 and 22–25 provide the high level design of the first and second stage hyper-heuristics, respectively.

Algorithm 7: *LocalSearch*(S_i) .

---

1 Let $LLH_{all} = \{LLH_1, LLH_2, \ldots, LLH_{\mathcal{M}}\}$ represent set of all low level heuristics with each heuristic being associated with a score, initially set to 1;

2 Let S_{best} represent the best schedule;

3 $S \leftarrow S_i$; $S_{best} \leftarrow S_i$; $LLH \leftarrow LLH_{all}$; $\epsilon \leftarrow \epsilon(S_{best})$;

4 **repeat**

5 $h \leftarrow SelectLowLevelHeuristic(LLH)$;

6 $S' \leftarrow h(S)$;

7   **if** $f(S') < f(S)$ **then**

8 $S \leftarrow S'$;

9   **if** $f(S') < f(S_{best})$ **then**

10 $S_{best} \leftarrow S'$;

11   **end**

12   **end**

13   **else**

14   **if** $f(S') < (1 + \epsilon)f(S_{best})$ **then**

15 $S \leftarrow S'$;

16   **end**

17   **end**

18   **if** $NoImprovement(elapsed - time_2)$ **then**

19 $S \leftarrow S_{best}$;

20 $\epsilon \leftarrow \epsilon(S_{best})$;

21   **end**

22   **if** $NoImprovement(elapsed - time_3)$ **then**

23 $\epsilon \leftarrow \epsilon(S_{best})$;

24 $(S, LLH) \leftarrow SecondStage(S_{best}, LLH_{all}, elapsed - time_1)$;

25   **end**

26 **until** $timeLimitExceeded(elapsed - time_1)$;

27 **return** S_{best};

6.2.1. First stage hyper-heuristic

The first stage hyper-heuristic maintains an active pool of low-level heuristics $LLH \subseteq LLH_{all}$ and a score $score_h$ associated with each heuristic $h \in LLH$. In each iteration, it randomly selects a low-level heuristic from the active pool with probability of picking $h \in LLH$ being proportional to $score_h$ (line 5). Then the selected heuristic is applied to the current solution (line 6). Initially, each heuristic has a score of 1, hence the selection probability of a heuristic is equally likely. The first stage hyper-heuristic always maintains the best solution found so far, denoted as S_{best} (lines 9–11) and keeps track of the time since last improvement.

The move acceptance component of this hyper-heuristic (lines 7–17) is an adaptive threshold acceptance method controlled by a parameter ϵ accepting all improving moves (lines 7–12) and sometimes non-improving moves (lines 14–16). If the quality of a new solution S' is better than $(1 + \epsilon)f(S_{best})$ (line 14), even if this is a non-improving move, S' gets accepted becoming the current solution S. Whenever S_{best} can no longer be improved for $elapsed - time_2$ (in our implementation $elapsed - time_2 = 1$ sec), the parameter ϵ gets updated as follows:

$$\epsilon(S_{best}) = \frac{\lceil \log(f(S_{best})) \rceil + rand}{f(S_{best})} \tag{12}$$

where $1 \leq rand \leq \lceil \log(x) \rceil$ is selected randomly at uniform. Note that 0 is a lower bound for $f(S)$ (see Section 2) and, hence, the algorithm will terminate if $f(S_{best}) = 0$.
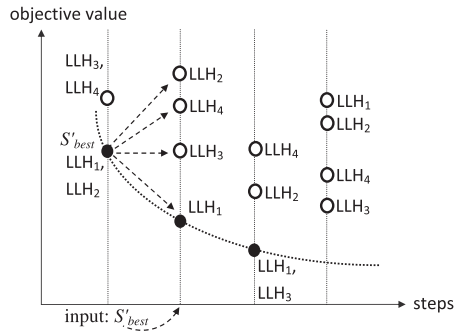
Fig. 3. An illustration of how the second stage greedy hyper-heuristic operates.

6.2.2. Second stage hyper-heuristic

The second stage hyper-heuristic dynamically starts operating (lines 22–25 of Algorithm 7) whenever there is no improvement in $f(S_{\text{best}})$ for *elapsed – time$_3$* (in our implementation *elapsed – time$_3$* = 3 sec) in line 22. The hyper-heuristic in this stage updates the active pool *LLH* of heuristics. $LLH \subseteq LLH_{\text{all}}$ is formed based on the idea of a dominance-based heuristic selection as introduced in [45] reflecting the trade-off between the objective value achieved by each low-level heuristic and number of steps involved. The method considers that a low-level heuristic producing a solution with a small improvement in a small number of steps has a similar performance to a low level heuristic generating a large improvement in large number of steps. This hyper-heuristic not only attempts to reduce the set of low-level heuristics but also assigns a score for each low-level heuristic in the reduced set, dynamically. Those scores are used as a basis for the heuristic selection probability of each low level heuristic to be used in the first stage hyper-heuristic.

In the second stage hyper-heuristic, firstly, ϵ is updated in the same manner as in the first stage hyper-heuristic and never gets changed during this phase. Then a greedy strategy is employed using all heuristics in LLH_{all} for a certain number of steps. Each move in this stage is accepted using the same adaptive threshold acceptance method as described in Section 6.2.1. LLH_{all} is partitioned into three subsets LLH_{small}, LLH_{medium}, LLH_{large} considering the number of activities processed (e.g., number of swaps) by a given heuristic.

