A Comparison of Random Task Graph Generation Methods for Scheduling Problems

Louis-Claude Canon, Mohamad El Sayah, and Pierre-Cyrille Héam

FEMTO-ST Institute, CNRS, Univ. Bourgogne Franche-Comté, France
{louis-claude.canon,mohamad.el_sayah,pierre-cyrille.heam}@univ-fcomte.fr

Abstract. How to generate instances with relevant properties and without bias remains an open problem of critical importance to compare heuristics fairly. When scheduling with precedence constraints, the instance is a task graph that determines a partial order on task executions. To avoid selecting instances among a set populated mainly with trivial ones, we rely on properties such as the *mass*, which measures how much a task graph can be decomposed into smaller ones. This property and an in-depth analysis of existing random instance generators establish the sub-exponential generic time complexity of the studied problem.

1 Introduction

How to correctly evaluate the performance of computing systems has been a central question for a long time [15]. Among the arsenal of available evaluation methods, relying on random instances allows comparing strategies in many diverse situations. However, random generation methods are prone to bias, which prevents a fair empirical assessment. Studying the problem characteristics to constrain the uniform generation on a category of difficult instances is thus critical.

In the context of parallel systems, instances for numerous multiprocessor scheduling problems contain the description of an application to be executed on a platform [17]. This study focuses on scheduling problems requiring a Directed Acyclic Graph (DAG) as part of the input. Such a DAG represents a set of tasks to be executed in a specific order given by precedence constraints. While this work studies the DAG structure for several scheduling problems, it illustrates and analyzes existing generators in light of a specific problem with unitary costs and no communication. This simple yet difficult problem emphasizes the effect of the DAG structure on the performance of scheduling heuristics.

After exposing related works in Section 2, Section 3 lists DAG properties and covers scheduling and random generation concepts. Section 4 analyzes the proposed properties on a set of special DAGs. Section 5 provides an in-depth analysis of existing random generators supported by consistent empirical observations. Finally, Section 6 studies the impact of these methods and the DAG properties on scheduling heuristics. A more detailed version of these results is also available in the extended version [4].

2 Related Work

Our approach is similar to the one followed in [6], which consists in studying the properties of randomly generated DAGs before comparing the performance of scheduling heuristics. Three properties are measured and analyzed for each studied generator: the length of the longest path, the distribution of the output degrees and the number of edges. The authors consider five random generators: two variants of the Erdős-Rényi algorithm, one layer-by-layer variant, the random orders method and the Fan-in/Fan-out method. Finally, for each generator, the paper compares the performance of four scheduling heuristics. The results are consistent with the observations done in Section 5 (Figures 1, 3 and 4) for the length and the number of edges.

Many tools have been proposed in the literature to generate DAGs in the context of scheduling in parallel systems. TGFF (Task Graphs For Free) is the first tool proposed for this purpose [7]. This tool relies on a number of parameters related to the task graph structure. The task graph is constructed by creating a single-vertex graph and then incrementally augmenting it. Until the number of vertices in the graph is greater than or equal to the minimum number of vertices, this approach randomly alternates between two phases: the expansion of the graph and its contraction. The main goal of TGFF is to gain more control over the input and output degrees of the tasks.

DAGGEN was later proposed to compare heuristics for a specific problem [8]. This tool relies on a layer-by-layer approach with four parameters in addition to the number of vertices. The number of elements per layer is uniformly drawn in an interval determined by the width parameter and with a range determined by the regularity parameter. Lastly, edges are added between layers separated by a maximum number of layers determined by the jump parameter. For each vertex, the method adds a uniform number of predecessors in an interval determined by the density parameter.

GGen has been proposed to unify the generation of DAGs by integrating existing methods [6]. The tool implements two variants of the Erdős-Rényi algorithm, one layer-by-layer variant, the random orders method and the Fan-in/Fan-out method. It also generates DAGs derived from classical parallel algorithms such as the recursive Fibonacci function, the Strassen multiplication algorithm, etc.

The Pegasus workflow generator[1] can be used to generate DAGs from several scientific applications [16] such as Montage, CyberShake, Broadband, etc. XL-STaGe[2] produces layer-by-layer DAGs using a truncated normal distribution to distribute the vertices to the layers [3]. This tool inserts edges with a probability that decreases as the number of layers between two vertices increases. A tool named RandomWorkflowGenerator[3] implements a layer-by-layer variant [12].

