Graph Algorithms in DROP

v5.04 5 September 2020

Priority Queue

Overview

1. Conception behind a Priority Queue: A *priority queue* is an abstract data type like a regular queue or stack, but where additionally each element has a *priority* associated with it. In a priority queue, an element with a high priority is served before an element with a low priority. In some implementations, if two elements have the same priority, they are served according to the order in which they were enqueued, while in other implementations, ordering of elements with the same priority is undefined.
2. Implementation Backing a Priority Queue: While priority queues are often implemented with heaps, they are conceptually different from heaps. A priority queue is a concept liker a list of a map; just as a list can be implemented using a linked list or an array, a priority queue can be implemented using a heap or a variety of other methods such as an unordered array.

Operations

1. Basic Operations of Priority Queue: A Priority Queue must at least support the following operations (Wikipedia (2020)):
    a. Is Empty: Check whether the queue has no elements.
    b. Insert with Priority: Add an element to the queue with an associated priority.
    c. Pull Highest Priority Element: Remove the element from the queue that has the highest priority, and return it.
        i. This is also known as *pop element off*, *get maximum element*, or *get front (most) element*.

ii. Some conventions reverse the order of the priorities, treating lower values to be higher priority, so this may be known as *get minimum element*.

iii. This may instead be specified as separate *peek at highest priority element* and *delete element* functions, which can be combined to produce *pull highest priority element*.

2. Peeking and Finding Minimum/Maximum: In addition, *peak* – in this context often called *find-max* or *find-min* – which returns the highest priority element but does not modify the queue, is very frequently implemented, and nearly always executes in $\mathcal{O}(1)$ time. This operation and its $\mathcal{O}(1)$ performance are crucial to many applications of priority queues.
3. Advanced Operations of Priority Queue: More advanced implementations may support mor complicated operations, such as *pull lowest priority element*, inspecting the first few highest and lowest priority elements, clearing the queue, clearing sub-sets of the queue, performing a batch insert, merging two or more queues into one, incrementing priority of any element, etc.
4. Stacks and Queues Modeled as Priority Queues: Stacks and queues may be modeled as particular kinds of priority queues. As a reminder, this is how stacks and queues behave:
    a. Stacks => Elements pulled in the *last-in first-out* order.
    b. Queues => Elements pulled in the *first-in first-out* order.

    In a stack, the priority of each inserted element monotonically increases; thus, the last element inserted is always the first retrieved. In a queue, the priority of each element monotonically decreases; thus, the first element inserted is always the first element retrieved.

Implementation

1. Naïve Implementation: There are a variety of simple, usually inefficient, ways to implement a priority queue. They provide an analogy to help one understand what a priority queue is. For instance, one can keep all elements in an unsorted list. Whenever the highest priority element is requested, search through all the elements for the one with the highest priority. In the big-$\mathcal{O}$ notation, this corresponds to $\mathcal{O}(1)$ insertion time and $\mathcal{O}(n)$ pull time due to search.
2. Heap Based Priority Queue Implementation: To improve performance, priority queues typically use a heap as their backbone, giving $\mathcal{O}(\log n)$ performance for inserts and removals, and $\mathcal{O}(n)$ to build initially from a set of n elements. Variants of the basic heap data structure such as pairing heaps or Fibonacci heaps can provide better bounds for some operations (Cormen, Leiserson, Rivest, and Stein (2009)).
3. Binary Search Tree Based Implementation: Alternatively, when a self-balancing binary search tree is used, insertion and removal also take $\mathcal{O}(\log n)$ time, although building trees from existing sequences of elements takes $\mathcal{O}(n \log n)$ time; this is typical where one might already have access to the data structures, such as with third-party or standard libraries.
4. Priority Queues and Sorting Algorithms: From a computational standpoint, priority queues are congruent to sorting algorithms. The section below on the equivalence between priority queues and sorting algorithms describes how efficient sorting algorithms can create efficient priority queues.

Specialized Heaps

1. Functionality Provided by Specialized Heaps: There are several specialized heap data structures that either provide additional operations of outperform heap-based implementations for specific types of keys. This section supposes that the set of possible keys is $\{1, 2, \cdots, C\}$
2. Bucket Queue for Integer Priorities: In the case of integer priorities, when only insert, find min, and extract min are needed, a bucket queue can be constructed as an array

of C linked lists plus a reference *top*, initially C. Inserting an item with key k appends the item to the k^{th} element of the linked list, as well as updated

$$top \leftarrow \min(top, k)$$

both in constant time. *Extract-min* deletes and returns one item from the list with index *top*, then increments top if needed until it points again to a non-empty list; this takes $\mathcal{O}(C)$ in the worst case. These queues are useful for sorting the vertexes of a queue by their degree (Skiena (2010)).

3. van Emde Boas Priority Queue: A van Emde Boas tree supports *minimum*, *maximum*, *insert*, *delete*, *search*, *extract-min*, *extract-max*, *predecessor*, and *successor* updates in $\mathcal{O}(\log\log C)$ time, but has a space cost for small queues of about $\mathcal{O}(2^{\frac{m}{2}})$ where m is the number of bits in the priority queue (van Emde Boas (1975)). The space can be significantly reduced with hashing.
4. Fusion Tree Based Priority Queue: The Fusion tree by Fredman and Willard (1994) implements the *minimum* operation in $\mathcal{O}(1)$ time and *insert* and *extract-min* operations in $\mathcal{O}(\frac{\log n}{\log\log C})$ time. However, as they clearly state, the algorithm has theoretical interest only, since the constant factors involved in the execution time preclude practicality.
5. Optimization for multiple Peek Operations: For applications that do many *peek* operations for every *extract-min* operation, the time complexity for peek actions can be reduced to $\mathcal{O}(1)$ in all tree and heap implementations by caching the highest priority element after every insertion and removal. For insertion, this adds at most a constant cost, since the newly inserted element is compared only with the previously cached minimum element. For deletion, this adds at most an additional *peek* cost, which is typically cheaper than the deletion cost, so overall time complexity is not significantly impacted.
6. Monotone Priority Queues: Monotone priority queues are specialized queues that are optimized for the case where no item is ever inserted that has a lower priority – in the

case of *min-heap* - than any item previously extracted. This restriction is met by several practical applications of priority queues.

Summary of Running Times

1. Catalog of Running Times: Listed below are the time complexities of various heap data structures. Function names assume a *min-heap*.

Operation	find-min	delete-min	References
Binary	$\Theta(1)$	$\Theta(\log n)$	• Cormen, T., C. E. Leiserson, R. Rivest, and C. Stein (2009)
Leftist	$\Theta(1)$	$\Theta(\log n)$	
Binomial	$\Theta(1)$	$\Theta(\log n)$	• Cormen, T., C. E. Leiserson, R. Rivest, and C. Stein (2009)
Fibonacci	$\Theta(1)$	$\mathcal{O}(\log n)$ (Amortized Time)	• Fredman, M. L., and R. E. Tarjan (1987) • Cormen, T., C. E. Leiserson, R. Rivest, and C. Stein (2009)
Pairing	$\Theta(1)$	$\mathcal{O}(\log n)$ (Amortized Time)	• Iacono (2014)
Brodal	$\Theta(1)$	$\mathcal{O}(\log n)$	• Brodal (1996a)
Rank-pairing	$\Theta(1)$	$\mathcal{O}(\log n)$ (Amortized Time)	• Haeupler, B., S. Sen, and R. E. Tarjan (2011)
Strict Fibonacci	$\Theta(1)$	$\mathcal{O}(\log n)$	• Brodal, G. S., G. Lagogiannis, and R. E. Tarjan (2012)
2-3 heap	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$ (Amortized Time)	• Takaoka, T. (1999)

Operation	insert	decrease-key	meld
Binary	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$	$\Theta(n)$
Leftist	$\Theta(\log n)$	$\mathcal{O}(\log n)$	$\Theta(\log n)$
Binomial	$\Theta(1)$ (Amortized Time)	$\Theta(\log n)$	$\mathcal{O}(\log n)$ (n is the size of the larger Heap)
Fibonacci	$\Theta(1)$	$\Theta(1)$ (Amortized Time)	$\Theta(1)$
Pairing	$\Theta(1)$	$o(\log n)$ (Amortized Time)	$\Theta(1)$
Brodal	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Rank-pairing	$\Theta(1)$	$\Theta(1)$ (Amortized Time)	$\Theta(1)$
Strict Fibonacci	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
2-3 heap	$\mathcal{O}(\log n)$ (Amortized Time)	$\Theta(1)$	

2. Pairing and Brodal Queues – Additional Notes: For pairing queues, the *decrease-key* operation has a lower bound of $\Omega(\log \log n)$ (Fredman (1999)) and an upper bound of $\mathcal{O}(2^{2\sqrt{\log \log n}})$ (Pettie (2005)). For Brodal queues, Brodal and Okasaki later describe a persistent variant with the same bounds except for *decrease-key*, which is not supported. However, heaps with n elements can be constructed bottom-up in $\mathcal{O}(n)$ (Goodrich and Tamassia (2005)).

Using a Priority Queue to Sort

1. Priority Queue vs. Sorting Algorithm Equivalence: The semantics of priority queue naturally suggest a sorting method: insert all the elements to be sorted into a priority queue, and sequentially remove them: they will come out in sorted order. This is actually the procedure used by several sorting algorithms, once the layer of abstraction provided by the priority queue is removed. This sorting method is equivalent to the following sorting algorithms laid out in the table below.

2. Heap-Based Priority Queue Implementation:

Name	Priority Queue Implementation	Best	Worst	Average
Heapsort	Heap	$n \log n$	$n \log n$	$n \log n$
Smoothsort	Leonardo Heap	n	$n \log n$	$n \log n$
Selection Sort	Unordered Array	n^2	n^2	n^2
Insertion Sort	Ordered Array	n	n^2	n^2
Tree Sort	Self-balancing Binary Search Tree	$n \log n$	$n \log n$	$n \log n$

Using a Sorting Algorithm to Make a Priority Queue

1. Thorup's Lemma on Priority Queues: Thorup (2007) presents a general deterministic linear space reduction from priority queues to sorting implying that if one can sort up to n keys in $S(n)$ time per key, then there is a priority queue supporting *delete* and *insert* in $\mathcal{O}\big(S(n)\big)$ and *find-min* in constant time.
2. Implication of the Thorup's Priority Queue Lemma: The lemma states that, if there is a sorting algorithm that can sort in $\mathcal{O}(S)$ time per key, where S is some function of n and the word size, then one can use the given procedure to create a priority queue when pulling the highest priority element in $\mathcal{O}(1)$ time, and inserting new elements – and deleting elements – in $\mathcal{O}(S)$ time. For example, if one has a $\mathcal{O}(\log \log n)$ sorting algorithm, one can create a priority queue with $\mathcal{O}(1)$ pulling and $\mathcal{O}(\log \log n)$ insertion.

Applications – Bandwidth Management

1. Priority Queue in Traffic Management: Priority queues can be used to manage limited resources such as bandwidth on a transmission line from a network router. In the event of outgoing traffic queuing due to insufficient bandwidth, all other queues can be halted to send traffic from the highest priority queue upon arrival. This ensure prioritized traffic – such as real-time traffic, e.g., an RTP steam of a VoIP connection – is forwarded with the least delay and the least likelihood of being rejected due to a queue reaching its maximum capacity. All other traffic can be handled when the highest priority queue is empty. Another approach used is to send disproportionately more traffic from higher priority queues.
2. Priority Queues in Network Standards: Many modern protocols for local area networks also include the concept of priority queues at the media access control (MAC) sub-layer to ensure that high-priority applications such as VoIP and IPTV experience lower latency than the other applications which can be served with the best-effort service. Examples include IEEE 802.11e – an amendment to IEEE 802.11 which provides quality of service – and ITU-T G.hn, a standard for high-speed local area network using existing home wiring – power lines, phone lines, and coaxial cables.
3. User/Network Level Priority Control: Usually, a limitation policer is set to limit the bandwidth that the traffic from the highest priority queue can take, in order to prevent high priority packets from choking off all other traffic. This limit is usually never reached due to high level control instances such as Cisco Call Manager, which can be used to inhibit calls which would exceed the programmed bandwidth limit.

