An application of the branch, bound, and remember algorithm to a new simple assembly line balancing dataset

David R. Morrison, Edward C. Sewell, Sheldon H. Jacobson

Abstract

The simple assembly line balancing problem (SALBP) is a well-studied NP-complete problem for which a new problem database of generated instances was published in 2013. This paper describes the application of a branch, bound, and remember (BB&R) algorithm using the cyclic best-first search strategy to this new database to produce provably exact solutions for 86% of the unsolved problems in this database. A new backtracking rule to save memory is employed to allow the BB&R algorithm to solve many of the largest problems in the database.

1 Introduction

The assembly line balancing problem is a well-studied problem with many applications, including the automotive industry, consumer electronics, and household items (Baybars, 1986; Sarker and Pan, 2001). This problem has many variants with different objectives and side constraints; see Battaïa and Dolgui (2013) for a recent survey of problem formulations and solution techniques. One of the most basic assembly line balancing problems is the Simple Assembly Line Balancing Problem (SALBP). In this problem, a set of tasks $T = \{1, 2, ..., n\}$ is given that must be accomplished by a set of workers or stations. In many applications, stations are designed to complete specific tasks; however, the SALBP relaxes this assumption so that all stations are considered identical. Each task requires a certain amount of time t_j (called the **processing time**) to complete, and each station has a specified fixed amount of time c (called the **cycle time**) that it can spend completing tasks.

Additionally, a directed acyclic graph G, called the **precedence graph**, is given with vertex set T and arc set A. An arc $(i, j) \in A$ indicates that task i must be completed before task j. A task i is a **predecessor** (alternately, **successor**) of j if there is a path from i to j (alternately, from j to i) in G; if this path has length 1, i is a **direct** predecessor or successor. The set of direct predecessors (successors) of j is denoted P_j (F_j), and the set of predecessors (successors) of j is P_j^* (F_j^*).

The objective of SALBP is to find the minimum number of stations needed to complete all tasks, subject to the cycle time at each station, that satisfies all relations given in the precedence graph. Given a set of tasks S_m assigned to the m^{th} station, define the **idle time** I_m as the amount of time the station is not working; that is, $I_m = c - \sum_{j \in S_m} t_j$. For a complete assignment of tasks to stations, the total idle time I is the sum of the idle times at each station.

Despite the fact that SALBP is NP-complete (the bin-packing problem is a special case where G has no edges), a number of well-known exact algorithms exist for solving SALBP. An early algorithm developed by Hoffmann (1992) called `Eureka` used an effective heuristic together with a branch-and-bound algorithm that explored in both the forward and reverse directions (i.e., by assigning tasks to either the first stations first or the last stations first). Extensions to the Hoffman heuristic were proposed in Fleszar and Hindi (2003) that were able to perform quite well on a subset of standard benchmark problems.

Johnson (1988) describes an algorithm called FABLE which incorporates a number of bounding rules and dominance rules for the SALBP problem; additionally, Nourie and Venta (1991) give an algorithm called OptPack which uses a dominance rule called the **tree dominance rule**. Another branch-and-bound algorithm, called `Salome` (Scholl and Klein, 1997, 1999), also incorporated a bi-directional search strategy together with several highly effective lower bounds, dominance rules, and a new branching strategy. Another lower bound, based on a LP relaxation of SALBP, is described in Peeters and Degraeve (2006). A dynamic-programming heuristic for SALBP is given in Bautista and Pereira (2009) that incorporates bounding mechanisms into the DP table. Finally, Scholl and Becker (2006) provide a comprehensive survey of SALBP discussing various bounds and solution methods (both exact and heuristic).

More recently, Sewell and Jacobson (2012) give a highly-effective algorithm for SALBP that is able to solve all 269 test instances in a list of benchmark instances, including one instance that had previously been open for over a decade. This algorithm incorporates a three-phase solution procedure together with the cyclic best-first search (CBFS) strategy, as well as a number of good lower bounds and a memory-based dominance rule. A new set of benchmark instances, as well as a instance generator called `SALBPGen`, was subsequently released by Otto et al. (2013); this dataset contains 6825 problem instances, ranging in size from small (20 tasks) to very large (1000 tasks), and incorporates a number of features commonly seen in real-world data.

