MISTA 2013

Combining Monte-Carlo and Hyper-heuristic methods for the Multi-mode Resource-constrained Multi-project Scheduling Problem

Shahriar Asta · **Daniel Karapetyan*** · **Ahmed Kheiri** · **Ender Özcan** · **Andrew J. Parkes**

December 8, 2013

Abstract Project scheduling is a common real world optimisation problem, and was addressed by a competition organised together with the MISTA 2013 conference. In the competition, multiple projects had to be scheduled whilst taking into account the availability of local and global resources. This paper presents the winning approach which combines Monte-Carlo tree search, memetic algorithms, and hyper-heuristic methods with the ability to exploit the computing power of multicore machines. The proposed algorithm significantly outperformed the other approaches during the competition producing the best solution for 17 out of the 20 test instances and performing the best in around 90% of all the trials.

1 Introduction

Project scheduling has been of 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. There are a variety of project scheduling problems and there are many relevant surveys on this topic in the literature (Brucker et al 1999; Herroelen et al 1998; Herroelen and Leus 2005; Özdamar and Ulusoy 1995; Hartmann and Briskorn 2010; Weglarz et al 2011). A well-known benchmark of project scheduling problems is provided in Kolisch and Sprecher (1997). Project scheduling can be broadly grouped under the following main categories; single-mode, multi-mode, resource-constrained single-mode and resource-constrained multi-mode. Single-mode project scheduling with resource constraints has been proven to be NP-hard (Blazewicz et al 1983), hence the problem dealt with in this study is also a complex problem to solve.

While solving computationally difficult problems, exact methods often fail, particularly when the problem instance size increases. Then alternative approaches, such as, heuristics, metaheuristics or hyper-heuristics are preferred in problem solving. The recent proposed solution methodologies for resource-constrained multi-mode project scheduling problem range from exact approaches, such as, MILP (Kyriakidis et al 2012), to metaheuristics, such as, differential evolution (Damak et al 2009), estimation of distribution algorithms (Wang and Fang 2012) and evolutionary algorithms (Elloumi and Fortemps 2010). Multi-project variant of this problem can still be reduced to a single project.

This paper presents our winning approach submitted to MISTA 2013 challenge[1] on multi-mode resource-constrained multi-project scheduling and the results on the competition instances. The full description of this problem domain can be found on the competition website and elsewhere; however, for completeness we also summarise it in this paper. The broad aim is to schedule a set of different and partially interacting projects, with each project consisting of a set of activities. The activities must respect a set of (hard) precedence constraints and project release times. Also the activities use resources and the appropriate resource limits are also hard constraints. Full details are given in Section 2.

Our overall approach search the space of sequences of activities, with each sequence being evaluated by using it to construct a schedule from which the objective function can then be determined. The search operates in two phases in a "construct and improve" fashion. Firstly, a heuristic constructor is used to create initial sequences. The novelty here is that we generate it using Monte-Carlo Tree Search (MCTS) methods (Browne et al 2012). Secondly, an improvement phase is applied using a large variety of heuristic neighbourhood moves. This phase is carefully controlled by a combination of methods arising from standard meta-heuristics, memetic algorithms, and also an extension of

University of Nottingham, School of Computer Science
Jubilee Campus, Wollaton Road, Nottingham, NG8 1BB, UK
sba@cs.nott.ac.uk, daniel.karapetyan@gmail.com, { axk, exo, ajp }@cs.nott.ac.uk
∗ Contact Author

[1] `http://allserv.kahosl.be/mista2013challenge/`

existing hyper-heuristic components (Kheiri et al 2012; Özcan and Kheiri 2012).

In the following sections, we describe the problem first, including the characteristics of the competition instances (Section 2) and then provide the details of our approach (Sections 3–6). The computational experiments and competition results are presented in Section 7. Finally, Section 8 concludes the paper.

2 Problem Description

The problem consists of a set of *projects* $P = \{1, 2, \dots, q\}$, where each project $p \in P$ is composed of a set of *activities*, denoted as A_p, a partition from all activities $A = \{1, 2, \dots, n\}$. 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 after all the 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 have a fixed capacity per time unit. Non-renewable resources 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.

$\mathscr{R}_p^\rho = \{1, 2, \dots, |\mathscr{R}_p^\rho|\}$ 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 point time unit. $\mathscr{R}_p^\nu = \{1, 2, \dots, |\mathscr{R}_p^\nu|\}$ 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. $\mathscr{G}^\rho = \{1, 2, \dots, |\mathscr{G}^\rho|\}$ 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 $\mathscr{M}_j = \{1, 2, \dots, |\mathscr{M}_j|\}$. Each mode $m \in \mathscr{M}_j$ determines the duration of the activity d_{jm} and the activity resource consumptions: r_{jkm}^ρ for each local renewable resource $k \in \mathscr{R}_p^\rho$, r_{jkm}^ν for each local non-renewable resource $k \in \mathscr{R}_p^\nu$ and g_{jkm}^ρ for each global renewable resources $k \in \mathscr{G}^\rho$.

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)

$$f_\mathrm{d}(D) = \left(\sum_{p \in P} \max_{j \in A_p} \left(T_j + d_{jM_j} \right) \right) - L \,, \tag{1}$$

where L is a lower bound (constant) calculated as

$$L = \sum_{p \in P} \left(\mathrm{CPD}_p + e_p \right) \,, \tag{2}$$

and CPD_p is a given pre-calculated value. Note that, since L is the lower bound, $f(D) \geq 0$ for any feasible solution D.

The tie-breaking secondary objective is to minimise the *total makespan*, which is the finishing time of the last activity:

$$f_\mathrm{m}(D) = \max_{j \in A} \left(T_j + d_{jM_j} \right) \,. \tag{3}$$

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

$$f(D) = f_\mathrm{d}(D) + \gamma f_\mathrm{m}(D) \,, \tag{4}$$

where $0 < \gamma \ll 1$ is a constant selected so that $\gamma f_\mathrm{m}(D) < 1$ 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.