Small: Swap activities, Shift and Change mode.
Medium:

- Reshuffle modes and Reshuffle positions and modes with the Uniform selection strategy;
- Reshuffle modes and Reshuffle positions and modes with the Local selection strategy;
- Reshuffle positions and modes with the Global resource driven selection strategy;
- Reshuffle positions and modes with the End biased selection strategy;
- Reshuffle positions and modes with the Project selection strategy;
- FILS swap activities, FILS shift and FILS change mode;

  Large:

- Flush projects applied to one project; the direction is picked randomly;
- Swap two projects;
- Compress project;
- Shift project.

At each step, each low-level heuristic is applied to the same input solution for a fixed number of iterations ($5n/q$ for small, n/q for medium and 1 for large heuristics).[7] If a low-level heuristic produces a new solution identical to the input, that invocation is ignored. Otherwise, the objective of the new solution together with the low-level heuristic which produced that solution gets recorded. Once all heuristics are applied to the same input and get processed at a given step, the best resultant solution propagates as input to the next greedy step.

A solution is considered to be a 'non-dominated' solution at a given step, if the quality of the solution is better than the best solution obtained in the previous steps. The active pool *LLH* of low-level heuristics is formed using the heuristics achieving the non-dominated solutions obtained at each step during the greedy phase. The score of a low-level heuristic is set to the number of non-dominated solutions that it produced. Note that a non-dominated solution could be generated by multiple low-level heuristics, in which case, their scores get increased.

Fig. 3 illustrates a run of the second stage hyper-heuristic with $LLH_{\text{all}} = \{LLH_1, LLH_2, LLH_3, LLH_4\}$ for four steps. The plot shows that the set of non-dominated solutions contains 3 points marked in black. The best solution achieved by LLH_3 in the last step is ignored, since that solution is dominated by the first three best solutions obtained in the first three

---

[7] Similarly to the memetic algorithm parameters, this partition of LLH_{all} and the associated numbers of iterations were selected by intuition and parameter tuning.
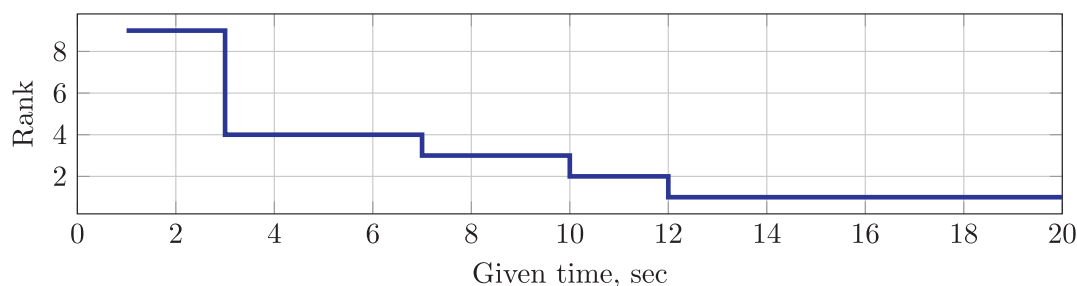
Fig. 4. The rank achieved by our algorithm with various time limits in the assumption that the other algorithms are given 300 s. Given just 12 s, our algorithm would already achieve the rank 1.

steps. The first, second and third points in the set of non-dominated solutions are associated with $\{LLH_1, LLH_2\}$, $\{LLH_1\}$ and $\{LLH_1, LLH_3\}$, respectively. Hence, LLH becomes $\{LLH_1, LLH_2, LLH_3\}$. The scores of LLH_1, LLH_2 and LLH_3 are assigned to 3, 1 and 1, respectively, yielding a heuristic selection probability of 60%, 20% and 20% for each heuristic. More details on the components of the proposed hyper-heuristic can be found in [25,26].

7. Experiments

In this section we report the results of our computational experiments as well as discuss the MISTA 2013 Challenge and its outcomes. For additional analysis of the performance of our and other submitted algorithms we refer to [61].

7.1. Test instances

In our experiments, we used three sets of MRCPSP instances provided by the organisers of the MISTA 2013 Challenge. The first set (set A) was provided at the beginning of the competition. The second set (set B) was provided after the qualification phase. The third set (set X) was hidden from the participants of the competition until the announcement of the results; it was used to evaluate the performance of the submitted algorithms. While designing and tuning our algorithm, we had no information on what properties would the hidden instances have, which drove our decisions towards self-adaptive methods and a rich range of moves.

All the three sets of instances were produced by combining PSPLIB benchmark instances [28] and can be downloaded from allserv.kahosl.be/mista2013challenge. In Table 1 we provide some properties of all the A, B and X instances.

7.2. Experimental results

During the competition, the submitted algorithms were tested by running each of the algorithms on each of the B and X instances 10 times. The results for X (hidden) instances were used to rank the teams. The detailed discussion of the ranking scheme and the analysis of the competition results are reported in [61].

To further analyse the performance of our algorithm, we ran it for 5 min on each of the B and X instances 2500 times. We used a machine with similar configuration[8] to that employed by the competition organisers.

The aggregated results of our experiment are reported in Table 2. The average and best solutions are obtained from our 2500 runs, while the competition ("Comp" column) result is the best solution found during the final phase of the competition by our or some other algorithm.

It is worth mentioning here that during the final phase of the competition where the submitted algorithms were tested on B and X instances, our approach outperformed, on average, all the other algorithms in 18 out of 20 cases, and produced the best solution for 17 instances. [61] note the consistency of the performance of our algorithm, and also provide ranks with several different ranking methodologies, showing that our approach would win the competition even if the rules of the competition were different.

