Random graph generation for scheduling simulations

Daniel Cordeiro, Grégory Mounié, Swann Pérarnau, Denis Trystram,
Jean-Marc Vincent, Frédéric Wagner

Random graph generation for scheduling simulations

Daniel Cordeiro, Grégory Mounié, Swann Perarnau, Denis Trystram,
Jean-Marc Vincent, Frédéric Wagner
LIG, Grenoble University
51, avenue Jean Kuntzmann
38330 Montbonnot Saint Martin, France
{cordeiro, mounie, perarnau, trystram, vincent, wagner}@imag.fr

ABSTRACT

In parallel and distributed systems, validation of scheduling heuristics is usually done by simulation on randomly generated synthetic workloads, typically represented by task graphs. Since there is no single generation method that models all possible workloads for scheduling problems, researchers often re-implement the classical generation algorithms or even implement *ad hoc* ones. A bad choice of generation method can mislead the validation of the algorithm due to biases it can induce. Moreover, different implementations of the same randomized generation method may produce slightly different graphs. These problems can harm the experimental comparison of scheduling algorithms. In order to provide a comparison basis we propose GGen – a unified and standard implementation of classical task graph generation methods used in the scheduling domain. We also provide an in-depth analysis of each generation method, emphasizing important graph properties that may influence scheduling algorithms.

Categories and Subject Descriptors

I.6.3 [**Simulation and Modeling**]: Applications

General Terms

Performance, experimentation

Keywords

Scheduling, simulation, algorithm validation, random graphs generation

1. INTRODUCTION

Scheduling in parallel and distributed systems is a classic Computer Science area. Although well studied both in theoretical and experimental aspects, scheduling is still a hot research topic.

Highly distributed computational environments like the ones provided by grid computing technologies interlink thousands of computers that can, together, solve complex problems. To choose how to spread the work that must be done among such large number of machines is, however, a very difficult task. Optimally choosing how to schedule a set of sequential jobs with different sizes between two processors in order to finish all jobs as soon as possible[1] is a NP-Hard problem [**?**] unless P = NP.

If performance is a critical issue, the developer must perform a case-by-case analysis to tune-up the used scheduling algorithm. This usually means the theoretical study of the problem and a new scheduling algorithm that must be validated. Typically, this validation is done through the simulation of the new algorithm against a synthetic workload, created either using execution logs collected from real users or, more often, by random generation.

In this article we argue that random generation methods can create biased results that can mislead the analysis of scheduling algorithms. The developer must know what are the properties of the generated workload and how they affect the performance of the classical scheduling algorithms. In this work we present some classic generation methods and how the generated graphs can affect the validation of scheduling algorithms.

The remaining of the article is organized as follows. An informal introduction to the scheduling problem is presented in Section 2. Section 3 describes the classical methods used by researchers to generate random task graphs for validation of scheduling algorithms. Section 4 presents GGen, a set of tools intended to help developers to better understand the typical workload of their applications; GGen generates random graphs of tasks that better represents this workload. An experimental analysis of the classical generation methods is discussed in Section 5 and a case-study with real scheduling algorithms is shown in Section 6. Finally, Section 7 presents some conclusions and future works.

2. SCHEDULING TASK GRAPHS

Scheduling a set of tasks among the (usually scarce) available computational resources (machines, processors, etc.) with the objective of optimize one or more performance metric is a fundamental problem in parallel and distributed systems [**?**].

An instance of the problem is composed by a directed acyclic graph $G = (V, E)$, by an integer m representing the

[1]In Graham's notation [**?**], this scheduling problem is known as $2 \mid p_j \mid C_{\max}$.

available resources and by the running time p_{ij} of job i on resource j, where $i \in V$ and $j \in [1, m]$. The edges E represent the precedence constraints. If $(i, j) \in E$ then job j can not start before the completion of job i. A given scheduling is a function $\sigma : V \to \mathbb{N} \times [1, m]$ that gives the start time and the resource allocated for each job. We denote the completion time of job i in scheduling σ as $C_i^\sigma = \sigma(i) + p_i$. Through the remaining of the article, we use the acronym *DAG* or the word *graph* to refer to *directed acyclic graph*, unless stated otherwise.

Makespan (the completion time of the last job to finish in schedule σ, denoted as $C_{\max} = \max\{C_i^\sigma\}$), the average completion time ($\frac{\sum C_i^\sigma}{|V|}$), the stretch ($S_j = \frac{C_j - r_j}{p_j}$, where r_i is the release date of the job), etc. are some examples of the performance metrics whose values researchers want to minimize. The problem of obtaining optimal scheduling regarding these objectives is know to be NP-Complete [**?**].

Representing jobs and their precedences constraints as a DAG is very convenient. It gives some interesting information about the problem like the degree of parallelism that application can attain, the minimum amount of time required by the application (the critical path of the graph), etc. These properties have been used on the development of heuristics to solve scheduling problems.

The validation of a new scheduling algorithm involves a test suite composed by representative workloads. Some researchers test their algorithms against some well-known static workload, like the execution logs of the Parallel Workloads Archive [**?**], maintained by Dror Feitelson. These static workloads are used as a comparison basis for the analysis of different works.

But it is also important to validate a new scheduling algorithm against some random workloads. The use of randomly generated data is a very important step in the design of scheduling algorithms because:

- it may help finding a counter-example for the algorithm. Even if the algorithm has been theoretically proved to be correct, random input data may help to find bugs in the implementation or help to identify performance bottlenecks;
- it helps to evaluate the performance of the algorithm in contexts not yet theoretically analyzed. It may help the developer to predict how the algorithm will perform in "real" conditions if used by users with different utilization profiles.

