Directed Acyclic Graphs

Ningyuan (Leo) Li, Joshua Dall'Acqua

April 7, 2022

This is the augmented transcript of a lecture given by Luc Devroye on the 7th of April, 2022 for the Honours Algorithms and Data Structures class (COMP 252, McGill University). The subject was an overview on directed acyclic graphs (DAGs).

Introduction

Definition 1 (Directed acyclic graph)**.** A **directed acyclic graph** (abbreviated as DAG hereafter), is a directed graph with no cycle[1]. ■

[1] Cormen et al. [1989]

DAGs have a large number of possible uses, which include:

- indicating partial orders ($\leq, \subseteq$);
- denoting arithmetic expressions and common sub-expressions;
- forming a prerequisite tree;
- forming partitions of regions;
- illustrating PERT[2] networks (used for job planning);
- illustrating critical paths;
- visualising the game of NIM.

[2] **P**rogram **E**valuation and **R**eview **T**echnique

We will provide a few examples to further illustrate its use immediately hereafter.

Example 2 (Partial order of sets)**.** We use Fig. 1 to illustrate how a DAG can be used to indicate the partial order of sets. ■

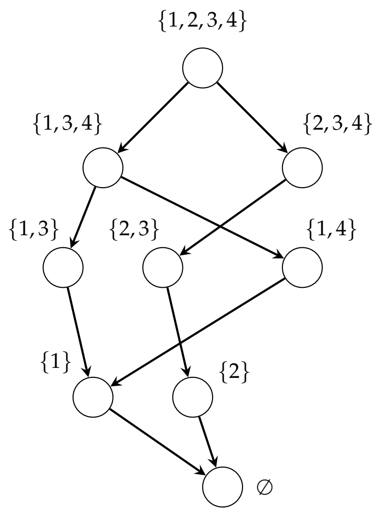


Figure 1: An example that illustrates the partial order of sets.

Example 3 (Expression DAG)**.** We can also use a DAG to represent an arithmetic expression. Particularly, the expression:

$$[\alpha * (\beta + \gamma) + \delta] * [\alpha / (\beta + \gamma) - \delta]$$

is illustrated by Fig. 2. We recall that arithmetic expressions can be represented by trees; however, notice that in this expression, the sum $\beta + \gamma$ appeared two times, and the symbols α and δ appeared two times respectively as well. If we were to use a tree, we would have to include the same information twice; on the contrary, by using a DAG, we could include the sum and the symbols exactly once, which saves space and performance. This is precisely why DAGs play a role in optimising compilers as well.

The DAG has leaves that represent operands (constants and symbols) and internal nodes for operators. If a compiler uses the DAG to produce machine learning instructions, then we might want to minimise the number of iterations, which is equivalent to minimizing the number of internal nodes. ■

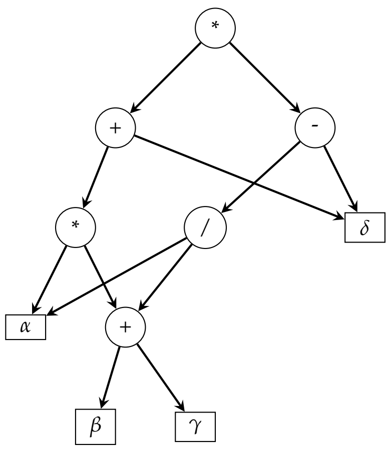


Figure 2: An example that illustrate the use of DAGs to represent arithmetic expressions.

Characteristics of DAGs

Linear ordering

We present the a theorem that characterizes the main property of DAGs:

Theorem 4. *There exists a linear ordering consistent with all direct edges of a* DAG*. In fact, all* DAGs *can be represented in the form described in Fig. 3. This is also known as topological sort or consistent labelling.*

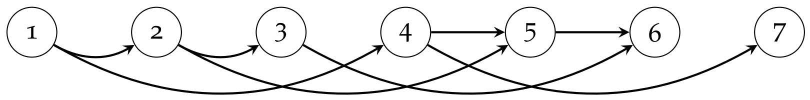


Figure 3: Linear-ordered form of a DAG.

Proof. We prove this theorem by construction. In particular, we use DFS and output nodes in $f[u]$ order. To see why this would give us the correct ordering, we need to recognise that:

Claim 5. *If $\overrightarrow{(u,v)} \in E$, then $f[v] < f[u]$.*

Essentially, during the unique time in the DFS when the traversal from u to v is processed, we have the three possible scenarios:

1. $colour[v] = grey$: this implies that v is an ancestor of u, which in turns implies existence of a cycle (cf. Fig. 4). Since we have a DAG, this case is impossible;
2. $colour[v] = black$: $\implies f[v] < d[u] < f[u]$;
3. $colour[v] = white$: v is a descendant of u, which by nesting implies that:

$$d[u] \underbrace{< d[v] < f[v]}_{\text{nested}} < f[u].$$

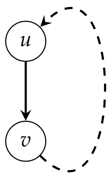


Figure 4: If v is an ancestor of u, then there exists a cycle.