We conducted an additional experiment to investigate the effect of the runtime on the ranking results, and so an indication of whether or not the code improvements as discussed in Section 3.1, might have been the predominant reason, as opposed to the algorithm itself, of achieving the first rank during the competition. In this experiment, we ran our algorithm (in the style which was used during the competition) with reduced time limits. After each run, we determined the rank of our algorithm among all other competitors (the data required for this ranking was received from competition organisers). Fig. 4 shows the result of the experiment for various time limits ranging from 1 to 25 s. For the first two or three seconds the algorithm runs MCTS thus not generating high quality solutions (recall that the goal of MCTS is to find a high-quality partial ordering of projects). However, as soon as the memetic algorithm starts, the solution quality rapidly improves. In fact, our algorithm manages to achieve the first rank after just 12 s. Given that the time limit during the competition was

---

[8] Intel i7 3.2 GHz CPU, 16GB of RAM, Microsoft Windows 7 x64 operating system.

Table 1

Characteristics of the instances: q is the number of projects, n is the number of activities, avg d is the average duration of activities in all possible modes, avg $|\mathcal{M}|$ is the average number of modes for each activity, avg $|Pred|$ is the average number of the predecessor activities for each activity, avg $|\mathcal{R}^{\rho}|$ is the average number of the local renewable resources per project, avg $|\mathcal{R}^{v}|$ is the average number of the local non-renewable resources per project, $|\mathcal{G}^{\rho}|$ is the number of global renewable resources, avg R^{ρ} is the average renewable resource capacities, avg C^{v} is the average non-renewable resource capacities, avg G^{ρ} is the average global renewable resource capacities, avg CPD is the average CPD per project, H is the upper bound on the time horizon. The last two columns give an indication of the properties of the network (precedence graph); max height is the length of the longest path, e.g. see https://users.soe.ucsc.edu/~manfred/pubs/J2.pdf and avg SP is a measure of the 'series vs. parallel' nature [57].

| Instance	q	n	avg d	avg $|\mathcal{M}|$	avg $|Pred|$	avg $|\mathcal{R}^{\rho}|$	avg $|\mathcal{R}^{v}|$	$|\mathcal{G}^{\rho}|$	avg R^{ρ}	avg C^{v}	avg G^{ρ}	avg CPD	H	max height	avg SP
A-1	2	20	5.53	3	1.20	1	2	1	18.5	51.3	16.0	14.5	167	4	0.33
A-2	2	40	4.63	3	1.70	1	2	1	23.5	117.3	23.0	22.5	303	8	0.34
A-3	2	60	5.51	3	1.73	1	2	1	38.5	154.8	49.0	33.5	498	9	0.26
A-4	5	50	4.37	3	1.20	1	2	1	15.2	44.9	12.0	14.2	409	5	0.40
A-5	5	100	5.43	3	1.70	1	2	1	24.0	92.4	13.0	23.0	844	8	0.32
A-6	5	150	5.13	3	1.73	1	2	1	23.8	175.4	13.0	25.6	1166	9	0.26
A-7	10	100	6.03	3	1.20	0	2	2	0.0	48.4	11.5	16.8	787	6	0.37
A-8	10	200	5.67	3	1.70	0	2	2	0.0	110.8	22.5	24.6	1569	9	0.31
A-9	10	300	5.61	3	1.73	1	2	1	27.5	168.0	27.0	29.6	2353	11	0.27
A-10	10	300	5.53	3	1.73	1	2	1	25.9	158.2	15.0	30.7	2472	10	0.26
B-1	10	100	5.33	3	1.20	1	2	1	17.1	44.8	11.0	12.9	821	5	0.34
B-2	10	200	5.67	3	1.70	0	2	2	0.0	94.0	21.0	23.9	1628	7	0.28
B-3	10	300	5.52	3	1.73	1	2	1	28.5	144.4	28.0	29.5	2391	10	0.26
B-4	15	150	5.03	3	1.20	1	2	1	17.5	52.3	10.0	15.8	1216	6	0.36
B-5	15	300	6.02	3	1.70	1	2	1	20.7	99.6	17.0	22.5	2363	8	0.31
B-6	15	450	4.62	3	1.73	1	2	1	25.0	141.8	34.0	31.1	3582	11	0.26
B-7	20	200	4.87	3	1.20	1	2	1	14.7	49.6	10.0	15.4	1596	5	0.37
B-8	20	400	5.48	3	1.70	0	2	2	0.0	104.7	10.0	23.7	3163	8	0.30
B-9	20	600	5.31	3	1.73	1	2	1	26.6	154.8	10.0	30.1	4825	12	0.26
B-10	20	420	5.28	3	1.66	0	2	2	0.0	115.9	18.0	24.5	3340	12	0.31
X-1	10	100	5.53	3	1.20	0	2	2	0.0	47.6	12.5	14.9	783	5	0.36
X-2	10	200	5.53	3	1.70	1	2	1	24.0	105.6	14.0	23.0	1588	8	0.32
X-3	10	300	4.98	3	1.73	1	2	1	27.9	167.0	33.0	29.9	2404	11	0.27
X-4	15	150	5.70	3	1.20	0	2	2	0.0	54.5	13.5	14.9	1204	5	0.36
X-5	15	300	5.52	3	1.70	1	2	1	19.9	100.1	12.0	23.6	2360	8	0.32
X-6	15	450	5.49	3	1.73	1	2	1	24.6	163.7	20.0	29.9	3597	10	0.26
X-7	20	200	5.03	3	1.20	1	2	1	13.9	53.9	10.0	15.0	1542	5	0.33
X-8	20	400	5.53	3	1.70	1	2	1	22.2	104.5	15.0	24.5	3217	8	0.32
X-9	20	600	5.54	3	1.73	1	2	1	23.9	146.3	11.0	28.9	4699	10	0.26
X-10	20	410	5.30	3	1.65	1	2	1	20.0	101.2	10.0	24.1	3221	9	0.30

Table 2
Summary of experimental results. (Note that the results for the A instances are for the algorithm of the first stage of the competition, and are included only for completeness).