---

[1] https://confluence.pegasus.isi.edu/display/pegasus/WorkflowGenerator

[2] https://github.com/nizarsd/xl-stage

[3] https://github.com/anubhavcho/RandomWorkflowGenerator

3 Background

Directed Acyclic Graphs Let $D = (V, E)$ be a Directed Acyclic Graph (DAG), where V is a finite set of vertices and $E \subseteq V \times V$ is the set of edges, such that there is no cycle in the graph. The *length* of a DAG is defined as the maximum number of vertices in any path in this DAG and is noted len or k. The *depth* of a vertex v in a DAG is inductively defined by: if v has no predecessor, then its depth is 1; otherwise, the depth of v is one plus the maximum depth of its predecessors. The *shape decomposition* of a DAG is the tuple $(X_1, X_2, \ldots, X_k)$ where X_i is the set of vertices of depth i. The *shape* of the DAG is the tuple $(|X_1|, \ldots, |X_k|)$. The maximum (resp. minimum) value of the $|X_i|$ is called the *maximum shape* (resp. *minimum shape*) of the DAG. Computing the shape decomposition and the shape of a DAG is easy. If $|X_i| = 1$, the unique vertex of X_i is called a *bottleneck vertex*. A *block* is a subset of vertices of the form $\cup_{i < j < i+\ell} X_j$ with $\ell > 1$ where X_i is either a singleton or $i = 0$, $X_{i+\ell}$ is either a singleton or $i + \ell = k + 1$, and for each $i < j < i+\ell$, $|X_j| \neq 1$. We denote by $\mathrm{mass}^{\mathrm{abs}}(B)$ the cardinal of $B = \cup_{i < j < i+\ell} X_j$ and by $\mathrm{mass}^{\mathrm{abs}}(D) = \max\{\mathrm{mass}^{\mathrm{abs}}(B) \mid B \text{ is a block}\}$ the *absolute mass* of D. The *relative mass*, or simply the *mass*, is given by $\mathrm{mass}(D) = \frac{\mathrm{mass}^{\mathrm{abs}}(D)}{n}$.

The *transitive reduction* of a DAG D [2] is the DAG D^T for which: D^T has a directed path between u and v iff D has a directed path between u and v; there is no graph with fewer edges than D^T that satisfies the previous property. Intuitively, this operation consists in removing redundant edges.

Among dozens of DAG properties, we measure the following ones on the transitive reduction of each DAG D: the number of edges m, maximum degree $\deg^{\mathrm{max}}$ and degree Coefficient of Variation[4] $\deg^{\mathrm{CV}}$. For these properties, we specify they are measured on a transitive reduction (e.g. $m(D^T)$ for the number of edges). Moreover, we measure the length, the mean shape $\mathrm{sh}^{\mathrm{mean}}$, the shape CV $\mathrm{sh}^{\mathrm{CV}}$ and the mass. The last measured property is the number of edges in D.

Scheduling We consider a classic problem in parallel systems noted $P|p_j = 1, prec|C_{\mathrm{max}}$ in Graham's notation [11]. The objective consists in scheduling a set of tasks on homogeneous processors such as to minimize the overall completion time. The dependencies between tasks are represented by a precedence DAG: before starting its execution, all the predecessors of a task must complete their executions. The execution cost p_j of task j on any processor is unitary and there are no costs on the edges (i.e. no communication). A schedule defines on which processor and at which date each task starts executing such that no processor executes more than one task at any time and all precedence constraints are met. The problem consists in finding the schedule with the minimum makespan.

This problem is strongly NP-hard [25], while it is polynomial when there are no precedence constraints, which means the difficulty comes from the dependencies. Many polynomial heuristics have been proposed for this problem (see Section 6). With specific instances, such heuristics may be optimal. This is the case when the width does not exceed the number of processors, which leads to a potentially

---

[4] The CV is the ratio of the mean degree to the degree standard deviation.

large length. Any task can thus start its execution as soon as it becomes available. This paper explores how DAG properties are impacted by the generation method with the objective to control them to avoid easy instances.

Although this paper studies random DAGs with heuristics for the specific problem $P|p_j = 1, prec|C_{\max}$, generated DAGs can be used for many scheduling problems with precedence constraints. While avoiding specific instances depending on their width and length is relevant for many scheduling problems, it is not necessary the case for all of them. For instance, with non-unitary processing costs, instances with large width and small length are difficult because the problem is strongly NP-Hard even in the absence of precedence constraints ($P||C_{\max}$) [10].

Mass and Scheduling Consider a DAG $D = (V, E)$ whose minimum shape is 1; there exists a bottleneck vertex v such that the shape of the DAG is of the form $(X_1, \ldots, X_\ell, \{v\}, X_{\ell+1}, \ldots, X_k)$. The scheduling problem for D can be decomposed into two subproblems. Using recursively this decomposition, the initial problem can be decomposed into $n_c + 1$ independent scheduling problems, where n_c is the number of bottleneck vertices.