Ideally a new scheduling algorithm under evaluation should be tested against all possible use-cases. This would require a random graph generator able to generate all possible graph structures with the same probability. The notion of "classes of structures" is formally described by the notion of *graph isomorphism*.

Formally, two graphs G and H are isomorphic if there is a mapping $\varphi : V(G) \to V(H)$ such that $(u, v) \in E(V) \iff (\varphi(u), \varphi(v)) \in E(H), \forall u, v \in V(G)$. A scheduling algorithm probably will produce the same result for two isomorphic graphs, even if they are very different from each other[2].

[2]For instance, isomorphic task graphs with vertices and edges properties (task duration, communication costs, etc.) generated with a uniform distribution will generally induce similar performances on scheduling algorithms.

The large number of unlabeled acyclic digraphs makes impractical (even for a small number of vertices) any generation method based on the enumeration of the complete set of possible graphs. Table 2 shows the number of non-isomorphic DAGs with a number of vertices up to 10 (reproduced from sequence A003087 in [**?**]). To the best of our knowledge, there is no method that uniformly generates non-isomorphic graphs. Interestingly, the graph isomorphism problem is suspected to be neither in P nor NP-complete classes of complexity [**?**].

The unavailability of a general method for creating truly random non-isomorphic task graphs implies that scheduling algorithm designers must choose which generation methods are more appropriate for the workload expected for their algorithm. This is a complex choice that must be carefully analyzed case-by-case.

3. GENERATION METHODS

In order to help scheduling algorithm designers to choose which is the more appropriate method for their needs, we present in this section the classical task generation algorithms used by researchers in the validation of their scheduling algorithms. An experimental study about the particularities of the graphs generated by each one of the methods described in this section is presented in Section 5.

For clarity, on the pseudo-algorithms presented in this section we use the auxiliary function RANDOM(), that returns a real number uniformly distributed on the interval $[0, 1[$.

3.1 The Erdős-Rényi methods

Paul Erdős and Alfréd Rényi defined in 1959 [**?**] two straightforward graph generation methods that are among the most popular methods. These two methods are referenced in the literature as the $\mathsf{G}(\mathsf{n}, \mathsf{p})$ and the $\mathsf{G}(\mathsf{n}, \mathsf{M})$ methods.

3.1.1 The $G(n, p)$ method

This is the most intuitive and most widely utilized graph generation method.

DEFINITION 1. *For a given n number of vertices, the $G(n, p)$ method generates a graph where each element of the $\binom{n}{2}$ possible edges is present with independent probability p.*

Erdős and Rényi first defined this method for non-oriented graphs, but it is easy to adapt this for DAG generation:

Algorithm 1 $\mathsf{G}(\mathsf{n}, \mathsf{p})$ method.

Require: $n \in \mathbb{N}, p \in \mathbb{R}$.
Ensure: a graph with n nodes.

```
Let M be an adjacency matrix n × n initialized as the zero matrix.
for all i = 1 to n do
  for all j = 1 to i do
    if Random() < p then
      M[i][j] = 1
    else
      M[i][j] = 0
return the graph represented by M.
```

Erdős and Rényi proved [**?**] several properties about the graphs generated by the $\mathsf{G}(\mathsf{n}, \mathsf{p})$ method. From these properties, we can cite some that may be particularly interesting for scheduling algorithm designers:

Number of vertices	1	2	3	4	5	6	7	8	9	10
Number of DAGs	1	2	6	31	302	5984	243668	20286025	3424938010	1165948612902

Table 1: Number of acyclic digraphs with unlabeled vertices.

- For sufficiently large values of n the number of edges in the generated graphs tends to $p\binom{n}{2}$;
- If np tends to a constant bigger than 1, then with high probability there exists a weakly connected subgraph with the majority of the nodes and no other connected component exists with more than $O(\log n)$ nodes;
- If $p > \frac{(1+\epsilon)\ln n}{n}$, then with high probability the generated graph will not have any isolated vertices.

3.1.2 The $G(n, M)$ method

Although less utilized, this model can be considered as the most appropriate method for generating random graphs with a fixed number of edges.

DEFINITION 2. *For a given number of nodes n and a given number of edges M, the $G(n, M)$ method is defined as the method that constructs the graph by choosing uniformly M edges from the list of edges of the complete DAG on n vertices.*

This is equivalent to say that the method chooses uniformly a graph from the list of all possible DAGs with M edges and n nodes.

3.2 Layer-by-Layer

This method was first proposed by Tobita and Kasahara [**?**]. It was designed specifically for the validation of scheduling heuristics and is based in the concept that they called "layers", i.e., an independent set of the graph with the additional property that if there is an edge from layer a to layer b, then there is no path from a vertex in b to a vertex in a. Edges are created with probability p exactly like in Erdős' G(n, p) method.

Algorithm 2 Layer-by-Layer method.

Require: $n, k, p \in \mathbb{N}$.
  Distribute n vertices between k different sets enumerated as $L_1, \ldots, L_k$.
  Let $layer(v)$ be the layer assigned to vertex v.
  Let M be an adjacency matrix $n \times n$ initialized as the zero matrix.
  for all i = 1 to n **do**
    for all j = 1 to n **do**
      if $layer(j) > layer(i)$ **then**
        if Random() < p **then**
          M[i][j] = 1
        else
          M[i][j] = 0
  return a random DAG with k layers and n nodes.

Although very simple, this method is very useful in practice because we can limit the size of the critical path only by limiting the value k.