Instance	TPD			TMS		
	Avg	Comp	Best	Avg	Comp	Best
A-1	1	1	1	23	23	23
A-2	2	2	2	41	41	41
A-3	0	0	0	50	50	50
A-4	65	65	65	42	42	42
A-5	155	153	150	105	105	103
A-6	141	147	133	808	96	99
A-7	605	596	590	201	196	190
A-8	292	302	272	153	155	148
A-9	208	223	197	128	119	122
A-10	880	969	836	313	314	303
B-1	352	349	345	128	127	124
B-2	452	434	431	167	160	158
B-3	554	545	526	210	210	200
B-4	1299	1274	1252	283	289	275
B-5	832	820	807	255	254	245
B-6	950	912	905	232	227	225
B-7	802	792	782	232	228	225
B-8	3323	3176	3048	545	533	523
B-9	4247	4192	4062	754	746	738
B-10	3290	3249	3140	455	456	436
X-1	405	392	386	143	142	137
X-2	356	349	345	164	163	158
X-3	329	324	310	193	192	187
X-4	960	955	907	209	213	201
X-5	1785	1768	1727	373	374	362
X-6	730	719	690	238	232	226
X-7	866	861	831	233	237	220
X-8	1256	1233	1201	288	283	279
X-9	3272	3268	3155	648	643	632
X-10	1613	1600	1573	383	381	373
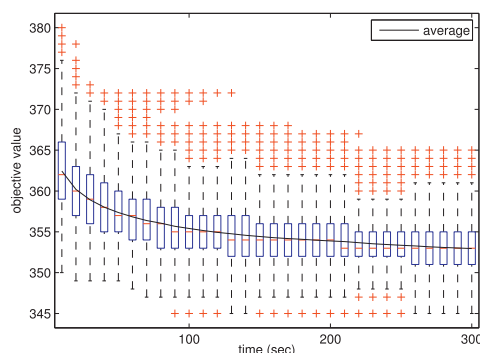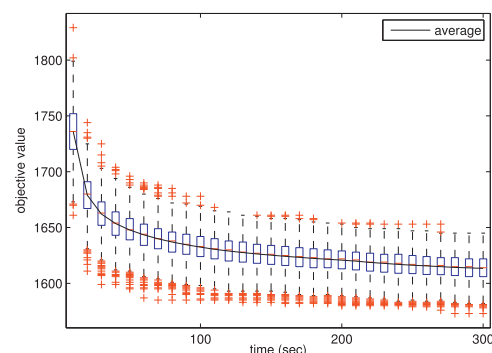
(a) B-1   (b) X-10

Fig. 5. Performance dynamics of the proposed approach. The horizontal axis corresponds to the time spent by the algorithm on optimisation and the vertical axis shows the solution quality. The results are averaged over 2500 runs and represented in the box plot form.

300 s, this means that the algorithm would have still ranked highest even if it had run 25 times slower. In contrast the improvements of Section 3.1 did not give that factor of a speed up. Assuming that the other submissions were reasonably implemented then we conclude that efficient implementation was not the only factor, but that the algorithm design also had a major role in the success of the algorithm.

To illustrate the performance of our approach in time, two boxplots are provided in Fig. 5a and b for instances B-1 and X-10, respectively. These instances have been chosen to demonstrate the performance of our algorithm on relatively small and large instances. However, our experiments show that the algorithm behaves in a similar manner with respect to the other instances. The central dividing line of each box in each of the figures presents the median objective value obtained by our algorithm. The edges of each box refer to 25th and 75th percentiles while the whiskers (demonstrated by a + marker) are the extreme objective values which are not considered as outliers. Also, the curve which passes through the plot demonstrates

the average performance of the algorithm. Our approach continuously makes improvement indicating that the algorithm does not prematurely converge.

It would obviously be interesting to link together the performance of the algorithm(s) with the features of the instances; for example, see [36,49] and others. There are many potential features that could be used to characterise the instances, and these have often been used as part of the process of generating instances, for example [12,29]. For example, one can characterise the precedence graph in various ways [15,23,51] and associated measures such as 'complexity' and 'Series-Parallel' indices could also be included. This would be far outside the scope of this paper, and warrants a future study. However, we can make some initial observations by analysing the instance properties in Table 1 and the results in Table 2, and in more detail in Table 33 of the final competition summary [63]. In particular, we focus on the property of the number of local renewable resources in the column "avg $|\mathcal{R}^\rho|$", and that there are none for instances B-2, B-8, B-10, X-1 and X-4. These seem to correlate with the performance in that the algorithm here only fails to rank first on instances B-2 and X-1, both of which have no local renewable resources. We remark that having no local renewables is a rather special case, and so if these were removed the relative performance would have further improved. However, it seems reasonably likely that the number of local renewable resources would be good instance feature for selecting algorithms, or for future studies comparing different algorithms in more details.

7.3. Results on single project (PSPLIB) instances

The main aim of this work is to develop methods for the MISTA benchmark problems having multiple projects. However, it is also of interest to study the methods on other scheduling benchmarks; in particular, the PSPLIB library[9] of instances [28] has been widely used and studied in the academic scheduling community. Unfortunately, despite it being long known that the majority of real world problems are multi-project [34], it seems that much of the work in project scheduling has been on single project instances, and PSPLIB does not currently contain any multi-project instances. Despite this limitation of PSPLIB, it has functioned well as a standard set of benchmarks, and so it is likely to interest how well the methods of this paper perform on these instances. The most relevant set of instances in PSPLIB is provided by the multi-mode single project instances.[10] The study of those instances is also limited considering the number of tasks which goes up to 30 tasks, whereas the MISTA competition included instances with up to 600 tasks. (Other PSPLIB instances are either much simpler as they are just single mode, or else introduce extra constraints of objectives for the timing, and so are outside the scope of this current paper.) Specifically, we studied the J30 instances from PSPLIB; they all have precisely 3 modes for each of the 30 tasks, with 2 renewable and 2 non-renewable resources, and we use the 552 of them that are satisfiable (a feasible solution is known).

