ORDERED DAGS: HYPERCUBESORT

MIKHAIL GUDIM

Abstract. We generalize the insertion into a binary heap to any directed acyclic graph (DAG) with one source vertex. This lets us formulate a general method for converting any such DAG into a data structure with priority queue interface. We apply our method to a hypercube DAG to obtain a sorting algorithm of complexity $\mathcal{O}(n\log^2(n))$. As another curious application, we derive a relationship between length of longest path and maximum degree of a vertex in a DAG.

1. Introduction

Consider a sorted linked list, a binary heap and a Young tableau (see Problem $6-3$ in [1]). The process of inserting a new element is very similar for all three: we repeatedly exchange newly added element with one of its neighbours until it is in correct place. This simple observation is the main motivation for the present work.

Now let us briefly outline the contents. In Section 2 we fix notation and terminology for the entire paper, in particular we define the notion of an ordered DAG. Our main technical result is in Section 3 where we prove that the structure of an ordered DAG can be easily maintained. This allows us to construct a data structure with priority queue interface from any ordered DAG (Section 4). Section 5 demonstrates that some classical algorithms can be viewed as a special case of our general construction. The interaction between sorting and DAGs can be applied to prove statement about DAGs. As an example, we derive a relationship between the maximum degree of a vertex in a DAG and length of longest path (Corollary 5.5). The most juicy part of the paper is Section 6. There we apply our method to a case where underlying DAG is a hypercube and arrive at a sorting algorithm HypercubeSort, which to our knowledge has not yet been described. In Proposition 6.1 we derive an exact expression for the complexity of HypercubeSort in the worst case. Asymptotically it is $\mathcal{O}(n\log^2(n))$.

The Java implementation of HypercubeSort is available at [2].

2. Terminology and Notation

Definition 2.1. Let G be a DAG and suppose that each node v of G has an integer attribute $v.label$. We call such a DAG **labeled DAG**. We denote the multiset of all labels in a labeled DAG G by $labels(G)$. Let (u,v) be an edge in a labeled DAG G. We use the following terminology:

(1) Vertex u is a **previous neighbour** of v and v is a **next neighbour** of u.

(2) If $u.label \leq v.label$ we say that (u,v) is a **good** edge. Otherwise, (u,v) is a **bad** edge. If (u,v) is a bad edge, we call u **violating previous neighbour** of v and we call v **violating next neighbour** of u.

(3) Labeled DAG G is called **ordered** if all its edges are good.

Example 2.2. An example of a ordered DAG is shown in Figure 1. Note that we allow equal labels for vertices.

Date: September 17, 2017.

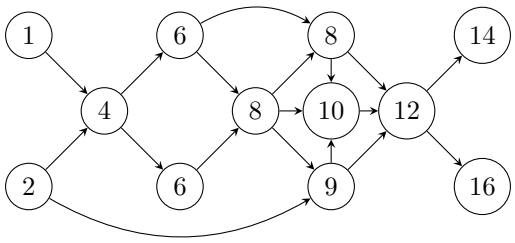


FIGURE 1. An example of ordered dag.

3. Maintaining Ordered DAGs

We now describe the procedure LOWERLABEL which is the workhorse of the entire paper. Generally speaking, it is just a generalization of insertion into a heap, but nevertheless we take care to prove its correctness rigorously. In the following pseudocode we assume that we have a procedure GETLARGESTVIOLATING(G, v) which when given a pointer v to a vertex in a labeled DAG G returns the pointer to a vertex u which is a violating previous neighbour of v with largest *label* attribute. If there is no previous violating neighbours (this includes the case when there are no previous neighbours at all), the procedure returns *null*.

Algorithm 1 lowerLabel

```
1: procedure LOWERLABEL(G, v, newLabel)
2:     ▷ G is a labeled DAG and v is a pointer to a vertex in G and newLabel is an integer less than v.label
3:     v.label = newLabel
4:     current = v
5:     violating = true
6:     while violating do
7:         largestViolating = GETLARGESTVIOLATING(G, current)
8:         if largestViolating == null then
9:             violating = false
10:        else
11:            exchange current with largestViolating
       return
```

Proposition 3.1. *Let G be an ordered DAG, v a vertex in G and newLabel an integer less than $v.label$.*

(1) The procedure LOWERLABEL(G, v, *newLabel*) *terminates.*

(2) After the termination G remains an ordered DAG.

