A Topology – Based Directed Acyclic Graph Generation Strategy

Mihai Paraschiv

mihai.paraschiv@cs.pub.ro

Abstract. Directed Acyclic Graphs (DAGs) are used in various fields of research. Generating these types of graphs has been studied mainly from the perspective of node and link constraints. This paper describes a strategy for building DAGs that focuses on their global structure. First, we look at constructing a node net. Starting from the desired properties of the resulting graph's topological ordering, we proceed with building an envelope in which to place the nodes. Afterwards we associate dependencies on a set of given criteria. Second, we focus on ways of recursively applying the algorithm in order to create a wide range of graphs. Finally, we discuss the implementation of a proof of concept application and possible improvements.

Keywords: directed acyclic graph generation, task graphs, testing, algorithm

1 Introduction

Developing scheduling algorithms requires extensive testing on various types of task graphs. Task collections with dependencies are modeled using Directed Acyclic Graphs in which the nodes are labeled with computation cost information and the links between them are labeled with communication cost.

In graph theory, a topological sort of a directed acyclic graph (DAG) is a linear ordering in which each node comes before all the ones to which it has outbound edges. In the field of scheduling algorithms, topological sorting is an operation commonly applied as the first step in computing some basic task attributes (Ahmad, Kwok, & Wu, 1995): *t – level* (top level), *b – level* (bottom level), *EST* (earliest start time) or *LST* (latest start time).

Consider a representation of a DAG (see Fig. 1) in which tasks are placed vertically based on their level in the topological ordering. The shape defined by outer nodes (in the context of the 2D plot) and the mesh formed by task dependencies can give us insights in the properties of our task set.

In this paper we will study an algorithm inspired by the topological sorting of DAGs. The motivation behind analyzing and implementing such a strategy is the possibility of employing structures that are intuitive and easy to explain in the field of graph generation. Furthermore, we can see that the node structure of a DAG can be adjusted with algorithms that handle its topological ordering shape as a wave or as an evolving geometric structure. As a consequence, using shapes opens up the path to introducing a wide range of already developed methods to the field of graph generation.

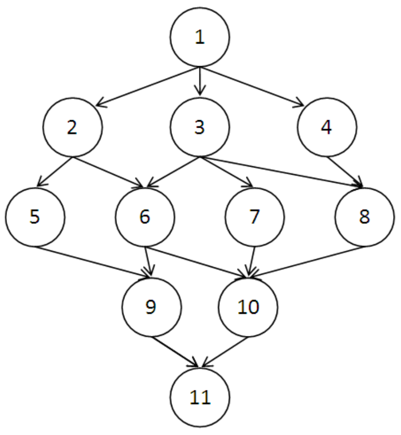

Fig. 1. Task graph with dependencies

The work described here considers two aspects of the generation strategy. First, we look at building the node net. This is done by creating the contour of the envelope in which we will place nodes. Afterwards we associate dependencies on a set of given criteria. Second, we focus on ways of recursively applying the algorithm in order to create a wide range of graphs.
Besides discussing the implications of the proposed strategy, this paper gives an overview of a proof of concept application.

2 Related work

Various methods for generating task graphs have been developed, usually with the purpose of testing a new algorithm. In (Almeida, Vasconcelos, Árabe, & Menascé, 1992) edge connections are built by independent random values. In (Yang & Gerasoulis, 1994), the testing is done by generating the number of layers in the task graph and randomly connecting the nodes. Tobita and Kasahara (2002) propose a standard task graph set for algorithm evaluation. The graphs are built using methods inspired from previous DAG generation strategies and from actual application programs (Kasahara, Honda, & Narita, 1990).
Methods for building graphs with optimal or non-optimal solution are described in the context of algorithm benchmarking by Kwok and Ahmad (1999).
In (Dick, Rhodes, & Wolf, 1998), a pseudorandom DAG generator called TGFF (Task Graphs for Free) is described. Based on a large number of user-controllable parameters, the application can generate independent and partially ordered tasks.