For the special case of single project, minimising the TPD objective becomes the same as minimising the makespan, TMS. Hence, the PSPLIB multi-mode instances can be regarded as a special single-project case of the MISTA formulation and so our methods could be directly applied to these instances to minimise the makespan. However, doing so would be rather inefficient as many portions of our methods are designed to handle multiple-projects and the associated TPD objective. Hence, we partially specialised our code in order to handle these simpler PSPLIB instances. In particular, we inactivated the Monte-Carlo Tree Search as it is only there in order to handle the structures arising from the TPD objective, replacing it with a simple random generation heuristic. We also inactivated the neighbourhoods that are specific to working on a specific project. Since making comparisons to previous work also needed a very limited time period and just a single thread, we also disabled the memetic component as it was designed to exploit multi-cores and longer runs; to partially compensate for the resulting expected loss in diversity in the search, we also added a simple restart of the search every 10k generated schedules.

These instances have been studied in various works. [48] used a Genetic algorithm and also Forward-Backward Iterations (FBI) [33] along with a simple method to repair infeasible mode choices. [9] again used FBI along with a more expensive SAT-based method to ensure that only feasible mode decisions were made (recently, [50] reported that this was the best method on these instances). In these works, the standard fashion is to impose a limit on the number of schedules generated. The reported performance measure is then the gap, averaged over all instances, from either a known lower bound or else from the current "Best Known Solutions (BKS)". Often, the number of schedules permitted is rather small, such a 5–10k, which would take our code less than a second, and so would not really match the intent of our methods. Hence, in Table 3 we compare with previous results at 50k and 500k from Table 4 of [9]. We also note that with 500k schedules, and under 2s, our heuristic found the best-known solutions in more than 85% of instances, and in 99% it was within 1 time unit of the BKS. We also performed much longer runs to exploit the full power of all the algorithm components: up to 500M generated schedules within the concurrent memetic, though still only taking up to about 400 s, due to the concurrency, and the more efficient implementations (see Section 3). We were able to find 3 new best solutions; these have been added to PSPLIB, and so giving the only improvements to these multi-mode J30 instances since 2011.

Hence, we conclude that the methods here perform competitively on these multi-mode instances. We believe this gives evidence of the power and flexibility of the overall approach. This is particularly noteworthy, as our methods were originally

---

Table 3
Summary of experimental results on PSPLIB multi-mode J30 instances for 50k and 500k generated schedules. Giving the average gap from the lower bound (the lower the better) and the average running time (the lower the better). Note that the runtimes are only indicative as the older results will be on machines that might well be 2x slower.

Schedules generated:	50k		500k	
	Gap, %	Time, s	Gap, %	Time, s
This work	13.68	0.12	12.84	1.14
Coelho and Vanhoucke [9]	12.77	25.1	12.41	210.1
Peteghem and Vanhoucke [48]	13.31	2.46	13.09	18.03

designed for the more general multi-project case with the extra delay based objective, which has quite different properties, as already discussed in Section 4; the difference between minimising TPD and TMS can lead to a significant difference in the nature of the solutions, and reasonable algorithms. For example, both of the compared approaches [9,48] used the successful forward-backward iterations, however our methods for did not include it, because it is not naturally applicable to the primary objective being TPD. Including it in our suite of improvement operators, might well improve performance for the special case of improving makespan.

Finally, we remark that recent work on these multi-mode single project instances by [36] has considered the problem of selecting algorithms. This is perhaps the closest in spirit to our goals, in that the aim is to take multiple options for algorithms and then to use intelligent or machine learning techniques in order to make the selection. In contrast to a more traditional approach in which the exploration of the combinations of many components is done manually.

8. Conclusions

This paper described the components of a multi-component hybrid approach to the MRCMPSP. Broadly, our algorithm is built on the serial generation scheme, in which sequences of activities are used to construct final schedules for their quality evaluation. We use a two-phase construct-and-improve method to search the space of activity sequences, with a variety of novel contributions, including algorithm mechanisms specifically designed to handle the multi-project structure and associated objectives functions.

Firstly, the primary objective is a sum of completion times of the individual projects, and we give evidence and arguments that this tends to lead to an imbalance in the completion times; some projects finish much earlier than others. Hence, the nature of the problem itself is such that an approximate partial ordering of projects occurs in good solutions, and this is quite different from the structure expected with makespan as the primary objective. We expect such approximate partial ordering may well be common in real-world multi-project problems as well. Accordingly, in the construction phase, a novel Monte-Carlo Tree Search method is given that creates and selects initial solutions with an approximately-similar structure.

Secondly, in the improvement phase, multiple novel neighbourhoods are included that are designed to work at the 'project-level' and so enable any needed changing of the approximate partial ordering of the projects. These are complemented with a wide range of neighbourhood moves also designed to work at lower levels within projects. The improvement phase itself is organised and controlled using a novel hybrid of a memetic algorithm and a hyper-heuristic, and in a fashion that furthermore makes effective use of a multi-core machine when it is available.

