Minimising Cycle Time in Assembly Lines: A Novel Ant Colony Optimisation Approach

Dhananjay Thiruvady, Atabak Elmi, Asef Nazari(✉), and Jean-Guy Schneider

School of IT, Deakin University, Geelong, Australia
{dhananjay.thiruvady,atabak.elmi,asef.nazari, jeanguy.schneider}@deakin.edu.au

Abstract. We investigate the problem of mixed model assembly line balancing with sequence dependent setup times. The problem requires that a set of operations be executed at workstations, in a cyclic fashion, and operations may have precedences between them. The aim is to minimise the maximum cycle time incurred across all workstations. The simple assembly line balancing problem (with precedence constraints) is proven to be NP-hard and is consequently computationally challenging. In addition, we consider setup times and mixed model product types, thereby further complicating the problem. In this study, we propose a novel ant colony optimisation (ACO) based heuristic, which unlike previous approaches for the problem, focuses on learning permutations of operations. These permutations are then mapped to workstations using an efficient assignment heuristic, thereby creating feasible allocations. Moreover, we develop a mixed integer programming formulation, which provides a basis for comparing the quality of solutions found by ACO. Our numerical results demonstrate the efficacy of ACO across a number of problems. We find that ACO often finds optimal solutions for small problems, and high quality solutions for medium-large problem instances where mixed integer programming is unable to find any solutions.

Keywords: Assembly line balancing · Sequence dependent setup times · Minimise cycle times · Ant colony optimisation · Mixed integer programming.

1 Introduction

Assembly line balancing deals with subdividing a manufacturing process into simple operations and to assign those operations to workstations to produce goods in an optimal fashion. Assembly lines generally consist of a set of workstations connected by material handling systems including moving belts or robots [12]. Finding an optimal assignment of operations to workstations considering precedences between operations, operation durations and the availability

M. Gallagher et al. (Eds.): AI 2020, LNAI 12576, pp. 125–137, 2020.
https://doi.org/10.1007/978-3-030-64984-5_10

of workstations is called an assembly line balancing problem (ALBP) [11]. The problem has several categories depending on line layouts, variability of durations, objective functions, and so forth. For example, a type I ALBP (ALBP-I) tries to minimise the number of workstations for given cycle time. In contrast, a type II ALBP (ALBP-II) minimises the cycle time or maximises the throughput for a given number of workstations. Moreover, there are other types, namely E and F, where type F focuses on maximising an effectiveness measure and type E is used when there is no specific objective [3].

Assembly line problems are also categorised based on the number of models processed on a line. A single model problem (SALBP), an early variant, aimed to address a high volume of demand for a particular product. To respond to diversified customer requirements for products and to avoid constructing multiple assembly lines, the multi-model and mixed model variants (MMALBP) variants emerged. Here, several models of a product are considered simultaneously but still only a single assembly line [11]. Production systems differ substantially, thus leading to new problem variants. The various ALBP type problems include general (GALBP), resource constrained (RCALBP), robotic (RALBP), worker assignment (ALWABP), robotic (RMMALBP), etc. Of particular interest in this study are the type II SALBP and MMALBP problems with setup times, which are abbreviated as SALBPS and MMALBPS, respectively [1,3,17].

ALBP, in its simplest form, is known to be computationally intractable or NP-hard [7]. A large body of research is devoted to solve the problem in a time efficient manner. Several exact algorithms have been proposed for the problem, but to deal with its complexity, incomplete approaches including dynamic programming, heuristics and meta-heuristics have been attempted [12,15]. Simaria and Vilarinho [14] maximise the production rate of a line (equivalent to minimising cycle time of a line) in a MMALBP with parallel workstations for a pre-determined number of operators using a genetic algorithm. The studies by Kilincci [9] and Hossain [8] investigate SALBP-II, where the first study proposes bounded exact methods while the second study proposes a progressive modelling approach. Seyed-Alagheband et al. [13] propose a simulated annealing algorithm to solve a general assembly line balancing problem with setups (GALBPS-II) by using the Taguchi method. In this work, the author represented the problem as an object-oriented graph. Zheng et al. [20] proposed a version of ant colony optimisation (ACO) method in solving a type II ALBP which essentially searches for different combinations of tasks for each workstation. Despite several approaches (exact and incomplete) being proposed for variants of the ALBP, there is still substantial room for improvement.