Applications – Discrete Event Simulation

Another use of the priority queues is to manage events in a discrete event simulation. The events are added to a queue with their simulation time used as a priority. The execution of the simulation proceeds by repeatedly pulling the top of the queue and executing the event thereon.

Application – Dijkstra's Algorithm

When the graph is stored in the form of an adjacency list or a matrix, a priority queue can be used to extract the minimum efficiently when implementing the Dijkstra's algorithm, although one needs the ability to alter the priority of a particular vertex in the priority queue efficiently.

Application – Huffman Coding

Huffman coding requires one to repeatedly obtain the two lowest coding trees. A priority queue is one method of achieving this.

Application – Best-First Search Algorithm

Best-first search algorithm, such as the A* algorithm, find the shortest path between two vertexes or nodes in a weighted graph, trying out the most promising routes first. A priority queue – also known as a *fringe* – is used to keep track of the unexplored routes; the one for which the estimate – a lower bound on the case of A* - of the total path length is smallest is given the highest priority. If memory limitations make the best-first search impractical, variants like SMA* algorithms can be used instead, with a double-ended priority queue to allow removal of low priority items.

Application – ROAM Triangulation Algorithm

1. Dynamic Triangulation of a Terrain: The Real-time Optimally Adapting Meshes (ROAM) algorithm computes the dynamically changing triangulation of a terrain. It works by splitting triangles where more detail is needed and merging them where less detail is needed. The algorithm assigns each triangle in the terrain a priority, usually related to the error reduction if the triangle would be split.
2. Separate Queues for Split/Merge: The algorithm uses two priority queues – one for triangles that can be split and another for triangles that can be merged. In each step, the triangle from the split queue with the highest priority is split, or the triangle from the merge queue with the highest priority is merged with its neighbors.

Application – Prim's Algorithm for Minimum Spanning Tree

Using *min-heap* priority queue in Prim's algorithm to find the minimum spanning tree of a connected and undirected graph, one can achieve a good running time. This *min-heap* priority queue uses the *min-heap* data structure that supports operations such as *insert*, *minimum*, *extract-min*, and *decrease-key* (Cormen, Leiserson, Rivest, and Stein (2009)). In this implementation, the weight of the edges is used to determine the priority of the vertexes. Lower the weight, higher the priority, and higher the weight, lower the priority.

Parallel Priority Queue

Parallelization can be used to speed up priority queues. There are different methods to parallelize priority queues. The easiest approach is to parallelize individual operations such as *insert* and *extract-min*. Another method allows concurrent access of multiple processes to the same priority queue. The last method uses operations that work on k elements, instead of just one element. For example, *extract-min* will remove the first k elements with the highest priority.

Parallelize Individual Operations

1. Speed-up achieved using Multiple Processors: An easy way to parallelize priority queues is to parallelize individual operations. This means that the queue works like a sequential one, but individual operations are sped up several processors. The maximum speed up for this technique is limited by $\mathcal{O}(\log n)$ because there are sequential implementations for priority queues whose slowest operation runs in $\mathcal{O}(\log n)$, e.g., binomial heap.
2. Operations Executable in $\mathcal{O}(1)$ Time: The operations *insert*, *extract-min*, *decrease-key*, and *delete* can be executed in constant time in Concurrent Read, Exclusive Write (CREW) PRAM with $\mathcal{O}(\log n)$ processors, where n is the size of the priority queue (Brodal (1996b)). In the following, the queue is implemented as a binomial heap. After every operation, only three trees of the same order are allowed, and the smallest tree of order i is smaller than all roots of trees of higher order.
3. *insert*: A new binomial tree with order 0, is inserted. Then two trees of the same order i get linked to a tree of order $i + 1$ if there are three trees of order i. The tree with the smallest root in order i is not involved in this procedure. Each processor executed this step for trees in one order. So $insert$ can be run in $\mathcal{O}(1)$.
4. *extract-min*: The minimal element is the smallest element of the tree of order 0. To make sure that the new minimum is in order 0 too, for each order i

$$i > 0$$

the tree is split in two trees of order $i - 1$. As each processor executes the step for one order, this step can be executed in parallel in $\mathcal{O}(1)$. After this step, a parallel link is executed. So, two trees of order i are joined to a tree of order $i + 1$ if at least three trees of order i exist. This is executable in parallel, so *extract-min* can be run in constant time.

5. *delete* and *decrease-key*: The concept behind constant time *insert* and *extract-min* can be extended to achieve a constant time for *delete* as well. The *decrease-key* operation can then be implemented as a *delete* followed by an *insert*. Thus, the *decrease-key* operation is also a constant time operation.
6. Benefits and Drawbacks of the Approach: The advantage of this type of parallelization of the priority queue is that it can be used like a sequential one. But the maximum speed-up is only $\mathcal{O}(\log n)$. This speed-up is further limited in practice because there is additional effort for communication between different processors.

Concurrent Parallel Access

1. Benefits offered by Concurrent Access: If a priority queue offers concurrent access, multiple processors can perform operations concurrently on that priority queue. However, this raises two issues.
2. Challenges encountered due to Concurrent Access: First of all, the definition of the semantics of the individual operations is no longer obvious. For example, if two processes want to extract the element with the highest priority, should they get the same element or different ones? This restricts parallelism to the level of the program using the priority queue. In addition, because multiple processors have access to the same element, this leads to contention.
3. CRCW Implementation using a Skip-List: The concurrent access to a priority queue can be implemented on a Concurrent Read, Concurrent Write (CRCW) PRAM model. In the following, the priority queue is implemented as a skip-list (Sundell and Tsigas (2005), Linden and Jonsson (2018). In addition, an atomic synchronization primitive, CAS, is used to make the skip-list lock free.
4. Structure of the Skip-List: The nodes of a skip-list consist of a unique key, a priority, an array of references for each level, to the nodes, and a *delete* mark. The *delete* mark marks of the node is about to be deleted by a processor. This ensures that the other processors can react to their deletion appropriately.

5. Skip-List Insertion - Step #1: First, a new node with a key and a priority is created. In addition, the node is assigned a number of levels, which dictate the size of the array references.
6. Skip-List Insertion - Step #2: Then a search is performed to find the correct position to insert the new node. The search starts from the first node and from the highest level. The skip-list is traversed down to the lowest level until the correct position is found.
7. Skip-List Insertion - Step #3: During the search, for every level, the last traversed node will be saved as a parent node for the new node at that level.
8. Skip-List Insertion - Step #4: In addition, the node to which the reference, at that level, of the parent node points towards, will be saved as the successor node of the new node at that level. Afterwards, for every level of the new node, the references of the parent will be set to the new node.
9. Skip-List Insertion - Step #5: Finally, the references, for every level, of the new node will be set to the corresponding successor nodes.
10. *extract-min*: First, a skip-list is traversed until a node is reached whose *delete* mark is not set. The *delete* mark is then set to true for that node. Finally, the references to the parent nodes of the deleted node are updated.
11. Conflict due to Concurrent Access: If the concurrent access to a priority queue is allowed, conflicts may arise between two processes. For example, a conflict arises if one process is trying to insert a new node, but at the same time another process is about to delete the predecessor of the same node (Sundell and Tsigas (2005)). There is a risk that a new node is added to the skip-list, yet is no longer reachable.

K-Element Operations

1. Simultaneous Operations on k Elements: For this type of parallelization, new operations are introduced, that operate on k elements simultaneously instead of only

one. The new operation *k_extract-min* then deletes k smallest elements of the priority queue and returns those.

2. Layout of the Parallelization Infrastructure: The queue is parallelized on distributed memory. Each processor has its own local memory, and a local, sequential, priority queue. The elements of global, parallel priority queue are distributed across all processors.
3. Insertion into the Local Queues: A *k_insert* operation assigns elements uniformly randomly to the processors which insert the elements into their local queues. Note that single elements can still be inserted into the queue.
4. Local vs Global Queue Elements: Using this strategy, the global smallest elements are in the union of the local smallest elements of every processor, with high probability. Thus, each processor holds a representative part of the global priority queue.
5. Application in *k_extract-min* Operation: The above property is used when *k_extract-min* is executed, as the smallest m elements each are removed and collected in a result set. The elements in the result set are still associated with their original processor. The number of elements m removed from each local queue depends on k and the number of processors (Sanders, Mehlhorn, Dietzfelbinger, and Dementiev (2019)).
6. Iterative Parallel Selection of Elements: By parallel selection, the k smallest elements of the result set are determined. With high probability, these are the global k smallest elements. If not, m elements are again removed from each local queue, and put into the result set. Now these k elements can be returned. All other elements of the result set are inserted back into their local queues.
7. Running Time of *k_extract-min* Operation: The running time of *k_extract-min* is $\mathcal{O}(\frac{k}{p}\log n)$ where

$$k = \Omega(p \log p)$$

and n is the size of the priority queue (Sanders, Mehlhorn, Dietzfelbinger, and Dementiev (2019)).

8. Improving the Local Processor Performance: The priority queue can be further improved by not moving the remaining elements of the result set directly back into the local queues after a *k_extract-min* operation. This saves moving elements back and forth all the time between the result set and the local queues.
9. Caveat Underlying *K_element* Merge: By removing several elements at once, a considerable speedup can be achieved. But not all algorithms can use this kind of priority queue. Dijkstra's algorithm can work on several nodes at once, for example. The algorithm takes the node with the smallest distance from the priority queue and calculates new distances for all its neighbors. If k nodes are taken out, working at one node could change the distance of another of the k nodes. So, using $k\_element$ operations destroys the label setting property of the Dijkstra's algorithm.