Applying a brute force algorithm for the scheduling problems computes the optimal results in a time $T \leq n_c T_m$, where T_m is the maximum time required to solve the problem on a DAG with $\text{mass}^{\text{abs}}(D)$ vertices. Since exponential brute force exact approaches exist, it follows that if $\text{mass}^{\text{abs}}(D) = O(\log^k n)$ for a constant k, then an optimal solution of the scheduling problem can be computed in sub-exponential time. Consequently, scheduling heuristics are irrelevant for task graph with polylogarithmic absolute mass. Similarly, the same arguments work to claim that interesting instances for the scheduling problem must have quite a large absolute mass (not in $o(n)$). It is therefore preferable to have instances with no or few bottleneck vertices, that is a unitary mass.
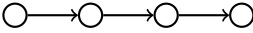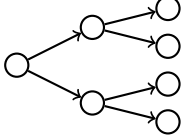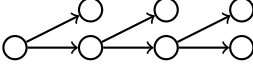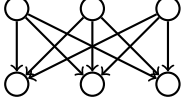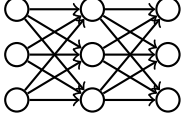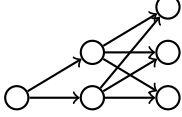
The relevance of the mass property is limited to the class of scheduling problems that contains all problems for which the instance can be cut into independent subinstances.

4 Analysis of special DAGs

To analyze the properties described in the previous section, we introduce in Table 1 a collection of special DAGs. The first three DAGs (D_{empty}, D_{complete} and D_{chain}) constitutes extreme cases in terms of precedence. The next two DAGs ($D_{\text{out-tree}}$ and D_{comb}), to which we can add the reversal of the complete binary tree ($D_{\text{in-tree}} = D_{\text{out-tree}}^R$), are examples of binary tree DAGs. The last three DAGs ($D_{\text{bipartite}}$, D_{square} and $D_{\text{triangular}}$) are denser with more edges and with a compromise between the length and the width for these last two DAGs.

Table 2 illustrates the properties for these special DAGs. The most extreme values are reached with the empty and complete DAGs for the length, number of edges, mass and mean shape. When considering only transitive reductions (i.e. when discarding the complete DAG), the maximum value for the maximum degree is n with a fork or a join (the bipartite DAG reaches half this value).

Table 1. Special DAGs.

Name	description	representation
Empty (D_{empty})	no edge	
Complete (D_{complete})	maximum number of edges	
Chain (D_{chain})	transitive reduction of the complete DAG	
Complete binary tree ($D_{\text{out-tree}}$)	each non-leaf/non-root vertex has a unique predecessor and two successors	
Comb (D_{comb})	a chain where each non-leaf vertex has an additional leaf successor	
Complete bipartite ($D_{\text{bipartite}}$)	$\frac{n}{2}$ vertices connected to $\frac{n}{2}$ vertices	
Complete layer-by-layer square (D_{square})	similar to the complete bipartite with $\sqrt{n}$ layers of size $\sqrt{n}$	
Complete layer-by-layer triangular ($D_{\text{triangular}}$)	similar to the complete layer-by-layer square but the size of each new layer increases by 1	

Proposition 1 states that the maximum number of edges among all transitive reductions is $\left\lfloor \frac{n^2}{4} \right\rfloor$ (reached with the bipartite DAG).

Proposition 1. *The maximum number of edges among all transitive reductions of size n is* $\left\lfloor \frac{n^2}{4} \right\rfloor$.

Proof. Transitive reductions do not contain triangle (i.e. clique of size three), otherwise there is either a cycle or a redundant edge. By Mantel's Theorem [20], the maximum number of edges in a n-vertex triangle-free graph is $\left\lfloor \frac{n^2}{4} \right\rfloor$. This is the case for the complete bipartite DAG because the number of edges is $\frac{n^2}{4} = \left\lfloor \frac{n^2}{4} \right\rfloor$ when n is even and $\frac{n^2-1}{4} = \left\lfloor \frac{n^2}{4} \right\rfloor$ when n is odd.
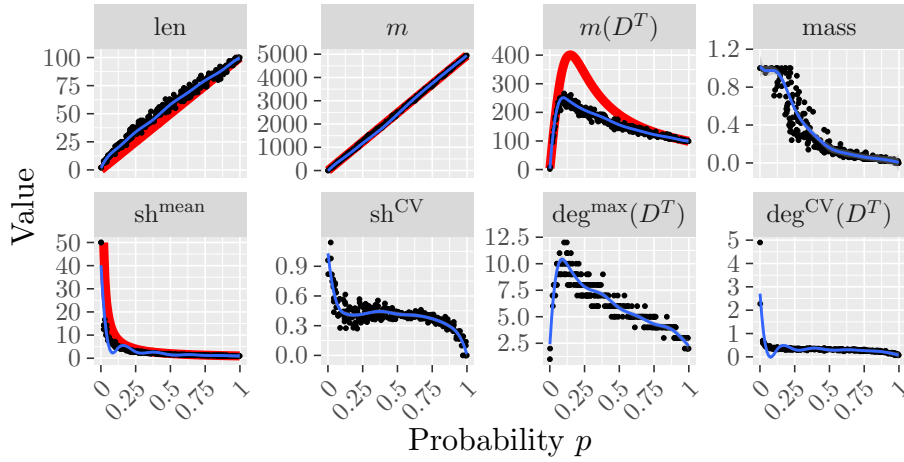
5   Analysis of Existing Generators

Table 2. Approximate properties of special DAGs (negligible terms are discarded for clarity). The exact properties are given in the extended version [4].

