Graph Algorithms in DROP

v4.79 15 April 2020

The Soft Heap: An Approximate Priority Queue with Optimal Error Rate

Abstract

1. Functionality Provided by Soft Heap: Chazelle (2000a) introduces a simple variant of a priority queue, called a *soft heap*. The data structure supports the usual operations; insert, delete, meld, and findmin.
2. Breaking the Information-Theoretic Bound: Its novelty is to beat the logarithmic bound on the complexity of a heap in a comparison-based model. To break this information-theoretic barrier, the entropy of the data structure is reduced by artificially raising the values of certain keys.
3. Error Rate and Implied Complexity: Given any mixed sequence of n operations, a soft-heap with an error rate ε - for any

   $$0 < \varepsilon \leq \frac{1}{2}$$

   ensures that, at any time, at most εn of its items have their keys raised. The amortized complexity of each operation is constant, except for insert, which takes $\mathcal{O}\left(\log \frac{1}{\varepsilon}\right)$ time.
4. Data Structures used by Soft Heap: The soft-heap is identical for any value of ε in a comparison-based model. The data structure is purely pointer based. No arrays are used and no numeric assumptions are made on the keys.
5. *Car Pooling* Groups of Data: A key idea behind soft heap is to move items across data structures not individually, as is customary, but in groups, in a data-structuring

equivalent of *car-pooling*. Keys must be raised as a result, in order to preserve the heap-ordering of the data structure.

6. Application Usage of Soft Heap: The soft heap can be used to compute exact or approximate medians and percentiles optimally. It is also useful for approximate sorting and for computing minimum spanning trees of general graphs.

Introduction

1. Soft-heap as a Priority Queue Variant: Chazelle (2000a) designs a simple variant of priority queue called a *soft-heap*. The data structure stored items in a heap with keys in a totally ordered universe, and supports the following operations.
2. Create S: Create a new empty soft-heap S.
3. Insert (S, x): Add a new item x to S.
4. Meld (S, S'): Form a new soft-heap with items stored in S and S'.
5. Delete (S, x): Remove an item x from S.
6. Findmin (S): Return an item in S with the smallest key.
7. Motivation behind Corruption of Keys: The soft heap may, at any time, increase the value of certain keys. Such keys, and by extension, the corresponding items, are called *corrupted*. Corruption is entirely at the discretion of the data structure and the user has no control over it. Naturally, findmin returns the minimum *current* key, and may or may not be corrupted. The benefit is speed: during heap updates, items travel together in packets in form of *car-pooling*, in order to save time.
8. Reducing Entropy of Data Stored: From an information-theoretic point of view, corruption is a way to decrease the entropy of the data stored in the data structure, and thus, facilitate its treatment. The entropy is defined as the logarithm, in base two, of the number of distinct key assignments, i.e., entropy of the uniform distribution over key assignments. To see the soundness of this idea, push it to its limit, and observe that if every key was corrupted by raising its value to ∞, then the set of keys would have zero entropy, and all operations can be performed in constant time. Interestingly,

soft heaps show that entropy need not drop to zero for the complexity to become constant.

9. Soft Heap Time Complexity Lemma: Beginning with no prior data, consider a mixed sequence of operations that includes n inserts, For any

   $$0 < \varepsilon \leq \frac{1}{2}$$

   a soft heap with error rate ε supports each operation in constant amortized time, except for inserts, which takes $\mathcal{O}\left(\log\frac{1}{\varepsilon}\right)$ time. The data structure never contains more than εn corrupted items at any time. In a comparison-based model, these bounds are optimal.
10. Actual Numbers of Items Corrupted: Note that this does not mean that only εn items are corrupted in total through the sequence of operations. In fact, it is not difficult to imagine a scenario where are items are eventually corrupted; for example, insert n items and keep deleting the corrupted ones. Despite this apparent weakness, the soft heap is optimal – and perhaps even mor surprising – useful. The data structure can be implemented on a pointer machine; no arrays are used, and no numeric assumptions on the keys are required.
11. Soft Heap as a Modified Binomial Queue: Soft heaps capture three distinct features, which are useful to understand at the outset. First, on setting

    $$\varepsilon = \frac{1}{2n}$$

    no corruption is allowed to take place and the soft heap behaves like a regular heap with logarithmic insertion time. Unsurprisingly, soft heaps include standard heaps as special cases. In fact, as will be seen, soft heap is nothing but a modified binomial queue (Vuillemin (1978)).

12. Soft Heap as a Median Finder …: More interesting is the fact that soft-heaps implicitly feature the median-finding technology. To see why, set ε to be a small constant; insert n keys and then perform $\left\lceil \frac{n}{2} \right\rceil$ findmins, each followed by a delete. This takes $\mathcal{O}(n)$ time. Among the keys deleted, the largest one is εn away from the median of the n original keys. To obtain such a number in linear time deterministically, as Chazelle (2000a) does, typically requires a variant of the median-finding algorithm of Blum, Floyd, Pratt, Rivest, and Tarjan (1973).
13. … but it is more than that: The previous remark should not lead one to think that a soft-heap is simply a dynamic median-finding data structure. Things are more subtle. Indeed, consider a sequence of n inserts of keys in decreasing order, inter-mixed with n findmins, and set

$$\varepsilon = \frac{1}{2}$$

Despite the high value of the error rate ε, the findmins must actually return the minimum key at least half the time. The reason is that at most $\frac{n}{2}$ keys inserted can ever be corrupted. Because of the decreasing order of the insertions, these uncorrupted keys must be reported by findmin right after their insertion, since they are minimum at that time. The requirement to be correct half the time dooms any strategy based on maintaining medians or near-medians for the purpose of findmin.

Applications

1. Overview of Soft Heap Uses: Soft heaps are useful for computing minimum spanning trees and percentiles, for finding medians, for near sorting, and generally for situations where approximate rank information is sought. Some examples below.
2. Usage in Minimum Spanning Trees: The soft-heap was designed with a specific application in mind, the minimum spanning tree. It is a key ingredient in what is

currently the fastest deterministic algorithm (Chazelle (2000b)) for computing the minimum spanning trees of a graph. Given a connected graph with n vertexes and m weighted edges, the algorithm finds a minimum spanning tree in time $\mathcal{O}\big(m\ \alpha(m,n)\big)$ where α is the classical functional inverse of Ackermann's function.

3. <u>Usage in Dynamic Rank Maintenance</u>: Another simpler application is the dynamic maintenance of percentiles. Suppose one wishes to maintain the grade point average of students in a college, so that at any time one request the name of a student with a GPA in the top percentile. Soft heaps support such operations in constant amortized time.
4. <u>Computing Selection in Linear Time</u>: Soft heaps give an alternate method for computing medians in linear time – or generally performing linear time selection (Blum, Floyd, Pratt, Rivest, and Tarjan (1973)). Suppose one wants to find the k^{th} largest element in a set of n numbers. Insert n numbers into the soft heap with error rate $\frac{1}{3}$. Next call findmin and delete $\frac{n}{3}$ times. The largest number deleted has rank between $\frac{n}{3}$ and $\frac{2n}{3}$. After computing this rank, one can remove therefore at least $\frac{n}{3}$ numbers from consideration. On recurses over the remainder in the obvious fashion. This allows finding the k^{th} largest element in time proportional to

$$n + \frac{2n}{3} + \left(\frac{2}{3}\right)^2 n + \cdots = \mathcal{O}(n)$$

5. <u>Usage in Weak Near Sorting:</u> A fourth application of soft heaps is approximate sorting. A weak version of near sorting requires that given n distinct numbers, the algorithm should output them in a sequence whose number of inversions is at most εn^2 instead of zero for exact sorting. As it turns out, this follows directly by inserting n numbers into a soft heap with error rate ε and then deleting the smallest keys repeatedly. The number I_k of inversions of the k^{th} deleted number of x is the number of keys deleted earlier whose original values are larger than x. But x must have been in a corrupted during those particular I_k earlier deletions. The total number of keys in

a corrupted state, counting over all deletions, is at most εn^2, and so the number of inversions $\sum I_k$ is also bunded by εn^2.

6. <u>Usage in Strong Near Sorting</u>: A stronger version of near sorting requires that in the output sequence the rank of no number differs from its true rank in sorted time by mor than εn.It is shown below how the soft heap enables that in $\mathcal{O}\left(\log \frac{1}{\varepsilon}\right)$ time. In particular, the lemma below provides simple linear algorithm for this strong form of near-sorting with, say 1% error in rank. Of course, the result itself is not new. It can be derived trivially from repeated mean computation. The idea is that it is done using soft heaps.
7. <u>Strong Near Approximation Sorting Lemma</u>: For an

$$0 < \varepsilon \leq \frac{1}{2}$$

   a soft-heap may be used to near-sort n numbers in time $\mathcal{O}\left(\log \frac{1}{\varepsilon}\right)$; this means that the rank of any number in the output sequence differs from its true rank by at most εn.
8. <u>Insertion followed by Minimum Finding</u>: Since the running time sought is $\mathcal{O}\left(\log \frac{1}{\varepsilon}\right)$, it may be assumed that $\frac{1}{\varepsilon}$ between a large constant and $\sqrt{n}$. Given n numbers, assumed to be distinct for simplicity, insert them into a soft heap and delete the item returned by findmin until the heap is empty.
9. <u>Dividing Findmins across Time Slices</u>: Using a soft heap with error rate ε, at most εn numbers are corrupted at any given time. The sequence of n-paits – findmin and delete – is divided into time intervals $T_1, \cdots, T_l$ each consisting of $\lceil 2\varepsilon n \rceil$ pairs; without loss of generality, it is assumed that

$$n = \lceil 2\varepsilon n \rceil l$$

10. Items Deleted/Corrupted during the Interval: Let S_i be the set of items deleted during T_i, and let U_i be the subset of S_i that includes only items that were at some point uncorrupted during T_i. It is assumed that items are time-stamped when first corrupted.

11. Smallest Uncorrupted Key in U_i: Finally, let x_i be the smallest original key among the items of U_i, and let s_i be the corresponding item; for convenience, set

$$x_0 = -\infty$$

and

$$x_{l+1} = +\infty$$

Given an item

$$s \in S_i$$

whose original key lies in $\left[x_j, x_{j+1}\right)$ one defines

$$\rho(s) = |i - j|$$

The following are some simple facts.

12. Bound on Cardinality of U_i:

$$|U_i| \geq \lceil 2\varepsilon n \rceil - \varepsilon n \geq \varepsilon n$$

because at most εn items are corrupted at the beginning of T_i.

13. Range of the Original Keys: The x_i's appear in increasing order, and the original key of any item in U_i lies in $[x_i, x_{i+1})$. Since s_{i+1} is uncorrupted during T_i, it's original key x_i is at most the current – and hence, original – key of any item deleted during T_i.

14. Upper Bound on the Corruption *Distance*:

$$\{\rho(s) \mid s \in S_i \setminus U_i\} < 2n$$

Given

$$s \in S_i \setminus U_i$$

let $\left[x_j, x_{j+1}\right)$ be the interval that contains the original keys of s. As just observed, the original key of s is less than x_{i+1} , and therefore

$$j \leq i$$

To avoid being selected by findmin, the item s must have been in a corrupted state during the deletion of x_k, for any

$$j < k < i$$

if such a k exists. The total number of items in a corrupted state during the deletions of $x_1, \cdots, x_l$ is at most εnl, and therefore so is the sum of distances

$$i - j - 1 = \rho(s) - 1$$

over all such items s. It follows that

$$\sum \rho(s) = \varepsilon nl + n < 2n$$

hence the claim.

15. Bounds on Original Keys in an Interval: The number of items whose original keys fall in $[x_i, x_{i+1})$ is less than $6\varepsilon n$. Indeed, suppose that the item does not belong to $S_i \cup$

S_{i+1}. It cannot be selected by findmin and deleted before the beginning of T_i, since S_i was not corrupted yet. By the time S_{i+1} was deleted, the item in question must have been corrupted, and it cannot have been deleted yet. So, there can be at most εn such items. Thus, the total number of items with original keys in $[x_i, x_{i+1})$ is at most

$$2\lceil 2\varepsilon n\rceil + \varepsilon n < 6\varepsilon n$$

16. <u>Origin of the $\log\frac{1}{\varepsilon}$ Bound</u>: Next, for each item

$$s \in S_i \setminus U_i$$

the search for which interval $\left[x_j, x_{j+1}\right)$ contains its original key can be done in $\mathcal{O}(\rho(s) + 1)$ time by sequential search. From the above lemma, this means that in $\mathcal{O}(n)$ post-processing time, the set of original keys have been partitioned into disjoint intervals, each containing between εn and $6\varepsilon n$ keys. So, in $\mathcal{O}\left(\log\frac{1}{\varepsilon}\right)$ time, any number can be output in a position of at most $6\varepsilon n$ off its rank. Replacing ε by $\frac{\varepsilon}{6}$ completes the proof.

The Data Structure

1. <u>Soft Queue as a Modified Binomial Tree</u>: Recall that a binomial tree of rank k is a rooted tree of 2^k nodes; it is formed by the combination of two binomial trees of rank $k - 1$ where the root of one becomes the new child of the other root (Vuillemin (1978)). A soft heap is a sequence of modified binomial trees of distinct ranks, called *soft queues*. The modifications come in the following two ways.
2. <u>Master Tree and Rank Invariance</u>: A soft queue q is a binomial tree with sub trees possibly missing, somewhat like tees of a Fibonacci heap (Fredman and Tarjan (1987)) after a few deletions. The binomial tree from which q is derived is called its

master tree. The *rank* of a node q is the number of children in the corresponding node in then master tree. Obviously, it is an upper bound on the number of children in q. The following *rank invariance* is therefore enforced: the number of children of the root should be no smaller than $\frac{rank\ (root)}{2}$

3. Items in a Soft Queue Node: A node v may store several items, in fact a whole *item-list*. The $ckey$ of v denotes the common value among all of the current keys in the $item - list\ (v)$; it is an upper bound on the original keys. The soft-heap is heap-ordered with respect to $ckey$, i.e., the $ckey$ of a node does not exceed the $ckey$ of any of its children. Fixing an integer parameter

$$r = r(\varepsilon)$$

all corrupted items are required to be stored at nodes of rank greater than r.

4. Structure of the Item-List: Turning to the actual code, an item list is a list of singly-linked items with one field indicating the original value of the key.
5. Structure of the Soft Queue: A node of a soft queue indicates its $ckey$ and its rank in the master tree. Pointers $child$ and $next$ give access to the children. If there are none, the pointers are NULL. Otherwise, the node is the parent of a soft-queue of rank one less – pointed to by $child$ – and the root of a soft queue of rank one less – pointed to by $next$. This is a standard artifice to represent the high degree nodes as sequences of degree-2 nodes. To facilitate the concatenation of item-lists a pointer to the tail of the list is also provided.
6. Top Structure of the Heap: The top structure of the heap consists of a doubly linked list consists of a doubly linked list $h_1, \cdots, h_m$ called the *head-list*; each *head* h_i has two extra pointers; one - $queue$ – points to the root r_i of a distinct queue, and another - $suffix\_min$ – points to the root of the minimum $ckey$ among all r_j for

$$j \geq i$$

It is required that

$$rank\ (r_1) < \cdots < rank\ (r_m)$$

By extension, the rank of a queue – resp. heap – refers to the rank of its root – resp. r_m. It is stored in the head h_i as the integer variable $rank$.

7. Initialization of the Soft Heap: The soft heap is initialized by creating two dummy heads as global variables; $header$ gives access to the head-list, while $tail$, of infinite rank, represents the end of the list. The functions new_head and new_node create and initialize a new head and a new node in the trivial manner. The third global variable is the parameter

$$r = r(\varepsilon)$$

Soft Heap Operations

This section discusses a minor variant of the data structure, whose implementation is slightly simpler. It is straightforward to modify the data structure into a full-fledged soft heap. First, the variant bypasses create operation and integrates with insert. Also, note that the operation delete can be implemented in *lazy* style by simply marking the item to be deleted accordingly. Then, actual work is required only when *findmin* returns an item that is marked as deleted. For this reason, the discussion of $findmin$ and $delete$ is skipped altogether, and instead the focus is on $deletemin$. It is obvious from this how to modify the data structure to accommodate $findmin$ and $delete$ separately.

Inserting an Item

To insert a new item, an uncorrupted one node queue is created, and melded into the heap.

Melding Two Heaps

1. Breaking Down Lower-Rank Heap: Consider melding two heaps S and S'. This section begins with a quick review, and then discusses the actual implementation of the algorithm.
2. Inserting a Queue into Heap: To meld a queue of rank k into S, the algorithm looks for the smallest index i such that

   $$rank\ (r_i) \geq k$$

   The dummy head *tail* ensures that i always exists. If

   $$rank\ (r_i) > k$$

   the head is inserted right before h_i instead. Otherwise, the two queues are melded into one of rank $k + 1$ by making the root with a larger key a new child of the other root.
3. Case of Pre-existing Rank: If

   $$rank\ (r_{i+1}) = k + 1$$

   a new conflict arises. The process is repeated as long as necessary like a carry propagation in binary addition.
4. Updating the *ckey suffix_min* Head: Finally, the suffix_min pointers between h_1 and the last visited head are updated. When melding not just a single queue but a while heap, the last step can be done at the very end in just one pass through S. The gist of the code for melding a soft queue into a soft heap is provided below.
5. Heap Rank corresponding to Queue: Let q be a reference to the soft queue to be melded into the soft heap. First, the *head-list* is scanned to locate the point at which

the melding proper can begin. This leads to the first head of rank at least that of q which is denoted by $toHead$. To facilitate the insertion of the new queue, the preceding head - $prevHead$ – is retained.

6. Carry Propagation to Handle Pre-existing Ranks: If there is already a queue of the same rank as q, the carry propagation is performed as discusses earlier. When merging two queues, the variables top and $bottom$ are used to specify which of the two queues end up at/below the $root$. A new node q pointing to top and $bottom$ is created. Its $item - list$ is inherited from the top and its rank is one plus that of the top, i.e., $top.rank() + 1$. Finally, the $toHead$ is updated to point the next element down the $head - list$.
7. Insertion of the New Queue: The new queue can be inserted into the list of heads. Chazelle (2000a) uses a trick; if a carry has actually taken place, the head pointed to by $prevHead.next()$ is unused now and so can be recycled as the head of a new queue. Otherwise, a new head h is created. h is inserted between $prevHead$ and $toHead$; all the heads in between can now be discarded. The call to $fixMinList\ (h)$ restores the $suffix\_min$ pointers.
8. Updating the Queue $ckey$ Pointers: Prior to calling $fixMinList\ (h)$ it is assumed that all $suffix\_min$ pointers are correct, except for those between $header$ and h. A simple walk from h back to $header$ updates all the $suffix\_min$ pointers.