References

- Brodal, G. S. (1996a): Worst-Case Efficient Priority Queues *Proceedings of the 7th ACM-SIAM Symposium on Discrete Algorithms* 52-58
- Brodal, G. S. (1996b): Priority Queue on Parallel Machines *Scandinavian Workshop on Algorithm Theory – SWAT '96* 416-427
- Brodal, G. S., G. Lagogiannis, and R. E. Tarjan (2012): Strict Fibonacci Heaps *Proceedings of the 44th Symposium on the Theory of Computing – STOC '12* 1177-1184
- Cormen, T., C. E. Leiserson, R. Rivest, and C. Stein (2009): *Introduction to Algorithms 3rd Edition* **MIT Press**
- 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): Surpassing the Information-theoretic Bound with Fusion Trees *Journal of Computer and System Sciences* **48 (3)** 533-551

- Fredman, M. L. (1999): On the Efficiency of Pairing Heaps and Related Data Structures *Journal of the Association of Computing Machinery* **46 (4)** 473-501
- Goodrich. M. T., and R. Tamassia (2005): *Data Structures and Algorithms in Java 4th Edition* **Wiley**
- Haeupler, B., S. Sen, and R. E. Tarjan (2011): Rank Pairing Heaps *SIAM Journal on Computing* **40 (6)** 1463-1485
- Iacono, J. (2014): Improved Upper Bounds for Pairing Heaps **arXiV**
- Linden, J., and B. Jonsson (2018): A Skip-list Based Concurrent Priority Queue with Minimal Memory Contention
- Pettie, S. (2005): Towards a final Analysis of Pairing Heaps *Proceedings of the 46th IEEE Symposium on Foundations of Computer Science FOCS '05* 174-183
- Sanders, P., K. Mehlhorn, M. Dietzfelbinger, and R. Dementiev (2019): *Sequential and Parallel Algorithms and Data Structures – A Basic Toolbox* **Springer**
- Skiena, S. (2010): *The Algorithm Design Manual 2nd Edition* **Springer**
- Sundell, H., and P. Tsigas (2005): Fast and Lock-free Concurrent Priority Queues for Multi-threaded Systems *Journal of Parallel and Distributed Computing* **65 (5)** 609-627
- Takaoka, T. (1999): Theory of 2-3 Heaps
- Thorup, M. (2007): Equivalence between Priority Queues and Sorting *Journal of the Association for Computing Machinery* **54 (6)**
- van Emde Boas, P. (1975): Preserving Order in a forest in less than Logarithmic Time *Proceedings of the 16th Annual Symposium on the Foundations of Computer Science* 75-84 **IEEE Computer Society**
- Wikipedia (2020): Priority Queue

Binary Heap

Overview

1. Binary Tree Based Data Structures: A *binary heap* is a heap data structure that takes the form of a binary tree. Binary trees are a common way of implementing priority queues (Cormen, Leiserson, Rivest. And Stein (2009), Wikipedia (2020)), and were introduced as a structure for heap-sort (Williams (1964)).
2. Binary Heap Property - Shape Constraint: A binary heap is defined as a binary tree with two additional constraints. First, the shape property states that a binary heap is a *complete binary tree,* i.e., all levels of the tree, except possibly the last, i.e., the deepest, one are fully filled, and if the last level of the tree is not complete, the nodes of that level are filled from the left to the right.
3. Binary Heap Property - Key Comparison: The heap property states that the key stored in each node is either greater than or equal ($\geq$), or less than or equal to ($\leq$), then keys in the node's children, according to some total order.
4. Min vs Max Binary Heaps: Heaps where the parent key is greater than or equal to ($\geq$) the child keys are called *max-heaps*; those where it is less than or equal to ($\leq$) are called *min-heaps*.
5. Efficient Algorithms for Insertion/Removal: Efficient logarithmic time algorithms are known for the two operations needed to implement a priority queue on a binary heap; inserting an element, and removing the smallest of the largest element from a *min-heap* or a *max-heap*, respectively.
6. In-place Heap-Sort Algorithm: Binary heaps are also commonly employed in the heapsort sorting algorithm, which is an in-place algorithm because binary heaps can be implemented as an implicit data structure, storing keys in an array, and using their relative positions within that array to represent child-parent relationships.
7. Time Complexity using Big-$\mathcal{O}$ Notation:

Operation	Average	Worst Case
Space	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Search	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Insert	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
Delete	$\mathcal{O}(\log n)$	$\mathcal{O}(\log n)$
Peek	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Heap Operations

Both insert and remove operations modify the heap to conform to the shape property first, by adding or removing the end of the heap. Then the heap property is restored by traversing up or down the heap. Both operations take $\mathcal{O}(\log n)$ time.

Insert

1. Binary Heap Insertion Algorithm Steps: To add an element to the heap, an *up-heap* operation must be performed – this is also known as *bubble-up*, *percolate-up*, *sift-up*, *trickle-up*, *swim-up*, *heapify-up*, or *cascade-up* – by following the steps below:
    a. Add an element to the bottom level of the heap at the most left.
    b. Compare the added element with its parent; if they are in the correct order, stop.
    c. If not, swap the element with its parent and return to the previous step.
2. Tim Complexity of Insertion Operation: The number of operations required depends only on the number of levels the new element must rise to satisfy the heap property, thus the insertion operation has a worst-case complexity of $\mathcal{O}(\log n)$ and an average case complexity of $\mathcal{O}(1)$.

Extract

1. Alternate Names for Extract Operation: The procedure for deleting the root from the heap – effectively extracting the maximum element in a *max-heap* or the minimum element in a *min-heap* – and restoring properties is called *down-heap* – also known as *bubble-down, percolate-down, sift-down, sink-down, trickle-down, heapify-down, cascade-down,* and *extract-min/max*.
2. Steps involved in the Extract Operation:
    a. Replace the root of the heap with the last element on the last level.
    b. Compare the new root with its children; if they are in the correct order, stop.
    c. It not, swap the element with one of its children and return to the previous step. Swap with its smaller child in a *min-heap* and its larger child in a *max-heap*.
3. Fulfillment of the Heap Property: The downward moving node is swapped with the *larger* of its children in a *max-heap* – in a *min-heap*, it would be swapped with its smaller child, until it satisfies the heap property in its new position.
4. $\mathcal{O}(\log n)$ Worst-Case Time Complexity: In the worst-case, the new root has to be swapped with its child on each level until it reaches the bottom of the heap, meaning that the delete operation has a time complexity relative to the height of the tree, or $\mathcal{O}(\log n)$.

Building a Heap

1. Williams' Method for Heap Construction: Building a heap from an array of n elements can be done by starting with an empty heap, then successively inserting each element. This approach, called Williams' method after the inventor of the binary heaps, is easily seen to run in $\mathcal{O}(n \log n)$ time; it performs n insertions at $\mathcal{O}(\log n)$ cost each. In fact, this procedure can be shown to take $\mathcal{O}(\log n)$ time in the worst case, meaning that $n \log n$ is also asymptotic lower bound on the complexity

(Cormen, Leiserson, Rivest, and Stein (2009)). In the average case, averaging over all the permutations of n inputs – the method takes linear time (Hayward and McDiarmid (1991)).

2. Floyd's Method for Heap Construction: However, Williams' method is sub-optimal. A faster method – due to Floyd (Hayward and McDiarmid (1991)) – starts by arbitrarily putting elements on a binary tree and respecting the shape property – the tree could be represented by an array – as shown below. Then starting from the lowest level and moving upwards, shift the root of each subtree downward as in the deletion algorithm until the heap property is satisfied.
3. Bottom up Heapification of the Input: More specifically, if all subtrees starting at some height h have already been heapified – here the bottom-most level corresponds to zero – the trees at height $h + 1$ can be heapified by sending their root down along the path of maximum valued children when building *max-heap* – and minimum valued children when building *min-heap*. This process takes $\mathcal{O}(h)$ swap operations per node. In this method, most of the heapification takes place at the lower levels.
4. Cost of Heapifying the Subtrees: Since the height of the heap is $\lfloor \log n \rfloor$, the number of nodes at height h is less than

$$\frac{2^{\lfloor \log n \rfloor}}{2^h} \leq \frac{n}{2^h}$$

Thus, the cost of heapifying all the subtrees is

$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{n}{2^h} \mathcal{O}(h) = \mathcal{O}\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) = \mathcal{O}\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = \mathcal{O}(n)$$

This uses the fact that $\sum_{i=0}^{\infty} \frac{i}{2^i}$ converges.

5. Worst Case Count of Comparisons: Th exact values of the above, i.e., the worst-case number of comparisons during the heap construction, is known to be equal to $2n +$

$2s_2(n) - e_2(n)$ (Suchenek (2012)), where $s_2(n)$ is the sum of all the digits of the binary representation of n, and $e_2(n)$ is the exponent of 2 in the prime factorization of n. Note that this does not mean that sorting can be done in linear time because building the heap is the first step in the heapsort algorithm.

6. Average Case Count of Comparisons: The average case is more complex to analyze, but it can be shown to asymptotically approach $1.8814n - 2\ log_2 n + \mathcal{O}(1)$ comparisons.
7. Input Array Bottom-up Heapification: The *max-heap* builder function can be constructed using the details specified above to convert an array that stores a complete binary array with n nodes to a *max-heap* by repeatedly heapifying it in a bottom up manner. This is based on the observation that the array of elements is indexed by $floor\left(\frac{n}{2}\right) + 1, floor\left(\frac{n}{2}\right) + 2, \cdots, n$ are all leaves of the tree – assuming that the indexes start from 1 – thus, each one is a one-element heap. Thus, the *max-heap* builder would run *max-heapify* on each of the remaining nodes.

Heap Implementation

1. Array Based Binary Tree Implementation: Heaps are commonly implemented with an array. Any binary tree can be stored in an array, but because a binary heap is always a complete binary tree, it can be stored compactly. No space is required for references; instead, the parent and the children of each node can be found by arithmetic on array indexes.
2. Binary Heap as an Implicit Data Structure: These properties make the heap implementation a simple example of an implicit data structure or an Ahnentafel list. Details depend on the root position, which may in turn depend on the constraints of the programming language used for then implementation, or the programmer preference. Specifically, sometimes the root is placed at 1 in order to simplify the arithmetic.

3. Indexes of Parents/Children #1: Let n be the number of elements in the heap and i be an arbitrary valid index of the array storing the heap. If the tree is at index 0, with valid indexes 0 through $n-1$, then each element a at index i has:
    a. Children at indexes $2i+1$ and $2i+2$
    b. Parent at index $floor\left(\frac{i-1}{2}\right)$
4. Indexes of Parents/Children #2: Alternatively, if the tree is at index 1, with valid indexes 1 through n, then each element a at index i has:
    a. Children at indexes $2i$ and $2i+1$
    b. Parent at index $floor\left(\frac{i}{2}\right)$
5. Use in Heapsort/Priority Queue: This implementation is used in the heapsort algorithm where it allows the space in the input array to be re-used to store the heap, i.e., the algorithm is done in-place. The implementation is also useful for use as a Priority Queue where use of a dynamic array allows for the insertion of an unbounded number of items.
6. Implementation of Up-heap and Down-heap: The up-heap/down-heap operations can be stated in terms of an array as follows: Suppose that the heap-property hold for indexes $b, b+1, \cdots, e$ The sift-down function extends the heap property to $b-1, b, b+1, \cdots, e$. Only index

$$i = b - 1$$

can violate the heap property. Let j be the index of the largest child of $a[i]$ – for a *max-heap*, or a smallest child for a *min-heap* – within the range $b, \cdots, e$. If no such index exists because

$$2i > e$$

then the heap property holds for the newly extended range and nothing needs to be done. By swapping the values $a[i]$ and $a[j]$ the heap property for position i is

established. At this point, the only problem is that the heap property may not hold for index j. The sift-down function is applied tail recursively for index j until the heap property is established for all the elements.

7. Implementation of Sift-down Operation: The sift-down function is fast. In each step it only needs two comparisons and one swap. The index value where it is working doubles on each iteration, so that at most $\log_2 e$ steps are required.
8. Memory Management for Big Heaps: For big heaps and for virtual memory, storing elements according to the above scheme is insufficient; almost every level is in a different page. B-heaps are binary heaps that keep sub-trees in a single page, reducing the number of pages accessed by up to a factor of 10 (Kamp (2010)).
9. Merging Two Equal Sized Heaps: The operation of merging two heaps takes $\Theta(n)$ for equal sized heaps. The best that can be done in terms of an array implementation is simply concatenating the two heap arrays and building the heap of the result.
10. Merging Two Different Sized Heaps: A heap on n elements can be merged with a heap on k elements using $\mathcal{O}(\log n \log k)$ key comparisons, or, in the case of a reference based implementation, in $\mathcal{O}(\log n \log k)$ time (Sack and Strothotte (1985)).
11. Splitting a Heap into Two: An algorithm for splitting a heap on n elements into to heaps of k and $n - k$ elements, respectively, based on a new view of heaps as an ordered collection of sub-heaps, was presented in Sack and Strothotte (1990). This algorithm requires $\mathcal{O}(\log n \log n)$ comparisons. This view presents a new and a conceptually simple algorithm for merging heaps.
12. Heap Implementations Optimized for Merging: When merging is a common task, a different heap implementation is recommended, such as binomial heaps, which can be merged in $\mathcal{O}(\log n)$.
13. Binary Tree Data Structure Implementation: Additionally, a binary tree can be implemented using a traditional binary tree data structure, but there is an issue of finding an adjacent element in the last level of the binary heap when adding an element. This element can be determined algorithmically or by extra data to the nodes, called *threading* the tree – instead of merely storing references to the children, the in-order succession of the nodes is stored as well.

