Homework

About "I/O-Efficient Algorithms for Topological Sort and Related Problems"

December 1, 2019

This assignment strives to present in a concise manner one of the accepted papers from this year's SODA Conference, called "I/O-Efficient Algorithms for Topological Sort and Related Problems". Initially, we will start by depicting this paper's title for getting a better understanding of it, followed by a short presentation of its main sections and we will conclude by detailing this paper's actual contribution.

The I/O model approach of algorithms can be easily distinguished from the classical RAM model approach by the way we compute its operations cost. It is known that an I/O model has two types of memory available, each one of them being organized in blocks of a certain size (called B in this paper). The first one is the internal memory, also called cache, which has a limited amount of space available (called M in this paper). Computations can be done only with data inside this memory, so we drastically depend on it having all the information needed. The second type of memory in an I/O model is the external memory, which has unlimited space available. Furthermore, computations themselves are considered free, since the access time for data inside cache is between 10-100 times faster than the one in the classical RAM model, and the so-called I/O cost represents the number of I/Os operations performed (i.e. a block of size B transferred between the internal and the external memory). Continuing with the title analysis, the phrase "I/O-Efficient" algorithms wants to emphasize that the presented algorithms are the first ones that, for a directed acyclic graph $G = (V, E)$, don't need at least one I/O operation per-vertex and their complexity is $\mathcal{O}(sort(E) \cdot poly(log(V))$ (with high probability, since both algorithms are randomized ones), where $sort(E) = \mathcal{O}(\frac{E}{B} log_{\frac{M}{B}}(E/B))$.

Moving forward, this paper discusses a new approach for topological sorting, which also influences the well known I/O-efficient time-forward processing technique. Since it positively affects the time-forward processing, this new randomized algorithm for topological sort also impacts other circuit-like problems for directed acyclic graphs, such as finding the shortest paths.

In order to describe the randomized algorithm for topological sort, I'll firstly state both the topological sort problem and the time-forward processing. Firstly, in both cases, we have a directed acyclic graph (let's call it $G = (V, E)$), where V is the set of vertices and E is the set of edges. The topological sort wants to find a certain order so that $\forall$ edges $L(v, w) \in E$, vertex v comes before vertex w. For the RAM approach, there are know two methods, both running in $\mathcal{O}(V + E)$ time complexity. Anyway, for the I/O model, there are known two algorithms so far, both of them having a worse time complexity than the classical

RAM approach. Moving forward, time-forward processing is an efficient technique for most circuit-like problems on directed acyclic graphs. Since the biggest issue encountered in I/O models is that they require at least one I/O operation per vertex, time-forward processing manages to simulate this exact process by propagating information.

During the overview of their approach, it is presented the main idea behind both of their algorithms, but there is also some terminology introduced. Initially, each vertex gets assigned a certain label represented by a number and the goal is that $\forall\ L(v, w) \in E$, the label of v is smaller than the label of w. All edges that satisfy that condition are called satisfied, while all the other ones are considered to be violated edges. One of the main advantages that this algorithm has is that is guarantees that once an edge becomes satisfied, it never goes back to being violated.

The proposed algorithm has about ten main steps. It starts by initializing a couple of global parameters which are K (the number of possible distances from which we can select a random distance) and λ (the maximum depth that may be considered). Afterward, the recursion starts by calling this topological sorting method for the whole graph G, where the first index of an edge is 1, and the last one is n. Going on, it generates a random number d which is uniformly distributed in $[d_{min}, d_{max}]$. Both d_{min} and d_{max} are computed based on λ and K and later on in this paper is discussed how to choose both of those parameters, and how their values either gives a higher probability of edges being violated or increases the I/O cost. The next step of this algorithm is to generate a random permutation of priorities for each vertex ($\rho(v)$ and based on that it is also computed the maximum priority from all the predecessors which are located at most d distance. Here, d distance between two vertices (u, v) means that there's a road which has at most d edges violated. Based on both $l(v)$(maximum label of predecessors at most d distance) and $index(v)$ (which is the actual position in the array), it is sorted lexicographically, vertices are partitioned based on their $l(v)$ value and a new recursion step takes place for each partition. That sorting step is the only one where edges may become satisfied. This algorithm stops when either recursion's depth is higher than λ or there's only one vertex left in the recursion.

The next part of the paper discusses in more detail how priorities are propagated between vertices and defines two types of propagation: the first one is called "satisfied-edge propagation" (its goal is to find the highest priority, but only by following paths with satisfied edges), and the second one is called "violated-edge propagation" (which allows one edge to be violated). By following one of these two types of propagation for $d + 1$ steps, it is stated in $Lemma 3.1$ the formula for computing the maximum priority which is also used in the previously mentioned algorithm. Details about implementing satisfied-edge propagation show that there can be done one single scan which identifies all those edges that are satisfied. With only those satisfied edges, we can immediately compute the maximum between a vertex's current value and all of its immediate predecessors. Regarding the violated-edge propagation, it is stated that it requires both sorting and scanning. Firstly, we sort by the index of u all edges $(u, v) \in E$, consider for each edge that its priority is u's priority, then sort by the index of v, so that all incoming edges are in consecutive order and compute the maximum priority as the maximum between the previously assigned priority (after u's index sorting) of v and all the other incoming edges.

