Theoretical Computer Science ••• (••••) •••–•••

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Sorting an array using the topological sort of a corresponding comparison graph

Balaram D. Behera

Computer Science & Engineering Department, University of California, Santa Cruz, United States of America

A R T I C L E   I N F O

Article history:
Received 28 July 2020
Accepted 5 September 2020
Available online xxxx
Communicated by D.-Z. Du

Keywords:
Graph algorithms
Topological sort
Sorting algorithms
Comparison graphs

A B S T R A C T

The quest for efficient sorting is ongoing, and we will explore a graph-based stable sorting strategy, in particular employing comparison graphs. We use the topological sort to map the comparison graph to a linear domain, and we can manipulate our graph such that the resulting topological sort is the sorted array. By taking advantage of the many relations between Hamiltonian paths and topological sorts in comparison graphs, we design a Divide-and-Conquer algorithm that runs in the optimal $O(n \log n)$ time. In the process, we construct a new merge process for graphs with relevant invariant properties for our use. Furthermore, this method is more space-efficient than the famous MergeSort since we modify our fixed graph only.

1. Introduction

Most sorting algorithms run a multitude of array comparisons, and from those results we decide how to manipulate the elements to eventually achieve a sorted ordering of elements. This process can be implemented and infused with all kinds of data structures. In particular, directed graphs are a great way to structure this problem, and this allows us to look at sorting in different light and realize new methods of sorting.

We can represent every element as a vertex and the result of every comparison as an arc. Thus we can construct a graph that essentially stores all the comparisons made. In fact, we can mathematically represent these comparisons as an order relation: construct an arc if and only if the origin is less than the terminus (in the case of distinct array elements). Now we must decipher such a graph, i.e. to find some meaning to all those arcs that we have created. Our end goal is to achieve a sort of our input array, and in this paper we plan to achieve this using the topological sort of the graph (defined later).

We will employ basic ideas of graph theory, the Depth-First Search algorithm, and the topological sort to efficiently sort an array using directed graphs. We will first explore a somewhat trivial way to solve this problem, and then build a more efficient algorithm that will give us an equivalent result.

2. Previous work

Rajat K. Pal in his first paper on this topic discussed his RKPianGraphSort algorithm which was a perfect-graph-based sorting algorithm which was more extensive than what we present here and it ran in time $O(n^2)$ [3]. In his second paper [2], Pal introduced a complete-graph-based sorting algorithm which we develop here, calling it the Trivial Algorithm, yet this algorithm also runs in $O(n^2)$ time [2].

E-mail address: bbehera@ucsc.edu.

https://doi.org/10.1016/j.tcs.2020.09.004
0304-3975/© 2020 Elsevier B.V. All rights reserved.

The main limitation in previous work has been the complete graph construction required to employ the topological sort effectively, so in this paper we counter that with our intermediary merge processes to enhance the run time to reach the optimal $O(n \log n)$. In the buildup to the novel GraphSort algorithm, we go through the TrivialGraphSort algorithm which has been expounded by Pal [2]. This assists in the building of the required theory which has a different approach to Pal's as we go in the direction of Hamiltonian paths imperative in the development of the improved algorithm.

3. Definitions

Let us define a few terms that will be used frequently throughout the course of this paper. First let's define the family of graphs we specifically are working with.

Definition 1. Let $G = (V, E)$ be a directed graph with vertex set V and edge set E. Let $V = \{1, \dots, n\}$ where n is the order. Let the following set $A = \{a_1, \dots, a_n\}$ represent the values of their corresponding vertex. Let the edge $(u, v) \in E$ for vertices u and v if and only if $a_u < a_v$. Then the graph G is a comparison graph.

Remark. We often consider the corresponding set A as the given array itself.

Let's define the topological sort which is the basis of this entire paper.

Definition 2. Let G be a directed acylic graph with order n. Let $S = (s_1, \dots, s_n)$ be a sequence of all vertices such that for all $1 \le i \le n$, the vertex s_i is not adjacent to vertices s_k such that $1 \le k \le i$. Then the sequence S is a topological sort of graph G.

Remark. The topological sort of a graph is not necessarily unique.

A term we will use to evaluate how close we are to achieving a directed acyclic graph with a unique topological sort is trueness.

Definition 3. Let G be a directed acyclic graph, and let S represent a topological sort of G. The number of elements in S that are not fixed, i.e. their immediate adjacent neighbors (previous and next element) are not unique, is the trueness of G, denoted by $\tau(G)$.

Remark. Note that the trueness is not a count of how many topological sorts (that's more of a combinatorial extension), rather we count the number of elements in the sequence that are not fixed.

Remark. A graph with only one topological sort has a trueness of one, i.e. $\tau(G) = 1$ as all vertices are fixed.

Remark. In general, we focus on improving the trueness, i.e. to decrease $\tau(G)$ rather than evaluate it.

Since we will be working with arrays and array comparisons, we may want to compare an element of an array with some of its neighbors, so we have the following definition.

Definition 4. Let A be an array of size n, and let $1 \le i \le n$. Let r be the comparison reach (or just reach) of A. Then we compare $A[i]$ only to $A[i-r], \dots, A[i-1]$ and $A[i+1], \dots, A[i+r]$. Note if an array index is out of bounds, we wrap it around the array, i.e. $A[-1]$ means $A[n-1]$.

Remark. In other words, we wish to compare an element $A[i]$ to the closest $2r$ neighbors (equally distributed on both "sides").

Now we define a comparison graph called the corresponding graph that is a graph constructed from an array and some chosen array comparisons.

Definition 5. Let G be a null directed graph, i.e. $V = \{1, \dots, n\}$ and $E = \varnothing$. Let A be an array, and r the comparison reach of A. Then add (i, j) for all vertices i and j such that $A[i] < A[j]$ and such that $j \in [i-r, i+r]$ but $j \neq i$. Then G is a corresponding graph of A of reach r.

Remark. A corresponding graph is a comparison graph, where our value set A is the corresponding array. The corresponding graph is simply a specific structure of comparison graphs.

Remark. The initial ordering of the array can generate different corresponding graphs, and different reach values also may generate different corresponding graphs.

Lastly, we define the directed Hamiltonian path which is well-known, but we will repeat the definition to aid with upcoming theorems.

Definition 6. Let G be a directed graph. Let P be a path such that P includes all the vertices of G. Then P is called a directed Hamiltonian path in G.

These definitions will come in play during the development of theory in this paper. Moreover, we will frequently use basic graph theory definitions of trees, forests, connectedness, acyclicity, etc. which the reader is assumed to know.

4. Preliminaries

Before we start diving into the algorithms let's note some important preliminary remarks, assumptions, theorems, data structures, and algorithms that will be used in this paper.

Concerning the reach of an array, we will mainly study algorithms with a reach of one or for the sake of simplicity. Constructing corresponding graphs with higher reach values definitely generates more complex graphs, but as we will see later, the algorithms concerning those values may be more efficient. Also, we will study the total reach which is a reach value equal to the length of the array, and this will be used for the trivial algorithm.

Throughout the paper all the theorems stated apply to both comparison and corresponding graphs in most cases. The only difference between the two is that the corresponding graph has a defined reach whereas the comparison graph has no such defined structure.

It's important to note that for the sake of proving correctness, we assume all arrays have distinct values (later we will discuss what happens when this is not the case).

Since we will be using the topological sort of graphs, let's first show that the graphs we are working with in fact do have topological sorts.

Theorem 1. *If G is a comparison graph, then G is acyclic. Hence G is a directed acyclic graph.*

Proof. Let G be a comparison graph of order n. We must show that G is acylic. To the contrary, assume G contains a cycle, call it C. Then for $k \leq n$ as the length of C, let $C = (1, \ldots, k, 1)$ where i (for $1 \leq i \leq k$) is a vertex in G and every two adjacent vertices in C is a directed arc in C. Let a_i also represent the corresponding value for every vertex in G. Then by the definition of a comparison graph, we have

$$a_1 < \ldots a_k < a_1$$

which is an obvious contradiction. Therefore, G is acyclic, i.e. G is a directed acylic graph. □

We can now easily derive the following corollaries.

Corollary 1. *If G is a comparison graph, then G has a topological sort.*

Proof. Let G be a comparison graph. By Theorem 1, G is a directed acylic graph. Then by definition, G has a topological sort. □

Corollary 2. *If G is a corresponding graph of array A with any valid reach r, then G is acylic and has a topological sort.*

Proof. Since every corresponding graph is a comparison graph with the value set being the array A, the graph G necessarily has a topological sort following immediately from Theorem 1 and Corollary 1. □

Although we can construct algorithms dealing simply with the abstract mathematical definitions of graphs, to implement these algorithms in practice, it's imperative to construct useful data structures to represent graphs. We will use the adjacency list representation of graphs where every vertex has a list of vertices adjacent to it. This data structure is very memory efficient and easily accessible. In fact, we will be choosing certain orientations of this list to best cater to our needs in our procedures.

We will be employing Depth-First Search (DFS) all throughout this paper. It is assumed the reader has firm knowledge of this procedure and understands its applications. In summary, DFS means to travel deep from a root vertex till we can only travel to a vertex that has already been visited which is when we start back-tracking to parent vertices to continue on a different traversal. Once we back-track to the root itself, we have discovered that part of the graph and we do the same process for some other unvisited root. In this way we traverse the entire graph. The run-time of DFS on some graph G is $O(n+m)$ where n and m are order and size of G, respectively. While running DFS we keep track of parent vertices and start and finish times, where the start time is the discrete time at which we visit a vertex and a finish time is the discrete time at which we finish visiting all its children. The following theorem is the main reason DFS is relevant to our purpose.

