https://pubsonline.informs.org/journal/ijoc

INFORMS JOURNAL ON COMPUTING

Vol. 00, No. 0, Xxxxx 0000, pp. 000–000

Submitted to *INFORMS Journal on Computing*

ISSN 0899-1499, EISSN 1526-5528

Appendix to Domain-Independent Dynamic Programming and Constraint Programming Approaches for Assembly Line Balancing Problems with Setups

Jiachen Zhang
Department of Mechanical and Industrial Engineering, University of Toronto, jasonzjc@mie.utoronto.ca

J. Christopher Beck
Department of Mechanical and Industrial Engineering, University of Toronto, jcb@mie.utoronto.ca

Abstract. We propose domain-independent dynamic programming (DIDP) and constraint programming (CP) models to exactly solve type-1 and type-2 assembly line balancing problem with sequence-dependent setup times (SUALBP). The goal is to assign tasks to assembly stations and to sequence these tasks within each station, while satisfying precedence relations specified between a subset of task pairs. Each task has a given processing time and a setup time dependent on the previous task on the station to which the task is assigned. The sum of the processing and setup times of tasks assigned to each station constitute the station time and the maximum station time is called the cycle time. For type-1 SUALBP, the objective is to minimize the number of stations, given a maximum cycle time. For type-2 SUALBP, the objective is to minimize the cycle time, given the number of stations. On a set of diverse SUALBP instances, experimental results show that our approaches significantly outperform the state-of-the-art mixed integer programming models for SUALBP-1. For SUALBP-2, the DIDP model outperforms the state-of-the-art exact approach based on logic-based Benders decomposition. By closing 76 open instances for SUALBP-2, our results demonstrate the promise of DIDP for solving complex planning and scheduling problems.

Funding: This work is supported by the Natural Sciences and Engineering Research Council of Canada.

Key words: Domain-independent dynamic programming, Constraint programming, Assembly line balancing

1. Transitions of the DIDP Models

An alternative perspective on the DIDP models arises from the set of transitions that define the recursive structure. We present the transitions for our two SUALBP models here.

1.1. The DIDP Model for SUALBP-1

The transitions of the DIDP model for SUALBP-1 are presented in Table 1. The first type of transition is opening a new station and assigning task i as its first task. The second type of transition

Table 1 Summary of the DIDP model for SUALBP-1.

State	Type	Objects	Preference
U	set	tasks V	
κ	integer resource		less
p	element	tasks V	
f	element	tasks V	
r	integer resource		more
Target state	$U = V, \kappa = 0, p = d_s, f = d_s, r = 0$		
Base case	$U = \emptyset \wedge f = d_s$		
Dual bound	$\max \begin{cases} \left\lceil \frac{\underline{\mu}_f + \sum_{i \in U} (\underline{\tau}_i + t_i) - (\overline{m} - \kappa) \cdot (\max_{i \in U} \underline{\tau}_i) + \max(\underline{m} - \kappa, 0) \cdot (\min_{i \in U} \underline{\mu}_i) - r}{c} \right\rceil \\ \left\lceil \frac{\underline{\mu}_f + \sum_{i \in U} t_i - r}{c} \right\rceil \\ \lceil \frac{\sum_{i \in U} t_i - r}{c} \rceil \\ \sum_{i \in U} w_i^2 + \lceil \sum_{i \in U} w_i'^2 - l^2 \rceil \\ \lceil \sum_{i \in U} w_i^3 - l^3 \rceil \end{cases}$		
Transition	Preconditions	Effects	Cost
`assign_first`$_i$	$i \in V,\ f = d_s,\ U \cap P_i^* = \emptyset$	$U \to U \setminus \{i\},\ r \to c - t_i,\ p \to i,\ f \to i,\ \kappa \to \kappa + 1$	1
`assign_next`$_i$	$i \in U,\ U \cap P_i^* = \emptyset,\ f \neq d_s,\ t_i + \tau_{pi} \leq r$	$U \to U \setminus \{i\},\ p \to i,\ r \to c - t_i - \tau_{pi}$	0
`close_station`	$\{i \notin U$ or $t_i + \tau_{pi} + \mu_{if} > r$ or $U \cap P_i^* \neq \emptyset, \forall i \in V\},\ f \neq d_s,\ \mu_{pf} \leq r$	$r \to 0,\ p \to d_s,\ f \to d_s$	0