This paper extends the algorithm of Sewell and Jacobson (2012) with a new backtracking procedure for very large problem instances, and presents computational results on the new dataset of Otto et al. (2013). In this new dataset, 1359 instances are listed as unsolved; the most significant contribution of this paper is to demonstrate that 1172 instances (i.e., all but 187 instances) can each be solved in under 1 hour of computation time, and a further 184 have the best-known solution improved upon. Moreover, all of the previously-solved instances in this database are also solved by this algorithm. Additionally, a proof of compatibility between the dominance rules used in this algorithm is provided. Finally, a brief study of the remaining unsolved instances is also performed to determine what characteristics make them challenging for the Sewell and Jacobson (2012) algorithm.

This paper is organized as follows: Section 2 provides a description of the new SALBP problem database. Section 3 describes the branch, bound, and remember algorithm used to solve these problems, and Section 4 presents the suite of computational results performed against the instances in the database. Finally, Section 5 gives some concluding remarks.

2 Problem Testbed Description

The `SALBPGen` algorithm of Otto et al. (2013) was designed to emulate properties seen in real-world assembly lines, particularly from the automotive industry. In particular, two graph properties were identified that commonly appear in the precedence graph G for these problems; these are **bottleneck tasks** and **chains**. A third property is that of **modular** design, which is an optional additional generation parameter that groups nodes into related clusters or modules, and builds a super-precedence graph on the modules. However, the modular design option is not used in the benchmark dataset.

A bottleneck task j has high in- and out-degree in G; furthermore, it is the only direct successor for at least two tasks in P_j, and it is the only direct predecessor for at least two tasks in F_j. A chain of tasks, on the other hand, is a set of tasks $C \subseteq T$ with $|C| \geq 2$ such that C forms a path in G and $|P_j| = |F_j| = 1$ for each $j \in C$.

Another important property of SALBP instances is the **order strength**; this value, denoted by OS, is computed as $|A(G^+)|/\binom{n}{2}$, where $A(G^+)$ is the arc set of G^+, the **transitive closure**

of G. That is, G^+ is the graph with vertex set T where arc (i, j) indicates that task i is a (not necessarily direct) predecessor of task j. As stated in Scholl and Klein (1999), "Small values of OS indicate that the precedence constraints are not very restrictive such that many sequences of tasks are feasible." There are some indications that middle values of OS are harder than low or high order strength values. The generator `SALBPGen` allows an input parameter to be given specifying the desired order strength of the graph.

A third important parameter that can be controlled by `SALBPGen` is the distribution of task times; for each instance, task times are randomly generated according to some pre-specified probability distribution. The problem database contains instances with task times that have been generated according to three different distributions, described below:

- Short task time distribution - task times are drawn from a normal distribution with the mean centered around small times
- Bimodal task time distribution - task times are drawn from a combination of two normal distributions with means centered around small and large times
- Centralized task time distribution - task times are drawn from a normal distribution with a mean task time of $c/2$

The first two task time distributions emulate properties seen in real-world instances of the assembly line balancing problem; the latter is designed to produce challenging instances.

The problem database used for testing in this paper was generated and described in Otto et al. (2013). The database contains instances with $n = 20, 50, 100$, and 1000 tasks (called small, medium, large, and very large, respectively). There are 525 instances of each problem size, which have been generated with varying order strengths and distribution of task times. A third of the problems (called BN instances) have been generated with bottleneck nodes having minimum degree eight (or minimum degree four in the small instances). A third (called CH instances) have been generated with 40% of the nodes in chains, and a third of the instances (called MIX instances) have no such requirements on the structure of the precedence graph.

For each problem instance in the medium dataset, there are 9 additional permuted instances, which share a common precedence graph and set of task times, but have randomly assigned the task

times to tasks. Thus, there are a total of 6825 instances in the dataset. Of these instances, Otto et al. (2013) report that 4 small instances, 846 medium instances (including permutations), 170 large instances, and 339 very large instances have not yet been solved, for a total of 1359 unsolved instances. No other papers were found in the literature that have improved upon these results thus far.

3 The BB&R Algorithm for SALBP

Algorithm 1: BBR(t, G, c)

```
ComputeDirection(t, G)
UB = ModifiedHoffmannHeuristic(t, G, c)
LB_root = max(LB1, LB2, LB3, BPLB) ⟪ Global lower bound is best lower bound at the root ⟫
UB = RunSearch(UB, LB_root, CBFS)
if τ < τ_lim and previous search was not (provably) optimal:
    UB = RunSearch(UB, LB_root, BrFS)
```