DAG	len	m	$m(D^T)$	mass	sh^{mean}	sh^{CV}	$\deg^{\max}(D^T)$	$\deg^{\text{CV}}$
D_{empty}	1	0	0	1	n	0	0	0
D_{complete}	n	$\frac{n^2}{2}$	n	0	1	0	2	$\frac{1}{\sqrt{2n}}$
D_{chain}	n	n	n	0	1	0	2	$\frac{1}{\sqrt{2n}}$
$D_{\text{out-tree}}$ $D_{\text{in-tree}}$	$\log_2(n)$	n	n	1	$\frac{n}{\log_2(n)}$	$\sqrt{\frac{\log_2(n+1)}{3}}$	3	$\frac{1}{2}$
D_{comb}	$\frac{n}{2}$	n	n	1	2	$\frac{1}{\sqrt{2n}}$	3	$\frac{1}{2}$
D_{comb}^{R}	$\frac{n}{2}$	n	n	$\frac{1}{2}$	2	$\sqrt{\frac{n}{8}}$	3	$\frac{1}{2}$
$D_{\text{bipartite}}$	2	$\frac{n^2}{4}$	$\frac{n^2}{4}$	1	$\frac{n}{2}$	0	$\frac{n}{2}$	0
D_{square}	$\sqrt{n}$	$n\sqrt{n}$	$n\sqrt{n}$	1	$\sqrt{n}$	0	$2\sqrt{n}$	$\frac{1}{\sqrt{2\sqrt{n}}}$
$D_{\text{triangular}}$	$\sqrt{2n}$	$\frac{2n\sqrt{2n}}{3}$	$\frac{2n\sqrt{2n}}{3}$	1	$\sqrt{\frac{n}{2}}$	$\frac{1}{\sqrt{3}}$	$2\sqrt{2n}$	$\frac{1}{2\sqrt{2}}$

Random Generation of Triangular Matrices This approach is based on the Erdős-Rényi algorithm [9] with parameter p: an upper-triangular adjacency matrix is randomly generated. For each pair of vertices (i, j) with $i < j$, there is an edge from i to j with an independent probability p. The approach is not uniform. For instance, a random generator that is uniform over all the DAGs generates the empty DAG with probability $1/25$. With $p = 0.5$, the Erdős-Rényi algorithm generates the empty DAG with probability $1/8$.

Figure 1 shows the effect of the probability parameter p on the properties of the generated DAGs. The most remarkable effect can be seen for the number
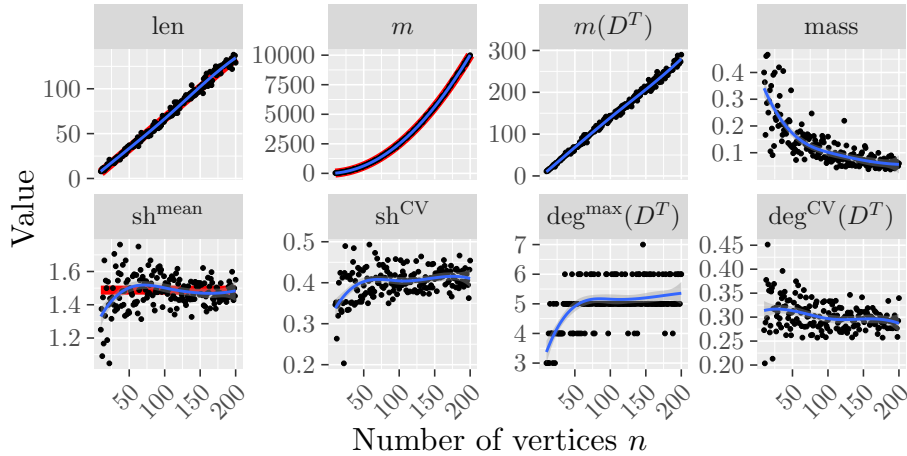

Fig. 1. Properties of 300 DAGs of size $n = 100$ generated with probability p uniformly drawn between 0 and 1 (Erdős-Rényi algorithm). Red lines correspond to formal bounds.

of edges in the transitive reduction $m(D^T)$. This property shows that after a maximum around $p = 0.10$, adding more edges with higher probabilities leads to redundant dependencies and simplifies the structure of the DAG by making it longer. A formal result in the extended version [4, Proposition 4] confirms this effect. DAGs generated with a probability below 5% are almost empty and most edges are not redundant. These DAGs lead to a simplistic scheduling process that consists in starting each task on a critical path as soon as possible and then distributing a large number of independent tasks. Analogously, DAGs generated with probabilities p greater than 15% contain many edges that simplify the DAG structure by increasing the length and thus reducing the mean shape (recall that with a small width, the problem is easy). At the same time, the mass decreases continuously, allowing the problem to be divided into smaller problems.