DeleteMin

1. $suffix\_min$ Pointing to Empty Item-List: The $suffix\_min$ pointer at the beginning of the $head - list$ points to the head h with the minimum $ckey$ – corrupted or not. The trouble is that the item-list at that node might be empty. In that case, the item-list must be filled with items taken lower down in the queue pointed to by h. To do that, one calls the function $sift\ (h.queue(), h.rank())$ which replaces the empty item-list by another one. If necessary, this process is iterated until the new item-list at the root is not empty.

2. Migrating Items up the Root: Taking an argument the node v at which the sifting takes place, the function $sift$ attempts to move items up the tree towards the root. This is a standard operation in classical heaps. Typically, there is a single recursive call in the procedure and the computation tree is a path. The twist is the call to the recursion tree twice occasionally, in order to make the recursion tree branching, i.e., truly a tree. This simple modification causes item-lists to collide on the way up, which is resolved by concatenating them.
3. Invoking the Secondary Recursion: The loop condition for the second recursion is what makes the soft heap special. Without it, $sift$ would be indistinguishable from the standard $deletMin$ operation of a binomial tree. The loop condition holds if:
    a. the recursive loop has not yet been executed during this invocation of $sift$, i.e., branching is at most binary;
    b. the rank of v exceeds the threshold of r and it is either odd or it exceeds the rank of the highest-ranked child of v by at least two.
4. Purpose behind the Recursion Criterion: The rank condition ensures that no corruption takes place too low in the queue; the parity condition is there to keep the branching from occurring too often; finally, the last condition ensures that branching occurs frequently enough. The variable T implements the car-pooling in the concatenation

$$T \leftarrow T \cup item - list\ (v.next())$$

   The cleanup is intended to prune the tree of nodes that have lost their item-lists to ancestors and whose keys have been set to $+\infty$.
5. Sift #1 - Ensuring v is NULL: The $item - list$ at v is worthless and it is effectively emptied at the beginning. The node v is tested to see if it is a leaf. If so, it is bottomed out by setting its $ckey$ to $+\infty$, i.e., a large integer which will cause the node to stay at the bottom of the queue. If v is not a leaf, then neither $v.next()$ nor $v.child()$ are NULL. In fact, this is a general statement; both are NULL or neither one is. This might change temporarily inside a call to $sift$ but it is restored before the call ends.

6. Sift #2 - Maintaining Heap Ordering: The new item-list at $v.next()$ might now have a large $ckey$ which violated heap ordering. If so, a rotation is performed by exchanging the children $v.next()$ and $v.child()$.
7. Sift #3 - Update Node Contents: Once the children of v are in place, the various pointers of v are updated. In particular, the item-list of $v.next()$ is passed on to v, and so is its $ckey$. Recall that while $v.child()$ is truly a child of v in the soft queue, the node $v.next()$ is a child of v only in the binary tree implementation of the queue.
8. Sift #4 - Secondary Recursion Criterion: Next in line, the most distinctive feature of soft-heaps; the possibility of sifting twice, i.e., of creating a branching process in the recursion tree for $sift$. If the secondary recursion condition is satisfied, meaning that the rank of v is off and large enough, the $sift$ is re-done.
9. Sift #5 - Heap Order Restoration: As a result of the sifting, another rotation may be required to restore heap ordering.
10. Sift #6 – Item-List Concatenation: The item-list at $v.next()$ should now be concatenated with the one at v, unless of course, it is empty or no longer defined. The latter case occurs when $ckey$ is $+\infty$ at both $v.child()$ and $v.next()$. Note that this could not happen after the previous $sift$.
11. Sift #7 - Terminal Cleanup: The queue is cleaned up by removing the nodes with infinite $ckey$. $v.rank()$ is *not* updated since it is defined with respect to master tree. Note that this is where the rank and the number of children can be made to differ. In fact, for any node v, the ranks of its children in the binary tree are always ensured to be equal, i.e.,

$$v.next().rank() = v.child().rank()$$

12. DeleteMin #1 - Child Node Count: The function $deleteMin$ returns the item with the smallest $ckey$ and deletes it. The $suffix\_min$ pointer takes us to the smallest $ckey$, which is precisely what is wanted, unless of course, the corresponding item list is empty. In that case, $sift$ invoked – perhaps more than once – to bring items back to the root. But first, the check for the rank invariance violation must be done. Indeed,

the previous sifting may have caused the loss of too many children of the root, and hence a violation of the invariant. The number of children is therefore counted – alternatively, a field could be added to keep track of this number.

13. DeleteMin #2 - Rank Invariance Verification: The advantage of detecting a rank invariance violation so late in the game is that fixing it now is much easier since the root's item-list is empty. If the rank invariance is violated, i.e.,

    $$childCount < \left\lfloor \frac{h.rank()}{2} \right\rfloor$$

    the queue is removed, and the head-list and $suffix\_min$ pointers are updated. The root is *dismantled* by re-melding back its children.
14. DeleteMin #3 - Item List Sifting: If the rank invariance holds, the item-list is refilled at the root by calling $sift$.
15. DeleteMin #4 - Deleting Minimum Key: The minimum-key item can now be deleted.

Complexity Analysis

1. Soft Queue Master Tree Correspondence: This section proves that the soft-heap meets all its claims using a few simple lemmas. Consider a mixed sequence of operations including n inserts. To begin with, the correspondence between a soft queue and its master tree must be explained. When no deletion takes place, the equivalence is obvious, and it is trivially preserved through the inserts and the melds. During sifting, the principal observation is that $v.next().rank()$ and $v.child().rank()$ always remain identical. Enforcement of this equality is what can cause a discrepancy between the rank and the number of children. But it allows viewing a rotation as an exchange between soft queues of the same rank, albeit with missing sub-trees. The corresponding master trees have the same rank, they are isomorphic, and therefore a

rotation has no effect in the correspondence. Similarly, the clean up prunes away sub-trees, with no consequence on the queue/master-tree correspondence.

2. Missing Leaves of the Master Tree: The interesting aspect of this correspondence is that the leaves of the master tree that are missing from the soft queue correspond to items that have migrated upward to join the item-lists of nodes of positive rank. Such items can never appear again in the leaves of any soft-queue. Note that dismantling a node by re-melding its children does not contradict this statement since it merely reconfigures the soft heap.

The Error Rate

1. Relation between r and the Error Rate: To achieve the desired error rate, one sets

$$r := 2 + 2\left\lceil \log \frac{1}{\varepsilon} \right\rceil$$

2. Vertex Item Cardinality Lemma:

$$item\_list\ (v) \leq \max\left(1, 2^{\left\lceil \frac{rank\ (v)}{2} \right\rceil - \frac{r}{2}}\right)$$

3. Operations Impact on Item List Size: Until the first call to $sift$, all item lists have size one, and the inequality holds. Afterwards, simple inspection shows that all operations have no impact on the lemma’s inequality, or sometimes a favorable one, e.g., $meld$. All of them, that is, except for $sift$, which could potentially cause a violation. Here it is shown that this is not the case, and the lemma’s inequality is proven by induction.
4. Sift Impact on Item List Size: If $sift$ calls itself recursively via $sift(v.next())$ only once, meaning that the loop condition is not satisfied, then the item list of $v.next()$ – after possible rotation – migrates to a higher ranking node by itself, and the lemma

holds by induction. Otherwise, the item-list at v becomes the union of two item lists associated with $v.next()$ after each call to $sift\ (v.next())$.

5. Impact of Sift Recursion #1: For the second sift recursion to happen, $v.rank()$ must exceed r, and one of two conditions must hold: either $v.rank()$ is odd, or it exceeds $v.child().rank() + 1$. In the first case, after the recursive call, the rank of $v.next()$ is strictly less than $v.rank()$, and by induction, the size of either one of the item-lists of $v.next()$ is at most

$$\max\left(1, 2^{\left\lceil\frac{rank\ (v)-1}{2}\right\rceil-\frac{r}{2}}\right) = 2^{\left\lceil\frac{rank\ (v)-1}{2}\right\rceil-\frac{r}{2}} = 2^{\left\lceil\frac{rank\ (v)}{2}\right\rceil-1-\frac{r}{2}}$$

Note that the max() disappears because

$$rank\ (v) > r$$

6. Impact of Sift Recursion #2: In the other case, the size of either one of the item-lists of $v.next()$ is at most

$$\max\left(1, 2^{\left\lceil\frac{rank\ (v)-2}{2}\right\rceil-\frac{r}{2}}\right) = 2^{\left\lceil\frac{rank\ (v)}{2}\right\rceil-1-\frac{r}{2}}$$

This time, the max() disappears because r and the rank of v are both even, and so

$$rank\ (v) \geq r + 2$$

In sum, the size of the union is at most $2 \times 2^{\left\lceil\frac{rank\ (v)}{2}\right\rceil-1-\frac{r}{2}}$, which proves the lemma.

7. Soft Heap Corrupted Item Bound Lemma: The soft heap contains at most $\frac{n}{2^{r-3}}$ corrupted items at any given time.

8. Binomial Tree Node Set Cardinality: The lemma proof starts with a simple observation. If S is the node set of a binomial tree, then

$$\sum_{v \in S} 2^{\frac{rank(v)}{2}} \leq 4|S|$$

This follows from the inequality

$$\sum_{v \in S} 2^{\frac{rank(v)}{2}} \leq 2^{k+2} - 3 \cdot 2^{\frac{k}{2}}$$

where k is the rank of the binomial tree. A proof by induction is immediate and can be omitted here.

9. Bound of Master Tree Node Count: Recall that the ranks of the nodes of a queue q is derived from the corresponding nodes in the master tree q'. So, the set R – respectively R' - of nodes of rank greater than r in q – respectively q' - is such that

$$|R| \leq |R'|$$

Within q', the nodes of R' number a fraction at most $\frac{1}{2^r}$ of all leaves. Summing over the master trees, it may be found that

$$\sum_{q'} |R'| \leq \frac{n}{2^r}$$

10. Master Tree Corrupted Item Bound: There is no corrupted item at any

$$rank \leq r$$

so, by the Vertex Item List Cardinality Lemma, their total number does not exceed

$$\sum_{q'}\sum_{v\in R'} 2^{\frac{rank(v)+1-r}{2}} = 2\sum_{q'}\sum_{v\in R'} 2^{\frac{rank(v)-1-r}{2}}$$

Each R' forms a binomial tree by itself, where the rank of node v becomes $rank(v) - 1 - r$. So, using

$$\sum_{v\in S} 2^{\frac{rank(v)}{2}} \leq 4|S|$$

and

$$\sum_{q'} |R'| \leq \frac{n}{2^r}$$

the sum in $2\sum_{q'}\sum_{v\in R'} 2^{\frac{rank(v)-1-r}{2}}$ is at most

$$\sum_{q'} 8|R'| \leq \frac{n}{2^{r-3}}$$

The Running Time

1. Big-$\mathcal{O}$ for Meld and Shift: Only meld and shift need to be looked at, all other operations being trivially constant-time. Assigning one credit per queue accounts for the carry propagation during the meld. Indeed, two queues of the same rank combine into one, which releases one credit to pay for the work. Updating $suffix\_min$ pointers can take time, however.
2. Modeling Melds using Binary Tree: Specifically, carries aside, the cost of melding two soft heaps S and S' is at most the smaller rank of the two, up to a constant factor. The entire sequence of the soft heap melds can be modeled as a binary tree $\mathcal{M}$. A leaf

z denotes a one-item heap, its cost is 1. An internal node z denotes the melding of two heaps.

3. Meld Cost using Node Count: Since heaps can grow only though melds, the added size of the master heap in the soft heap at z is proportional to the number $N(z)$ of descendants in $\mathcal{M}$. The cost of the node z, i.e., of the meld, is $1 + \log(\min(N(x), N(y)))$ where x and y are the right children of z – this uses the fact that the rank of the soft queue is exactly the logarithm of the number of nodes in its master tree. A simple recurrence, e.g., see Hoffman, Mehlhorn, Rosenstiehl, and Tarjan (1986), shows that adding together all these costs gives a total melding cost that is linear in the size of $\mathcal{M}$, i.e., $\mathcal{O}(n)$.
4. Accounting for Dismantle-Induced Melds: For the analysis to be correct, no node dismantling should ever take place. This can be easily amended, however, to cover the general case. For the purposes of the analysis, re-meldings are assumed to be caused not by dismantlement of heap melds, but by queue melds. The benefit is to leave the tree $\mathcal{M}$ unchanged. The dismantle-induced melds associated with the node z of $\mathcal{M}$ re-configure the soft heap at z by removing some of its nodes and restoring the rank invariant. This can only decrease the value of $N(z)$, so the previous analysis remains correct.
5. Analysis of the Node Dismantlement Costs: Of course, the queue melds associated with node x must now be accounted for. Dismantling node v causes no more than $rank(v)$ queue melds. By the violation of the rank invariant, node v has at least one missing child of

$$rank \geq \left\lceil \frac{rank(v)}{2} \right\rceil$$

In the master tree there are at least $2^{\left\lceil \frac{rank(v)}{2} \right\rceil - 1}$ leaves at or below that child, and all have disappeared from the soft queue. So, the dismantle-induced melds can be charged against these leaves, thereby concluding that melding takes $\mathcal{O}(n)$ time.

6. Consolidated Cost of Sift #1: Finally, it is shown that the cost of all calls to sift is $O(m)$. Consider any decreasing sequence of integers. An integer m is called good if it is odd or if its successor is less than $m-1$. Clearly, any sub-sequence of size two contains at least one good integer. Now consider a computation tree corresponding to an execution of $sift(v)$. By examining the sequence of ranks along any root-to-leaf path, the previous observation leads to the conclusion that along any path size at least r, at least one branching must occur, and not necessarily many more than that, because no branching occurs at rank r and below. It follows that, excluding the update of $suffix\_min$ pointers, the running time is $O(rC)$, where C is the number of times the loop-condition succeeds.
7. Consolidated Cost of Sift #2: It is easy to see, by induction, that if v is the root of a sub-tree with fewer than two finite $ckey$'s below, the computation tree of $sift\ (v)$ is of constant size. Conversely, if the sub-tree contains at least two finite $ckey$'s at distinct nodes, then if the loop condition is satisfied at v, both calls of the form $sift\ (v.next())$ bring finite $ckey$'s to the root, and the two non-empty item-lists are thus concatenated. There can be at most $n-1$ such merges, therefore

$$C \leq n$$

and the claim holds.
8. Cost of Updating $suffix\_min$ Pointers: The cost of updating $suffix\_min$ pointers after each call to $sift$ has been ignored so far. Maintaining the rank invariant makes the cost of $suffix\_min$ updating negligible. Indeed, each update takes time proportional to the rank of the queue. If the rank invariant holds, the updating time is dominated by the $sift$ itself, and is already accounted for. Otherwise, the root v is dismantled, which, as just seen, releases $2^{\left\lceil \frac{rank(v)}{2} \right\rceil - 1}$ leaves against which the update cost may be charged. By the Soft Heap Corrupted Bound Item Lemma, the number of corrupted items is bounded by $\frac{n}{2^{r-3}}$. Setting

$$r := 2 + 2\left\lceil \log \frac{1}{\varepsilon} \right\rceil$$

proves the Soft Heap Complexity Lemma, except for the optimality claim.

9. Algorithm Modification under Storage Constraints: The storage is linear in the number of insertions n, but not necessarily in the number of items present. If storage is a premium, here is a simple fix. As soon as the number live items falls below, say $\frac{n}{2}$, the soft-heap is entirely re-configured. All uncorrupted items are r-inserted, and separately, all corrupted items are strung together into a single item-list, whose key is set to $+\infty$. The time taken for re-configuring the heap can be charged to the $\frac{n}{2}$ deleted elements. Regarding corruption, re-configuration merely doubles the number of insertions, and so the number of corrupted items after I user requested insertions will be at most $2\varepsilon I$. So, it suffices to replace ε by $\frac{\varepsilon}{2}$ to achieve an error rate of ε. The modified soft heap is optimal in storage, and as shown below, in time.

Optimality

1. Proof of Soft-Heap Time Complexity Lemma: To complete the proof of the soft-heap time complexity lemma, this section shows that the soft-heap is optimal. Without loss of generality, one can assume that $\frac{1}{\varepsilon}$ lies between a large constant and $\sqrt{n}$, and that $\frac{n}{\lfloor \varepsilon n \rfloor}$ is an integer l.
2. True Rank of a Selection: Apply the Sting Near Approximation Sorting Lemma to n distinct numbers, and pick out every $2\lfloor \varepsilon n \rfloor^{th}$ element in the output sequence. By the near-sortedness of the output, the chosen subsequence is already sorted. It partitions the set of numbers into disjoint intervals, within with one can easily locate the other numbers in linear time, each number being at most a constant number of intervals off its enclosing interval. From this, in particular, the true rank of the selected numbers is derived.

3. $\mathcal{O}(n)$ Partitioning of the Selection: By using linear selection-finding within each interval, the $\lfloor \varepsilon n \rfloor^{th}$ largest element can be easily retrieved, for

$$k = 1, 2, \cdots, l$$

Thus, the set of n numbers can be partitioned into disjoint intervals of size $\lfloor \varepsilon n \rfloor$.

4. Computation Height of the Comparison-Based Tree: Let

$$n_1 = \cdots = n_l = \lfloor \varepsilon n \rfloor$$

A standard counting argument shows that any comparison-based tree for performing this computation is of height at least

$$\log \binom{n}{n_1, \cdots, n_l} = \Omega\left(n \log \frac{1}{\varepsilon}\right)$$

5. Optimality of Amortized Complexity: So, the entire algorithm requires $\Omega\left(n \log \frac{1}{\varepsilon}\right)$ time, but it involves $\mathcal{O}(n)$ operations on a soft-heap with error rate ε, followed by $\mathcal{O}(n)$-time post-processing. It follows that the $\mathcal{O}\left(\log \frac{1}{\varepsilon}\right)$ amortized complexity of the soft-heap is optimal.