is assigning task i as the next task on the current station that already has at least one task assigned. The forward setup time to task i is handled with this transition. The third type of transition is closing the current station. The backward setup time from task p to task f is handled with this transition. The precondition of the final transition ensures that there is no task in U that can be assigned as the last task of the current station. However, it does not mean we cannot assign more tasks to the current station. It is possible that $\exists i, j \in U$ such that $t_j + \tau_{ij} + \mu_{jf} < \mu_{if}$ and hence $t_i + \tau_{pi} + t_j + \tau_{ij} + \mu_{jf} < t_i + \tau_{pi} + \mu_{if}$. As a result, we can assign tasks i and j to the current state without exceeding the remaining time r. In order to incorporate this possibility, we allow transition $\texttt{assign\_next}_i$ to consider any task i as long as $i \in U$, $U \cap P_i^* = \emptyset$, and $t_i + \tau_{pi} \leq r$, where backup setup times are not involved. At the same time, we endow the transition `close_station` with the freedom to close the current station if no task in the unscheduled task set can be selected as the next task for the current station. If no transition $\texttt{assign\_next}_i$ can occur at the current state and $\mu_{pf} > r$, the current state is a dead-end.

1.2. The DIDP Model for SUALBP-2

The transitions in this model are similar to the DIDP model of SUALBP-1. In particular, we also use $\texttt{assign\_first}_i$, $\texttt{assign\_next}_i$, and `close_station`. However, the state variables and costs are updated differently to account for the different cost function. The transitions are summarized in Table 2.

1.3. The DIDP Model for the sequencing subproblem of the local improvement algorithm

In the sequencing subproblem of the local improvement algorithm, tasks assigned to the same station are re-sequenced to minimize the total station time. The DIDP model for this problem is summarized in Table 3. In this model, V_k denotes the set of tasks assigned to station k, d_k represents the dummy task of station k, P_i^k is the set of all predecessors of task i at station k, $\underline{\tau}_i^k$ is the smallest forward setup time from any task at station k to task i, and $\underline{\mu}_f^k$ is the smallest backward setup time from any task at station k to task f. Note that processing times are excluded from the DIDP model since they are constant after task assignment is fixed.

There are three state variables. U is a set variable representing unscheduled tasks and $U = V_k$ in the target state. p is an element variable representing the previous task at station k and $p = d_k$ in the target state. f is an element variable representing the first task at station k and $f = d_k$ in the target state. There are three types of transitions. The first type is assigning task i as the first task at station k. The second type is assigning task i as the next task at station k. This transition also handles the forward setup times. The third type is closing the station and it deals with the backward setup time.

Table 2 Summary of the DIDP model for SUALBP-2.

State	Type	Objects	Preference
U	set	tasks V	
κ	integer		
p	element	tasks V	
f	element	tasks V	
t^c	integer resource		less
c	integer resource		less

Target state	$U = V, \kappa = 0, p = d_s, f = d_s, t^c = 0, c = 0$
Base case	$U = \emptyset \wedge f = d_s$
Dual bound	$\max \left\{ \left\lceil \frac{\sum_{i \in U} (\underline{\tau}_i + t_i) + t^c + (m-\kappa) \cdot (\min_{i \in U} \underline{\mu}_i - \max_{i \in U} \underline{\tau}_i) + \underline{\mu}_f}{\min(m, m-\kappa+1)} - c \right\rceil, \left\lceil \frac{\sum_{i \in U} t_i + t^c}{\min(m, m-\kappa+1)} - c \right\rceil \right\}$