2.1 Test Instances

Three sets of instances have been used during the MISTA 2013 Challenge. These instances are produced by combining several multi-mode resource-constrained project scheduling problem (MRCPSP) instances. The MRCPSP instances are generated by the standard project generator ProGen (Kolisch and Sprecher 1997). Briefly, the construction of the instances requires a construction of network subject to a set of constraints, resource factor which reflects the average portion of resources per activity and resource strength to express the degree of availability of the resources. Based on those factors, it is possible to determine the level of the difficulty of the instances. The format of the MRCMPSP data is based on the PSPLIB[2] MRCPSP data format. More about the generation of the instances can be found on (Kolisch and Sprecher 1997) and the PSPLIB website.

The first set of 10 instances (set-A) was released during the qualification phase and the participants were invited to submit the solvers and solutions to those instances. The organisers compared the solvers under uniform conditions where the imposed time limit was five minutes of multi-threaded execution per instance on the organisers' computer. A set of qualified teams were determined at the end of the qualification phase. Subsequent to the qualification phase, a second set of 10 instances (set-B) were published. Again, the finalists were invited to submit the solvers and solutions to those instances. The organisers ran the solvers on a set of instances from set-B and another set of hidden (unpublished) instances (set-X) to decide on the winner of the challenge. The hidden instances (set-X) were published after the challenge ended.

Table 1 summarises the main characteristics of these instances. In the table, 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. $|\mathscr{M}|$ is the average number of modes for each activity, avg. $|Pred|$ is the average number of the predecessor activities for each activity, avg. $|\mathscr{R}^\rho|$ is the average number of the local renewable resources per project, avg. $|\mathscr{R}^\nu|$ is the average number of the local non-renewable resources per project, $|\mathscr{G}^\rho|$ is the number of global renewable resources, avg. R^ρ is the average renewable resource capacities, avg. C^ν 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 and H is the upper bound on the time horizon. The information provided on the table gives some indication about the size of each instance.

3 Schedule Generator

In designing an algorithm, there are two 'natural' representations to be used in the search for an assignment of activities to times:

Schedule-based: A direct representation using the assignment times of activities.

Sequence-based: This is based on a selecting a total order on all the activities. Given such a sequence then 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 that it will go in the schedule.

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. 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.

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. Permutation π has to obey all the precedence relations, i.e., $\pi(j) > \pi(j')$ for each $j \in A$ and $j' \in Pred(j)$. 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 position. Corresponding procedure is formalised in Algorithms 1 and 2.

Algorithm 1: Schedule construction

```
1 Let $S = (\pi, M)$ be the solution;
2 for $i \leftarrow 1, 2, \ldots, n$ do
3     Let $j \leftarrow \pi(i)$;
4     Schedule $j$ in mode $M_j$ to the earliest available
      position;
5 end
```

Such an implementation has an $O(n(\zeta + T\rho d))$ time complexity, where $\zeta = \max_{j \in A} |Prec(j)|$ is the maximum length of the precedence relation list, T is the make span, $\rho = |\mathscr{G}^\rho| + \max_{p \in P} |\mathscr{R}^\rho_p|$ is the maximum number of local and global renewable resources and $d = \max_{j \in A} d_{jM_j}$ is the maximum activity duration. Considering that, typically, $T = \Theta(n)$, the procedure is quadratic to n, and, hence, it is the performance bottleneck of the heuristic (note that most of the local search moves described below have constant or linear to

[2] PSPLIB benchmark: `http://129.187.106.231/psplib/`

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

Table 1 Instances Characteristics

n time complexity). In particular, schedule generation was usually taking over 98% of the CPU time in our experiments. By introducing several improvements described below we managed to reduce the running times of the schedule generator by order of tens. As a result, the number of iterations of the higher level algorithm within the given time frame dramatically increased.

3.1 Algorithmic Improvements

Observe (see Algorithm 2) that the schedule generation algorithm spends most of the time finding the first available position 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 3.

Another speed-up heuristic is exploiting the nature of the neighbourhood moves. In particular, 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, it is likely that $\pi^1(i) = \pi^2(i) = j$ and $M^1_j = M^2_j$ for $i = 1, 2, \dots, x$, where x is the prefix length. Then, by construction, $T^1_j = T^2_j$ for each $j = \pi^2(1), \pi^2(2), \dots, \pi^2(x)$. Hence, having D^1, we do not need to calculate the values T^2_j for $j = \pi^2(1), \pi^2(2), \dots, \pi^2(x)$. For details, see Algorithm 4.

3.2 Code Optimisation

In addition to the algorithmic improvements, we used several programming tricks to optimise the implementation performance:

– Note that the schedule generator has to maintain the remaining resource availability for each global renewable and each local renewable resource and each time unit, i.e., $|\mathscr{G}^\rho| + \sum_{p \in P} |\mathscr{R}^\rho_p|$ arrays of size T. To avoid dynamic memory reallocation, one has to allocate U memory for each of such arrays, where U is the upper bound of T. Due to the lack of a good upper bound, U is often much larger than T,

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

```
1  Let $j$ be the activity to be scheduled;
2  Let $m$ be the mode associated with $j$; Let $p$ be an index such that $j \in A_p$;
3  Calculate the earliest start time of $j$ as
   $t_0 \leftarrow \max\left\{e_p, \max_{j' \in Prec(j)} (t_j + d_{pjm})\right\}$;
4  for $t \leftarrow t_0, t_0+1, t_0+2, \dots$ do
5      for $t' \leftarrow t, t+1, \dots, t+d_{pjm}-1$ do
6          for $k \in \mathscr{R}^\rho_p$ do
7              Let $a$ be the remaining capacity of $k$ at $t'$;
8              if $r^\rho_{jkm} > a$ then proceed to the next $t$
9          end
10         for $k \in \mathscr{G}^\rho$ do
11             Let $a$ be the remaining capacity of $k$ at $t'$;
12             if $r^\rho_{jkm} > a$ then proceed to the next $t$
13         end
14     end
15 end
16 Allocate activity $j$ at $t$ in mode $m$ and update the remaining capacities;
```