Algorithm 2: RunSearch$(UB, LB_{\text{root}}, \text{mode})$

```
c = 0
root = (∅, T) ⟪ The root node has no assigned tasks to any station ⟫
X = root
while search tree is non-empty, c < n_lim, τ < τ_lim, and LB_root < UB:
    if max(LB1_X, LB2_X, LB3_X, BPLB_X) ≤ UB or X is dominated:  prune X
    else if X dominates another subproblem Y:  delete Y
    else if X is terminal:  UB = min(m, UB)
    else:
        for each valid S_{m+1}, given A_X:
            Y = (A_X ∪ S_{m+1}, U_X \ S_{m+1}, S_1^X, S_2^X, ..., S_m^X, S_{m+1})
            Insert Y into search tree and increment c
            ⟪ If too many subproblems are generated at a node, the search continues in a
              heuristic manner ⟫
            if more than s_lim subproblems have been generated from X:
                stop generating subproblems
    Select a new X according to the mode (CBFS or BrFS)
return UB
```

To solve the instances in this new database, a branch, bound, and remember (BB&R) algorithm called `BBR` was used. Branch, bound, and remember is an extension of branch-and-bound that stores, or remembers, all of the subproblems generated over the course of the search. The remember phase

allows for the use of the memory-based dominance rule in Section 3.2. The `BBR` algorithm is described in detail in Sewell and Jacobson (2012); individual components are briefly described herein, together with some enhancements that allow the algorithm to be used for the very large problem instances.

The `BBR` algorithm, described in Algorithms 1 and 2, operates in a three-phase procedure; the first phase uses a heuristic method called the Modified Hoffman Heuristic (described in Section 3.1) to produce a valid solution as a good upper bound. The next phase of the algorithm uses this upper bound together with a number of pruning and dominance rules (Section 3.2) in a branch-and-bound search for the optimal solution. Given a subproblem X, these lower bounds are denoted $LB1_X, LB2_X, LB3_X$, and $BPLB_X$; in this phase, `BBR` uses the cyclic best-first search (CBFS) exploration strategy to guide the search. If the second phase cannot prove optimality for the problem, the final phase repeats the branch-and-bound search using breadth-first search (BrFS) instead of CBFS. Both branch-and-bound phases are subject to a (global) CPU time limit τ_{lim} and search tree size limit n_{lim}.

Formally, a subproblem in the branch-and-bound tree maintains the following information: $\mathcal{S} = \{A, U, S_1, S_2, ..., S_m\}$, where A is the set of currently-assigned tasks, U is the set of tasks that still need to be assigned to stations (that is, $U = T \setminus A$), and $S_i \subseteq A, i \in \{1, 2, ..., m\}$ is the sets of tasks assigned to station i.

Given a current subproblem $X = (A_X, U_X, S_1^X, S_2^X, ..., S_m^X)$, new subproblems are generated and added to the search tree using the **station-oriented** branching method, which computes a number of possible **full loads** for the next available station, where station S_i is fully loaded if there are no tasks with satisfied precedence constraints that can be added to S_i and satisfy the cycle time constraint at S_i. A depth-first search mechanism is used to generate children at the current subproblem, in the following manner: each task which has all its precedence constraints satisfied at the current subproblem is considered for addition to the next station S_{m+1}. For each such task i, depth-first search is used to enumerate all possible full loads for the next station such that it contains task i. Once a full load has been generated for S_{m+1}, a new subproblem is generated and either pruned or inserted into the search tree. If the number of possible full loads at the current subproblem exceeds a hard limit s_{lim}, no additional children are generated at that subproblem, and the algorithm proceeds heuristically.

3.1 Initialization

To compute a good initial upper bound for SALBP, `BBR` uses the modified Hoffmann heuristic (MHH) described in Sewell and Jacobson (2012). This heuristic is based off the procedure of Hoffmann (1963), which generates good, but not necessarily optimal solutions to the problem. The MHH constructs a solution to the SALBP instance, one station at a time, by generating up to 1000 possible assignments of tasks to the next station. The MHH differs from the Hoffman heuristic in that it chooses a generated assignment to maximize

$$\sum_{j \in U} (t_j + \alpha w_j + \beta |F_j| - \gamma), \tag{1}$$

where α, β, and γ are parameters, and $w_j = t_j + \sum_{k \in F_j^*} t_k$ is the weight of task j and all its successors. Increasing α encourages assignments that satisfy precedence constraints for tasks with large completion times. Additionally, increasing β encourages assignments that satisfy a large number of precedence constraints. Finally, γ acts as a tie-breaker, encouraging assignments with relatively few items. Setting $\alpha = \beta = \gamma = 0$ yields the (regular) Hoffman heuristic.

The initialization phase of the `BBR` algorithm calls the MHH routine with a range of parameter values, chosen as $\alpha, \beta \in \{0, 0.005, 0.01, 0.015, 0.02\}$ and $\gamma \in \{0, 0.01, 0.02, 0.03\}$. After trying all combinations of these parameter values, the MHH routine returns the best solution found as a good initial upper bound.