References

- Blum, M., R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan (1973): Time Bounds for Selection *Journal of Computer and System Sciences* **7 (4)** 448-461
- Chazelle, B. (2000a): The Soft-Heap: An Approximate Priority Queue with Optimal Error Rate *Journal of the ACM* **47 (6)** 1012-1027

- Chazelle, B. (2000b): A Minimum Spanning Tree Algorithm with inverse-Ackermann Type Complexity *Journal of the ACM* **47 (6)** 1028-1047
- Fredman, M. L., and R. E. Tarjan (1987): Fibonacci Heaps and their Use in Improved Network Optimization Algorithms *Journal of the ACM* **34 (3)** 596-615
- Hoffman, K., K. Mehlhorn, P. Rosenstiehl, and R. E. Tarjan ((1986): Sorting Jordan Sequences in Linear-Time using Level-linked Search Trees *Information and Control* **68 (1-3)** 170-184
- Vuillemin, J. (1978): A Data Structure for Manipulating Priority Queues *Communications of the ACM* **21 (4)** 309-315

Spanning Tree

Overview

A *spanning tree* of an undirected graph G is a subgraph that includes all of the vertexes of G, with minimum possible number of edges (Wikipedia (2020)). In general, a graph may have several spanning trees, but a graph that is not connected will not contain a spanning tree. If all of the edges of G are also edges of a spanning tree T of G, then G is a tree and identical to T, that is, a tree has a unique spanning tree and that is itself

Applications

1. Use in Path Finding Algorithms: Several path-finding algorithms, including Dijkstra's algorithm and the A* search algorithm, internally build a spanning tree as an intermediate step in solving the problem.
2. Use in Cost Minimization Problems: In order to minimize the cost of power networks, wiring connections, piping, automatic speech recognition, etc., people often use algorithms that gradually build a spanning tree – or many such trees – as intermediate steps in the process of finding the minimum spanning tree (Graham and Hell (1985)).
3. Use in Link-State Protocols: The internet and many other telecommunications networks have transmission links that connect nodes together in a mesh topology that includes some loops. In order to avoid *bridge loops* and *routing loops*, many protocols design for such networks – including Spanning Tree Protocol, Open Shortest Path First, Link-State Routing Protocol, Augmented Tree-Based Routing, etc. – require each router to remember a spanning tree.
4. Graph Embeddings with Maximum Genus: A special kind of tree, the Xuong tree, is used in topological graph theory to find graph embeddings with maximum genus. A

Xuong tree is a spanning tree such that, in the remaining graph, the number of connected components with an odd number of edges is as small as possible. A Xuong tree and an associated maximum genus embedding can be found in polynomial time (Beineke and Wilson (2009)).

Definitions

A tree is a connected, undirected graph with no cycles. It is a spanning tree of a graph G if it spans G – that is, it includes every vertex of G – and is a subgraph of G, i.e., every edge in the tree belongs to G. A spanning tree of a connected graph G can also be defined as a maximal set of edges G that contains no cycle, or as a minimal set of edges that connect all vertexes.

Fundamental Cycles

Adding just one edge to the spanning tree will create a cycle; such a cycle is called a *fundamental cycle*. There is a distinct fundamental cycle for each edge not in the spanning tree; thus, there is a one-to-one correspondence between fundamental cycles and edges not in the spanning tree. For a connected graph with V vertexes, and spanning tree will have $V - 1$ edges, and thus, a graph of E edges and one of its spanning trees will have $E - V + 1$ fundamental cycles. For any given spanning tree, the set of all $E - V + 1$ fundamental cycles forms a cycle basis, a basis for the cycle space (Kocay and Kreher (2004)).

Fundamental Cut-sets

1. Motivation behind the Fundamental Cut-set: Dual to the notion of a fundamental cycle is the *fundamental cut-set*. By deleting just one edge of the spanning tree, the vertexes are partitioned into two disjoint sets. The fundamental cut-set is defined as the set of edges that must be removed from the graph G to achieve the same partition. Thus, each spanning tree defines a set of $V - 1$ fundamental cut-sets, one for each edge of the spanning tree (Kocay and Kreher (2004)).
2. Duality between Cut-sets and Cycles: The duality between fundamental cut-sets and fundamental cycles is established by noting that the cycle edges not in the spanning tree can only appear in the cut-sets of the other edges of the cycle; and *vice versa*; edges in a cut-set can only appear in those cycles containing the edge corresponding to the cut-set. This duality can be expressed using the theory of matroids, according to which the spanning tree is the base of a graphic matroid; a fundamental cycle is the unique circuit within the set formed by adding one element to the base, and fundamental cut-sets are defined in the same way as the dual matroid (Oxley (2006)).

Spanning Forests

1. Competing Definitions of Spanning Forests: In graphs that are not connected, there can be no spanning tree, and on must consider *spanning forests* instead. Here, there are two competing definitions:
    a. Some authors consider a spanning forest to be the maximal acyclic subgraph of a given graph, or equivalently, a graph consisting of a spanning tree in each connected component of the graph (Bollobas (1998), Mehlhorn (1999)).
    b. For other authors, a spanning forest is a forest that spans all of the vertexes, meaning only that each vertex in the graph is a vertex in the forest. Under this definition, even a connected graph may have a disconnected spanning forest, such as a forest in which each vertex forms a single-vertex tree (Cameron (1994)).

2. Full versus Maximal Spanning Forest: To avoid confusion between these two definitions, Gross and Yellen (2005) suggest the term *full spanning forest* for a spanning forest with the same connectivity as a given graph, while Bondy and Murthy (2008) instead call this kind of forest a maximal spanning forest.

Counting Spanning Trees

The number $t(G)$ of the spanning trees of a connected graph is a well-studied invariant.

In Specific Graphs

1. When G is a Tree: In some cases, it is easy to calculate $t(G)$ directly. If G is a tree itself, then

$$t(G) = 1$$

2. G is a Cycle Graph: When G is a cycle graph with n vertexes, then

$$t(G) = n$$

3. G is Complete with n Vertexes: For a complete graph with n vertexes, Cayley's formula (Aigner and Ziegler (1998)) gives the number of spanning trees as n^{n-2}.
4. G is Complete Bipartite: If G is a complete bipartite graph $K_{p,q}$ then

$$t(G) = p^{q-1}q^{p-1}$$

(Hartsfield and Ringel (2003)).

5. G is an n-dimensional Hyper-cube: For an n-dimensional hyper-cube graph, the number of spanning trees is

$$t(G) = 2^{2^n - n - 1} \prod_{k=2}^{n} k^{\binom{n}{k}}$$

(Harary, Hayes, and Wu (1988)).

In Arbitrary Graphs

1. Arbitrary Graph Spanning Tree Count: More generally, for any graph G, the number $t(G)$ can be calculated in polynomial time as a determinant of a matrix derived from the graph, using Kirchoff's matrix-tree theorem.
2. Kirchoff's Matrix-Tree Theorem Method: Specifically, to compute $t(G)$, one constructs a square matrix in which both the rows and the columns are indexed by vertexes of G. The entry in row i and column j is one of three values:
   a. The degree of vertex i if

   $$i = j$$

   b. -1 if vertexes i and j are adjoint, OR
   c. 0 if vertexes i and j are different from each other but not adjacent.

   The resulting matrix is singular, so its determinant is zero. However, deleting a row and a column for an arbitrarily chosen vertex leads to a smaller matrix whose determinant is exactly $t(G)$.

Deletion-Contraction

1. The Deletion-Contraction Recurrence Formula: If G is a graph or a multi-graph, and e is an arbitrary edge of G, then the number $t(G)$ of spanning trees of G satisfies the *deletion-contraction recurrence*

$$t(G) = t(G - e) + t(G/e)$$

where $G - e$ is the multi-graph obtained by deleting e, and G/e is the contraction of G by e (Kocay and Kreher (2004)). The term $G - e$ in the formula counts the number of spanning trees of G that do not use the edge e, and the term $t(G/e)$ counts the spanning trees of G that use the edge e.
2. Retention of Redundant Graph Edges: In the above formula, is the given graph G is a multi-graph, or if a contraction causes two vertexes to be connected to each other by multiple edges, then the redundant edges should not be removed, as that would lead to the wrong total. For instance, a bond graph connecting two edges by k edges has k different spanning trees, each consisting of one of these edges.

Tutte Polynomial

1. Sum over Internal/External Activity: The Tutte polynomial of a graph can be defined as a sum, over the spanning trees of the graph, of terms computed from *internal activity* and *external activity* of the tree. Its value at the arguments $(1, 1)$ is the number of spanning trees, or, in a disconnected graph, the number of maximal spanning forests (Bollobas (1998)).
2. Computational Complexity using Contraction-Deletion Recurrence: The Tutte polynomial can be computed using a deletion-contraction recurrence, but its computational complexity is high: for many values of its arguments, computing it is exactly **#P**-complete, and it is also hard to approximate with a guaranteed approximation ratio. The point $(1, 1)$ at which it can be evaluated using Kirchoff's

theorem, is one of the few exceptions (Jaeger, Vertigan, and Welsh (1990), Goldberg and Jerrum (2008)).

Algorithms – Construction

1. Spanning Tree using BFS/DFS: A single spanning tree of a graph can be found in linear time by either depth-first search or breadth-first search. Bothe of these algorithms explore then given graph, starting from an arbitrary vertex V, by looping through the neighbors of the vertexes they discover and adding each unexplored neighbor to a data structure to be explored later. They differ in whether the data structure is a stack – in the case of depth-first search – or a queue – in the breadth-first search. In either case, one can form a spanning tree by connecting each vertex, other than the root vertex V, to a vertex from which it was discovered. This tree is known as the depth-first search tree or the breadth-first search tree according to the graph exploration algorithm used to construct it (Kozen (1992)). Depth-first search trees are a special case of a class of spanning trees called the Tremaux trees, named after the 19th century discoverer of depth-first search (de Fraysseix and Rosenstiehl (1982)).
2. BFS/DFS in Parallel/Distributed Environments: Spanning trees are important in parallel and distributed computing, as a way of maintaining communications between a set of processors; see, for instance, the spanning tree protocol used by the OSI link-layer devices of the Shout protocol used for distributed computing. However, the breadth-first and the depth-first methods for constructing spanning trees on sequential computes are not well-suited for parallel and distributed computer (Reif (1985)). Instead, researchers have devised more specialized algorithms for finding spanning trees in these models of computation (Gallagher, Humblet, and Spira (1983), Gazit (1991), Bader and Cong (2005)).

Algorithms - Optimization

1. Spanning Trees under Optimal Condition: In certain fields of optimization theory, it is often useful to find a minimum spanning tree of a weighted graph. Other optimization problems in spanning trees have also been studied, including the maximum spanning tree, the maximum tree that spans k vertexes, the spanning tree with the fewest edges per vertex, the spanning tree with the largest number of leaves, the spanning tree with the fewest leaves – closely related to the Hamiltonian path problem, the maximum diameter spanning tree, and the maximum dilation spanning tree (Eppstein (1999), Wu and Cao (2004)).
2. Optimal Spanning Trees in Euclidean Space: Optimal spanning tree problems have also been studied for a finite set of points in a geometric space such as the Euclidean space. For such an input, the spanning tree is again a set of trees that has as its vertexes the given points. The quality of the tree is measured in the same way as in a graph, using the Euclidean distance between pairs of points as the weight for each edge. Thus, for instance, a Euclidean minimum spanning tree is the same as the graph minimum spanning tree in a complete graph with Euclidean edge weights. However, it is not necessary to construct the graph in order to solve the optimization problem; the Euclidean minimum spanning tree problem, for instance, can be solved more efficiently in $\mathcal{O}(n \log n)$ time by constructing Delaunay triangulation and then applying a linear planar graph minimum spanning tree algorithm to the resulting triangulation (Eppstein (1999)).

Randomization

1. Generation of Uniform Spanning Trees: A spanning tree chosen from among all the spanning trees with equal probability is called a uniform spanning tree. Wilson's algorithm can be used to generate uniform spanning trees in polynomial time by a

process of taking a random walk on the given graph and erasing the cycles created by this walk (Wilson (1996)).

2. Generating Random Minimal Spanning Tree: An alternative model for generating spanning trees randomly but not uniformly is the random minimal spanning tree. In this model, the edges of the graph are assigned random weights and then the minimum spanning tree of the weighted graph is constructed (McDiarmid, Johnson, and Stone (1997)).

Enumeration

Because a graph may have exponentially many spanning trees, it is not possible to list them all in polynomial time. However, algorithms are known for listing all spanning trees in polynomial time per tree (Gabow and Myers (1978)).

In Infinite Graphs

1. Infinite Graph – Axiom of Choice: Every finite connected graph has a spanning tree. However, for infinite connected graphs, the existence of spanning trees is equivalent to the axiom of choice. An infinite graph is connected if every pair of its vertexes forms the pair of end-points of a finite path. As with finite graphs, a tree is a connected graph with no finite cycles, and a spanning tree can be defined either as a maximal acyclic set of edges or as a tree that contains every vertex (Serre (2003)).
2. Equivalence to Zorn's Lemma: The trees within a graph may be partially ordered by their subgraph relation, and infinite chain in this partial order has an upper bound, i.e., the union of the trees in the chain. Zorn's lemma, one of many equivalent statements to the axiom of choice, requires that a partial order in which all chains are upper bounded have a maximal element; in the partial order on the trees of the graph, this

maximal element must be a spanning tree. Therefore, if Zorn's lemma is assumed, every infinite connected graph has a spanning tree (Serra (2003)).

3. Spanning Tree as a Choice Function: In the other direction, given a family of sets, it is possible to construct an infinite graph such that every spanning tree of the graph corresponds to a choice function of the family of sets. Therefore, if every infinite graph has a spanning tree, then the axiom of choice is true (Soukup (2008)).

In Directed Multi-graphs

The idea of a spanning tree can be generalized to directed multigraphs (Levine (2011)). Given a vertex v on a directed multi-graph G, an *oriented spanning tree* T rooted at v is an acyclic subgraph of G in which every vertex other than v has an out-degree of 1. This definition is only satisfied when the *branches* of T point towards v.

References

- Aigner, M., and G. M. Ziegler (1998): *Proofs from THE BOOK* **Springer-Verlag**
- Bader, D. A., and G. Cong (2005): A Fast, Parallel Spanning Tree Algorithm for Symmetric Multi-processors (SMPs) *Journal of Parallel and Distributed Computing* **65 (9)** 994-1006
- Beinecke, L. W., and R. J. Wilson (2009): Topics in Topological Graph Theory *Encyclopedia of Mathematics and its Applications* **128 Cambridge University Press**
- Bollobas, B. (1998): *Modern Graph Theory – Graduate Texts in Mathematics* **184 Springer**
- Bondy, J. A., and U. S. R. Murty (2008): *Graph Theory – Graduate Texts in Mathematics* **244 Springer**
- Cameron, P. J. (1994): *Combinatorics; Topics, Techniques, Algorithms* **Cambridge University Press**

- de Fraysseix, H., and P. Rosenstiehl (1982): A Depth-First-Search Characterization of Planarity *Annals of Discrete Mathematics* **13** 75-80
- Eppstein, D. (1999): Spanning Trees and Spanners
- Gabow, H. N., and E. W. Myers (1978): Finding all Spanning Trees of Directed and Undirected Graphs *SIAM Journal on Computing* **7 (3)** 280-287
- Gallagher, R. G., P. A. Humblet, and P M. Spira (1983): *ACM Transactions on Programming Languages and Systems* **5 (1)** 66-77
- Gazit, H. (1991): An Optimal Randomized Parallel Algorithm for finding Connected Components in a Graph *SIAM Journal on Computing* **20 (6)** 1046-1067
- Goldberg, L. A., and M. Jerrum (2008): Inapproximability of the Tutte Polynomial *Information and Computation* **206 (7)** 908-929
- Graham, R. L., and P. Hell (1985): On the History of the Minimum Spanning Tree Problem *Annals of the History of Computing* **7 (1)** 43-57
- Gross, J. L., and J. Yellen (2005): *Graph Theory and its Applications 2nd Edition* **CRC Press**
- Harary, F., J. P. Hayes, and H. J. Wu (1988): A Survey of the Theory of Hypercube Graphs *Computers and Mathematics with Applications* **15 (4)** 277-289
- Hartsfield, N. and G. Ringel (2003): *Pearls in Graph Theory – A Comprehensive Introduction* **Courier Dover Publications**
- Jaeger, F., D. J. Vertigan, and D. J. A. Welsh (1990): On the Computational Complexity of the Jones and the Tutte Polynomials *Mathematical Proceedings of the Cambridge Philosophical Society* **108 (1)** 35-53
- Kocay, W., and D. L. Kreher (2004): *Discrete Mathematics and its Applications* **CRC Press**
- Kozen, D. (1992): *The Design and Analysis of Algorithms: Monographs in Computer Science* **Springer**
- McDiarmid, C., T. Johnson, and H. S. Stone (1997): On finding a Minimum Spanning Tree in a Network with Random Weights *Random Structures and Algorithms* **10 (1-2)** 187-204

- Mehlhorn, K. (1999): *Leda: A Platform for Combinatorial and Geometric Computing* **Cambridge University Press**
- Oxley, J. G. (2006): *Matroid Theory; Oxford Graduate Texts in Mathematics* **3 Oxford University Press**
- Reif, J. H. (1985): Depth-first Search is inherently Sequential *Information Processing Letter* **20 (5)** 229-234
- Serra, J. P. (2003): *Trees* **Springer Monographs in Mathematics**
- Soukup, L. (2008): Infinite Combinatorics: From Finite to Infinite *Horizon of Combinatorics, Bolyai Society of Mathematical Studies* **17** 189-113
- Wikipedia (2020): Spanning Tree
- Wilson, D. B. (1996): Generating Random Spanning more quickly than the Cover Time *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing (STOC '96)* 296-303
- Wu, B. Y., and K. M. Chao (2004): *Spanning Trees and Optimization Problems* **CRC Press**

Minimum Spanning Tree

Overview

1. Definition of Minimum Spanning Tree: A *minimum spanning tree (MST)* or *minimum weight spanning tree* is the subset of the edges of a connected, edge-weighted, undirected graph that connects all vertexes together, without any cycles and with minimum possible total edge weight. That is, it is a spanning tree whose sum of edges is as small as possible (Wikipedia (2020)).
2. Minimum Spanning Tree vs. Forest: More generally, any edge-weighted undirected graph – not necessarily connected – has a *minimum spanning forest*, which is a union of minimum spanning trees for its connected components.
3. Specialization of the Generic Spanning Tree: A *spanning tree* for that graph would be a subset of those paths that have no cycles but still connects every vertex; there might be several spanning trees possible. A *minimum spanning tree* would be the one with the lowest total cost, representing the least expensive path.