Algorithm 3: Scheduling an activity to the earliest available position with early exploration of resource insufficiency.

```
1  Let $j$ be the activity to be scheduled;
2  Let $m$ be the mode associated with $j$; Let $p$ be an index such that $j \in A_p$;
3  Calculate the earliest start time of $j$ as
   $t_0 \leftarrow \max\left\{e_p, \max_{j' \in Prec(j)} (t_j + d_{pjm})\right\}$;
4  for $t \leftarrow t_0, t_0+1, t_0+2, \dots$ do
5      for $t' \leftarrow t+d_{jm}-1, t+d_{jm}-2, \dots, t$ do
6          for $k \in \mathscr{R}^\rho_p$ do
7              Let $a$ be the remaining capacity of $k$ at $t'$;
8              if $r^\rho_{jkm} > a$ then proceed to $t \leftarrow t'+1$
9          end
10         for $k \in \mathscr{G}^\rho$ do
11             Let $a$ be the remaining capacity of $k$ at $t'$;
12             if $r^\rho_{jkm} > a$ then proceed to $t \leftarrow t'+1$
13         end
14     end
15 end
16 Allocate activity $j$ at $t$ in mode $m$ and update the remaining capacities;
```

and the initialisation phase of schedule generator, though linear, might take significant CPU time. Before each schedule generator run and instead of allocating and initialising $(|\mathscr{G}^\rho| + \sum_{p \in P} |\mathscr{R}^\rho_p|)U$ memory, we reuse the previously allocated memory. Let D' be the previous schedule produced by the generator. To evaluate it, we calculate its exact make span $U' = f_\text{d}(D')$ (see (3)). Now, to make sure that the resource availability arrays are initialised correctly,