Theorem 2. *Let G be a directed acyclic graph. Run Depth-First Search (DFS) on G, and generate a stack S where the top has the greatest finish time and the bottom has the lowest finish time (ordered by decreasing finish times). Then that stack S is a topological sort of G [1].*

We will take this theorem as is without proof [1]. This theorem is the backbone of many of the algorithms we will study further in this paper.

Lastly, we will call the components of a directed graph as the components of the underlying graph. Hence, there is no conclusion about weak or strong connectedness involved.

5. Construction of the corresponding graph

The basis of sorting using graphs is to first construct the graph. The definition of the corresponding graph given before essentially lays out our method of construction. Note that algorithms may vary immensely depending on the reach, so we fix the comparison reach for every algorithm. However, to preserve generality, consider r to be the reach of an array A of n elements. We will use the adjacency list representation for our corresponding graph G of reach r on A. By definition, we will be comparing $A[i]$ with $A[i+k]$ for all indices $1 \leq i \leq n$ and all k in the range $1 \leq k \leq r$ (not if $i+k > n$, then we simple take $i+k$ modulo n). In this way we will loop through every single index i and make the necessary comparisons. Note that although a reach r denotes that we compare an element to the closest r neighbors to the left and r neighbors to the right, it is equivalent in practice to simply compare just the r right neighbors or the r left neighbors.

However, before we implement a process to construct the graph, it's important to discuss the process of adding an arc. We use the adjacency list operation, and we can choose for the list to be sorted, reverse sorted, or completely arbitrary (similar to a set). By sorted, we mean with respect to the given array values. Of course it benefits us that it is sorted, since we will see that we normally want to traverse to the next smallest valued vertex. We achieve this sorted invariant by inserting every new vertex in the correct position in our adjacency list so it remains sorted.

There are two strategies for this: a binary search inspired insert or an insertion sort inspired insert. For the binary search inspired algorithm, the basic idea is that we check the middle of the list and determine which half the new vertex should be in, and in this way we continue till there is only one spot. This has the condition that it requires a fast way to access adjacency list elements. On the other hand, the insertion sort inspired algorithm where the basic idea is to go through the list from the first to last element and find where the new vertex should fit. This algorithm is more versatile and compatible with the general list data structure, so use this. Moreover, the asymptotic run-time for our algorithms is not affected by this choice.

5.1. Pseudo-code

Let's first see the process of adding an arc using the insertion sort inspired algorithm (note this is completely independent of our construction process, it simply acts as a method of our graph data structure). Let G be our graph, A the array that has the values of the corresponding graph G, and we want to add the arc (u, v). Our adjacency list is represented by an array of doubly-linked lists adj[] with general double-linked list and array functionality. The following is the pseudo-code for this process.

```
function AddArc($G, A, u, v$)
    $w = \text{adj}[u]$.front
    while $\text{adj}[u]$.index $> 0$ do
        if $A[v] < A[w]$ then
            Insert(adj[$u$], $w$.index, $v$)
            $G$.size $++$
            return
        if $\text{adj}[u]$.index $= \text{adj}[u]$.length then
            Append(adj[$u$], $v$)
            $G$.size $++$
        $w = w$.next
```

This particular function will keep the adjacency list sorted in ascending order of values. This configuration can vary depending on what is required for a parent process.

To construct the graph, we will add an arc depending on the value of the two elements being compared. Let A be an array of n elements, and r the comparison reach for the corresponding graph G of A. The following is the pseudo-code of the algorithm described for the construction.

```
function ConstructGraph($A, n, r$)
    $G =$ null graph of order $n$
    for $i = 1$ to $n$ do
        for $k = 1$ to $r$ do
            $j = i + k \mod n$
            if $A[i] < A[j]$ then
                AddArc($G, A, i, j$)
            else
                AddArc($G, A, j, i$)
    return $G$
```

Note that we assume A contains distinctly valued elements. Further this process is quite simple as it always does the required number of comparisons.

5.2. Run-time

For adding an edge, the worst case is when we must go through the entire adjacency list and then choose to add the vertex to the end, since we need to compare the new vertex with all the other elements. Let a be the maximum length of the adjacency list that we wish to add to, then our worst-case run-time for adding an edge is $\Theta(a)$. Note that normally the worst run-time is $\Theta(n)$ where n is the order of graph, but for our sake we will keep the run-time in terms of a.

The construction of the graph is also a very simple process. By definition, every vertex can only be adjacent to a maximum of $2r$ vertices since that is our comparison reach. Thus, we let $a = 2r$, i.e. maximum length of any adjacency list. Since we use the compare-right-only method, for every element we add r arcs. Lastly, we have n elements, and for each one we follow this method. In summary, we have a naive run-time of $O(n \cdot r \cdot 2r)$ which equates to $O(nr^2)$ which is quadratic in terms of r and linear in terms of n.

6. Properties of the topological sort

Before we discuss the core algorithms, let's first explore the properties and applicability of the topological sort for our sorting problem. Firstly, let's show that the existence of the topological sort of the corresponding graph being an actual sort of the array.

Theorem 3. *Let G be a corresponding graph of the array A. Then there exists a topological sort of G such that it is the sequence of indices of the sorted array of A.*

Proof. Firstly, there exists at least one topological sort in G by Corollary 2. Let A' be the sorted array of A. Let S be the sequence of indices in A such that the A-values for the these indices in A' are sorted. Then our claim is S is a topological sort of G. Assume to the contrary, that S is not a topological sort, i.e. it contains a back-edge (i, j) such that $i > j$ where $i, j \in S$, i.e. indices of A. By definition of a corresponding graph, if there exists an arc (i, j), then $A[i] < A[j]$, then since A' is sorted, we conclude $i < j$ which contradicts our hypothesis. Hence, there exists such a topological sort of G. □

We have the following immediate corollary that will be important in some time.

Corollary 3. *If there exists precisely one topological sort of the corresponding graph G of the array A, then that topological sort is the sequence of indices of the sorted array of A.*

Proof. From Theorem 3, we know there must always exist a topological sort of G that is the sequence of indices from the sorted array of A. Since there is only one topological sort, it must be such a sequence. □

Now let's work toward finding a case where there is only one topological sort.

Lemma 1. *There exists at most one Hamiltonian path, i.e. a path that visits every vertex, in any comparison graph (includes corresponding graphs).*

Proof. By way of contradiction, assume there exist two distinct Hamiltonian paths in comparison graph G called H and H'. Then there exists an arc $e = uw \in E(H) - E(H')$ for vertices u and v. Then there must exist an edge $e' = vw \in E(H') - E(H)$ for vertices v and w since every Hamiltonian path must have an arc for every vertex. Thus $A[u] < A[w]$ and $A[v] < A[w]$. Without loss of generality, assume $A[u] < A[v]$. Note the array A represents the A-values of G. Then in H, we cannot visit v after visiting w since $A[v] < A[w]$ and all vertices after w in H also have greater value than v. Thus we must visit v before we traverse the edge e which follows that we cannot visit u again with the same reasoning. This contradicts the hypothesis that we visit v before u. Hence there can exist at most one Hamiltonian path in G. □

Now we have enough material to show a relationship between a Hamiltonian path and the topological sort of some corresponding graph.

Theorem 4. *If there exists a Hamiltonian path in the comparison graph G (includes corresponding graphs), then there exists precisely one topological sort of G.*

Proof. Assume there exists a Hamiltonian path $H = (v_1, \dots, v_n)$ in the comparison graph G with corresponding values from array A. Let $S = (v_1, \dots, v_n)$, i.e. the vertices in H in the same order. Our claim is that S is a topological sort. By way of contradiction, assume not, i.e. there exists an arc $e = (v_i, v_j)$ where $i > j$. In H there is a path from v_j to v_i since $j < i$ by definition of a Hamiltonian path and a corresponding graph. Thus the arc e produces a cycle which contradicts Corollary 2. Hence, there exists a topological sort S of G if the graph contains a Hamiltonian path.

Now to show uniqueness, by way of contradiction, assume there exists another topological sort

$$S' = (v_1, \dots, v_k, u_{k+1}, \dots, u_n)$$

distinct from S where u_{k+1} is the first element that is different between the two sequences. So the value of u_{k+1} is greater than v_{k+1} because otherwise S would contain a back-edge. Then, $u_i \neq v_{k+1}$ for all $k+1 \leq i \leq n$, so $v_{k+1} \notin S'$ which contradicts our assumption of S' a topological sort since it doesn't contain a vertex. Therefore, there exists a unique, i.e. precisely one, topological sort of G as claimed. □

Thus we can show the big theorem for this section whose conditions we wish to satisfy through our algorithms.

Theorem 5. *If there exists a Hamiltonian path in the comparison graph G (includes corresponding graphs) with corresponding values from array A, then the topological sort of G is the sequence of indices of the sorted array of A.*

Proof. By Theorem 4, there is precisely one topological sort of G, and by Corollary 3 that topological sort is the desired sequence of indices as claimed. □

A Hamiltonian path in the comparison or corresponding graph implies that we have an arc from a vertex to the smallest greater value vertex, i.e. we have an arc between adjacent elements in the sorted array. Hence, we must either get to the point where we have a Hamiltonian path in the corresponding graph which makes it very easy to find the sorted array, or we develop a method to find all the topological sorts and find which one is the sorted array.

This paper will delve into the former strategy as we wish to use DFS. The DFS algorithm is integral to the first strategy because it gives an efficient method to finding the topological sort, and since there should be only one, we are done after running DFS.