Multiplicity Properties

If there are n vertexes in the graph, then each spanning tree has $n - 1$ edges. There may be several minimum spanning trees of the same weight; in particular, if all the edges of a given graph are the same, then every spanning tree of that graph is a minimum.

Uniqueness Property

1. Statement of the Uniqueness Property: If each edge has a distinct weight, then there will be only one unique, minimum spanning tree. This generalizes to spanning forests as well.
2. Proof of the Uniqueness Property:
    a. Assume the contrary, that there are two different MST's – A and B.
    b. Since A and B differ despite containing the same nodes, there is at least one edge the belongs to one but not the other. Among such edges, let e_1 be the one with least weight; this choice is unique because the edge weights are all distinct. Without loss of generality, assume e_1 is in A.
    c. Since B is an MST, $\{e_1\} \cup B$ must contain a cycle C with e_1.
    d. As a tree, A contains no cycles, therefore C must have an edge e_2 that is not in A.
    e. Since e_1 was chosen as the unique lowest weight edge among those belonging to exactly one of A and B, the weight of e_2 must be greater than the weight of e_1.
    f. As e_1 and e_2 are part of the cycle C, replacing e_2 with e_1 in B therefore yields a spanning tree with smaller weight.
    g. This contradicts the assumption that B is an MST.

    More generally, of the edge weights are all not distinct, then only the multi-set of weights in minimum spanning trees is certain to be unique; it is the same for all minimum spanning trees.

Minimum Cost Subgraph Property

If the weights are *positive*, then a minimum spanning tree is in fact a minimum cost subgraph connecting all vertexes, since subgraphs containing cycles necessarily have more total weight.

Cycle Property

1. Statement of the Cycle Property: For any cycle C in the graph, if the weight of an edge e in C is larger than the individual weights of all other edges of C, then this edge cannot belong to an MST.
2. Proof of the Cycle Property: Assume the contrary, i.e., that e belongs to an MST T_1. Then deleting e will break T_1 into two subtrees with two ends of e in different subtrees. The remainder of C connects the subtrees, hence there is an edge f of C in different subtrees, i.e., it reconnects subtrees into a tree T_2 with weight less than that of T_1, because the weight f is less than that of weight e.

Cut Property

1. Statement of the Cut Property: For any cut C of the graph, if the weight of an edge e in the cut-set C is strictly smaller than the weight of all other edges in the cut-set of C, then this edge belongs to all MST's of the graph.
2. Proof - Case of Single Minimum Edge: Assume that there is an MST T that does not contain e. Adding e to T will produce a cycle that crosses the cut once at e and crosses back another edge e'. Deleting e' produces a spanning tree $T \setminus \{e'\} \cup \{e\}$ of strictly smaller weight than T. This contradicts the assumption that T was an MST.
3. Extension to Multiple Maximum Edges: By a similar argument. If more than one is of the same weight across the cut, then such edge is contained in some minimum spanning tree.

Minimum-Cost Edge Property

1. Statement of the Property: If the minimum cost edge e of a graph is unique, then this edge is included in any MST.
2. Proof of the Statement: If e was not included in the MST, removing any of the larger cost edges in the cycle formed after adding e to the MST would yield a spanning tree of smaller weight.

Contraction Property

If T is a tree of MST edges, then T can be contracted into a single vertex while maintaining the invariant that the MST of the contracted graph plus T gives the MST of the graph before contraction (Pettie and Ramachandran (2002a)).

Algorithms

1. Classical MST Algorithm #1 – Boruvka: The first algorithm for finding a minimum spanning tree was developed by Otakar Boruvka. In each stage, called the *Boruvka step*, it identifies a forest F consisting of the minimum-weight edge incident to each vertex in the graph G, then forms the graph

   $$G_1 = G \setminus F$$

   as the input to the next step. Here $G \setminus F$ denotes the graph derived from G by contracting edges in F – by the cut property, these edges belong to the MST. Each Boruvka step takes linear time on m. Since the number of vertexes is reduced by at least half in each step, Boruvka's algorithm takes $\mathcal{O}(m \log n)$ time (Pettie and Ramachandran (2002a)).
2. Classical MST Algorithm #2 - Prim: A second algorithm is Prim's algorithm, which grows the MST T one edge at a time. Initially, T contains an arbitrary vertex. In each

step, T is augmented with the least-weight edge (x, y) such that x is in T and y is not in T. By the cut property, all edges added to T are in the MST. Its runtime is either $\mathcal{O}(m \log n)$ or $\mathcal{O}(m + n \log n)$, depending on the structure used.

3. Classical MST Algorithm #3 - Kruskal: The third algorithm commonly in use is the Kruskal's algorithm, which also takes $\mathcal{O}(m \log n)$ time.
4. Classical MST Algorithm – Reverse-Delete: A fourth algorithm, not as commonly used, is the reverse-delete algorithm, which is the reverse of the Kruskal's algorithm. Its runtime is $\mathcal{O}(m \log n\, [\log \log n]^3)$
5. Greedy Algorithms with Polynomial Runtime: All these four are greedy algorithm. Since they run in polynomial time, the problem of finding such trees is in **FP**, and related decision problems such as finding whether an edge is in the MST or determining if the total minimum weight exceeds a certain value are in **P**.

Faster Algorithms

1. Hybrid Linear-Time Randomized Algorithm: In a comparison model, in which only allowed operations on edge-weights are pair-wise comparisons, Karger, Klein, and Tarjan (1995) found a linear time randomized algorithm based on a combination of the Boruvka's algorithm and the reverse-delete algorithm (Pettie and Ramachandran (2002b)).
2. Non-Randomized Comparison Based Algorithm: The fastest randomized comparison-based algorithm with known complexity, by Chazelle (2000a, 2000b), is based on soft-heap, and approximate priority queue. It's running time is $\mathcal{O}\big(E\ \alpha(E, V)\big)$ where $\alpha(E, V)$ is the classical functional inverse of the Ackermann function. The function $\alpha(E, V)$ grows extremely slowly, so that for all practical purposes it may be considered a constant no greater than 4, thus Chazelle's algorithm takes very close to linear time.

Linear-Time Algorithms in Special Cases – Dense Graphs

If the graph is dense, i.e.,

$$\frac{m}{n} \geq \log\log\log n$$

then a deterministic algorithm by Fredman and Tarjan (1987) finds the MAST in time $O(m)$. The algorithm executes in a number of phases. Each phase executes Prim's algorithm many times, each for a limited number of steps. The runtime for each phase is $O(m + n)$. If the number of vertexes before a phase is n', then the number of phases remaining after a phase is at most $\frac{n'}{2^{\frac{m}{n'}}}$. Hence, at most $\log n$ phases are needed, which gives a linear runtime for dense graphs (Pettie and Ramachandran (2002a)). There are other algorithms that work in linear-time on dense graphs (Gabow, Galil, Spencer, and Tarjan (1986), Chazelle (2000b)).

Linear Time Algorithm – Integer Weights

If the edge weights are integers represented in binary, then deterministic algorithms are known that solve the problem in $O(m + n)$ integer operations (Fredman and Willard (1994)). Whether the problem can be solved *deterministically* for a *general graph* in *linear time* by a comparison-based algorithm remains an open question.

Decision Trees

1. Idea behind the Decision Tree: Given graph G where the nodes and the edges are fixed but the weights are unknown, it is possible to construct a binary decision tree (DT) for calculating the MST for any permutation of weights. Each internal node of a

DT contains a comparison between two edges, i.e., “is the weight of the edge between x and y larger than that between w and z?” The two children of the node correspond to the two possible answers “yes” and “no”. In each leaf of the DT, there is a list of edges from G that correspond to an MST. The runtime complexity of the DT is the largest number of queries required to find the MST, which is just the depth of the DT. A DT for a graph G is called *optimal* if it has the smallest depth of all correct DT’s for G.

2. Steps for Determining Optimal Decision Trees: For every integer r, it is possible to find the optimal decision trees for all graphs on r vertexes by brute-force search. This search proceeds in two steps:
    a. Generate all potential DT’s
    b. Identify the correct DT’s
3. Generating all Potential DT’s:
    a. There are $2^{\binom{r}{2}}$ different graphs on r vertexes.
    b. For each graph an MST can always be found using $r(r-1)$ comparisons, e.g., by using Prim’s algorithm.
    c. Hence, the depth of an optimal DT is less than r^2.
    d. Hence, the number of internal nodes in an optimal DT is less than 2^{r^2}
    e. Every internal node compares two edges. The number of edges is at most r^2, so the different number of comparisons is at most r^4.
    f. Hence, the number of potential DT’s is less than

$$(r^4)^{2^{r^2}} = r^{2^{r^2+2}}$$

4. Identifying the correct Decision Trees: To check if a DT is correct, it must be checked on all possible permutations of the edge weights.
    a. The number of such permutations is at most $(r^2)!$
    b. For each permutation, the MST problem is solved on a given graph using any existing algorithm, and the result is compared to the answer given by the DT

c. The running time of any MST algorithm is at most r^2, so the total time required to check all permutations is at most $(r^2+1)!$

Hence the total time required for finding an optimal DT for *all* graphs with r vertexes is:

$$2^{\binom{r}{2}} \cdot r^{2^{r^2+2}} \cdot (r^2+1)! = 2^{2^{r^2+O(r)}}$$

(Pettie and Ramachandran (2002a)).

Optimal Algorithm

1. Provably Optimal Deterministic Comparison based MST: Pettie and Ramachandran (2002a) have constructed a provably optimal deterministic comparison based minimum spanning tree algorithm. The following is a simplified description of the algorithm steps:
2. Optimal Decision Trees on r Vertexes: Let

   $$r = \log\log\log n$$

   where n is the number of vertexes. Find all optimal decision trees on r vertexes. This can be done in $O(n)$ using the Decision Trees above.
3. Graph Partition - r Vertexes per Component: Partition the graph into components with at most r vertexes per each component. This partition uses a *soft heap*, which *corrupts* a small number of edges of the graph.
4. MST using DT in each Component: Use the optimal decision trees to find an MST for the uncorrupted subgraph within each component.
5. Contraction of the Connected Components: Contract each component spanned by the MST's to a single vertex, and apply any algorithm that works on dense graphs in time $O(m)$ to the contraction of the uncorrupted subgraph.

6. Adding back the Corrupted Edges: Add back the corrupted edges to the resulting forest to form the subgraph guaranteed to contain the minimum spanning tree, and smaller by a smaller factor than the starting graph. Apply the optimal algorithm recursively to this graph.
7. Runtime of the Algorithm: The runtime of all steps in the algorithm is $O(m)$, *except for the step of using decision trees*. The runtime of this step is not known, but is known to be optimal – no algorithm can do better than the optimal decision tree. Thus, this algorithm has a peculiar property that it is *provably optimal* although its runtime complexity is *unknown*.

Parallel and Distributed Algorithms

1. $O(\log n)$ Runtime in Parallel Environments: Research has also considered parallel algorithms for the MST problem. With a linear number of processors, it is possible to solve the problem in $O(\log n)$ time (Chong, Han, Tak (2001), Pettie and Ramachandran (2002c)). Bader and Cong (2006) demonstrate an algorithm that can compute the MST's 5 times faster on 8 processors than an optimized sequential algorithm.
2. Specialized Algorithms for Large Graphs: Other specialized algorithms have been designed for computing MST's of a graph so large that it must be stored on disk at all times. These *external storage* algorithms, for example described by Dementiev, Sanders, Schultes, and Sibeyn (2004), can operate, by the author's claims, as little as 2 to 5 times slower than a traditional in-memory algorithm. These rely on efficient external storage sorting algorithms and graph contraction techniques for reducing the graph's size efficiently.
3. Distributed Approaches for Calculating the MST: The problem can also be approached in a distributed manner. If each node is considered a computer and no node knows anything except its own connected links, one can still calculate the distributed MST.

MST on Complete Graphs

1. Complete Graphs with i.i.d. Weights: Alan M. Frieze showed that given a complete graph on n vertexes, with edge weights that are i.i.d. random variables with a distribution function F satisfying $F'(0) > 0$ then as n approaches $+\infty$ the expected weight of the MST approaches $\frac{\zeta(3)}{F'(0)}$, where ζ is Riemann Zeta function – more specifically, $\zeta(3)$ is the Apery's constant. Frieze and Steele also proved convergence in probability. Svante Janson proved a CLT for the weight of the MST.
2. MST Sizes for $U[0,1]$ Weights: For uniform random weights in $[0,1]$, the exact expected size of the MAST has been computed for small complete graphs (Steele (2002)).

Vertexes	Expected Size	Approximate Expected Size
2	$\frac{1}{2}$	0.5000000
3	$\frac{3}{4}$	0.7500000
4	$\frac{31}{35}$	0.8857143
5	$\frac{893}{924}$	0.9664502
6	$\frac{278}{273}$	1.0183151
7	$\frac{30739}{29172}$	1.0537160
8	$\frac{199462271}{184848378}$	1.0790588
9	$\frac{126510063932}{115228853025}$	1.0979027

Applications

1. Use in Network/Algorithm Construction: MST's have direct applications in the design of networks, including computer networks, telecommunication networks, transportation networks, water supply networks and electrical grids – which they were first invented for, as indicated above (Graham and Hell (1985)). They are invoked as sub-routines in algorithms for other problems, including the Christofides algorithm for approximating the traveling salesman problem (Christofides (1976)), approximating the multi-terminal minimum cut problem – which is equivalent in the single terminal case to the maximum flow problem (Dahlhaus, Johnson, Papadimitriou, Seymour, and Yannakakis (1994)), and approximating the minimum-cost weighted perfect matching (Supowit, Plaistead, and Reingold (1980)).
2. Practical Use of MSTs:
    a. Taxonomy (Sneath (1957))
    b. Cluster Analysis: Clustering points in the plane (Asano, Bhattacharya, Kiel, and Yao (1988)), single-linkage clustering – a method of hierarchical clustering (Gower and Ross (1969)), and clustering gene expression data (Paivinen (2005)).
    c. Constructing trees for broadcasting in computer networks (Dalal and Metcalfe (1978))
    d. Image Registration (Ma, Hero, Gorman, and Michel (2000)), and Segmentation (Felzenszwalb and Huttenlocher (2004))
    e. Curvilinear feature extraction in computer vision (Suk and Song (1984))
    f. Hand-writing Recognition of Mathematical Expressions (Tapia and Rojas (2004))
    g. Circuit Design – Implementing Efficient Multiple Constant Multiplications, as used in Finite Impulse Response Filters (Ohlsson (2004))

h. Regionalization of socio-geographic areas, the grouping of areas into homogenous, contiguous regions (Assuncao, Neves, Camara, and Da Costa Freitas (2006))

i. Comparing eco-toxicology data (Devillers and Doro (1989))

j. Topological Observability in Power Systems (Mori and Tsuzuki (1991))

k. Measuring Homogeneity of Two-dimensional Materials (Filliben, Kafadar, and Shier (1983))

l. Minimax Process Control (Kalaba (1963))

m. Minimum spanning trees can also be used to describe financial markets (Mantegna (1999), Djauhari and Gan (2015)). A correlation matrix can be created by calculating a coefficient of correlation between any two stocks. This matrix can be represented topologically as a complex network and an MST can be constructed to visualize relationships.

Related Problems

1. Subset Vertexes Steiner Tree: The problem of finding a Steiner tree of a subset of vertexes, that is, the minimum tree that spans a given subset, is known to be **NP**-complete (Garey and Johnson (1979)).
2. k-Minimum Spanning Tree: A related problem is the k minimum spanning tree (k-MST), which is the tree that spans some subset of k vertexes in a graph with minimum weight.
3. k Smallest Spanning Trees: A set of k *smallest spanning trees* is a subset of k spanning trees such that no spanning tree outside the subset has a smaller weight (Gabow (1997), Eppstein (1992), Frederickson (1997)). Note that this problem is unrelated to k minimum spanning tree.
4. Euclidean Minimum Spanning Tree: The Euclidean minimum spanning tree is a spanning tree of a graph with edge weights corresponding to the Euclidean distance between vertexes which are points in the plane – or space.

5. Rectilinear Minimum Spanning Tree: The rectilinear minimum spanning tree is a spanning tree of a graph with edge weights corresponding to the rectilinear distance between vertexes which are points in the plane – or space.
6. Distributed Minimum Spanning Tree: In the distributed model where each node is considered a computer and no node knows anything except its own connected links, one can consider distributed minimum spanning tree. The mathematical definition of the problem is the same, but there are different approaches for a solution.
7. Capacitated Minimum Spanning Tree: The capacitated minimum spanning tree is a tree that a marked node – origin, or root – and each of the subtrees attached to the node contains no more than c nodes. c is called the tree capacity. Solving the CMST optimally is **NP**-hard (Jothi and Raghavachari (2005)), but good heuristics such as Esau-Williams and Sharma produce solutions close to optimal in polynomial time.
8. Degree Constrained Minimum Spanning Tree: The degree constrained minimum spanning tress is a spanning tree in which each of the vertexes is connected to no more than d other vertexes for some given number d. The case $d = 2$ is a special case of traveling salesman problem, so the degree-constrained minimum spanning tree is **NP**-hard in general.
9. Directed Graph Minimum Spanning Tree: For directed graphs, the minimum spanning tree problem is called the Arboresence problem and can be solved in quadratic time using the Chu-Liu/Edmonds algorithm.
10. Maximum Spanning Tree: A *maximum spanning tree* is a spanning tree with weight greater than or equal to the weight of every other spanning tree. Such a tree can be found with algorithms such as Prim's or Kruskal's after multiplying the edge weights by -1 and solving the MST problem on the new graph. A path in the minimum spanning tree is the widest path in the graph between its two end-points; among all the possible paths, it maximizes the weight of the minimum-weight edge (Hu (1961)). Maximum spanning trees find applications in parsing algorithms for natural languages (McDonald, Pereira, Ribarov, and Hajic (2005)) and in training algorithms for conditional random fields.