For implementing the recursion, it is emphasized that the algorithms shouldn't be considered to make multiple recursive calls, but it should be seen as a problem for which we solve one level at a time, by setting each sub-problem's id as an extra component besides index and priority label. This could be done according to them in $O(scan(N))$ I/O operations, where $scan(N) = \Theta(N/B)$. Also, another important part of this paper is the *Lemma* 3.2 which states that there is a constant c (where c is the number of blocks in cache), so that the I/O cost for the recursive algorithm is at most $O(K\lambda^2 sort(E)$, with both K and λ initially chosen. For proving this result, it is considered that the maximum number of recursive iterations is λ, and for each of those iterations, the random number d which is uniformly distributed in $[d_{min}, d_{max}]$ can be at most $\lambda \cdot K$ labels and the label propagation step can be achieved in $O(sort(E))$ I/O operations. Therefore, each recursive iteration may request at most $O(\lambda K sort(E))$ I/O operations.

Two other important results are presented in *Theorem* 4.1 and *Theorem* 4.2 which focus on how λ and K should be chosen. The first theorem mentioned for this topological sorting algorithm analysis states how both λ and K should be chosen for high success probability. Their result shows that for chosen $\lambda \geq 32lg(V)$ and $K \geq \lambda log(V)$, any $c > 0$ with failure probability of $\frac{1}{|V|^c}$ gets the graph topologically sorted after at most $(c+2)log(V)$ steps. The second theorem that I've mentioned comes in to demonstrate the existence of two such variables K and λ, so that this previously mentioned algorithm's complexity is $O(sort(E)log^5(V))$.

Moving forward, the second algorithm presented in this paper is for strongly connected components. I'll start by defining the problem of strongly connected components, continue by summarizing the algorithm presented in the paper and conclude with their analysis about I/O efficiency. Given a graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, a strongly connected components is a maximal subset of V where for any two vertices (u, v) from that subset, there is a road both from u to v and from v to u. Starting from this initial notion, this paper also defines a condensation of G, called H, which has a smaller number of vertices than G. Each vertex from H is itself a strongly connected component and by making this compression for each strongly connected component into one single vertex, the resulting graph is a directed acyclic graph (called $H = (V_H, G_H)$). By reducing our problem to a directed acyclic graph, it is presented an I/O-efficient algorithm similar to the one for topological sorting.

The strongly connected components algorithm has two global, fixed parameters similar to the topological sorting one, also called K and λ. In this case, both λ and K have the same meaning, which is that λ sets a maximum recursion depth, while K limits the range of possible distances. Initially, the compressed graph H is equal to G, since there was no strongly connected component discovered yet and each vertex is assigned to its own one. There are three information stored throughout the whole process: a mapping for each vertex to the corresponding strongly connected component, H and the order of vertices V_H. Moving one, I will present a high-level structure of the strongly connected components algorithm. Starting from the initial graph, we will loop over H until it becomes topologically sorted. During each loop, the first step is calling the strongly connected components method for the current H graph, from the first index to its last one, with an initial depth of 0. The second step is performing the contraction for vertices with the same label and updating H. Anyway, the main method is the recursive one for strongly connected components which has the same stopping condition as the one for topological sorting. This method stops if depth has become higher than the allowed maximum depth λ or we encounter a unit case (there is only one vertex left in the recursion). If none of those cases are applicable, it is computed a lower bound and an upper bound, similarly to the ones from topological search, based on K and λ. With those two d_{min} and d_{max} set, it is elected d, uniformly distributed in $[d_{min}, d_{max}]$. Identically, each node will get assigned a label (or a priority) based on which there will be done both a forward and backward search for finding the maximum label that is at most d distance. In this case distance, d, represents the same thing as it did for the topological search algorithm, which is the maximum number that of edges that can be violated for reaching another vertex. Based on the two maximum labels that we get from forward-searching and backward searching (called $f(v)$ and $b(v)$, where $v \in V_H$, there are described three possible cases. For each vertex $v \in V_H$, if $f(v) \geq b(v)$, then v will get assigned $f(v)$ priority (so that if a vertex has higher priority in the forward search, we would want that node to come as close to the beginning during the sorting step), if $b(v) \geq f(v)$ then we want that vertex to be pushed to the end of the indices array, while for the last case where $b(v) = f(v)$ we want to compress that whole subset of vertices to a single component. By finding $b(v) = f(v)$, it is discovered a strongly connected component which can now be represented by one single vertex in H. The next steps are the same as in the topological sorting algorithm, meaning that it is done a sorting for all vertices based on their label/priority so that later on there will be formed partitions by grouping all the vertices that have the same priority label. With this partitioning done, there comes the next step where graph contractions may be done. For each first element of each partition, it is checked if the backward search priority for that vertex is the same as the forward search priority. If those two are equal, we've found a strongly connected component and we stop the recursion for that subset of vertices. Otherwise, the recursion gets called again for that smaller range of indices which describe the vertices subset with the same label.

For this strongly connected components algorithm, there are three main results illustrated. The first one sets the time-bound for one execution of the recursive strongly connected components method, which has I/O cost of $O(K\lambda^2 sort(E))$. This lemma is the equivalent of the one presented for the topological search and the only significant difference is that in this case labels are propagated both backward and forward. The second one that I've considered to be an important result is *Lemma* 5.8 which focuses on K and λ's values so that an edge that's violated at the beginning has $\frac{1}{2}$ of being satisfied at the end of one complete recursion of the strongly connected components method. Furthermore, it is guaranteed that an edge that is initially satisfied, will stay satisfied. The last lemma that I want to mention is the *Lemma* 5.9 which states that only vertices that are strongly connected will get contracted by this method.

As conclusion, there are mentioned two future research paths in this direction. The first one deals with finding if there is any other $c < 5$ so that the I/O cost will remain $O(sort(E)log^c(V))$, while the second path would be looking if there is a possible way to eliminate the random factor from the algorithms presented in this paper.