6.1. Run-time of DFS

When we introduced DFS we mentioned its asymptotic run-time is $O(n+m)$ where n and m are the order and size of the graph G on which we run DFS on. Let G be a corresponding graph of array A with n elements and a reach of r. Then the order for G is also n. We know every vertex will be adjacent to or from precisely $2r$ vertices by definition of the reach. Thus by the degree-sum formula:

$$\sum_{v \in V(G)} \deg(v) = n(2r) = 2m$$
$$\therefore m = nr.$$

Thus the run-time of DFS on G, a corresponding graph, is $O(n + nr)$ which equates to $O(nr)$ since r is greater than one.

However it is important to note that if any comparison graph contains a Hamiltonian path and we start DFS at the minimum value vertex, we only have a run-time of $\Theta(n)$ for DFS which is the time it takes to traverse the Hamiltonian path after which we have visited every vertex as required.

7. A trivial algorithm

From Theorem 5 we know that if we can construct a corresponding graph such that it contains a Hamiltonian path, it is a trivial process to find the sorted array. The simplest method to achieve this is to construct the most complete corresponding graph, i.e. we compare every element of an array A with every other element of A. In this way, our graph necessarily must contain a Hamiltonian path; then we trivially we can determine the sorted array. So let G be a corresponding graph of array A of n elements with a reach of $r = \lfloor n/2 \rfloor$, i.e. we compare every element of A with the closest $2r \geq n-1$ neighbors (there are $n-1$ elements other than the current). This graph G is the most complete corresponding graph of A since we cannot add more arcs. Now we can run DFS on G starting at the minimum value element of A. Then we claim that the DFS stack is the sorted array of A.

7.1. Pseudo-code

Let A be an array of n elements. Let FindMin(A) return the index of the minimum value element of A that runs in linear time. Secondly let DFS(A, S) be the function that runs the DFS algorithm where S is the order by which DFS does its uppermost visits, and after the process is done S is the DFS stack. For us we will have $S = (x)$ where x is the index of the minimum value element. Lastly, the procedure ToArray(A, S) constructs an array from a list S of indices of A. Then we have the following pseudo-code for the trivial algorithm.

```
function TrivialGraphSort($A$)
    $n = A$.length
    $G =$ ConstructGraph($A, n, \lfloor n/2 \rfloor$)
    $x =$ FindMin($A$)
    $S = (x)$
    DFS($G, S$)
    return ToArray($A, S$)
```

Note that S is a stack, so that must be converted to an array technically. The trivial algorithm is not very complex as it simply fulfills the requirements of Theorem 5 to achieve the sort with the most naive strategy.

7.2. Correctness

Before we delve into proving the correctness of the algorithm, let's define a term we have casually used.

Definition 7. Let G be a corresponding graph. If we can't add more arcs to G, it is a complete corresponding graph, i.e. it is maximal with respect to arcs.

Now let's prove an important statement related to complete corresponding graphs.

Theorem 6. *Every complete corresponding graph G of array A of length n contains a Hamiltonian path.*

Proof. Let v_1 be the index of the minimum value element of A. Then let v_i be the smallest greater valued index than v_{i-1} for all $1 < i \le n$. Since G is a complete corresponding graph and since for every i we have $A[v_{i-1}] < A[v_i]$, the edge $v_{i-1}v_i$ is in G. Thus the path $P = (v_1, \dots, v_n)$ is in G and spans all vertices of G. Hence P is a Hamiltonian path in G. $\square$

Let's show something about our particular case in the trivial algorithm.

Proposition 1. *If G is a corresponding graph of an array A with n elements and reach $\lfloor n/2 \rfloor$, then G is a complete corresponding graph.*

Proof. By way of contradiction assume that there exists an arc $e = (i, j)$ that we can add to G where i and j are indices. Then there are at most $\lfloor n/2 \rfloor$ elements in between i and j in A since there are $n - 1$ elements besides i or j. Thus e is in G which is a contradiction. Hence G is a complete corresponding graph. $\square$

Although the design of this algorithm is simple to follow, let's formally prove the correctness of the trivial algorithm with the following theorem.

Theorem 7. *Let $A' =$ TrivialGraphSort(A). Then A' is the sorted array of A.*

Proof. In TrivialGraphSort, we first construct a corresponding graph G of A of n elements with reach of $\lfloor n/2 \rfloor$. Proposition 1 guarantees that G is a complete corresponding graph. Then by Theorem 6, we know G contains a Hamiltonian path. Then we run DFS starting at the minimum valued element which is an end-vertex of the Hamiltonian path, which gives us the topological sort which is the sequence of indices for the sorted array of A by Theorem 5. Then we get the actual array from ToArray which is necessarily the sorted array A' of A as claimed. $\square$

Thus we have completed showing that this algorithm indeed works where we sort an array using a corresponding graph and its topological sort.

7.3. Run-time

First we have $r = \lfloor n/2 \rfloor$, so to construct the graph we have a run-time of $O(n(\lfloor n/2 \rfloor)^2)$ by a previously stated formula, which equates to $O(n^3)$. Next to find the minimum, our function has a run-time of $\Theta(n)$. We have size

$$\begin{aligned} m &= (n-1) + (n-2) + \cdots + (1) \\ &= \sum_{k=1}^{n-1} (n-k) \\ &= n(n-1) - \frac{n(n-1)}{2} \\ &= \frac{n(n-1)}{2} \end{aligned}$$

for G. But, since G contains a Hamiltonian path and we start at the minimum value vertex, we actually have a run-time of $\Theta(n)$ for DFS in this case. Lastly to convert to an array it has a run-time of $\Theta(n)$ obviously. Therefore, in total we have a run-time of

$$O(n^3) + \Theta(n) + \Theta(n) + \Theta(n) = O(n^3)$$

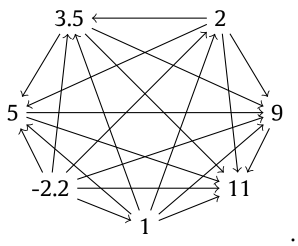
which is extremely inefficient since we also know the algorithm runs in time $\Omega(n^2)$ obviously too. However, this algorithm mainly provides us with base strategy that we can employ further in the paper.

7.4. Example

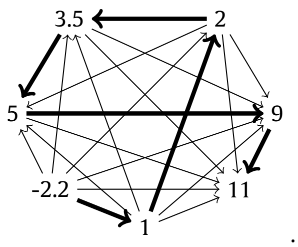
Let's review an example of the above described algorithm. Let,

$$A = [3.5, 2, 9, 11, 1, -2.2, 5].$$

Then our corresponding graph is,



To construct we take at most $7 \cdot \lfloor 7/2 \rfloor \cdot 7 = 147$ comparisons to construct the 21 edges of the above graph since we must also account for the account for the background work of adding to the adjacency list. Now we run DFS on our graph as specified for the algorithm (starting at the minimum element). The bold path represents the path of discovery:



To run DFS we witness that it takes only 6 edge and 7 vertex traversals since there exists a Hamiltonian path. Then our DFS stack is our sorted array:

$$A' = [-2.2, 1, 2, 3.5, 5, 9, 11].$$

Thus we have sorted the array as desired using the trivial algorithm. Notice that the most costly operation is constructing the graph which is bounded at a whopping 147 array comparisons.

8. Properties of the DFS forest

The DFS algorithm will visit vertices in some fashion and the edges it uses to traverse the graph can be collected into a forest.

Definition 8. Let G be a graph, and let $F \subseteq G$ be a graph with $V(F) = V(G)$ and every edge $e = uv$ is in F if and only if DFS on G traverses that edge. Then F is a resulting DFS forest of G.

Remark. We know every DFS forest is a forest because DFS visits every vertex once which allows for no cycles.

Remark. Note that if G is a comparison graph, then the resulting DFS forest F is also a comparison graph. In this way, the DFS forest of F is also a comparison graph.

It is assumed the reader has prior knowledge of DFS forests and forests in general. Now let's prove a point that is the basis of our next algorithm.

Theorem 8. *Let G be a comparison graph of array A, and let F be the resulting DFS forest of G. Then any path in F is a subsequence of indices from the sorted array of A (not necessarily consecutive).*

Proof. Since $F \subseteq G$, every arc $e = uv$ in F has the property that $A[u] < A[v]$. Let $P = (v_1, \ldots, v_k)$ be a path in F where v_i is a vertex for all $1 \leq i \leq k$. Then we have

$$A[v_1] < \ldots < A[v_k]$$

which is necessarily sorted. Thus the path P is a subsequence of indices from the sorted array of A. □

Note we will start abusing the notion of the connected component (or general component) by having it refer to the connected components of the underlying graph of some directed graph. For our next theorem, let's first introduce some new definitions.

Definition 9. A rooted tree is a tree with a distinguished vertex called the root such that the root has no incoming edges.

Remark. The root may also be the head of a Hamiltonian path in a comparison graph. This is just abuse of the traditional definition.

Definition 10. A leaf of a rooted tree is an end-vertex of the tree.

Definition 11. An internal vertex of a rooted tree is a vertex of the tree that is not the root or a leaf of the tree.

Remark. The connected components of a resulting DFS forest are all rooted trees.

Now let's show an important feature of corresponding graphs of a reach one and their resulting DFS forests.