Principles used in generating more general types of graphs may be used in DAG building. This field has received much attention over time, scientists usually trying to develop algorithms that replicate real-life structures. An example of such a generator is R-MAT (Chakrabarti, Zhan, & Faloutsos, 2004). Another example is (Leskovec, Chakrabarti, Kleinberg, & Faloutsos, 2005). Here, the authors describe a realistic graph generation strategy using Kronecker multiplication and prove the correctness of their theory.

3 Building and filling a DAG shape

Consider the example DAG in Fig. 1. One topological ordering of the nodes is: [1], [2, 3, 4], [5, 6, 7, 8], [9, 10], [11]. In this list, the brackets are used to group nodes on the same level.

3.1 Motivation for the described algorithm

To understand the motivation behind the DAG generation that will be described in Section 3.2, we should take a closer look at how a graph is plotted. One way of representing the DAG in 2 dimensions is to place the nodes in a rectangular-cell matrix. Each cell can contain a single node and each node is attached to one cell only. A special case of this type of plot is the *compressed shape* representation, where there are no gaps between filled cells. This type of plot has two properties that we will use:

- The number of rows in the matrix is equal to the number of levels determined by applying a topological ordering algorithm.
- The number of filled columns is equal the number of nodes for a given level.

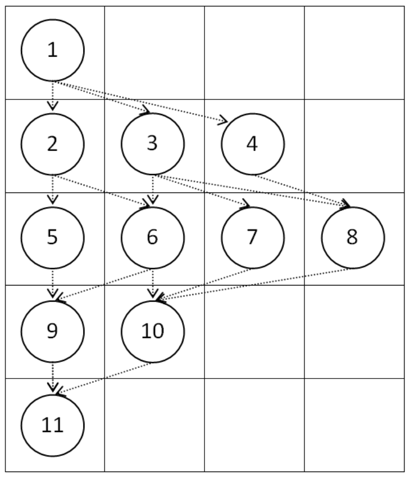

Fig. 2. Compressed shape of a DAG

Removing the links between the nodes allows us to focus on the shape the DAG takes when plotted as described above. Fig. 3 shows the contour obtained from the compressed shape representation of the graph in Fig. 1. It is easily observable that the marked boundary in the figure below can be the plot of a scalar function. It is interesting to observe that the removal of links also gives the opportunity to decouple the generation of nodes from the linking process.

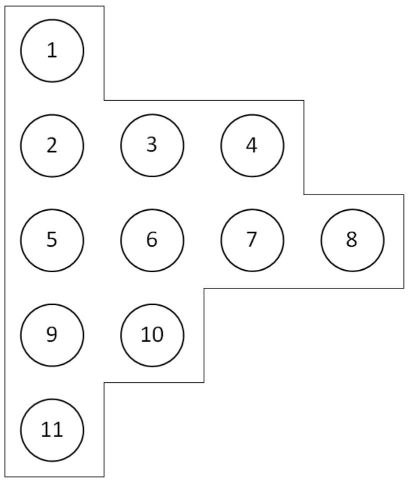

Fig. 3. 2D representation of a DAG's compressed shape

3.2 Algorithm

Following the ideas described above, a base algorithm for DAG generation has been developed:

1. Select a function $f: \mathbb{N}^* \to \mathbb{N}^*$ and a number $M \in \mathbb{N}^*$.
2. Build a node structure with M levels.
3. For each level l of this structure create $f(l)$ nodes.
4. Create links between the nodes.

This algorithm can be applied recursively by transforming nodes from the built structure into DAGs.

4 Algorithm implementation

From the beginning, the implementation was done considering possible enhancement paths. The application created to test the concept was written in Java and contains the following main components and structures:

- Internal graph model
- Boundary generators

- Matrix builders
- Task splitters
- Node linkers
- Factories: Task, Channel, Splitter, Function

4.1 Internal graph model

The internal graph is built using Java Universal Network/Graph Framework[1]. Information is stored on two levels, the first one being the actual graph structure: Node objects linked by Edge objects. The second level, which deals with the properties relevant to task processing, contains the following object classes:

- SimpleTask: contains the cost properties for a task.
- ComplexTask: contains a DAG.
- SimpleChannel: contains the cost properties of the communication channel.