3.3 Fan-in / Fan-out

Dick et al. [**?**] introduced the Fan-in/Fan-out method. Fan-in/Fan-out constructs each graph incrementally, allowing more control over properties like in-degree/out-degree of the vertices or even over the general structure of the graph.

We describe here the method slightly adapted to use the number of vertices as input instead of generating it automatically (like described in [**?**]).

Algorithm 3 Fan-in/Fan-out method.

Require: $n, id, od \in \mathbb{N}$.
Ensure: a graph with at least n nodes, where each node has an out-degree $\leq od$ and an in-degree $\leq id$.
  Initialize $G = (V, E)$, with $E = \emptyset$ and $V = \emptyset$.
  Add an vertex to G.
  while $|V| \leq n$ **do**
    if Random() < 0.5 **then** {Fan-out phase}
      Find the vertex v with the biggest difference between its out-degree and od. Let m_o be this difference.
      Add a random number of vertices between 1 and m_o to V and add edges from v to these new vertices.
    else {Fan-in phase}
      Find the set S of all vertices that have out-degree $< od$.
      Compute a subset T of S of size at most id.
      Add a new vertex v and add new edges (v, t) for all $t \in T$.

We can think about the *Fan-in* and *Fan-out* phases as an operation to expand/contract the graph. In some sense each phase tries to emulate the scatter/gather phases of parallel applications.

3.4 Random Orders

The Random Orders method utilizes properties from the branch of mathematics called Order Theory to analyse and generate random graphs. Proposed by Winkler [**?**], this method generates random partially ordered sets that can be used to generate task graphs. In fact, a DAG is a partial order where the vertices are ordered by reachability. The idea of the algorithm 4 is to create a partial order by intersecting several total orders.

Algorithm 4 Random Orders method.

Require: $n, k \in \mathbb{N}$.
Ensure: a graph with n vertices obtained from an order of dimension at most equal to k.
  Generate k total orders (random permutations of the vertices).
  Intersect the k generated orders to obtain a partial order.
  Apply a transitive reduction to transform the partial order obtained in a DAG.

It is hard to analyse the influence of the parameter k over the structure of the generated graphs. We can note, however, that if we choose $k = 1$, then the generated graphs will

represent task chains, i.e., in this case the partial order will also be a total order.

3.5 Related Tools

Although this work was motivated by scheduling simulations, the use of random graph generators also interests researchers from other Computer Science research areas. In this section we list a few tools that are widely utilized for graph manipulation.

Networking researchers utilize graphs extensively for the analysis and simulation of distributed algorithms and protocols. In this context, NetworkX [**?**] provides a Python implementation of some network topology generators and a large list of utilities and algorithms that help researchers to extract interesting graph properties from their network topologies (connectivity, basic isomorphism, etc.). For the same finality, the igraph [**?**] library was conceived to be easily embeddable into other high level languages. It is implemented in C++ and in GNU R language, but is distributed also as a Ruby and Python packages. Both libraries provide a very interesting set of utilities for graph analysis, but they do not directly help developers to create a consistent and unbiased validation scheme for their algorithms.

The Stanford GraphBase (SGB) [**?**] and the Boost Graph Library [**?**] are implementations of graph data structures and algorithms to analyse graphs written in C and in C++ respectively. The Boost Graph Library implements also some random generation methods, but the main goal of both libraries is to provide an abstract data type independent of its utilisation. GGen (that will be described in more details in Section 4) is implemented on top of the Boost Graph Library.

The Task Graphs for Free (TGFF) [**?**] is a tool designed specifically for scheduling simulation. It offers an implementation of the method described in Section 3.3. A more recent version[3] provides an option to generate also series-parallel DAGs. TGFF provides a built-in model to generate random communication costs and deadlines. Unfortunately, there is no way to control the random distribution of the attributes generated by TGFF.

With the exception of the SGB, all the cited tools (including GGen) have some kind of support (either native of through the use of conversion scripts) for the DOT format [**?**], which makes these tools interoperable and complementary to each other.

4. GGEN: A GRAPH GENERATION TOOL

Suppose that a developer wants to validate a scheduling algorithm that depends on a well-know graph property such as the critical path. If the algorithm performance depends on this property, the chosen generation method must randomly generate graphs with a large variety of values for the critical path. If the method has a tendency of generate graphs with limited values of critical path, the validation of the scheduling algorithm may be affected.

If the developer is not aware of the biases produced by the utilized generation method, an algorithm validated by simulation may perform poorly with a more heterogeneous workload. Detecting which graph properties impact the performance of the algorithm and creating an appropriate validation suite are very hard.

Motivated by the difficulty in how to characterize and generate a truly representative synthetic workload (for a good introduction on the problem of workload modeling, please refer to [**?**]), we started the development of *GGen*, a random graph generator and graph analyzer. GGen's generation and analysis algorithms were chosen to assist scheduling algorithm designers to create a proper and representative validation suite.

GGen is implemented in C++ on top of the Boost Graph Library [**?**] and the GNU Scientific Library [**?**]. GGen inherits the efficient graph data structures implemented by Boost and its extensibility, allowing the definition of new graph internal representations if necessary. GGen was designed to be easily extensible – it is straightforward to include a new graph generation or graph property analyser. The source code is freely available at GGen's website (`http://ggen.ligforge.imag.fr/`) and it is released under the CeCILL[4] free software license.

The tool provides three distinct interfaces for the user: a graph generator, a graph analyzer and an interface that allows the developer to add or remove independent properties for the vertices and edges of the graph.