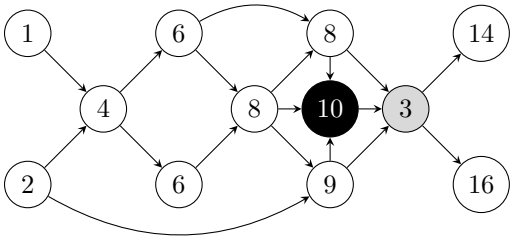
(3) Let L denote the multiset of labels of G before the call to LOWERLABEL *and L' denote the multiset of labels of G after the call. Then L' is L with $v.label$ replaced by newLabel.*

Example 3.2. Before the proof, it would be illustrative to look at an example. Figure 2 shows the execution of LOWERLABEL on a DAG from Example 2.2.

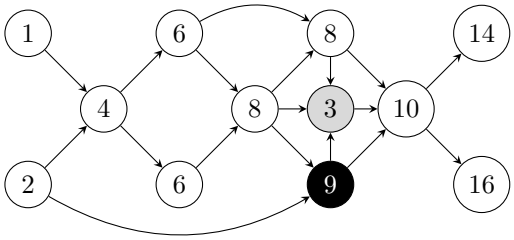
Proof. It is easy to see that the procedure terminates because G is a finite DAG. Statement (3) is also clear, since the only step which changes the multiset of labels is in line 3 of the pseudocode.

We prove (2) by showing that the LOWERLABEL maintains the following two-part invariant:

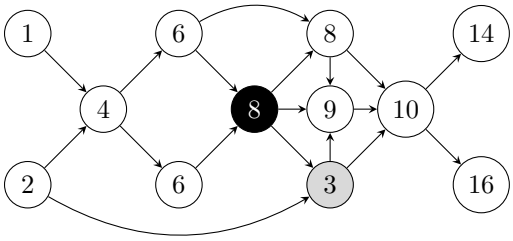
At the end of each iteration of the **while** loop in lines 6-11:



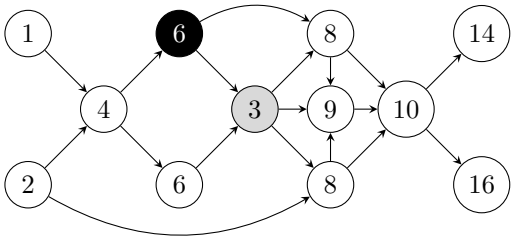
(A) After execution of line 4



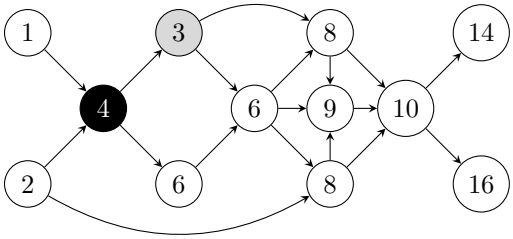
(B) End of iteration 1



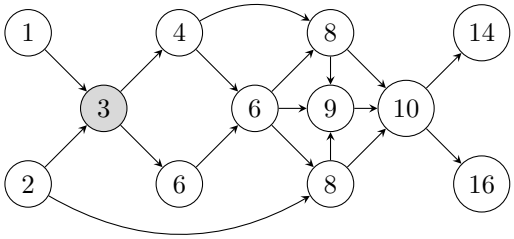
(C) End of iteration 2



(D) End of iteration 3



(E) End of iteration 4



(F) End of iteration 5. The DAG G is now ordered, procedure exits.

FIGURE 2. Procedure LOWERLABEL applied to ordered DAG from Figure 1 to lower label of a vertex from 12 to 3. The vertex *current* is highlighted in gray and the vertex that will be returned by GETLARGESTVIOLATING at next iteration is black.

(1) All bad edges in G (if any) are entering *current*.

(2) For any previous neighbour p of *current* and any next neighbour n of *current* $p.label \leq n.label$.

Before line 4 G is ordered. Right after line 4 executes *current* is v and it is the only vertex whose *label* attribute changed, so part (2) of the invariant is maintained and the only edges that could become bad are those entering and leaving *current*. Since the *label* of v becomes smaller, all edges leaving *current* remain good, but edges entering *current* could become bad, so part (1) of the invariant is maintained.

Now assume the invariant was maintained for the first m iterations of the **while** loop. We show that it is maintained after $(m+1)$-st iteration. If GETLARGESTVIOLATING returns *null* it means that there are no bad edges entering *current* and by part (1) of the invariant G is ordered DAG. The procedure exists. Now we consider the case when GETLARGESTVIOLATING returns non-null value.