All generated graphs (the root and the ones contained in ComplexTask objects) have only one node at each end.

4.2 Boundary generators

Boundaries (contours) are created as described in Section 3.2. Different function generators are provided and they can be mixed by random selection. After picking the desired functions, they are fed to a boundary generator.

4.3 Matrix builders

The DAG nodes are built considering only one constraint: to fit in a given boundary. After building the matrix, tasks are created and associated to the nodes.

4.4 Task splitters

At the core of the application we find a splitting strategy. A Splitter object is built using a SplitterFactory and its strategy is applied on multiple ComplexTask objects, which are the basis for recursive DAG structures. For the testing application, one type of splitter was employed: TopologicalOrderingSplitter. Objects of this class are instantiated on each level of the recursive structure and implement the core logic of the algorithm described in the previous section.

4.5 Node linkers

The node linking process is done after the nodes have been place into the bounded matrix. Because the linking step is separated from node building, various algorithms

[1] http://jung.sourceforge.net/

can be used. Examples of graphs representing trees that result from running parallel algorithms are shown in (Rădulescu & van Gemund, 1999).

4.6 Factories and generators

In order to ease further development and control graph generation both at global and at local level, the Factory method pattern was used throughout the test application. In addition, various generators (ex: cost, identifiers) were implemented.

5 Results

This section presents two graphs that resulted from running the application. The labels are node identifiers and show the order in which objects have been created, while each node color signifies a different level in the DAG hierarchy (as described in Section 4.4).
Fig. 4 shows the shape at the top level repeated at the second level for three sub graphs, while Fig. 5 depicts a graph that was generated by alternating two patterns.