Algorithm 4: Schedule construction 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 $i \leftarrow 1, 2, \dots, n$ do
5      Let $j \leftarrow \pi^2(i)$;
6      if $prefix = true$ and $\pi^1(i) = j$ and $M^1_j = M^2_j$ then
7          Allocate activity $j$ to $T^1_j$ in mode $M^2_j$ and update the remaining capacities;
8      else
9          Schedule $j$ in mode $M^2_j$ to the earliest available position;
10         $prefix \leftarrow false$;
11     end
12 end
```

it is enough to reinitialise the first U' elements of each of them.

- We optimised data structures for the Intel x86 architecture, i.e., avoided multi-dimensional arrays, ensured proper alignment of data and tried to reuse memory as much as possible to improve the CPU cache efficiency.
- Our metaheuristic spends most of the time on running local search. 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.
- We noticed that most of the test problems have exactly one global renewable and one local renewable resources. Several more problems have two global renewable and none local renewable resources. We implemented corresponding special cases of the schedule generator, which had a significant impact of the running time of the algorithm. Indeed, in case of one global and one local resources, we were able to notably simplify data structures and exclude one of the nested loops. In case of none local resources, we excluded all the logic related to the local resources.
- Finally, by tuning the compiler parameters, we improved the performance of the most time critical fragments of the code.

4 Construction Phase

This section describes how we construct an initial solution to be supplied to the later improvement phase. The constructor is not only designed to produce feasible solutions, but also with the goal that they have a structure similar to that expected in good solutions.

This naturally raises the question of what is a good approximate structure. Hence, we inspected some good overall schedules; we observed that many such cases had an approximate ordering of the projects. That is, there would be time periods in the schedule when the general focus would be on one or few projects and during the schedule this focus would change between projects. Such a structure naturally arises because the primary objective is the delay-based TPD, **??**, rather than on the overall makespan. Suppose that we look at the two projects that finish latest. It can well be that the penultimate project can move its last activities earlier by adjusting the activities of the last finishing project (except its very last activity) then this will improve the TPD. Unlike the makespan objective, the TPD objective encourages unfairness between the finish times of projects, and can drive some projects to finish as early as possible.

Overall, the structures were only partial orderings of the projects, but with more of a tendency for the latest projects to be critical. Consequently, our constructor attempts to create initial sequences that mimic such project (partial) orderings. Furthermore, we expect that only a partial ordering is needed because we can expect that the subsequent improvement phase can be expected to make make small or medium size adjustments to the overall project ordering structure. However, the improvement phase would have more difficulty, and take more iterations, if the general structure of the project ordering were not good. Consequently, 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' of the overall project time. We required the numbers of projects in each partition to be equal - or with a difference of at most one 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 Browne et al (2012). 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 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 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 partition and then to select the partition between the start and middle partitions.

The first stage considers 100 random choices for the partition of the projects, and then selects between these using 120 samples or the rollout. 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 hillclimbing on the space of mode selections. We use moves that randomly flip one mode at a time, and an objective functions 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.

The first stage ends by making a selection of the best partitioning, using the quality of the 25'th percentile of the final solution qualities (the best quartile) of the results of the rollouts. The 'end' partition of this best partition is then fixed to that of the best partition. The decision to fix the 'end' partition 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) partitions.

This entire process only took of the order of 5 seconds, however, (preliminary) experiments did indicate that it made a significant improvement to the final results after the improvement phase.

5 Neighbourhood Operators

In this section, all the neighbourhood operators are described in detail. These operators are restricted, as needed, to only generate feasible sequences. 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 sequence.

5.1 Swap, Insert and Set Operators

Operators belonging to this category are simple swap, insertion and set operators with various coverage areas ranging from a single activity to multiple projects. These operators can be used within local search procedures and mutation operators. The following is the list of the operators utilised in our algorithm.

- SwapActivities: Swap two activities within the sequence. First a random activity j_1 is chosen. Positions of the last predecessor (pos_p) and the first successor (pos_s) of j_1 are then determined. The last predecessor of j_1 is the closest preceding activity with respect to activity j_1. Likewise, the first successor of j_1 is the closest succeeding activity of j_1. Subsequent to determining pos_p and pos_s, a second activity (j_2), is randomly chosen such that $pos_p < pos_{j_2} < pos_s$. That is to say that the activity j_2 is positioned between the two premises pos_p and pos_s. The two activities j_1 and j_2 are then swapped.
- InsertActivity: Inserts a given activity into a new location in the solution sequence. Similar to what was described for SwapActivities, an activity is randomly selected and, if feasible, inserted into a location between the last predecessor and the first successor of the selected activity.
- SetMode: This move is also based on choosing an activity randomly, the mode of which is set randomly while guaranteeing that the new mode is different than the previous mode of the selected activity. This is of course the case when the activity has more than one mode. Otherwise, the move does not perform any operations, leaving the activity mode intact.
- FILS swapActivities/insertActivity/setMode: These are First Improvement Local Search (FILS) procedures based on the swapping, inserting or mode changing. Considering the FILS swapActivities, the selection process starts with choosing a random activity (j_1) based on a uniform distribution. The positions of the last preceding activity (pos_p) and first succeeding activity (pos_s) of j_1 are determined. A window (W) of length l is then considered and placed randomly on the solution sequence such that $pos_{W_{start}} > pos_p$ and $pos_{W_{end}} < pos_s$. W_{start} and W_{end} are the positions of the start and the end of the window respectively. Starting from the first activity that falls within the window boundaries and for each such activity (j_2) the operator swap is performed. In case the operation improves the objective value, the solution is accepted and returned. Otherwise, the operation is undone and we move on to the subsequent position in the window. The same procedure applies to FILS insertActivity and setMode.
- SwapTwoProjects: Swaps two randomly selected projects in the sequence.
- SwapNeighbourProjects: This move is similar to SwapTwoProjects with the difference that the two projects selected for swapping operation should be neighbouring each other on the solution sequence.

5.2 Project-wise Mutational Operators

The main idea behind the operators belonging to this category is to perturb the position of a number of activities within the sequence in a project-wise manner. The major reason behind designing project-wise operators is that, according to our observations, good quality solutions roughly tend to cluster the activities of each project in a close vicinity of each other. Moreover, it is important to have the projects in the right order as such an order has a tremendous effect on the quality of the solution. Thus, the operators described below are designed to manipulate the sequence solution to achieve a solution in which activities are roughly clustered according to the project they belong to as well as manipulating the order of projects within the sequence.

- MutationOneExtreme: the activities of a randomly selected project are all collected and squeezed into a randomly-selected position in the sequence. This way, all the activities which belong to a certain project are placed in adjacency of each other.
- MutationOne: Shifts all the activities of a randomly selected project by a number of positions in the sequence. The scale of the shift (the number of positions by which the activities are shifted) is chosen to be a random number which varies between the position of the first and last activities of the selected project.