14. *Min-max* and *Min-max-median* Heaps: It is possible to modify the heap structure to allow the extraction of both the smallest and the largest elements in $\mathcal{O}(\log n)$ time (Atkinson, Sack, Santoro, Strothotte (1986)). To do this, the rows alternate between *min-heap* and *max-heap*. The algorithms are roughly the same, but in each step, one must consider the alternating rows with alternating comparisons. The performance is roughly the same as a normal single-direction heap. This idea can be generalized to a *min-max-median* heap.

Derivation of Index Equations

1. Relation between Roots and Nodes: In an array-based heap, the children and the parent of a node can be located via a simple arithmetic on the node's index. This section derives the relevant equation for heaps with their root at index 0, with additional notes on heaps with their roots on index 1.
2. Defining the Level of a Node: To avoid confusion, the *level* of a node is defined as its distance from the root, such that the root itself occupies level 0.

Child Nodes

1. Deriving Index for Right Child: For a general node located at index i beginning from 0, the index of the right child will first be derived, i.e.,

$$right = 2i + 2$$

2. Number of Nodes in a Layer: Let node i be located in level L, and note that any level L contains exactly up to 2^l nodes. Furthermore, there are exactly $2^{l+1} - 1$ nodes contained in layers up to and including layer l. Because the root is stored at 0, the k^{th} node will be stored at index $k - 1$.

3. Last Node of a Layer: Putting the above observations together yields the following expression for the index of the last node in layer l:

$$Last(l) = (2^{l+1} - 1) - 1 = 2^{l+1} - 2$$

4. Indexes for Left/Right Children: Each of these j nodes must have exactly 2 children, so there must be $2j$ nodes separating i's right child from the end of its layer $L + 1$

$$Right = Last(L + 1) - 2j = (2^{l+2} - 2) - 2j = 2(2^{l+1} - 2 - j) + 2 = 2i + 2$$

Since the left child of any node is always 1 child before its right,

$$Left = 2i + 1$$

5. Root Node Located at 1: If the root node is located at index 1 instead of 0, the last node at each level is instead at index $2^{l+1} - 1$ Using this throughout yields

$$Right = 2i + 1$$

and

$$Left = 2i$$

for heaps with their root at 1.

Parent Node

Every node is either the left or the right child of its parent, so either of the following is true:

$$i = 2 \times Parent + 1$$

$$i = 2 \times Parent + 2$$

Therefore, the expression for the parent index is $\left\lfloor \frac{i-1}{2} \right\rfloor$

Related Structures

1. Swapping the Child Nodes: Since the ordering of the siblings in a heap is not specified by the heap property, a single node's two children can be freely interchanged unless doing so violates the shape property. Note, however, that in the common array-based heap, simply swapping the children might also necessitate moving the children's subtree nodes to retain the heap property.
2. Special Case of d-array Heap: The binary heap is a special case of the d-array heap in which

$$d = 2$$

References

- Atkinson, M. D., J. R. Sack, N. Santoro, and T. Strothotte (1986): *Min-max* Heaps and Generalized Priority Queues *Communications of the ACM* **29 (10)** 996-1000
- Cormen, T., C. E. Leiserson, R. Rivest, and C. Stein (2009): *Introduction to Algorithms 3rd Edition* **MIT Press**
- E. E. Doberkat (1984): An Average Case Analysis of Floyds Algorithm to construct Heaps *Information and Control* **6 (2)** 114-131

- Hayward, R. and C. McDiarmid (1991): Average Case Analysis of Heap-building by Repeated Insertion *Journal of Algorithms* **12 (1)** 126-153
- Kamp. P. H. (2010): You're doing it Wrong
- Pasanen, P. (1996): *Elementary Average Case Analysis of Floyd's Algorithm to construct Heaps* **Turku Center for Computer Science**
- Sack, J. R., and T. Strothotte (1985): An Algorithm for Merging Meaps *Acta Informatica* **22** 171-186
- Sack, J. R., and T. Strothotte (1990): A Characterization of Heaps and its Applications *Information and Computation* **86 (1)** 69-86
- Suchenek, M. A. (2012): Elementary yet Precise Worst-case Analysis of Floyd's Heap Construction Program *Fundamenta Informaticae* **120 (1)** 75-92
- Wikipedia (2020): Binary Heap
- Williams, J. W. J. (1964): Algorithm 232 – Heapsort *Communications of the ACM* **7 (6)** 347-348

Binomial Heap

Overview

A *binomial heap* is a data structure that acts as a priority queue but allows pairs of heaps to be merged together. It is important as an implementation of the mergeable heap abstract data type – also called meldable heap – which is a priority queue supporting merge operation. It is implemented as a heap similar to binary heap, but using a special structure that is different from the complete binary trees used by binary heaps (Vuillemin (1978), Cormen, Leiserson, Rivest, and Stein (2009), Wikipedia (2019)).

Definition

1. Trees in a Binomial Heap: A binomial heap implemented as a set of *binomial trees* – as opposed to a binary heap, which has the shape of a single binary tree – whose trees are recursively defined as follows (Cormen, Leiserson, Rivest, and Stein (2009)):
    a. A binomial tree of order zero is a single node.
    b. A binomial tree of order k has a root node whose children as roots of binomial trees of order $k-1, k-2, \cdots, 2, 1, 0$ – in that order.
2. Rationale behind the Heap Layout: A binomial tree of order k has 2^k nodes, and height k. The name comes from the shape: a binomial tree of order k has $\binom{k}{d}$ nodes at depth d, a binomial coefficient. Because of its structure, a binomial tree of order k can be constructed from two trees of order $k-1$ by attaching one of them as the left-most child of the other root of the tree. This feature is central to the *merge* operation of a binomial heap, which is its major advantage over conventional heaps (Brown (1978), Cormen, Leiserson, Rivest, and Stein (2009)).

Structure of a Binomial Heap

1. Properties of the Binomial Heap: A binomial heap is implemented as a set of binomial trees that satisfy the *binomial heap properties* (Cormen, Leiserson, Rivest, and Stein (2009)):
    a. Each binomial tree in a heap obeys the *minimum-heap property*; the key of a node is greater than equal to the key of its parent.
    b. There can be only either *one* or *zero* binomial trees of each order, including the zero order.
2. Rationale behind the Binomial Heap Property: The minimum heap property ensures that the root of each binomial tree contains the smallest key in the tree. It follows that the smallest key in the entire heap is one of the roots (Cormen, Leiserson, Rivest, and Stein (2009)).
3. Representing the Trees in the Heap: The second property implies that a binomial heap with n nodes consists of at least $1 + \log_2 n$ binomial trees. The numbers and the orders of these trees are completely determined by the number of nodes n; there is one binomial tree for each non-zero bit in the binary representation for the number n. For example, the decimal number 13 is 1101 in binary, $2^3 + 2^2 + 2^0$, thus a binomial heap with 13 nodes will consist of three binomial trees of order 3, 2, and 0 (Brown (1978), Cormen, Leiserson, Rivest, and Stein (2009)).
4. n Items with Distinct Keys: The number of distinct ways that n items with distinct keys can be arranged into a binomial heap equals the largest odd divisor of $n!$. For

$$n = 1, 2, 3, \cdots$$

these numbers are $1, 1, 3, 3, 15, 45, 315, 315, 2835, 14175, \cdots$ (sequence $A049606$ in the $OEIS$). If the n items are inserted into a binomial heap in a uniformly random order, each one of these arrangements is equally likely (Brown (1978)).

Implementation

1. Tree Roots in a Linked List: Because no operation requires random access to the roots of the binomial trees, the roots of the binomial tree can be stored in a linked list, ordered by the increasing order of the tree.
2. Linked List of Sibling Nodes: Because the number of children for each node is variable, it does not work well for each node to have separate links to each of its children; instead, it is possible to implement this tree using links from each node to its highest ordered child in the tree, and to its sibling in the next smaller order after that. These sibling pointers can be interpreted as next pointers in a linked list of children of each node, but with opposite order from the linked list of roots; from the largest to the smallest order, rather than vice versa.
3. Ease of Linking Equi-Ordered Trees: This representation allows two trees of the same order to be linked together, making a tree of the next larger order, in constant time (Brown (1978), Cormen, Leiserson, Rivest, and Stein (2009)).

Merge

1. Merging Trees of Same Order: The operation of *merging* two heaps is used as a sub-routine in most other operations. The basic subroutine within this procedure merges pairs of binomial trees of the same order. This may be done by comparing keys at the roots of the two trees, which are the smallest keys in both trees. The root node with the larger key is made into a child of the root node with the smaller key, increasing its order by one (Brown (1978), Cormen, Leiserson, Rivest, and Stein (2009)).
2. Simultaneously increasing Root List Traversal: To merge two heaps more generally, the lists of roots of both heaps are traversed simultaneously in a manner similar to

that of the merge algorithm, in a sequence from the smaller orders of trees to the larger orders.

3. Merging Trees with Unique/Non-Unique Orders: When only one of the two heaps being merged contains a tree of order j, this tree is moved to the output heap. When both of the two trees contain a tree of order j, the two trees are merged into one tee of order $j + 1$ so that the minimum heap property is satisfied. It may later become necessary to merge this tree with some other tree of order $j + 1$ in one of the two input heaps.
4. At most Three Trees Examined: In the course of the algorithm, it will examine at most three trees of any order, two from the heaps merged, and one composed of two smaller trees (Brown (1978), Cormen, Leiserson, Rivest, and Stein (2009)).
5. Analogy with Binary Carry Propagation: Because each binomial tree in a binomial heap corresponds to a bit in the binary representation of its size, there is an analogy between merging of two heaps and the binary addition of the *sizes* of the two heaps, from right to left. Whenever a carry occurs during an addition, this corresponds to a merging of the two binomial trees during the merge (Brown (1978), Cormen, Leiserson, Rivest, and Stein (2009)).
6. $\mathcal{O}(\log n)$ Running Time of Merge: Each tree has an order of at most $\log_2 n$, and therefore, the running time is $\mathcal{O}(\log n)$ (Brown (1978), Cormen, Leiserson, Rivest, and Stein (2009)).

Insert

1. Insertion using *Create* and *Meld*: *Inserting* a new element into a heap can be done by simply crating a new heap containing only this heap element and then merging it with the original heap. Because of the merge, a single insertion takes $\mathcal{O}(\log n)$ time.
2. Speed-up Using Amortized Merge: However, this can be sped up using a merge procedure that shortcuts the merge after it reaches a point where only one of the merged heaps has trees of larger order. With this speed up, across a series of k

consecutive insertions, the total time for insertions is $\mathcal{O}(k + \log n)$. Another way of stating this is that, after logarithmic overhead for the first insertion in the sequence, each successive *insert* has an *amortized* time of $\mathcal{O}(1)$, i.e. constant, per insertion (Brown (1978), Cormen, Leiserson, Rivest, and Stein (2009)).

3. $\mathcal{O}(1)$ Worst Case Insertion Time: A variant of the binomial heap, the skew binomial heap, achieves constant worst-case insertion time by using forests whose tree sizes are based on the skew binary system rather than on the binary system (Brodal and Okasaki (1996)).

Find Minimum

1. Minimum Element Among the Roots: To find the *minimum* element of the heap, one finds the minimum among the roots of the binomial trees. This can be done in $\mathcal{O}(\log n)$ time, as there are just $\mathcal{O}(\log n)$ tree roots to examine (Cormen, Leiserson, Rivest, and Stein (2009)).
2. Updating Root in $\mathcal{O}(1)$ Time: By using a reference to the binary tree that contains the minimum element, the time for this operation can be reduced to $\mathcal{O}(1)$ time. The reference may be updated when performing any operation other than finding the minimum. This can be done in $\mathcal{O}(\log n)$ per update, without raising the overall asymptotic running time of any operation.

