Strip Packing Heuristics for the DAGs Packing Problem

Eduardo Bogue[1] Ricardo Santos[2] Edna Hoshino[2] Rubia Oliveira[2]
Hilda Alves[3]
[1]State University of Campinas - Campinas-SP-Brazil
[2]Federal University of Mato Grosso do Sul - Campo Grande-MS-Brazil
[3]Dom Bosco Catholic University - Campo Grande-MS-Brazil

Abstract

Consider two input graphs G_1 and G_2 for a subgraph isomorphism procedure where G_1 is the small graph and G_2 is the large graph and the goal is to find a subgraph G_2' of G_2 which is isomorphic to G_1. In this work we deal with the problem of finding a minimum size for G_2 to be part of a subgraph isomorphism procedure where the inputs are directed acyclic graphs (DAGs). We named this problem as DAGs Packing Problem (DPP). In the DPP, G_1 and G_2 are DAGs. We have adopted an integer linear formulation and designed heuristics for the problem. The heuristics are based on the Strip Packing problem. The results show that the combination of heuristics to convert DAGs into rectangles and strip packing heuristics provide good solutions (minimum strip size) and best performance.

Keywords. dags packing, heuristics, strip packing.
Main Area. TAG - Theory and Algorithms in Graphs.TAG - Teoria e Algoritmos em Grafos.

1 Introduction

Given two graphs G_1 and G_2, the subgraph isomorphism problem consists in finding a subgraph G_2' of G_2 which is isomorphic to G_1 [Garey and Johnson, 1979]. In this work, we deal with the problem of determining the minimum size for graph G_2 such that is still possible to find G_2'. We named this problem as the DAGs Packing Problem (DPP). In the context of DPP G_1 is called an input graph and G_2 is named a base graph. Input graphs can be seen as items to be packed, and the base graphs are the bins where the items should fit.

The DPP is applied in different research areas such as graph theory and optimization. It meets some application in the compiler area as a procedure for scheduling program instructions into a set of processing elements. In that context, the program instructions are grouped in DAGs (items) and the processing elements are organized as a graph representing the resources interconnection (bin). Despite the similarities to the bin-packing problem, the classical bin-packing does not match very well to DPP when applied to the context of compilers. We have to consider sets of processing elements as an unique element to pack the DAGs. Moreover, we can combine sets of processing elements together to form up a new larger set to pack a DAG. The height of this new larger graph represents the execution time and the width the amount of available resources simultaneously. These characteristics make DPP akin to a variation of the bin-packing problem: the Strip Packing Problem [Baker et al., 1980] [Han et al., 2006] [Lee and Sewell, 1999].

In this paper, we propose an integer linear programming formulation for the DPP. Moreover, we present fast heuristics for this problem and we compare the solutions and performance of those heuristics to the exact approach. We also present and discuss results by solving the DPP on instances of well known benchmarks from the compiler area.

The paper is organized as follows: The DAGs Packing Problem is stated in Section 2. We present some algorithms for converting DAGs into rectangles in Section 3. A detailed description of the problem formulation and the proposed heuristics are described in Section 4. Results of the experiments are presented in Section 5. Finally, Section 6 summarizes the main conclusions and future work.

2 The DAGs Packing Problem

Definition 1: A directed graph $D = (V, E)$ where V is the set of vertices and E is the set of edges is acyclic if and only if there is not non-trivial connected strongly components.

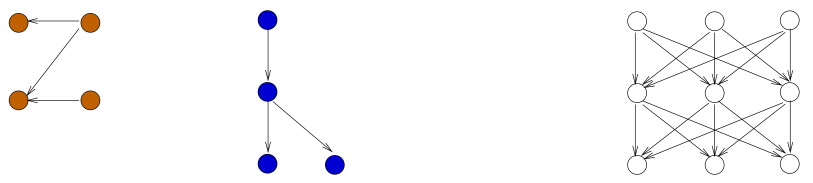
That is, D is comprised of vertices and directed edges, each edge connecting one vertex to another, such that there is no way to start at some vertex V_i and follow a sequence of edges that eventually goes back to V_i again. DAGs may be used to model different kinds of structure in mathematics and computer science [Bondy and Murty, 1976, Garey and Johnson, 1979]. DAGs are used in compilers to represent programs' data flow [Aho et al., 2007]. In the compiler structure, the program operations are represented by the vertices of a DAG while the data dependence among operations are represented by the edges of a DAG.