- MoveProjects: Extracts the sequence of the projects in the solution (based on the positions of the last activities in each project), selects several consequent projects, and then moves them to either the beginning or end of the sequence.

5.3 Ruin & Recreate Operators

The Ruin&Recreate (R&R) operators consist of moves in which a list of activities is selected based on a specific distribution, forming the R&R list. The distribution with which the activities in the R&R list are chosen varies according to the type of the move (this is explained later in this section). The selected R&R activities are all guaranteed to be different. The position of the selected activities on the solution sequence are considered to be vacant, ready to be occupied as the move completes it's operation. Subsequent to activity selection, each operator performs a move which can be restricted to moving the selected activities or changing their respective modes or both. That is, three options are incorporated into the move:

– Moving activities: prior to moving the selected activities, they are reshuffled randomly. Also, a matrix of precedence feasibilities of the activities within the list is utilised which is constructed during an earlier pre-processing phase. The precedence relationship of the selected activities in the R&R list constitutes a directed graph where there is an edge from a node to it's successor(s). There are always nodes without incoming edges (nodes for which no predecessor can be found among those activities in the list). Thus, when moving activities, the R&R list is scanned for the first such activity with zero predecessor. This activity is then placed in the sequence, if feasible, and the graph is updated accordingly where new nodes without any preceding activities emerge. This procedure continues recursively until no activities remain.
– Changing modes: for each activity in the R&R list, a new random mode is chosen. In case the list contains one or more activities with infeasible modes (resulting in negative availability of the local non-renewable resource), the entire sequence of chosen modes are rejected and another sequence of modes are chosen randomly. This process continues until a feasible set of modes is found for the activities within the list.
– Both: changing modes followed by moving activities, both as described above.

A number of variants of R&R operators is designed where the major distinguishing feature among the variants is the distribution with which the activities of the R&R list are chosen. Please note that, the higher level heuristic which makes it's choice among the move operators of this category, considers each option of an R&R operator as a separate move. In what follows, the activity selection strategy in each move has been described. Furthermore, the intuition behind employing such strategies are explained.

– MoveUniform: As the name suggests the R&R activities are selected according to a uniform distribution, giving each activity an equal chance to move within the sequence and/or change mode.
– MoveLocal: The activities are selected using a specific and non-uniform distribution centred around a controllable position (p), and with a controllable width (w), within the sequence. Specifically, a random activity (j) is selected according to the following probability.

$$p = \frac{1}{\frac{t_j/T - pos}{width} + 1} \tag{5}$$

where, t_j is the scheduled activity start time and T is the overall time-span.
– MoveBiasedGlobalResource: While selecting the activities, this move favours those activities which are scheduled at a time in which remaining capacity on global resources is higher. Hence, This move aims at avoiding the under-usage of global resources. The consumption of the global resource is a project bottleneck. Therefore, maximising it contributes to the minimisation of the first and second objectives substantially.
We start by selecting a random activity j by employing a roulette wheel selection strategy. In other words, the probability of selecting an activity is proportional to the global resource consumption ratio.

$$\frac{\sum_{k=1}^{|\mathscr{G}^\rho|} g^\rho_{jkm}}{\sum_{k=1}^{|\mathscr{G}^\rho|} G^k_\rho} \tag{6}$$

where $\sum_{k=1}^{|\mathscr{G}^\rho|} g^\rho_{jkm}$ is the available global resources for activity j in mode m. $\sum_{k=1}^{|\mathscr{G}^\rho|} G^k_\rho$ is the sum of global resource capacities.
– MoveEndBiased: Favouring the activities with a position close to the end of projects, this move aims at polishing the project endings. The project endings are particularly important. Consider a project which is neatly scheduled within the sequence such that the majority of its activities are adjacent to each other. Often, such a schedule has a lower TDP value compared to a project whose activities are scattered all over the time horizon. Hopefully, biasing the activity selection process towards the project endings leads to a sequence in which a larger number of activities belonging to the same project are adjacently positioned. MoveEndBiased, employs such an strategy. Selecting activities is based on the roulette wheel approach. The probability of selecting an activity $j \in A_p$ is proportional to the ratio $pos^j_p/|A_p|$. pos^j_p is the position of activity j within the project p while $|A_p|$ is the number of activities in project p. Obviously, the chances of selecting an activity are higher if it is positioned closer to the project ending.

6 Improvement Phase

Most of the time (typically more than 95% of the given 5 minutes) our algorithm spends on improving the initial solutions. We use a multi-thread implementation of a simple memetic algorithm with a powerful local search procedure based on a hyper-heuristic which controls low-level moves.

6.1 Memetic Algorithm

A genetic algorithm is a population based metaheuristic combining principles of natural evolution and genetics for problem solving (Sastry et al 2014). A pool of candidate solutions (individuals) for a given problem is evolved to obtain a high quality solution at the end. Mate/parent selection, recombination, mutation and replacement are the main components of an evolutionary algorithm. The usefulness of recombination is still under debate in the research community (Doerr et al 2008; Mitchell et al 1993). A memetic algorithm (MA) hybridises a genetic algorithm with local search which is commonly applied after mutation on the new individuals Moscato (1989); Moscato and Norman (1992). An MA is often tailored for the problem dealt with Neri and Cotta (2012). Many improvements for MAs have been suggested, for example the population sizing (Karapetyan and Gutin 2011) and interleaved mode of operation (Özcan et al 2012). MAs have been successfully applied to many different problems ranging from generalised traveling salesman (Gutin and Karapetyan 2009) to nurse rostering (Özcan 2007). The improvement phase of our algorithm is controlled by a simple multi-threaded MA which effectively manages the solution pool and utilises all the cores of the CPU. Our MA is based on quantitative adaptation at a local level according to the classification in Ong et al (2006).

Within the MA, we use a powerful local search procedure that takes significant time to converge. To achieve sufficient number of generations, we keep the population size small. In particular, we maintain one solution per CPU core, i.e., 8 solutions since the test machine has Intel i7 CPU with 8 virtual cores. Each local search run takes exactly 5 seconds and is performed on a dedicated core. Since local search takes virtually all computational time, this simple parallelisation provides over 95% CPU utilisation.

Because of the small population size and limited number of generations, we decided to use a simple version of the MA, see Algorithm 5. The population consists of solutions S_i, $i = 1, 2, \dots, 8$. The memetic algorithm uses the following subroutines:

- *Construct*() returns a new random solution generated according to the initial partial project sequence as described in 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) if $f(S_i) > 1.05 f(S_{i'})$ for some $i' \in \{1, 2, \dots, 8\}$ or (2) if 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 $f_\text{d}(S_i) + \mathit{idle}$, 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 simple tournament selection with two individuals.
- *Mutate*(X) returns a new solution produced from X by applying a mutation operator. The mutation operator to be applied is selected randomly and uniformly among the available options.

The algorithm contains five mutation operators (for details see Section 5):

- Apply R&R for both positions and modes using the MoveLocal selection mode. Repeat the procedure 20 times, each time selecting 3 activities near a randomly and uniformly chosen position $p \in [0, 1]$. The selection 'width' w is 0.1.
- SwapNeighbourProjects.
- MoveProjects for one project being moved to the end of the sequence.
- MoveProjects for two projects being moved to the beginning of the sequence.
- MoveProjects for three projects being moved to the beginning of the sequence.

Recall that the objective consists of two competing components. 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, as it was shown in Section 4, 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 disable the second objective ($\gamma \leftarrow 0$, see Section 2) and enable it only after 70% of the given time is elapsed.

6.2 A Dominance based Hyper-heuristic Using an Adaptive Threshold Move Acceptance

There is a growing interest towards self configuring/tuning automated search methodologies. Hyper-heuristics are

Algorithm 5: Improvement Phase.