Name	Preconditions	Effects	Cost
`assign_first`$_i$	$i \in U$, $f = d_s$, $U \cap P_i^* = \emptyset$, $\kappa < m$	$U \rightarrow U \backslash \{i\}$, $t^c \rightarrow t_i$, $p \rightarrow i$, $f \rightarrow i$, $\kappa \rightarrow \kappa + 1$, $c \rightarrow \max(c, t_i)$	$\max(0, t_i - c)$
`assign_next`$_i$	$i \in U$, $f \neq d_s$, $U \cap P_i^* = \emptyset$	$U \rightarrow U \backslash \{i\}$, $p \rightarrow i$, $t^c \rightarrow t^c + t_i + \tau_{pi}$, $c \rightarrow \max(c, t^c + t_i + \tau_{pi})$	$\max(0, t^c + t_i + \tau_{pi} - c)$
`close_station`	$f \neq d_s$	$t^c \rightarrow t^c + \mu_{pf}$, $p \rightarrow d_s$, $f \rightarrow d_s$, $c \rightarrow \max(c, t^c + \mu_{pf})$	$\max(0, t^c + \mu_{pf} - c)$

Base case. The base case of the DIDP model is: $U = \emptyset \wedge p = d_s$. Note that $p = d_s$ is necessary since backward setup time has to be included in the base case, with the help of transition `close_station`.

Recursive function. We use $\mathcal{V}(U, p, f)$ to represent the cost of a state. Let $U_1 = \{j \in U | U \cap P_j^k = \emptyset\}$. The recursive function of the DIDP model is as follows:

$$\texttt{compute}\ \mathcal{V}(V, d_k, d_k) \tag{1a}$$

$$\mathcal{V}(U, p, f) = \tag{1b}$$

$$\begin{cases} 0 & \text{if } U=\emptyset, p=d_k, \quad \text{(i)} \\ \min_{j\in U_1} \mathcal{V}(U\backslash\{j\}, j, j) & \text{if } U_1\neq\emptyset, f=d_k, \quad \text{(ii)} \\ \min_{j\in U_1}(\tau_{pj}+\mathcal{V}(U\backslash\{j\}, j, f)) & \text{if } U_1\neq\emptyset, f\neq d_k, \quad \text{(iii)} \\ \min(\mu_{pf}+\mathcal{V}(U, d_k, d_k)) & \text{if } U=\emptyset, f\neq d_k, \quad \text{(iv)} \end{cases}$$

$$U_1=\{j\in U | U\cap P_j^k=\emptyset\},$$

$$\mathcal{V}(U,p,f)\geq \begin{cases} \sum_{i\in U}\underline{\tau}_i^k+\underline{\mu}_f^k, & \text{if } p\neq d_k \\ 0, & \text{if } p=d_k \end{cases} \tag{1c}$$

The term (1a) is to compute the objective of the target state. Equation (1b) is the main recursion of the DIDP model. Specifically, (20b-i) handles the base cases, while (20b-ii) refers to assigning the first task to the current station. (20b-iii) corresponds to assigning the next task after other tasks have been assigned. (20b-iv) deals with closing the current station. Term (1c) formulates the state-based dual bounds. It uses $\underline{\tau}_i^k$ to underestimate the forward setup time to task $i\in U$ and $\underline{\mu}_f^k$ to underestimate the backward setup time to task f.

Table 3 Summary of the DIDP model for the sequencing subproblem.

State	Type	Objects	Preference
U	set	tasks V_k	
p	element	tasks V_k	
f	element	tasks V_k	
Target state	$U=V, p=d_k, f=d_k$		
Base case	$U=\emptyset\wedge p=d_k$		
Dual bound	$\begin{cases}\sum_{i\in U}\underline{\tau}_i^k+\underline{\mu}_f^k, & \text{if } p\neq d_k \\ 0, & \text{if } p=d_k\end{cases}$		