Definition 2: In a DAG D, a vertex V_i is named root if there is not any input edge to V_i. Similarly, a vertex $V_j,\ i \neq j$ is named leaf if there is not any output edge from V_j.

Definition 3: A critical path (longest path) of a DAG D, with weights on the vertices, is the path with the highest sum of weights.

Definition 4: The problem of determining a minimum size (minimum height) for a base graph G_2 such that G_2 remains subgraph isomorphic to a set of DAGs $S = (D_1, D_2, \ldots, D_n)$, where $D_i = (V_i, E_i), \forall 1 \leq i \leq n$, is named the DAGs Packing Problem (DPP).

Figure 1 exemplifies the DPP on two different DAGs.


(a) Example DAGs (b) Graph G (base graph)

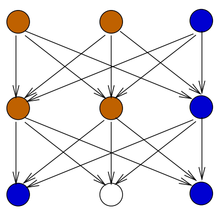

(c) Packing of DAGs into G.

Figure 1: Example of the DAGs Packing Problem

The DAGs in Figure 1(a) can be packed in just one base graph because: (a) nodes of these DAGs do not have more edges than what is available for each node of the base graph in Figure 1(b); (b) the number of nodes of the two DAGs is less than the number of nodes in one base graph, and (c) the length of the longest path of each DAG is less than the number of levels of the base graph.

For small DAGs is quite easy to match each vertex and edge of a DAG to their corresponding vertex and edge of G. However, considering DAGs as a basic tool for modelling dataflow of real-world applications, the DAGs' sizes (heights and widths) depend on the program's characteristics and some applications have more than 100k DAGs with up to 300 vertices each [Santos et al., 2007]. It is necessary to come up with solutions that can map the whole DAG to the graph and, most important, to organize the amount of DAGs under the same graph. In the context of compilers, one DPP application of particular interest is the code generation process performed by compilers for today computer processors. Current computer processor architectures have several hardware resources available to programs' operations. Since DAGs represent the dataflow of programs, it seems straight to map whole DAGs on the processor resources in order to improve the hardware elements usage at runtime. On the other hand, it is a challenge to handle whole DAGs instead of one simple operation (one vertex) by the back-end compiler.

One possible approach to suitably resize a base graph is to represent input DAGs as rectangles and the base graph as a strip of fixed width and infinite height. The problem of matching DAGs moves to another problem aiming at packing those rectangles onto the strip in order to get of minimum height. This is named the Strip Packing Problem and there are heuristics [Han et al., 2006, Imreh, 2001, Lee and Sewell, 1999, Chazelle, 1983, Kenmochi et al., 2009, Ntene and van Vuuren, 2006] and linear programming models proposed for this problem. Before solving the strip packing, we have observed that DAGs can be converted into rectangles on different ways.

One requirement to carry out efficient heuristics to solve the packing of DAGs into a larger graph comes from the time complexity of the "unrolling task". The unrolling task is the step performed after the strip packing, were the base (large) graph must be enlarged in order to be matched to the input DAGs. Given the size of the original input graph, the unrolling task can be a time-consuming activity and it makes it necessary to set a minimum unrolling according to the input DAGs. For instance, edges grow quadratically on the number of vertices [Santos et al., 2007].

Enlarging a base graph is obtained by stacking together copies of the base graph thus creating a new larger base graph able to pack all input DAGs. This "graph stack" combination represents the availability of the resources (vertices and edges) along the time and it is performed by putting each resource on top of the previous one. Figure 2 shows how an "unrolled" base graph is obtained. The original base graph is a 4×4 matrix of vertices where each vertex i has two output edges for vertices $(i + 4 mod 16)$ and $(i + 5 mod 16)$. For simplicity, we are not representing all the edges between vertices as we increase the graph.