```
1  $\gamma \leftarrow 0$;
2  for $i \leftarrow 1, 2, \ldots, 8$ do
3      $S_i \leftarrow Construct()$;
4  end
5  while there is time remaining do
6      if $elapsedtime \geq 0.7 giventime$ then
7          $\gamma \leftarrow 0.000001$;
8      end
9      for $i \leftarrow 1, 2, \ldots, 8$ (multi-threaded) do
10         $S_i \leftarrow LocalSearch(S_i)$;
11     end
12     for $i \leftarrow 1, 2, \ldots, 8$ do
13         if $Accept(S_i) = false$ then
14             $X \leftarrow Select(S)$;
15             $S_i \leftarrow Mutate(X)$;
16         end
17     end
18 end
```

such approaches which explore the space of heuristics (i.e., move operators) rather than the solutions in problem solving (Burke et al 2013). There are two common types of hyper-heuristics in the literature Burke et al (2010): *selection* methodologies that choose/mix a heuristic from a set of preset low-level heuristics and attempt to control those heuristics during the search process; and *generation* methodologies that aim to build new heuristics from a set of given components. The main components 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 quality of new and input solution is compared using the move acceptance method 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 Burke et al (2013, 2003); Ross (2005); Özcan et al (2008). In this study, we combine two selection hyper-heuristics under a single framework by employing them successively in a structured and staged manner for performing local search. The approach extends the heuristic selection and move acceptance methods introduced in Özcan and Kheiri (2012) and Kheiri et al (under review), respectively to control seventeen low level heuristics which are designed for the particular problem in hand. The details of the hyper-heuristics and their components are explained in the following subsection.

6.2.1 Heuristic Selection and Move Acceptance Components

The pseudocode of the approach is in Algorithm 6. Lines 6-22 and lines 24-25 illustrate the first and second hyper-heuristics, respectively. The first hyper-heuristic randomly selects a low level heuristic from an active pool of heuristics, denoted as LLH in a score proportionate manner using a roulette wheel strategy (Line 6). If $score_i$ is the score of the i^{th} heuristic, then the probability of a heuristic being selected is $score_i / \sum_{\forall j}(score_j)$. Then the selected heuristic is applied to the solution in hand (Line 7). Initially, each heuristic has a score of 1, making the selection probability of each heuristic equally likely. The first hyper-heuristic always maintains the best solution found so far, denoted as S_{best} (Lines 10-12) and keeps track of the time since last improvement. The move acceptance component of this hyper-heuristic (Lines 8-18) is a threshold acceptance method controlled by a parameter, ϵ, accepting all improving moves. A non-improving solution is accepted only if the quality is better than $(1 + \epsilon)$ of the quality of the best solution obtained (Line 15). Whenever the best solution can no longer be improved for a complete second which is the setting of $timeLimit_2$ in line 19, ϵ gets updated (Line 21) according to the Equation 7.

$$\text{UpdateEpsilon}(x) = \frac{\lceil \log(x) \rceil + rand(1, \lceil \log(x) \rceil)}{x} \quad (7)$$

where $x = f(S_{best})$ which is the objective value of the best solution obtained and $rand(lb, ub)$ returns a random integer in $[lb, ub]$. If $f(S_{best})$ is 0, the algorithm terminates and so this case is not considered in the threshold update.

This novel move acceptance method operates in an unusual way while dealing with non-improving moves. After ϵ gets updated, during the initial iterations of the search process, moves *slightly* worse than the best solution which is achieved right before the update are accepted. At a later stage after the update, if a new best solution is obtained, the method relaxes the bound on the objective value of worsening solutions further and starts accepting the ones with larger changes in the objective value.

The second hyper-heuristic dynamically starts operating (Lines 23-26) whenever there is no improvement in the quality of the solution for three seconds which is the setting of $timeLimit_3$ in line 23. It determines the active pool of heuristics (LLH) from the full set of low level heuristics, denoted as LLH_{all} will be used in the following stage extending the idea of a dominance-based heuristic selection as introduced in Özcan and Kheiri (2012) and adjusts the score of each low level heuristic dynamically. Firstly, ϵ is updated in the same manner as in the first hyper-heuristic and never gets changed during this phase. Then a greedy strategy is employed using all low level heuristics for a certain number of steps which is fixed as the number of low level heuristics

Algorithm 6: *LocalSearch*(S_i)