In this study, we propose a novel ACO based heuristic and adapt a mixed integer programming approach for tackling the MMALBPS.[1] Previous studies with the MMALBPS have focused on either modifying solutions based on a neighbourhood structure [19] or the assignment of operations to workstations [1,10]. In contrast to these approaches (including also ACO), our ACO approach splits the solution creation into two components. The motivation for this is that ACO

[1] Note, SALBPS is a special case of MMALBPS.

is very effective at 'sequence learning' but not as effective in dealing with complex constraints. Hence, the first component learns sequences (or equivalently permutations) of operations (which ACO is very effective at) and the second component takes a sequence and maps it into a feasible assignment of operations to workstations. We show that this approach is indeed very effective for MMALBPS, and our results demonstrate that ACO can find high quality (often optimal) solutions in short time-frames. Moreover, ACO scales very efficiently with problem size.

The paper is organized as follows: Sect. 2 defines the MMALBPS problem and provides its mixed integer programming (MIP) formulation. Section 3 discusses the ACO-based heuristic specifically designed for this problem. Section 4 details the experimental setting followed by a presentation of the results. Section 5 concludes the paper and provides some insights into future work.

2 Problem Definition and Mathematical Model

The MMALBPS can be formally described as follows. We are given N operations which must complete M models of a product in a manufacturing system. The operations are allowed to use W workstations.[2] The assignment of operations to workstations (sequencing) must satisfy precedence relations P_{ij} (an indicator variable), where $P_{ij} = 1$ if operation i must precede operation j. An operation $i \in N$ in model $m \in M$ occupies a workstation $s \in W$ that it is allocated to for a duration T_{im} (process time of operation i in model m). Moreover, when two operations i and j are allocated sequentially to the same workstation, they incur a forward setup[3] time F_{ij}^{m} (setup time between operations i and j in model m). In this paper we aim to solve an MMALBPS with parameters (N, M, W, P_{ij}, T_{im}, F_{ij}^{m}), which seeks an optimal assignment of N operations pertaining to M models to workstations so that the maximum cycle time (sum of durations and setup times) of any workstation is minimised. When $M = 1$, we are dealing with the SALBP.

Given the problem definition above, we define the following MIP. Note, we have adapted the MIP from the study of Akpinar [2] where we relax the constraints on cycle time but rather impose constraints on the number of machines. In the following discussion we provide the list of decision variables and detail the objective function and the constraints of the MIP model. In addition to the parameters specified above, we make use of an indicator variable Q_{im}, which is 1 if $T_{im} > 0$.

The model makes use of binary variables y, w and x. y_{is} is 1 if operation i is assigned to workstation s, w_{ijs} is 1 if operation i precedes operation j at machine s and x_{ijms} is 1 if operation j directly follows operation i of model m at workstation s. Additionally, the variable c represents the cycle time.

[2] We use workstations and stations synonymously in the remainder of this paper.

[3] Backward setups are assumed to be done within the initial setups for all workstations, at the beginning of a cycle.

$$\boldsymbol{Minimize}\ \ c = \max_{s\in W, m\in M}\left\{\sum_{i=1}^{N}\left(y_{is}T_{im} + \sum_{j=1}^{N}(x_{ijms}F_{ijm})\right)\right\} \tag{1}$$

$$\boldsymbol{subject\ \ to}\ \sum_{s=1}^{W} y_{is} = 1 \quad \forall i \in N \tag{2}$$

$$\left(\sum_{s=1}^{W} s y_{is} - \sum_{s=1}^{W} s y_{js}\right) P_{ij} \leq 0 \quad \forall i,j \in N,\ \ i \neq j \tag{3}$$

$$w_{ijs} + w_{jis} + x_{ijms} + x_{jims} \leq 2(1 - y_{is} + y_{js}) \quad \forall i,j \in N, m \in M, s \in W \tag{4}$$

$$w_{ijs} + w_{jis} + x_{ijms} + x_{jims} \leq 2(1 + y_{is} - y_{js}) \quad \forall i,j \in N, m \in M, s \in W \tag{5}$$

$$w_{ijs} + w_{jis} + x_{ijms} + x_{jims} \leq 2(y_{is} + y_{js}) \quad \forall i,j \in N, m \in M, s \in W \tag{6}$$

$$P_{ij}(y_{is} + y_{js}) \leq w_{ijs} \quad \forall i,j \in N, i \neq j, s \in W \tag{7}$$

$$w_{iis} = 0 \quad \forall i \in N, s \in W \tag{8}$$

$$w_{iks} + w_{kjs} - 1 \leq w_{ijs} \quad \forall i,j,k \in N, i \neq j \neq k, s \in W \tag{9}$$

$$\left|\sum_{k=1}^{N}\sum_{l=1}^{N} w_{kls} - \sum_{v|v<u}^{N} v\right| \leq N\left|u - \sum_{p=1}^{N} y_{ps}\right| \quad \forall u \in N, s \in W \tag{10}$$

$$\sum_{j=1}^{N}\sum_{s=1}^{W} x_{ijms} \leq 1 \quad \forall i \in N, m \in M \tag{11}$$