Delete Minimum

1. Merging Subtrees of Minimum Element: To *delete the minimum element from the heap*, first locate the element, then remove it from the root of its binomial tree, and obtain a list of subtrees, which are each themselves binomial trees. Transform the list of subtrees into a separate binomial heap by re-ordering them from the smallest to the largest order. Then merge this range with the original heap.

2. $\mathcal{O}(n)$ Time for Delete Minimum: Since each root has at most $\log n$ children, creating this heap takes $\mathcal{O}(\log n)$. Merging heap takes time $\mathcal{O}(\log n)$, so the entire delete minimum operation takes $\mathcal{O}(\log n)$ time (Cormen, Leiserson, Rivest, and Stein (2009)).

Decrease Key

1. Maintaining the Minimum Heap Property: After *decreasing* the key of an element, it may become smaller than the key of its parent, violating the minimum heap property. In such a case, the element is exchanged with its parent, and possibly also with its grand-parent, and so on, until the minimum heap property is no longer violated.
2. Specific Representations Demanded by Decrease-Key: Each binomial tree has at most $\log_2 n$ height, so this takes $\mathcal{O}(\log n)$ time (Cormen, Leiserson, Rivest, and Stein (2009)). However, this operation requires that the representation of the tree include references from each node to its parent tree, somewhat complicating the implementation of other operations (Brown (1978)).

Delete

To *delete* any element from the heap, decrease its key to negative infinity – or equivalently to some vale lower than any element in the heap – and then delete the minimum value from the heap (Cormen, Leiserson, Rivest, and Stein (2009)).

References

- Brodal, G. S., and C. Okasaki (1996): Optimal Purely Functional Priority Queues *Journal of Functional Programming* **6 (6)** 839-857

- Brown, M. R. (1978): Implementation and Analysis Binomial Queue Algorithms *SIAM Journal on Computing* **7 (3)** 298-319
- Cormen, T., C. E. Leiserson, R. Rivest, and C. Stein (2009): *Introduction to Algorithms 3rd Edition* **MIT Press**
- Vuillemin, J. (1978): A Data Structure for manipulating Priority Queues *Communications of the ACM* **21 (4)** 309-315
- Wikipedia (2019): Binomial Heap

Soft Heap

Overview

1. Operations Supported by Soft Heap: A *soft heap* is a variant on the simple data structure that has constant amortized time for 5 types of operations. This is achieved by carefully corrupting, i.e., increasing, the keys of at most a certain of values in the heap. The constant time operations are:
    a. $create(S)$ => Create a new soft heap
    b. $insert(S, x)$ => Insert an element into a soft heap
    c. $meld(S, S')$ => Combine contents of two soft heaps into one, destroying both
    d. $delete(S, x)$ => Delete an element from a soft heap
    e. $findmin(S)$ => Get the element with the minimum key in the soft heap
2. Constant Time Delete Operation Bound: Other heaps such as Fibonacci heaps achieve most of the bounds without any corruption, but cannot provide a constant-time bound on the critical $delete$ operation. The amount of corruption can be controlled by the choice of a parameter ε, but the lower this is set, the more time insertions require - $\mathcal{O}\left(\log\frac{1}{\varepsilon}\right)$ - for an error rate ε.
3. Specific Guarantee offered by Soft Heap: More precisely, the guarantee offered by soft heap is the following: for a fixed ε between 0 and $\frac{1}{2}$, at any point in time there will be at most εn corrupted keys in the soft heap, where n is the number of elements so far. Note that this does not guarantee that only a fixed percentage of keys *currently* in the heap are corrupted; in an unlucky sequence of insertions and deletions, it can happen that all the elements in the heap will have corrupted keys. Similarly, there are no guarantees that in a sequence of elements extracted from the heap with $findmin$ and $delete$, only a fixed percentage will have corrupted keys; in an unlucky scenario only corrupted elements are extracted from the heap.

4. Core Idea behind the Construct: The soft heap was designed by Chazelle (2000). The term *corruption* in the structure is the result of what Chazelle calls *car-pooling* in a soft heap. Each node in the structure contains a linked list of keys and one common key. The common key is a upper bound on the values of the keys in the linked list. Once a key is added to the linked list, it is considered corrupted and its value is never again used in any of the operations; only the common keys are compared. This is what makes the soft heaps *soft*; one cannot be sure whether any particular value put into it will be corrupted. The purpose of these corruptions is to lower the information entropy of data, enabling the data structure to break-through the information barrier regarding heaps.

Applications – Selection Algorithm

1. Design of Deterministic Algorithms: Despite their limitations and unpredictable nature, soft heaps are useful in the design of deterministic algorithms. They have been used to achieve the best complexity to date for finding a minimum spanning tree. They can also be used to easily build an optimal selection algorithm, as well as *near-sorting* algorithms, which are algorithms that place every element near its final position, a situation in which the insertion sort is fast.
2. Use in the Selection Algorithm: One of the simplest examples is the selection algorithm. Say that one wants to find the k^{th} largest set from a group of n numbers. First, one chooses an error rate of $\frac{1}{3}$; that is, at most 33% of the keys inserted will be corrupted. Now, all n elements are inserted into the heap – the original values are called the *correct* keys, and the values stored in the heap are called the *stored* keys. At this point, at most $\frac{n}{3}$ keys are corrupted, that is, for at most $\frac{n}{3}$ keys are the *stored* key larger than the *correct* key, for all others the stored key equals the correct key.
3. Repeated Deletion of the Minimum Element: Next, the minimum element is deleted from the heap $\frac{n}{3}$ times, and this is done according to the *stored* key. As the total

number of insertions made is still n, there are still at most $\frac{n}{3}$ corrupted keys remaining in the heap. Accordingly, at least

$$2\frac{n}{3} - \frac{n}{3} = \frac{n}{3}$$

keys remaining in the heap are not corrupted.

4. Runtime for the Selection Algorithm: Let L be the element with the largest correct key among the elements removed. The stored key of L is possibly larger than its correct key – if L was corrupted – and even this larger value is smaller than all the keys of the remaining elements in the heap, as the minimum keys are being removed. Therefore, the correct key of L is smaller than the remaining $\frac{n}{3}$ uncorrupted elements in the soft heap. Thus, L divides the elements somewhere between $33\%/66\%$ and 66%/33%. The set is then partitioned about L using the partition algorithm from quicksort, and the same algorithm is applied again to either set of numbers less than L or the set of numbers greater than L, neither of which can exceed $\frac{2n}{3}$ elements. Since insertion and deletion requires $\mathcal{O}(1)$ amortized time, the total deterministic time is

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

Using the master theorem for divide-and-conquer, this becomes

$$T(n) = \Theta(n)$$

References

- Chazelle, B. (2000): The Soft-Heap: An Approximate Priority Queue with Optimal Error Rate *Journal of the ACM* **47 (6)** 1012-1027

- Wikipedia (2017): Soft Heap

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 reference 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 $\mathcal{O}(m\,\alpha(m,n))$ where α is the classical functional inverse of Ackermann's function.

3. Usage in Dynamic Rank Maintenance: 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. Computing Selection in Linear Time: 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. Usage in Weak Near Sorting: 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. Usage in Strong Near Sorting: 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. Strong Near Approximation Sorting Lemma: 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. Insertion followed by Minimum Finding: 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. Dividing Findmins across Time Slices: 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 $[x_j, x_{j+1})$ 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 $[x_j, x_{j+1})$ 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. Origin of the $\log \frac{1}{\varepsilon}$ Bound: Next, for each item

$$s \in S_i \setminus U_i$$

the search for which interval $[x_j, x_{j+1})$ 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. Soft Queue as a Modified Binomial Tree: 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. Master Tree and Rank Invariance: 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. References $child$ and $next$ give access to the children. If there are none, the references 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 reference 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 references; 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 references 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 $suffix\_min$ references.
8. Updating the Queue $ckey$ References: Prior to calling $fixMinList\ (h)$ it is assumed that all $suffix\_min$ references are correct, except for those between $header$ and h. A simple walk from h back to $header$ updates all the $suffix\_min$ references.

DeleteMin

1. $suffix\_min$ Pointing to Empty Item-List: The $suffix\_min$ reference 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 references 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 $suffix\_min$ reference 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 $suffix\_min$ references 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 $suffix\_min$ references 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 $\mathcal{O}(m)$. 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 $suffix\_min$ references, the running time is $\mathcal{O}(rC)$, 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 $suffix\_min$ References: The cost of updating $suffix\_min$ references 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

A Simpler Implementation and Analysis of Chazelle's Soft Heaps

Introduction

1. An Approximate Meldable Priority Queue: Chazelle (2000a, 2000b) devised an approximate meldable priority queue data structure, called *soft heaps*, and used it to obtain the fastest known deterministic comparison-based algorithm for computing minimum spanning trees (Chazelle (2000a, 2000c)) as well as some new algorithms for selection and approximate sorting problems.
2. Origin of the Corrupted Elements: If n elements are inserted in a collection of soft heaps, then up to εn of the elements still contained in these heaps, for a given *error parameter* ε, may be *corrupted*, i.e., have their keys artificially raised. Note that n here is the number of elements inserted into the heaps, not the current number of elements in the heaps, which may be considerably smaller.
3. Soft Heap Operations Amortized Time: In exchange for allowing these corruptions, each soft heap operation is performed in $\mathcal{O}\left(\log\frac{1}{\varepsilon}\right)$ amortized time.
4. Optimal Deterministic Comparison Based Algorithm: Soft heaps are also used by Pettie and Ramachandran (2002, 2008) to obtain an optimal deterministic comparison-based algorithm for finding minimum spanning trees, with a yet unknown running time, and for obtaining a randomized linear time algorithm for a problem that uses only a small number of random bits.
5. Heaps used in Chazelle's Algorithm: Chazelle's soft heaps are derived from the *binomial heaps* data structure in which each priority queue is composed ot a collection of *binomial trees*.

6. Heaps used by Kaplan and Zwick: Kaplan and Zwick (2009) describe a simpler and more direct implementation of soft heaps in which each priority queue is composed of a collection of standard *binary trees*.
7. No Clean-up Operations Required: Their implementation also has the advantage that no *clean-up* operations similar to the ones used in Chazelle's implementation are required.
8. Unified Potential-Based Amortized Analysis: They also present a concise and unified potential-based amortized analysis of their new implementation.

Soft Heaps

1. Operations Supported by Soft Heaps: Soft heaps are approximate meldable priority queue data structures that support the following operations.
2. $make - heap\ (e)$: Generate and return a new soft heap containing a single element e whose original key is $key[e]$.
3. $insert(P, e)$: Insert an element e, with original $key[e]$, into the soft heap P.
4. $delete(e)$: Delete element e from the soft heap currently containing it.
5. $meld(P, Q)$: Meld two soft heaps P and Q, destroying them in the process, and return the melded heap.
6. $extract - min(P)$: Return an element with the smallest *current key* in the soft heap P and delete it from P.
7. Elements with Minimal Current Key: It is important to note that $extract - min(P)$ operation returns an element with the smallest *current* key contained in P. The current key of an element e may be *larger* than its *original* key $key[e]$, which is never changed by the implementation. Elements whose current keys are larger than their original keys are said to be *corrupted*. The user has no control as to which elements become corrupted.
8. Corrupted Elements Upper Bound Count: If soft heaps were allowed to corrupt all elements their implementation would be trivial, but they would be useless. A

surprisingly useful data structure is obtained if one requires that at most εn of the elements still contained in the soft heaps are corrupted, where n is the total number of elements inserted into the soft heaps, and

$$0 < \varepsilon < 1$$

is a pre-specified *error parameter*.

