FEDERAL UNIVERSITY OF RIO GRANDE DO SUL
INFORMATICS INSTITUTE
BACHELOR OF COMPUTER SCIENCE

CÉSAR GARCIA DAUDT

Applying Dynamic Programming to Assembly Line Balancing and Sequencing Problems

Graduation Thesis

Prof. Dr. Marcus Ritt
Advisor

Porto Alegre, July 2013

CIP – CATALOGING-IN-PUBLICATION

Daudt, César Garcia

Applying Dynamic Programming to Assembly Line Balancing and Sequencing Problems / César Garcia Daudt. – Porto Alegre: Graduação em Ciência da Computação da UFRGS, 2013.

38 f.: il.

Graduation Thesis – Federal University of Rio Grande do Sul. BACHELOR OF COMPUTER SCIENCE, Porto Alegre, BR–RS, 2013. Advisor: Marcus Ritt.

1. Assembly line problems. 2. Sequencing problems. 3. SALBP-1. 4. BPP-P. 5. Combinatorial optimization. 6. Operations research. 7. Dynamic programming.

"... as botas apertadas são uma das maiores venturas da terra, porque, fazendo doer os pés, dão azo ao prazer de as descalçar."

— MACHADO DE ASSIS

ACKNOWLEDGEMENTS

Agradeço minha família por todo o suporte durante a vida acadêmica e, principalmente, durante a realização deste trabalho. Nisso, sintam-se especialmente lembrados meu pai, mãe, avó e namorada. Que o exemplo de Estudante dado por minha mãe e o de Homem por meu pai sejam cada vez mais bem assimilados e postos em prática por mim. Conviver com vocês é algo extraordinário e ser como vocês é no mínimo desafiador.

Também agradeço ao ensino de qualidade recebido no Instituto de Informática da Universidade Federal do Rio Grande Sul. Os professores bons e motivadores serão sempre lembrados por mim. Em especial, agradeço meu orientador pela ajuda e verdadeira orientação na concepção deste trabalho.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

ALBP	Assembly Line Balancing Problem
SALBP	Simple Assembly Line Balancing Problem
SALBP-1	Simple Assembly Line Balancing Problem of type 1
BPP-P	Bin-Packing Problem with Precedence Constraints
BPP	Bin-Packing Problem

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This work presents two dynamic programing algorithms to treat simple assembly line balancing problem (SALBP) and bin-packing problem with precedence constraints (BPP-P). While the former has been explored for many years, the latter has been studied only recently. For BPP-P, our approach is the first to use dynamic programming and we provide one new optimal answer that was unknown until our algorithm was proposed (from the instances used in the literature, 2 still remain unsolved). For both variants, our implementations are able to deal with the small instances commonly used in the literature. In average, we treat these instances with execution times from miliseconds to few minutes.

We also present, for each algorithm explained, one way to reduce the search space: an implementation of Jackson Dominance Rule and our approximation of Jackson Maximally Loaded station principle. The impact of these optimizations is discussed, measured and compared to the state of the art algorithms.

Remarks are made about important works (from the past and current state of the art algorithms) and surveys in order to make the interested reader able to find further information regarding assembly line balancing problems (specially SALBP and BPP-P variants).

Keywords: Assembly line problems, sequencing problems, SALBP-1, BPP-P, combinatorial optimization, operations research, dynamic programming.

RESUMO

Este trabalho apresenta dois algoritmos de Programação Dinâmica que tratam os problemas Simple Assembly Line Balancing Problem (SALBP) e Bin-Packing Problem with Precedence Constraints (BPP-P). Enquanto o primeiro problema já foi longamente explorado, o segundo só foi estudado anteriormente em um único artigo. Para o BPP-P, nossa abordagem é a primeira a utilizar Programação Dinâmica e nós fornecemos uma nova solução ótima que, até a publicação de nosso algoritmo, era desconhecida (duas instâncias do conjunto de testes consagrado pela literatura ainda continuam sem uma resposta ótima). Para ambas variações, nossas implementações conseguem lidar com instâncias pequenas comumente utilizadas na literatura. Em média, tratamos tais instâncias com tempos de execução que vão de milissegundos até poucos minutos.

Também apresentamos, para cada algoritmo explicado, uma forma de reduzir o espaço de busca: uma implementação da regra de corte Jackson Dominance Rule e uma aproximação do princípio de utilizar estações preenchidas de maneira ótima proposto por Jackson. Os impactos dessas otimizações são discutidos, medidos e comparados com os algoritmos do estado-da-arte.

Observações sobre trabalhos importantes (incluindo trabalhos antigos e algortimos que são o estado-da-arte) e pesquisas são feitas com o intuito de direcionar ao leitor da área mais informações sobre problemas de balanceamento de linhas de montagem (em especial, as variantes SALBP e BPP-P).

Palavras-chave: Assembly line problems, sequencing problems, SALBP-1, BPP-P, combinatorial optimization, operations research, dynamic programming.

1 INTRODUCTION

Problems related to assembly lines have been subject of study for many years in fields like Operations Research, Combinatorial Optimization, Computational Complexity and Mathematics. In a very simple and not formal manner, we could define such problems as a set of tasks, each one with a cost information associated, and workstations which can execute the referred tasks. An assignment of each task to one station is a solution to such assembly line problem.

Interesting problems arise due to the large number of variations and constraints that can be applied to this simple description. Using the workstations dimension, we could think of different station models: parallel stations executing their tasks simultaneously; a linear flow of stations; stations with different performance depending on which task is being executed or the skills of the worker which operates the work station. Regarding the tasks dimension, we can mention the presence or not of some kind of order (or dependence) between tasks.

Towards the optimization subject, the most common variables are: the number of work stations necessary to execute all tasks; the maximum processing time that can be assigned to a single workstation (called cycle time); the amount of time that one station executes no task (idle time).

Due to the combinatorial nature of these problems, besides computing a solution, we are also interested in strategies to reduce the time necessary to retrieve a solution. The combination of the above cited points – variations of models, constraints and optimization subjects – may lead to different ways of treating these problems. Classical and state-of-the-art algorithms may use branch-and-bound, dynamic programming or a combination of these techniques (aiming to reduce the amount of time to evaluate partial solutions and generate the final and optimal answer). Fathoming rules, lower and upper bounds and different strategies to explore the generated partial solutions may be used in order to optimize memory usage. Both problems presented here (SALBP-1 and BPP-P) are NP-hard (see (Scholl e Becker 2006) and (Dell'Amico et al. 2012) for more information about problems' complexity).

Our goal in this thesis is to study the use of dynamic programming to solve ALBP. SALBP-1 and BPP-P variants were used because they represent very simple models of assembly lines. We focused on the implementation and adaptation of already known algorithms and dynamic programming models. The analysis of the results also directed us to explore how the problems' properties could be used in order to reduce the search space and, therefore, improve our implementation memory and execution time performance.

For further information regarding ALBP, consider accessing www.assembly-line-balancing.de. This website provides extensive information about many variations of ALBP, including important data sets, references and optimal (or best) answers to each instance.

1.1 Structure of this thesis

This work is organized as follows. Chapter 2 presents the SALBP-1 variant and Chapter 3 presents the BPP-P. Both chapters present the problems with the same structure. First we explain the relationship of the ALBP variant with this work and why we studied it (2.1 and 3.1). Then we make a brief review of important works that were used as basis of our implementation, are considered to be on the state of the art or that compile a good source of information for ALBP in general(2.2 and 3.2). Formal poblem definition is given in 2.3 and 3.3, together with examples of instances and optimal solutions. For each variant, we present two algorithms to solve it: one that generates partial solutions in a task-based way (2.4.1 and 3.4.1) and other that uses a station-based version to do this (2.4.2 and 3.4.2). Sections 2.5 and 3.5 will present fathoming rules to reduce the search space of these algorithms. The results for each association of algorithm and fathoming rule and the appropriate remarks are given in 2.6 and 3.6.

We observe that, since SALBP-1 and BPP-P are very similar, we will focus on explaining and defining each topic in Chapter 2. Chapter 3 will consider how SALBP-1 algorithms and definitions are adapted to deal with BPP-P.