Proposition 2. *Let G be a corresponding graph of array A and reach one, and let F be the resulting DFS forest of G. Let T be any component of F. Then every root of T has at most two outgoing arcs, every leaf of T has at most one incoming arc, and every internal vertex of T has one incoming and one outgoing arc.*

Proof. First note that every vertex of G has a total degree of two, i.e. the sum of the in-degree and out-degree by definition of a reach of one. Then since $F \subseteq G$, every vertex of F has a total degree of at most two. Let T be any connected component of F. Then since, by definition, roots have an in-degree of zero, every root of T can have at most two outgoing arcs. Since we visit every vertex once in DFS, no vertex in T has an in-degree of two, and by definition a leaf has no outgoing arc, so it can have at most one incoming arc. Since by definition, an internal vertex in T is not the root or a leaf of T, so it has at least one incoming and outgoing arc, and since our total degree is at most two, necessarily every internal vertex in T has one incoming and one outgoing arc precisely. □

Remark. Every rooted tree of F has at most two sub-trees stemming from the root, and those sub-trees are paths.

9. The merge process

We observed that in the trivial algorithm, the most expensive procedure was to construct the graph which took $O(n^3)$ time since the reach was dependent on n. If r is fixed at a constant value, the run-time for construction simply becomes $\Theta(n)$ since run-time is $O(nr^2)$. However, when we have constant r, the corresponding graph does not necessarily contain a Hamiltonian path. Hence, in this section we will develop a method of tackling this issue.

Let G be the corresponding graph of array A with constant reach r. Then by DFS on G we achieve a DFS forest F which contains k connected components. Now let's say we merge the components in a way such that new graph H and its resulting DFS forest F', have $\lceil k/2 \rceil$ connected components. This idea will be more refined into a concrete algorithm in the next section; right now, we just want to develop such a merge process.

There are many ways we can approach this problem, but we will particularly attempt at merging pairs of components to generate a component that contains a Hamiltonian path. This is easier to tackle with a reach of one since by Proposition 2, every tree in F has at most two sub-trees stemming from the root. Let a vertex v be the root of a tree T in F such that u and w are the neighbors of v. Then we cannot conclude anything about the order or values of the vertices in the sub-trees that stem from u and w, call them R and S. Hence, we should merge sub-trees of trees where they exist before we start merging the components themselves. By Proposition 2, both R and S are paths, and we wish to merge them into one graph H that is also a comparison graph such that we conserve the comparison edge property and H contains a Hamiltonian path. It is important that H contains a Hamiltonian path because then if we connect v to the beginning of the path in H, we achieve a connected comparison graph with vertex set $V(T)$ and a Hamiltonian path.

Note we will use the term Hamiltonian path for individual components of a graph too from now on than the traditional definition for the entirety of a graph.

The merge process will take advantage of the fact that $R = (a_1, \ldots, a_x)$ and $S = (b_1, \ldots, b_y)$ are paths. First we will compare a_1 to b_1 and add the corresponding arc between a_1 and b_1. Without loss of generality, assume a_1 has lesser value than b_1. Then we compare a_2 to b_1 and add their corresponding arc, and continue this process till we find some a_i that has greater value than b_1 for some $1 < i \leq x$. Then we compare a_i to b_2 and continue the process till we reach the end of either R or S. Let the resulting graph of these operations be H. Now we must show that H contains a Hamiltonian path to satisfy our requirements for our merge process.

Theorem 9. *Let T be a connected component of F, the resulting DFS forest of graph G. If T contains two sub-trees stemming from the root, merge the two sub-trees using the above process, then the merged graph, call it H, contains a Hamiltonian path.*

Proof. We will show that H contains a Hamiltonian path by construction. Let $v \in V(T)$ be the root of T, and let u_1 and w_1 be the two vertices adjacent from v. Let P be our path to which we will add arcs. Without loss of generality, assume u_1 has lesser value than w_1, so add the edge vu_1 to P. Let u_2 be the vertex adjacent from u_1 that is not w_1 (must exist because u_1 and u_2 are part of the same sub-tree of T). Then if u_2 has lesser value than w_1, add the edge u_1u_2 to P, else add u_1v_1 to P. In this way we construct our path by adding an arc from the current vertex to an adjacent vertex of minimum value. These "cross" arcs between vertices exist since we have merged in a particular fashion which allows for their existence. Hence our path P will eventually contain all the vertices, so it is a Hamiltonian path by definition in H as claimed. □

Remark. The construction of this path is analogous to how DFS would operate on H with a sorted adjacency list representation of H.

Thus we have merged the sub-trees of T and achieved a new comparison graph that contains a Hamiltonian path. Any tree in F that has a root with only one outgoing vertex is a Hamiltonian path itself. Thus the process of merging pairs of connected components boils down to merging paths, similar to above. However it is important to know how to traverse only the Hamiltonian path of the connected components which is where we take advantage of the sorted property of the adjacency list. We will now develop another merge process for a pair of components that contain a Hamiltonian path.

Let H_1 and H_2 be two components of H where every component of H contains a Hamiltonian path. Although we can find the Hamiltonian paths of both graphs by running DFS on both individually, let's simply start at the minimum valued vertices of both H_1 and H_2, call them a_1 and b_1. Now we compare the two and add a corresponding arc. Without loss of generality, assume a_1 has lesser value than b_1. Then let a_2 be the smallest value vertex adjacent from a_1 (similar to a recursive DFS visit), and compare a_2 to b_1 and add the corresponding arc. Continue this process till we reach some $a_i \in V(H_1)$ that has value greater than b_1. Then we compare a_i to b_1 and add their corresponding arc, and continue in the same fashion with $b_j \in V(H_2)$. We stop this process till we have no vertices adjacent from the current one in either H_1 and H_2. We have essentially followed the DFS strategy but added our corresponding arcs in the process similar to the merging of the sub-trees. Lastly we must show that this merge of H_1 and H_2 produces a graph that includes a Hamiltonian path.

Theorem 10. *Let F be a resulting DFS forest of G. Let $H = F$ if all components of F are paths, otherwise merge the sub-trees of F and let H be the result. Let H_1 and H_2 be a pair of two distinct components of H. Then if we run the above mentioned process, our resulting graph H' contains a Hamiltonian path.*

Proof. If $H = F$, then H_1 and H_2 are Hamiltonian paths, so let $P_1 = H_1$ and $P_2 = H_2$. Otherwise, by Theorem 9, we know H_1 and H_2 contains Hamiltonian paths, call them P_1 and P_2. Then traverse the graph similar to how we merge, i.e. we only "cross" between the two graphs when the direction of edges switches between to the two graphs. Thus we get a path P that starts at the minimum value vertex of both H_1 and H_2, and then from there we continue to the next minimum and so on. All these arcs exist due to our merge process. Hence our resulting graph H' contains a Hamiltonian path as claimed. □

Now let's summarize the merge processes. In the first, we run DFS on our corresponding graph G with reach one, and let F be the resulting forest then merge the sub-trees of the trees in F. And the second process is to merge consecutive components of H such that all the components of H contain Hamiltonian paths to gain a new graph called H' which contains approximately half number of components in H as desired. Also every component of H' contains a Hamiltonian path. Note in practice, H will either be the DFS forest itself or the graph after the sub-tree merge process on the DFS forest.

9.1. Pseudo-code

In our case, we have two merge processes actually: one that merges sub-trees and one that merges the components. Notice that both these processes simply merge on the Hamiltonian path starting at some root. In the case of the sub-trees we let the two roots be the neighbors of the root of the component. On the other hand, when we merge the components let the roots be the roots of the contained Hamiltonian paths. Again we will use the adjacency list representation. So the pseudo-code for this helper function is the following where H is a modified version of the DFS forest, $F \subseteq H$ resulting from corresponding graph G of array A. Note H is a comparison graph of array A.

```
function Merge($H, A, x, y$)
    $u = x,\ v = y$
    while $u \neq$ nil and $v \neq$ nil do
        if $A[u] < A[v]$ then
            AddArc($H, A, u, v$)
            $u = \text{adj}[u].\text{front}$
        else if $A[u] > A[v]$ then
            AddArc($H, A, v, u$)
            $v = \text{adj}[v].\text{front}$
```

Note x and y are the roots that we define for that specific tree. When we traverse to the "child" of the current vertex we are essentially traversing to the smallest value vertex adjacent from the current one.

Before we discuss the main merge procedures, let's first find the components of some DFS forest, specifically their roots. After running DFS on some graph G, all vertices in G with no incoming vertices will have no parent from DFS. Note the converse is also true since we start DFS from the minimum valued element which necessarily is not the terminus of any arc. Thus all the vertices after DFS that have no parent are the roots of the components of the DFS forest. The following is the pseudo-code for this algorithm given the resulting DFS forest F.

```
function FindRoots($F$)
    roots = ()
    for $v \in V(F)$ do
        if $v$.parent = nil then
            Append(roots, $v$)
    return roots
```

Note roots is a list. Also note that the list is in no specific order; it is completely arbitrary.

Now that we have our helper functions ready, we can delve into the actual merge process for a DFS forest F. First for every connected component of F we will merge the sub-trees if needed, i.e. if the root of the component has a degree of 2.

```
function MergeSubTrees($H, A$)
    roots = FindRoots($F$)
    for $r \in$ roots do
        if adj[$r$].length = 2 then
            $x = \text{adj}[r].\text{front}$
            $y = \text{adj}[r].\text{back}$
            Merge($H, A, x, y$)
```

Note the roots of the components of H remain unchanged.