We see that all three scenarios support the correctness of Claim 5, which in turn means that the proof is complete. □

We present an illustrated example of obtaining this linear ordering on a particular DAG:

Example 6. We have a DAG in the form of Fig. 5. By performing DFS and order the $f[\cdot]$ values, we can obtain a linear ordering like in Fig. 6.

■

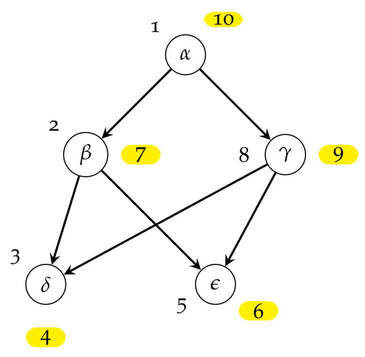


Figure 5: The DAG that we construct the linear ordering on. Particularly, the non-highlighted numbers are the $d[\cdot]$ values, and the highlighted numbers are the $f[\cdot]$ values.

Remark 7. The time complexity for generating this linear ordering is $\mathcal{O}(|V| + |E|)$, which follows directly from the time complexity of the DFS algorithm.

Reversing a DAG

Given the adjacency lists of each vertex, we would reverse a DAG simply by emptying the adjacency list, and reconstructing the adjacency list by having the right side point to the left side. It is perhaps more expedient to consider an example:

1	→	5 2 3
2	→	4
3	→	
4	→	
5	→	4

(a) The adjacency list before the reversal.

1	→	
2	→	1
3	→	3
4	→	2 5
5	→	1

(b) The adjacency list after the reversal.

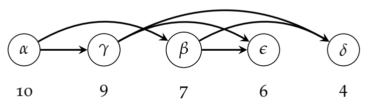


Figure 6: The linear ordering of the DAG in Fig. 5. The numbers underneath the nodes are their associated $f[\cdot]$.

Note that in a sense, we simply reversed the direction of the arrows. Clearly, this procedure takes $\mathcal{O}(|V| + |E|)$ time, since that is precisely the size of the adjacency list (all vertices and edges).

This reversal procedure will come in handy in later discussions.

Remark 8. By using this procedure twice, we can also sort the adjacency list of a graph in linear time!

Applications of DAGs

PERT (also activity) networks and critical paths

We are given a DAG, with nodes denoting a critical moment in time of a project, and an edge $\overrightarrow{(u, v)}$, denoting the activity between moments u, v, taking time $\text{Time}[u, v]$.

We refer to Fig. 7 for the illustration of such problem. Here, we have one starting node (α) and one leaf node (θ) denoting a job finished. The numbering of each edge describes the time it takes for the activity occurring between two moments in time. Let $T[v]$ be the time at which we have finished all jobs leading to node v. Our task is to find the minimum time to complete the entire job. We first recognise that, for all nodes u:

$$T[v] = \max_{u:\overrightarrow{(u,v)}\in E} (T[u] + \text{Time}[u,v]). \tag{1}$$

Using the observation, we could now write the algorithm for this operation:

Algorithm 1 (Minimum time to complete the job).

```
$T[\text{root}] \leftarrow 0$
for all $v$, in topological order, do (1)
return $T[\text{leaf}]$ # time of the project
```

†

As promised, we also define the notion of a critical path:

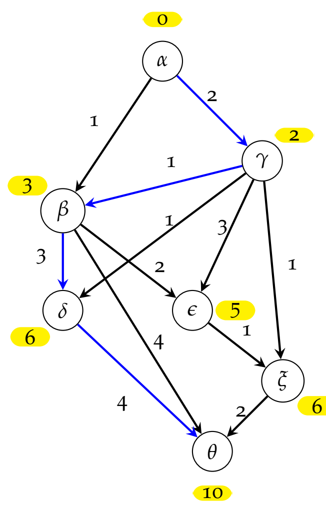


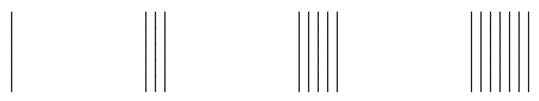
Figure 7: The given DAG in the problem. Here, the blue edges form a critical path. The numbers marked in yellow are the $T[\cdot]$'s for each moment.

Definition 9 (Critical path)**.** A **critical path** is a path on which any delay causes a project delay. ■

With the definition in mind, it is simple to find the critical path: from the leaf node, it can be easily found using Eq. (1), but with arg max instead of max.