11. Dynamic Minimum Spanning Tree: The *dynamic MST* problem concerns the update of a previously computed MST after an edge weight change in the original graph of the insertion/deletion of a vertex (Spira and Pan (1975), Chin and Houck (1978), Holm, de Lichtenberg, and Thorup (2001)).
12. Minimum Labeling Spanning Tree: The minimum labeling spanning tree problem is to find a spanning tree with the least type of labels if each edge in a graph is associated with a label from a finite label set instead of a weight (Chang and Leu (1997)).
13. Minimum Bottleneck Spanning Trees: A bottleneck is the highest weight edge in a spanning tree. A spanning tree is a *minimum bottleneck spanning tree* – or *MBST* – if the graph does not contain a spanning tree with a smaller bottleneck edge weight. An MST is necessarily an MBST – provable by the cut property – but an MBST is not necessarily an MST.

References

- Asano, T., B. Bhattacharya, M. Keil, and F. Yao (1988): Clustering Algorithms based on Minimum and Maximum Spanning Trees *4th Annual Symposium on Computational Geometry (SCG '88)* 252-257
- Assuncao, R. M., M. C. Neves, G. Camara, and C. Da Costa Freitas (2006): Efficient Regionalization Techniques for Socio-economic Geographical Units using Minimum Spanning Trees *International Journal of Geographical Information Science* **20 (7)** 797-811
- Bader, D. A., and G. Cong (2006): Fast Shared-memory Algorithms for Computing the Minimum Spanning Forests of Sparse Graphs *Journal of Parallel and Distributed Computing* **66 (11)** 1366-1378
- Chang, R. S., and S. J. Leu (1997): The Minimum Labeling Spanning Trees *Information Processing Letters* **63 (5)** 277-282

- Chazelle, B. (2000a): A Minimum Spanning Tree Algorithm with inverse-Ackermann Type Complexity *Journal of the ACM* **47 (6)** 1028-1047
- Chazelle, B. (2000b): The Soft-Heap: An Approximate Priority Queue with Optimal Error Rate *Journal of the ACM* **47 (6)** 1012-1027
- Chin, F., and D. Houck (1978): Algorithms for updating Minimal Spanning Trees *Journal of Computer and System Sciences* **16 (3)** 333-344
- Chong, K. W., Y. Han, T. W. Lam (2001): Concurrent Threads and Optimal Minimum Spanning Tree Algorithm *Journal of the ACM* **48 (2)** 297-323
- Christofides, N. (1976): Worst-case Analysis of a new Heuristic for the Traveling Salesman Problem
- Dahlhaus, E., D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis (1994): The Complexity of Multi-dimensional Cuts *SIAM Journal on Computing* **23 (4)** 864-894
- Dalal, Y. K., and R. M. Metcalfe (1978): Reverse Path-forwarding of Broadcast Packets *Communications of the ACM* **21 (12)** 1040-1048
- Dementiev, R., P. Sanders, D. Schultes, and J. F. Sibeyn (2004): Engineering an External Memory Spanning Tree Algorithm *Proceedings of the IFIP 18th World Computer Congress, TC1 3rd International Conference on Theoretical Computer Science (TCS2004)* 195-208
- Devillers, J., and J. C. Doro (1989): Heuristic Potency of the Minimum Spanning Tree (MST) Method in Toxicology *Ecotoxicology and Environmental Safety* **17 (2)** 227-235
- Djauhari, M. A., and S. L. Gan (2015): Optimality Problem of Network Topology in Stock Market Analysis *Physica A* **419** 108-114
- Eppstein, D. (1992): Finding the *k* Smallest Spanning Trees *BIT* **32 (2)** 237-248
- Felzenszwalb, P. F., and D. P. Huttenlocher (2004): Efficient Graph-Based Image Segmentation *International Journal of Computer Vision* **59** 167-181
- Filliben, J. J., K. Kafadar, and D. R. Shier (1983): Testing for Homogeneity of Two-dimensional Surfaces *Mathematical Modeling* **4 (2)** 167-189

- Frederickson, G. N. (1997): Ambivalent Data Structures for Dynamic Two-edge Connectivity and *k* Smallest Spanning Trees *SIAM Journal on Computing* **26 (2)** 484-538
- Fredman, M. L., and R. E. Tarjan (1987): Fibonacci Heaps and their use in improved Network Optimization Algorithms *Journal of the ACM* **34 (3)** 596-615
- Fredman, M. L., and D. E. Willard (1994): Trans-dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths *Journal of Computer and System Sciences* **48 (3)** 533-551
- Gabow, H. N. (1977): Two Algorithms for generating Weighted Spanning Trees in Order *SIAM Journal of Computing* **6 (1)** 139-150
- Gabow, H. N., Z. Galil, T. Spencer, and R. E. Tarjan (1986): Efficient Algorithms for finding Minimum Spanning Trees in Undirected and Directed Graphs *Combinatorica* **6 (2)** 109-122
- Garey. M. R., and D. S. Johnson (1979): *Computer and Intractability: A Guide to the Theory of NP-Completeness* **W. H. Freeman**
- Gower, J. C., and G. J. S. Ross (1969): Minimum Spanning Tree and Single-Linkage Cluster Analysis *Journal of the Royal Statistical Society C (Applied Statistics)* **18 (1)** 54-64
- Holm. J., K. de Lichtenberg, and M. Thorup (2001): Poly-logarithmic, Deterministic, Fully Dynamic Algorithms for Connectivity, Minimum Spanning Tree, Two-edge, and Bi-connectivity *Journal of the ACM* **48 (4)** 723-760
- Jothi, R., and B. Raghavachari (2005): Approximation Algorithms for the Capacitated Minimum Spanning Tree Problem and its Variants in Network Design *ACM Transactions on Algorithms* **1 (2)** 265-282
- Kalaba, R. E. (1963): Graph Theory and Automatic Control
- Karger, D. R., P. N. Klein, and R. E. Tarjan (1995): A Randomized Minimum-tree Algorithm to find Minimum Spanning Trees *Journal of the ACM* **42 (2)** 321-328
- Ma, B., A. Hero, J. Gorman, and O. Michel (2000): Image Registration with Minimum Spanning Tree Algorithm

- Mantegna, R. N. (1999): Hierarchical Structure in Financial Markets *The European Physical Journal B – Condensed Matter and Complex Systems* **11 (1)** 193-197
- McDonald, R., F. Pereira, K. Ribarov, and J. Hajic (2005): Non-projective Dependency Parsing using Spanning Tree Algorithms
- Mori, H., and S. Tsuzuki (1991): A fast Method for Topological Observability Analysis using a Minimum Spanning Tree Technique *IEEE Transactions on Power Systems* **6 (2)** 491-500
- Ohlsson, H. (2004): Implementation of Low Complexity FIR Filters using a Minimum Spanning Tree *12th IEEE Mediterranean Electro-technical Conference (MELECON 2004)* 261-264
- Paivinen, N. (2005): Structuring with a Minimum Spanning Tree of Scale-free-like Structure *Pattern Recognition Letters* **26 (7)** 921-930
- Pettie, S., and V. Ramachandran (2002a): An Optimal Minimum Spanning Tree Algorithm *Journal of the ACM* **49 (1)** 16-34
- Pettie, S., and V. Ramachandran (2002b): Maximizing Randomness in Minimum Spanning Tree, Parallel Connectivity, and Set Maxima Algorithms *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms (SODA '02)* 713-722
- Pettie, S., and V. Ramachandran (2002c): A Randomized Time-Work Optimal Parallel Algorithm for finding a Minimum Spanning Forest *SIAM Journal on Computing* **31 (6)** 1879-1895
- Sneath, P. H. A. (1957): The Application of Computers to Taxonomy *Journal of Microbiology* **17 (1)** 201-226
- Spira, P. M., and A. Pan (1975): On finding and updating Spanning Trees and Shortest Paths *SIAM Journal on Computing* **4 (3)** 375-380
- Steele, M. J. (2002): Minimum Spanning Trees for Graphs with Random Edge Lengths *Mathematics and Computer Science II* 223-245 **Birkhauser** Basel
- Suk, M. and O. Song (1984): Curvilinear Feature Extraction using Minimum Spanning Trees *Computer Vision, Graphics, and Image Processing* **26 (3)** 400-411

- Supowit, K. J., D. A. Plaistead, E. M. Reingold (1980): Heuristics for Weighted Perfect Matching *12th Annual ACM Symposium on Theory of Computing (STOC '80)* 398-419
- Tapia, E., and R. Rojas (2004): Recognition of On-line Handwritten Mathematical Expressions Using a Minimum Spanning Tree Construction and Symbol Dominance
- Wikipedia (2020): Minimum Spanning Tree

Prim's Algorithm

Overview

1. Purpose of the Prim's Algorithm: Prim's algorithm – also known as Jarnik's algorithm – is a greedy algorithm that finds the minimum spanning tree for a weighted, undirected graph. This means that it finds the subset of the edges that form a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex (Wikipedia (2019)).
2. Developers of the Algorithm: The algorithm was developed by the Czech mathematician Jarnik (1930) and later re-discovered and re-published by Prim (1957) and Dijkstra (1959). Therefore, it is also sometimes called the *Jarnik's algorithm* (Sedgewick and Wayne (2011)), *Prim-Dijkstra algorithm* (Cheriton and Tarjan (1976)), *Prim-Jarnik algorithm* (Rosen (2011)), or the *DJP algorithm* (Pettie and Ramachandran (2002)).
3. Comparison with Kruskal's and Boruvska's Algorithms: Other well-known algorithms for this problem include Kruskal's algorithm and Boruvska's algorithm (Tarjan (1983)). These algorithms find a minimum spanning forest in a possibly disconnected graph; in contrast, the most basic form of the Prim's algorithm only finds minimum spanning trees in connected graphs. However, by running Prim's algorithm separately for each connected component of the graph, it can also be used to find the minimum spanning forest (Kepner and Gilbert (2011)). In terms of their asymptotic time complexity, these three algorithms are equally fast for sparse graphs, but slower than other more sophisticated algorithms (Cheriton and Tarjan (1976), Pettie and Ramachandran (2002)). However, for graphs that are sufficiently dense,

Prim’s algorithm can be made to run in linear time, meeting or improving the time bounds for other algorithms (Tarjan (1983)).

Description

1. Overview of the Algorithm Steps: The algorithm may be informally described as performing the following steps:
    a. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
    b. Grow the tree by one edge. Of the edges that connect the tree to vertices not yet in the tree, find the minimum weight edge, and transfer it to the tree.
    c. Repeat the previous step until all the vertices are in the tree.

    In more detail, it may be implemented following the pseudocode below.
2. Initialization of Edges and Costs: Associate each vertex v of the graph with a number $C[v]$ - the cheapest cost of connection to v - and an edge $E[v]$ - the edge providing the cheapest connection. To initialize these values, set all values of $C[v]$ to $+\infty$ - or to any number larger than the maximum edge weight – and set each $E[v]$ to a special flag indicating that there is no edge connecting v to earlier vertices.
3. Initializing the Forest and the Vertices: Initialize an empty forest F and a set Q of vertices that have not yet been included in F – initially all vertices.
4. Iteration over the Queue Elements: Repeat the following steps until the Q is empty:
    a. Find and remove a vertex v from the Q having the minimum possible value of $C[v]$
    b. Add v to F and, if $E[v]$ is not the special flag value, also add $E[v]$ to F
    c. Loop over the edges vw connecting v to other vertices w. For each such edge, if w still belongs to Q and vw has a smaller weight than $C[w]$, perform the following steps:
        i. Set $C[w]$ to the cost of edge vw
        ii. Set $C[w]$ to point to edge vw
5. Retrieve the Forest containing the MST’s: Return F

6. <u>Starting Vertex for the Algorithm</u>: As described above, the starting vertex for the algorithm will be chosen arbitrarily, because the first iteration of the main loop will have a set of vertices in Q that all the same weight, and the algorithm will automatically start a new tree in $F$when it completes the spanning tree of each connected component of the input graph. The algorithm may be modified to start with any particular vertex s by setting $C[s]$ to be a number smaller than other values of C – for instance, zero – and it may be modified to find only a single spanning tree rather than an entire spanning forest – matching more closest the informal description – by stopping whenever it encounters another vertex flagged as having no associated edge.
7. <u>Choices for implementing the Queue</u>: Different variations of the algorithm differ from each other in how the object Q is implemented: as a simple linked-list, as an array of vertexes, or as a more complicated priority queue data structure. The choices lead to differences in time complexity of the algorithm. In general, a priority queue will be much quicker at finding the vertex v with minimum cost, but will entail more expensive updates when the value of $C[v]$ changes.

Time Complexity

1. <u>Determinants of the Time Complexity</u>: The time complexity for the Prim's algorithm depends on the data structures used for the graphs and for ordering the edges by weight, which can be done using a priority queue. The table below shows the typical choices.
2. <u>Table Time Complexity by Algorithm</u>:

Minimum Edge Weight Data Structure	**Total Time Complexity**
Adjacency Matrix, Searching	$O(\|V\|^2)$
Binary Heap and Adjacency List	$O([\|V\| + \|E\|] \log\|V\|) = O(\|E\| \log\|V\|)$
Fibonacci Heap and Adjacency List	$O([\|E\| + \|V\|] \log\|V\|)$

3. Implementation using Adjacency Matrix/List: A simple implementation of Prim's, using an adjacency matrix or an adjacency list graph representation and linearly using an array of weights to find the minimum weight edge to add, requires $\mathcal{O}(|V|^2)$ running time. However, this running time can be greatly improved further by using heaps to implement finding minimum weight edges in the algorithm's inner loop.
4. Heap Based Edge Weight Ordering: A first improved version uses a heap to store all edges of the input graph, ordered by their weight. This leads to an $\mathcal{O}(|E| \log|E|)$ worst-case running time. But storing vertexes instead of edges can improve it still further. The heap should order their vertexes by their smallest edge weight that connects them to any vertex in a partially constructed minimum spanning tree (MST) – or ∞ if no such edge exists. Every time a vertex v is chosen and added to the MST, a decrease-key operation is performed on all vertexes w, setting the key to the minimum of its previous value and the edge cost of (v, w).
5. Impact of Denseness and Queue Implementation: Using a simple binary heap data structure, Prim's algorithm can be shown to run in time $\mathcal{O}(|E| \log|V|)$ where $|E|$ is the number of edges and $|V|$ is the number of vertexes. Using a more sophisticated Fibonacci heap, this can be brought down to $\mathcal{O}([|E| + \ |V|] \log|V|)$ which is asymptotically faster when the graph is dense enough that $|E|$ is $\omega(|V|)$, and linear time when is at least $\mathcal{O}(|V| \log|V|)$. For graphs of even greater density, having at least $|V|^c$ edges for some

$$c > 1$$

Prim's algorithm can be made to run in linear time even more simply, by using a d-ary heap in place of a Fibonacci heap (Johnson (1975), Tarjan (1983)).

Proof of Correctness

1. Basic Thrust of the Algorithm: Let $\boldsymbol{P}$ be a connected, weighted graph. At every iteration of Prim's algorithm, an edge must be found that connects a vertex in the subgraph to a vertex outside the subgraph. Since $\boldsymbol{P}$ is connected, there will always be a path to every vertex. The output Y of Prim's algorithm is a tree, because the edge and the vertex added to Y are connected.
2. An Alternate Minimum Spanning Tree: Let Y_1 be a minimum spanning tree of the graph $\boldsymbol{P}$. If

   $$Y_1 = Y$$

   then Y is a minimum spanning tree. Otherwise, let e be the first edge added during the construction of tree Y that is not in Y_1, and let V be the set of vertexes connected by edges added before edge e. Then one endpoint of edge e is in set V and the other is not.
3. Differences between the Current and the Alternate MSTs: Since tree Y_1 is a spanning graph of $\boldsymbol{P}$, there is a path in tree Y_1 joining the two endpoints. As one travels along the path, one encounters an edge f joining as vertex in set V to one that is not in V. Now, at the iteration where edge e was added to tree Y, edge f could have also been added, and it would have been added instead of edge e if its weight was less than e. Since it was not added, it may be concluded that

   $$w(f) \geq w(e)$$

4. Reconstructing Current from Alternate MST: Let tree Y_2 be a graph obtained by removing edge f and adding edge e to the tree Y_1. It is easy to show that tree Y_2 is connected, has the same number of edges as tree Y_1, and the total weight of its edges is not larger than that of tree Y_1, therefore it is also a minimum spanning tree of graph $\boldsymbol{P}$, and it contains e and all the edges added to before it during the construction of set V.

5. Metrics Comparison between the MSTs: On repeating the steps above, eventually a minimum spanning tree of graph $\boldsymbol{P}$ that is identical to tree Y is obtained. This shows that Y is a minimum spanning tree. The minimum spanning allows for the first subset of the first subregion to be expanded into a smaller subset X, which is assumed to be minimum.

Parallel Algorithm

1. Parallelizable Component of the Prim's Algorithm: The main loop pf the Prim's algorithm is inherently sequential and thus not parallelizable. However, the inner loop, which determines the next edge of the minimum weight that does not form a cycle, can be parallelized by dividing the vertexes and edges between the available processors (Grama, Gupta, Karypis, and Kumar (2003)). The pseudocode below demonstrates this.
2. Partition the Vertexes among the Processors: Assign each processor P_i a set V_i of consecutive vertexes of length $\frac{|V|}{|P|}$.
3. Dividing the Edges among the Processors: Create C, E, F, and Q as in the sequential algorithm above and divide C and E as well as the graph between all the processors such that each processor holds the incoming edges to its set of vertexes. Let C_i, E_i denote the parts of C and E stored on processor P_i.
4. Local Minimum Vertexes and their Eventual Union: Repeat the following steps until Q is empty:
    a. On every processor, find the vertex v_i having the minimum value in $C_i[v_i]$ - the local solution.
    b. Min-reduce the local solution to find the vertex v having the minimum possible value of $C[v]$ – the global solution.
    c. Broadcast the selected node to every processor.
    d. Add v to F, and if $E[v]$ is not special value flag, add $E[v]$ to F.

   e. On every processor, update C_i and E_i as in the sequential algorithm.