Then we will merge consecutive components of $F \subseteq H$ where every component of H contains a Hamiltonian path. So we have the following pseudo-code for given H and corresponding array A.

```
function MergeTrees($H, A$)
    roots = FindRoots($F$)
    for $j = 1$ to $\lfloor \text{roots.length}/2 \rfloor$ do
        $p = \text{roots.get}(2j-)$
        $q = \text{roots.get}(2j)$
        if $A[p] > A[q]$ then
            Delete(roots, $p$)
        else
            Delete(roots, $q$)
        Merge($H, A, p, q$)
    return roots
```

We return the list of "new roots" in the merged graph H. It's important we get this list for our next algorithm where we need the specific list and the number of "roots." Note that by roots we mean the roots of the Hamiltonian path; a little abuse of terminology again.

9.2. Run-time

We will first analyze the run-time for the helper functions before analyzing the main merge process. Let T be a component of a DFS forest, and let T be of order n. Let T_1 and T_2 be two sub-trees of T. Let their respective orders be n_1' and n_2' where $n = n_1' + n_2' + 1$ (where the extra addition is for the root). To merge these two graphs according to our merge processes, we are essentially traversing the respective Hamiltonian paths which are of the same order as their respective graph. Every vertex can produce at most one arc originating from it because after we create that arc we traverse to another vertex. Hence, in the worst-case we produce an arc for every vertex (excluding the root), i.e. we produce $n_1' + n_2'$ arcs. Also since T_1 and T_2 have a maximum of one outgoing vertex (otherwise they would not be part of a sub-tree), the run-time to add an arc takes $\Theta(1)$. Hence to add $n_1' + n_2'$ arcs it takes time $O(n_1' + n_2')$ which is $O(n-1)$ which is the run-time of the sub-tree merge.

Now let's analyze the merging of any two components that contain Hamiltonian paths. Let H_1 and H_2 be these two components of order n_1 and n_2. In the worst-case every vertex in H_1 and H_2 has at most two outgoing vertices, thus to add an arc will still take $\Theta(1)$. Again the worst-case merge of these two components will be where we add an arc for every vertex, so we have a total run-time of $O(n_1 + n_2)$ in general.

To find the roots for a forest F we simply have a run-time of $O(|V(F)|)$ since it is only one loop, so it's linear with respect to the order.

Now for the main merge processes. Let k be the number of components of some DFS forest F of order n. In the worst-case, every component (tree) has two sub-trees, so in total we have a run-time of $O(n-k)$ which is also the number of arcs in the original F. Since $1 \le k \le n$, the merging of sub-trees takes $O(n)$ time.

Then we must merge consecutive components of some graph H (where every component of H contains a Hamiltonian path), and in the worst-case we merge to the fullest, i.e. we add every possible arc. Since we merge consecutive trees, if we have $n = n_1 + \cdots + n_k$ for n_i the order of component i of H $(1 \le i \le k)$, then to merge all the trees is simply

$$O(n_1 + n_2) + \cdots + O(n_{k-1} + n_k) = O(n)$$

assuming k is even (equal asymptotically when k is odd). Hence, in general the merging of components takes time $O(n)$ as well. Thus for both merge processes we take linear time to complete the procedures.

9.3. Example

Let's illustrate the merge process we have just discussed. Consider the following forest with just two trees:

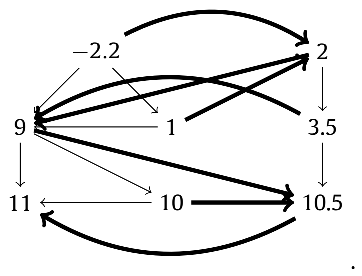
$$\begin{tikzcd}
 & -2.2 \arrow[dl] \arrow[dr] & & 2 \arrow[d] \\
9 \arrow[d] & & 1 \arrow[d] & 3.5 \arrow[d] \\
11 & & 10 & 10.5 \, .
\end{tikzcd}$$

First we will merge the sub-trees:

$$\begin{tikzcd}
 & -2.2 \arrow[dl] \arrow[dr] & & 2 \arrow[d] \\
9 \arrow[d] \arrow[drr, thick] & & 1 \arrow[d] \arrow[ll, thick] & 3.5 \arrow[d] \\
11 & & 10 \arrow[ll, thick] & 10.5 \, .
\end{tikzcd}$$

We add 3 more arcs to the forest to merge the sub-trees which is definitely under the upper bound of $8 - 2 = 6$ arcs (the number of vertices that are not roots).

We continue using the merged components from above; let's merge two components together:



We add 7 more arcs to our graph which again satisfies our set upper bound of $8 - 1 = 7$ arcs (the number of vertices subtracted by 1). Thus the size of our graph went from 6 to $6 + 3 = 9$ and then eventually to $9 + 7 = 16$ arcs. And these two steps complete the merge process for a forest of only two trees. Of course with multiple trees we just merge consecutive pairs of trees, using this process.

10. A divide-and-conquer algorithm

After the buildup of the merge process, it begs us to discuss a Divide-and-Conquer algorithm. Most importantly, Theorem 10 suggests that if we can generate a graph through a sequence of merges, we will end up with a graph containing a Hamiltonian path, and Theorem 5 asserts that the topological sort of that graph is in fact the sequence of indices of the sorted array. Thus our goal is to create an algorithm that generates such a sequence of merges.

Let A be an array n elements which we wish to sort. Also let G be a corresponding graph of an array A with reach one. Let F be the resulting DFS forest of G, and let F have k components which is in the range $1 \leq k \leq n$. We first merge the sub-trees of the components in F, and then we run the merge process on F to achieve a graph H_1 with $\lceil k/2 \rceil$ components, and note each component of H_1 contains a Hamiltonian path (locally). Then we run DFS on H_1 and let F_1 be the resulting forest which should also have $\lceil k/2 \rceil$ components (we run DFS on the roots of the components of H_1). Again we run the merge process on F_1 and get H_2 with $\lceil k/2^2 \rceil$. We continue this process till we achieve an F_i such that F_i is a tree, i.e. we have only one component. Let S be the topological sort of F_i which will be our sequence of indices from the sorted array since F_i is a Hamiltonian path (shown in the proof of correctness). Therefore, to sort the array using our algorithm it takes i merges, i.e. the number of components in F_i is exactly one.

This algorithm is definitely a Divide-and-Conquer algorithm. The division is the first run of DFS on the corresponding graph of the array. The combine and conquer part is the merge process which reduces the trueness of our comparison graphs till we reach a trueness of one, i.e. $\tau(H_{i-1}) = \tau(F_i) = 1$. Notice however that the number of merges depends on the number of components we have in F, the DFS forest of G, not the order of G. Yet k is bounded by n, so in the worst case we may achieve n components. Hence this algorithm can perform less computations for certain graph distributions; we will discuss the best-case run-time later in this section.

10.1. Pseudo-code

First we construct our corresponding graph G of array A of length n and reach of one. Then we must run DFS for the first time to get the resulting forest F (we assume DFS replaces the given graph, so technically $G = F$). Note to run DFS here, the order we visit the provisional roots can be arbitrary for the first time. However, for the latter runs we must define the visit sequence by the roots, so that in this way we actually find the components of the graph. Then we merge the sub-trees of F, and then merge the trees of F. We continue this process till we have one component only, i.e. a trueness of one (shown in proof of correctness), which is when we stop, and then we run DFS for the last time to get the topological sort. The following is the pseudo-code for this algorithm.

```
function GraphSort($A$)
    $n = A$.length
    $G =$ ConstructGraph($A, n, 1$)
    $S = (1, \ldots, n)$
    DFS($G, S$)
    MergeSubTrees($G, A$)
    roots = FindRoots($G$)
    while roots.length > 1 do
        roots = MergeTrees($G, A$)
        $S$ = roots
        DFS($G, S$)
    return ToArray($A, S$)
```

Note that our DFS algorithm will essentially replace the graph given with its resulting DFS forest. The dynamics with the DFS stack remain the same as the trivial algorithm. Also note that we need to merge the sub-trees only once (at the beginning). We will explore why we do this in the next part where we prove for correctness.

10.2. Correctness

Before we prove for correctness let's first prove why we only merge the sub-trees at the beginning.

Lemma 2. *Let G be a corresponding graph of an array A with n elements and reach of one. Let F be the resulting DFS forest of G. Then we merge the sub-trees and components of F, and let H be the resulting graph. Now run DFS on H with our visiting list as the roots of H, and let F' be the resulting forest. Then, all the components of F' are directed paths.*

Proof. First from Theorem 9, we know that after merging the sub-trees of the components of F, the components of the resulting graph H' contain a Hamiltonian path. Now let's merge the components of H' and let the resulting graph be H. By Theorem 10, every component of H contains a Hamiltonian path. Now let R be the list of new roots, i.e. the roots of the Hamiltonian paths of components in H. Then if we run DFS on H with our visit list as R, we discover each component by traversing down the Hamiltonian path since our adjacency list is sorted. Hence, our DFS forest F' is a collection of disjoint paths which correspond to the Hamiltonian paths of the components of H. Therefore, all the components of F' are directed paths as claimed. □

Corollary 4. *If H' be the graph after merging the components of F', then if we run DFS on H' on the roots, the components of the resulting forest are all paths.*

Proof. By Theorem 10, we know that the components of H' contain Hamiltonian paths, so if we run DFS on the roots of those paths, obviously the resulting DFS forest is a graph of disjoint paths as claimed. □

Hence, we see that sub-trees in a secondary DFS forest will never exist, so we may ignore merging them after the first run of DFS.

Also let's conclude some properties of sequences of merges on a particular forest along with DFS.