The game of NIM

NIM is a removal game based on an integer vector, where each integer represents a number of sticks. For example, the vector $(1, 3, 5, 7)$ corresponds to four piles of sticks:



Players take turns. In one move, they can remove any number of sticks from one pile. The player who is forced to take the *last* stick loses.

This set of rules prompts the question: given the game vector, can we determine if the first player will win or lose? To answers this, we notice that it is expedient to represent the NIM game using a DAG. Say, we take the $(1, 3, 5)$ game as an example, and represent the possible game progression using a DAG (cf. Fig. 8).

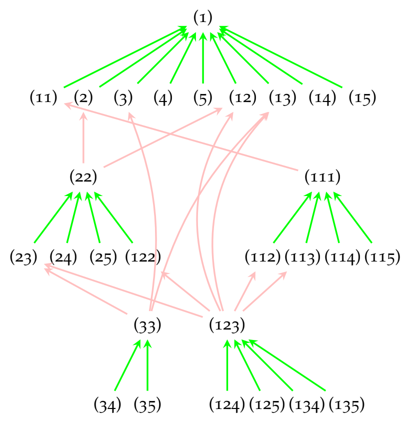


Figure 8: Game of NIM, with a starting state-vector of $(1, 3, 5)$.

We linearly order the positions and determine whether each position is *good* ($\mathcal{G}$) or *bad* ($\mathcal{B}$). For a position v, $v \in \mathcal{G}$ guarantees a NIM if the player plays perfectly; on the contrary, $v \in \mathcal{B}$ means that against a perfect player, a loss will result.

Data Structure 10. kind$[v]$: an array of elements $v \in V$ where each v represents a good position or a bad position, i.e., $v \in \{\mathcal{G}, \mathcal{B}\}$. ■

We are now able to describe this algorithm. We first topologically sort all the positions and store them in a queue Q.

Algorithm 2.

```
1 while |$Q$| > 0 do
2     $u \leftarrow$ DEQUEUE($Q$)
3     kind[$u$] $\leftarrow \mathcal{B}$
4     for all $v$ that can be reached from $u$ in one move do
5         if kind[$v$] $= \mathcal{B}$ then
6             kind[$u$] $\leftarrow \mathcal{G}$
```

†

The operation with greatest contribution to the time complexity in this algorithm is checking the kind of v in line 5. This must be checked for each edge in the graph taking time $\mathcal{O}(|E|)$. Initializing the *kind* array takes time $\mathcal{O}(|V|)$ as it is only done for the vertices. Thus, the overall time complexity of the algorithm is $\mathcal{O}(|E|)$.

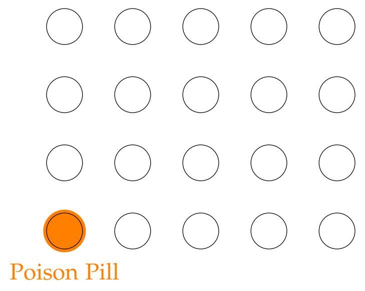


Figure 9: 4 $\times$ 5 game of CHOMP example.

Question 11. *For some game of* NIM $(a_1, a_2, \ldots, a_n)$*, is it possible to construct a mathematical function which takes the game as input and outputs whether it is* "Good" *or* "Bad" *in constant time?*

For a possible answer to the previous answer, one can consult *Winning Ways for Your Mathematical Plays* by Berlekamp, Conway, and Guy[3].

[3] Berlekamp et al. [1982]

CHOMP

Another example of a game solvable using DAGS is CHOMP[4]. Given some $n \times m$ grid of "pills", players alternate turns taking rectangular chomps from the top right corner of the grid. The goal is to avoid eating the poison pill in the bottom left corner of the grid.

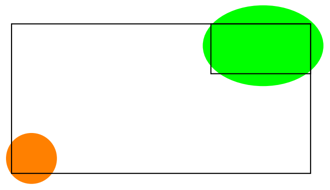


Figure 10: Example of a "chomp" that can be taken out of the grid.

[4] Gale [1974]

Decomposition of Directed Graphs

We first establish the notion of equivalence on two nodes:

Definition 12 (Node equivalence)**.** For a graph $G = (V, E)$, we say that two nodes u, v are **equivalent** $(u \equiv v)$ if there exists a path from u to v and one from v to u, see Fig. 11. ■

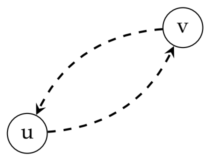


Figure 11: u and v equivalence

This creates a set of equivalence classes in the graph, and we call those equivalence classes *strongly connected components* (abbreviated as the SCC's hereafter). The equivalence classes (on SCC's) form a DAG as showcased in Fig. 12.