5. Return the Forest containing the MSTs: Return F.
6. Performance of the Parallelized Version: This algorithm can generally be implemented on a distributed machine (Grama, Gupta, Karypis, and Kumar (2003)) as well as on shared memory machines (Quinn and Deo (1984)). It has also been implemented in graphical processing units (GPUs) (Wang, Huang, and Guo (2011)). The running time is $\mathcal{O}\left(\frac{|V|^2}{|P|}\right) + \mathcal{O}(|V|\log|P|)$, assuming that the *reduce* and the *broadcast* operations can be performed in $\mathcal{O}(\log|P|)$ (Grama, Gupta, Karypis, and Kumar (2003)). A variant of the Prim's algorithm for shared memory machines, in which Prim's sequential algorithm is being run in parallel, starting from different vertexes, has also been explored (Setia, Nedunchezhian, and Balachandran (2015)). It should, however, be noted that more sophisticated algorithms exist to solve the distributed minimum spanning tree problem in a more efficient manner.

References

- Cheriton, D., and R. E. Tarjan (1976): Finding Minimum Spanning Trees *SIAM Journal on Computing* **5 (4)** 724-742
- Dijkstra, E. W. (1959): A Note on Two Problems in Connexion with Graphs *Numerische Mathematik* **1 (1)** 269-271
- Grama, A., A. Gupta, G. Karypis, and V. Kumar (2003): *Introduction to Parallel Computing 2nd Edition* **Addison Wesley**
- Jarnik, V. (1930): O Jistem Problemu Minimalnim *Prace Moravske Prirodovedecke Spolecnosti* **6 (4)** 57-63
- Johnson, D. (1975): Priority Queues with Updates and Finding Minimum Spanning Trees *Information Processing Letters* **4 (3)** 53-57
- Kepner, J., and J. Gilbert (2011): *Graph Algorithms in the Language of Linear Algebra* **Society for Industrial and Applied Mathematics**

- Pettie, S., and V. Ramachandran (2002): An Optimal Minimum Spanning Tree Algorithm *Journal of the ACM* **49 (1)** 16-34
- Prim, R. C. (1957): Shortest Connection Networks and some Generalizations *Bell System Technical Journal* **36 (6)** 1389-1401
- Quinn, M. J., and N. Deo (1984): Parallel Graph Algorithms *ACM Computing Surveys* **16 (3)** 319-348
- Rosen, K. (2011): *Discrete Mathematics and its Application 7th Edition* **McGraw-Hill Science**
- Sedgewick, R. E., and K. D. Wayne (2011): *Algorithms 4th Edition* **Addison-Wesley**
- Setia, R., A. Nedunchezhian, and S. Balachandran (2015): A New Parallel Algorithm for Minimum Spanning Tree Problem
- Tarjan, R. E. (1983): *Data Structures and Network Algorithms* **Society for Industrial and Applied Mathematics**
- Wang, W., Y. Huang., and S. Guo (2011): Design and Implementation of GPU-Based Prim's Algorithm *International Journal of Modern Education and Computer Science* **4** 55-62
- Wikipedia (2019): Prim's Algorithm

Kruskal's Algorithm

Introduction

1. Principal Idea behind Kruskal's Algorithm: *Kruskal's algorithm* (Kruskal (1956), Wikipedia (2020)) is a minimum spanning tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest (Cormen, Leiserson, Rivest, and Stein (2009)).It is a greedy algorithm in graph theory as it finds a minimum spanning tree for a connected, weighted graph adding increasing cost arc at each step. This means that it finds the subset of edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. If the graph is not connected, then it finds a *minimum spanning forest* – a minimum spanning tree for each component.
2. Alternate Algorithms for Extracting MSFs: Other algorithms for this problem include Prim's algorithm, reverse-delete algorithm, and Boruvka's algorithm.

The Algorithm

1. Forest with Vertexes Per Tree: Create a forest F - a set of trees – where each vertex in the graph is a separate tree.
2. Set of all Graph Edges: Create a set S containing all the edges in the graph.
3. Edge-Based Processing and Tree Update: While S is not empty and F is not yet spanning:
    a. Remove and edge with minimum weight from S
    b. If the removed edge connects two different trees, then add it to the forest F, combining the two trees into a single tree.

4. Minimum Spanning Tree and Forest: At the termination of the algorithm, the forest forms a set of minimum spanning trees of the graph. If the graph is connected, the forest has a single component, and forms a minimum spanning tree.

Complexity

1. Asymptotic Bounds Using $|E|/|V|$: Kruskal's algorithm can be shown to run in $\mathcal{O}(|E| \log|E|)$ or equivalently, in $\mathcal{O}(|E| \log|V|)$, where $|E|$ is the number of edges in the graph and $|V|$ is the number of vertexes, all using simple data structures. These running times are equivalent because:
    a. $|E|$ is at most $|V|^2$ and

    $$\log|V|^2 = 2 \log|V|$$

    is $\mathcal{O}(\log|V|)$

    b. Each isolated vertex is a separate component of the minimum spanning forest. If one ignores isolated vertexes, one obtains

    $$|V| \leq 2|E|$$

    so $\log|V|$ is $\mathcal{O}(\log|E|)$.
2. Rationale behind the Bound Estimate: This bound may be achieved as follows. First, sort the edges by weight using a comparison sort in $\mathcal{O}(|E| \log|E|)$ time; this allows the step that removes an edge with minimum weight from S to operate in constant time. Next, a disjoint-set data structure is used to keep track of which vertexes are in which components. This needs $\mathcal{O}(|V|)$ operations; since in each iteration where a vertex is connected to a spanning tree, two *find* operations and possibly one union for each edge are needed. Even a simple disjoint-set data structure such as disjoint set

forests with union by rank can perform $\mathcal{O}(|V|)$ operations in $\mathcal{O}(|V|\log|V|)$ time. Thus, the total time is

$$\mathcal{O}(|E|\log|E|) = \mathcal{O}(|E|\log|V|)$$

3. Sophisticated Disjoint Sets and Sorters: Provided that the edges are already sorted or can be sorted in linear time – for example, with counting sort or radix sort, the algorithm can use a more disjoint set data structure to run in $\mathcal{O}\big(|E|\alpha(|V|)\big)$, where α is an extremely slowly growing inverse of the single-valued Ackermann function.

Proof of Correctness

The proof consists of two parts. First, it is proved that the algorithm produces a spanning tree. Second, it is proved that the constructed tree is of minimal weight.

Spanning Tree

Let G be a connected, weighted graph, and Y be a subgraph produced by the algorithm. Y cannot have a cycle, being within one subtree and not between two different trees. Y cannot be disconnected, since the first encountered edge that joins the two components of Y would have been added by the algorithm. Thus Y is a spanning tree of G.

Minimality

1. Proposition: Existence of an MST: It may be shown, using induction, that the following proposition $\boldsymbol{P}$ by induction is true; if F is the set of edges at any stage in the algorithm, then there is some minimum spanning tree that contains F.

2. Induction Proof - Validity at Start: Clearly $\boldsymbol{P}$ is tree at the beginning when F is empty; any spanning tree will do, and there exists one, because a connected weighted graph always has a minimum spanning tree.
3. Inductive Proof - Intermediate Stage Validity: Now assume that $\boldsymbol{P}$ is true for some non-final edge state F and let T be a minimum spanning tree that contains F.
    a. If the next chosen edge e is also in T, then $\boldsymbol{P}$ is true for $F + e$.
    b. Otherwise, if e is not in T, then $T + e$ has a cycle C. This cycle contains edges that do not belong to F, since E does not form a cycle when added to F but does in T. Let f be an edge which is in C but not in $F + e$. Note that F also belongs to T, and by $\boldsymbol{P}$ has not been considered by the algorithm. f must therefore haver a weight at least a large as e. Then $T - f + e$ is a tree, and it has the same or less weight as T. So $T - f + e$ is a minimum spanning tree containing $F + e$ and again $\boldsymbol{P}$ holds.
4. Completing the Inductive Proof: Therefore, by the principle of induction, $\boldsymbol{P}$ holds when F has become a spanning tree, which is only possible if F is a minimum spanning tree itself.

Parallel Algorithm

1. Strategies for Parallelizing the Algorithm: Kruskal's algorithm is inherently sequential and hard to parallelize. It is, however, possible to perform the initial sorting of the edges in parallel, or alternatively, to use a parallel implementation of the binary heap to extract the minimum-weight edge in every iteration (Quinn and Deo (1984)). As parallel sorting is possible in time $\mathcal{O}(n)$ on $\mathcal{O}(\log n)$ processors (Grama, Gupta, Karypis, and Kumar (2003)), the runtime of the Kruskal's algorithm can be reduced to $\mathcal{O}\big(|E|\alpha(|V|)\big)$, where α is again the inverse of the single-valued Ackermann function.
2. The Filter-Kruskal Parallel Version: The variant of Kruskal's algorithm, named Filter-Kruskal, has been described by Osipov, Sanders, and Singler (2009) and is

better suited for parallelization. The basic idea behind Filter-Kruskal is to partition the edges in a way similar to quicksort and to filter out the edges that connect the vertexes of the same tree to reduce the cost of sorting.

3. Advantages of the Filter-Kruskal Scheme: Filter-Kruskal lends itself better for parallelization as sorting, filtering, and partitioning can be performed easily by distributing the edges between the processors (Osipov, Sanders, and Singler (2009)).
4. Other Approaches to Kruskal Parallelization: Finally, other variants of a parallel implementation of Kruskal's algorithm have been explored. Examples include a scheme to that uses helper threads to remove edges that are definitely not part of the MST in the background (Katsigiannis, Anastopoulos, Konstantinos, and Koziris (2012)), and a variant that runs the sequential algorithm in p subgraphs, and then merges those subgraphs until only one, the final MST, remains (Loncar, Skrbic, and Balaz (2014)).

References

- Cormen, T., C. E. Leiserson, R. Rivest, and C. Stein (2009): *Introduction to Algorithms 3rd Edition* **MIT Press**
- Grama, A., A. Gupta, G. Karypis, and V. Kumar (2003): *Introduction to Parallel Computing 2nd Edition* **Addison Wesley**
- Katsigiannis, A., N. Anastopoulos, K. Nikas, and N. Koziris (2012): An Approach to Parallelize Kruskal's Algorithm using Helper Threads
- Kruskal, J. B. (1956): On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem *Proceedings of the American Mathematical Society* **7 (1)** 48-50
- Loncar, V., S. Skrbic, and A. Balaz (2014): Parallelization of Minimum Spanning tree Algorithms using Distributed Memory Architectures *Transactions on Engineering Technologies* 543-554
- Osipov, V., P. Sanders, and J. Singler (2009): The Filter-Kruskal Minimum Spanning Tree Algorithm

- Quinn, M. J., and N. Deo (1984): Parallel Graph Algorithms *ACM Computing Surveys* **16 (3)** 319-348
- Wikipedia (2020): Kruskal's Algorithm

Boruvka's Algorithm

Overview

1. Definition of Boruvka's Algorithm: Boruvka's algorithm is a greedy algorithm for finding a minimum spanning tree in a graph for which all edge weights are distinct, or a minimum spanning forest in case of a graph that is not connected (Wikipedia (2019)).
2. Principal Steps behind the Algorithm: The algorithm begins by finding the minimum-weight edge incident to each vertex of the graph, and adding all of the edges to the forest. Then, it repeats a similar process of finding the minimum-weight edges from each tree constructed so far to a different tree, and adding all of these edges to the forest. Each repetition of this process reduces the number of trees, within each connected component of the graph, to at most half of the former value, so after logarithmically many repetitions the process finishes. When it does, the set of edges it has added forms the minimum spanning forest.

Special Cases

If the edges do not have distinct weights, a consistent tie-breaking rule, i.e., breaking ties by the object identifiers of the edge, can be used. An optimization is to remove G from each edge that is found to connect two vertexes in the same component as each other.

Complexity

Boruvka's algorithm can be shown to take $\mathcal{O}(\log V)$ iterations of the outer loop until it terminates, and therefore to run in time $\mathcal{O}(E \log V)$. In planar graphs, and more generally in families of graphs closed under graph minor operations, it can be made to run in linear time, by removing all but the cheapest edge between each pair of components after each stage of the algorithm (Eppstein (1999), Mares (2004)).

Other Algorithms

1. Prim's and Kruskal's MST Algorithms: Other algorithms for this problem include Prim's algorithm and Kruskal's algorithm. Fast parallel algorithms can be obtained by combining Prim's algorithm with Boruvka's (Bader and Cong (2006)).
2. Fast Randomized and Deterministic Algorithms: A faster randomized MST algorithm based in part on Boruvka's algorithm dur to Karger, Klein, and Tarjan (1995) runs in $\mathcal{O}(E)$ time. The best known minimum spanning tree algorithm by Chazelle (2000) is also based in part on Boruvka's and runs in $\mathcal{O}\big(E\ \alpha(E, V)\big)$ time, where α is the inverse of the Ackermann's function. These randomized and deterministic algorithms combine steps of the Boruvka's algorithm, reducing the number of components that need to be connected, with steps of a different type that reduce the number of edges between pairs of components.

References

- Bader, D. A., and G. Cong (2006): Fast Shared-memory Algorithms for Computing the Minimum Spanning Forests of Sparse Graphs *Journal of Parallel and Distributed Computing* **66 (11)** 1366-1378
- Chazelle, B. (2000): A Minimum Spanning Tree Algorithm with inverse-Ackermann Type Complexity *Journal of the ACM* **47 (6)** 1028-1047
- Eppstein, D. (1999): Spanning Trees and Spanners

- Karger, D. R., P. N. Klein, and R. E. Tarjan (1995): A Randomized Minimum-tree Algorithm to find Minimum Spanning Trees *Journal of the ACM* **42 (2)** 321-328
- Mares, M. (2004): Two Linear-time Algorithms for MST on Minor Closed Graph Cases *Archivum Mathematicum* **40 (3)** 315-320
- Wikipedia (2020): Boruvka's Algorithm

Reverse-Delete Algorithm

Overview

1. Purpose of the Reverse-Delete Algorithm: The reverse-delete algorithm is an algorithm in graph theory used to obtain a minimum spanning tree from a given connected, edge-weighted graph. It first appeared in Kruskal (1956), but it should not be confused with the Kruskal algorithm, which appears in the same publication. If the graph is disconnected, the algorithm will find a minimum spanning tree for each disconnected part of the graph. The set of these minimum spanning trees is called a minimum spanning forest, which contains every vertex in the graph Wikipedia (2019)).
2. Greedy Nature of the Algorithm: The algorithm is a greedy algorithm. choosing the best choice given any situation. It is the reverse of Kruskal's algorithm, which is another greedy algorithm to find a minimum spanning tree. Kruskal's algorithm starts with an empty graph and adds edges while the reverse-delete algorithm starts from an empty graph and deletes edges from it.
3. Steps in Reverse-Delete Algorithm:
    a. Start with a graph G, which contains a list of edges E.
    b. Go through E in decreasing order of edge weights.
    c. For each edge, check if deleting the edges will further disconnect the graph.
    d. Perform any deletion that does not lead to additional disconnection.

Running Time

The algorithm can be shown to run in $\mathcal{O}(E \log V \,[\log \log V]^3)$ using the big-$\mathcal{O}$ notation, where E is then number of edges and V is the number of vertexes. This bound is achieved as follows:

a. Sorting the edges by weight using comparison sort takes $\mathcal{O}(E \log E)$ time, which can be simplified to $\mathcal{O}(E \log V)$ using the fact that the largest E can be is V^2.
b. There are E iterations in the loop.
c. Deleting an edge, checking the continuity in the resulting graph, and, if it is disconnected, re-inserting the edge, can be done in $\mathcal{O}(\log V \,[\log \log V]^3)$ time per operation (Thorup (2000)).

Proof of Correctness: Approach

The proof consists of two parts. First, it is proved that the edges that remain after the algorithm is applied form a spanning tree. Second, it is proved that the spanning tree is of minimal weight.

Proof of Correctness: Part One

The remaining subgraph g produced by the algorithm is not disconnected since the algorithm checks for that. The result subgraph cannot contain a cycle, since if it does, a move along the edges would encounter the maximum edge in the cycle and that would be deleted. Thus, g must be a spanning tree of the main graph G.

Proof of Correctness: Part Two

1. Inductive Proof for the Algorithm: This part shows that the following proposition $\boldsymbol{P}$ is true by induction; if F is the set of edges that remain at the end of the edge-removal loop, then there is some minimum spanning tree with edges that are a subset of F.
2. Starting Point for the Induction: Clearly $\boldsymbol{P}$ holds before the start of the edge-removal loop. Since a weighted, connected graph always has a minimum spanning tree, and since F contains all the edges of the graph, the minimum spanning tree must be a subset of F.
3. Validity for an Intermediate Edge Set: Assume $\boldsymbol{P}$ is true for some non-final edge set F and let T be a minimum spanning tree that is contained in F. It needs to be shown that, after deleting the edge e in the algorithm, there exists some - possible different – spanning tree T' that is a subset of F.
4. Edge not a Member of T: If the next deleted edge e does not belong to T, then

   $$T = T'$$

   is a subset of F, and $\boldsymbol{P}$ continues to hold.
5. Edge a Member of T: Otherwise e belongs to T; first, note that the algorithm only removes edges that do not cause a disconnected ness in F. Thus, e does not cause disconnectedness. However, since e is a member of T, deleting e causes a disconnectedness in T. Assume that e separates T into subgraphs T_1 and T_2. Since the whole graph is connected after deleting e, there must exist another path between T_1 and T_2, there must exist a cycle C in F before removing e. There must be another edge in this cycle – call it f – that is not in T but is in F, since if all the cycle edges were in T, it would not be a tree anymore. The next step is to show that

   $$T' = T - e + f$$

   is a minimum spanning tree that is a subset of F.
6. Proof that T' is Spanning: First, it is shown that T' is a *spanning tree*. It is clear that deleting an edge in the tree and adding another one does not cause a cycle, but instead

another tree with the same vertexes. Since T was a spanning tree, T' must also be a *spanning tree*, as adding f does not cause any cycles when e is removed.