3.2 Pruning Rules

The `BBR` algorithm uses four different lower bounds and four dominance rules to prune the search space explored by the branch-and-bound tree. The lower bound rules are defined below (for a more detailed explanation of $LB1, LB2$, and $LB3$, see Scholl and Becker (2006) and Scholl and Klein (1997)):

$$LB1 = \left\lceil \textstyle\sum_{j \in T} t_j / c \right\rceil, \quad LB2 = |\{j \in T \mid t > c/2\}| + \left\lceil \frac{|\{j \in T \mid t_j = c/2\}|}{2} \right\rceil, \quad LB3 = \left\lceil \textstyle\sum_{j \in T} w_j \right\rceil,$$

where

$$w_j = \begin{cases} 1 & \text{if } t_j > 2c/3 \\ 2/3 & \text{if } t_j = 2c/3 \\ 1/2 & \text{if } c/3 < t_j < 2c/3 \\ 1/3 & \text{if } t_j = c/3. \end{cases}$$

At each subproblem in the branch-and-bound tree, the three lower bounds $LB1, LB2$, and $LB3$ are computed; if the maximum of these three values is greater than or equal to the value of the incumbent solution, then the subproblem can be pruned. On the other hand, if the three lower bounds above do not allow the subproblem to be pruned, a fourth lower bound, called *BPLB* (or **bin-packing lower bound**) is used to solve SALBP with the precedence constraints relaxed. As the bin-packing problem itself is NP-hard, a separate branch-and-bound solver is used to find good solutions for *BPLB*; if no good solutions can be found within 1 second of computation time, then the *BPLB* solver is terminated so that more progress can be made in the primary search tree.

Additionally, four different dominance rules are used to attempt to prune subproblems; a subproblem X dominates another subproblem Y if for every subproblem of Y, there exists a subproblem of X that is at least as good. In this case, only X needs to be explored, since if Y has an optimal subproblem, so will X. The four dominance rules used in `BBR` are as follows:

- The Maximal Load Rule - If a partial solution contains a station load S_i and an unassigned task j such that $S_i \cup \{j\}$ does not violate the cycle time c or the precedence constraints, then that partial solution can be pruned.

- The Extended Jackson Rule - For a given partial solution, if the set of tasks assigned to the last station contains some task j, and there exists a task i such that $(i, j) \notin A$, $t_i \geq t_j$, and $F_j \subseteq F_i^*$, and task i can replace task j without exceeding the cycle time at the station or the precedence constraints, then a subproblem containing this partial solution can be pruned.

- The No-Successors Rule - If the set of tasks assigned to the last station in a partial solution at some subproblem has no successors, and there exists an unassigned task which has at least one successor, then the current subproblem can be pruned.

- The Memory-based Dominance Rule - For this rule, it is necessary to store every subproblem

that has been identified in the branch-and-bound tree in a hash table so that the rule can be checked efficiently. The rule states that if there exists a previously-identified subproblem in the search tree that has assigned all of the tasks as the current subproblem, and uses the same number or fewer stations, then the current subproblem can be pruned.

Whenever multiple dominance rules are used, it is important to ensure that there is no mutual dominance that could prevent the optimal solution from being found. First note that the memory-based dominance rule is only ever applied to two subproblems that are already in the search tree, and it only deletes subproblems with a strictly worse solutions. Therefore, it is impossible for the memory-based rule to ever prune an optimal solution. Furthermore, no rule above ever yields a non-maximally-loaded station, so the first rule will never conflict.

The only remaining possible conflict is between the extended Jackson rule (EJR) and the no-successors rule (NSR). The following lemma establishes that these two rules can be used in concert.

Lemma 1. *Let X and Y be two subproblems in the search tree for an instance of SALBP. If Y dominates X via the EJR, and Y is dominated by the NSR, then X is also dominated by the NSR.*

Proof. Suppose not. Let $X = (A, U, S_1, S_2, ..., S_{m-1}, S_m)$ and $Y = (A', U', S_1, S_2, ..., S_{m-1}, S'_m)$ (since X and Y are related by the EJR, it must be the case that they each have m assigned stations, and the first $m-1$ are identical). Then, there must exist $j \in S_m$ and $i \in U$ such that $F_j \subseteq F_i^*$ and $S'_m = (S_m - \{j\}) \cup \{i\}$. By the NSR, $F_i^* = \varnothing$, which implies that $F_j = \varnothing$ as well. All other tasks in S'_m are identical to tasks in S_m, which means that S_m has no successors and can also be pruned by the NSR. ■