The effect of probability p illustrates the compromise between the length and mean shape to avoid simplistic instances that are easily tackled.

Uniform Random Generation One way to uniformly generate elements consists in using a classical recursive/counting approach [22] based on generating functions. This counting approach relies on recursively counting the number of DAGs with a given number of source vertices, that is vertices with no in-going edges. See [4, Section 5.2] for a complete algorithm that uniformly generates random DAGs with this approach.

Fig. 2. Properties of DAGs generated by the recursive algorithm for each size n between 10 and 200. Red lines correspond to formal results.
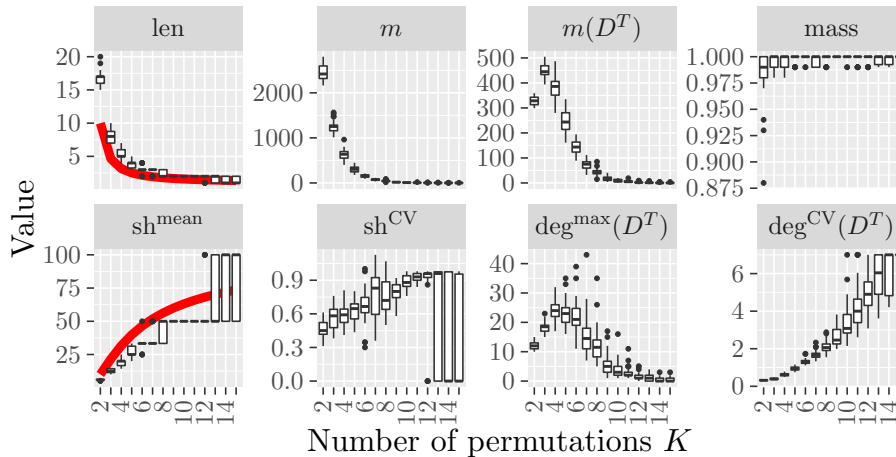
Figure 2 depicts the effect of the number of vertices on the selected DAG properties. The length closely follows the function $\frac{3n}{2}$. This effect is consistent with a theoretical result stating that the expected number of source vertices

sh[1] in a uniform DAG is asymptotically 1.488 as $n \to \infty$ [19]. This implies that the expected value for each shape element is close to this value by construction of the shape, which makes the DAG an easy instance for scheduling problems. Moreover, the number of edges m is almost indistinguishable from the function $\frac{n^2}{4}$, which is indeed the average number of edges in a uniform DAG [21, Theorem 2]. We finally observe that the mass decreases as the size n increases. This is confirmed by the following result (proved in the extended version [4]):

Theorem 1. *Let D be a DAG uniformly and randomly generated among the labeled DAGs with n vertices. One has $\mathbb{P}(\mathrm{mass}^{\mathrm{abs}}(D) \geq \log^4(n)) \to 0$ when $n \to +\infty$.*

Therefore, the mass converges to zero as the size n tends to infinity. As shown in Section 3, such instances can be decomposed into independent problems and efficiently solved with a brute force strategy. This leads to a sub-exponential generic time complexity with uniform instances.

Random Orders The random orders method derives a DAG from randomly generated orders [26]. The first step consists in building K random permutations of n vertices. Each of these permutations represents a total order on the vertices, which is also a complete DAG with a random labeling. Intersecting these complete DAGs by keeping an edge iff it appears in all DAGs with the same direction leads to the final DAG.

Fig. 3. Properties of 420 DAGs of size $n = 100$ generated by the random orders algorithm for each K between 2 and 15. Red lines correspond to formal results.

Figure 3 shows the effect of the number of permutations K on the DAG properties with boxplots[5]. The extreme cases $K = 1$ and $K \to \infty$ are discarded from the figure for clarity. They correspond to the chain and the empty DAG, respectively. The number of permutations quickly constrains the length. For instance, the length is already between 15 and 20 when $K = 2$ and at most 5 when $K \geq 5$. A formal analysis suggests that the length is almost surely in $O(n^{1/K})$ [26, Theorem 3], which is consistent with our observation. Moreover, the mass is always close to one for $K > 1$.