Perhaps a distinguishing characteristic of the methods here is that there are far more neighbourhoods than in other submissions to the competition, or indeed the literature in general. From Table 6 in [61], other submissions have around 1–4 neighbourhoods but we have 13–17 (though many neighbourhoods have different variants, and are used in different stages, and so there is not a single meaningful count). At first this might seem to be a step towards extra complexity, and that might make the algorithm more difficult to use in real-world practice. However, in many problem domains there is a drive towards a larger number of neighbourhoods, for example, in realistic educational timetabling where 10–15 neighbourhoods is not uncommon (for example [27, and others]). Having many neighbourhoods is a natural reaction to the increasing complexity of the structure of the problem. They are designed to work on each level of the roughly hierarchical structure of many projects with each project containing many activities. We only eliminated neighbourhoods in a stage when they clearly never helped; instead preferring to keep neighbourhoods, with the intention that the diversity would help in robustness and effectiveness (though admittedly at the cost of 'neatness'). Also, the objective itself is a type of weighted completion time and so is sensitive to the internal structure of solutions. Hence, it seems reasonable that many neighbourhoods are needed; and this seems to be borne out by the much better performance. Furthermore, in the case of scheduling using the serial generation there is an advantage and opportunity in that the neighbourhoods are mostly 'lightweight' to implement. They work by modifying the activity sequence, and when the majority of the runtime is spent on the evaluation of a sequence, then designing and efficiently implementing many different neighbourhoods is much less onerous then many other problem domains.

Another significant difference to other submissions to the challenge, and as described in [61], it seems that many others in the competition used forward-backward iteration (FBI) methods. We did not use it due to the extra complexity, and

also uncertainty of how to best use it for the TPD (weighted completion time) rather than the standard makespan. Also, the results in [50] said it did not "distinguish between a good or bad performing metaheuristic". However, it would be interesting future work to also include FBI moves; possibly with appropriate modification for the TPD objective.

The effectiveness of the resulting hybrid algorithm was demonstrated when it convincingly won the MISTA 2013 challenge, outperforming, on average, the other algorithms on 18 of the 20 instances [61]. The many-component and many-neighbourhood structure was a deliberate decision on our part, and was aimed at giving a more robust solver. In particular, part of the design goal was that the methods would be robust and so work well on the hidden instances, and it did indeed do particular well on those; ranking first on 9 out of the 10 hidden instances. We also observed that the performance was weakest (though still high ranking) on those instances with no local renewable resources. Such instances are rather special, and arguably less realistic, however, it does suggest that there is good potential for future work to investigate the links between features of the instances and the algorithm performance. Nevertheless, Section 7.3 gives additional evidence of the robustness of our algorithm as even a much-reduced version performs well on the simple single-project but multi-mode instances from PSPLIB.

The broader contribution of this work is to give evidence for the general approach and methodology of investigating algorithms that consist of many components (such as many neighbourhoods, but also other aspects) and that aims to exploit the potential benefits of simple, but carefully controlled, interactions between them. It is possible that the algorithm could be simplified (e.g. by reducing the number of neighbourhood moves); however, we suspect that doing so is not worth the risk of it only working well on the seen set of instances, and would lose its robustness and hence performance on the unseen instances. Hence, we believe that the work gives evidence that individual components should not be prematurely discarded. Instead, ultimately, the decision as to which combination works best over a particular suite should be automated; that is, as a form of algorithm assembly, for example in the style of [1]. Of course, such assembly can also use a form of algorithm selection for the appropriate scheduling algorithm, based on the features of instances (e.g. [36,49]).

Such 'automated assembly' work would be outside the scope of this particular paper; however, by providing a broad range of options and components that have the proven potential to give a successful and robust solver, this work hence gives a good foundation for such studies in the MRCMSP, and other variants of project scheduling, and arguably intelligently-coordinated metaheuristic methods in general.

Acknowledgements

References

[1] L.C.T. Bezerra, M. López-Ibáñez, T. Stützle, Automatic design of evolutionary algorithms for multi-objective combinatorial optimization, in: T. Bartz-Beielstein, J. Branke, B. Filipič, J. Smith (Eds.), PPSN 2014, Lecture Notes in Computer Science, 8672, Springer, 2014, pp. 508–517.

[2] J. Blazewicz, J. Lenstra, A. Kan, Scheduling subject to resource constraints: classification and complexity, Discrete Appl. Math. 5 (1) (1983) 11–24.

[3] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton, A survey of Monte Carlo tree search methods, Comput. Intell. AI Games IEEE Trans. 4 (1) (2012) 1–43.

[4] P. Brucker, A. Drexl, R. Mohring, K. Neumann, E. Pesch, Resource-constrained project scheduling: notation, classification, models, and methods, Eur. J. Oper. Res. 112 (1) (1999) 3–41.

[5] E.K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, R. Qu, Hyper-heuristics: a survey of the state of the art, J. Oper. Res. Soc. 64 (12) (2013) 1695–1724, doi:10.1057/jors.2013.71.

[6] E.K. Burke, E. Hart, G. Kendall, J. Newall, P. Ross, S. Schulenburg, Hyper-heuristics: an emerging direction in modern search technology, in: F. Glover, G. Kochenberger (Eds.), Handbook of Metaheuristics, Kluwer, 2003, pp. 457–474.

[7] E.K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, J.R. Woodward, A classification of hyper-heuristics approaches, in: M. Gendreau, J.-Y. Potvin (Eds.), Handbook of Metaheuristics, International Series in Operations Research & Management Science, 57, 2nd, Springer, 2010, pp. 449–468.

[8] W. Chen, Y. Shi, H. Teng, X. Lan, L. Hu, An efficient hybrid algorithm for resource-constrained project scheduling, Inf. Sci. 180 (6) (2010) 1031–1039. Special Issue on Modelling Uncertainty.