7. Proof that T' is an MST: Second, it is shown that T' is a *minimum* spanning tree. In the treatment below, w is the weight function. There are three cases for the comparison between the weights of e and f.
    a. The case

    $$w(e) < w(f)$$

    is impossible, since this causes the weight of T' to be less than that of T, as it contradicts the notion that T is a minimum spanning tree.

    b. Likewise, it is impossible to have

    $$w(e) > w(f)$$

    since going through the edges in decreasing order of the edge weights would result in f being seen first. Since there is a cycle C, removing f would not cause any disconnectedness in F, so the algorithm would have removed it from F earlier. This would imply that f does not exist in F, which is a contradiction, since it has been proved earlier that f exists.

    c. Therefore,

    $$w(e) = w(f)$$

    so T' is also a *minimum* spanning tree, so $\boldsymbol{P}$ holds again.

8. Final Step of the Inductive Proof: As a result, $\boldsymbol{P}$ holds when the edge removal loop is completed, i.e., all the edges have been seen, proving that at the end F becomes a *spanning tree*. Since F must have a *minimum* spanning tree as its subset, F must be a *minimum spanning tree* itself.

References

- Kruskal, J. B. (1956): On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem *Proceedings of the American Mathematical Society* **7 (1)** 48-50
- Thorup, M. (2000): Near-optimal Fully-dynamic Graph Connectivity *Proceedings on the 32nd ACM Symposium on the Theory of Computing* 343-350
- Wikipedia (2019): Reverse-delete Algorithm

A Minimum Spanning Tree Algorithm with Inverse Ackermann Type Complexity

Abstract

Chazelle (2000b) presents a deterministic algorithm for computing a minimum spanning tree of a connected graph. Its running time is $\mathcal{O}(m\,\alpha(m,n))$ where α is the classical functional inverse of the Ackermann's function and n - respectively, m - is the number of vertexes – respectively, edges. The algorithm is comparison-based; it uses pointers, not arrays, and it makes no numeric assumptions on edge costs.

Introduction

1. History of the MST Problem: The history of the minimum spanning tree (MST) problem is long and rich, going as far back as Boruvka's work in 1926 (Boruvka (1926), Graham and Hell (1985), Nesetril (1997)). In fact, MST is perhaps the oldest open problem in computer science and combinatorial optimization (Nesetril (1997)).
2. Standard Solutions in $\mathcal{O}(m \log n)$ Time: Textbook algorithms run in $\mathcal{O}(m \log n)$ time, where n and m denote, respectively, the number of vertexes and edges in the graph. Improvements to $\mathcal{O}(m \log \log n)$ were given independently by Yao (1975) and Cheriton and Tarjan (1976).
3. Algorithm with Time Complexity $\mathcal{O}(m\,\beta(m,n))$: Later, Fredman and Tarjan (1987) lowered the complexity to $\mathcal{O}(m\,\beta(m,n))$ where $\beta(m,n)$ is the number of log iterations necessary to map n to a number less than $\frac{m}{n}$. In the worst case

$$m = \mathcal{O}(n)$$

and the running time is $\mathcal{O}(m\ log^{*}m)$.

4. Algorithm with Time Complexity $\mathcal{O}(m \log \beta(m, n))$: Soon after, the complexity was further reduced to $\mathcal{O}(m \log \beta(m, n))$ by Gabow, Galil, Spencer, and Tarjan (1986).
5. Randomized Algorithm with Linear Expected Runtime: Recently, Karger, Klein, and Tarjan (1995) have discovered a randomized algorithm with linear expected complexity.
6. Integer Cost Linear Time Algorithm: If the edge costs are integers, and the model allows bucketing and bit manipulation, it is possible to solve the problem in linear time deterministically, as was shown by Fredman and Willard (1994).
7. Linear Time Comparison Based Algorithm: To achieve a similar result in a comparison-based model has long been a high priority objective in the field of algorithms. The reason for that is, first, the illustrious history of the MST problem; second, the fact that it to the heart of matroid optimization.
8. Deterministic $\mathcal{O}\left(m\ \alpha(m, n)\right)$ Time MST Algorithm: This chapter does not resolve the MST problem, but it takes a significant step towards to a solution and charts out a new line of attack. The main result is a deterministic algorithm for computing MST of a connected graph in time $\mathcal{O}\left(m\ \alpha(m, n)\right)$ where α is the functional inverse of Ackermann's function defined in Tarjan (1975). The algorithm is comparison-based it uses pointers, not arrays, and it makes no numeric assumptions on the edge costs.
9. Non greedy Approach to Matroid Optimization: In addition to providing a new complexity bound, the larger contribution of Chazelle (2000b) is to introduce a non-greedy approach to matroid optimization, which will hopefully prove useful beyond minimum spanning trees. The key idea is to compute sub-optimal independent sets in a *non-greedy* fashion, and them progressively improve upon them until an optimal basis is reached.
10. Gradual MST Build-out using Soft-Heap: Specifically, an approximate priority queue, called a *soft heap* (Chazelle (2000a)), is used to construct a good, but not

necessarily minimum, spanning tree. The quality of the tree is progressively refined until an MST is finally produced.

11. Chazelle (2000b) MST Run-time Lemma: The MST of a connected graph with n vertexes and m edges can be computed in $\mathcal{O}\big(m\ \alpha(m,n)\big)$.
12. Origin of the Inverse Ackermann Bounds: While it is doubtful that $\mathcal{O}\big(m\ \alpha(m,n)\big)$ bound it optimal, it is definitely natural. Given a spanning tree T, to verify that it is a minimum can be done in linear time (Komlos (1985), Dixon, Rauch, and Tarjan (1992), King (1997)); the problem is to check that any edge outside T is the most expensive along the cycle it forms with T. With real costs, this can be viewed as a problem of computing over the semi-group $(\mathbb{R}, max)$ along the paths of a tree. Interestingly, this problem requires $\Omega\big(m\ \alpha(m,n)\big)$ time over an arbitrary sub-group (Tarjan (1978), Chazelle and Rosenberg (1992)). The lower bound suggests that in order to improve upon the algorithm, specific properties of $(\mathbb{R}, max)$ will have to be exploited. This is done statically in Komlos (1985), Dixon, Rauch, and Tarjan (1992), King (1997). Chazelle (2000b) speculates that an answer might come from a dynamic equivalent.
13. Organization of the Chapter: The chapter is organized as follows. This section precedes a brief overview of the algorithm, a discussion of the concept of edge corruption, and a review of soft heaps. The main structural variants of the algorithm are then introduced and its components are discussed in detail. The next section proves its correctness, and the complexity is analyzed later.

The Algorithm at a Glance

1. Input – A Connected, Undirected Graph: The input is a connected, undirected graph G with no self-loops, where each edge e is assigned a cost $c(e)$. These costs are assumed to be distinct elements from a totally ordered universe – a non-restrictive assumption, s ties can be broken arbitrarily. As is well known, the MST of such a graph is unique.

2. Criterion for the Graph Contractibility: A subgraph C of G is *contractible* if its intersection with $MST(G)$ is connected. What makes this notion useful is that $MST(G)$ can be assembled directly from $MST(C)$ and $MST(G')$, where G' is the graph derived directly from G by contracting C into a single vertex.
3. Identification of Contractible Subgraph: Previous algorithms identify the contractible subgraphs on the fly as the explored portion of the MST grows. The first idea is to reverse this process, i.e., to certify contractibility of C *before* computing its MST.
4. Advantages of Pre-computing Subgraph Contractibility: The advantage should be obvious. Computing MST is bound to be easier if it is known that C is contractible, for then one need to look only at edges with *both* end-points in C. Otherwise, the edges in one end-point in C must also be visited. This makes genuine divide-and-conquer possible.
5. Soft Heaps to compute Contractibility: The challenge is to discover a contractible subgraph without computing its MST at the same time. Since current techniques are not useful, Chazelle (2000b) turns to soft heaps.
6. Vertex Disjoint Contractible Subgraph Decomposition: To compute $MST\ (G)$, G is first decomposed into vertex disjoint contractible subgraphs of suitable size. Next each subgraph is contracted into a single vertex to form a minor of G (a minor is a subgraph derived from a sequence of edge contractions and their implied vertex deletions), which are similarly decomposed into vertex disjoint contractible subgraphs, etc.
7. Forming the Contractible Subgraphs Hierarchy: This process is iterated until G becomes a single vertex. This forms a hierarchy of contractible subgraphs, which can be modeled by a perfectly balanced tree $\mathfrak{T}$; its leaves are vertexes of G; an internal node z with children $\{z_i\}$ is associated with a graph C_z whose vertexes are contractions of the graphs $\left\{C_{z_i}\right\}$.
8. Contractible Nodes at Different Levels: Each level of $\mathfrak{T}$ represents a certain minor of G, and each C_z is a contractible subgraph of the minor associated with the level of its children. In this associated, the leaf level corresponds to G, while the root corresponds to the whole graph G contracted into a single vertex.

9. Recursive Computation of the Subgraph MST: Once $\mathfrak{T}$ is available, the MST of each C_z is computed recursively. Because the C_z's are contractible, gluing together the trees $MST\ (C_z)$ produces the $MST\ (G)$.
10. Tree Height and Level Vertex Count: There is considerable freedom in choosing the height d of $\mathfrak{T}$ and the number n_z of vertexes of each C_z – which, it should be noted, is also the number of children of z.
11. Time Required for Constructing $\mathfrak{T}$: The tree $\mathfrak{T}$ is computed in $O(m + d^3 n)$ time.
12. Tree Height/Level Vertex Trade-off: If d is chosen large then n_z's can be kept small; the recursive computation within each C_z is very fast, but building $\mathfrak{T}$ is slow. Conversely, a small height speeds up the construction of $\mathfrak{T}$ but, by making the C_z's bigger, it makes recursion more expensive. This is where Ackermann's function comes into play, by providing the best possible trade-off.
13. Choice for Level Vertex Count: Let d_z denote the height of z in $\mathfrak{T}$, which is defined as the maximum of edges from z to a leaf below. A judicious choice is

$$n_z = S(t, 1)^3 = 8$$

if

$$d_z = 1$$

and

$$n_z = S\big(t - 1, S(t, d_z - 1)\big)^3$$

if

$$d_z > 1$$

where

$$t > 0$$

is minimum such that

$$n_z \leq S(t, 1)^3$$

with

$$d = c\left\lceil\frac{m}{n}\right\rceil^{\frac{1}{3}}$$

for a large enough constant c, and

$$S(1, j) = 2j$$

for any

$$j > 0$$

$$S(i, 1) = 2$$

for any

$$i > 0$$

$$S(i, j) = S(i, j - 1)S\big(i - 1, S(i, j - 1)\big)$$

for any

$$i, j > 1$$

14. <u>Expansion Node Level Vertex Count</u>: It is easily proven by induction that the *expansion* of C_z relative to G, i.e., the subgraph of G whose vertexes end up in C_z, has precisely $S(t, d_z)^3$ vertexes. This follows from the identity

$$S(t, d_z - 1)^3 n_z = S(t, d_z - 1)^3 S(t - 1, S(t, d_z - 1)^3)^3 = S(t, d_z)^3$$

15. <u>Level Vertex Count of $\mathfrak{T}$</u>: If it is assumed for simplicity that n is actually equal to $S(t, d_z)^3$, the previous identity is true for all z, including the root of $\mathfrak{T}$. It follows immediately that d coincides with the height of $\mathfrak{T}$.
16. <u>MST Runtime Inductive Proof #1</u>: This section now proves by induction on t that if the number of vertexes of G satisfies

$$n = S(t, d_z)^3$$

then $MST\ (G)$ can be computed in $bt(m + d^3 n)$ time, where b is a large enough constant. For the sake of this overview, the basis case

$$t = 1$$

is omitted.
17. <u>MST Runtime Inductive Proof #2</u>: To apply the induction hypothesis on the computation cost of $MST\ (C_z)$, note that the number of nodes in C_z satisfies

$$n_z = S\big(t - 1, S(t, d_z - 1)\big)^3$$

so it can be seen by visual inspection that, in the formula above, t must be replaced by $t - 1$ and d by $S(t, d_z - 1)$. This gives a cost of $b(t - 1)[m_z + S(t, d_z - 1)^3 n_z]$ where m_z is the number of edges in C_z.

18. MST Runtime Inductive Proof #3: Summing up over all internal nodes

$$z \in \mathfrak{T}$$

allows bounding the computational costs of all $MST\ (C_z)$'s by

$$\begin{aligned} b(t-1)&\left[m + \sum_{z} S(t, d_z - 1)^3 S\big(t-1, S(t, d_z - 1)\big)^3\right] \\ &= b(t-1)\left[m + \sum_{z} S(t, d_z)^3\right] \end{aligned}$$

which is

$$b(t-1)[m + \#nodes\ in\ expansion\ of\ C_z] = b(t-1)[m + dn]$$

19. MST Runtime Inductive Proof #4: Adding to this time claimed earlier for computing $\mathfrak{T}$ yields, for b large enough

$$b(t-1)[m + dn] + \mathcal{O}(m + d^3 n) \leq bt(m + d^3 n)$$

which proves the claim. As shown later, the choice of t and d implies that

$$t = \mathcal{O}\big(\alpha(m, n)\big)$$

so the running time of the MST algorithm is $\mathcal{O}\big(m\ \alpha(m, n)\big)$.

20. Edge Cases Impact on Runtime: This informal discussion leads to the heart of the matter: how to build $\mathfrak{T}$ in $\mathcal{O}(m + d^3 n)$ time. A number of peripheral difficulties have been swept under the rug, which in the end results in an algorithm significantly more sophisticated than the one just outlined. Indeed, quiet a few things can go wrong

along the way, non as serious as *edge corruption*, an unavoidable by product of soft heaps, which is discussed next.

Edge Corruption

1. Edge Corruption due to Soft Heap: In the course of computing $\mathfrak{T}$, certain edges of G become *corrupted*, meaning that their costs are raised. To make matters worse, the cost of a corrupted edge can be raised more than once. The reason for all this has to do with soft heap, the approximate priority queue used for identifying contractible subgraphs – more on this later.
2. Corrupted Edges that are Problematic: Some corrupted edges are trouble, while other are not. To understand this phenomenon, one of the most intriguing aspects of the analysis, the scheduling of the overall $\mathfrak{T}$ constructed must be discussed.
3. Weighted, Post Order DFS Run: It would be tempting to build $\mathfrak{T}$ bottom-up level by level, but that would be a mistake. Indeed, it is imperative to maintain a connected structure. So, instead, $\mathfrak{T}$ is computed in post-order: children first, parent last.
4. Active Path of the Visited Nodes: Let z be the current visited node in $\mathfrak{T}$, and let $z_1, \cdots, z_k = z$ be the *active path*, i.e., the path from the root z_1. The subgraphs $C_{z_1}, \cdots, C_{z_k}$ are currently being assembled, and as soon as C_{z_k} is ready, it is contracted into one vertex, which is then added to $C_{z_{k-1}}$.
5. Eliminating the Single Child Parents: A minor technical note: if z_{i+1} is the leftmost child of z_i, that is, the first child visited chronologically, then C_{z_i} does not yet have any vertex, and so it makes sense to omit z_i from the path altogether. The benefit of such short-cuts is that by avoiding one-child parents, each of $C_{z_1}, \cdots, C_{z_k}$ is sure to have at least on vertex, which itself is a vertex of G or a contraction of a connected subgraph.
6. Tree Construction Induced Edge Type Change: As long as an edge of G has exactly one vertex in $C_{z_1} \cup \cdots \cup C_{z_k}$, it is said to be of a *border* type. Of course, the type of an

edge changes over time: successively, unvisited, border, $\in C_z$ along the active path contracted.

7. Cases when Corruption is Problematic: Corruption can only strike edges when they are of the border type, since it is then that they are in soft heaps. At first, corruption might seem fatal; if all the edges become corrupted, doesn't it become an MST problem with entirely wrong costs? But in fact, corruption causes harm only in one specific situation: an edge is said to have become *bad* if it is a corrupted border edge at the time its incident C_z is contracted into one vertex. Once bad always bad, but like any corrupted edge its cost can still rise.
8. Consequences of Zerto Bad Edges: Remarkably, it can be shown that if no edges ever turn bad, the algorithm would behave as though no corruption ever occurred, regardless of how much actually took place. The goal, this, is to fight badness rather than corruption.
9. Limiting the Bad Edges Count: Chazelle (2000b) shows that the number of bad edges may be limited to within $\frac{m}{2} + d^3 n$. The number of edges corrupted, but never bad, is irrelevant.
10. Producing the MST from $\mathfrak{T}$: Once $\mathfrak{T}$ is built, all the edge costs are restored to their original values, and all the bad edges are removed. A recursion is carried out within what is left of the C_z's to produce a spanning forest F. Finally, the bad edges are thrown back in, and a recursion is carried out again to produce the minimum spanning tree. There are subtleties in these various recursions, which is explained later. This overview is now closed with a quick sketch of the soft heap.

The Soft Heap

1. Edge Selection: A simple priority queue, called a *soft heap*, is the main vehicle for selecting good candidate edges. The data structures stores items from a totally ordered universe, and supports the following operations.