9. Operations that Insert or Delete: Note that an element is inserted into a soft-heap by either the $make - heap$ or the $insert$ operation. Also note that the corrupted elements that were removed from the soft heap by the $extract - min$ operations are not counted.
10. Description of the Soft Heap Algorithm: Following Chazelle (2000b), this chapter describes an implementation of soft heaps with error parameter ε in which the amortized cost of $make - heap$ and $insert$ operations is $\mathcal{O}\left(\log\frac{1}{\varepsilon}\right)$ and the amortized cost of all other operations is 0.

Implementation

This section describes the implementation of all soft heap operations, except for the $delete$ operation which is added later. It is to be noted that many of the operations listed in Chazelle (2000b) do not use $delete$ operation.

The Data Structure

1. Collection of Binary Trees: Each soft heap priority queue is composed of a collection of binary trees. A node x of a binary tree may have a left child $left[x]$ and a right child $right[x]$.

2. Rank and Children of Binary Trees: Every node x in a binary tree has an integer *rank* denoted by $rank[x]$, associated with it. The rank of a node never changes. If x is a node of rank k, then the ranks of $left[x]$ and $right[x]$, of they exist, are $k-1$. The rank of a tree is defined as the rank of its root.
3. Binary Tree Node Target Size: Each node x has a *target size* $size[x]$ associated with it. Let

$$r = \log_2 \frac{1}{\varepsilon} + 5$$

where ε is the desired error rate. The target size of a node of rank k is s_k, where

$$s_k = \begin{cases} 1 & k \leq r \\ \left\lceil \frac{3s_{k-1}}{2} \right\rceil & otherwise \end{cases}$$

The choice of $\frac{3}{2}$ in the definition of s_k is arbitrary. Any constant strictly between 1 and 2 would do. Thus

$$s_0 = s_1 = \cdots = s_r = 1$$

which implies that

$$s_{r+1} = 2$$

$$s_{r+2} = 3$$

$$s_{r+3} = 4$$

$$s_{r+4} = 8$$

$$s_{r+5} = 12$$

etc. It is easy to prove that

$$\left(\frac{3}{2}\right)^{k-r} \leq s_k \leq 2\left(\frac{3}{2}\right)^{k-r} - 1$$

for

$$k \geq r$$

4. Actual Size of Each Node: Each node x has a list of elements $list[x]$. The number of elements in $list[x]$ is *roughly* $size[x]$ – this will be more precise below.
5. Keys of the Binary Node: A node x also has a key $ckey[x]$ which is an upper bound on the keys contained in $list[x]$, and

$$key[e] < ckey[x]$$

then e is corrupted. The data structure behaves as if the key of e is artificially raised to $ckey[x]$. Using the terminology of the introduction, if e is an element of $list[x]$, then $ckey[x]$ is the current key of e.

6. Heap Ordering of the Trees: Each tree is *heap-ordered* with respect to the $ckey$ values, i.e., if x is a node and $left[x]$ exists, then

$$ckey[x] \leq ckey\big[left[x]\big]$$

Similarly, if $right[x]$ exists, then

$$ckey[x] \leq ckey\big[right[x]\big]$$

7. Priority Queue as a Sequence of Trees: A priority queue P is composed as a sequence of trees, at most one of each rank. The rank $rank[P]$ is defined to be largest rank of a tree in P. The trees composing P are arranged in a linked list in which the trees appear in an increasing order of rank. $first[P]$ points to the P with the smallest rank belonging to P.
8. Structure of the Binary Tree: If T is a tree contained in a priority queue P, then $root[T]$ is the root node of T, $next[T]$ is the tree following T in the linked list of P, and $prev[T]$ is the tree preceding T in the list. Both $next[T]$ and $prev[T]$ may be NULL.
9. The $sufmin[T]$ Reference Member: Finally, if T is a tree, then $sufmin[T]$ points to the tree whose root has the smallest $ckey$ among all the trees that follow T in the linked list of P. In this case, trees that appear earlier are preferred.

The Sift Operation

1. Purpose of the Sift Operation: As in Chazelle's implementation, the keystone of soft heaps' operation is the $sift$ operation. As indicated earlier, the number of elements in $list[x]$ should be about $size[x]$. If the number of elements in $list[x]$ drops below $\frac{size[x]}{2}$, and x is not a leaf, then $sift(x)$ operation is used to add more elements to $list[x]$.
2. Ensuring Existence of Left Child: A $sift(x)$ works as follows. By exchanging $left[x]$ and $right[x]$, if necessary, it is ensured that $left[x]$ exists, and that

$$ckey[left[x]] \leq ckey[right[x]]$$

if $right[x]$ exists. Recall that x is not a leaf.

3. Car-pooling of Child List: Then, using what Chazelle refers to as a *data structures version of car-pooling*, all elements of $list[left[x]]$ are moved to $list[x]$. This can be done in constant time by concatenating the lists $list[x]$ and $list[left[x]]$.
4. Left Child is now Empty: $ckey[left[x]]$ is the new $ckey$; this leaves the list $list[left[x]]$ empty.
5. Recursive Invocation of $sift$ Operation: If $left[x]$ is a leaf, it is simply removed from the tree by setting it to NULL. Otherwise $sift(left[x])$ is recursively called to replenish $list[left[x]]$.
6. Criteria for the $sift$ Run: Finally, if the number of elements in $list[x]$ is still below $size[x]$ and x is still not a leaf, the operations are performed again.
7. $sift$ maintains the Heap Order: It is easy to check that $sift$ operations maintain the heap order. It is shown below that if x is not a leaf after a $sift(x)$ operation, then

$$size[x] \leq |list[x]| \leq 3size[x]$$

The Combine Operation

1. Combining Nodes of Same Rank: The second most important operation in the implementation of the soft heaps is the $combine$ operation. A $combine(x, y)$ takes two root nodes x and y of the same rank, say k, and combines them into a single tree of rank $k + 1$. This is done by generating a new node z and setting $left[z]$ to x, $right[z]$ to y, and $list[z]$ to empty. A $sift(x)$ is then performed to move enough elements into $list[z]$. The $combine$ is, of course, instrumental in the implementation of the $meld$ and the $insert$ operations, as explained below.
2. Structural Changes to the Tree: Note that no re-balancing of the binary tree occurs. The only structural changes performed on the trees are:
    a. Discarding a leaf, done in $sift$

b. Combining two trees of rank k into a larger tree of rank $k+1$ by allocating a new root – this is done in $combine$.

The $Update - Suffix - Min$ Operation

1. Purpose of $Update - Suffix - Min$: An $update - suffix - min(T)$ operation updates the $sufmin$ references of T and all the trees that precede T in the linked list of trees. Such an operation is performed when $ckey[x]$, where

$$x = root[T]$$

   is changed, e.g., by a $sift(x)$ operation, when T is a new tree added to the list of trees, or when the tree following T in the list is deleted.
2. Operational Sequence of the Update: An $update - suffix - min(T)$ operation traverses the list of trees backward from T. If T' is a tree such that $sufmin[next[T']]$ was already set to its correct value, the $sufmin[T']$ is set to T', if

$$ckey[root[T']] \leq ckey\left[root[next[T']]\right]$$

   or to $sufmin[next[T']]$

The $make - heap$ Operation

A $make - heap(e)$ operation receives an element e and returns a priority queue P composed of a single tree T containing a single node x of rank 0.

The $meld$ Operation

1. Purpose of the $meld$ Operation: A $meld(P, Q)$ receives two priority queues P and Q and returns a new priority queue obtained by merging P and Q.
2. Melding Linked Lists of Queues: Melding P and Q is done in a fairly straightforward way. The linked lists of trees of P and Q are combined, keeping a non-decreasing order of rank.
3. Two Trees of the Same Rank: Next, if two consecutive trees T_1 and T_2 in the list above have the same rank, they are combined using $combine(root[T_1], root[T_2])$ operation and the combined tree replaces them in the list.
4. Three Trees of the Same Rank: If three consecutive trees T_1, T_2, and T_3 in the list above have the same rank, then T_1 is left alone, while T_2 and T_3 are replaced by $combine(root[T_2], root[T_3])$.
5. Updating Suffix Min of the Final Tree: Finally, if T is the last tree in the list affected by these operations, and $update - suffix - min(T)$ is performed to update the $sufmin$ references.
6. Time Required by Meld: If

$$k = rank[P] \leq rank[Q]$$

then it is easy to perform $meld(P, Q)$ operation in $\mathcal{O}(k + 1)$ time.

The $insert$ Operation

To add an element e to a priority queue P, one uses $make - heap(e)$ to generate priority queue containing a single element e and then melds this priority queue with P.

The $extract - min$ Operation

1. Purpose of the $extract - min$ Operation: An $extract - min(P)$ operation returns an element with a minimum *current* key contained in P. If e is contained in $list[x]$, then the current key of e is $ckey[x]$.
2. Root with the Smallest Key: The implementation of $extract - min$ is extremely simple. Let

$$T = sufmin[first[P]]$$

and

$$x = root[T]$$

Thus x is the root of a tree of P with the smallest $ckey$.

3. Invoking $sift$ and Updating as Needed: An arbitrary element is returned from $list[x]$. Note that $list[x]$ is never empty. If the number of elements in $list[x]$ drops below $\frac{size[x]}{2}$, $sift(x)$ is called to replenish x, and then $update - suffix - min(T)$ updates the $sufmin$ references.

Correctness

1. Minimal Current Key Extraction Lemma: An $extract - min(P)$ always returns an element of P with a minimal current key.
2. Mandatory Maintenance of Heap Order: All elements inserted into P are contained in the list of nodes that are part of the trees forming P. All operations performed on soft heaps maintain heap order. Thus, elements with the smallest current key in a tree always reside at the root of the tree.

3. Guaranteed Pointing to the Smallest Key: An $extract - min(P)$ operation uses the $sufmin$ reference to the first tree in P to access a tree whose root has a minimal $ckey$, and returns an element e contained in $list[x]$, which is guaranteed to be non-empty. This element has a minimal key, as is required.
4. Bounding the Count of Corrupted Elements: The next two lemmas will be used to bound the number of corrupted elements.
5. Bounding the Size of Nodes Lemma: If x is a node of rank at most r, then

$$|list[x]| = 1$$

If x is a non-leaf of rank

$$k \geq r$$

then

$$\frac{size[x]}{2} \leq |list[x]| < 3size[x]$$

6. Size of a Rank r Node: First, if x is a node of rank at most r, then

$$|list[x]| = 1$$

If $list[x]$ becomes empty, then $sift(x)$ brings exactly one element into $list[x]$.

7. Nodes with Rank $\geq r$: Suppose now that

$$rank[x] \geq r$$

If $|list[x]|$ drops below $\frac{size[x]}{2}$ adds elements to $list[x]$ until either

$$|list[x]| \geq size[x]$$

or until x becomes empty.

8. Inductive Proof for Upper Bound #1: The next step is to prove by induction that

$$|list[x]| \leq 3size[x]$$

If

$$rank[x] \geq r$$

the claim is obvious. It is shown now that if the claim holds for all vertexes of rank $k-1$, then it also holds for vertexes of rank k.

9. Inductive Proof for Upper Bound #2: Let x be a node of rank k. New elements are added to $list[x]$ only when

$$|list[x]| < size[x]$$

A $sift$ operation concatenates $list[y]$ to $list[x]$, where y is a child of x. As y is of rank $k-1$, it follows by induction that

$$|list[y]| < 3size[y] \leq 3 . \frac{2}{3} size[x] = 2size[x]$$

Thus

$$|list[x] \cup list[y]| < 3size[x]$$

as claimed.