[9] J. Coelho, M. Vanhoucke, Multi-mode resource-constrained project scheduling using rcpsp and sat solvers, Eur. J. Oper. Res. 213 (1) (2011) 73–82.

[10] C.A. Coello Coello, A comprehensive survey of evolutionary-based multiobjective optimization techniques, Knowl. Inf. Syst. 1 (3) (1999) 269–308.

[11] N. Damak, B. Jarboui, P. Siarry, T. Loukil, Differential evolution for solving multi-mode resource-constrained project scheduling problems, Comput. Oper. Res. 36 (9) (2009) 2653–2659.

[12] E. Demeulemeester, M. Vanhoucke, W. Herroelen, RanGen: a random network generator for activity-on-the-node networks, J. Scheduling 6 (2003) 17–38.

[13] B. Doerr, E. Happ, C. Klein, Crossover can provably be useful in evolutionary computation, in: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation, GECCO '08, 2008, pp. 539–546.

[14] S. Elloumi, P. Fortemps, A hybrid rank-based evolutionary algorithm applied to multi-mode resource-constrained project scheduling problem, Eur. J. Oper. Res. 205 (1) (2010) 31–41.

[15] F. Emmert-Streib, M. Dehmer, Exploring statistical and population aspects of network complexity, PLoS One 7 (5) (2012).

[16] M.J. Geiger, Iterated variable neighborhood search for the resource constrained multi-mode multi-project scheduling problem, CoRR (2013). arXiv:1310.0602v1.

[17] H.C. Gomes, F. de Assis das Neves, M.J.F. Souza, Multi-objective metaheuristic algorithms for the resource-constrained project scheduling problem with precedence relations, Comput. Oper. Res. 44 (2014) 92–104.

[18] J. Gonçalves, J. Mendes, M. Resende, A genetic algorithm for the resource constrained multi-project scheduling problem, Eur. J. Oper. Res. 189 (3) (2008) 1171–1190.

[19] G. Gutin, D. Karapetyan, A memetic algorithm for the generalized traveling salesman problem, Nat. Comput. 9 (1) (2009) 47–60.

[20] S. Hartmann, D. Briskorn, A survey of variants and extensions of the resource-constrained project scheduling problem, Eur. J. Oper. Res. 207 (1) (2010) 1–14.

[21] W. Herroelen, R. Leus, Project scheduling under uncertainty: survey and research potentials, Eur. J. Oper. Res. 165 (2) (2005) 289–306.

[22] W. Herroelen, B.D. Reyck, E. Demeulemeester, Resource-constrained project scheduling: a survey of recent developments, Comput. Oper. Res. 25 (4) (1998) 279–302.

[23] J. Kamburowski, D.J. Michael, M.F. Stallmann, Minimizing the complexity of an activity network, Networks 36 (1) (2000) 47–52.

[24] D. Karapetyan, G. Gutin, A new approach to population sizing for memetic algorithms: a case study for the multidimensional assignment problem., Evol. Comput. 19 (3) (2011) 345–371.

[25] A. Kheiri, Multi-stage Hyper-heuristics for Optimisation Problems, University of Nottingham, School of Computer Science, 2014 Ph.D. thesis.

[26] A. Kheiri, E. Özcan, An iterated multi-stage selection hyper-heuristic, Eur. J. Oper. Res. 250 (1) (2016) 77–90.

[27] A. Kheiri, E. Özcan, A.J. Parkes, HySST: hyper-heuristic search strategies and timetabling, in: Proceedings of the 9th International Conference on the Practice and Theory of Automated Timetabling (PATAT 2012), 2012.

[28] R. Kolisch, A. Sprecher, PSPLIB - a project scheduling problem library : OR software - ORSEP operations research software exchange program, Eur. J. Oper. Res. 96 (1) (1997) 205–216.

[29] R. Kolisch, A. Sprecher, A. Drexl, Characterization and generation of a general class of resource-constrained project scheduling problems, Manage. Sci. 41 (10) (1995) 1693–1703.

[30] G. Koulinas, L. Kotsikas, K. Anagnostopoulos, A particle swarm optimization based hyper-heuristic algorithm for the classic resource constrained project scheduling problem, Inf. Sci. 277 (2014) 680–693.

[31] D. Krüger, A. Scholl, A heuristic solution framework for the resource constrained (multi-)project scheduling problem with sequence-dependent transfer times, Eur. J. Oper. Res. 197 (2) (2009) 492–508.

[32] T.S. Kyriakidis, G.M. Kopanos, M.C. Georgiadis, MILP formulations for single- and multi-mode resource-constrained project scheduling problems, Comput. Chem. Eng. 36 (0) (2012) 369–385.

[33] K. Li, R. Willis, An iterative scheduling technique for resource-constrained project scheduling, Eur. J. Oper. Res. 56 (3) (1992) 370–379.

[34] A. Lova, C. Maroto, P. Tormos, A multicriteria heuristic method to improve resource allocation in multiproject scheduling, Eur. J. Oper. Res. 127 (2) (2000) 408–424.

[35] J. Mendes, J. Gon?alves, M. Resende, A random key based genetic algorithm for the resource constrained project scheduling problem, Comput. Oper. Res. 36 (1) (2009) 92–109.

[36] T. Messelis, P.D. Causmaecker, An automatic algorithm selection approach for the multi-mode resource-constrained project scheduling problem, Eur. J. Oper. Res. 233 (3) (2014) 511–528.

[37] M. Mitchell, J.H. Holland, S. Forrest, When will a genetic algorithm outperform hill climbing, in: J.D. Cowan, G. Tesauro, J. Alspector (Eds.), NIPS, Morgan Kaufmann, 1993, pp. 51–58.