$$x_{ijms} + x_{jims} \leq 1 \quad \forall i,j \in N, i \neq j, m \in M, s \in W \tag{12}$$

$$\sum_{s=1}^{W} x_{iims} = 0 \quad \forall i \in N, m \in M \tag{13}$$

$$\begin{aligned}(y_{is}Q_{im} + y_{js}Q_{jm} - 1) - \left|\sum_{k=1}^{N} w_{iks}Q_{km} - \sum_{l=1}^{N} w_{jls}Q_{lm} - 1\right| \leq x_{ijms} \\ \forall i,j \in N, m \in M, s \in W\end{aligned} \tag{14}$$

The objective function minimises the cycle time of the assembly line as given in (1), which is the maximum cycle time across all workstations. Constraints sets (2–3) assign tasks to workstations ensuring that the precedence relations are satisfied. Constraints sets (4–9) order the assigned tasks to each workstation ensuring that the precedence relations within a workstation are satisfied. The constraint set (10) determines the number of orderings between all pairs of tasks at each workstation. Constraints sets (11–13) guarantee that each task can have a single immediate successor in a workstation. Finally, the constraint set (14) determines the immediate successor of each task and relevant setup operation that should be performed between them.

Algorithm 1. ACS for MMALBPS

```
1: Input: An MMALBPS instance, $t_{max}$, $n_s$, $q_0$
2: Initialise($\mathcal{T}$)
3: while time elapsed $< t_{max}$ do
4:     for $j = 1$ to $n_s$ do
5:         $\pi^j$ := ConstructPermutation($\mathcal{T}$, $q_0$)
6:         $\hat{\pi^j}$ := AssignOperations($\pi^j$)
7:     $\pi^{ib} := \min_{j=1,\ldots,n_s} f(\pi^j)$
8:     $\pi^{bs}$ := Update($\pi^{ib}$)
9:     $\mathcal{T}$ := PheromoneUpdate($\pi^{bs}$)
10: return $\pi^{bs}$
```

3 Ant Colony Optimisation

The ACO variant used in this study is ant colony system (ACS) [5]. ACS is proven as one of the most practically effective ACO methods. In particular, its convergence characteristics are a lot more effective on a range of problems compared to that of the original Ant System [5].

We use π – a permutation that represents a solution to the problem. The permutation has an associated assignment $\hat{\pi}$, which takes the operations in the permutation and assigns them one-by-one in the order they appear into workstations. This assignment heuristic (Sect. 3) ensures that precedence feasible solutions are constructed, and hence the permutation does not need to be precedence feasible.

An ACS implementation for the MMALBPS is shown in Algorithm 1. The algorithm has the following inputs: (a) an MMALBPS problem instance, (b) a terminating criteria, we use a time limit (t_{max}) for this study, (c) the number of permutations n_s to be constructed at each iteration of the algorithm and (d) a pre-defined parameter q_0 that specifies the level of deterministic selection. In the first step of the algorithm, the pheromone trails are initialised (Initialise($\mathcal{T}$)). Here, we set $\tau_{ij} = \frac{1}{|N|}$ $\forall i, j \in N$, where τ_{ij} is the desirability of picking $\pi_i = j$.

The algorithm executes Lines 4–9 until a terminating criteria, or in this study, until a time limit expires. Within this time limit, a number of iterations are executed. In an iteration, the first step is to construct n_s solutions from scratch. Each solution is built by starting out with an empty permutation, and adding operations in successive positions of the permutation using the pheromone trails (ConstructSequence($\mathcal{T}$, q_0)). In the usual ACS way, operations are selected either deterministically or stochastically. For this purpose, a random number q is generated from the interval $(0, 1]$ and is compared to the pre-defined parameter q_0. If $q < q_0$, operation k is chosen in position i of the permutation as:

$$k = \underset{j \in N}{\text{argmax}} \{\tau_{ij} \cdot \eta_{ij}\} \tag{15}$$

If $q \geq q_0$, the operation is selected stochastically according to:

$$P(\pi_i = k) = \frac{\tau_{ik} \cdot \eta_{ik}}{\sum_{j \in N} \tau_{ij} \cdot \eta_{ij}} \tag{16}$$

where a heuristic η_{ik} is used to bias the selection of operation i in position k. In preliminary testing, we investigated several heuristics, for example setup times μ, but the most effective choice was found to be $\eta_{ij} = \frac{1}{T_{im}}$.