Finally, Chapter 4 summarizes our contributions, considering also the possibilities of how our algorithm could be extended.

2 DYNAMIC PROGRAMMING FOR THE SIMPLE ASSEMBLY LINE BALANCING PROBLEM

2.1 Motivation

SALBP has been subject of study for many years. We can find publications from the middle of the 50's trying to deal with this balancing problem - like (Jackson 1956) - until state of the art works as (Sewell e Jacobson 2012). Although more complex and interesting variants of ALBP exist, SALBP is a good starting point, since its definitions and constraints can be easily understood and could be adapted and mapped to more difficult variants. Moreover, properties observed from SALBP behaviour were the basis for many heuristics, reduction and fathoming rules, lower and upper bounds computation (used in branch and bound algorithms), constructing an important framework that can be adapted and used with less explored task-assignment problems.

SALBP is used here as a means to i) see how ALBP could be solved using purely dynamic programming techniques (the mainstream technique nowadays is branch and bound or a mix of branch and bound and dynamic programming - called branch, bound and remember) and ii) see how we could use our implementations from SALBP in BPP-P, an ALBP with almost no work done before. To the best of our knowledge, the algorithm proposed here is the second to solve exactly this problem, being the first one to use dynamic programming.

2.2 Related Work

From the 50s to the 70s, the focus was directed to strategies regarding successive approximations to the optimal answer (like (Held et al. 1963) and (Horowitz e Sahni 1976)), using dynamic programing techniques together with formal propreties of the problem's definition. Since computer memory is also a hard constraint and, as a consequence, many techniques to store, encode and decode generated partial solutions were born in this context. Papers (Kao e Queyranne 1982), (Schrage e Baker 1978), (Lawler 1979) and (Hoffmann 1963) are a good evidence of this evolution.

Current works tend to use branch and bound methods (sometimes associated with dynamic programming), since the great number of fathoming, problem reducing and preprocessing rules that were proposed can boost these algorithms. One of the best algorithms for treating the SALBP-1 that evidence this strategy is presented by (Sewell e Jacobson 2012), where also different strategies to find solutions in the search space are presented.

For completeness, we remark the survey works in (Scholl e Klein 1999),

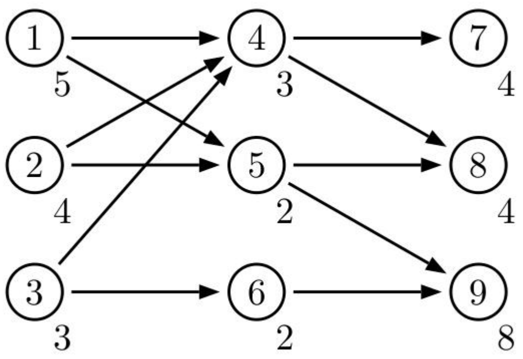

Figure 2.1: Graph of an assembly line balancing problem. Inside the nodes, we have the tasks' labels. Below each node, the processing time for that task.

(Scholl e Becker 2006) and (Fleszar e Hindi 2003) that compile important data related to properties of SALBP and how the classical algorithms were adapted and improved in new methods for exact and approximate solutions of SALBP.

Finally, the interested reader may look to (Nicosia et al. 2002) where SALBP algorithms are adapted to treat the case where the workstations are not identical.

2.3 Problem Definition

The simple assembly line balancing problem (SALBP) is defined as follows. Let $T = \{1, 2, 3, ..., t\}$ be the set of tasks to be assigned to the workstations. Each task i has a processing time, called t_i (a positive integer). Let P be a partial order relation expressed as $i \to j$ (or $i \leq j$), meaning that "task i should be executed before or with task j" (also defined as i precedes j). All workstations are identical and have a capacity, called cycle time (c), expressed in the same unit as the cost information of tasks.

When c is given and we try to minimize the necessary number of stations needed to assign all tasks without violating the cycle time (i.e. the total time of the tasks allocated to a station is less then c) and precedence constraints, we have the variant commonly known as SALBP-1. We can represent this scenario graphically mapping the partial order relation into the edges of a directed graph. T is the set of nodes of such graph - as presented at Figure 2.1.

One possible optimal assignment for the precedence graph at Figure 2.1 with cycle time 10 is shown at Figure 2.2.

2.4 Applying Dynamic Programming to SALBP-1

A feasible subset of tasks S is a set that for every task $t \in S$, all predecessors of t are also in S. For a feasible subset S, $Q(S)$ is defined as the set of tasks in S without successors in S (formally, $Q(S) = \{j \in S : S - \{j\} \text{ is feasible}\}$). The minimum cost function for a feasible set $F(S)$ is given by the recurrence relation 2.1 due to (Kao e Queyranne 1982) and (Lawler 1979).

$$F(S) = \begin{cases} 0\,, & S = \emptyset \\ \min_{j \in Q(S)}\{F(S), F(S-j) + \Delta(F(S-j), t_j)\} & \text{otherwise} \end{cases} \tag{2.1}$$

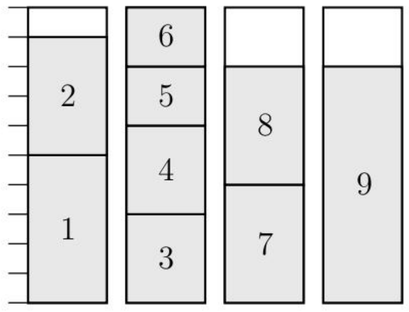

Figure 2.2: Task assignment of one possible solution for SALBP-1 of instance in Figure 2.1 and cycle time = 10. The vertical axis represents the time units. From left to right, we show the order the stations should appear in this assembly line.

Δ function definition is given in Equation 2.2:

$$\Delta(F, t_j) = \begin{cases} t_j\ , & t_j \leq C - F(S) \bmod C \\ C - F(S) \bmod C + t_j & \text{otherwise} \end{cases} \tag{2.2}$$

This cost is computed through the Δ function, reaching to two possibilities:

- Task j fits in the last station, so we just add its cost t_j or,
- Task j does not fit. In this case, we open a new station and the new cost value is incremented by the idle time of the last station and t_j.

2.4.1 Task-based loop version

Algorithm 1 shows how we can translate Equations 2.1 and 2.2. Basically, at the i-th step of the outer **for** loop, we generate feasible sets with i tasks, storing them and their cost in L_{p_i}. These feasible sets are obtained by means of adding the current *free* tasks to the previous generated feasible sets in $L_{p_{i-1}}$. Initially we set L_{p_0} with pair $(\emptyset, 0)$. Task t is said to be free if none of its predecessors are unassigned (δ^- function computation). The tasks already assigned of element S from L_{p_i} are accessed using *.tasks* operation and, at the moment we add one configuration to L_{p_i}, we also update δ^- information.

Algorithm 1 SALBP-1 task-based pseudocode

```
1: for $m = 1 \to |T|$ do                              ▷ $T$ is the set containing all tasks
2:     for all $S \in L_{p_{m-1}}$ do
3:         for all $j \in T : \delta^-(j) = 0$ do
4:             $L_{p_m} \leftarrow L_{p_m} + (S.tasks + j, F(S.tasks + j))$
5:         end for
6:     end for
7: end for
```

2.4.2 Station-based loop version

One of the main contributions regarding ALBP heuristics is the station-based strategy proposed in (Hoffmann 1963). The goal here is very clear: at each step, we should generate a new partial solution with one more station. The idle time will be the minimum possible for all stations processed until this step.

We will present the implementation proposed by (Fleszar e Hindi 2003). The main difference is that, while the original algorithm encourages using operations over the precedence matrix of the instance graph, this one relies on expanding lists with partial solutions.

The Algorithm 2 assumes that we have a global list L with the current *free* tasks (in the same sense as in Subsection 2.4.1). Configurations with the minimum idle time are obtained as follows:

1. Call GETBESTLOAD(1) (start expanding from the first possible free task);
2. Use ADDNEWAVAILABLE to update L;
3. Recursively try to add the next element in L;
4. Returning from recursive calls, remove task added at step 2 and update L with REMOVENEWAVAILABLE;
5. Try to add next element in L (equivalent to jump to first step).

If we reach the end of L list, we are at the base case: we should evaluate if the idle time found is better than the current one for this configuration.