The graph generator is the most important interface to GGen. It allows the generation of new graphs using some generation methods found in scheduling literature. Up to now, we have implemented the random graph generation methods described in Section 3: Erdős' $\mathsf{G}(\mathsf{n},\mathsf{p})$ and $\mathsf{G}(\mathsf{n},\mathsf{M})$, Layer-by-Layer, Fan-in/Fan-out and Random Orders. We believe that the graphs generated by these methods present the most interesting graph properties for algorithm designers interested in scheduling.

The graph analyzer allows the developer to analyze a generated graph and collect some properties about it. The list of properties that GGen can analyze includes, but are not limited to, the Minimum Spanning Tree, the Max Independent Set, the in/out-degree of the vertices, etc.

GGen can also randomly generate vertices and edges properties, i.e. a key-value pair associated to each graph component (vertex and/or edge) that represents some application semantic. This values can mean, for instance, the size of the jobs associated to each vertex or the communications cost involved on each job dependency associated to each edge. The tool allows a fine control on the random distribution used to generate these values.

Decoupling the generation of these properties of the generation of the graph *per se* allows a better control of the experiments with less overhead of the graph generation. It makes easy to experiment with different random distributions for these values, while keeping the same kind of graph structures.

To make GGen interoperable with most of the available graph analysis tools, we have chosen the Graphviz's DOT language [**?**] as the default graph representation format for GGen.

The DOT language is a widely-adopted plain text format that provides syntax for describing graphs, nodes and edges and the properties associated with the graph components. All graphs generated by GGen can take advantage of the set of tools distributed with Graphviz such as tools to display graphs using different heuristics (`dotty, neato`,

[3]Publicly available to download at: `http://ziyang.eecs.umich.edu/~dickrp/tgff/`

[4]CeCILL is a GPL compatible license. See `http://www.cecill.info/` for more information.

etc.) or tools to modify graphs in an interactive (`dotty`) or script/*awk-like* way (`gvpr`).

5. METHODS ANALYSIS

The previously discussed methods produce random graphs that are structurally different. Unfortunately, few papers presenting those methods try to analyse the nature of the generated graphs. Since scheduling performance is tightly related to the characteristics of the initial task graph, we consider this analysis essential.

We conducted experiments that measure the impact of variations in the parameters of each generation method described in Section 3. For each parameter and for each method, we generated a thousand random graphs with exactly one hundred nodes (except for Fan-in/Fan-out, that generated around 102 ± 2 nodes). We measured some of the graph properties that are most likely to influence scheduling algorithms.

All these measures were computed using GGen's graph analyzer. Each measurement is presented with a 95% confidence interval. The confidence interval is indicated in all figures, but due to its small size it is only visible in Figure 13.

5.1 Longest Path

The longest path in a DAG is defined as the path with maximal length (i.e., the path with the biggest number of nodes) in a given graph. This is one of the most studied graph properties. In scheduling problems, the longest path length gives a lower bound to the value of the minimum completion time required to execute the application.

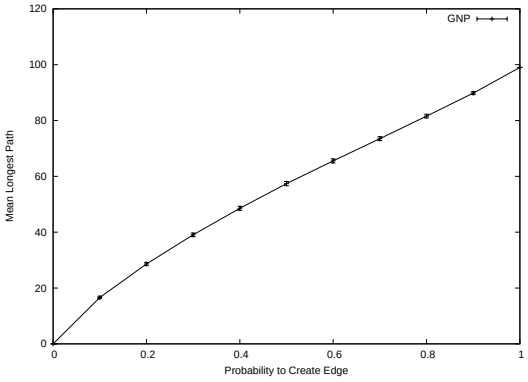

Figure 1: Mean longest path length of graphs generated using $\mathsf{G}(n, p)$ for different probabilities.

Figure 1 shows how the probability of creating an edge impacts the average length of the longest path in $\mathsf{G}(n, p)$ method. As expected, increasing the probability implies a bigger mean longest path length. Taking $p = 0$ and $p = 1$ we get, respectively, the graph were all nodes are disconnected and the complete graph where the longest path contains all vertices.

The average length of the longest path increases almost linearly with the number of allowed edges in the $\mathsf{G}(n, M)$ method. Figure 2 illustrates this behaviour for M varying from 100 to 4500 (which is approximately the maximal number of edges possible in a DAG with 100 vertices).

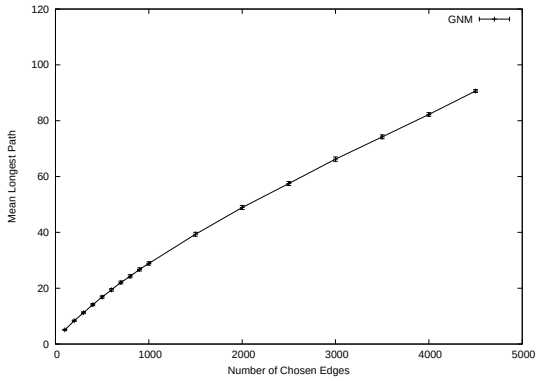

Figure 2: Mean longest path length of graphs generated using $\mathsf{G}(n, M)$ for different number of edges.