In ACS, a local pheromone update is used at every selection step as a mean to reduce the chance of making a repeated selection. That is, when an operation j is selected at position i the pheromones are updated according to:

$$\tau_{ij} = \max\{\tau_{ij} \cdot (1.0 - \rho), \tau_{min}\} \tag{17}$$

where, to ensure that no selection of an operation to a position becomes too low, we set $\tau_{min} = 0.0001$. This ensures that operation j at least has a small chance of being selected in position i.

A complete permutation can be mapped in a greedy way to a feasible assignment of operations to workstations in $\mathsf{AssignOperation}(\pi_j)$ (Sect. 3). In Line 7, an iteration best solution is selected as one that minimises the cycle time across all workstations. In Line 8, π^{bs} is updated to π^{ib} if π^{ib} is an improvement. The final step is to update the pheromone trails ($\mathcal{T} := \mathsf{PheromoneUpdate}(\pi^{bs})$) for those solution components seen in π^{bs}:

$$\tau_{ij} = \tau_{ij} \cdot (1.0 - \rho) + \delta \tag{18}$$

where the reward factor $\delta = Q/f(\pi^{bs})$ applies to all solution components and Q is a normalising factor that is chosen to ensure that the reward, relative to the pheromone values, is always between $0.01 \leq \delta \leq 0.1$. For the evaporation rate, it was found that a value of 0.1 is very effective. This is not unusual as for ACO implementations with relatively short run-times (as is the case in this study), a high reward is favourable.

Assigning Operations to Workstations

Previous studies in scheduling have demonstrated the efficacy of learning permutations and mapping these to a schedule [4,18]. Using this idea as a guide, we develop an assignment heuristic, which maps operations in a permutation to workstations.

The high-level algorithm is shown in Algorithm 2. Starting with a permutation π (input), the heuristic assigns operations to workstations ensuring that a minimum cycle time is always maintained. The first step is to initialise the following data structures: (a) $\hat{\pi}$ - the assignment of operations to workstations, (b) W - a waiting list and (c) g - the workstation cycle times. The main loop starts in Line 3 where, for each operation, a feasible assignment to a workstation is found. The operation is tested to see if its preceding operations have been

Algorithm 2. Assigning Operations to Workstations

```
1: INPUT: $\pi$
2: $\hat{\pi} \leftarrow \emptyset\ \forall s \in S$, $W \leftarrow \emptyset$, $g_i \leftarrow 0\ \forall i \in S$
3: for $t \in \pi$ do
4:     $\hat{t} \leftarrow t$
5:     if Prec($t$) not done then
6:         $W \leftarrow W \cup \hat{t}$
7:     else
8:         while $\hat{t} \neq \emptyset$ do
9:             $s :=$ BestWorkStation($\hat{t}$)
10:            $(\hat{\pi}, g_s) \leftarrow$ Update($\hat{t}$)
11:            $\hat{t} \leftarrow \emptyset$
12:            for $j \in$ W do
13:                if Prec($j$) done then
14:                    $\hat{t} \leftarrow j$, $W \leftarrow W \setminus j$
15:                    break
16: OUTPUT: $\hat{\pi}$
```

completed (Lines 5–7), and if not, it is put on a waiting list W. If the precedences are satisfied, then the operation is assigned to the workstation with the minimum overall cycle time divided by the number of jobs on the workstation - that is, workstation j is chosen for operation i under mode m:

$$j = \operatorname*{argmin}_{s \in S} \left\{ \frac{\hat{C}_s + T_{im} + F_{i-1,i,m}}{|N_s|} \right\} \tag{19}$$

where $\hat{C}_s$ is the current cycle time of workstation s and $|N_s|$ is the number of operations already allocated to station s. In Line 10, the final assignment $\hat{\pi}$ and cycle time of workstation s are updated. An assignment of an operation leads to examining W, to determine if any other operations can now be assigned (Lines 12–15). The procedure completes by returning the final precedence-feasible assignment.

Figure 1 illustrates how the heuristic works. The figure on the left shows a permutation of operations and 5 workstations. The operations are assigned in order to workstations while workstations still do not have operations in them (Fig. 1 – middle). In the figure on the right we see that a new operation is added to a workstation when an operation is already assigned to it. In this case, the operation is assigned to the workstation with the lowest overall cycle time.

4 Experimental Setting and Results

ACS for MMALBPS was implemented in C++ and compiled in GCC-5.4.0. The MIP was implemented in Gurobi 9.0.1 (http://www.gurobi.com/). The experiments were conducted on MonARCH - a Linux cluster at Monash University.[4]

[4] https://confluence.apps.monash.edu/display/monarch/MonARCH+Home.