Algorithm 2 can be adapted to generate all maximally loaded states. We modify GETBESTLOAD function, which gives us the Algorithm 3. In summary, the following situations occur:

1. A new parameter, called m, will store the current max idle value for one recursion branch. Max idle value is defined as the minimum processing time of the current excluded tasks.
2. We have an external loop (that does not appear in Algorithm 3) that will call GETMAXLOADS, expanding all partial solutions generated at the previous step. List L is set at this loop and GETMAXLOADS(1, ∞) is called here too. This means that we are expanding the first element of L and no tasks were already excluded. The number of steps of this loop indicate how many workstations were used. The execution stops when we generate one partial solution that have all tasks assigned;
3. GETMAXLOADS will now fill the list L_p with all new maximal loads;
4. The condition to find one maximally loaded configuration is: if none of the already excluded tasks fit in the last station, we are facing a maximally loaded partial solution.

2.5 Fathoming rules for SALBP-1

We present two fathoming rules, both relying on the same principle (as said in (Jackson 1956)): an optimal task assignment will always have stations that are *maximally loaded* (i.e., no other task could be added to one station).

Algorithm 2 Hoffmann best-load heuristic pseudocode

```
function ADDNEWAVAILABLE($i$)
    for all $j \in S_i^*$ do                          ▷ $S_i^*$ are the direct successors of $i$.
        $\delta^-(j) = \delta^-(j) - 1$
        if $\delta^-(j) = 0$ then
            $L \leftarrow L + \{j\}$
        end if
    end for
end function
function REMOVENEWAVAILABLE($i$)
    for all $j \in S_i^*$ do
        if $\delta^-(j) = 0$ then
            $L \leftarrow L - \{j\}$
        end if
        $\delta^-(j) = \delta^-(j) + 1$
    end for
end function
function GETBESTLOAD($q$)                             ▷ look to $r$-th element of $L$
    for $r = q \rightarrow |L|$ do
        $i \leftarrow L_r$
        if $t_i \leq I(A)$ then
            $A \leftarrow A + i$
            ADDNEWAVAILABLE($i$)
            GETBESTLOAD($r + 1$)
            REMOVENEWAVAILABLE($i$)
        end if
        if $I(A) < I(A^*)$ then                        ▷ $I()$ returns the idle time.
            $A^* \leftarrow A$                         ▷ $A^*$ is the current best configuration
        end if                                         ▷ with minimum idle time.
    end for
end function
```

Algorithm 3 SALBP-1 station-based pseudocode

```
function GETMAXLOADS($q, m$)
    for $r = q \rightarrow |L|$ do
        $i \leftarrow L_r$
        if $t_i \leq I(A)$ then                        ▷ $I()$ is the station's idle time.
            $A \leftarrow A + i$                       ▷ $A$ is the current configuration.
            ADDNEWAVAILABLE($i$)                       ▷ being expanded.
            GETBESTLOAD($r + 1, m$)
            REMOVENEWAVAILABLE($i$)
            $m \leftarrow min(m, t_i)$
        end if
        if $I(A) < m$ then                             ▷ maximally loaded station found
            $L_p \leftarrow L_p + A$
        end if
    end for
end function
```

2.5.1 Task-based loop rule

In the spirit of just expanding states that are maximally loaded (see maximum load rule at (Jackson 1956)), we present Algorithm 4, which should be called between lines 6 and 7 of Algorithm 1.

Informally, we could define this rule as follows. For two partial solutions $S1$ and $S2$ with t tasks assigned, if $S1$ uses less workstations than $S2$, then we can cut $S2$.

A important remark is that this rule will not generate always maximally loaded configurations. Once we are using a task-based procedure, there will be cases that is possible to add (to all current partial solutions) more than one task to the last station. In such cases, we should not cut these configurations, as they may lead (after some steps) to maximally loaded partial solutions. This said, we should consider Algorithm 4 as an approximation of what is done with the Maximum load rule.

Therefore, this rule will only be useful when the new partial solutions differ in the number of stations. In this case, we should favor those configurations with less stations, because they process the same amount of tasks with less station resources.

Algorithm 4 SALBP-1 task-based cut rule

```
1:                                    ▷ let $current\_stations$ be the minimum number
2:                                    ▷ of stations of current partial solutions.
3: function cutOpenedStations($L_{p_m}$)
4:     for all $L \in L_{p_m}$ do
5:         if $stations(L) > current\_stations$ then
6:             $L_{p_m} \leftarrow L_{p_m} - L$            ▷ cut partial solutions that
7:         end if                                ▷ needed to open a new station.
8:     end for
9: end function
```

2.5.2 Station-based loop rule

The Jackson Dominance rule (Jackson 1956) aims to cut the partial solutions that are not maximally loaded. Informally, we could state that the Jackson Dominance rule will exclude all partial solutions that, in their last station, have one task that is *dominated* by an unassigned task. Task j dominates i if:

1. The followers of j (F_j) are, at least, the same as the followers of i;
2. The cost of j (t_j) is, at least, the same of i;
3. i could be perfectly replaced by j (meaning that the cycle time and precedence relationship are not violated).

What is intended with this procedure is to prioritize configurations that are maximally loaded and that would allow more assignments in the next step (achieved through the comparison of the followers sets).

Items 1 and 2 of this rule are formally expressed through Equation 2.3 (with addition of one clause to prevent the existence of symmetric pairs).

$$D_i = \begin{cases} \{j : t_j \geq t_i \wedge F_j \supsetneq F_i\} \\ \{j : t_j = t_i \wedge F_j \supsetneq F_i\} \\ \{j : t_j = t_i \wedge F_j = F_i \wedge i < j\} \end{cases} \tag{2.3}$$

The D_i sets different from $\emptyset$ for the example of Figure 2.1 are

$$D_2 = \{1\}$$
$$D_6 = \{5\}$$
$$D_8 = \{7\}$$

It is important to remark that these two steps take no cycle time information into consideration. It is sufficient just to know the precedence relationship. The consequence is that D_i can be precomputed and retrieved during the execution of Algorithm 5, which will effectively prune unnecessary partial solutions.

Algorithm 5 SALBP-1 station-based cut rule

```
1: function JACKSONRULE($A$)
2:     for all $t \in A$ do
3:         for all $j \in D_t$ do
4:             if $j \notin A \wedge$ CANREPLACE$(t, j)$ then
5:                 return                                  ▷ A is pruned
6:             end if
7:         end for
8:     end for
9:     return $L_p \leftarrow L_p + A$                          ▷ A is not pruned
10: end function
```

Function CANREPLACE(t, j) used in Algorithm 5 will deal with item "*i could be perfectly replaced by j (meaning that the cycle time and precedence relationship are not violated)*" defined before. It will return true if:

1. All predecessors of j are already assigned;
2. Exchanging j with t does not violate the cycle time in workstation A.

This fathoming rule should substitute line 12 of Algorithm 3.

2.6 Results

The tests were conducted with a Core i7 930 processor with 12 GiB of memory. The development environment was: C++ language (together with Boost Libraries and Standard Template Library), Linux operating system and GCC compiler.

We used a time limit of one hour and also put limits to memory usage: no instance can generate more than 10^6 states from one step to the other (column **max** L_p). However, there is no limit to the total number of explored states.

For SALBP-1, we executed our algorithm for 269 well known instances of the literature. This data set is generated from 25 precedence graphs, varying the cycle time constraint. For the four possible combinations of feasible set generation procedure and cut rule (task-/station-based and without/with rule), we present the results grouped by precedence graph, giving the number of tasks (**t**), instances (**i**), number of instances with optimal answer found without reaching memory limit (**s**), number of instances with optimal answer found reaching the memory limit (**h**) and instances where optimal answer is not found (**n**). The averages of the following variables are also provided:

diff: absolute difference from the optimal answer;

max L_p: maximum number of states generated at one loop step;

exp. states: total number of explored states;

time: execution time, in seconds;

alloc($\%$): percentage of assigned tasks;

In the following subsections, we present the results in Tables 2.1-2.8 and the remarks about improvements and differences of each implementation.

Then, Table 2.10 compare the results of those tests with state-of-the-art algorithm in (Sewell e Jacobson 2012).