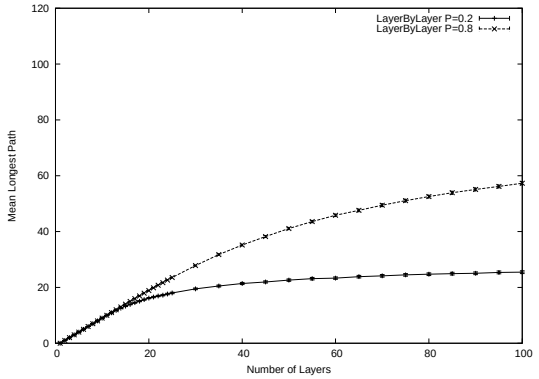

Figure 3: Mean longest path length of graphs generated using Layer-by-Layer for different numbers of layers and probabilities (without the confidence intervals to increase readability).

Graphs generated by the Layer-by-Layer method are less sensitive to different probabilities when the number of layers is small. In this case, even for small probabilities, the method have a tendency to produce graphs with the biggest longest path possible. In Layer-by-Layer this value is equal to the chosen number of layers.

Figure 3 presents the results taking 0.2 and 0.8 as probabilities of connecting two nodes on different layers. The additional constraint imposed by Layer-by-Layer (that forbids edges between vertices in a same layer) makes the average length of the longest path smaller than in the previous methods. It is interesting to note that the average longest path length does not always correspond to the chosen number of layers. The method distributes all vertices among the layers but does not guarantee that each layer will have at least one vertex. Moreover, even for high edge creation probabilities there is no guarantee of the existence of a path that traverses all layers.

The Fan-in/Fan-out method – described in Section 3.3 – allows the generation of graphs with a fixed maximum in and

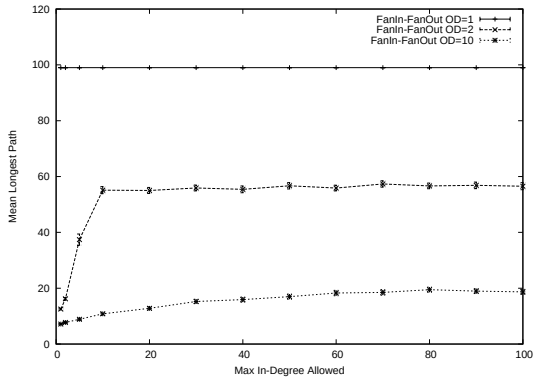

Figure 4: Mean longest path length of graphs generated using Fan-in/Fan-out for different values of maximum allowed out-degrees and in-degrees.

out-degrees for all vertices. These constraints impose some limits on the length of the longest path. Figure 4 shows the mean length of the longest path for graphs generated by the Fan-in/Fan-out method. Three curves are shown in this figure. The curve on top shows that when the maximum allowed out-degree for the vertices is exactly one, then the method only generates chains, i.e. the length of the longest path is exactly equal to the number of vertices. The curve on the middle shows that for a fixed out-degree of 2, the length of the longest path does not grow linearly with the maximal in-degree. The Fan-in phase allows the clustering of more vertices, resulting in bigger values for the mean length of the longest path. Finally, the bottom curve shows that a large value for the maximum allowed out-degree imposes the generation of compact graphs with short longest paths.

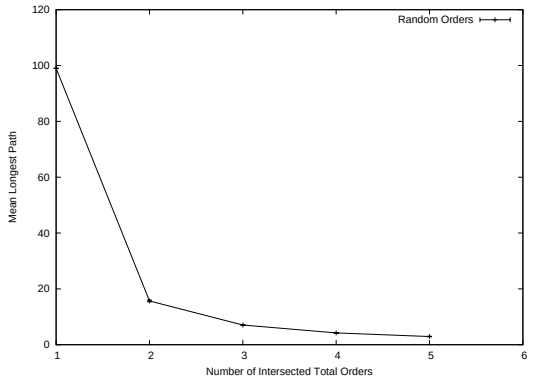

Figure 5: Mean longest path length of graphs generated using Random Orders for different number of intersected total orders.

The Random Orders method only accepts as parameter the number of vertices to be generated and the number of total orders to be intersected. In Figure 5 we can see, as expected, that for a bigger number of intersected total orders we have graphs with less number of edges and, as consequence, smaller values for the longest path length.

5.2 Distribution of the Out-degree

In scheduling, the notion of out-degree of a given vertex v in a DAG (i.e., the number of directed edges in this graph that start in vertex v) can be used to estimate the number of processors that can be used concurrently to execute some task graph.

Using the same graphs generated for the analysis done in the previous section, we analyse in this section the out-degree of their vertices and show how this number is influenced by the choice of parameters for each generation method implemented in GGen.

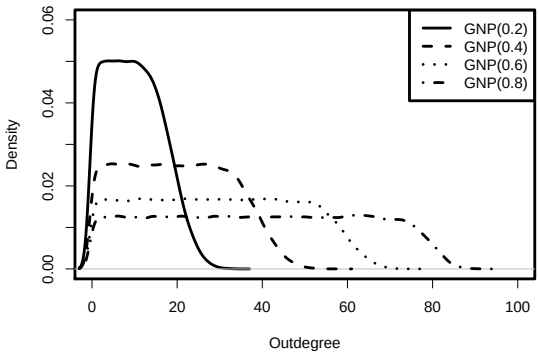

Figure 6: Distribution of out-degrees of graphs generated using $\mathsf{G}(\mathsf{n},\mathsf{p})$ for different probabilities.

Figures 6 and 7 give the density curves of the mean out-degree for the $\mathsf{G}(\mathsf{n},\mathsf{p})$ and $\mathsf{G}(\mathsf{n},\mathsf{M})$ methods, respectively. As expected, an increasing probability implies in increasing values of the mean out-degree. These figures also show that for the same number of edges, the out-degree distribution is the same for both methods.