Let x denote the vertex in *current* variable before the exchange in line 11. Let $\alpha = x.label$, denote by $p_1, p_2, \ldots, p_k$ previous neighbours of x, by $n_1, n_2, \ldots, n_l$ next neighbours of x and suppose GETLARGESTVIOLATING returns p_k with $p_k.label = \beta$. By definition of GETLARGESTVIOLATING the following inequalities are true:

$$\beta > \alpha \tag{1}$$
(because p_k is violating)

and

$$\beta \geq p_i.label \text{ for all } i \text{ with } 1 \leq i \leq (k-1) \tag{2}$$
(because $p.k$ is largest violating previous neighbour)

Because part (1) of the invariant was maintained up to this moment, the following inequalities are true:

$$\alpha \leq n_j.label \text{ for all } j \text{ with } 1 \leq j \leq l \tag{3}$$

Because part (2) of the invariant was maintained up to this moment, the following inequalities are also true:

$$p_i.label \leq n_j.label \text{ for all } i \text{ and } j \tag{4}$$

After the exchange in line 11 *current* is p_k and *label* attributes of only two vertices changed, namely $x.label = \beta$ and $p_k.label = \alpha$. All edges entering x are of the form (p_i, x) for $1 \leq i \leq k$. By inequalities in (2) all edges (p_i, x) for $1 \leq i \leq (k-1)$ are good and by inequality (1) the edge (p_k, x) is good. So all the edges entering x are good. All the edges leaving x are of the form (x, n_j) for $1 \leq j \leq l$. They remain good by inequalities in (4) applied with $i = k$. Now let us look at $current = p_k$. Before the exchange all edges entering and leaving p_k were good. After the exchange, by the inequality (1) value of $p_k.label$ lowered. This means all the edges leaving p_k remain good and only edges entering p_k could become bad. So part (2) of the invariant is maintained. □

The following pseudocode shows that in abstract sense the processes of lowering and raising value of *label* attribute are equivalent:

Algorithm 2 raiseLabel

```
1: procedure RAISELABEL(G, v, newLabel)
2:     ▷ G is a labeled DAG and v is a pointer to a vertex in G and newLabel is an integer greater than v.label
3:     Multiply the label of each vertex of G by (−1)
4:     Reverse each edge in G                                  ▷ after this G is ordered
5:     Run the procedure LOWERLABEL(G, v, −newLabel)
6:     Multiply the label of each vertex of G by (−1)
7:     Reverse each edge in G
```

In practice one can implement RAISELABEL in a completely symmetrical way to LOWERLABEL by reversing the directions of edges and meaning of comparisons.

4. Ordered DAG Data Structure

With Algorithms 1 and 2 at hand, it is probably clear how to make any DAG G with only one source vertex induces a data structure OrderedDag_G. However, there are some details which we do not want to neglect.

The constructor of OrderedDag_G assigns value of ∞ to every vertex in the DAG.

To insert a new element with label l in principle one can pick *any* vertex v with $v.label = \infty$ and then call LOWERLABEL(v, l) to restore the order in G. But arbitrary choice is ambiguous and may be non-optimal. Therefore, we also assume that we have a (stateful) procedure GETNEXT() which returns the next vertex of G in breadth-first order. In particular, the first call to GETNEXT returns the source vertex s. The second call returns one of next neighbours of s. After all neighbours of s were returned, it returns neighbours of neighbours of s and so on. Thus we have:

Algorithm 3 insert

```
1: procedure INSERT(l)
2:     ▷ The element l is the new label to be inserted into G.
3:     nextVertex = GETNEXT()
4:     LOWERLABEL(G, nextVertex, l)
```

Since G is ordered DAG, the source vertex s must have the minimum label. The procedure GETMIN returns this vertex.

Given a vertex v in G we can remove v by calling RAISELABEL$(G, v, \infty))$. In particular, we can remove the minimum value this way:

Algorithm 4 removeMin

```
1: procedure REMOVEMIN
2:     ▷ Removes the vertex with minimum label attribute from G.
3:     s = GETMIN()
4:     RAISELABEL(G, s, ∞)
```

For the reference we make the following Summary:

Summary 4.1. *Any DAG G with N vertices and only one source vertex induces a data structure* ORDEREDDAG$_G$*, which implements the following interface:*

(1) ORDEREDDAG(G)*: creates the data structure* ORDEREDDAG$_G$*, with $v.label = \infty$ for each vertex v in G.*

(2) INSERT(l)*: replaces one of ∞'s (if any left) with l in labels(G).*

(3) GETMIN()*: returns a vertex whose label attribute is a minimum value of labels(G).*

(4) REMOVEMIN()*: removes (a) minimum value from labels(G).*

(5) LOWERLABEL$(v, newLabel)$*: replaces $v.label$ in labels(G) with $newLabel < v.label$.*

(6) RAISELABEL$(v, newLabel)$*: replaces $v.label$ in labels(G) with $newLabel > v.label$.*

5. General DAGSort

Given a DAG G with n vertices we can use ORDEREDDAG$_G$ to sort array of n elements.