Consider a set of subproblems in the search space that are all dominated by either the EJR or the NSR, and suppose that all optimal solutions to SALBP are descendants of some subproblem in this set. Then, it must be the case that some subproblem X in the set is pruned by the EJR and not the NSR, and furthermore, the subproblem Y that dominates X must also be in the set. In particular, there must exist a pair X and Y for which X is dominated only by the EJR and Y is dominated only by the NSR (otherwise, all subproblems in the set would be prunable by only a single dominance rule, and both the EJR and NSR have been proven correct independently). However, Lemma 1 implies that no such pair can exist. This proves the following corollary:

Corollary 1. *The EJR and NSR are compatible dominance rules for SALBP*

3.3 Cyclic Best-First Search

The cyclic best-first search (CBFS) strategy determines the search order in the branch-and-bound tree. This search strategy, described in detail in Sewell and Jacobson (2012) and Sewell et al. (2012), can be thought of as a hybrid method between depth-first search (DFS) and best-first search (BFS). This strategy maintains a value m, and the next subproblem chosen for exploration is selected to be the "most promising" subproblem among all subproblems with a partial solution containing exactly m stations. Once this subproblem is explored, m is incremented by 1; if $m \geq UB$, then m is reset to 0. In this way, the search tree is explored cyclically, which allows for complete solutions to be explored early in the search process (as in DFS), but also uses a measure of best to guide the search process (as in BFS).

The most promising subproblem within a level m of the search tree is determined with a heuristic function,

$$v(s) = I/m - 0.02 \cdot |U|,$$

which attempts to weight subproblems with a higher priority if they are more likely to lead to an optimal solution. In particular, the I/m term encourages the exploration of subproblems which have relatively low idle time per station. The remaining term acts as a tie-breaking function that encourages the exploration of subproblems with large numbers of remaining tasks, since these tasks are likely to be smaller and easier to schedule. The value of the parameter can be empirically chosen, and was selected to match the value in Sewell and Jacobson (2012).

3.4 Search Directions

Some instances of SALBP appear to be substantially easier to solve if tasks are assigned in reverse order, that is, to the last station first. A heuristic measure is used by `BBR` to determine which direction to construct partial solutions. This heuristic computes, for each task in T, the following quantities:

$$E_j = \left\lceil \frac{t_j + \sum_{j \in P_j^*} t_j}{c} \right\rceil \quad \text{and} \quad L'_j = \left\lceil \frac{t_j + \sum_{j \in F_j^*} t_j}{c} \right\rceil.$$

Here, E_j is a lower bound on the earliest station to which task j can be assigned, and given an upper bound M on the number of stations needed, $L_j = M - L'_j + 1$ is an upper bound on the latest station to which task j can be assigned. These quantities are used to determine the approximate size of the search tree for the first five levels; if the forward-built search tree appears to be smaller, the instance is solved in the forward direction, and vice versa.

To compute this approximation on the size of the search tree, let $f_m = |\{j \mid E_j \leq m\}|$ and $r_m = |\{j \mid L'_j \leq m\}|$ for some m; that is, f_m is a bound on the number of tasks that could be assigned to the m^{th} station in the forward search tree, and r_m is a bound on the number of tasks that could be assigned to the m^{th} station in the reverse search tree. Then, if $\prod_{i=1}^{5} f_i \leq \prod_{i=1}^{5} r_i$ the instance is explored in the forward direction, and in the reverse direction otherwise. The quantities computed here give a heuristic estimate of the growth of the search tree in the forward and reverse directions; the idea is that if the search tree in the forward direction for the first five levels is smaller than the corresponding tree in the reverse direction, this trend is extrapolated to deeper regions of the tree. If the two bounds are equal, the forward direction is chosen arbitrarily.

3.5 Backtracking

For instances that are particularly large, or for which each station can hold relatively few tasks, it is not practical to store the entire list of assigned and unassigned tasks, as well as the complete list of stations used in a partial solution at some subproblem in the search tree. In these settings, a backtracking method is incorporated that attempts to minimize memory usage within the branch-and-bound tree. This backtracking method was not incorporated in the algorithm described in Sewell and Jacobson (2012).

In this mode of operation, a subproblem in the branch-and-bound tree is represented by $\mathcal{S} = \{p, S_m\}$, where p is a pointer to the subproblem's parent, and S_m is a list of tasks assigned to the current station. Since all subproblems must be stored in the tree for the memory-based dominance rule to be used, the complete partial solution represented by subproblem $\mathcal{S}$ can be reconstructed by following the parent pointers from $\mathcal{S}$ to the root of the search tree. Furthermore, the idle time and the hash value used to store the subproblem in the hash table for the memory-based dominance rule can be computed by tracing the parent pointers, and thus do not need to be stored at each subproblem.