Lemma 3. *Let F be a comparison forest with k components where every component contains a Hamiltonian path. Then let H be the resulting graph after running the merge process (at the beginning both the sub-tree merge and component merge and later just the component merge) on F. Also if $k > 1$, H has fewer than k components. In fact, for any k, F has $\lceil k/2 \rceil$ components.*

Proof. First if $k = 1$, then after the merge process, our components cannot increase, so we still have 1 component in H. Also $\lceil 1/2 \rceil = 1$, so the claim holds for $k = 1$.

Assume $k > 1$, and let the components of F be $T_1, \dots, T_k$. Then the merge process essentially merges the components T_1 and T_2, and then T_3 and T_4, and so on. If k is even, the merge process will merge T_{k-1} and T_k too. Thus the number of components in H, the resulting graph, will be exactly $k/2$ which is equal to $\lceil k/2 \rceil$ since k is even. On the other hand, if k is odd, the merge process will merge T_{k-2} and T_{k-1}, but not T_k with any graph. Hence we do $\frac{k-1}{2}$ merges, so H has $\frac{k-1}{2} + 1$ components which is equal to $\lceil k/2 \rceil$ since k is odd.

Lastly, assume $k > 1$. If k is even, then obviously

$$\left\lceil \frac{k}{2} \right\rceil = \frac{k}{2} < k$$

as required. If k is odd, then

$$\left\lceil \frac{k}{2} \right\rceil = \frac{k+1}{2} < k$$

since $k > 1$, so $k/2 > 1/2$. Thus it holds that F' has fewer components than F if F has more than one component. □

From the previous lemma it follows immediately that we require a finite number of merges to reach one component.

Corollary 5. *There exists a finite sequence of merge processes and DFS runs such that from our original comparison forest F (of corresponding graph G) we will get a resulting a comparison graph T that will have one component, i.e. T is a tree.*

Proof. Assume F has more than one component, because otherwise F is the tree T. Let F have $k > 1$ components, and let F_1 be the resulting DFS forest of the merged graph of F. Let k_1 be the number of components of F_1. Then $k_1 < k$ by Lemma 3 since $k > 1$. Now generate the graphs F_i for $i > 1$ in the same way, and let k_i be the number of components of F_i. Then some $k_i = 1$ since the sequence $(k, k_1, \dots)$ is a strictly decreasing sequence until some $k_i = 1$. □

Now let's prove the correctness of our Divide-and-Conquer algorithm with the following theorems.

Theorem 11. *Let A be an array of n elements, and let G be the corresponding graph of A with reach one. Then let F be the resulting DFS forest of G. Then we merge the sub-trees of F to get a comparison graph H'. Now let H_1 be the resulting graph of the merge process on H', and let F_1 be the resulting DFS forest on H_1 with the visiting list as a list of the roots of H_1. Then let H_i be the resulting graph by the merge process on the forest F_{i-1} for all $i > 1$, and let F_i be the resulting DFS forest of H_i with the visiting list as a list of the roots of H_i. Then we claim that there exists a finite i such that F_i is a Hamiltonian path, and that the topological sort of F_i is in fact the sequence of indices of the sorted array of A.*

Proof. First by Lemma 2, we know that the components of F_1 are all paths. Now H_2 is the resulting graph of the merge process on F_1, and by Theorem 10 we know that the components of H_2 all contain a Hamiltonian path. Now we let the new roots of H_1 be the list R, and run DFS on H_2 with the visit list as R. By Corollary 4, the resulting DFS forest F_2 is a collection of disjoint paths, i.e. the components of F_2 are all paths. Now we run the same process with F_2 to generate F_3 and so on till some F_i such that F_i is a tree and that F_{i-1} contains more than one component. By Corollary 5, we verify the existence of such an i is finite. By the above defined process for generating F_j for some $1 < j \leq i$, all the components of F_j are paths. Hence, F_i is necessarily a path since it is connected as it has only one component which is necessarily a path. Therefore, F_i is a Hamiltonian path, and necessarily the topological sort of F_i is the sequence of indices of the sorted array of A by Theorem 5. □

Corollary 6. *If A is an array of n elements, then $A' = \textsc{GraphSort}(A)$ is sorted.*

Proof. Since the process of the algorithm is laid out by Theorem 11, it immediately follows that we attain the sorted array of A by calling GraphSort(A) as claimed. Also the loop in GraphSort will terminate, again by Theorem 11. □

This concludes our proof of correctness for our algorithm. Note we have only shown that the loop in the algorithm will terminate, and at that instant we have sorted the array. In the next section, we will discuss when that termination occurs.

10.3. Run-time

This analysis of the run-time will be a sequential analysis of the main procedures in GraphSort, and then combine them to gain a full upper bound on the run-time.

First recall that the run-time to construct a corresponding graph for an array of n elements is $O(nr^2)$ where r is the reach. Since our reach is one, our construction takes time $O(n)$ asymptotically.

Recall that the run-time to merge sub-trees is $\Theta(n)$ in the worst-case where n is the number of elements. On the other hand, also recall that to merge the components of a comparison graph it takes time $\Theta(n)$ also. Also to find the roots we have a time of $\Theta(n)$.

Now let's discuss the run-time of DFS. For the first run of DFS we do not know what our corresponding graph looks like, so we use the general run-time of DFS which is $O(|V(G)| + |E(G)|)$ where $G = (V, E)$ is our corresponding graph. Obviously we have n vertices in G because every element maps to a vertex. Further, since we have a reach of one, we have an out-degree of one for all vertices, so by the Degree-Sum Formula, we conclude $|E(G)| = n$ also. Hence the first run of DFS runs in time $O(2n)$ which is asymptotically equal to $O(n)$.

For the latter runs of DFS we assure that the given graphs contain Hamiltonian paths, and our visit list for DFS is a list of roots of those paths. Hence we never do an unnecessary check for a vertex visited or not in DFS, so our run-time is $\Theta(n)$ only.

Now let's compute the number of times we go through the main loop in our algorithm. In the worst-case, we can have n components in our first resulting DFS forest (this occurs only when the array is in reverse order). Then through our first merge process we generate the next forest with $\lceil n/2 \rceil$, and then the next forest with $\lceil n/2^2 \rceil$. This halts once we have one component, i.e. after some i iterations we must have,

$$\lceil n/2^i \rceil = 1$$
$$\therefore i = \lceil \log_2(n) \rceil.$$

Hence we are done merging and running DFS after $\lceil \log_2(n) \rceil$ iterations in the worst-case.

Now that we have computed the run-time of all the individual parts, it's time to combine them. Prior to the loop we first generate the corresponding graph which has time $O(n)$. Then we run DFS on that graph which runs in time $O(n)$. Then we merge the sub-trees which also has run-time of $O(n)$. And lastly to find the roots we run in $\Theta(n)$ too. Thus all these parts have a total run-time of

$$O(n) + O(n) + O(n) + \Theta(n) = \Theta(n)$$

which is linear. Then in the body of the loop, we take time $\Theta(n)$ to merge the components, and then $\Theta(n)$ to run DFS on them. So in total, the body takes time

$$O(n) + \Theta(n) = \Theta(n)$$

which is also linear. Also the loop in the worst-case runs $\lceil \log_2(n) \rceil$ times which is logarithmic. And finally, after the termination of the loop we convert our stack into the sorted array which takes time $\Theta(n)$. Thus in total, our Divide-and-Conquer algorithm runs in time

$$\Theta(n) + (\lceil \log_2(n) \rceil) \cdot \Theta(n) + \Theta(n) = \Theta(n \log n)$$

in the worst-case. In fact, $\Omega(n \log n)$ is the asymptotic lower bound of any comparison based sorting algorithm for worst-case. Hence our algorithm is asymptotically as efficient as any mainstream sorting algorithm.

In the best-case which is when we have a sorted array, our corresponding graph itself contains a Hamiltonian path, so by Theorem 5, we have already sorted the array essentially without the loop. Hence we have a run-time of $\Theta(n)$ in the best-case. Hence, in general our algorithm runs in time $O(n \log n)$.

Additionally, the exact run-time in terms of basic operations (comparisons), we have an approximate run-time (in terms of basic operations) of

$$2n \log_2(n) + 4n$$

with many other computational optimizations added to the existing algorithm which we won't cover extensively in this paper. One of the most efficient sorting algorithms called QuickSort runs in approximately

$$2n \ln(n) + 2 \ln(n) - 4n$$

time. So our algorithm is $\log_2(e)$ times slower than QuickSort which is approximately 1.44×. Thus our algorithm isn't computationally ground-breaking, although it can be inspiration for one. However we will discuss later some advantages of such an algorithm in practical applications.