```
Let $LLH_{all} = \{LLH_1, LLH_2, ..., LLH_{\mathcal{M}}\}$ represent set of
    all low level heuristics with each heuristic being
    associated with a score, initially set to 1;
Let $S_{best}$ represent the best schedule;
$S \leftarrow S_i; S_{best} \leftarrow S_i; LLH \leftarrow LLH_{all}$;
repeat
    $heuristicID \leftarrow$ SelectLowLevelHeuristic($LLH$);
    $S' \leftarrow$ ApplyHeuristic($LLH_{heuristicID}, S$);
    if $f(S') < f(S)$ then
        $S \leftarrow S'$;
        if $f(S') < f(S_{best})$ then
            $S_{best} \leftarrow S'$;
        end
    end
    else
        if $f(S') < (1+\epsilon)f(S_{best})$ then
            $S \leftarrow S'$;
        end
    end
    if noImprovement $(timeLimit_2)$ then
        $S \leftarrow S_{best}$;
        $\epsilon \leftarrow$ UpdateEpsilon ($f(S_{best})$);
    end
    if noImprovement $(timeLimit_3)$ then
        $\epsilon \leftarrow$ UpdateEpsilon ($f(S_{best})$);
        $S, LLH \leftarrow$ DecideLowLevelHeuristics
        ($S_{best}$,$LLH_{all}$,$timeLimit_1$);
    end
until isExceeded$(timeLimit_1)$;
return $S_{best}$;
```

(seventeen). All low-level heuristics are partitioned into three sets as LLH_{small}, LLH_{medium}, LLH_{large} considering the number of activities processed (e.g., number of swaps) by a given heuristic. Step by step, this hyper-heuristic builds a set of solutions associating each with the low level heuristic producing that solution reflecting the trade-off between the objective achieved by each low level heuristic and number of steps involved. At the end of this phase, a pareto front is obtained using the non-dominated solutions from the whole set. The low level heuristics on the pareto front are used to form the active pool of low level heuristics. If more than one low level heuristic generates the same objective value which ends up on the pareto front, they all get to enter into this pool. The number of occurrences of each low level heuristic is assigned as its score to be used in the first hyper-heuristic.

At each step, each low level heuristic is applied to the same input solution for a fixed number of iterations, that is $5n/q$, n/q and 1 iterations for each heuristic in LLH_{small}, LLH_{medium} and LLH_{large}, respectively. 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. If all heuristics cannot generate a new solution, then they are reconsidered all together. Once all heuristics are applied to the input and gets processed for the step, the best solution propagates as input to the next greedy step. If the overall given time limit ($timeLimit_1$) is exceeded, then the second hyper-heuristic terminates before completing through all steps and uses the solution set in hand.

Figure 1 illustrates a run of the second hyper-heuristic with four low level heuristics ($LLH_{all} = \{LLH_1$, LLH_2, LLH_3, $LLH_4\}$). Assuming that the pareto front contains 3 points. The first, second and third points on the front are associated with $\{LLH_1, LLH_2\}$, $\{LLH_1\}$ and $\{LLH_1, LLH_3\}$, respectively. Hence, in the next stage, the first hyper-heuristic ignores the fourth low level heuristic ($LLH = \{LLH_1$, LLH_2, $LLH_3\}$) and scores of LLH_1, LLH_2 and LLH_3 are assigned to 3, 1 and 1, respectively. Hence, selection probability of LLH_1 becomes 60%, while it is 20% for LLH_2 and LLH_3.

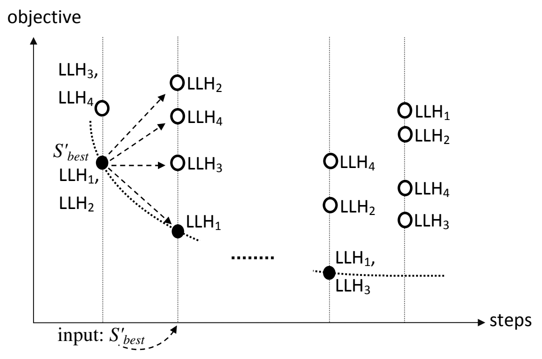

Fig. 1 An illustation of how the second greedy hyper-heuristic operates.

7 Experimental Results

As might be expected, the many of our experiments were concerned with selecting the algorithms and then (partially) tuning the many parameters that are possible within the overall search control algorithm. It was also important to a careful implementation so that the construction of the schedule from a sequence was as fast as possible. For example, it helped to use "prefix sharing" in which the construction from the current sequence can reuse the results of a previous construction, at least up until the point at which the two activity sequences differ.

We provide some preliminary experimental results in Table 2. The entries in *Comp.* columns are taken from the competition website for reference purposes. They demonstrate the best objective values achieved during the qualification (A instances) and final (B and X instances) phases of the competition. Furthermore, the *Avg.* column shows the mean performance of our approach over 2500 runs per instance whereas the *Best* column represents the best objective value that we have obtained for each instance by running our algorithm for 2500 runs per instance. The experiments with 2500 runs per instance are performed on a group of identical 64-bit Intel i7 (3.2 GHz) machines where each machine has 16 GB of RAM operating on Microsoft Windows operating system.

It is worth mentioning that during the final phase where the competing algorithms were applied on B and X instance classes only, our approach found the best solution for 17 out of 20 instances. That is, except for instances B-2, B-4 and X-1, all the *Comp.* entries in Table 2 are achieved by our approach. Furthermore, it is clear from Table 2 (*Best* entries) that our algorithm achieved the best solution for all instances during 2500 run per instance experiments.

	TPD			TMS		
Instance	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

Table 2 Summary of experimental results

To illustrate the performance of our approach better, two boxplots are provided in figures 2 and 3 for B-1 and X-10 instances 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 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.

8 Conclusions