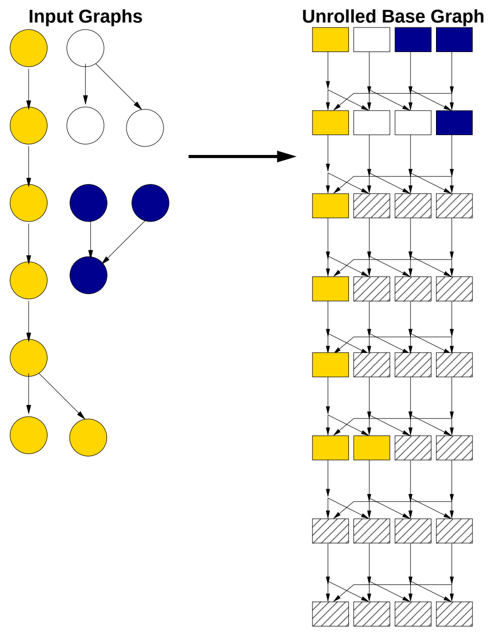


Figure 2: An example of the packing of three DAGs onto an unrolled base graph.

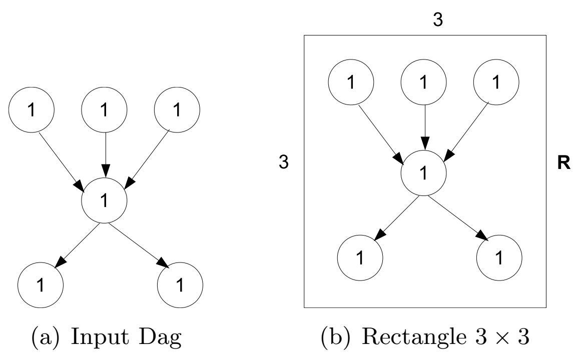
3 Heuristics for Converting DAGs into Rectangles

Consider S the set of input DAGs and G the "unrolled" base graph in the context of DPP. As mentioned before, we represent G as a strip with fixed width and infinite height. The goal of the heuristics to convert DAGs into rectangles is to provide suitable rectangles for solving the strip packing problem. The challenge is how to convert DAGs in S into rectangles in such way to minimize the height necessary to packing all of them in the strip. The design of the heuristics below is primarily based on the Breadth-First Search (BFS) algorithm so that the worst-case complexity for all heuristics is $O(|V| + |E|)$.

3.1 Complete DAG

Given the set of DAGs S, the Complete DAG approach converts each DAG D_i in S into a single rectangle, whose height is the value of the longest path (critical path) of D_i and its width is given as the amount of vertices of the largest level of D_i. Notice that finding the longest path of a weighted DAG can be solved in linear time using an algorithm for tree traversal.

Figure 3 shows the transformation of a *DAG* into a rectangle using the *Complete-DAG* approach.



(a) Input Dag (b) Rectangle 3×3

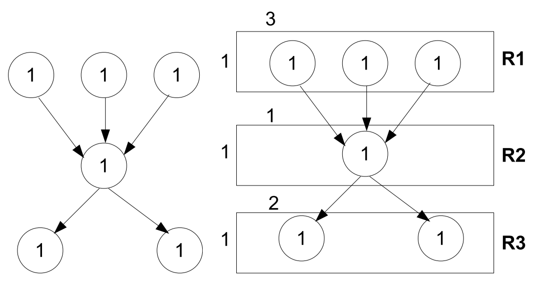
Figure 3: Example of a conversion of a *DAG* into a rectangle using the *Complete-DAG* approach.

In Figure 3, we can observe that, although three resources are allocated, only one of them is used in the second level, and two are used in the third level.

3.2 Level DAG

The *Level DAG* approach proposes the creation of n rectangles R_1,R_2, ..., R_n for each DAG D_i in S, where n is the number of levels of D_i. For each level, we calculate its height through the vertex of highest weight of the level and width as the number of vertices of the level. Furthermore, we store the precedence between the rectangles, where R_{i-1} must be packaged before R_i, $\forall i \in \{2, \ldots, n\}$.

Figure 4 presents three rectangles, with sizes 3×1, 1×1, and 2×1, obtained using the *Level DAG* approach to the same DAG showed in Figure 3. We can notice that the sum of areas of rectangles R_1, R_2, R_3 is less than the area of the rectangle of the Complete-DAG heuristic.