10.4. Example

Consider the same array from before:

$$A = [3.5, 2, 9, 11, 1, -2.2, 5].$$

Then, we first make our corresponding graph of reach one:

$$\begin{tikzcd}
3.5 \arrow[rrrrrr, bend right] & 2 \arrow[l] \arrow[r] & 9 \arrow[r] & 11 & 1 \arrow[l] & -2.2 \arrow[l] \arrow[r] & 5
\end{tikzcd}.$$

We add 7 edges to construct the corresponding graph. We will run DFS in the following order starting at -2.2, the minimum valued vertex. Now we run DFS for the first time (bold lines denote path of discovery):

$$\begin{tikzcd}
3.5 \arrow[rrrrrr, bend right] & 2 \arrow[l] \arrow[r, very thick] & 9 \arrow[r] & 11 & 1 \arrow[l, very thick] & -2.2 \arrow[l, very thick] \arrow[r, very thick] & 5
\end{tikzcd}.$$

We traverse 4 arcs and visit all 7 vertices by running DFS. We generate a DFS forest with 3 trees with roots $(-2.2, 3.5, 2)$:

$$\begin{tikzcd}
 & -2.2 \arrow[dl] \arrow[dr] & & 3.5 & 2 \arrow[d] \\
1 \arrow[d] & & 5 & & 9 \\
11 & & & &
\end{tikzcd}.$$

After the merge process on the above forest, we get:

$$\begin{tikzcd}
 & -2.2 \arrow[rr, thick] \arrow[dl] \arrow[dr] & & 3.5 \arrow[dl, thick] & & 2 \arrow[d] \\
1 \arrow[urrr, thick] \arrow[rr, thick] \arrow[d] & & 5 \arrow[dll, thick] & & & 9 \\
11 & & & & & .
\end{tikzcd}$$

We add 5 arcs to merge which is under $(7-3)+7=11$ (the maximum number of arcs we can add) by the merge process. So our “new” roots are $(-2.2, 2)$ since the root 3.5 got merged. Now let’s run DFS again starting at the new roots (note the list of roots being sorted is irrelevant):

$$\begin{tikzcd}
 & -2.2 \arrow[rr] \arrow[dl, thick] \arrow[dr] & & 3.5 \arrow[dl, thick] & & 2 \arrow[d, thick] \\
1 \arrow[urrr, thick] \arrow[rr] \arrow[d] & & 5 \arrow[dll, thick] & & & 9 \\
11 & & & & & .
\end{tikzcd}$$

Here DFS traverses 5 arcs and visits 7 vertices. Thus we get the following DFS forest with 2 trees (paths in this case):

$$\begin{tikzcd}
-2.2 \arrow[d] & & 2 \arrow[d] \\
1 \arrow[d] & & 9 \\
3.5 \arrow[d] & & \\
5 \arrow[d] & & \\
11 & & .
\end{tikzcd}$$

Notice they both are Hamiltonian paths. Then again we run the merge process on this forest and our “new” root is (-2.2) which is the minimum of the array also:

$$\begin{tikzcd}
-2.2 \arrow[d] \arrow[r, thick] & 2 \arrow[d] \arrow[ddl, thick] \\
1 \arrow[d] \arrow[ur, thick] & 9 \arrow[dddl, thick] \\
3.5 \arrow[d] \arrow[ur, thick] & \\
5 \arrow[d] \arrow[uur, thick] & \\
11 & .
\end{tikzcd}$$

We add 6 arcs to our graph which is at the upper bound of $7-1=6$ (number of vertices subtracted by 1). Now we run DFS for the last time starting at the root -2.2 (the minimum value vertex):

$$\begin{tikzcd}
-2.2 \arrow[r] \arrow[d, very thick] & 2 \arrow[d] \arrow[ddl, very thick] \\
1 \arrow[ur, very thick] \arrow[d] & 9 \arrow[dddl, very thick] \\
3.5 \arrow[ur] \arrow[d, very thick] & \\
5 \arrow[uur, very thick] \arrow[d] & \\
11 &
\end{tikzcd}.$$

Here since the graph contains a Hamiltonian path we have 6 edge traversals and visit 7 vertices. Also our DFS stack is our sorted array as desired:

$$A' = [-2.2, 1, 2, 3.5, 5, 9, 11].$$

Thus, we complete our Divide-and-Conquer algorithm. Although visually the process is more extensive, its run-time is much more efficient than the trivial algorithm.

11. Resemblance of MergeSort

The MergeSort algorithm is a quintessential Divide-and-Conquer algorithm, and we will explore some similarities between MergeSort and GraphSort.

Firstly the merge process for our algorithm is so similar to MergeSort's merge process, except in GraphSort we traverse a path, whereas in MergeSort we traverse sub-arrays. However, one big difference is that in MergeSort we equally divide the array into sub-arrays of length one and build them up from there. In GraphSort, we generate a graph with a certain reach and then after running DFS our "building blocks" are essentially the components of the DFS forest. Our components have no fixed size which differs from MergeSort which is very organized and structures. This is in fact an advantage of using graphs since we have to perform less unnecessary overhead operations. Lastly, in MergeSort we keep an invariant on sub-arrays to always be sorted after a merge, and in our algorithm the invariant is essentially Theorem 8. This is in fact equivalent since we are dealing with comparison graphs which are essentially graphical depictions of arrays which is what our corresponding graph tries to accomplish actually.

These are the most apparent and relevant similarities between the two. In fact, we see that we can mimic MergeSort with our graphical strategy.

11.1. A graph version of MergeSort

In MergeSort we build up from sub-arrays of length one, then two, then four, etc. So let's generate our corresponding graph as such: we generate an arc between consecutive pairs of elements using our same comparison property. In this way every component after the first run of DFS has exactly 2 vertices (except one if we have an odd number of vertices). Thus we have $\lceil n/2 \rceil$ components exactly in our first DFS forest. Then we can continue the original GraphSort algorithm with the same merging process to sort the given array. Notice the only difference is that our components are fixed at $\lceil n/2 \rceil$ rather than an arbitrary k in the range $1 \leq k \leq n$. Let's call this algorithm GraphMergeSort.

11.2. Run-time

The worst-case for our algorithm was when we had n components in our first DFS forest, but with GraphMergeSort, we guarantee $\lceil n/2 \rceil$ for any instance, i.e. we guarantee only one less iteration of the loop in the worst-case. Hence our total run-time adapted from the run-time analysis for our algorithm is

$$\Theta(n) + (\lceil \log_2(n) \rceil - 1) \cdot \Theta(n) + \Theta(n) = \Theta(n \log n).$$

Hence, GraphSort and GraphMergeSort have equal efficiency asymptotically, but one just guarantees a certain number of iterations of the loop for our algorithm and one depends on an arbitrary integer at most n.

11.3. Example

Lastly, we analyze the GraphMergeSort algorithm which merely differs in the first corresponding graph construction. First consider the following array again:

$A = [3.5, 2, 9, 11, 1, -2.2, 5]$.

Then our corresponding graph is the following:

$$\begin{tikzcd} 3.5 & 2 \arrow[l] & 9 \arrow[r] & 11 & 1 & -2.2 \arrow[l] & 5 \end{tikzcd}.$$

We only add 3 arcs this time which is approximately half the number of array elements. Then we continue the same process as for the previous algorithm, and again with our last run of DFS, the stack gives us the sorted array:

$A = [-2.2, 1, 2, 3.5, 5, 9, 11]$.

This completes the GraphMergeSort example.

12. Practical implementation

In this section we will focus less on the theory, but more on the implementation of the theory discussed throughout this paper. This is imperative to convey since graphs can be represented by many data structures, so understanding which methods are efficient. Furthermore, we can implement multiple computational optimizations to reduce the memory load and operations required. These were mostly skipped in the development of the theory aspect since they add unnecessary complexity and do not affect the conclusion vastly. We will also discuss how we can tackle equal-valued elements in an array which hasn't been discussed yet since we have assumed distinct values up till now for the ease in proving correctness. Also, we will explore some machine dependent issues that we could face and how we can tackle them. Lastly we will examine practical applications of this algorithm and what advantages it brings to the table relative to other efficient sorting algorithms out there.

12.1. Data structures

As we have used throughout the pseudo-code sections of the paper, we will use the adjacency list representation of a graph. This is imperative since we traverse through vertices rather than probing on edges as a whole, so quickly accessing neighboring vertices is important. Moreover, we require our adjacency list to be sorted according to corresponding value for both our algorithms. We can implement this sorted invariant as shown before by how we add arcs to the graph.

Graphs are a combinatorial mathematical structure, so it is important we remain memory efficient in our representations. Since every vertex has a limited out-degree for our Divide-and-Conquer algorithm for the least, an adjacency list for every vertex is thus more memory efficient. We take $\Theta(n+m)$ space to represent a graph of order n and size m.

12.2. Memory optimizations

Following our focus on memory efficiency, when we run DFS for the Divide-and-Conquer algorithm, we have DFS generate our forest and replace the given graph with the resulting forest. However, we do not necessarily need to run secondary DFS operations; we included it primarily to reduce complexity in proofs. Since when we merge we essentially traverse the contained Hamiltonian paths of the components, any overhead of arcs in the component will not affect the merge traversal. Thus, we eliminate the middle uses of DFS; we still need to run DFS on the corresponding graph and at the end to gain the topological sort. Note this optimization comes with a grain of salt, since now our graph data structure will be larger in terms of memory, but we limit alterations in the data structure as we use only one rather than many (the intermediate forests).

In order to optimize the construction of the forest separately, we can also implement a strategy where we remove edges from the given graph while running DFS. We won't go into detail for a game plan to accomplish this, but leave it to the reader to conceive. In the same discussion of space, we can alter our method of adding such that we delete those unnecessary arcs simultaneously. This can be done in situations where have a list of consecutive vertices that all create an arc to one vertex in the other component when we are "moving" down a component in the process of merging. So instead of our process of generating an arc and then "moving," we simply compare the next vertex we would "move" to with the vertex we would add an arc to, and if we know we will "move," we skip the addition of that arc. This will reduce the number of "cross"-arcs we have after merging, but it is much harder to also reduce the number of original arcs that already existed but are deemed unnecessary after merging. Moreover, this increases the number of comparisons to be made during merging too. We leave the implementation of a solution to the reader for the latter more difficult part.

12.3. Introducing randomness

In QuickSort if our pivot choice is not random, let's say we choose the last element, then it's easy to construct an array that will force QuickSort to run in worst-case time. However we tackled this by choosing a random pivot element. Similarly, our algorithm highly depends on the number of components in our first DFS forest. As we saw in the worst-case we may end up with a situation of n components. Our algorithm has a DFS visit ordering that is arbitrary (in the first run), and for simplicity we just go from the first element to the last in order. However, if we implement a completely random order we may achieve a more consistent run-time overall.