Layer-by-Layer The layer-by-layer method was first proposed by [1] but popularized later by the introduction of the STG data set [23]. This method produces DAGs in which vertices are distributed in layers and vertices belonging to the same layer are independent. This section analyzes the effect of three parameters (size n, number of layers k and connectivity probability p) using the following variant inspired from [6, 12]. First, k vertices are affected to distinct layers to prevent any empty layer. Then, the remaining $n - k$ vertices are distributed to the layers using a balls into bins approach (i.e. a uniformly random layer is selected for each vertex). For each vertex not in the first layer, a random parent is selected among the vertices from the previous layer to ensure that the layer of any vertex equals its depth. Finally, random edges are added by connecting any pair of vertices from distinct layers from top to bottom with probability p.
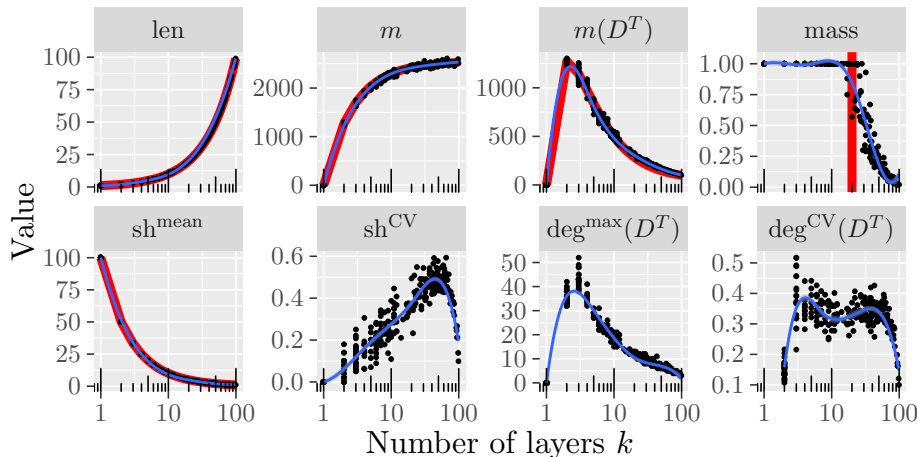
This method always generates DAGs with a length equal to k and mean shape equal to n/k. Moreover, when all layers have the same size n/k, the expected number of edges is $\mathbb{E}(m) = n \left(1 - \frac{1}{k}\right) \left(p \left(\frac{n}{2} - 1\right) + 1\right)$ and the expected number of edges in the transitive reduction is $\mathbb{E}(m(D^T)) \geq p(k-1) \left(\frac{n}{k}\right)^2 + (1-p)n \left(1 - \frac{1}{k}\right)$.

Figure 4 represents the effect of the number of layers k. The numbers of edges in the DAG and its transitive reduction are close to the expected values for the case when all layers have the same size n/k. Finally, the mass is unitary when there are at least two balls in each bin. Since there is initially one ball per bin, this occurs when there is at least one of the $n - k$ additional balls in each of the k bin. Using a bound for the coupon collector problem [18, Proposition 2.4], this occurs with probability greater than 0.5 when $\lceil k \log(2k) \rceil + k < n$, which is the case for $k \leq 20$ when $n = 100$. This is consistent with Figure 4 where the mass becomes non-unitary around this value.

To avoid non-unitary mass, the layer-by-layer method can be adapted to ensure that each layer has two vertices initially. For instance, we can rely on a uniform distribution between two and a maximum value, or on a balls into bins approach with two balls per bin initially.

---

[5] Each boxplot consists of a bold line for the median, a box for the quartiles, whiskers that extend at most to 1.5 times the interquartile range from the box and additional points for outliers.
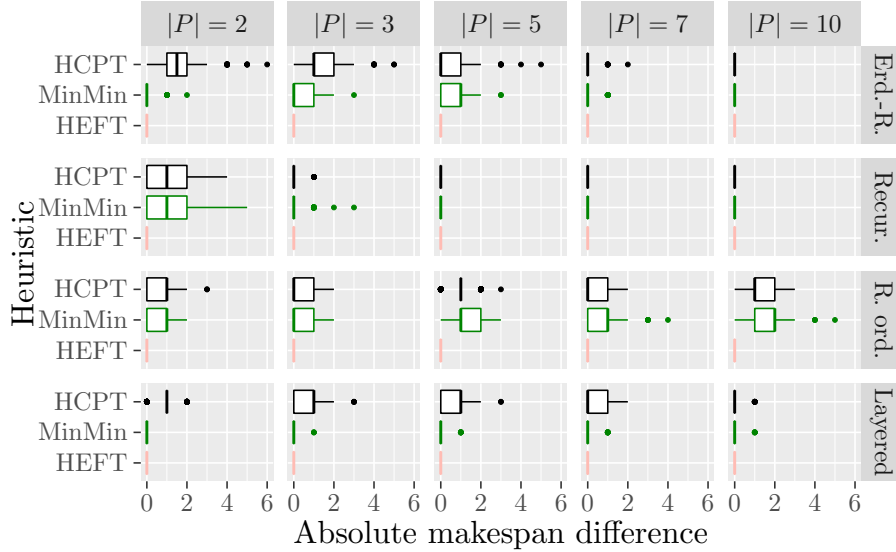
Fig. 4. Properties of 300 DAGs of size $n = 100$ generated by the layer-by-layer algorithm with probability $p = 0.5$ and a number of layers k randomly drawn between 1 and 100 (uniformly on the logarithmic scale). Red lines correspond to formal results.