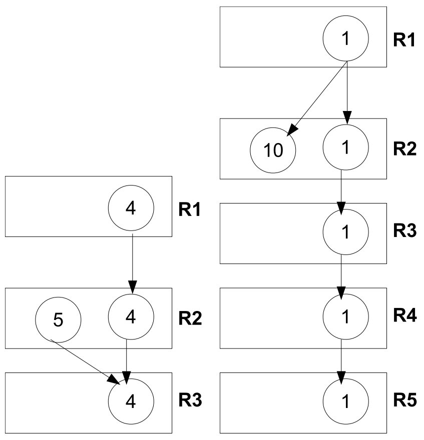


(a) Input DAG (b) 3 rectangles (3×1, 1×1, and 2×1)

Figure 4: Example of converting a *DAG* into rectangles using the *Level-DAG* approach.

3.3 Perfect Level DAG and Perfect Union Level DAG

For the *Level DAG* heuristic, there are cases where the sum of the height of the rectangles is greater than that obtained in the *Complete DAG* approach. Figures 5(a) and 5(b) show some examples. In Example 1, the summation of height is 13 for all rectangles obtained by the *Level DAG* whilst the single rectangle obtained using the *Complete DAG* approach has height 12. The second example shows a DAG where *Level DAG* creates rectangles whose height summation is 14 and the single rectangle obtained by *Complete DAG* has height equal to 11.



(a) Example 1 (b) Example 2

Figure 5: Two examples where the solutions of the *Complete DAG* approach are better than the *Level DAG* approach.

The worst solution of Level DAG is that the heuristic does not take the precedence of the vertices, in each level, into account. The Perfect Level DAG approach takes advantage of this by separating all vertices in oriented sequences of *weight* = 1 thus preventing the influence of a vertex that does not belong to the critical path.

Initially, we calculate the critical path to find out the number of levels. Then, for every vertex V_i with weight $W_{V_i} > 1$, we split vertex V_i into an oriented sequence of P_{V_i} vertices of weight 1. After that, we apply the algorithm of the *Level DAG* approach. Despite solving the problem of increasing height, we have noticed a significant increase in the number of levels. In order to minimize the number of levels, an optimization on the Perfect Level DAG heuristic is adopted. The optimization is named *Perfect Union Level DAG* and it performs the union of levels that have the same area (height and width) since the precedence of levels is preserved.

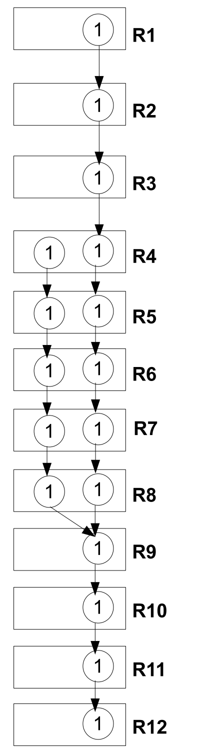
Using the example in Figure 5(a), we demonstrate the approach using *Perfect Level Dag* and *Perfect Union Level Dag* in Figures 6(a) and 6(b), respectively.

4 Heuristics for DAGs Packing

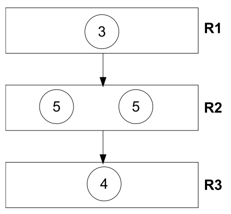
In this section we present heuristics to the DAGS Packing Problem and an integer programming formulation, based on the strip packing problem, which has been used to solve the DPP.

4.1 Bottom Left

Bottom Left (BL) is a heuristic that can be used for solving strip packing problems. The design of BL has a $O(n^2)$, $n =$ number of rectangles, time complexity [Chazelle, 1983]. Initially the rectangles of a set I are sorted in decreasing order of width. Each rectangle $R_i \in I$ is packed closest to the bottom left of the strip. *Bottom Left* is not an algorithm oriented by levels. Figure 7 shows an example using the *Bottom Left* approach.



(a) Results of the *Perfect Level DAG* approach.



(b) Results of the *Perfect Union Level DAG* approach.