2.6.1 Task-based without fathoming rule

Our algorithm provides the optimal answer in 76.95% cases with an average time of 194.30 seconds (in 57.25% of instances the best answer is obtained without the memory limit). We can only reach the optimal answer (without reaching memory limit barrier) for instances with less than 89 tasks (e.g., LUTZ2 and LUTZ3). For instances that are completely unsolved, we can assign approximately 119, 122 and 71 tasks respectively (BARTHOL2, BARTHOLD and SCHOLL). Since these values are close to 89, we could loosely consider this value as the maximum number of tasks for an instance that we are going to give the optimal answer. Differences occur since other variables may influence the number of partial solutions (e.g., a bigger number of precedence relations implies in more restrictions to assign tasks. So, the number of feasible configurations is reduced).

It is not our goal here to discuss and define the properties of the precedence graphs. We suggest the reader to access the website mentioned in 2.6 for further information.

2.6.2 Task-based with fathoming rule

The optimal answer is reached in 80.30% cases with an average time of 167.81 seconds (in 57.25% of instances the best answer is obtained without the memory limit). In average, with the cut rule, time is reduced by 18.47%, **max** L_p is reduced by 13.65% and the total number of searched states is reduced by 15.00%.

Regarding the maximum number of tasks between the solved instances, no improvement occurred (we continue with 89). The instances solved remain the same. We also note that a small number of instances changed from **Not solved** to **Optimal with heuristic** status: 9. The percentage of allocated tasks in not solved instances improved (this can be seen comparing entries BARTHOL2, BARTHOLD and SCHOLL of tables 2.1 and 2.3).

2.6.3 Station-based without fathoming rule

In 77.32% cases the optimal assignment is found (57.25% of instances without the memory limit). This result is similar to what is shown in Subsection 2.6.1. Memory usage is reduced by an order of magnitude in relation to task-based implementation without fathoming rule. When we found the correct answer, the average time taken is 138.01 seconds.

Instances with at most 89 tasks are the ones that can be solved (which is similar to the task-based version). The number of solved, optimal with heuristic and unsolved instances is also close to the ones obtained before (154, 54 and 61, respectively).

Table 2.1: Results for SALBP-1 grouped by graph - task-based without fathoming rule.

graph	**t**	**i**	**diff**	**max L_p**	**exp. states**	**time(s)**	**alloc(%)**	**s**	**h**	**n**
ARC83	83	16	0.00	255418.00	6499481.00	90.80	100.00%	16	0	0
ARC111	111	17	0.00	1000000.00	85925996.00	1719.89	100.00%	0	17	0
BARTHOL2	148	27	N/A	1000000.00	112649293.07	3599.97	81.13%	0	0	27
BARTHOLD	148	8	N/A	1000000.00	115575219.00	3599.91	83.11%	0	0	8
BOWMAN8	8	1	0.00	3.00	15.00	0.00	100.00%	1	0	0
BUXEY	29	7	0.00	164.00	2062.00	0.01	100.00%	7	0	0
GUNTHER	35	7	0.00	152.00	2289.00	0.01	100.00%	7	0	0
HAHN	53	5	0.00	989.00	6489.00	0.05	100.00%	5	0	0
HESKIA	28	6	0.00	38920.00	326601.00	2.41	100.00%	6	0	0
JACKSON	11	6	0.00	9.00	51.00	0.00	100.00%	6	0	0
JAESCHKE	9	5	0.00	4.00	17.00	0.00	100.00%	5	0	0
KILBRID	45	10	0.00	48786.00	626574.00	5.99	100.00%	10	0	0
LUTZ1	32	6	0.00	16.00	244.00	0.00	100.00%	6	0	0
LUTZ2	89	11	0.00	9431.00	122565.00	1.69	100.00%	11	0	0
LUTZ3	89	12	0.00	9431.00	122565.00	1.50	100.00%	12	0	0
MANSOOR	11	3	0.00	8.00	46.00	0.00	100.00%	3	0	0
MERTENS	7	6	0.00	5.00	21.00	0.00	100.00%	6	0	0
MITCHELL	21	6	0.00	26.00	199.00	0.00	100.00%	6	0	0
MUKHERJE	94	13	0.00	1000000.00	56231258.00	1593.43	100.00%	0	13	0
ROSZIEG	25	6	0.00	32.00	299.00	0.00	100.00%	6	0	0
SAWYER30	30	9	0.00	330.00	3995.00	0.02	100.00%	9	0	0
SCHOLL	297	26	N/A	1000000.00	60330039.46	3599.97	24.73%	0	0	26
TONGE70	70	16	0.00	227050.00	2514263.00	31.23	100.00%	16	0	0
WARNECKE	58	16	0.00	72884.00	861122.00	8.85	100.00%	16	0	0
WEE-MAG	75	24	0.04	1000000.00	51078881.00	818.93	100.00%	0	23	1

Table 2.2: Results for SALBP-1 - task-based loop without fathoming rule.

	Instances	**% of total**
Solved	154	57.25%
Optimal with heuristic	53	19.70%
Not solved	62	23.05%

Table 2.3: Results for SALBP-1 grouped by graph - task-based with fathoming rule.

graph	t	i	diff	max L_p	exp. states	time(s)	alloc(%)	s	h	n
ARC83	83	16	0	235655.25	5980416.87	79.52	100%	16	0	0
ARC111	111	17	0	1000000	79662572.82	1466.44	100%	0	17	0
BARTHOL2	148	27	0.11	1000000	129230883.37	3577.54	93.56%	0	8	19
BARTHOLD	148	8	N/A	1000000	121068414.75	3599.95	86.82%	0	0	8
BOWMAN8	8	1	0	3	15	0	100%	1	0	0
BUXEY	29	7	0	139.14	1650	0	100%	7	0	0
GUNTHER	35	7	0	94.71	1301.71	0.00	100%	7	0	0
HAHN	53	5	0	874.8	5607.6	0.04	100%	5	0	0
HESKIA	28	6	0	34018.33	280677.33	1.91	100%	6	0	0
JACKSON	11	6	0	6.16	35.33	0	100%	6	0	0
JAESCHKE	9	5	0	2	11.8	0	100%	5	0	0
KILBRID	45	10	0	48503.2	620010.5	5.638	100%	10	0	0
LUTZ1	32	6	0	14.33	205.33	0	100%	6	0	0
LUTZ2	89	11	0	6132.54	85723.54	1.08	100%	11	0	0
LUTZ3	89	12	0	9161.58	112043.08	1.35	100%	12	0	0
MANSOOR	11	3	0	8	45	0	100%	3	0	0
MERTENS	7	6	0	4.66	19.5	0	100%	6	0	0
MITCHELL	21	6	0	20.33	149.33	0	100%	6	0	0
MUKHERJE	94	13	0	1000000	54739888.69	1430.6	100%	0	13	0
ROSZIEG	25	6	0	26	242	0	100%	6	0	0
SAWYER30	30	9	0	275.33	3310.33	0.01	100%	9	0	0
SCHOLL	297	26	N/A	1000000	64678185.26	3599.95	26.21%	0	0	26
TONGE70	70	16	0	198975.75	2150713.93	25.19	100%	16	0	0
WARNECKE	58	16	0	66217.62	772246.31	7.61	100%	16	0	0
WEE-MAG	75	24	0	1000000	46406823.87	672.60	100%	0	24	0

Table 2.4: Results for SALBP-1 - task-based loop with fathoming rule.