6 Evaluation on Scheduling Algorithms

Generating random task graphs allows the assessment of existing scheduling algorithms in different contexts. Numerous heuristics have been proposed for the problem denoted $P|p_j = 1, prec|C_{max}$ or generalizations of this problem. Such heuristics rely on different principles. Some simple strategies, like MinMin [14, Algorithm D], execute available tasks on the processors that minimize completion time without considering precedence constraints. In contrast, many heuristics sort tasks by criticality and schedule them with the Earliest Finish Time (EFT) policy. This is the case for both HEFT [24] and HCPT [13]. HEFT first computes the upward rank of each task, which can be seen as a reverse depth, and then consider tasks by decreasing order of their upward ranks. Backfilling is performed following an insertion policy. In contract, HCPT starts by considering any task on a critical path by decreasing order of their depth. The objective is to prioritize the ancestors of such tasks and in particular when their depths are large.

Figure 5 shows the absolute difference between MinMin, HEFT and HCPT for each generator covered in Section 5. Despite guaranteeing an unbiased generation, instances built with the recursive algorithm fail to discriminate heuristics except when there are two processors. Recall that the mean shape is close to 1.5 for such DAGs and few processors are sufficient to obtain a makespan equal to the DAG length (i.e. an optimal schedule). In contrast, instances built with the random orders algorithm lead to different performance for each scheduling heuristics. However, this generator has no uniformity guarantee and its discrete parameter K limits the diversity of generated DAGs. Finally, the last two algorithms fail to highlight a significant difference between MinMin and HEFT even though

Fig. 5. Difference between the makespan obtained with any heuristic and the best value among the three heuristics for each instance. Each boxplot represents the results for 300 DAGs of size $n = 100$ built with the following algorithms: Erdős-Rényi ($p = 0.15$), recursive, random orders ($K = 3$) and layer-by-layer ($p = 0.5$ and $k = 10$). Costs are unitary and $|P|$ represents the number of processors.

the former scheduling heuristic can be expected to be inferior to the latter as it discards the DAG structure.

To support these observations, we analyze below the maximum difference between the makespan obtained with HEFT and the ones obtained with the other two heuristics. Because it lacks any backfilling mechanism, HCPT performs worse than HEFT with an instance composed of the following two elements. First, a chain of length k with $|P| - 1$ additional tasks with predecessor the $(k-2)$th task of the chain and successor the kth task of the chain. The second element is a chain of length $k - 1$. HCPT schedules the first element and then the second one afterward, leading to a makespan of $2k - 1$ whereas the optimal one is k. With our settings, the difference from HEFT with this instance is greater than or equal to 45. Moreover, MinMin also performs worse with specific instances. Consider the ad hoc instances considered in [5] each consisting of one chain of length k and a set of $k(|P| - 1)$ independent tasks. Discarding the information about critical tasks prevents MinMin from prioritizing tasks from the chain. With $n = 100$ tasks and with $|P| \leq 10$, the worst-case absolute difference can be greater than or equal to 9. It is interesting to analyze the properties of these difficult instances for MinMin. Each DAG is characterized by a length equal to len $= \frac{n}{|P|}$ and a number of edges in the transitive reduction $m(D^T) = \text{len} - 1$. Moreover, worst-case DAGs for HCPT are characterized by a large length and width.

These experiments illustrate the need for better generators that control multiple properties while avoiding any generation bias. In particular, they highlight the need for a generator that uniformly samples all existing DAGs with a given size n, number of edges m, $m(D^T)$, length, width, and with a unitary mass.

7   Conclusion

This work contributes in multiple ways to the final objective of uniformly generating random DAGs belonging to a category of instances with desirable characteristics. First, we select eight DAG properties, among which the mass quantifies how much an instance can be decomposed into smaller ones. Second, existing random generators are formally analyzed and empirically assessed with respect to the selected properties. Establishing the sub-exponential generic time complexity for decomposable scheduling problems with uniform DAGs constitutes the most noteworthy result of this paper. Last, we study how the generators impact scheduling heuristics with unitary costs.