Figure 6: Results of Perfect Level DAG and Perfect Union Level DAG heuristics.

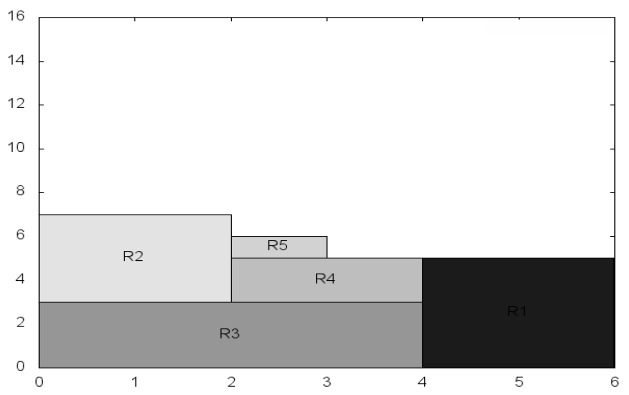


Figure 7: Example of rectangle packing using heuristic *BL*.

In the context of the DPP, we have adopted the Bottom Left heuristic together with Complete DAG heuristic. Complete DAG converts each DAG into a rectangle that can be an input for the Bottom Left heuristic. Section 5 presents the results of the combination between Complete DAG and Bottom Left heuristics.

4.2 DPacking

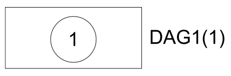
Classical heuristics for the packing problem such as *Best-Fit* and *First-Fit* [Valenzuela and Wang, 2001] are not able to solve the packing problem with constraints on precedence of rectangles. Such precedence arises when heuristics *Level DAG*, *Perfect Level DAG* and

Perfect Union Level DAG are used to convert DAGs into rectangles. To deal with this problem, we propose a new packing heuristic called *DPacking.*

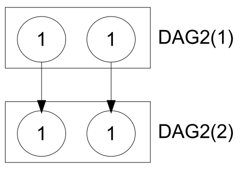
DPacking heuristic is based on the algorithm for the maximum subset sum problem (MSSP) using dynamic programming [Martello and Toth, 1990]. Given an integer c and a set of integers X, the MSSP consists in finding a subset $X' \subseteq X$ such that the sum of the integers in X' is maximum and less than c. The MSSP is a special case of the knapsack problem, where the values of items are equal to their weights. The MSSP based on dynamic programming has a pseudo-polynomial time complexity.

Let a set of DAGs S converted into rectangles and an "unrolled" base graph G represented by a strip with fixed width W. The *DPacking* heuristic selects a set $S' \subset S$ and it finds a subset of rectangles in S' (whose sum of width is closer to W) to be packed first. The rectangles selected to be in S' are those that satisfy the precedence of rectangles. In fact, all rectangles related to the lowest levels of the DAGs are selected first. The heuristic runs this procedure successively until all rectangles have been packed.

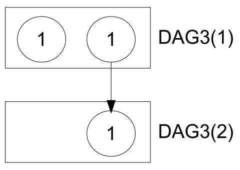
Figure 8 presents three *DAGs* and the levels (rectangles) of each *DAG* are numbered sequentially. Figure 9 shows the result of packing the three DAGs using the *DPacking* heuristic.



(a) DAG 1



(b) DAG 2



(c) DAG 3

Figure 8: Three DAGs converted into five rectangles.