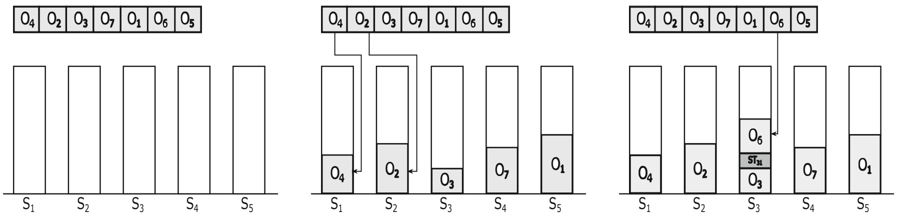

Fig. 1. An example of an assignment of operations to workstations. The figure on the left shows a permutation of 7 operations and 5 workstations. The figure in the middle shows that the first 5 operations are placed in each available workstation (height of a job is its duration). The figure on the right shows that Operation 6 (O_6) is assigned to the workstation with the shortest cycle time (Station 3 here). A setup time ST_{31} is required between the two operations on the workstation.

Table 1. The details of the problem instances; n is the number of operations and **W** the number of workstations; The workstation levels are also provided.

		Instance number																			
		1	2	3	4	5	6	**7**	8	**9**	10	11	12	13	14	15	16	17	18	19	20
n		8	11	12	14	15	16	17	19	21	25	28	29	30	32	35	45	53	58	70	75
W	WL:1	5	6	7	8	8	9	9	10	11	13	15	15	16	17	18	23	27	30	36	38
	WL:2	6	8	9	10	11	11	12	13	15	17	19	20	21	22	24	31	36	39	47	51
	WL:3	7	9	10	11	12	13	13	15	16	19	22	22	23	25	27	34	40	44	53	57

To investigate the performance of the algorithms, we consider the problem instances from [2]. We made two changes: (a) remove the constraint on cycle time and (b) specify the number of workstations as a function of the number of operations n: $\{\frac{n}{2}, \frac{2 \cdot n}{2}, \frac{3 \cdot n}{4}\}$.[5] The details of these problem instances are shown in Table 1. The table shows, for each problem, the number of operations (n), the number of workstations (**W**) split by levels (**WL**). Overall, we have a total of 120 problem instances, with 60 per model type. For ACS, we allow a run-time of 2 min and conduct 30 runs per instance. The MIP executes only once per instance (as it is deterministic) and will allow a time limit of 10 min.[6] ACS is memory efficient requiring only up to 1 GB memory, while the MIP often used up to 10 GB for the large instances.

We used a subset of the problem instances to tune the parameters of ACS. For ρ, we tested values in the range $\{0.2, 0.1, 0.0.5, 0.01\}$ and found that 0.01 was most effective. For q_0 we tested $\{0.5, 0.2, 0.1, 0.01\}$ and found that 0.1 was best. $\tau_{min} = 0.0001$ was used to ensure that an operation at a workstation will always have some small chance of being selected.

[5] In the industry, workstations are limited in the number of operations they can handle.
[6] The MIP is given larger run-time as it struggles to find solutions for large problems.

In the following, we use *workstation level (WL)* to denote the varying level of workstations: $1 = \frac{n}{2}$, $2 = \frac{2 \cdot n}{2}$ and $3 = \frac{3 \cdot n}{4}$. Here, WL = 1 can be thought of as the hardest problem consisting of the fewest machines. The instances are numbered such that the number of workstations increases with increasing numbers.

Table 2 shows a comparison of ACS and the MIP solver. The results are split by model M (single = 1, two = 2), workstation level WL and solver type. The table shows the cycle time obtained by the MIP, and for ACS, the best, mean and standard deviations of 30 runs are reported. The best solutions found are marked in boldface. When a method fails to find a feasible solution (only in the case of the MIP) it is marked by a "-".

For small problem instances, the MIP can often find optimal solutions (up to problem 8 for single model problems). On instances with many workstations, i.e., WL = 2,3, ACS often finds the optimal solution (on average). In fact for a few small problem instances with two models (e.g., Problem 5), ACS outperforms the MIP. However, for these instances the ACS solutions are not provably optimal. When the problem size increases ($\geq$9), ACS easily proves to be the best method, where the MIP is unable to even find a feasible solution on most occasions. Even when solutions are found (e.g.. Problem 10 - Model 1 - WL 2), they have very large cycle times. Interestingly, splitting by models, we see that ACS performs better when there are two models compared to one despite the instances with two models being more complex. These results overall demonstrate that despite running ACS for short run-times, we see very good results.