The relevance of many other properties such as the number of critical tasks need to be investigated further. Moreover, extending current results to instances with communications represents a challenging perspective. Finally, adapting properties to instances with non-unitary costs is left to future work.

Data Availability Statement

The datasets generated and/or analyzed during the current study are available in the Figshare repository: https://doi.org/10.6084/m9.figshare.8397623.

References

1. Adam, T.L., Chandy, K.M., Dickson, J.: A comparison of list schedules for parallel processing systems. Communications of the ACM 17(12), 685–690 (1974)
2. Aho, A.V., Garey, M.R., Ullman, J.D.: The transitive reduction of a directed graph. SIAM Journal on Computing 1(2), 131–137 (1972)
3. Campos, P., Dahir, N., Bonney, C., Trefzer, M., Tyrrell, A., Tempesti, G.: Xl-stage: A cross-layer scalable tool for graph generation, evaluation and implementation. In: Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), 2016 International Conference on. pp. 354–359. IEEE (2016)
4. Canon, L.C., El Sayah, M., Héam, P.C.: A comparison of random task graph generation methods for scheduling problems. arXiv preprint arXiv:1902.05808 (2019)
5. Canon, L.C., Marchal, L., Simon, B., Vivien, F.: Online scheduling of task graphs on hybrid platforms. In: Euro-Par. pp. 192–204. Springer (2018)
6. Cordeiro, D., Mounié, G., Perarnau, S., Trystram, D., Vincent, J.M., Wagner, F.: Random graph generation for scheduling simulations. In: ICST. p. 60 (2010)
7. Dick, R.P., Rhodes, D.L., Wolf, W.: TGFF: task graphs for free. In: International workshop on Hardware/software codesign. pp. 97–101. IEEE (1998)

8. Dutot, P.F., N'takpé, T., Suter, F., Casanova, H.: Scheduling parallel task graphs on (almost) homogeneous multicluster platforms. IEEE TPDS 20(7), 940–952 (2009)
9. Erdős, P., Rényi, A.: On random graphs I. Publ. Math. Debrecen 6, 290–297 (1959)
10. Garey, M., Johnson, D.: Strong NP-completeness results: motivation, examples, and implications. J. Assoc. Comput. Mach. 25(3), 499–508 (1978)
11. Graham, R.L., Lawler, E.L., Lenstra, J.K., Kan, A.H.G.R.: Optimization and approximation in deterministic sequencing and scheduling: a survey. Annals of Discrete Mathematics 5, 287–326 (1979)
12. Gupta, I., Choudhary, A., Jana, P.K.: Generation and proliferation of random directed acyclic graphs for workflow scheduling problem. In: International Conference on Computer and Communication Technology. pp. 123–127. ACM (2017)
13. Hagras, T., Janecek, J.: A simple scheduling heuristic for heterogeneous computing environments. In: ISPDC. p. 104. IEEE (2003)
14. Ibarra, O.H., Kim, C.E.: Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors. Journal of the ACM 24(2), 280–289 (Apr 1977)
15. Jain, R.: The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling. John Wiley (1990)
16. Juve, G., Chervenak, A., Deelman, E., Bharathi, S., Mehta, G., Vahi, K.: Characterizing and profiling scientific workflows. Future Generation Computer Systems 29(3), 682–692 (2013)
17. Leung, J.Y.: Handbook of scheduling: algorithms, models, and performance analysis. CRC Press (2004)
18. Levin, D.A., Peres, Y.: Markov chains and mixing times, vol. 107. American Mathematical Society (2017)
19. Liskovets, V.: On the number of maximal vertices of a random acyclic digraph. Theory Probab. Appl. 20(2), 401–409 (1975)
20. Mantel, W.: Problem 28. Wiskundige Opgaven 10(60-61), 320 (1907)
21. Melançon, G., Dutour, I., Bousquet-Mélou, M.: Random generation of directed acyclic graphs. Electronic Notes in Discrete Mathematics 10, 202–207 (2001)
22. Robinson, R.W.: Counting labeled acyclic digraphs. In: Harray, F. (ed.) New Directions in the Theory of Graphs. pp. 239–273. Academic Press, New York (1973)
23. Tobita, T., Kasahara, H.: A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. Journal of Scheduling 5(5), 379–394 (2002)
24. Topcuoglu, H., Hariri, S., Wu, M.y.: Performance-effective and low-complexity task scheduling for heterogeneous computing. IEEE TPDS 13(3), 260–274 (2002)
25. Ullman, J.: NP-complete scheduling problems. J. Comput. System Sci. 10, 384–393 (1975)
26. Winkler, P.: Random orders. Order 1(4), 317–331 (1985)