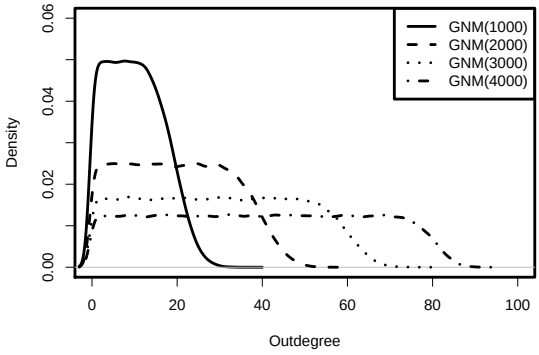

Figure 7: Distribution of out-degrees of graphs generated using $\mathsf{G}(\mathsf{n},\mathsf{M})$ for different number of edges.

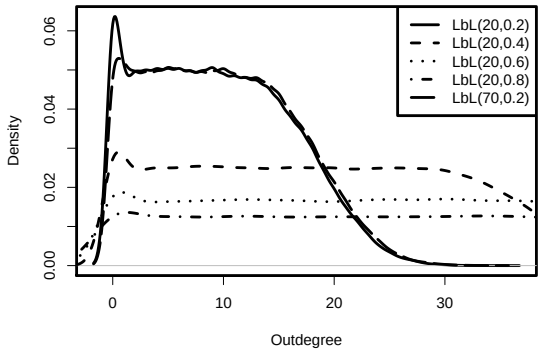

Figure 8: Distribution of out-degrees of graphs generated using Layer-by-Layer for different numbers of layers and edge probabilities.

The distribution of the out-degree for the Layer-by-Layer method is depicted in Figure 8. We show the mean out-degree for different numbers of layers and probabilities. The figure clearly shows that the number of layers has practically no influence on the mean out-degree. The variation in the mean out-degree is due to the variation in the probability of creating new edges. For instance, the curves LbL(20,0.2) and LbL(70,0.2) almost overlap each other. They were generated from graphs with different numbers of layers (20 and 70), but the same probability of creating a new edge (0.2). This behaviour is also observed for higher probabilities of creating new edges.

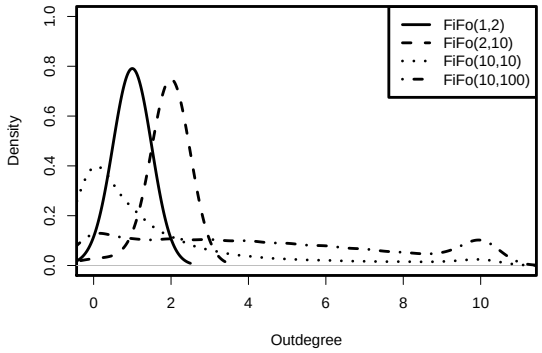

Figure 9: Distribution of out-degrees of graphs generated using Fan-in/Fan-out for different values of maximum allowed out-degrees and in-degrees.

Figure 9 shows the out-degree distribution for Fan-in/Fan-out method. We have two kinds of curves depicted in this figure: curves generated with a low maximal allowed out-degree and curves generated with higher allowed values. For the curves from the first type – FiFo(1,2) and FiFo(2,10) (the first number denotes the out-degree and the second the in-degree) – we can see that almost all nodes get saturated with the maximum number of edges possible. For the curves generated for FiFo(10,10) and FiFo(10,100), we can see that the out-degree distribution is quite uniform. In both cases, the average out-degree is quite low. This happens because the number of edges on the graphs generated by this method is low if compared with the others, as we will see in Section 5.3.

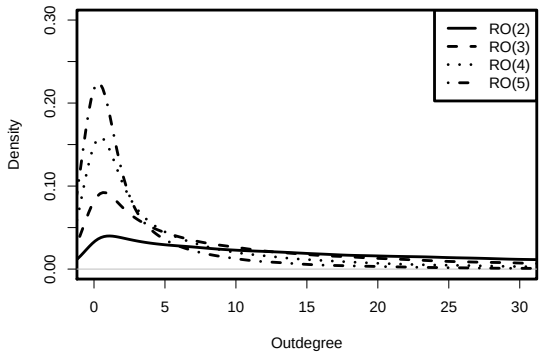

Figure 10: Distribution of out-degrees of graphs generated using Random Orders for different number of intersected total orders.

The out-degree distribution of Random Orders is depicted in Figure 10. The same reasoning used for the length of the longest path can be applied here: if we increase the number of total orders to be intersected, we will generate graphs that have less edges and lower mean out-degree.

5.3 Number of edges

The described properties of the graph generated by each method presented a direct relationship with the number of edges of the generated graphs. In scheduling, edges in a task graph represent the precedence constraints of each task that composes the application. In this section we analyse how each generation method behaves regarding the number of edges. The G(n, M) method is excluded from this analysis since the number of edges of its graphs is specified as an input parameter.

The G(n, p) method has only one parameter: the probability of creating new edges. The final number of edges is directly proportional to the chosen probability, as Figure 11 shows.

Figure 12 shows how the number of edges of the graphs generated by Layer-by-Layer method is impacted by different numbers of layers. Each curve shows the variation of the number of layers for a fixed probability. The number of edges stabilizes very quickly, which means that bigger values for the number of layers have little influence on the mean number of edges.

Fan-in/Fan-out produces a smaller number of edges in average, as shown in Figure 13. For small values of maximum allowed out-degree, varying the maximum allowed in-degree does not change the number of edges. For an out-degree of 10, we can see that the produced graphs still present a low number of edges in average.