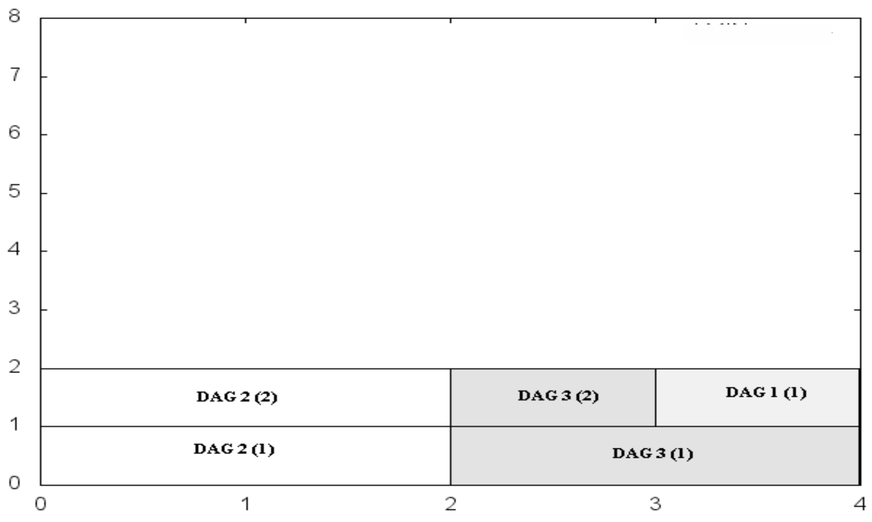


Figure 9: Packing DAGs with rectangle precedences using DPacking.

4.3 ILP Model Formulation

In this section we present the integer linear programming (ILP) formulation proposed by [Lee and Sewell, 1999] for the Strip Packing problem. To apply the formulation to solve the DPP, we first convert DAGs into rectangles using heuristics described in Section 3.

Let $I = \{R_1, R_2, \ldots, R_n\}$ be a set of n rectangles. Each rectangle $R_i \in I$, has width W_{R_i} and height H_{R_i}. The problem of rectangles packing requires that n rectangles are placed in a strip of width W, without overlapping, minimizing the height of the strip.

The proposed model uses four decision variables xl_{R_i}, yl_{R_i}, xu_{R_i} and yu_{R_i}, for each rectangle R_i. Variables xl_{R_i} and yl_{R_i} (xu_{R_i} and yu_{R_i}) refer to coordinates of the lower left (upper right) corner where rectangle Ri is placed. The value of the $HMIN$ variable represents the height of the strip to be minimized. Moreover, two binary variables, l and b, are defined for each pair of rectangles in such way that $l_{R_iR_j}$ is 1 if and only if the rectangle R_i is left of R_j and $b_{R_iR_j}$ is 1 if and only if R_i is below R_j.

$$\begin{array}{rlll}
\text{(F)} & \min HMIN & & \\
\text{subject to} & xu_{R_i} \leq W & , \forall R_i \in I & (1) \\
& yu_{R_i} \leq HMIN & , \forall R_i \in I & (2) \\
& xu_{R_i} = xl_{R_i} + W_{R_i} & , \forall R_i \in I & (3) \\
& yu_{R_i} = yl_{R_i} + H_{R_i} & , \forall R_i \in I & (4) \\
& xu_{R_i} \leq xl_{R_j} + W(1 - l_{R_iR_j}) & , \forall R_i, R_j \in I \text{ such that } R_i \neq R_j & (5) \\
& yu_{R_i} \leq yl_{R_j} + \overline{H}(1 - b_{R_iR_j}) & , \forall R_i, R_j \in I \text{ such that } R_i \neq R_j & (6) \\
& l_{R_iR_j} + l_{R_jR_i} + b_{R_iR_j} + b_{R_jR_i} \geq 1 & , \forall R_i, R_j \in I \text{ such that } R_i < R_j & (7) \\
& HMIN, xu_{R_i}, xl_{R_i}, yu_{R_i}, yl_{R_i} \geq 0 & , \forall R_i \in I & (8) \\
& l_{R_iR_j}, b_{R_iR_j} \in \{0, 1\} & , \forall R_i, R_j \in I. & (9)
\end{array}$$

Constant $\overline{H}$ is an upper bound to $HMIN$ that is the sum of the heights of all rectangles. Constraints (1) and (2) restrict the packing coordinates of each rectangle to the area (width and height) of the strip. The lower left corner coordinates are defined in (3) and (4). Constraints (5) and (6) avoid overlapping of rectangles. Constraints (7) require each pair of rectangles to have a spatial relationship. Constraints (8) require the result and coordinates variables to be positive. Finally, (9) are integrality constraints.

The model proposed in [Lee and Sewell, 1999] takes individual rectangles to be packed on the strip. The model can be adapted to solve Strip Packing Problem when there exists precedence between rectangles, however. In that case, we should set variable $b_{R_iR_j} = 1$ if rectangle R_i must be packed before R_j.