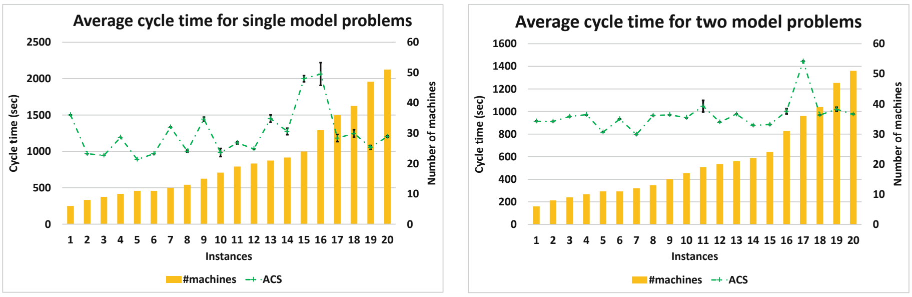

Fig. 2. The performance of ACS on single model and two model instances.

We now delve into the results by focusing on ACS in Fig. 2. The left axis represents the cycle time, the right axis shows the number of workstations and the horizontal axis represents the instances. The bars in these plots show the number of machines. The figure on the left shows the results for single models while the figure on the right shows the same for two models. The ACS run is indicated by the green line. An obvious pattern we see is that the variation in single model problems is much larger than two model problems, demonstrating the usefulness of ACS in dealing with increased complexity.

Table 2. The results the cycle times obtained by the MIP and ACS. Since ACS is run 30 times on each problem instance, the best, average and standard deviations are shown. The results are split by model type (single or two model) - M - and three workstation levels (WL); the values in boldface indicate the best solution found for a problem instance; "-" indicates that no feasible solution was found; b is the best solution, μ is the average solution and sd is the standard deviation considering 30 runs.

M	WL	Solver		Instances																			
				1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	1	MIP		**1223.0**	**1017.0**	**997.0**	**1045.0**	**889.0**	**968.0**	**1134.0**	**1183.0**	–	–	–	–	–	–	–	–	–	–	–	–
		ACS	b	1796.0	1206.0	1341.0	1728.0	1040.0	1470.0	1611.0	1313.0	2589.0	1424.0	1232.0	1950.0	1845.0	3529.0	2583.0	2812.0	4766.0	5638.0	1500.0	1519.0
			μ	1796.0	1208.3	1341.0	1728.0	1088.9	1481.4	1613.0	1409.8	**2640.7**	**1469.4**	**1289.2**	**2600.1**	**2096.7**	**3529.0**	**2583.0**	**2884.2**	**4766**	**6009.7**	**1646.4**	**1598.0**
			sd	0.0	11.17	0.0	0.0	94.5	38.5	5.5	26.4	54.1	137.7	51.5	340.9	107.2	0.0	0.0	196.0	0.0	295.9	92.6	50.8
	2	MIP		**1067.0**	969.0	945.0	**969.0**	889.0	968.0	**1000.0**	962.0	1994.0	5007.0	–	–	–	–	–	–	–	–	–	–
		ACS	b	1501.0	969.0	945.0	1181.0	889.0	968.0	1333.0	962.0	1439.0	883.0	1084.0	1011.0	1235.0	1184.0	1880.0	1770.0	1096.0	1174.0	966.0	1171.0
			μ	1501.0	969.0	945.0	1192.9	889.0	968.0	1333.0	1003.3	**1444.7**	**985.0**	**1116.2**	**1035.0**	**1451.2**	**1274.2**	**1999.0**	**2063.8**	**1183.7**	**1246.7**	**1054.9**	**1204.4**
			sd	0.0	0.0	0.0	58.2	0.0	0.0	0.0	33.5	26.8	70.0	45.6	5.0	63.8	93.9	81.6	309.4	93.5	118.8	65.7	15.2
	3	MIP		**915.0**	969.0	945.0	969.0	889.0	968.0	**828.0**	962.0	2112.0	1694.0	–	–	–	–	–	–	–	–	–	–
		ACS	b	1404.0	969.0	945.0	969.0	889.0	968.0	1333.0	962.0	1195.0	883.0	947.0	783.0	1235.0	955.0	1448.0	946.0	971.0	949.0	956.0	992.0
			μ	1404.0	969.0	945.0	969.0	889.0	968.0	1333.0	962.0	**1197.7**	**883.0**	**947.0**	**803.0**	**1272.6**	**955.0**	**1523.0**	**1065.5**	**971.0**	**949.0**	**956.0**	**1027.8**
			sd	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	9.5	0.0	0.0	67.8	86.2	0.0	78.5	63.9	0.0	0.0	0.0	21.5
2	1	MIP		**1066.0**	**1196.0**	**956.0**	972.0	932.0	933.0	**1008.0**	1232.0	–	–	–	–	–	–	–	–	–	–	–	–
		ACS	b	1134.0	1300.0	1059.0	972.0	899.0	933.0	1187.0	1138.0	970.0	1363.0	1003.0	905.0	1585.0	2191.0	898.0	1837.0	2500.0	1013.0	1464.0	1250.0
			μ	1134.0	1300.0	1059.0	972.0	**899.0**	933.0	1246.2	**1162.0**	**976.8**	**1363.0**	**1171.4**	**905.0**	**1895.2**	**2191.0**	**1109.0**	**2432.1**	**2500.0**	**1061.9**	**1608.1**	**1275.6**
			sd	0.0	0.0	0.0	0.0	0.0	0.0	78.1	44.1	23.6	0.0	146.9	0	279.8	0.0	73.0	272.4	0.0	33.9	77.9	24.1
	2	MIP		996.0	912.0	956.0	972.0	932.0	933.0	809.0	966.0	3186.0	–	–	–	–	–	–	–	–	–	–	–
		ACS	b	913.0	912.0	956.0	972.0	817.0	933.0	797.0	965.0	970.0	946.0	962.0	905.0	975.0	877.0	885.0	967.0	1444.0	969.0	965.0	0975.0
			μ	**913.0**	912.0	956.0	972.0	**817.0**	933.0	**797.0**	**965.0**	**970.0**	**946.0**	**1047.4**	**905.0**	**975.0**	**877.0**	**885.0**	**1002.6**	**1444.6**	**969.0**	**1019.7**	**975.0**
			sd	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	173.9	0.0	0.0	0.0	0.0	54.2	2.2	0.0	51.2	0.0
	3	MIP		731.0	912.0	956.0	972.0	932.0	933.0	809.0	966.0	5116.0	–	–	–	–	–	–	–	–	–	–	–
		ACS	b	731.0	912.0	956.0	972.0	817.0	933.0	797.0	965.0	970.0	946.0	962.0	905.0	975.0	877.0	885.0	967.0	1444.0	969.0	965.0	975.0
			μ	731.0	912.0	956.0	972.0	**817.0**	933.0	**797.0**	965.0	**970.0**	**946.0**	**1047.4**	**905.0**	**975.0**	**877.0**	**885.0**	**967.0**	**1445.0**	**969.0**	**965.0**	**975.0**
			sd	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	173.9	0.0	0.0	0.0	0.0	0.0	2.6	0.0	0.0	0.0