Finally, Figure 14 clearly shows how the number of intersected total orders can influence the number of edges in

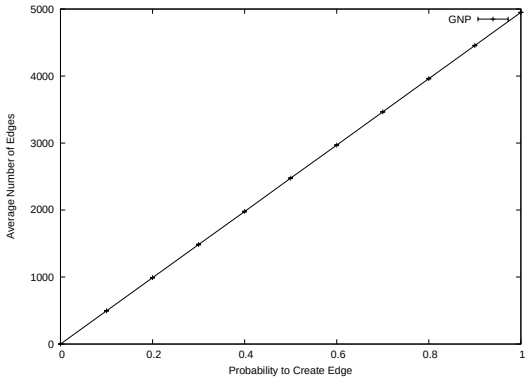

Figure 11: Average number of edges of graphs generated using $G(n, p)$ for different probabilities.

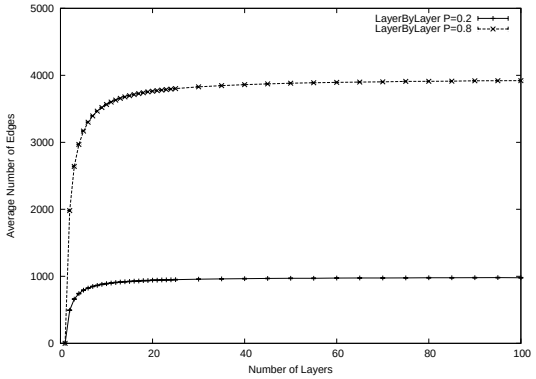

Figure 12: Average number of edges of graphs generated using Layer-by-Layer for different number of layers and probabilities.

graphs generated with Random Orders.

5.4 Summary

The experiments shown in the previous sections made clear that each method generates graphs that are very different from each other. It is important to understand the particularities of each method in order to create a truly representative workload for the validation of a scheduling algorithm.

Erdős' $G(n, p)$ and $G(n, M)$ methods present very similar graph properties, even if they take such different parameters (probability of edges creation and total number of edges, respectively). These are the most general methods and they can be used to generate any possible DAG. They are important for the validation of algorithms against unpredicted workloads.

The other methods have additional constraints that give some structural properties to the generated graphs. These constraints make each method generate a very particular set of graphs, and this can be useful for the study of algorithms developed for a specific workload.

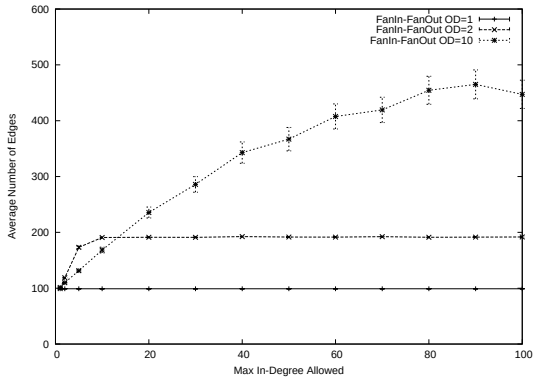

Figure 13: Average number of edges of graphs generated using Fan-in/Fan-out for different values of maximum allowed out-degrees and in-degrees.

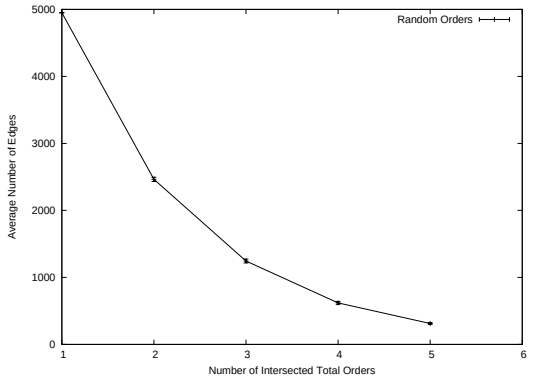

Figure 14: Average number of edges of graphs generated using Random Orders for different number of intersected total orders.

The analysis of Layer-by-Layer showed that there exists a threshold for the number of layers. After this threshold, the only parameter that really changes the properties of the generated graphs is the probability of connecting vertices. We suspect that this threshold depends on the total number of vertices. The analysis also showed that the average length of the longest path not necessarily corresponds to the number of layers chosen as parameter even for high probabilities.

Fan-in/Fan-out method gives the impression that the user have a tight control over the out-degree and the in-degree of the generated graphs. On one side, when the chosen out-degree is very small, the resulting graphs are close to chains. On the other side, the analysis shows that the interaction of both parameters is hard to analyse and the resulting graphs are counter-intuitive.

The task graphs generated by Random Orders model applications composed by jobs that are independent from each other. We can note a sharp decrease on the length of longest path from values of intersected orders bigger than 2.

6. SIMULATING SCHEDULING ALGORITHMS: A CASE STUDY

In this section, we present an analysis of different scheduling algorithms made with the tools provided by GGen. We analyse how random workloads generated using GGen's graph generators impact the performance obtained by each scheduling algorithm.

The scheduling algorithms used in this case study are examples of a well-known class of algorithms called *List Scheduling* algorithms.

6.1 List Scheduling

List Scheduling algorithms are probably the most studied class of scheduling algorithms. They were first studied by Graham in his seminal paper called "Bounds on multiprocessor timing anomalies" [**?**] in 1969, who proved that any List Scheduling algorithm is a $(2 - 1/m)$-approximation for the optimal possible scheduling.