[38] R.H. Möhring, A.S. Schulz, F. Stork, M. Uetz, Solving project scheduling problems by minimum cut computations, Manage. Sci. 49 (3) (2003) 330–350.

[39] P. Moscato, On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms, Caltech Concurrent Comput. Program C3P Report 826 (1989) 1989.

[40] P. Moscato, M.G. Norman, A memetic approach for the traveling salesman problem implementation of a computational ecology for combinatorial optimization on message-passing systems, Parallel Comput. Transputer Appl. 1 (1992) 177–186.

[41] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: engineering an efficient sat solver, in: Design Automation Conference, 2001. Proceedings, 2001, pp. 530–535.

[42] Y.-S. Ong, M.-H. Lim, N. Zhu, K.-W. Wong, Classification of adaptive memetic algorithms: a comparative study, Trans. Sys. Man Cyber. Part B 36 (1) (2006) 141–152.

[43] E. Özcan, Memes, self-generation and nurse rostering, in: E.K. Burke, H. Rudova (Eds.), Practice and Theory of Automated Timetabling VI, Lecture Notes in Computer Science, 3867, Springer Berlin Heidelberg, 2007, pp. 85–104.

[44] E. Özcan, B. Bilgin, E.E. Korkmaz, A comprehensive analysis of hyper-heuristics, Intell. Data Anal. 12 (1) (2008) 3–23.

[45] E. Özcan, A. Kheiri, A hyper-heuristic based on random gradient, greedy and dominance, in: E. Gelenbe, R. Lent, G. Sakellari (Eds.), Computer and Information Sciences II, Springer London, 2012, pp. 557–563.

[46] E. Özcan, A.J. Parkes, A. Alkan, The interleaved constructive memetic algorithm and its application to timetabling, Comput. Oper. Res. 39 (10) (2012) 2310–2322.

[47] L. Özdamar, G. Ulusoy, A survey on the resource-constrained project scheduling problem, IIE Trans. 27 (5) (1995) 574–586.

[48] V.V. Peteghem, M. Vanhoucke, A genetic algorithm for the preemptive and non-preemptive multi-mode resource-constrained project scheduling problem, Eur. J. Oper. Res. 201 (2) (2010) 409–418.

[49] V.V. Peteghem, M. Vanhoucke, Using resource scarceness characteristics to solve the multi-mode resource-constrained project scheduling problem, J. Heuristics 17 (6) (2011) 705–728.

[50] V.V. Peteghem, M. Vanhoucke, An experimental investigation of metaheuristics for the multi-mode resource-constrained project scheduling problem on new dataset instances, Eur. J. Oper. Res. 235 (1) (2014) 62–72.

[51] B.D. Reyck, W. Herroelen, On the use of the complexity index as a measure of complexity in activity networks, Eur. J. Oper. Res. 91 (2) (1996) 347–366.

[52] P. Ross, Hyper-heuristics, in: E.K. Burke, G. Kendall (Eds.), Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques, Springer, 2005, pp. 529–556.

[53] K. Sastry, D.E. Goldberg, G. Kendall, Genetic algorithms, in: E.K. Burke, G. Kendall (Eds.), Search Methodologies, Springer US, 2014, pp. 93–118.

[54] M.H. Sebt, Y. Alipouri, Y. Alipouri, Solving resource-constrained project scheduling problem with evolutionary programming, J. Oper. Res. Soc. 64 (2013) 1327–1335.

[55] D. Sudholt, Crossover speeds up building-block assembly, in: Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation, GECCO '12, ACM, New York, NY, USA, 2012, pp. 689–702.

[56] T. Toffolo, H. Santos, M. Carvalho, J. Soares, T. Wauters, G. Vanden Berghe, An integer programming approach to a generalized project scheduling problem, Matheuristics. Hamburg, 11-13 June 2014, 2014.

[57] M. Vanhoucke, An overview of recent research results and future research avenues using simulation studies in project management, ISRN Comput. Math. 2013 (2013).

[58] M. Vanhoucke, E. Demeulemeester, W. Herroelen, An exact procedure for the resource-constrained weighted earliness? tardiness project scheduling problem, Ann. Oper. Res. 102 (2001) 179–196.

[59] L. Wang, C. Fang, An effective shuffled frog-leaping algorithm for multi-mode resource-constrained project scheduling problem, Inf. Sci. 181 (20) (2011) 4804–4822. Special Issue on Interpretable Fuzzy Systems.

[60] L. Wang, C. Fang, An effective estimation of distribution algorithm for the multi-mode resource-constrained project scheduling problem, Comput. Oper. Res. 39 (2) (2012) 449–460.

[61] T. Wauters, J. Kinable, P. Smet, W. Vancroonenburg, G.V. Berghe, J. Verstichel, The multi-mode resource-constrained multi-project scheduling problem, J. Scheduling 19 (3) (2016) 271–283.

[62] T. Wauters, K. Verbeeck, P.D. Causmaecker, G.V. Berghe, A learning-based optimization approach to multi-project scheduling, J. Scheduling 18 (1) (2015) 61–74.

[63] T. Wauters, J. Verstichel, K. Verbeeck, G. Vanden Berghe, A learning metaheuristic for the multi mode resource constrained project scheduling problem, Learning and Intelligent OptimizatioN (LION 2009): Trento, 2009.

[64] J. Weglarz, J. Józefowska, M. Mika, G. Waligóra, Project scheduling with finite or infinite number of activity processing modes - a survey, Eur. J. Oper. Res. 208 (3) (2011) 177–205.