	Instances	% of total
Solved	154	57.25%
Optimal with heuristic	62	23.05%
Not solved	53	19.70%

Table 2.5: Results for SALBP-1 grouped by graph - station-based without fathoming rule.

graph	t	i	diff	max L_p	exp. states	time(s)	alloc(%)	s	h	n
ARC83	83	16	0	668452.81	1203141.68	3328.28	59.86%	1	2	13
ARC111	111	17	N/A	268117.94	34035.70	3599.99	28.19%	0	0	17
BARTHOL2	148	27	N/A	442207.74	3709.66	3599.99	7.55%	0	0	27
BARTHOLD	148	8	N/A	575616	575616	3599.91	15.03%	0	0	8
BOWMAN8	8	1	0	4	5	0	100%	1	0	0
BUXEY	29	7	0	171.42	783.42	0	100%	7	0	0
GUNTHER	35	7	0	135.71	472	0	100%	7	0	0
HAHN	53	5	0	104.4	165	0.06	100%	5	0	0
HESKIA	28	6	0	3236.66	4674.83	1.31	100%	6	0	0
JACKSON	11	6	0	6	13.33	0	100%	6	0	0
JAESCHKE	9	5	0	1.8	6	0	100%	5	0	0
KILBRID	45	10	0	43460.7	122123.1	39.38	100%	10	0	0
LUTZ1	32	6	0	18.83	75.5	0	100%	6	0	0
LUTZ2	89	11	0	33513	244827.81	9.00	100%	11	0	0
LUTZ3	89	12	0	30944.75	88664.5	20.79	100%	12	0	0
MANSOOR	11	3	0	5.33	5	0	100%	3	0	0
MERTENS	7	6	0	4.66	8.5	0	100%	6	0	0
MITCHELL	21	6	0	17	27.5	0	100%	6	0	0
MUKHERJE	94	13	N/A	210809.61	211511.76	3599.94	23.56%	0	0	13
ROSZIEG	25	6	0	31	51.16	0	100%	6	0	0
SAWYER30	30	9	0	341.77	1479.66	0.00	100%	9	0	0
SCHOLL	297	26	N/A	244350.88	244350.88	3599.98	6.63%	0	0	26
TONGE70	70	16	0	566189.75	1207735.25	1227.34	100%	16	0	0
WARNECKE	58	16	0	138674.5	814144.31	29.40	100%	16	0	0
WEE-MAG	75	24	0.5	1000000	30503038.87	1294.97	100%	0	15	9

We can also note that most ARC83 instances are not solved, although these instances have 83 tasks (less than our supposed limit). This is the case where other properties influence the results (in this case, the precendence relation, as shown in the website referred in 2.6).

2.6.4 Station-based with fathoming rule

Using the Jackson rule, the number of instances solved is not improved. The average time took to end the algorithm's execution is reduced by 11.26% and memory usage is reduced by 12.12%. The total explored states are reduced by 10.37%. Observing the maximum memory usage, the improvement of this fathoming rule is similar to task-based one. The impact is bigger in execution time.

Table 2.6: Results for SALBP-1 - station-based loop without fathoming rule.

	Instances	% of total
Solved	154	57.25%
Optimal with heuristic	54	20.07%
unsolved	61	22.68%

Table 2.7: Results for SALBP-1 grouped by graph - station-based with fathoming rule.

graph	t	i	diff	max L_p	exp. states	time(s)	alloc(%)	s	h	n
ARC83	83	16	0.00	701628.38	1646792.75	3336.67	57.76%	1	2	13
ARC111	111	17	N/A	290693.06	338706.82	3599.99	28.93%	0	0	17
BARTHOL2	148	27	N/A	423450.59	130366.15	3599.99	7.33%	0	0	27
BARTHOLD	148	8	N/A	485882.25	485882.25	3599.94	15.29%	0	0	8
BOWMAN8	8	1	0.00	4.00	5.00	0.00	100.00%	1	0	0
BUXEY	29	7	0.00	170.57	776.00	0.00	100.00%	7	0	0
GUNTHER	35	7	0.00	116.00	423.00	0.00	100.00%	7	0	0
HAHN	53	5	0.00	41.80	60.40	0.03	100.00%	5	0	0
HESKIA	28	6	0.00	1651.00	2512.50	0.74	100.00%	6	0	0
JACKSON	11	6	0.00	5.33	11.83	0.00	100.00%	6	0	0
JAESCHKE	9	5	0.00	1.60	5.20	0.00	100.00%	5	0	0
KILBRID	45	10	0.00	27256.90	74938.10	28.71	100.00%	10	0	0
LUTZ1	32	6	0.00	16.00	65.17	0.00	100.00%	6	0	0
LUTZ2	89	11	0.00	28353.73	201037.73	7.70	100.00%	11	0	0
LUTZ3	89	12	0.00	10648.08	29655.83	7.11	100.00%	12	0	0
MANSOOR	11	3	0.00	5.00	5.00	0.00	100.00%	3	0	0
MERTENS	7	6	0.00	4.67	8.50	0.00	100.00%	6	0	0
MITCHELL	21	6	0.00	12.67	24.50	0.00	100.00%	6	0	0
MUKHERJE	94	13	N/A	145378.77	145970.85	3599.93	23.57%	0	0	13
ROSZIEG	25	6	0.00	24.67	44.33	0.00	100.00%	6	0	0
SAWYER30	30	9	0.00	331.44	1431.44	0.01	100.00%	9	0	0
SCHOLL	297	26	N/A	218242.08	218242.08	3599.99	6.50%	0	0	26
TONGE70	70	16	0.00	516153.88	1090615.75	1005.00	100.00%	16	0	0
WARNECKE	58	16	0.00	103443.50	648038.56	18.82	100.00%	16	0	0
WEE-MAG	75	24	0.04	1000000.00	28230963.08	578.47	100.00%	0	23	1

The status of the instances are exactly the same (154 were solved, 62 needed the heuristic and 53 were not solved). The maximum number of tasks between solved instances is also equal: 89.

2.6.5 Comparison to the state of the art

Table 2.9 gathers information from the four variants of the algorithm. The best case is achieved with the task-based loop with fathoming rule. In Table 2.10 we compare the number of solved (optimal answer without memory limit) and unsolved (memory limit or not solved) instances of this implementation with the state of the art algorithm in (Sewell e Jacobson 2012). As we observed before, all variants have a similar behaviour. The bigger impact of the fathoming rules is in the execution time and happens in the

Table 2.8: Results for SALBP-1 - station-based loop with fathoming rule.

	Instances	% of total
Solved	154	57.25%
Optimal with heuristic	54	20.07%
Not solved	61	22.68%

Table 2.9: Different SALBP-1 implementations results.

	solved	**heuristic**	**not solved**
task	154	53	62
task with cut	154	62	53
station	154	54	61
station with cut	154	54	61

Table 2.10: Number of instances solved in SALBP-1 - comparisson with (Sewell e Jacobson 2012).

	Our algorithm	Sewell and Jacobson	**Our algorithm**	Sewell and Jacobson
solved	154	269	57.25%	100.00%
unsolved	115	0	42.75%	0.00%

station-based variant.

The best version of the algorithm presented in (Sewell e Jacobson 2012) solves all 269 intances with an average time of 0.43 seconds. Our best implementation is far from this value. In only 67 instances we have average execution times comparable to them.

3 DYNAMIC PROGRAMMING FOR THE BIN-PACKING PROBLEM WITH PRECEDENCE CONSTRAINTS

3.1 Motivation

Until now, BPP-P has recieved little attention. Besides what is proposed here and in (Dell'Amico et al. 2012), we could find no other related works regarding exact solution of this problem. Although BPP-P is very close to SALBP and BPP, problems that have been studied for many years, not even adaptations to treat BPP-P were proposed by other authors.

As we noticed that algorithms and strategies studied by us for SALBP could be easily extended to treat BPP-P, we decided to invest efforts in trying to solve and analyzing this variant of ALBP.

3.2 Related Work

Although theoretical and history purposes may have motivated SALBP studies (as it can be used as basis for real ALBP), (Dell'Amico et al. 2012) list several practical situations that could be mapped into BPP-P: a special case of a multiprocessor scheduling problem with a single resource constraint; strict precedences between parts and its metal shields in a assembly line case-study from Motorola; dynamic reconfiguration of FPGAs (Field-Programmable Gate Arrays) when implementing image processing applications, like JPEG encoding.

3.3 Problem Definition

The bin packing problem (BPP) is defined by n items with non-negative weight (t_j) that should be allocated to identical bins of capacity c. Bin packing problem with precedence constrains (BPP-P) extends BPP adding a precedence relation somehow different from the one presented at Section 2.3: P' is a strict partial order relation expressed as $i \rightarrow j$ (or $i < j$), meaning that "task i should be executed *before* task j".