10. Nodes in a Rank Lemma: If n elements are inserted into soft heaps, then the number of nodes of rank k is at most $\frac{n}{2^k}$.
11. Proof of the above Lemma: Proof is done by induction on k. A node of rank 0 is generated only when a new element is inserted – using a $make - heap$ operation – into a soft heap. Thus, the number of elements of rank 0 is at most n as claimed. An element of rank k is generated only when two roots of rank $k - 1$ are combined.
12. Bound of Corrupted Elements Lemma: If n elements are inserted into soft heaps, then the total number of corrupted elements contained in the heaps, at any given time, is at most εn.
13. Nodes containing the Corrupted Elements: Each node of rank at most r contains a single element. Thus, all corrupted elements belong to nodes of rank greater than r.
14. Count of Nodes and Elements: By the Nodes in a Rank Lemma, the number of nodes in a rank k is at most $\frac{n}{2^k}$. By the Node Size Bounds Lemma, a node of rank

$$k > r$$

contains at most

$$3s_k < 6\left(\frac{3}{2}\right)^{k-r}$$

elements.

15. Total Count of Corrupted Elements: As

$$r = \left\lceil \log_2 \frac{1}{\varepsilon} \right\rceil + 5$$

the number of corrupted elements is at most

$$\sum_{k>r} \frac{n}{2^k} . 3s_k < \frac{n}{2^r} \sum_{k>r} 6\left(\frac{1}{2}\right)^{k-r}\left(\frac{3}{2}\right)^{k-r} = 6\frac{n}{2^r}\sum_{i}\left(\frac{3}{4}\right)^{i} = \frac{18n}{2^r} < \varepsilon n$$

Amortized Analysis

1. Potentials Assigned to Heaps/Trees: Kaplan and Zwick (2009) assign potentials to heaps, trees, and nodes. A heap of rank k has a potential of $k + 1$. A tree whose root is x has a potential of $(r + 2) . del(x)$ where $del(x)$ is the number of elements deleted from x since the last $sift(x)$ operation, or since the creation of x.
2. Potentials Assigned to Root/Leaf Nodes: If x is a root node of rank k, then x has a potential of $k + 7$. If x is a non-root node, it has a potential of 1.
3. Amortized Analysis of $sift$ Operation: This section starts with an analysis of $sift$. Suppose that x is a node of rank k, that y is a child of x, and that the elements of $list[y]$ are moved to $list[x]$.
4. Case where Child is a Leaf: If

$$|list[y]| < \frac{size[y]}{2}$$

then y is a leaf, and y disappears as a result of this operation. The unit potential released by the disappearance of y pays for this operation.

5. Car-pooling of Non-leaf Children's Elements: If, on the other hand

$$|list[y]| \geq \frac{size[y]}{2}$$

and hence

$$|list[y]| \geq \left\lceil \frac{size[y]}{2} \right\rceil$$

the unit cost of the operation is split among the $|list[y]|$ elements participating in the *car-pool*. The charge for each element is at most

$$\left\lceil \frac{size[y]}{2} \right\rceil^{-1} = \left\lceil \frac{s_{k-1}}{2} \right\rceil^{-1}$$

6. Cumulative Cost of Car-pooling: An element is charged at most once at each rank, thus its total *travel expenses* are

$$\sum_{k \geq 0} \left\lceil \frac{s_k}{2} \right\rceil^{-1} \leq r + 2 \sum_{i \geq 0} \left(\frac{2}{3}\right)^i = r + 6 = \mathcal{O}\left(\log \frac{1}{\varepsilon}\right)$$

7. Potential Change due to Combine: The next operation considered is a $combine(x, y)$ operation which combines two nodes of rank k, rooted at x and y, into a tree of rank $k + 1$ rooted at z. The potentials for both x and y decrease from $k + 7$ to 1, releasing $2k + 2$ units of potential.
8. Costs Incurred during the Operation: One of these units pay for the constant cost of the operation, $k + 8$ units are assigned to z, one unit is used to increase the potential of the heap containing the newly formed tree, if it is now the tree with the largest rank.
9. Accounting of the Combine Costs: The remaining $k + 2$ units are used, if needed, to pay for an ensuing $update - suffix - min$ operation. Note that

$$1 + (k + 2) + 1 + (k + 2) = 2k + 12$$

10. Meld Operation - Merging of Tree Lists: A $meld(P, Q)$ operation receives two heaps of ranks k and k' respectively. Suppose that

$$k \leq k'$$

The $meld$ operation first merges that lists of P and Q, effectively destroying P. This takes $\mathcal{O}(k+1)$ time, which is paid for by the released potential of P.

11. Zero Amortized Cost for Component Operations: The subsequent $combine$ operations pay for themselves and for the ensuing $update - suffix - min$ operation. If no $combine$ operations are performed, the potential released by the destruction of P pays for the $update - suffix - min$ operation.
12. Amortized Cost of $extract - min$: Finally, the amortized cost of $extract - min$ operations are bounded. An $extract - min(P)$ operation locates an element e with minimal current key in constant time.
13. Case where No Sifting is Required: Suppose that x is a root node containing e. If after deleting e from $list[x]$ one still has

$$|list[x]| \geq \frac{size[x]}{2}$$

or if

$$|list[x]| > 0$$

and x is a leaf, then no further action is taken.

14. Incrementing the Deleted Element Count: Note, however, that $del(x)$, the number of elements deleted from $list[x]$ since the last $sift(x)$ operation, is increased by 1, and the potential of the tree whose root is x is increased by $r+2$. This total cost of

$$1 + (r+2) = r + 3$$

is charged to e, which would never be charged again.

15. Case when Sifting is Required: If

$$list[x] < \frac{size[x]}{2}$$

and x is not a leaf, then a $sift(x)$ is performed.

16. Potential of Tree before $Sift$: As x is not a leaf, it must have had at least $size[x]$ elements in its list after the previous $sift(x)$ operation. Thus

$$del(x) \geq \left\lceil \frac{size[x]}{2} \right\rceil$$

and the potential of the tree, prior to the $sift$ operation, is at least

$$(r+2)\left\lceil \frac{size[x]}{2} \right\rceil = (r+2)\left\lceil \frac{s_k}{2} \right\rceil$$

where

$$k = rank[x]$$

17. Bounding of the above Potential: It is not difficult to verify that for every

$$k \geq 0$$

one has

$$(r+2)\left\lceil \frac{s_k}{2} \right\rceil \geq k+1$$

Indeed for

$$0 \leq k \leq r$$

one has

$$(r+2)\left\lceil \frac{s_k}{2} \right\rceil = r+2 \geq k+1$$

For

$$k = r+2$$

one has

$$s_{r+2} = 3$$

and thus

$$(r+2)\left\lceil \frac{s_{r+2}}{2} \right\rceil = 2(r+2) > r+3$$

For

$$k \geq r+3$$

one has

$$(r+2)\left\lceil \frac{s_k}{2} \right\rceil \geq \frac{r+2}{2}\left(\frac{3}{2}\right)^{k-r} \geq k+1$$

as

$$\frac{1}{2}\left(\frac{3}{2}\right)^{k-r} \geq \frac{k+1}{r+2}$$

for

$$k \geq r+3$$

18. Amortized $update - suffix - min$ Operation: This decrease of at least $k+1$ in the potential of the tree pays for the $update - suffix - min$ operation that follows the $sift(x)$ operation.
19. Clean-up after the Last Leaf: Finally, if the root node x is a leaf, and e is the last element in $list[x]$, then x and the tree rooted at x are removed. The $k+7$ units of potential of x are more than enough to pay for the $update - suffix - min$ operation performed after the removal of x.
20. Inserting an Element into the Heap: When an element e is inserted into a soft heap, a new heap, a new tree, and a new node are created. The heap, of rank 0, is assigned one potential unit, the tree zero units, while the new node, of rank 0, is assigned 7 units.
21. Costs Incurred during the Elements' Lifetime: During its lifetime, at most $r+6$ potential units are charged to e to pay for its movements. Finally, an additional $r+3$ units are charged to e when it is deleted.
22. Bounds on the Total Costs Incurred: On charging then one unit of actual work involved in inserting e into a soft heap of its own, and the 8 potential units assigned to this heap to e, one gets that the total charge for e, from its insertion until its deletion, is at most

$$8 + (r+6) + (r+3) = 2r + 17 = \mathcal{O}\left(\log\frac{1}{\varepsilon}\right)$$

23. Amortized Heap Operations Cost Lemma: The amortized cost of inserting an element into a soft heap with error rate ε is $\mathcal{O}\left(\log\frac{1}{\varepsilon}\right)$. The amortized cost of all other operations is 0.

Adding a Delete Operation

1. Amortized Analysis of $delete$ Operation: The next item to consider is the implementation of the $delete$ operation. One option, used by Chazelle (2000b), is to implement the $delete$ operations in a *lazy* manner. Deleted elements are simply marked as such.
2. Chazelle's Lazy Implementation of $delete$: If an $extract - min$ operation returns an element marked as deleted, the operations is simply called again until a non-deleted element is returned.
3. Drawback of Lazy $delete$: The drawback of such an implementation is that it is not space optimal, and the space used by deleted elements cannot be reclaimed immediately. This can be fixed by re-building the data structure when more than half the elements are deleted.
4. Direct Implementation of the $delete$ Operation: The $delete(e)$ operation is implemented directly as follows. e is deleted from the linked list $list[x]$ currently containing it. This can be easily done in constant time, using the forward and the backward references of e in the list, without knowing the identity of x.
5. Extraction of the Node containing the Element: The node x can be retrieved using a union-find data structure, but this is too expensive in the current context. It is assumed, however, that the first element in the linked list knows to which list it belongs to.
6. Node after Deletion of the Element: If e is the last remaining element of $list[x]$, a $sift(x)$ operation can be initiated to bring new elements into $list[x]$. If x is a leaf, it

is removed from the data structure. To implement this, however, a reference to the parent of x in the tree needs to be maintained.

7. Ghost Elements in the List: For each node x, one keeps a number $num[x]$ that gives the number of elements in $list[x]$, including *ghost* elements that were deleted from $list[x]$ of from lists that were appended from $list[x]$. One may thus have

$$|list[x]| < num[x]$$

8. Running Count of all Elements: When $list\big[left[x]\big]$ is appended to $list[x]$, one does

$$num[x] \leftarrow num[x] + num\big[left[x]\big]$$

$num[x]$ is not decremented when an element is deleted from $list[x]$, as it is not known when that happens.

9. Maintaining Count through the Operations: In fact, $min[x]$ values should be maintained through all the implementations. All occurrences of $|list[x]|$ discussed earlier should be replaced by $num[x]$, and $num[x]$ values should be updated when the lists are created and moved.
10. Ghost Element Impact on Amortized Analysis: A moment's reflection shows that the amortized analysis of the previous section remains valid. Some of the operations are charged to ghost elements, i.e., elements that were already deleted, but this is legal.

Comparison with Chazelle's Implementation

1. Simplified Binary-Tree Soft Heap: The main difference between Kaplan and Zwick (2009) implementation and the Chazelle (2000b) implementation is that the former uses *binary* trees, whereas Chazelle's implementation uses *binomial* trees. Kaplan and Zwick (2009) believe that their implementation is simpler and more intuitive, as argued in this section.