We are mainly interested in obtaining the SCC's. To this end, we shall use the following algorithm. We first notice that, if we are using

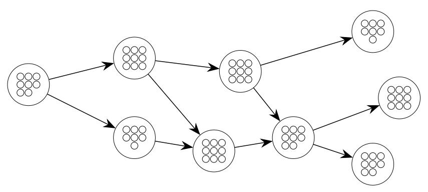


Figure 12: DAG of equivalence classes

the adjacency matrix notation for the graph $G = (V, E)$, then the reversed graph is exactly the transposition $G^T = (V, E^T)$. We shall use the above notation in the algorithm.

ALGORITHM 3 (DETERMINATION OF ALL SCC'S).

```
1   Perform \textsc{dfs} on $G = (V, E)$ and list all nodes in $f[\cdot]$ order;
2   Reverse all  edges, obtaining $G^T = (V, E^T)$;
3   Perform \textsc{dfs} on $G^T = (V, E^T)$ in descending order of $f[\cdot]$
```

†

It is clearer to explain the mechanisms of this algorithm with the aid of illustrations. In Fig. 14 we see a highlighted node α within an SCC $\mathcal{S}$. This is the first node discovered in that SCC in step 1. Being the first node discovered, we know that $\forall u \in \mathcal{S}, d[\alpha] < d[u] < f[u] < f[\alpha]$. Thus α is the node in the SCC with the largest f-value.

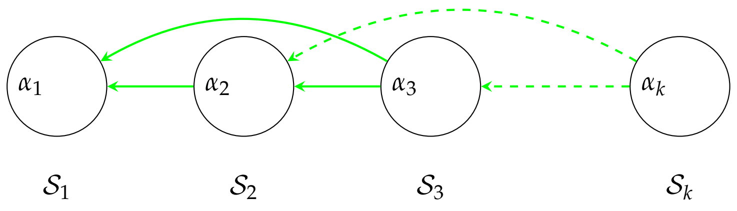


Figure 13: The SCC's in linear order, after the reversal in step 2.

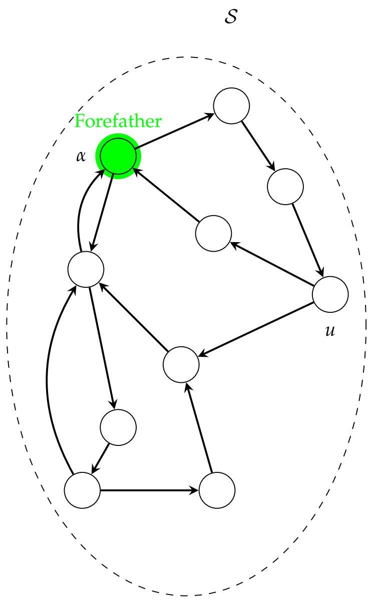


Figure 14: Strongly Connected Component.

Fig. 13 shows a representation of our DAG of SCC's after step 2 of the algorithm. Recall Theorem 4 that a DFS on a DAG gives us the linear ordering of the DAG when ordered by $f[\cdot]$. In other words, the DFS in step 1 gives us the marked nodes $(\alpha_1, \alpha_2, \ldots, \alpha_k)$ of the SCC's in linearly-ordered form. Step 1 prepares us for step 2, which is the (arguably) most central and beautiful step in this algorithm, and the importance of this reversal would become obvious in the third step.

In step 3, the DFS started at α must visit all nodes in $\mathcal{S}$ by the definition of an SCC; however, due to step 2, it **cannot** search outside of the SCC that α is part of. We explain now why this is the case:

- if α is located in the very first SCC of the reversed linearly-ordered DAG, then it is impossible for the DFS to reach another SCC since all 'bridges' from other SCC's must be pointing in, i.e., the out-degree of the first SCC is zero. Hence, the DFS halts after finding all elements in this SCC;
- if α is in an SCC that is somewhere in the middle of the reversed linearly-ordered DAG, then by the same argument it could not go to the other SCC's that 'bridge' towards it. However, it also could not go through any edge that points out, since by linear ordering, all nodes in the previous SCC's would have been black already, which does not permit another search.

As a result, every restart of DFS visits exactly one of the SCC's, in the order given in Fig. 13.

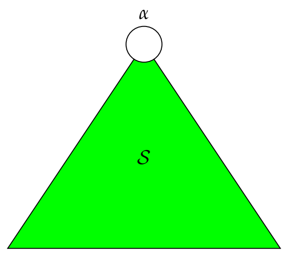


Figure 15: DFS tree of an SCC. Note, α also has children which are not in S. This is how components become connected.

References

Elwyn R. Berlekamp, John H. Conway, and Richard K. Guy. *Winning Ways for Your Mathematical Plays*. Academic Press, 1982.

T.H Cormen, C.E. Leiserson, R.L.Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3rd edition, 1989. ISBN 9780262033848.

David Gale. A curious NIM-type game. *American Mathematical Monthly*, 81:876–879, 1974.