The advantage of this method is that it substantially reduces the amount of memory used at a particular subproblem; this is most beneficial when the number of tasks assigned to any particular station is small compared to the total number of tasks. The principle disadvantage of this method is that it increases the computational time needed to process a subproblem. However, for the medium-sized instances in the database, it was observed that this method only increased the computation time needed to solve instances by about 30%.

4 Computational Results

The `BBR` algorithm for SALBP was implemented in C++, and run on all instances in the database generated by Otto et al. (2013) using a single core of an Intel Core i7-930 2.8GHz quad-core processor, with 12GB of available memory. All running times reported are given in CPU seconds, and do not include the time needed to initialize the memory for the branch-and-bound tree, which is performed at the beginning of the algorithm. Each test was run with a time limit of one hour; the small, medium, and large instances each had a limit on the size of the search tree of 60 000 000 nodes, and the very large instances had an imposed limit of 80 000 000 search tree nodes, since the use of the backtracking code allows for more subproblems to be stored. All problems had a limit of $s_{lim} = 10\,000$ children generated at a subproblem. The backtracking code was used for the very large problem instances; however, the bin-packing lower bound was disabled, due to its relative ineffectiveness and the large computation time for these problems. The results in this section are compared against the best results found by `Salome` (Scholl and Klein, 1997); `Salome` was run with relatively short time limits (20s, 50s, 70s, and 100s for the small, medium, large, and very large instances, respectively). An Excel spreadsheet containing the results from all experiments is available as an online supplement, and Table 1 describes the notation used in the remainder of this section.

Table 2 summarizes the results for the runs against all problem instances. As shown, the `BBR` algorithm is able to solve all of the small- and medium-sized instances in the database, and all but 12 of the large instances. Finally, it is able to solve 350 of the very large instances. Additionally, `Salome` did not solve any problem instances that were unsolved by `BBR`. Moreover, for the large instances, the `BBR` algorithm was able to improve upon the best-known upper bound in ten cases,

Size	The size of the instance: small, medium, large, or very large
`Salome` Solved	The number of instances solved by `Salome` in Otto et al. (2013)
`BBR` Solved	The number of instances for which `BBR` is able to verify optimality
`BBR` Improved	The number of instances for which `BBR` improved `Salome`'s solution, but did not very optimality
`BBR` Matched	The number of instances for which `BBR` matched `Salome`'s solution, but did not verify optimality
`BBR` Worse	The number of instances for which `BBR`'s solution was worse than `Salome`'s
τ	Total running time in CPU seconds of `BBR`, including the time spent computing the MHH and *BPLB*
μ_τ, σ_τ	Average and standard deviation of computation time (in CPU seconds)
$\lvert\tau < \tau_S\rvert$	Number of instances solved in the time limits given to `Salome` in Otto et al. (2013)
Hit n_{lim}	Number of instances for which `BBR` exceeded n_{lim}
Hit t_{lim}	Number of instances for which `BBR` exceeded t_{lim}
τ_{MHH}	Average time spent computing the MHH
MHH Opt	Number of instances proved optimal by the MHH
$\overline{\mu_\tau}$	Average computation time in CPU seconds for instances not solved at the root
τ_{BPLB}	Average time spent computing *BPLB* for instances which are not solved at the root
$\mu_{\tau\%}$	Average percentage of total computation spent solving the *BPLB* for instances not solved at the root

Table 1: Description of the headings and entries used in Table 2-5.

leaving only two unsolved large instances which could not be solved to optimality or improved. Similarly, the `BBR` algorithm was able to improve the best-known upper bound for 149 very large instances, and it matches the best upper bound in five instances. However, for 21 of the very large instances, the solution found by `BBR` was worse than the solution found by `Salome`.

Size	`Salome` Solved	`BBR` Solved	`BBR` Improved	`BBR` Matched	`BBR` Worse
Small	521	525	0	0	0
Medium	4404	5250	0	0	0
Large	355	513	10	2	0
Very Large	186	350	149	5	21

Table 2: Number of solved problem instances overall.