This study describes the components of an effective approach which won the MISTA 2013 challenge with a mean rank of 1.1 for multi-mode resource-constrained multi-project scheduling. A basic performance analysis of the proposed approach and characteristics of the problem instances are also provided. Our algorithm consists of a two-phase construct-and-improve method working on the sequence in which activities are given to a schedule constructor. The construction of an initial activity sequence is done by a (novel) hybrid of MCTS and partitioning of the projects. The improvement phase uses a memetic algorithm in a multi-threaded fashion and large number of neighbourhood moves controlled by hyper-heuristics. The success provides evidence of the utility of carefully designed and controlled hybrids of mixes of (mostly) pre-existing search concepts.

References

Blazewicz J, Lenstra J, Kan A (1983) Scheduling subject to resource constraints: classification and complexity. Discrete Applied Mathematics 5(1):11 – 24

Browne C, Powley E, Whitehouse D, Lucas S, Cowling P, Rohlfshagen P, Tavener S, Perez D, Samothrakis S, Colton S (2012) A survey of Monte Carlo tree search methods. Computational Intelligence and AI in Games, IEEE Transactions on 4(1):1–43, DOI 10.1109/TCIAIG.2012.2186810

Brucker P, Drexl A, Mohring R, Neumann K, Pesch E (1999) Resource-constrained project scheduling: Notation, classification, models, and methods. European Journal of Operational Research 112(1):3–41

Burke EK, Hart E, Kendall G, Newall J, Ross P, Schulenburg S (2003) Hyper-heuristics: An emerging direction in modern search technology. In: Glover F, Kochenberger G (eds) Handbook of Metaheuristics, Kluwer, pp 457–474

Burke EK, Hyde M, Kendall G, Ochoa G, Özcan E, Woodward JR (2010) A classification of hyper-heuristics approaches. In: Gendreau M, Potvin JY (eds) Handbook of Metaheuristics,

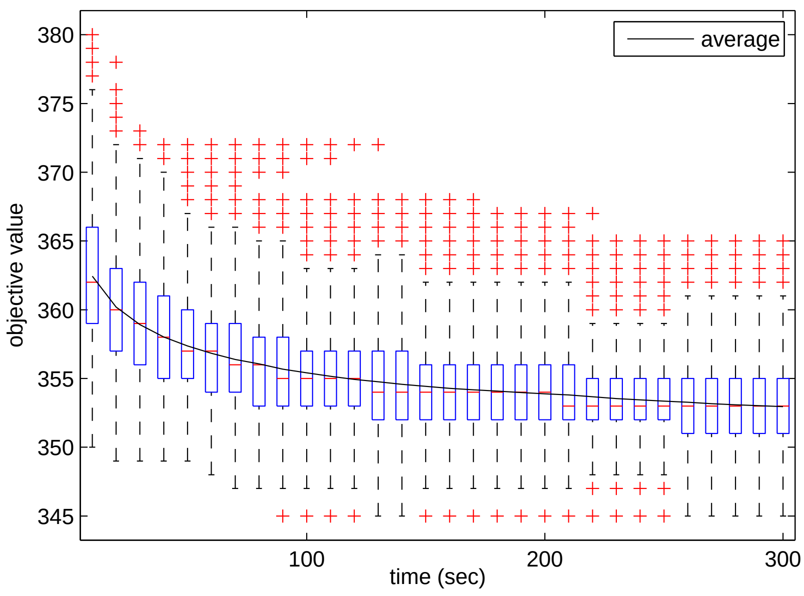

Fig. 2 Performance of the proposed approach on instance B-1.

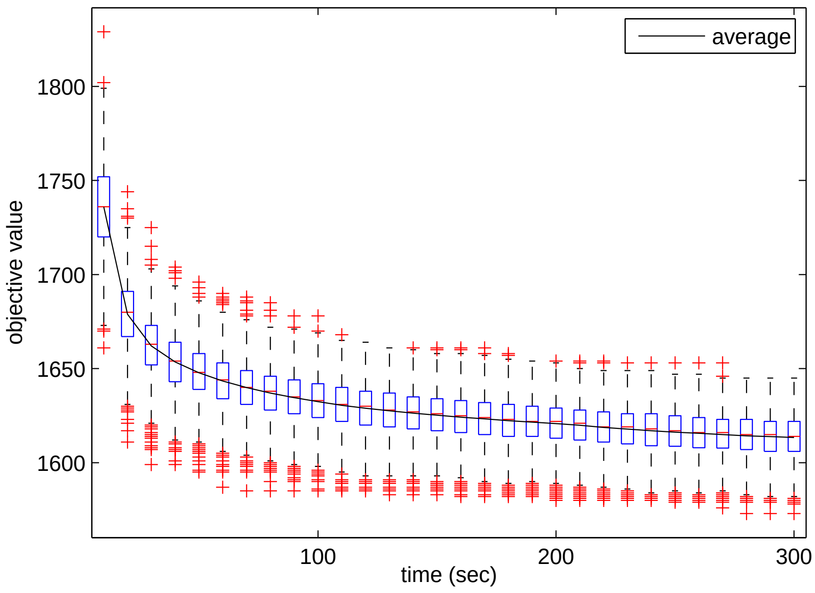

Fig. 3 Performance of the proposed approach on instance X-10.

International Series in Operations Research & Management Science, vol 57, 2nd edn, Springer, chap 15, pp 449–468

Burke EK, Gendreau M, Hyde M, Kendall G, Ochoa G, Özcan E, Qu R (2013) Hyper-heuristics: A survey of the state of the art. Journal of the Operational Research Society DOI 10.1057/jors.2013.71

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

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

Elloumi S, Fortemps P (2010) A hybrid rank-based evolutionary algorithm applied to multi-mode resource-constrained project scheduling problem. European Journal of Operational Research 205(1):31 – 41

Gutin G, Karapetyan D (2009) A memetic algorithm for the generalized traveling salesman problem. Natural Computing 9(1):47–60

Hartmann S, Briskorn D (2010) A survey of variants and extensions of the resource-constrained project scheduling problem. European Journal of Operational Research 207(1):1 – 14

Herroelen W, Leus R (2005) Project scheduling under uncertainty: Survey and research potentials. European Journal of Operational Research 165(2):289 – 306

Herroelen W, Reyck BD, Demeulemeester E (1998) Resource-constrained project scheduling: A survey of recent developments. Computers & Operations Research 25(4):279 – 302

Karapetyan D, Gutin G (2011) A new approach to population sizing for memetic algorithms: a case study for the multidimensional assignment problem. Evolutionary computation 19(3):345–71

Kheiri A, Özcan E, Parkes AJ (2012) HySST: Hyper-heuristic search strategies and timetabling. In: Proceedings of the Ninth International Conference on the Practice and Theory of Automated Timetabling (PATAT 2012)

Kheiri A, Özcan E, Parkes AJ (under review) A stochastic local search algorithm with adaptive acceptance for high-school timetabling. Special Issue of Annals of Operations Research

Kolisch R, Sprecher A (1997) PSPLIB - a project scheduling problem library : OR software - ORSEP operations research software exchange program. European Journal of Operational Research 96(1):205–216

Kyriakidis TS, Kopanos GM, Georgiadis MC (2012) MILP formulations for single- and multi-mode resource-constrained project scheduling problems. Computers & Chemical Engineering 36(0):369 – 385

Mitchell M, Holland JH, Forrest S (1993) When will a genetic algorithm outperform hill climbing. In: Cowan JD, Tesauro G, Alspector J (eds) NIPS, Morgan Kaufmann, pp 51–58

Moscato P (1989) On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Caltech concurrent computation program, C3P Report 826:1989

Moscato P, Norman MG (1992) A memetic approach for the traveling salesman problem implementation of a computational ecology for combinatorial optimization on message-passing systems. Parallel Computing and Transputer Applications 1:177–186

Neri F, Cotta C (2012) Memetic algorithms and memeting computing optimization: A literature review. Swarm and Evolutionary Computation 2:1–14

Ong YS, Lim MH, Zhu N, Wong KW (2006) Classification of adaptive memetic algorithms: A comparative study. Trans Sys Man Cyber Part B 36(1):141–152

Özcan E (2007) Memes, self-generation and nurse rostering. In: Burke EK, Rudova H (eds) Practice and Theory of Automated Timetabling VI, Lecture Notes in Computer Science, vol 3867, Springer Berlin Heidelberg, pp 85–104, DOI 10.1007/978-3-540-77345-0_6

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

Özcan E, Bilgin B, Korkmaz EE (2008) A comprehensive analysis of hyper-heuristics. Intelligent Data Analysis 12(1):3–23

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

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

Ross P (2005) Hyper-heuristics. In: Burke EK, Kendall G (eds) Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques, Springer, chap 17, pp 529–556

Sastry K, Goldberg DE, Kendall G (2014) Genetic algorithms. In: Burke EK, Kendall G (eds) Search Methodologies, Springer US, pp 93–118

Wang L, Fang C (2012) An effective estimation of distribution algorithm for the multi-mode resource-constrained project scheduling problem. Computers & Operations Research 39(2):449 – 460

Weglarz J, Józefowska J, Mika M, Waligóra G (2011) Project scheduling with finite or infinite number of activity processing modes - a survey. European Journal of Operational Research 208(3):177–205