5 Experiments and Results

This section presents the experiments on the heuristics and the ILP model for the DAGs packing problem. The experiments have used DAGs extracted out of programs from well known benchmarks (SPECint and Mediabench) from the computer architecture and compiler areas. The strip has a width equals 4 for all experiments.

Table 1 shows the number of rectangles obtained from each heuristic described in Section 3 for the evaluated programs.

In Table 1, it is worth noting that the number of rectangles of heuristic Complete DAG is the number of DAGs from each program, since Complete DAG performs 1 (DAG) to 1 (rectangle) conversions. Heuristics Level DAG and Perfect Level (PL) create, on average, $1.5\times - 4.6\times$ more rectangles than Complete DAG. Despite not been shown, heuristic Perfect Union Level (PUL) has the best performance among the heuristics (11 out of 12 programs). The average runtime for all heuristics is less than 1 second for the evaluated programs. The

Programs	Complete-DAG	Level-DAG	Perfect-Level	Perfect-Union-Level
168	8,106	16,513	21,055	10,943
175	44,190	86,341	101,170	59,389
181	7,578	16,572	19,140	11,791
197	54,748	105,599	117,172	72,288
255	50,842	89,689	97,914	61,527
256	16,700	35,847	41,157	26,026
433	202	307	532	223
458	181	329	351	203
464	82	172	208	134
470	961	1,821	2,604	1,080
pegwit	60	249	280	162
rasta	393	743	974	513

Table 1: Number of rectangles for instances from Benchmarks SPEC and MediaBench.

performance experiments were run on a Core 2 Duo processor 2.8GHz with 4GB of RAM, and Linux Operating System version 3.4.

Table 2 presents the height of the strip by combining the rectangles conversion heuristics and packing heuristics. All the experiments using the DPacking heuristic were performed together with the PL heuristic. Based on previous experiments, the combination DPacking+PL heuristics has achieved the best results when compared to the DPacking+Level DAG and DPacking+PUL heuristics.

Programs	Complete DAG Bottom Left	Perfect Level DPacking	% Improvement
168	8,663	5,227	39.7
175	53,682	31,282	41.7
181	9,911	4,555	54
197	55,939	28,243	49.5
255	42,643	37,520	12
256	22,014	11,257	48.9
433	146	68	63.4
458	131	125	4.6
464	113	83	26.5
470	734	225	69.3
pegwit	274	61	77.7
rasta	397	362	8.8
Average	7,969.7	13,016.7	34.2

Table 2: Height of the Strip: DPacking and Bottom Left.

In Table 2, the combination between heuristics PL and DPacking have achieved the best results (shorter height of the strip) for all cases. The improvement of DPacking over Bottom Left (fourth column) is up to 77.7% and the average improvement is 34.2%. The results show that real programs have DAGs with complex geometries so that even a simple heuristic such as DPacking can bring better results for the problem when compared to a classical heuristic that does not take the DAG level geometry into account.

It is worth notice that there is not a significant performance difference between bottom left and DPacking. The average runtime of Bottom left is 5.3×10^{-2} seconds and DPacking is 6.6×10^{-1} seconds.

We have performed experiments comparing the ILP model to the strip packing heuristics for a specific set of programs. Despite performing experiments considering all programs, the ILP model has found the solution, according to the time constraint (1 hour), only for 168, 181, and 401 programs. Table 3 presents results of the ILP model considering DAGs from specific functions of programs 168, 181, and 401 (complete). The "*" in the first row means that the ILP model has not found the solution according to the time constraint.

Functions	Complete DAG ILP Model	Complete DAG Bottom-Left	Perfect Level ILP Model	Perfect Level DPacking
168				
dcabs1	11	11	*	*
lsame	36	36	32	32
wupwise	06	06	05	05
181				
_bea_is_dual_infeasible	11	11	11	11
_primal_update_flow	38	38	37	37
_bea_compute_red_cost	10	10	10	10
_flow_org_cost	77	77	77	77
_compute_red_cost	08	08	08	08
401				
maingtu_blocksort	242	242	242	242

Table 3: Height of the Strip: ILP Model, Bottom-Left, and DPacking.