A List Scheduling algorithm works as follows:

1. Build a priority list of all tasks in the graph according to some metric (each algorithm has its own).
2. At each step of the scheduling:
   (a) Greedily choose from the list a task with the highest priority that respects the precedence constraints;
   (b) Assign the task to an available resource (a processor, for instance).

List Scheduling algorithms differ from each other on the strategy used to build their priority list. The four algorithms used in this case study – BottomLevel, OutDegree, MinDegree, and Random – use the following strategies:

- **BottomLevel**: the priority of a task is determined by the length of its longest path to a sink (task without children);
- **OutDegree**: tasks are sorted ascending by their number of children;
- **MinDegree**: tasks are sorted descending by their number of children;
- **Random**: the tasks are randomly chosen with equal probability.

6.2 Simulation

We simulated the four previous algorithms using different task graphs generated with GGen. Using the notation introduced by Graham [**?**], we used these algorithms to solve the NP-hard problem known as $P \mid p_i = 1; prec \mid C_{\max}$. In other words, we want to schedule a set of tasks with unit size and arbitrary precedence constraints through a set of parallel identical machines in order to minimize the completion time of the last task to finish (makespan).

Each algorithm was executed against graphs produced by each generation method available in GGen. For each method analysed, 1,000 graphs were generated. For the simulation with the Random algorithm, each graph was scheduled 20 times. More than 1,500,000 scheduling simulations were done in this experiment.

6.3 Analysis

Table 2 presents the average makespan obtained by our simulations. We can achieve a speed-up of about 3.5 for all the four algorithms being tested only changing the generation method being used to create the synthetic workload.

The difference on the results can be explained with a theoretical analysis of the problem. The performance of algorithms for the problem $P \mid p_j; prec \mid C_{\max}$ heavily depends on the critical path of the input task graphs. When the number of available processors are sufficiently high, the optimal makespan is equal to the sum of the sizes of the jobs that are in the critical path of the task graph. Since, in this case study, all jobs have size equal to 1, the optimal makespan is equal to the length of the longest path.

Our simulation results corroborates this theoretical analysis. In fact, for a small number of processors, the performance obtained is greater than the average longest path of the graphs generated by each generation method (the average longest path of each generation method is presented at Section 5.1). When we double the number of processors, all algorithms produces results close to the optimal solution possible. Using the chosen parameters, GGen produced graphs with larger longest path for $\mathsf{G}(\mathsf{n}, \mathsf{p})$ than for the other generation methods. As consequence, the performance obtained by the scheduling algorithms using the workload generated with $\mathsf{G}(\mathsf{n}, \mathsf{p})$ is worse than the performance obtained with the other methods.

It is important to note, however, that the four List Scheduling algorithms used produce some pretty similar results. Graham proved [**?**] that the ratio of the results obtained by any two List Scheduling algorithms can not be bigger than 2. In practice, this ratio is even smaller.

7. CONCLUSIONS

In this paper we presented GGen, a unified and standard implementation of random task graph generation methods used in the scheduling domain and an analysis tool built to help algorithm developers to design a random – but representative – synthetic workload for validation of scheduling algorithms.

We described some of the classical random generation methods used by researchers – $\mathsf{G}(\mathsf{n}, \mathsf{p})$, $\mathsf{G}(\mathsf{n}, \mathsf{M})$, Layer-by-Layer, Fan-in/Fan-out, and Random Orders – and conducted an experimental study to show how the graphs generated by these methods differ from each other. Since there is no single method that is able to generate all possible non-isomorphic graphs, the differences between the results of each generation method must be taken in account by the developer who wants to validate his/her algorithm using some random input.

Choosing which workload is the best for validating an algorithm is a very difficult task. Either the developer knows how to characterize the workload expected to be used with the algorithm, or he/she knows how to generate a good set of random graphs that explores the strength and weakness of his/her algorithm. A case study conducted with some List Scheduling algorithms showed that a same algorithm may obtain a speedup of 3.5 times only by changing the graph generation method used for the performance evaluation.

GGen provides some analysis tools to help developers to understand the used workload and to avoid that a badly chosen set of random graphs mislead the validation of the

	OutDegree		BottomLevel		MinDegree		Random	
	average	std. dev.	average	std. dev.	average	std. dev.	average	std. dev.
GNP(100,0.25)	36	3	35	3	37	3	36	3
GNM(100,300)	25	< 0.5	25	< 0.5	27	1	26	1
FiFo(100,10,10)	28	1	28	1	29	2	29	2
Layer(100,10,0.5)	26	< 0.5	26	< 0.5	27	1	26	1
RandomOrders(100,2)	25	1	25	< 0.5	29	1	27	1
GNP(100,0.25)	35	3	35	3	35	3	35	3
GNM(100,300)	12	2	12	2	13	2	12	2
FiFo(100,10,10)	12	2	12	2	13	2	13	2
Layer(100,10,0.5)	10	< 0.5	10	< 0.5	10	< 0.5	10	< 0.5
RandomOrders(100,2)	17	2	17	2	17	2	17	2

Table 2: Makespan obtained by the simulation of List Scheduling algorithms using 4 (top) and 16 (bottom) processors against a randomly generated workload of 1,000 graphs.

algorithm.

In our future work, we would like to expand our library of generation methods in order to cover more research contexts with possibly different requirements. We are particularly interested in new methods to generate workloads representing real parallel applications like matrix multiplication, LU decomposition, Fast Fourier Transforms, etc. In addition, we would like to conduct analysis on the methods in order to classify them according to their theoretical properties. This classification could help developers to better choose which method generates the needed workload.

8. ACKNOWLEDGMENTS

This work was supported by the GdR RO (Operational Research) of the CNRS.