For the same graph of Figure 2.1, and cycle time $c = 10$, one optimal solution is that of Figure 3.1. The difference of one workstation (or bin) comes from the *strict* partial order relation defined before: second station should be split into two, since task 3 should not be allocated together with tasks 4 and 6.

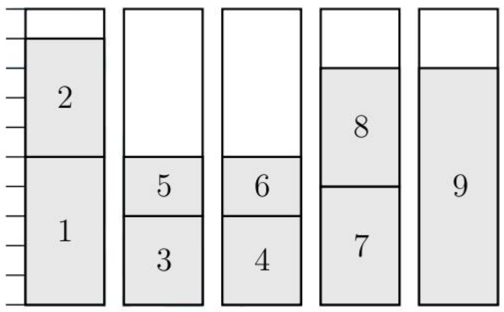

Figure 3.1: Task assignment of one possible solution for BPP-P of instance in Figure 2.1 and cycle time = 10. The vertical axis represents the time units. From left to right, we show the order the bins should appear in this assembly line.

3.4 Applying Dynamic Programming to BPP-P

Assuming that the difference between SALBP-1 and BPP-P is the strict precedence relation of the latter, we propose adding one more condition to the function presented in Section 2.4. The added condition will only consider that a task j fits in the last station (called A) if none of its predecessors are assigned to A (as shown in Equation 3.1):

$$\Delta(F(S),t_j) = \begin{cases} t_j\ , & (t_j \leq C - F(S) \bmod C) \land\ (preds(j) \cap A = \emptyset) \\ C - F(S) \bmod C + t_j & \text{otherwise} \end{cases} \tag{3.1}$$

3.4.1 Task-based loop version

Besides the extension of the recurrence relation shown at Section 3.4, Algorithm 1 can also be used to deal with BPP-P with no other changes.

3.4.2 Station-based loop version

The modifications of Algorithm 3 to deal with BPP-P rely on the auxiliary functions ADDNEWAVAILABLE and REMOVENEWAVAILABLE. In order to forbid one task to be assigned to the same station of its predecessors, these auxiliary methods will just update the current in-degree of task t followers. No task t with $\delta^{-}(t) = 0$ will be appended to or deleted from L at these functions anymore. Once we finished the main procedure, the external loop (the same referred at Subsection 2.4.2) will take care of retrieving the in-degree information and updating the free tasks list L.

The adaptations are shown at Algorithm 6.

3.5 Fathoming rules for BPP-P

The motivation here is the same when we presented the pruning rules at Section 2.5 for SALBP-1: try to only generate configurations that are maximally loaded.

The adaptations (if necessary) are shown in the following subsections.

Algorithm 6 BPP-P station-based adaptations pseudocode

```
function ADDNEWAVAILABLE($i$)
    for all $j \in S_i^*$ do
        $\delta^-(j) = \delta^-(j) - 1$
    end for
end function
function REMOVENEWAVAILABLE($i$)
    for all $j \in S_i^*$ do
        $\delta^-(j) = \delta^-(j) + 1$
    end for
end function
```

3.5.1 Task-based loop rule

The rule described at Section 2.5.1 can be applied to BPP-P without modification, because the observations made about maximal loads suffer no interference from the different definitions of BPP-P.

Observe again that the soundness of this fathoming rule simply relies on the fact that a partial solution $P1$ with one or more non-maximal loads could be substituted by another partial solution $P2$ that has the same number of workstations (all of them maximally loaded). It is easy to see that $P2$ has more tasks assigned than $P1$ (which moves us to a better solution) and that a solution generated through $P2$ will have, at most, the same number of stations of one generated through $P1$.

3.5.2 Station-based loop rule

The adaptation of the Jackson Rule (presented in subsection 2.5.2) is very simple for BPP-P. In fact, Algorithm 5 stays the same as presented before. We just need to add a third clause to the definition of CANREPLACE, resulting in:

1. All predecessors of j are already assigned;

2. Exchanging j with t does not violate the cycle time in station A.

3. No predecessor of j is currently assigned to station A.

3.6 Results

The same information related to the system configuration, data sets and measurements provided at Section 2.6 is valid here. (Dell'Amico et al. 2012) proposed to use the same data set of SALBP-1 and could prove optimality for 266 of the 269 instances. We use their data as basis and, at the best case, we found the correct answer for 191 instances (161 of them found without limiting memory usage). Regarding the three instances without solution, we could find one new solution (WARNECKE, cycle time = 62. The best number of bins is 30). Tables 3.1-3.8 collect the results for each implementation already explained. Remarks regarding improvements and technical details are given in the following appropriate subsections.

Table 3.1: Results for BPP-P grouped by graph - task-based without fathoming rule.

graph	t	i	diff	max L_p	exp. states	time(s)	alloc(%)	s	h	n
ARC83	83	16	0.00	1000000.00	45630859.44	787.14	100.00%	0	16	0
ARC111	111	17	1.06	1000000.00	85922549.82	1906.73	100.00%	0	6	11
BARTHOL2	148	27	N/A	1000000.00	82197346.11	3599.98	59.71%	0	0	27
BARTHOLD	148	8	N/A	1000000.00	88492764.13	3599.98	64.27%	0	0	8
BOWMAN8	8	1	0.00	5.00	20.00	0.00	100.00%	1	0	0
BUXEY	29	7	0.00	2435.00	26229.14	0.16	100.00%	7	0	0
GUNTHER	35	7	0.00	2148.29	24854.71	0.17	100.00%	7	0	0
HAHN	53	5	0.00	58762.00	299348.00	3.03	100.00%	5	0	0
HESKIA	28	6	0.17	1000000.00	12542968.83	145.85	100.00%	0	5	1
JACKSON	11	6	0.00	30.50	133.00	0.00	100.00%	1	5	0
JAESCHKE	9	5	0.00	6.80	24.60	0.00	100.00%	5	0	0
KILBRID	45	10	0.00	1000000.00	20317685.90	281.34	100.00%	0	10	0
LUTZ1	32	6	0.00	88.17	1001.17	0.00	100.00%	6	0	0
LUTZ2	89	11	0.00	756335.55	8395843.73	150.20	100.00%	7	4	0
LUTZ3	89	12	0.00	1000000.00	12168979.50	208.26	100.00%	0	12	0
MANSOOR	11	3	0.00	24.00	120.33	0.00	100.00%	3	0	0
MERTENS	7	6	0.00	12.17	39.50	0.00	100.00%	6	0	0
MITCHELL	21	6	0.00	130.50	836.83	0.00	100.00%	6	0	0
MUKHERJE	94	13	1.00	1000000.00	60372147.69	2213.95	100.00%	0	4	9
ROSZIEG	25	6	0.00	237.17	1746.17	0.00	100.00%	6	0	0
SAWYER30	30	9	0.00	6012.67	64074.11	0.43	100.00%	9	0	0
SCHOLL	297	26	N/A	1000000.00	54024362.92	3599.92	21.98%	0	0	26
TONGE70	70	16	0.06	1000000.00	31316595.50	504.95	100.00%	0	15	1
WARNECKE	58	16	0.06	1000000.00	18394216.50	263.53	100.00%	0	16	0
WEE-MAG	75	24	0.04	1000000.00	58149551.88	1110.89	100.00%	0	23	1

3.6.1 Task-based without fathoming rule

Optimal answer is reached in 68.77% (in only 27.51% of them this result is not due to the memory limit), needing 172.79 seconds, in average. This great number of instances that face the memory limit is reached because the state saved in memory here differs from SALBP-1: when dealing with BPP-P in a task-based loop approach, besides from the already allocated tasks, we need to store which tasks were assigned to the last bin.

For example, imagine state S_1 with tasks 1,2,3,4 already assigned. Consider that tasks 3 and 4 are at the last bin and represent S_1 as $\{\{1,2,3,4\},\{3,4\}\}$. We could construct a state S_2 with the same tasks as S_1, but with just task 4 at the last bin ($S_2 = \{\{1,2,3,4\},\{4\}\}$). Both S_1 and S_2 have to be stored and treated apart. This particular behaviour is due to BPP-P's definition.