For single model instances, the cycle times tend to increase when the problem consist of 20–30 stations (problem instances 11–16 where the number of tasks are between 28–45). However, for problems with more than 30 stations, the cycle times return to previous levels (900–1500 time units). A similar effect is seen for two model problems at around 35 stations. While this aspect warrants further investigation (e.g.. examining the proportion of operations to workstations or precedence graphs), we leave this to further work due to space limitations.

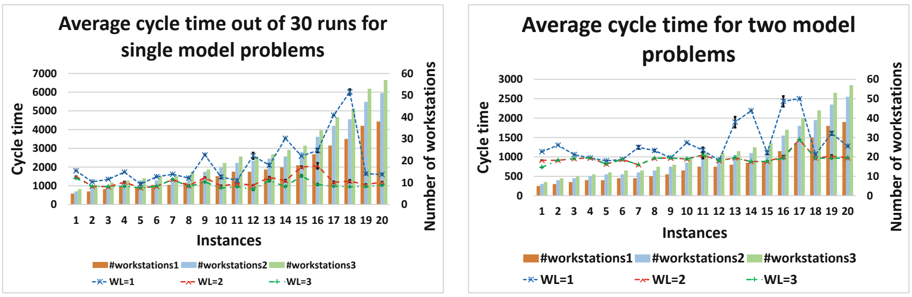

Fig. 3. The performance of ACS with 120 s execution time cap with different workstation levels.

In Fig. 3 we focus on ACS with respect to the level of workstations WL. Like with Fig. 2, the cycle time is presented on the left, the number of workstations on the right and the instances on the horizontal axis. The plots show a split of the performance of ACS depending on the number of workstations. We see that, for single and two models with WL = 1, the cycle times are large and also have high variability. This is not surprising since the instances with a small number of workstations are the hardest ones in the sense of minimising cycle times. In fact, when the instances are "easy", ACS find very low cycle times consistently (a number of which are provably optimal, see Table 2). The large spikes in cycle times seen in Fig. 2 are attributable to those problem instances with relatively few workstations (where workstations were selected as $n \div 2$) for both models.

5 Conclusion and Future Work

We investigate the multi-model assembly line balancing problem with setup times, and propose a novel ant colony system based heuristic for solving it. In comparison to previous ACS methods on similar problems, we focus on learning permutations of operations, which are then mapped to workstations via an efficient assignment heuristic. We compare ACS to a mixed integer programming model and find that ACS performs well overall. In particular, ACS demonstrates clear advantages in three aspects: (a) high quality solutions are found in short time-frames when the number of work stations increase compared to the MIP,

(b) improved performance on two model problems compared to single model problems (c) outperforms the MIP for all medium to large problem instances.