6

References

- Boruvka, O. (1926): O Jistem Problemu Minimalnim *Prace Moravske Prirodovedecke Spolecnosti* **3** 37-58
- Chazelle, B. and B. Rosenberg (1991): The Complexity of Computing Partial Sums Offline *International Journal of Computing Geometry and Applications* **1 (1)** 33-45
- Chazelle, B. (2000a): The Soft-Heap: An Approximate Priority Queue with Optimal Error Rate *Journal of the ACM* **47 (6)** 1012-1027
- Chazelle, B. (2000b): A Minimum Spanning Tree Algorithm with inverse-Ackermann Type Complexity *Journal of the ACM* **47 (6)** 1028-1047
- Cheriton, D., and R. E. Tarjan (1976): Finding Minimum Spanning Trees *SIAM Journal on Computing* **5 (4)** 724-742
- Dixon, B., M. Rauch, and R. E. Tarjan (1992): Verification and Sensitivity Analysis of Minimum Spanning Trees in Linear Time *SIAM Journal on Computing* **21 (6)** 1184-1192
- Fredman, M. L., and R. E. Tarjan (1987): Fibonacci Heaps and their Use in Improved Network Optimization Algorithms *Journal of the ACM* **34 (3)** 596-615
- Fredman, M. L., and D. E. Willard (1994): Trans-dichotomous Algorithms for Minimum Spanning Trees and Shortest Paths *Journal of Computer and System Sciences* **48 (3)** 533-551
- Gabow, H. N., Z. Galil, T. Spencer, and R. E. Tarjan (1986): Efficient Algorithms for finding Minimum Spanning Trees in Undirected and Directed Graphs *Combinatorica* **6 (2)** 109-122
- Graham, R. L., and P. Hell (1985): On the History of the Minimum Spanning Tree Problem *Annals of the History of Computing* **7 (1)** 43-57
- Karger, D. R., P. N. Klein, and R. E. Tarjan (1995): A Randomized Minimum-tree Algorithm to find Minimum Spanning Trees *Journal of the ACM* **42 (2)** 321-328
- King, V. (1997): A Simpler Minimum Spanning Tree Verification Algorithm *Algorithmica* **18 (2)** 263-270

- Komlos, J. (1985): Linear Verification for Spanning Trees *Combinatorica* **5** 57-65
- Nesetril, J. (1997): A few Remarks on the History of the MST-problem *Archivum Mathematicum* **33 (1-2)** 15-22
- Tarjan, R. E. (1975): Efficiency of a Good but Not Linear Set Union Algorithm *Journal of the ACM* **22 (2)** 215-225
- Tarjan, R. E. (1978): Complexity of Monotone Networks for Computing Conjunctions *Annals of Discrete Mathematics* **2** 121-133
- Yao, A. C. (1975): An $\mathcal{O}(m \log \log n)$ Algorithm for finding Minimum Spanning Trees *Information Processing Letters* **4 (1)** 21-23

Breadth-first Search

Overview

1. Algorithm for Traversing a Graph: *Breadth-first search (BFS)* is an algorithm for traversing or searching a tree or a graph data structure. It starts at the tree root – or some arbitrary vertex of a graph, referred to as the search key – and explores all neighboring nodes at the present depth prior to moving onto the vertexes at the next depth level (Wikipedia (2020)).
2. Comparison to Depth-first Search: It uses the opposite strategy as depth-first search, which instead explores the vertexes as far as possible before being forced to backtrack and expand other nodes (Cormen, Leiserson, Rivest, and Stein (2009)).
3. Characteristics of the Algorithm:

Class	Search Algorithm
Data Structure	Graph
Worst-case Performance	$\mathcal{O}(\|V\| + \|E\|) = \mathcal{O}(b^d)$
Worst-case Space Complexity	$\mathcal{O}(\|V\|) = \mathcal{O}(b^d)$

Objective

1. Input: A *graph* and a *starting vertex root* of the graph
2. Output: The goal state. The *parent* link traces the shortest path back to the *root*.

Implementation

1. Similarity to Non-recursive DFS: This non-recursive implementation is similar to the non-recursive implementation of depth-first search, but differs from it in two ways:
    a. It uses a queue – First In First Out – instead of a stack, and
    b. It checks whether a vertex has been discovered before enqueueing the vertex rather than delaying this check until the vertex is dequeued from the queue.

    The queue contains the frontier along which the algorithm is currently searching.
2. Labeling Mechanism for Discovered Vertexes: Vertexes can be labeled as having been discovered by storing them in a set, or by an attribute in each vertex, depending on the implementation.
3. Parent Attribute of each Vertex: The *parent* attribute for each vertex is useful for accessing the vertexes in the shortest path, for example, by backtracking from the destination vertex up to the starting vertex once the BFS has been run, and the predecessor vertexes have been set.
4. Constructing the Breadth-first Tree: Breadth-first search produces a so-called *breadth-first tree*.

Time and Space Complexity Analysis

1. Time-complexity incurred by BFS: The time complexity can be expressed as $\mathcal{O}(|V| + |E|)$, since every vertex and every edge will be explored in the worst-case. $|V|$ is the number of vertexes and $|E|$ is the number of edges in the graph. Note that $\mathcal{O}(|E|)$ may vary between $\mathcal{O}(1)$ and $\mathcal{O}(|V|^2)$, depending upon how sparse the input is.
2. Space Complexity of BFS: When the number of vertexes in the graph is known ahead of time, and additional data structures have been used to determine which data structures have been already added to the queue, the space complexity can be expressed as $\mathcal{O}(|V|)$ where $|V|$ is the cardinality of the set of vertexes. This is in

addition to the space required for the graph itself, which may vary depending on the graph representation used by an implementation of the algorithm.

3. Space Time Complexity for Large Graphs: When working with graphs that are too large to store explicitly – or infinite – it is more practical to describe the complexity of the breadth-first search in different terms; to find the number of vertexes d from the starting vertex – measured in terms of the number of edge traversals – BFS takes $\mathcal{O}(b^{d+1})$ time and memory, where b is the *branching factor* of the graph, the average out-degree (Russell and Norvig (1995)).

Completeness Analysis

1. Completeness in Infinite Graphs: In the analysis of the algorithm, the input to the BFS is assumed to be a finite graph, represented explicitly using an adjacency list or a similar representation. However, in application of graph traversal methods in artificial intelligence, the input may be an implicit representation of an infinite graph. In this context, a search method is described as being complete f it is guaranteed to find a goal state if one exists.
2. Completeness in BFS vs. DFS: Breadth-first search is complete, but depth-first search is not. When applied to infinite graphs represented implicitly, breadth-first search will eventually find the goal state, but depth first search may get lost in parts of the graph that have no goal state and never return (Coppin (2004)).

BFS Ordering

1. Enumeration of BFS Ordered Vertexes: An enumeration of the vertexes of a graph is said to be a BFS ordering if it is the possible output of the application of BFS to the graph.
2. Symbology for BFS Ordering Statement: Let

$$G = (V, E)$$

be a graph with n vertexes. Let $N(v)$ be the neighbors of v. Let

$$\sigma = (v_1, \cdots, v_m)$$

be a list of distinct elements of V. For

$$v \in V \setminus \{v_1, \cdots, v_m\}$$

let $\nu_\sigma(v)$ be the least i such that v_i is a neighbor of v, if such an i exists, and ∞ otherwise.

3. Formal Statement of BFS Enumeration: Let

$$\sigma = (v_1, \cdots, v_n)$$

be an enumeration of the vertexes of V. The enumeration σ is said to be a BFS ordering – with source v_1 – if, for all

$$1 < i \leq n$$

v_i is the vertex

$$w \in V \setminus \{v_1, \cdots, v_{i-1}\}$$

such that $\nu_{(v_1, \cdots, v_{i-1})}(w)$ is minimal. Equivalently, σ is a BFS ordering if, for all

$$1 \leq i < j < k \leq n$$

with

$$v_i \in N(v_k) \setminus N(v_j)$$

there exists a neighbor v_m of v_j such that

$$m < i$$

Applications

Breadth-first search can be used to solve many problems in graph theory, for example:

a. Copying garbage collection, Cheney's algorithm
b. Finding the shortest path between nodes v and u, with path length measured by the number of edges – an advantage over DFS (Aziz and Prakash (2010))
c. Reverse Cuthill-McKee mesh numbering
d. Ford-Fulkerson method for computing maximum flow in a flow network
e. Serialization/de-serialization of a binary tree vs. serialization in sorted order, allows the tree to be re-constructed in an efficient manner
f. Construction of the *failure function* of the Aho-Corasick pattern matcher
g. Testing bipartiteness of a graph

References

- Aziz, A., and A. Prakash (2010): Algorithms for Interviews
- Coppin, B. (2004): *Artificial Intelligence Illuminated* **Jones and Bartlett Learning**
- Cormen, T., C. E. Leiserson, R. Rivest, and C. Stein (2009): *Introduction to Algorithms 3*rd *Edition* **MIT Press**

- Russell, S., and P. Norvig (2003): *Artificial Intelligence: Modern Approach 2nd Edition* **Prentice Hall**
- Wikipedia (2020): Breadth-first Search

Depth-first Search

Overview

1. Description of Depth-first Search: *Depth-first Search (DFS)* is an algorithm for searching tree or graph data structures. The algorithm starts at the root vertex – selecting some arbitrary vertex as a root vertex in the case of a graph – and explores as far as possible in each branch before backtracking (Wikipedia (2020)).
2. Tremaux Trees for Solving Mazes: A version of depth-first search was investigated by Charles Tremaux as a strategy for solving mazes (Sedgewick (2002), Even (2011)).

Properties

1. Characteristics of Depth-first Search:

Class	Search Algorithm
Data Structure	Graph
Worst-case Performance	$\mathcal{O}(\lvert V\rvert + \lvert E\rvert)$ for explicit graphs traversed without repetition, $\mathcal{O}(b^d)$ for implicit graphs with branching factor b searched to a depth d
Worst-case Space Complexity	$\mathcal{O}(\lvert V\rvert)$ if the entire graph is traversed without repetition $\mathcal{O}(Longest\ Path\ Length\ Searched) = \mathcal{O}(bd)$

	for implicit graphs without elimination of duplicate values

2. Time/Space Analysis of DFS: The time and the space analysis of DFS differs according to its application area. In theoretical computer science, DFS is typically used to traverse and entire graph, and takes time $\mathcal{O}(|V| + |E|)$ (Cormen, Leiserson, Rivest, and Stein (2009)); linear in the size of the graph. In these applications, it also uses space $\mathcal{O}(|V|)$ in the worst case to store the stack of vertexes in the current search path as well as the set of already visited vertexes. Thus, in this setting, the time and the space bound are the same as that for breadth-first search and the choice of which of these two algorithms to use depends less on their complexity and more on the different properties of the vertex orderings the two algorithms produce.
3. Case of Large/Infinite Graphs: For applications of DFS in relation to specific domains, such as searching for solutions in artificial intelligence or web crawling, the graph to be traversed is often either too large to be visited in its entirety, or infinite; DFS may suffer from non-termination. In such cases, the search is performed only to a limited depth; due to limited resources such as memory or disk space, one typically does not use data structures to keep track of the set of all previously visited vertexes.
4. Limited Depth-Search Complexity Analysis: When search is performed to a limited depth, the time is still linear in terms of the number of expanded vertexes and edges – although this number is not the same as the size of the entire graph because some vertexes may be searched more than once and the others not at all – but the space complexity for this version of DFS is only proportional to depth limit, and as a result, is much smaller than the space needed for searching to the same depth using breadth-first search. For such applications, DFS also lends itself much better to heuristic methods for choosing a likely-looking branch.
5. Iterative Deepening Depth-first Search: When an appropriate depth limit is not known a priori, iterative deepening depth-first search applies a DFS repeatedly with a sequence of increasing limits. In the artificial intelligence model of analysis, with a branching factor greater than one, iterative deepening increases the running time only

by a constant factor over the case in which the correct depth is known due to the geometric growth in the number of vertexes per level.

DFS Traversal

1. Output Search Tree from DFS: DFS may also be used to collect a sample of graph vertexes. However, incomplete DFS, similar to incomplete BFS, is biased towards vertexes of high degree.
2. Tremaux Tree and Iterative Deepening: The edges traversed from a DFS search form a Tremaux tree, a structure with important applications in graph theory. Performing the search without remembering previously remembered vertexes results in a cycle. Iterative deepening is one technique to avoid infinite loop and to reach all nodes.

Edges from a DFS Output

A convenient description of a DFS of a graph is in terms of the spanning tree of vertexes reached during the search. Based on this spanning tree, the edges from the original graph can be divided into three classes; *forward edges*, which point from the vertex of a tree to one of its descendants, *back edges*, which point from a vertex to one of the ancestors, and *cross edges*, which do neither. Sometimes *tree edges*, which belong to the spanning tree itself, are classified separately from forward edges. If the original graph is undirected, then all of its edges are tree edges or back edges.

Ordering of the DFS Output

1. Enumeration of the DFS Vertexes: An enumeration of the vertexes of a graph is said to be a DFS ordering if it is the possible output of the application of DFS to the graph.

2. <u>Symbology for DFS Ordering Statement</u>: Let

$$G = (V, E)$$

be a graph with n vertexes. Let

$$\sigma = (v_1, \cdots, v_m)$$

be a list of distinct elements of V. For

$$v \in V \setminus \{v_1, \cdots, v_m\}$$

let $\nu_\sigma(v)$ be the greatest i such that v_i is a neighbor of v, if such an i exists, and 0 otherwise.

3. <u>Formal Statement of DFS Enumeration</u>: Let

$$\sigma = (v_1, \cdots, v_n)$$

be an enumeration of the vertexes of V. The enumeration σ is said to be a DFS ordering – with source v_1 – if, for all

$$1 < i \leq n$$

v_i is the vertex

$$w \in V \setminus \{v_1, \cdots, v_{i-1}\}$$

such that $\nu_{(v_1, \cdots, v_{i-1})}(w)$ is maximal. Let $N(v)$ be the neighbors of v. Equivalently, σ is a BFS ordering if, for all

$$1 \leq i < j < k \leq n$$

with

$$v_i \in N(v_k) \setminus N(v_j)$$

there exists a neighbor v_m of v_j such that

$$i < m < j$$

Vertex Orderings

1. DFS Run Linear Vertex Ordering: It is possible to use depth-first search to linearly order the vertexes of a graph or tree. There are four possible ways of doing this.
2. Pre-Ordering: A *pre-ordering* is a list of vertexes in the order in which they were first visited by the DFS algorithm. This is a compact and a natural way of describing the progress of the search. A pre-ordering of an expression tree is the expression in Polish notation.
3. Post-Ordering: A *post-ordering* is a list of vertexes in the order that they were *last* visited by the algorithm. A post-ordering of an expression tree is the expression in reverse Polish notation.
4. Reverse Pre-ordering: A *reverse pre-ordering* is the reverse of a pre-ordering, i.e., a list of vertexes in the opposite order of their first visit. Reverse pre-ordering is not the same as post-ordering.
5. Reverse Post-ordering: A *reverse post-ordering* is the reverse of a post-ordering, i.e., a list of vertexes in the opposite order of their last visit. Reverse post-ordering is not the same as pre-ordering.
6. DFS Ordering for Binary Trees: For binary trees, there is additionally *in-ordering* and *reverse in-ordering*.

7. Topological Sorting of a DAG: Reverse post-ordering produces a topological sorting of any directed acyclic graph. The ordering is also useful in control-flow analysis as it often represents a natural linearization of the control flows.

Implementation

1. Input and Outputs of DFS:
    a. Input => A graph G and a vertex v of G
    b. Output => All vertexes reachable from v are labeled as discovered
2. Recursive and Non-recursive Implementations: The order in which the vertexes are discovered in the recursive DFS is called the lexicographic order (Goodrich and Tamassia (2001), Cormen, Leiserson, Rivest, and Stein (2009)). An example of non-recursive implementation of DFS with worst-case space complexity of $\mathcal{O}(|E|)$ is shown in Kleinberg and Tardos (2006).
3. Order of the Neighbors Processed: The recursive and the non-recursive variations of the DFS visit the neighbors of each vertex in the opposite order from each other: the first neighbor of v visited by the recursive variation is the first one in the list of adjacent edges, while in the non-recursive variation, the first visited neighbor is the last one in the list of adjacent edges.
4. Comparison between Non-recursive DFS and BFS: The non-recursive implementation is similar to BFS, but it differs from it in two ways:
    a. It uses a stack instead of a queue.
    b. It delays checking whether a vertex has been discovered until the vertex is popped from the stack rather than making this check before adding the vertex.

Applications

Applications the use DFS as a building block include:

a. Finding connected components.

b. Topological sorting.

c. Finding components connected by 2 edges or 2 vertexes.

d. Finding components connected by 3 edges or 3 vertexes.

e. Finding the bridges of a graph.

f. Generating the words in order to plot the limit set of a group.

g. Finding strongly connected components.

h. Planarity Testing (Hopcroft and Tarjan (1974), de Fraysseix, de Mendez, and Rosenstiehl (2006)).

i. Solving puzzles with only one solution, such as mazes. DFS can be adapted to find all solutions to a maze by including only vertexes in the current path in the visited set.

j. Maze generation using randomized DFS.

k. Finding bi-connectivity in graphs.

Complexity

1. Parallelizability of Recursive Lexicographic DFS: Given a graph G, let

$$O = (v_1, \cdots, v_n)$$

   be the ordering computed by the standard recursive DFS algorithm. Reif (1985) considered the complexity of computing this lexicographic depth-first search ordering from the graph, given a source. A decision version of the problem, i.e., testing whether some vertex u occurs before some vertex v in this order, is **P**-complete, meaning that it is a *nightmare for parallel processing* (Mehlhorn and Sanders (2008)).
2. DFS Parallelization using Randomization Algorithms: A depth-first search ordering – not necessarily the lexicographic one – can be computed by a randomized parallel

algorithm in the complexity class RNC (Aggarwal and Anderson (1988)). Until recently, it has remained unknown whether a DFS traversal could be constructed by a deterministic parallel algorithm, in the complexity class NC (Karger and Motwani (1997)).

References

- Aggarwal, A., and R. J. Anderson (1988): A Random NC Algorithm for Depth-first Search *Combinatorica* **8 (1)** 1-12
- Cormen, T., C. E. Leiserson, R. Rivest, and C. Stein (2009): *Introduction to Algorithms 3rd Edition* **MIT Press**
- de Fraysseix, H., O. de Mendez, and P. Rosenstiehl (2006): Tremaux Trees and Planarity *International Journal of Foundations of Computer Science* **17 (5)** 1017-1030
- Even, S. (2011): *Graph Algorithms 2nd Edition* **Cambridge University Press**
- Goodrich. M. T., and R. Tamassia (2001): *Algorithm Design: Foundations, Analysis, and Internet Examples* **Wiley**
- Hopcroft, J., and R. E. Tarjan (1974): Efficient Planarity Testing *Journal of the ACM* **21 (4)** 549-568
- Karger, D. R., and R. Motwani (1997): An NC Algorithm for Minimum Cuts *SIAM Journal on Computing* **26 (1)** 255-272
- Kleinberg, J., and E. Tardos (2006): *Algorithm Design* **Addison Wesley**
- Mehlhorn, K., and P. Sanders (2008): *Algorithms and Data Structures: The Basic Tool-box* **Springer**
- Reif, J. H. (1985): Depth-first Search is inherently Sequential *Information Processing Letters* **20 (5)** 229-234
- Sedgewick, R. (2002): *Algorithms in C++: Graph Algorithms 3rd Edition* **Pearson Education**
- Wikipedia (2020): Depth-first Search