In the cases where we do not provide the optimal answer, in average, we allocate approximately 83 tasks (graphs BARTHOL2, BARTHOLD and SCHOLL). Between instances solved with no memory limit, the maximum number of allocated tasks is 89. This range could be considered as the limit to when we give the correct answer and when this does not happen.

Refer to Table 3.1 to see the results by precedence graph.

Table 3.2: Results for BPP-P - task-based loop without fathoming rule.

	Instances	**% of total**
Solved	74	27.51%
Optimal with heuristic	111	41.26%
Not solved	84	31.23%

3.6.2 Task-based with fathoming rule

The presence of the proposed fathoming rule plays great influence here: although the number of correct answers stays similar (69.52%), in 48.70% instances the result is obtained without the 10^6 states limit. Cutting the time almost by the half (48.31%), we achieve 131 cases of solution with no heuristic, reducing memory usage in 71.53%. See Tables 3.3 for the average results by graph.

Regarding the maximum number of allocated tasks when we solve one instance, we have an improvement: the fathoming rule makes this number change from 89 to 111 (graph ARC111). However, there are instances with less tasks that remain unsolved (MUKHERJE, with 94 tasks). The data set from the website referred in Section 2.6 indicates that this may occurr due to the low number of edges present in this graph.

3.6.3 Station-based without fathoming rule

Hoffmann's station-based strategy has a bigger impact than the task-based fathoming rule. Actually, this partial solution generation procedure results in less memory usage because it is not necessary anymore to store last bin's information. Since we are always loading one bin completely, in the next step, we just need to know which tasks were already assigned (without taking care of last bin's situation, since no other task would fit in it).

The algorithm proposed by us, in a mean time of 81.05 seconds, gets the correct number of bins in 65.06% of the instances. See Table 3.5 results grouped by precedence graph.

Seven of thirteen instances of ARC111 estabilish the maximum number of tasks allocated when we give the optimal answer without memory limit: 111 tasks. The same situation regarding MUKHERJE instance explained in last subsection happens here. This just evidences the importance of the precedence constraint when we analyze the memory usage.

3.6.4 Station-based with fathoming rule

Associating Jackson's Dominance rule has little influence on the number of correctly answered instances: 71.00%. Execution time is reduced by 11.26% and memory consumption is reduced by 12.12%.

The percentage of assigned tasks does not improve too much. The same can be said about the maximum number of allocated tasks when we provide the optimal answer.

Accurate and detailed information is provided by Tables 3.7 and 3.8.

3.6.5 Comparison to the state of the art

Table 3.9 gathers information from the 4 implementations. The best case is achieved with the station-based loop with fathoming rule. In Table 2.10 we compare this implementation with the state of the art algorithm in (Dell'Amico et al. 2012).

Table 3.3: Results for BPP-P grouped by graph - task-based with fathoming rule.

graph	**t**	**i**	**diff**	**max L_p**	**exp. states**	**time(s)**	**alloc(%)**	**s**	**h**	**n**
ARC83	83	16	0.00	155424.94	1463115.44	21.87	100.00%	15	1	0
ARC111	111	17	0.76	824868.53	34950929.71	717.78	100.00%	4	3	10
BARTHOL2	148	27	N/A	1000000.00	109554788.78	3599.98	79.13%	0	0	27
BARTHOLD	148	8	N/A	1000000.00	122794115.63	3599.98	89.78%	0	0	8
BOWMAN8	8	1	0.00	2.00	11.00	0.00	100.00%	1	0	0
BUXEY	29	7	0.00	207.86	2330.57	0.01	100.00%	7	0	0
GUNTHER	35	7	0.00	30.14	298.71	0.00	100.00%	7	0	0
HAHN	53	5	0.00	27.00	248.60	0.00	100.00%	5	0	0
HESKIA	28	6	0.00	291152.67	2614904.67	21.71	100.00%	6	0	0
JACKSON	11	6	0.00	7.00	36.67	0.00	100.00%	6	0	0
JAESCHKE	9	5	0.00	3.00	15.40	0.00	100.00%	5	0	0
KILBRID	45	10	0.00	1000000.00	12914264.40	150.19	100.00%	0	10	0
LUTZ1	32	6	0.00	6.00	70.83	0.00	100.00%	6	0	0
LUTZ2	89	11	0.00	22108.64	178332.27	2.65	100.00%	11	0	0
LUTZ3	89	12	0.00	2482.25	11608.00	0.15	100.00%	12	0	0
MANSOOR	11	3	0.00	5.00	34.33	0.00	100.00%	3	0	0
MERTENS	7	6	0.00	6.50	22.33	0.00	100.00%	6	0	0
MITCHELL	21	6	0.00	6.17	45.33	0.00	100.00%	6	0	0
MUKHERJE	94	13	0.92	1000000.00	48546920.31	1510.83	100.00%	0	4	9
ROSZIEG	25	6	0.00	7.83	62.33	0.00	100.00%	6	0	0
SAWYER30	30	9	0.00	444.44	4884.67	0.03	100.00%	9	0	0
SCHOLL	297	26	N/A	1000000.00	59539219.73	3599.95	24.46%	0	0	26
TONGE70	70	16	0.00	14065.19	158943.38	1.76	100.00%	16	0	0
WARNECKE	58	16	0.06	1000000.00	15972212.25	207.49	100.00%	0	16	0
WEE-MAG	75	24	0.04	1000000.00	52951168.83	915.59	100.00%	0	23	1

Table 3.4: Results for BPP-P - task-based loop with fathoming rule.

	Instances	**% of total**
Solved	131	48.70%
Optimal with heuristic	56	20.82%
Not solved	82	30.48%

Table 3.5: Results for BPP-P grouped by graph - station-based without fathoming rule.

graph	t	i	diff	max L_p	exp. states	time(s)	alloc(%)	s	h	n
ARC83	83	16	0.00	2729.94	8649.75	0.26	100.00%	16	0	0
ARC111	111	17	0.33	664014.29	6258362.59	1606.47	97.09%	7	4	6
BARTHOL2	148	27	N/A	468673.81	38724.52	3599.99	7.48%	0	0	27
BARTHOLD	148	8	N/A	639573.75	639573.75	3599.95	12.67%	0	0	8
BOWMAN8	8	1	0.00	1.00	4.00	0.00	100.00%	1	0	0
BUXEY	29	7	0.00	56.14	331.86	0.00	100.00%	7	0	0
GUNTHER	35	7	0.00	13.86	80.43	0.00	100.00%	7	0	0
HAHN	53	5	0.00	2.00	33.00	0.00	100.00%	5	0	0
HESKIA	28	6	0.00	108.83	279.00	0.00	100.00%	6	0	0
JACKSON	11	6	0.00	3.50	12.00	0.00	100.00%	6	0	0
JAESCHKE	9	5	0.00	1.60	7.20	0.00	100.00%	5	0	0
KILBRID	45	10	0.00	898.10	1997.10	0.04	100.00%	10	0	0
LUTZ1	32	6	0.00	3.00	44.67	0.00	100.00%	6	0	0
LUTZ2	89	11	0.00	3751.36	32390.91	0.36	100.00%	11	0	0
LUTZ3	89	12	0.00	117.25	226.00	0.00	100.00%	12	0	0
MANSOOR	11	3	0.00	2.67	10.67	0.00	100.00%	3	0	0
MERTENS	7	6	0.00	3.33	8.50	0.00	100.00%	6	0	0
MITCHELL	21	6	0.00	2.00	14.50	0.00	100.00%	6	0	0
MUKHERJE	94	13	N/A	63385.38	64162.23	3599.92	36.01%	0	0	13
ROSZIEG	25	6	0.00	4.00	15.50	0.00	100.00%	6	0	0
SAWYER30	30	9	0.00	113.56	639.56	0.00	100.00%	9	0	0
SCHOLL	297	26	N/A	394398.27	127541.31	3599.96	7.87%	0	0	26
TONGE70	70	16	0.00	1858.69	5850.69	0.12	100.00%	16	0	0
WARNECKE	58	16	0.06	137445.75	840400.63	13.81	100.00%	16	0	0
WEE-MAG	75	24	0.61	1000000.00	31424986.13	1293.12	99.39%	0	10	14

Table 3.6: Results for BPP-P - station-based loop without fathoming rule.