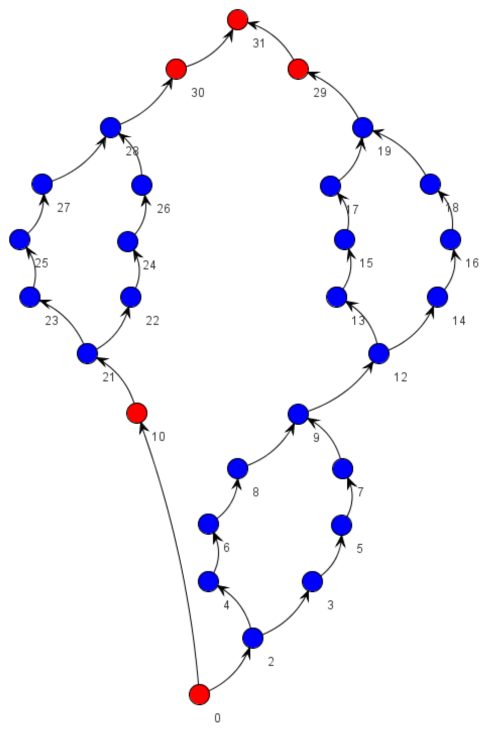

Fig. 4. Two level DAG

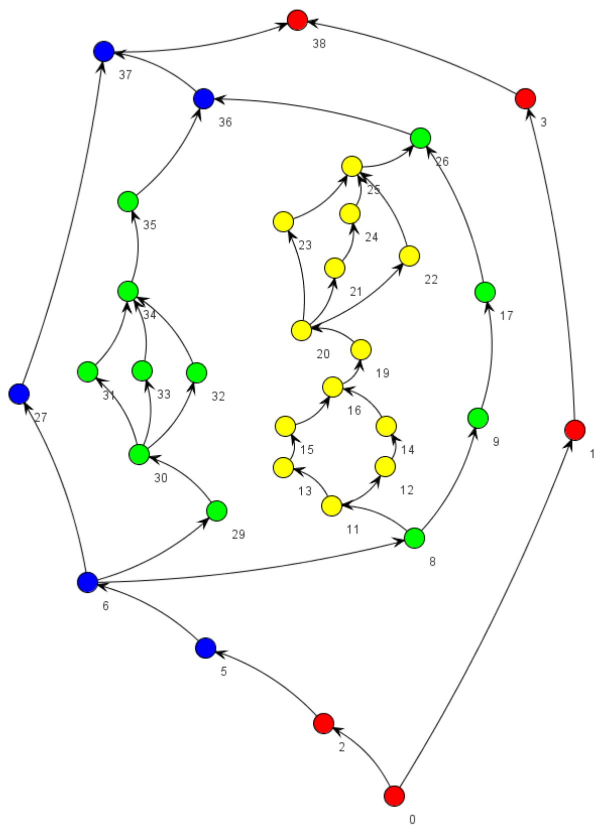

Fig. 5. Four level DAG

6 Further work

The proof of concept application can be enhanced by implementing various types of node creation algorithms, which are not limited to specifying the DAG's shape. These generation strategies can be added as options when splitting a node. Another way to improve the application is the addition of linking algorithms inspired by real life parallel processing graphs.

The algorithm for shape based DAG generation that was described in this paper can be easily parallelized. Work may be split on multiple machines by making use of the hierarchy of graph structures. In addition, we can specify global criteria and rules that state how the generation algorithm is applied at different levels of the hierarchy, making the parallel processes independent. While parallelization may not provide much practical use in building task graphs, these principles can be applied in fields where we need very large hierarchical structures, with highly controllable characteristics.

7 Conclusions

The compressed shape based DAG generation described in this paper gives researchers a new method to manage the scheduling algorithm test process. Mixing various node matrix contours with different linking strategies facilitates the production of large, controllable test sets. The algorithm may be implemented in a general graph building framework and it can provide an intuitive description of the generated structure.

8 References

Ahmad, I., Kwok, Y.-K., & Wu, M.-Y. (1995). Performance Comparison of Algorithms for Static Scheduling of DAGs to Multiprocessors. *Second Australasian Conference on Parallel and Real-time Systems,* (pp. 185-192).

Almeida, V. A., Vasconcelos, I. M., Árabe, J. N., & Menascé, D. A. (1992). Using random task graphs to investigate the potential benefits of heterogeneity in parallel systems. *Proceedings of the 1992 ACM/IEEE conference on Supercomputing* (pp. 683-691). Minneapolis, Minnesota, United States: IEEE Computer Society Press Los Alamitos, CA, USA.

Chakrabarti, D., Zhan, Y., & Faloutsos, C. (2004). R-MAT: A Recursive Model for Graph Mining.

Dick, R. P., Rhodes, D. L., & Wolf, W. (1998). TGFF: task graphs for free. *Proceedings of the 6th international workshop on Hardware/software codesign* (pp. 97-101). Seattle, Washington, United States : IEEE Computer Society Washington, DC, USA.

Kasahara, H., Honda, H., & Narita, S. (1990). Parallel processing of near fine grain tasks using staticscheduling on OSCAR (optimally scheduled advanced multiprocessor). *Proceedings of the 1990 ACM/IEEE conference on Supercomputing* (pp. 856-864). New York, NY, United States : IEEE Computer Society Press Los Alamitos, CA, USA.

Kwok, Y.-K., & Ahmad, I. (1999). Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing* , 381-422.

Leskovec, J., Chakrabarti, D., Kleinberg, J., & Faloutsos, C. (2005). Realistic, Mathematically Tractable Graph Generation and Evolution, Using Kronecker Multiplication. In *Knowledge Discovery in Databases: PKDD 2005* (pp. 133-145). Springer Berlin.

Rădulescu, A., & van Gemund, A. J. (1999). On the complexity of list scheduling algorithms for distributed-memory systems. *Proceedings of the 13th international conference on Supercomputing* (pp. 68-75). Rhodes, Greece: ACM, New York, NY, USA.

Tobita, T., & Kasahara, H. (2002). A standard task graph set for fair evaluation of multiprocessor scheduling algorithms. *Journal of Scheduling* , 379-394.

Yang, T., & Gerasoulis, A. (1994). DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems* , 951-967.