Algorithm 5 DAGSort

```
1: procedure DAGSort(G, A)
2:     ▷ G is a DAG with n vertices and A is array with n elements.
3:     dag = OrderedDag(G)
4:     for all elements a in A do
5:         dag.insert(a)
6:     A' = new array of size n
7:     for i from 0 to (n − 1) inclusively do
8:         A'[i] = dag.removeMin()
       return A'
```

For a general DAG we have the following obvious coarse upper-bound on complexity:

Proposition 5.1. *Let G be any DAG with n vertices and only one source vertex s. Let D_{in} and D_{out} denote the highest in- and out- degree of a vertex in G respectively and let L denote the maximum length of a simple path starting at s. Then* DAGSort$(G, \bullet)$ *makes at most $nL(D_{in} + D_{out})$ comparisons.*

Proof. During insertion new element will be exchanged with at most L vertices. To make one exchange we need at most D_{in} comparisons, so the cost of inserting all n elements into OrderedDag$_G$ is bounded above by $nD_{in}L$. The other term $nD_{out}L$ comes from extracting minimum element n times. □

Now we put some well-know algorithms in the context of general DAGSort.

Example 5.2. Let G be a DAG in Figure 3. With this G as an underlying DAG the DAGSort$(G, \bullet)$ is the selection sort algorithm. The vertex s always contains the minimum value.

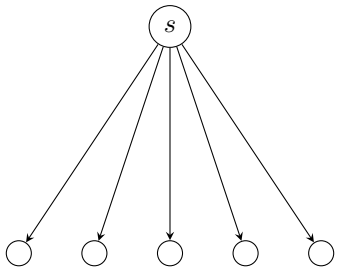


Figure 3

Example 5.3. Let G be a DAG in Figure 5. With this G as an underlying DAG the DAGSort$(G, \bullet)$ is the insertion sort algorithm.

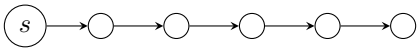


Figure 4

Example 5.4. Let G be a DAG in Figure 5. With this G as an underlying DAG the DAGSort$(G, \bullet)$ is the sorting algorithm using Young tableau.

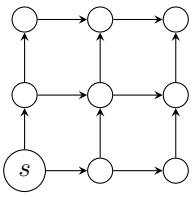


FIGURE 5

One can generalize two-dimensional Young tableaux to k-dimensional tableaux: the underlying DAG is a k-dimensional grid. If a grid has n elements then the length of longest path is $k\sqrt[k]{n}$ and each vertex has at most k previous and k next neighbours. Thus by upper bound of 5.1 the DAGSORT using k-dimensional Young tableaux is of complexity $\mathcal{O}(kn^{1+\frac{1}{k}})$.

If we allow very high in-degree of a vertex in a DAG, we can make the longest path in the DAG small - consider DAG with one source vertex connected to other vertices. At the other extreme, consider a DAG where all the vertices are arranged in a linked list: all the vertices have small degree but the longest path is long. What can we say in a generic case?

Corollary 5.5. *Let G be any DAG with $n \geq 2$ vertices and only one source vertex s. Let D_{in} and D_{out} denote the highest in- and out- degree of a vertex in G respectively and let L denote the maximum length of a simple path starting at s. Then the following inequality holds:*

$$\frac{1}{n}\log(n!) \leq L(D_{in} + D_{out})$$

In other words, to densely pack (length of longest path is small) n vertices into a DAG one cannot avoid vertices with high in-degrees.

Proof. Let G be any DAG with $n \geq 2$ vertices and let DAGSORT_G denote the sorting algorithm defined by G. Let I denote the input on which DAGSORT_G makes at least $\log(n!)$ comparisons to sort I. Such input always exists (see Section 8-1 of [1]). Let us denote the number of comparisons made by DAGSORT_G to sort I by $T(I)$. We have the following lower bound on $T(I)$:

$$\log(n!) \leq T(I) \tag{5}$$

On the other hand, by the bound in Proposition 5.1 we have:

$$T(I) \leq nL(D_{in} + D_{out}) \tag{6}$$

Combining the two inequalities above gives the result. □

6. HYPERCUBESORT

Let S be a set with k elements. The set of 2^k subsets of S forms a DAG: vertices are subsets there is an arrow from subset S to subset T if T is obtained from S by adding one element. We denote this DAG by $DAG(S)$. An example with $S = \{1, 2, 3\}$ is shown in Figure 6.