Table 3 presents timing data for the `BBR` algorithm, broken down by problem size. From this table it can be seen that, despite the relatively large amount of computation time afforded to `BBR` compared to `Salome`, in most cases this extra time was unnecessary. In 99% of the problem instances (6594 out of 6638), `BBR` was able to solve the instance in the same time limit as `Salome`. Furthermore, while performing comparisons between different implementations and environments

is difficult at best, it was estimated that the machine performing the BBR experiments is at most 7 times as fast as the machines running Salome. Even under this very conservative estimate, BBR is able to solve 6434 of the instances in the dataset faster than the (adjusted) time limits given to Salome.

Size	μ_τ	σ_τ	$\lvert\tau < \tau_S\rvert$	$\min \tau$	$\max \tau$	Hit n_{lim}	Hit t_{lim}
Small	0.0018	0.0040	525	0	0.02	0	0
Medium	0.21	5.2	5243	0	220	0	0
Large	38	170	478	0	1600	12	0
Very Large	1100	1600	348	3.0	>3600	86	89

Table 3: Average running times (in CPU seconds) for all instances

Additional data were collected on the performance of the MHH and the bin-packing lower bound, shown in Table 4. These data show that about two-thirds of the problem instances could be solved by the MHH; in general the MHH can be computed quite quickly. Even so, 97% (2064 instances out of 2108) that were not proved optimal by the MHH were solved to optimality within the time limits imposed on Salome. For instances which were not solved at the root, about 40% of the total computation time was spent solving the bin-packing lower bound.

Size	τ_{MHH}	MHH Opt.	$\overline{\mu_\tau}$	τ_{BPLB}	μ_τ%
Small	0.0001	428	0.0057	0.0007	11%
Medium	0.15	3447	0.58	0.41	53%
Large	0.09	305	91	17	61%
Very Large	7.6	322	2800	-	-

Table 4: Details of the MHH and the *BPLB* from the computational experiments.

For the 26 very large instances that BBR was unable to improve or match the best-known solution, it was hypothesized that the heuristic choosing the exploration direction was picking the worse direction, so these 26 instances were rerun in the opposite direction. For the 5 instances in which BBR matched the best-known solution, four were improved by running in the opposite direction. Additionally, for the 21 instances where BBR was unable to match the best-known solution, running in the opposite direction allowed the algorithm to improve the best-known solution for 19 of them, leaving only two very large instances for which BBR was unable to match or improve the best-known solution reported by Otto et al. (2013). These results imply that in some cases, the direction-finding

heuristic is not choosing the most effective exploration direction (the results from running in the opposite direction are not included in Table 2-5).

To see if any further additional improvements could be gleaned from the results, the bin-packing lower bound was computed at the root node for each of the very large instances. The bin-packing lower bound was greater than $LB1, LB2$, and $LB3$ in 156 instances. Furthermore, the bin-packing lower bound was better than the lower bound reported by `Salome` in 33 instances. However, the root bin-packing lower bound did not allow any additional problem instances to be solved.

A further analysis of the instances unsolved by `BBR` was performed, and the results are presented in Table 5. These results show that instances with lower order strength are often more challenging for `BBR` (45% have order strength of approximately 0.2, and 87% have order strength of less than 0.6); the graph structure has a less-clear relationship to instance difficulty. However, the most telling indicator of problem difficulty is the task time distribution: all 187 unsolved instances have a central distribution of task times.

	Structure			Order Strength			Peak location		
Size	BN	CH	MIX	0.2	0.6	0.9	bottom	central	bimodal
Large	7	4	1	10	2	0	0	12	0
Very large	50	50	75	75	75	25	0	175	0

Table 5: Problem statistics for the unsolved instances by `BBR`.

Additionally, an analysis of order strength and task distribution times with respect to the performance of `BBR` was performed on the large and very large problem instances. It was first observed that the MHH was able to prove optimality for only three of the large or very large instances with the central task time distribution. Moreover, the average computation time for the `BBR` algorithm was largest for the large and very large problem instances with the central task time distribution. These observations support the hypothesis that the central distribution of task times creates challenging instances of SALBP.

Furthermore, it was observed that as the order strength increased, the MHH was less able to prove optimality for instances in the database: the MHH was able to prove optimality for two-thirds of the large and very large instances with $OS = 0.2$, 63% of the instances with $OS = 0.6$, but only 29% of the instances with $OS = 0.9$. The relationship between the average computation time of `BBR` and the OS was less clear; however, one interesting relationship that was observed is

that for the unsolved instances, those with low OS hit the node limit more frequently (77 of the 85 unsolved large and very large instances with $OS = 0.2$), and instances with higher OS hit the time limit more frequently (82 of the 102 large and very large instances with $OS \geq 0.6$). This can be explained by noting that instances with low order strength have more viable station loads, and thus more subproblems in the search tree must be explored before the tree is exhausted.