If we assume that a random visit ordering implies an equal probability that we have either $k = 1, \dots, n$ components, then in the average-case we have $n/2$ components which is similar to our MergeSort version. However this implication is probably false since DFS will first visit all the reachable vertices from where it started, and the fashion it reaches those vertices and which vertices are visited in the process can alter the upcoming DFS visits. Hence we save this uncertainty with randomness and the average-case run-time analysis for another time.

12.4. Dealing with equal value elements

One of the biggest issues we have yet to discuss is what does the algorithm do for equal value elements. All the theorems and algorithms in the paper as of now assume that our given array contains only distinct values. This is important to satisfy our comparison property which is a strict inequality. In the case of equal-valued elements, we compare the indices of those elements in the given array to determine which direction an arc between the two elements will go. Since indices are necessarily distinct, we will never face an issue there, and having this second condition for equal-valued elements will still produce an order relation for our comparison property. Note this issue is purely computational and does not hinder with the basic algorithmic process or theorems; we simply modify our ordering definition for a comparison graph.

12.5. A system bottleneck

An issue witnessed during an implementation of this algorithm on massive arrays was that DFS would fail midway. This is because DFS is recursive, and from the second DFS run and onward (in the Divide-and-Conquer algorithm) we are running DFS on disjoint paths essentially which leads to very deep recursions which may lead to a stack overflow on machines with limited memory. We will discuss an iterative version of DFS solely for scenarios where we are dealing with forests of components that contain Hamiltonian paths, and in fact we notice that we can translate this for our first DFS run.

12.6. An iterative DFS solution

We will now discuss an iterative method of DFS for our particular case. We assume that we are running DFS on a comparison graph G such that every component of G contains a Hamiltonian path. Also assume we have a set of the roots of those Hamiltonian paths called R. Recall that DFS will visit some start vertex, and then visit a vertex adjacent to the start, and so on. Once all neighbors of the current vertex have already been visited, we “back-track” to its parent and check for the same. Once we have reached the start vertex, and we have no adjacent unvisited vertices, we stop DFS starting from that root. Then we let our new start be the next unvisited vertex in our visit list given and continue the same process till we visit all the vertices.

The biggest problem with this algorithm is that with our convention of the adjacency list, it becomes unnecessary to back-track as we have visited all the vertices in a component once we have the need to back-track since every component of G contains a Hamiltonian path. Therefore, an iterative solution would be to start at some vertex $x \in R$ and then continue to the first vertex y in the sorted adjacency list of x, and then continue the first vertex z in the sorted adjacency list of y, until we reach a vertex w where w has no adjacent vertices. Then it is sufficient to conclude that we have discovered the Hamiltonian path starting at x for the component since G has no back-edges since it is a comparison graph. We run this iterative process for all vertices in R, and the paths we generate are equal to the DFS forest as required.

Note we can only do this when we know every component contains a Hamiltonian path, so this can replace all the DFS runs after the first. This can also translate to our first DFS run since we have shown that only roots of the trees in the resulting DFS forest may have two vertices adjacent to them. Then we can start this iterative process whenever we start discovering the respective provisional sub-trees which we have shown are paths. Thus we eliminate the recursive nature of DFS for our particular algorithm which allows for more versatility on machines with limited stack sizes.

12.7. Graphical approach advantages

During the discussion of the run-time of our algorithms, it was mentioned that computationally, our algorithm still lacks to perform as well as QuickSort, the leading sorting algorithm right now. Also memory-wise, we construct a separate data structure to sort which adds extra overhead and cost. However, there do exist some advantages of using the graphical approach.

Firstly, if there is a situation where we wish to sort a comparison graph itself, which matches our algorithm’s objective essentially, we can employ our merge techniques and DFS to achieve a more true comparison graph than the original. This eliminates the process of converting a graph into a linear structure to sort using a general sorting algorithm. Now we do not need to transform the input as we can modify the graph itself. Many applications use comparison graphs and directed graphs to represent networks and what not, and our algorithm provides a way to better detail those graphs and rank the nodes of those networks in an efficient manner.

Secondly, if the array distribution contains long increasing sub-sequences, our algorithm can take full advantage of this feature which algorithms like QuickSort and MergeSort fail to achieve. Furthermore, some tailing computations as in other sorting algorithms can be eliminated since an edge can concatenate two "sub-arrays." Additionally, since our best-case run-time is $\Theta(n)$ which is when the given array is already sorted, our algorithm performs extremely well in partially sorted arrays, similar to InsertionSort. A complete analysis of the run-time in terms of inversions is not present in this paper and is saved for another time.

Lastly, some procedures presented in this paper can provide a better representation for some linear structures like linked-lists. We need not necessarily sort the input, rather we can provide those intermediate forests if they are sufficient for a user. This is useful in an ongoing insertion situation where more elements are being inserted and keeping the structure somewhat sorted is important. Then at the end we may complete the algorithm and sort the array. Note that the trueness of the forest will be approximately the same for every insert since fairly quickly we can choose where to insert the element.

13. Similar algorithm ideas

We will now discuss some ideas that may improve the algorithms discussed in this paper, and what parameters we can tweak to achieve vastly different results and uncover new problems.

First we can modify the reach values for our corresponding graphs. In GraphSort we define a fixed reach of one, but there may be different implementations of algorithms with different reach values. The higher our reach, the more complex our corresponding graph is, and the harder it is to parse and process the graph as graphs are combinatorial structures.

Additionally, we can implement a k-way merge process instead of our 2-way merge. It doesn't seem to better asymptotic run-time in some preliminary analysis, but in terms of computation we may achieve more efficient algorithms because the logarithm base would be larger. However, again implementing merges for multiple components turns into a very massive problem very quickly.

Moreover, we can expand on the idea of GraphMergeSort and instead of generating components by pairs of consecutive elements, we can generate components by triples of consecutive elements, or even higher k-tuples of consecutive elements. In this way, we first solve each of the components in a specific way and then continue with our merge process. This is a way to implement Divide-and-Conquer with the corresponding graph itself.

There are many conventions we have set for our algorithms, and we have absolute freedom to experiment with different conventions to realize newer algorithms employing comparison graphs.

14. Conclusion

In summary, this paper explored properties of comparison graphs, corresponding graphs, topological sorts, and DFS to fuse together procedures and algorithms that solve the old-age sorting problem. Our best algorithm ran in time $\Theta(n \log n)$ in the worst-case and $\Theta(n)$ in the best-case. Hence, it is on par with mainstream sorting algorithms although the graphical strategy seems more complex. Further, it serves extremely well for particular distributions of arrays, and even for less efficient distributions it's competitive with other such algorithms.

To remind you again, sorting an array is simply an application of the techniques provided here. There are many other applications of our procedures available with comparison graphs, in particular. Our merge process is probably the most important and eye-opening part of this paper as it gives us a way to generate comparison graphs that are more true. It retains all the properties and conditions of its original graph and generates a new one that is more detailed and concrete, and this procedure can be applied for solving many other problems.

This is the direct outcome of employing graphs since they provide another layer of complexity and information. The algorithms presented took advantage of these properties of graphs to develop an interesting method to sort although it may even render to other problems.

Hopefully, the algorithms, procedures, and theorems discussed may inspire you, the reader, to embark on a discovery of more applications and algorithms using the ideas discussed in this paper as a foundation.

Declaration of competing interest

I was and am not funded by any organization for my research. I have received the UC Regents Scholarship in my time as an undergraduate at UC Santa Cruz for only my tuition.

Acknowledgements

I would like to thank Professor Patrick Tantalo of the Computer Science and Engineering department at the University of California, Santa Cruz. His lectures and discussions inspired me to look into this method and write this paper.

I would also like to thank Professor Seshadhri Comandur of the Computer Science and Engineering department at the University of California, Santa Cruz. He mentored me throughout the latter stages of the paper to refine and polish it. Without his help I would not be able to finish this paper.

Further, the teaching of Nathan Marianovsky of the Mathematics department at the University of California, Santa Cruz, in graph theory gave me the perfect arsenal to formulate many of these ideas formally.

Lastly, I would like to sincerely thank Dr. Rajat K. Pal from the University of Calcutta for sharing insights of his previous work in this line of research and for kindly assiting me through the publishing process.

References

[1] Thomas H. Cormen, et al., Introduction to Algorithms, 3rd ed., MIT Press, 2009.
[2] Rajat K. Pal, CompleteGraphSort: a complete graph structure based sorting algorithm, in: 2011 IEEE International Conference on Computer Science and Automation Engineering, vol. 4, 2011, pp. 193–197.
[3] Rajat K. Pal, RKPianGraphSort, in: Ubiquity 2007, Jan. 2007, pp. 1–16.

Further reading

[4] Gary Chartrand, Ping Zhang, A First Course in Graph Theory, Dover Publications, 2012.
[5] Nathan Marianovsky, Graph Theory Lecture Notes, Canvas Class Files for UCSC Math, vol. 115, Mar. 2020 (online); accessed March 2020.
[6] Patrick Tantalo, Graph algorithms handout, https://classes.soe.ucsc.edu/cse101/Fall19/Handouts/GraphAlgorithms.pdf, Nov. 2019 (online); accessed December 2020.
[7] Patrick Tantalo, Graph theory handout, https://classes.soe.ucsc.edu/cse101/Fall19/Handouts/GraphTheory.pdf, Nov. 2019 (online); accessed December 2020.