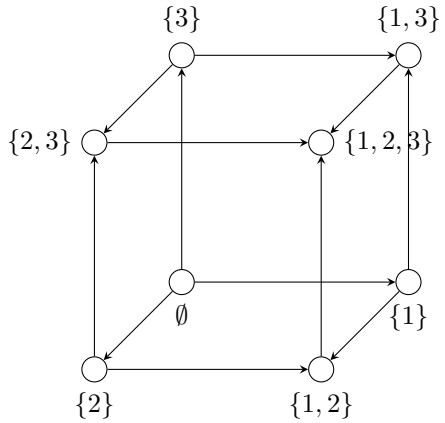


FIGURE 6. $DAG(\{1,2,3\})$

The DAG of subsets $DAG(S)$ is a k-fold direct product of a DAG in Figure 7 with itself. Such a DAG is called k-dimensional **hypercube**. Since the vertices can be identified with subsets we have a notion of **cardinality of a vertex**: we say that a vertex v is hypercube is of cardinality m if v corresponds to an m-element subset of S. There is only one vertex s of cardinality zero (it corresponds to the empty subset) which is the only source vertex. Let u be a vertex of cardinality $m > 0$. Then length of any path from s to u is m, u has m previous neighbours and $(k-m)$ next neighbours.

FIGURE 7

We can apply the general DAGSORT to the hypercube DAG and we call the resulting algorithm HYPERCUBESORT. To implement HYPERCUBESORT we can identify a vertex of a hypercube with the subset it represents which can be written uniquely as a binary string of length k. For example, the subset $\{2,4,5\}$ of $S = \{1,2,3,4,5,6,7,8\}$ is written as 01011000. In turn, we can convert each binary string to a number, which can serve as an index into array. The procedure GETNEXT can be implemented by first listing all 0-element subsets, then all 1-element subsets, then all 2-element subsets and so on. The exact algorithm that does not require any extra memory and each call takes $\mathcal{O}(1)$ time is described in Section 3 of [3]. Our Java implementation of HYPERCUBESORT with all the details is available at [2], but in our version for simplicity we precompute the order of vertices by a recursive procedure and store the result.

By upper bound in Proposition 5.1 it takes $\mathcal{O}(n\log^2(n))$ comparisons for HYPERCUBESORT to sort n element array. Unfortunately, this bound cannot be improved by a more detailed analysis as the proof of the following proposition shows.

Proposition 6.1. *To sort n elements the algorithm* HYPERCUBESORT *makes at most $\mathcal{O}(n\log^2(n))$ comparisons.*

Proof. We assume that n is a power of two. By symmetry between insertion and deletion operations, it is enough to estimate the number of comparisons to insert $n = 2^k$ elements. Denote this number by $T(n)$. To insert one vertex of cardinality i it takes in the worst case $i + (i-1) + \cdots + 1 = \frac{1}{2}(i+1)i$ comparisons. There are exactly $\binom{k}{i}$ vertices of cardinality i. Thus

$$T(n) = \sum_{i=0}^{k} \frac{1}{2}\binom{k}{i}(i+1)i \tag{7}$$

To evaluate the above summation, consider the function f defined by

$$f(x) = x(1+x)^k = \sum_{i=0}^{k} \binom{k}{i} x^{i+1}$$

Take derivatives of the two expressions of f:

$$f'(x) = (1+x)^k + kx(1+x)^{k-1} = \sum_{i=0}^{k} \binom{k}{i} (i+1)x^i$$

And the second derivatives:

$$f''(x) = k(1+x)^{k-1} + k(1+x)^{k-1} + k(k-1)x(1+x)^{k-2} = \sum_{i=1}^{k} \binom{k}{i} (i+1)ix^{i-1}$$

By comparing right-hand side of the last equation with equation (7) we see that $T(n) = \frac{1}{2}f''(1)$. But we can evaluate $f''(1)$ using formula on the left-hand side of the above equation with $x = 1$. The precise expression we get is

$$k2^k + k(k-1)2^{k-2}.$$

which is clearly $\mathcal{O}(2^k k^2)$. □

References

[1] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.
[2] M. Gudim. hypercubesort. github.com/mgudim/hypercubesort, 2017.
[3] Albert. Nijenhuis and Herbert S. Wilf. *Combinatorial algorithms*. Academic Press New York, 1975.

E-mail address: mgudim@gmail.com