Name	Preconditions	Effects	Cost
`assign_first`$_i$	$i\in U, f=d_k, U\cap P_i^k=\emptyset$	$U\rightarrow U\backslash\{i\}, p\rightarrow i, f\rightarrow i$	0
`assign_next`$_i$	$i\in U, f\neq d_k, U\cap P_i^k=\emptyset$	$U\rightarrow U\backslash\{i\}, p\rightarrow i$	τ_{pi}
`close_station`	$U=\emptyset, f\neq d_k$	$p\rightarrow d_k, f\rightarrow d_k$	μ_{pf}

2. Complete Anytime Beam Search

Algorithm 1: Beam Search for DyPDL

Input: $\langle \mathcal{V}, \mathcal{S}^0, \mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{C}, h \rangle$ - DIDP model in DyPDL,
c - primal bound, $\mathbf{X}$ - initial primal incumbent solution, b - beam width

1 $O \leftarrow \{\mathcal{S}^0\}, c' \leftarrow c, \mathbf{X}' \leftarrow \mathbf{X}$; // Initialization
2 **repeat**
3 $G \leftarrow \emptyset$; // Clear the list of generated states
4 **for** $S \in O$ **do**
5 **for** $S' \in$ `DirectSuccessors`(S) **do**
6 **if** not `PruneState`$(S', \langle \mathcal{V}, \mathcal{S}^0, \mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{C}, h \rangle, c')$ **then**
7 $G \leftarrow G \cup \{S'\}$;
8 **if** $S' \models B$ and $f(S') < c'$ **then**
9 $\mathbf{X}' \leftarrow$ `RetrieveSolution`$(S', \mathcal{T})$; // Update the incumbent solution
10 $c' \leftarrow f(S')$; // Update the primal bound
11 **if** $|G| > b$ **then**
12 $G \leftarrow$ `KeepTopStates`(G, b); // Keep exactly b states according to f values
13 $O \leftarrow G$;
14 **until** Stop condition s is met;
15 **return** $s, c', \mathbf{X}'$;

Complete Anytime Beam Search (CABS) performs beam search for DyPDL models with increasing beam widths. The beam search algorithm is shown in Algorithm 1. Beam search searches the state transition graph layer by layer and maintains states in the same layer (i.e., states that are reached with the same number of transitions) in the open list O, which is initialized with the target state, as shown in line 1. From line 4 to line 13, beam search expands all states in O, stores all the direct successor states in list G, inserts the best b states from G into O, and discards the remaining successor states, where b is a constant parameter called the beam width. This algorithm takes advantage of heuristic search, where a f value is calculated for each state and used as a dual bound of the best solution cost the corresponding state can lead to. The `PruneState` function in line 6 avoids

Algorithm 2: Complete Anytime Beam Search

Input: $\langle \mathcal{V}, \mathcal{S}^0, \mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{C}, h \rangle$ - DIDP model in DyPDL

1 $c \leftarrow \infty, b \leftarrow 1$; // Initialization
2 **repeat**
3 $\quad s, c, \mathbf{X} \leftarrow$ `BeamSearch`$(\langle \mathcal{V}, \mathcal{S}^0, \mathcal{K}, \mathcal{T}, \mathcal{B}, \mathcal{C}, h \rangle, c, b)$; // Beam search for DyPDL
4 $\quad b \leftarrow 2b$; // Double the beam width
5 **until** Stop condition is met;
6 **return** $s, c, \mathbf{X}$;

expanding a state if the f value of it is worse than the primal bound. The `RetrieveSolution` function in line 9 returns a solution according to the transitions leading to state S' if it satisfies the base cases and has a better cost than primal bound. Beam search may discard all successor states leading to solutions, so it is incomplete (i.e., it may not find a solution). Thus, the stop condition can be 'complete', 'incomplete', or 'time-out'.

CABS is shown in Algorithm 2. The cost c and beam width b are intialized in line 1. The main loop of CABS keeps executing the beam search algorithm with the beam width being doubled over CABS iterations. As shown in line 3, CABS takes the DIDP model, the beam width b, and the current best cost c as inputs. With increasing beam widths, the memory consumption increases and the states searched over may overlap in two different iterations. When the beam width is sufficiently large, `BeamSearch` will traverse the entire search space and return 'complete'.