	Instances	% of total
Solved	161	59.85%
Optimal with heuristic	14	5.20%
Not solved	94	34.94%

Table 3.7: Results for BPP-P grouped by graph - station-based with fathoming rule.

graph	t	i	diff	max L_p	exp. states	time(s)	alloc(%)	s	h	n
ARC83	83	16	0.00	2052.94	6726.94	0.18	100.00%	16	0	0
ARC111	111	17	0.06	612408.35	5561866.71	963.89	100.00%	7	9	1
BARTHOL2	148	27	N/A	484637.37	487945.37	3599.96	7.73%	0	0	27
BARTHOLD	148	8	N/A	539869.63	539869.63	3599.96	11.57%	0	0	8
BOWMAN8	8	1	0.00	1.00	4.00	0.00	100.00%	1	0	0
BUXEY	29	7	0.00	51.71	305.43	0.00	100.00%	7	0	0
GUNTHER	35	7	0.00	10.71	68.86	0.00	100.00%	7	0	0
HAHN	53	5	0.00	2.00	33.00	0.00	100.00%	5	0	0
HESKIA	28	6	0.00	86.17	222.67	0.01	100.00%	6	0	0
JACKSON	11	6	0.00	2.67	10.83	0.00	100.00%	6	0	0
JAESCHKE	9	5	0.00	1.60	6.40	0.00	100.00%	5	0	0
KILBRID	45	10	0.00	301.80	727.30	0.01	100.00%	10	0	0
LUTZ1	32	6	0.00	3.00	44.67	0.00	100.00%	6	0	0
LUTZ2	89	11	0.00	2730.91	22623.82	0.26	100.00%	11	0	0
LUTZ3	89	12	0.00	22.83	118.17	0.00	100.00%	12	0	0
MANSOOR	11	3	0.00	2.67	10.33	0.00	100.00%	3	0	0
MERTENS	7	6	0.00	3.33	8.50	0.00	100.00%	6	0	0
MITCHELL	21	6	0.00	2.00	14.33	0.00	100.00%	6	0	0
MUKHERJE	94	13	N/A	35096.00	35641.62	3599.99	35.02%	0	0	13
ROSZIEG	25	6	0.00	4.00	15.50	0.00	100.00%	6	0	0
SAWYER30	30	9	0.00	110.11	616.56	0.00	100.00%	9	0	0
SCHOLL	297	26	N/A	445202.50	448014.12	3599.98	8.21%	0	0	26
TONGE70	70	16	0.00	1732.94	5364.50	0.11	100.00%	16	0	0
WARNECKE	58	16	0.06	100461.25	671751.56	11.06	100.00%	16	0	0
WEE-MAG	75	24	0.13	1000000.00	30192088.04	560.58	100.00%	0	21	3

Table 3.8: Results for BPP-P - station-based loop with fathoming rule.

	Instances	% of total
Solved	161	59.85%
Optimal with heuristic	30	11.15%
Not solved	78	29.00%

Table 3.9: Different BPP-P implementations results.

	solved	**heuristic**	**not solved**
task	74	111	84
task with cut	131	56	82
station	161	14	94
station with cut	161	30	78

Table 3.10: Number of instances solved in BPP-P - comparison with (Dell'Amico et al. 2012).

	Our algorithm	Dell'Amico et al.	**Our algorithm**	Dell'Amico et al.
solved	161	266	59.85%	98.88%
unsolved	108	3	40.15%	1.12%

The fathoming rules implemented here have different impact in time and memory usage. The bigger decrease is due to the task-based prune rule.

In (Dell'Amico et al. 2012), the average time to obtain the optimal solution is 157.20 seconds. In our best algorithm variant, when we obtain the optimal answer, our average execution time is 73.14 seconds. Although their time is arround the double of ours, we should observe that their solved instances percentage is much better.

4 CONCLUSION

For both ALBP studied, we present two ways to apply dynamic programing model to them: a task-based version (as shown in (Kao e Queyranne 1982), originaly proposed by (Lawler 1979)) and our adaptation from (Hoffmann 1963) heuristic (taking in mind the adaptation available in (Fleszar e Hindi 2003) and following the principle of Maximally Loaded station observed by (Jackson 1956)). For each strategy, we provide two ways of pruning the search space: station-based fathoming rule (mainly known as Jackson Dominance Rule (Jackson 1956)) and a rule for the task-based version, which approximates Jackson's Maximally Loaded Rule.

Pseudocodes, detailed algorithm explanation and adaptations from SALBP-1 to BPP-P are given. The results present and measure the impact of each kind of implementation, giving also a comparison to state of the art algorithm in each variant. For BPP-P problem, where 3 from 269 instances were still with no optimal solution, we prove a new optimal answer (WARNECKE, cycle time 62).

As we could see, simply using dynamic programming and one fathoming rule is not enough to solve large instances of ALBP. Although the rules used here (and better versions of them) are implemented in state-of-the-art algorithms, definitely other strategies like pre-processing techniques and lower and upper bounds tests are also very important when we are trying to reduce the search space.

Regarding future works, the framework to treat ALBP produced here could be extended and aggregated with other methods and procedures used in ALBP handling, in order to analyze their possible influence and usage in more complex scenarios. Also other fathoming rules could be attached to the existing ones, in order to evaluate their impact on the SALBP-1 and the BPP-P variants.

REFERENCES

[Dell'Amico et al. 2012]DELL'AMICO, M.; DÍAZ, J. C. D.; IORI, M. The bin packing problem with precedence constraints. *Operations Research*, v. 60, n. 6, p. 1491–1504, 2012.

[Fleszar e Hindi 2003]FLESZAR, K.; HINDI, K. S. An enumerative heuristic and reduction methods for the assembly line balancing problem. *European Journal of Operational Research*, v. 145, n. 3, p. 606–620, 2003.

[Held et al. 1963]HELD, M.; KARP, R. M.; SHARESHIAN, R. Assembly-line balancing - dynamic programming with precedence constraints. *Operations Research*, v. 11, n. 3, p. 442–459, 1963.

[Hoffmann 1963]HOFFMANN, T. R. Assembly line balancing with a precedence matrix. *Management Science*, v. 9, n. 4, p. 551–562, 1963.

[Horowitz e Sahni 1976]HOROWITZ, E.; SAHNI, S. Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the Association for Computing Machinery*, v. 23, n. 2, p. 317–327, 1976.

[Jackson 1956]JACKSON, J. R. A computing procedure for a line balancing problem. *Management Science*, v. 2, n. 3, p. 261–271, 1956.

[Kao e Queyranne 1982]KAO, E. P. C.; QUEYRANNE, M. On dynamic programming methods for assembly line balancing. *Operations Research*, v. 30, n. 2, p. 375–390, 1982.

[Lawler 1979]LAWLER, E. L. *Efficient Implementation of Dynamic Programming Algorithms for Sequencing Problems*. Stichting Mathematisch Centrum, 1979. (BW (Series)). Available from Internet: <http://books.google.com.br/books?id=DX9OHAAACAAJ>.

[Nicosia et al. 2002]NICOSIA, G.; PACCIARELLI, D.; PACIFICI, A. Optimally balancing assembly lines with different workstations. *Discrete Applied Mathematics*, v. 118, n. 1-2, p. 99–113, 2002.

[Scholl e Becker 2006]SCHOLL, A.; BECKER, C. State-of-the-art exact and heuristic solution procedures for simple assembly line balancing. *European Journal of Operational Research*, v. 168, n. 3, p. 666–693, 2006.

[Scholl e Klein 1999]SCHOLL, A.; KLEIN, R. Balancing assembly lines effectively – a computational comparison. *European Journal of Operational Research*, v. 114, n. 1, p. 50–58, 1999.

[Schrage e Baker 1978]SCHRAGE, L.; BAKER, K. R. Dynamic programming solution of sequencing problems with precedence constraints. *Operations Research*, v. 26, n. 3, p. 444–449, 1978.

[Sewell e Jacobson 2012]SEWELL, E. C.; JACOBSON, S. H. A branch, bound, and remember algorithm for the simple assembly line balancing problem. *INFORMS Journal on Computing*, v. 24, n. 3, p. 433–442, 2012.