5 Conclusion

This paper describes computational results obtained from applying the branch, bound, and remember algorithm with cyclic best-first search of Sewell and Jacobson (2012) to the new database of simple assembly line balancing problems presented by Otto et al. (2013). The algorithm `BBR` is able to solve all unsolved small and medium instances, and over 90% of the large instances. Furthermore, using a backtracking procedure designed to reduce memory usage, the `BBR` algorithm is able to solve 164 of the 339 unsolved very large instances, and improve the best-known solution for an additional 172 of these instances.

Future research on this problem should focus on improving the various pruning rules and dominance relations to enable greater exploration of the search space; additionally, it would be beneficial to develop methods for optimizing memory usage for the very large instances to enable a larger state space to be explored for these instances. Furthermore, it is apparent that choosing the search direction properly may have an impact on the solution quality of the algorithm; thus, future research will attempt incorporate the bi-directional search rule from `Salome` into `BBR`. Moreover, it may be possible to develop more specialized algorithms that are able to perform more efficiently for instances with specialized structure (for example, instances with a large number of bottleneck nodes, or a centralized distribution of task times). Finally, more study can be done on the impact of various structural parameters on the performance of `BBR` and other similar algorithms. For example, the exact relationship between OS and instance difficulty is currently unknown; thus, more experiments can be done to determine this relationship. Along these lines, it would also be helpful to build a permuted data set of larger problem instances to allow for a more detailed study of problem characteristics.

Acknowledgments

The computational results reported were obtained using the Simulation and Optimization Laboratory. This research has been supported in part by the Air Force Office of Scientific Research (FA9550-10-1-0387), the Department of Defense (DoD) through the National Defense Science & Engineering Graduate Fellowship (NDSEG) Program, and the National Science Foundation through the Graduate Research Fellowship Program. The third author is supported in part by (while serving at) the National Science Foundation. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government. The authors would like to thank three anonymous referees and the associate editor for comments which significantly improved this paper.

References

O. Battaïa and A. Dolgui. A taxonomy of line balancing problems and their solution approaches. *Internat. J. of Production Econom.*, 142(2):259–277, 2013.

J. Bautista and J. Pereira. A dynamic programming based heuristic for the assembly line balancing problem. *European Journal of Operational Research*, 194(3):787–794, 2009.

I. Baybars. A survey of exact algorithms for the simple assembly line balancing problem. *Management science*, 32(8):909–932, 1986.

K. Fleszar and K. Hindi. An enumerative heuristic and reduction methods for the assembly line balancing problem. *European Journal of Operational Research*, 145(3):606–620, 2003.

T. Hoffmann. Assembly line balancing with a precedence matrix. *Management Science*, 9(4): 551–562, 1963.

T. Hoffmann. EUREKA: a hybrid system for assembly line balancing. *Management Science*, 38 (1):39–47, 1992.

R. Johnson. Optimally balancing large assembly lines with "FABLE". *Management Science*, 34 (2):240–253, 1988.

F. Nourie and E. Venta. Finding optimal line balances with OptPack. *Operations Research Letters*, 10(3):165–171, 1991.

A. Otto, C. Otto, and A. Scholl. Systematic data generation and test design for solution algorithms on the example of SALBPGen for assembly line balancing. *European Journal of Operational Research*, 228(1):33–45, 2013.

M. Peeters and Z. Degraeve. An linear programming based lower bound for the simple assembly line balancing problem. *European Journal of Operational Research*, 168(3):716–731, 2006.

B. Sarker and H. Pan. Designing a mixed-model, open-station assembly line using mixed-integer programming. *J. Oper. Res. Soc.*, 52(5):545–558, 2001.

A. Scholl and C. Becker. State-of-the-art exact and heuristic solution procedures for simple assembly line balancing. *European Journal of Operational Research*, 168(3):666–693, 2006.

A. Scholl and R. Klein. SALOME: a bidirectional branch-and-bound procedure for assembly line balancing. *INFORMS Journal on Computing*, 9(4):319–334, 1997.

A. Scholl and R. Klein. Balancing assembly lines effectively a computational comparison. *European Journal of Operational Research*, 114(1):50–58, 1999.

E. Sewell and S. Jacobson. A branch, bound, and remember algorithm for the simple assembly line balancing problem. *INFORMS Journal on Computing*, 24(3):433–442, 2012.

E. Sewell, J. Sauppe, D. Morrison, S. Jacobson, and G. Kao. A BB&R algorithm for minimizing total tardiness on a single machine with sequence dependent setup times. *Journal of Global Optimization*, 54(4):791–812, 2012.