In Table 3, the input DAGs have been converted into rectangles using heuristics Complete DAG and PL. For each conversion heuristic, we have performed experiments considering the ILP model and the packing heuristics. The solutions provided by the ILP formulation with a PL conversion heuristic are equal to the result from the DPacking heuristic for all evaluated DAGs. Despite being experimented for a small set of inputs, the results show that DPacking heuristic is a promising approach for the DAGs Packing problem.

6 Conclusions

In this paper, we presented a set of Strip packing heuristics to solve the DAGs Packing Problem. Our solution approach is based on the strip packing problem so that we designed heuristics to transform DAGs into rectangles and another set of heuristics to pack those rectangles into a strip.

The results show that a proper combination of conversion and packing heuristics bring good solutions and performance. The combination between Perfect Level and DPacking has provided the best results when compared to the Bottom Left heuristic (average improvement of 34.2% on the strip height). The DPacking heuristic has achieved the same results of the ILP model. The performance of Bottom Left and DPacking are quite the same and better than the performance of the ILP model. Program 401 has been packed in $4ms$ using DPacking and Bottom Left, and $222ms$ using the ILP model.

Future work is focusing on the evaluation the ILP model formulation for a wide range of datasets. The integration of the conversion and packing heuristics to a compiler software is under development.

7 Acknowledgments

The authors would like to thank Brazilian Research Agencies CAPES, CNPq, and Fundect-MS for their financial support to this work and research projects in the High Performance Computing Systems Laboratory (LSCAD) of the School of Computing/UFMS. The authors also thank the anonymous reviewers for their feedback to this work.

References

[Aho et al., 2007] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2007). *Compilers - Principles, Techniques, & Tools.* Addison Wesley, 2 edition.

[Baker et al., 1980] Baker, B. S., Coffman, E. G., and Rivest, R. L. (1980). Orthogonal Packing in Two-Dimensions. *SIAM Journal on Computing*, 9(4):846–855.

[Bondy and Murty, 1976] Bondy, J. A. and Murty, U. S. R. (1976). *Graph Theory with Applications.* The Macmillan Press.

[Chazelle, 1983] Chazelle, B. (1983). The bottom-left bin-packing heuristic: An efficient implementation. *IEEE Transactions on Computers*, 32(7):697–707.

[Garey and Johnson, 1979] Garey, M. R. and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company.

[Han et al., 2006] Han, X., Iwama, K., Ye, D., and Zhang, G. (2006). Strip Packing vs. Bin Packing. Technical Report TR06-112, Electronic Colloquium on Computational Complexity Report.

[Imreh, 2001] Imreh, C. (2001). Online Strip Packing with Modifiable Boxes. *Operation Research Letters*, 66:78–86.

[Kenmochi et al., 2009] Kenmochi, M., Imamichi, T., Nonobe, K., Yagiura, M., and Nagamochi, H. (2009). Exact algorithms for the 2-dimensional strip packing problem with and without rotations. *European Journal of Operational Research*, 198:73–83.

[Lee and Sewell, 1999] Lee, H. F. and Sewell, E. C. (1999). The Strip-Packing Problem for a Boat Manufacturing Firm. *IIE Transactions*, 31(7):639–651.

[Martello and Toth, 1990] Martello, S. and Toth, P. (1990). *Knapsack Problems: Algorithms and Computer Implementations.* John Wiley & Sons.

[Ntene and van Vuuren, 2006] Ntene, N. and van Vuuren, J. H. (2006). A survey and comparison of level heuristics for the 2d-oriented strip packing problem. *Discrete Optimization*, 24:157–183.

[Santos et al., 2007] Santos, R., Azevedo, R., and Santos, R. M. O. (2007). A DAGs-Packing Heuristic for a High Performance Processor Architecture. In *Proceedings of the 39th Brazilian Symposium on Operational Research*, Fortaleza-CE.

[Valenzuela and Wang, 2001] Valenzuela, C. L. and Wang, P. Y. (2001). Heuristics for Large Strip Packing Problems with Guillotine Patterns: an Empirical Study. MIC'2001 - 4th Metaheuristics International Conference.