While the proposed ACS approach is effective, there are areas where its performance can certainly be improved. This is especially in the case where there are a small number of workstations leading to large cycle times. For example, the assignment heuristic could be enhanced with probabilistic selection for operations to stations. Furthermore, time-based MIP models could prove very effective, leading to decompositions [6] and hybrid approaches [16].

References

1. Akpinar, S., Baykasoğlu, A.: Modeling and solving mixed-model assembly line balancing problem with setups. part I: a mixed integer linear programming model. J. Manuf. Syst. **33**(1), 177–187 (2014)
2. Akpinar, S., Elmi, A., Bektaş, T.: Combinatorial benders cuts for assembly line balancing problems with setups. Eur. J. Oper. Res. **259**(2), 527–537 (2017)
3. Battaïa, O., Dolgui, A.: A taxonomy of line balancing problems and their solution approaches. Int. J. Prod. Econ. **142**(2), 259–277 (2013)
4. Blum, C., Thiruvady, D., Ernst, A.T., Horn, M., Raidl, G.R.: A biased random key genetic algorithm with Rollout evaluations for the resource constraint job scheduling problem. In: Liu, J., Bailey, J. (eds.) AI 2019. LNCS (LNAI), vol. 11919, pp. 549–560. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-35288-2_44
5. Dorigo, M., Stützle, T.: Ant Colony Optimization. MIT Press, Cambridge (2004)
6. Fisher, M.L.: The Lagrangian relaxation method for solving integer programming problems. Manage. Sci. **50**(12_supplement), 1861–1871 (2004)
7. Gutjahr, A., Nemhauser, G.: An algorithm for the line balancing problem. Manage. Sci. **11**(2), 308–315 (1964)
8. Hossain, S.K.M.: Solving assembly line balancing type II problem using progressive modeling. In Proceedings of the International Annual Conference of the American Society for Engineering Management, pp. 1–10 (2017)
9. Kilincci, O.: A petri net-based heuristic for simple assembly line balancing problem of type 2. Int. J. Adv. Manuf. Technol. **46**(1–4), 329–338 (2010)
10. Kucukkoc, I., Zhang, D.Z.: Mixed-model parallel two-sided assembly line balancing problem: a flexible agent-based ant colony optimization approach. Comput. Ind. Eng. **97**, 58–72 (2016)
11. Scholl, A.: Balancing and sequencing of assembly lines. Physica-Verlag HD, Contributions to Management Science (1999)
12. Scholl, A., Becker, C.: State-of-the-art exact and heuristic solution procedures for simple assembly line balancing. Eur. J. Oper. Res. **168**(3), 666–693 (2006)
13. Seyed-Alagheband, S.A., Ghomi, S.M.T.F., Zandieh, M.: A simulated annealing algorithm for balancing the assembly line type II problem with sequence-dependent setup times between tasks. Int. J. Prod. Res. **49**(3), 805–825 (2011)
14. Simaria, A.S., Vilarinho, P.M.: A genetic algorithm based approach to the mixed-model assembly line balancing problem of type II. Comput. Ind. Eng. **47**(4), 391–407 (2004)
15. Sivasankaran, P., Shahabudeen, P.: Literature review of assembly line balancing problems. Int. J. Adv. Manuf. Technol. 1665–1694 (2014). https://doi.org/10.1007/s00170-014-5944-y

16. Thiruvady, D., Morgan, K., Amir, A., Ernst, A.T.: Large neighbourhood search based on mixed integer programming and ant colony optimisation for car sequencing. Int. J. Prod. Res. **58**(9), 1–16 (2019)
17. Thiruvady, D., Nazari, A., Elmi, A.: An ant colony optimisation based heuristic for mixed-model assembly line balancing with setups. In: 2020 IEEE Congress on Evolutionary Computation (CEC), pp. 1–8 (2020)
18. Thiruvady, D., Wallace, M., Gu, H., Schutt, A.: A Lagrangian relaxation and ACO hybrid for resource constrained project scheduling with discounted cash flows. J. Heuristics **20**(6), 643–676 (2014)
19. Vilarinho, P.M., Simaria, A.S.: ANTBAL: an ant colony optimization algorithm for balancing mixed-model assembly lines with parallel workstations. Int. J. Prod. Res. **44**(2), 291–303 (2006)
20. Zheng, Q., Li, M., Li, Y., Tang, Q.: Station ant colony optimization for the type 2 assembly line balancing problem. Int. J. Adv. Manuf. Technol. **66**(9–12), 1859–1870 (2013)