2. Binarization Inherent in Binomial Trees: Chazelle's binomial trees are *binarized*. Each binomial tree is represented as a binary tree, with each node of the binomial tree corresponding to a *left* path in the binary tree. Thus, only root nodes, or nodes that are right children of their parents, have elements and keys associated with them. In the binary trees of Kaplan and Zwick (2009), all nodes play the same role.
3. Use of Partial Binomial Trees: The trees in Chazelle's implementation are actually *partial* binomial trees, as tree nodes that remain without elements are deleted. In a standard binomial tree, a node of rank k has exactly k children. In Chazelle's partial binomial trees, a node of rank k may have less than k children. If the number of children of an empty root node of rank k drops below $\frac{k}{2}$, Chazelle resorts to a *clean-up* operation that breaks the tree into a collection of trees. This slightly complicates the implementation and makes the analysis somewhat subtler. No such complications arise in the Kaplan and Zwick (2009) implementation.
4. Explicit Control of Element Count: Another important difference is that Kaplan and Zwick (2009) explicit control the number of elements contained in the list of a node of rank k. They believe that this makes their implementation more intuitive and the analysis more transparent.
5. Space Efficient Implementation of $delete$: The way Kaplan and Zwick (2009) implement the $delete$ operation is also different from the way suggested by Chazelle. Their implementation is automatically space efficient, without the need for periodic re-buildings.
6. Simplified and Unified Amortized Analysis: These changes in the implementation enable them to present a simplified and unified amortized analysis.

Concluding Remarks

1. Worst-Case Efficient Implementation: This chapter presented a simpler implementation of Chazelle's soft heaps. It would be interesting to find additional

applications of this data structure. It would also be interesting to know whether the soft heap operations can be implemented in $\mathcal{O}\left(\log\frac{1}{\varepsilon}\right)$ *worst-case* time.

2. Reducing the Number of Comparisons: In the implementation presented, the amortized number of *comparisons* made for each element inserted into soft heaps is $2\log\frac{1}{\varepsilon}+\mathcal{O}(1)$. It is possible to reduce this number to $[1+o(1)]\log\frac{1}{\varepsilon}$ by maintaining $sufmin$ references only for trees of rank greater than, say, $2r$, and keeping the $ckey$ values of the roots of the trees of rank at most $2r$ in a small priority queue.
3. Case of no Corrupted Elements: It is interesting to point out that if one sets

$$r=\infty$$

in the construction above, i.e.,

$$size[x]=1$$

for every node x, a standard meldable priority queue data structure in which no corruptions occur is obtained. Each operation is performed in $\mathcal{O}(\log n)$ amortized time, where n is the number of elements inserted into the priority queues. This may be viewed as an alternative to the celebrated *binomial heaps* data structure of Vuillemin (1978) (see also Cormen, Leiserson, Rivest, and Stein (2009)).
4. Efficient Comparison Based Selection Algorithms: As noted by Chazelle (2000a, 2000b), soft heaps give rise to a new linear-time median selection algorithm, very different from the algorithms of Blum, Floyd, Pratt, Rivest, and Tarjan (1973), Schonhage, Paterson, and Pippenger (1976), and Dor and Zwick (1999). It would be interesting to explore the possibility of using soft heaps to obtain an algorithm that finds the median of n elements using less than $2.95n$ comparisons. The best lower bound on the number of comparisons needed to find the median is currently $(2+\varepsilon)n$ for a fixed but tiny

$$\varepsilon > 0$$

(Dor and Zwick (2001)).

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 Discrepancy Method: Randomness and Complexity* **Cambridge University Press**
- Chazelle, B. (2000b): The Soft-Heap: An Approximate Priority Queue with Optimal Error Rate *Journal of the ACM* **47 (6)** 1012-1027
- Chazelle, B. (2000c): A Minimum Spanning Tree Algorithm with inverse-Ackermann Type Complexity *Journal of the ACM* **47 (6)** 1028-1047
- Cormen, T., C. E. Leiserson, R. Rivest, and C. Stein (2009): *Introduction to Algorithms 3rd Edition* **MIT Press**
- Dor, D., and U. Zwick (1999): Selecting the Median *SIAM Journal on Computing* **28 (5)** 1722-1758
- Dor, D., and U. Zwick (2001): Median Selection Requires $(2 + \varepsilon)n$ Comparisons *SIAM Journal on Discrete Mathematics* **14 (3)** 312-325
- Kaplan, H., and U. Zwick (2009): A Simpler Implementation and Analysis of Chazelle's Soft Heaps *Proceedings of the 2009 Annual ACM-SIAM Symposium on Discrete Algorithms*
- Pettie, S., and V. Ramachandran (2002): An Optimal Minimum Spanning Tree Algorithm *Journal of the ACM* **49 (1)** 16-34
- Pettie, S. and V. Ramachandran (2008): Randomized Minimum Spanning Tree Algorithms using exponentially fewer Random Bits *ACM Transactions on Algorithms* **4 (1)** 1-27

- Schonhage, A., M. Paterson, and N. Pippenger (1976): Finding the Median *Journal of Computer and System Sciences* **13 (2)** 184-199
- 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 19^{th} 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 $\mathcal{O}(m)$. 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 $\mathcal{O}(m+n)$. If the number of vertexes before a phase is n', then the number of phases remaining after a phase is at most $\frac{n'}{\frac{m}{2n'}}$. 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 $\mathcal{O}(m+n)$ 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:

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

(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 $\mathcal{O}(n)$ 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 $\mathcal{O}(m)$ 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 $\mathcal{O}(m)$, *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. $\mathcal{O}(\log n)$ 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 $\mathcal{O}(\log n)$ 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 $\frac{\zeta(3)}{F\prime(0)}$, 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. Starting Vertex for the Algorithm: 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. Choices for implementing the Queue: 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. Determinants of the Time Complexity: 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. Table Time Complexity by Algorithm:

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

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 ***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 ***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 ***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 ***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 ***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 ***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 ***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 ***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 ***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 ***P*** holds again.
8. Final Step of the Inductive Proof: As a result, ***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 references, 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 $\mathcal{O}\big(m\ \alpha(m,n)\big)$ 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 $\mathcal{O}\big(m\ \alpha(m,n)\big)$ where α is the functional inverse of Ackermann's function defined in Tarjan (1975). The algorithm is comparison-based it uses references, 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 $\{C_{z_i}\}$.
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 $\mathcal{O}(m + d^3 n)$ 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. Expansion Node Level Vertex Count: 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. Level Vertex Count of $\mathfrak{T}$: 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. MST Runtime Inductive Proof #1: 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. MST Runtime Inductive Proof #2: 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

$$b(t-1)\left[m + \sum_z S(t, d_z - 1)^3 S\big(t-1, S(t, d_z-1)\big)^3\right]$$
$$= b(t-1)\left[m + \sum_z S(t, d_z)^3\right]$$

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 3rd 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	$\mathcal{O}(\|V\| + \|E\|)$ 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	$\mathcal{O}(\|V\|)$ 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. Symbology for DFS Ordering Statement: 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. Formal Statement of DFS Enumeration: 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

Dijkstra's Algorithm

Introduction

1. Focus of the Dijkstra's Algorithm: *Dijkstra's algorithm* (or *Dijkstra's Shortest Path First Algorithm*, *SPF Algorithm*) (Wikipedia (2020)) is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks (Dijkstra (1959)).
2. Variants of the Dijkstra Algorithm: The algorithm exists in many variants. Dijkstra's original algorithm found the shortest path between two given nodes (Dijkstra (1959)), but a more common variant fixes a single node as the *source* node and finds the shortest path from the source to all other nodes in the graph, producing a shortest path tree.
3. Applications of the Algorithm: For a given source node in the graph, the algorithm finds the shortest path between that node and every other (Mehlhorn and Sanders (2008)). It can also be used for finding the shortest path to a single destination node by stopping the algorithm once the shortest path to the destination has been determined. For example, if the nodes of the graph represent the cities and the edge path costs represent driving distances between pairs of cities connected by a direct road, Dijkstra's algorithm can be used to find the shortest route from one city to all other. A widely used application of the shortest path algorithm is network routing protocols, notably IS-IS (Intermediate System to Intermediate System) and OSPF (Open Shortest Path First). It is also employed as a sub-routine in other algorithms such as Johnson's.
4. Generalization of the Dijkstra's Algorithm: The Dijkstra algorithm uses labels that are positive or real numbers, which are totally ordered. It is be generalized to use any labels that are partially ordered, provided that the subsequent labels – produced when traversing an edge – are monotonically non-decreasing. This generalization is called

the generic Dijkstra shortest-path algorithm (Szczesniak, Jajszczyk, and Wozna-Szczesniak (2019)).

5. Performance Customization for Specialized Solutions: Dijkstra algorithm uses a data structure for storing and querying partial solutions sorted by distance from the start. The original algorithm uses a $min - priority$ queue and runs in time $\mathcal{O}(|V|^2)$ – where $|V|$ is the number of nodes. The idea of this algorithm is also given in Leyzorek, Gray, Johnson, Ladew, Meaker, Petry, and Seitz (1957). Fredman and Tarjan (1987) propose using a Fibonacci heap min-priority queue to optimize the running time complexity to $\mathcal{O}(|E| + |V| \log|V|)$ where $|E|$ is the number of edges. This is asymptotically the fastest-known single-source shortest path algorithm for arbitrary directed graphs with unbounded non-negative weights. However, specialized cases – such as bounded/integer weights, directed acyclic graphs, etc. – can indeed be improved further as detailed in the section on Specialized Variants.
6. Characteristics of the Algorithm:

Class	Search Algorithm
Data Structure	Graph
Worst-case Performance	$\mathcal{O}(\|E\| + \|V\| \log\|V\|)$

7. Special Variant – Uniform Cost Search: In some fields, artificial intelligence in particular, Dijkstra's algorithm or a variant of it is known as *uniform cost search* and formulated as an instance of the more general idea of best-first search.

Algorithm

1. Idea behind the Dijkstra Algorithm: Let the starting node be called the *initial node*. Let the *distance of node* Y be the distance from the initial node to Y. Dijkstra's algorithm will assign some initial distance values and try to improve them step-by-step.

2. Maintain the Set of Unvisited Nodes: Mark all nodes unvisited. Create a set of all unvisited nodes called the *unvisited set*.
3. Assignment of Beginning Node Distances: Assign to every node a tentative distance value; set it to zero for the initial node and to infinity for all other nodes. Set the initial node as current.
4. Comparison Based Node Distance Update: For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances through the current node. Compare the newly created *tentative* distance and assign the smaller value.
5. Criteria for Marking as *Unvisited*: When all of the unvisited neighbors of the current node have been visited, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.
6. Criterion for Algorithm Termination: If the destination node has been marked visit – when planning a route between two specific nodes – or if the smallest tentative distance among the nodes in the *unvisited set* is infinity – when planning a complete traversal; this occurs when there is no connection between the initial node and the remaining unvisited nodes – then stop. The algorithm has finished.
7. Criteria for the Algorithm Iteration: Otherwise, select the unvisited node that is marked with smallest tentative distance, set it as the current node, and go back to the step *Comparison based Distance Update*.
8. Termination before Setting *Destination Visited*: When planning route, it is actually not necessary to wait until the destination node is *visited* as above; the algorithm can stop once the destination node has the smallest tentative distance among all *unvisited* nodes – and thus could be selected as the next *current*.

Characteristics

The algorithm makes no attempt of direct exploration towards the destination as one might expect. Rather, the sole consideration in determining the next *current* intersection is its distance from the starting point. This algorithm therefore expands outward from the

starting point, interactively considering every node that is closest in terms of the shortest path distance until it reaches the destination. When understood this way, it is clear how the algorithm necessarily finds the shortest path. However, this may also reveal one the algorithm's weaknesses; its relative slowness in some topologies.

Generalization of the Problem

1. Source to Destination Complete Path Set: A more general problem would be to find all the shortest paths between *source* and *target* – there may be several ones of the same length. Then, instead of storing only a single node in each entry of $prev[]$, all the vertexes that satisfy the relaxation condition would be stored.
2. Storing the Set of Parents: For example, if an intermediate node r and $source$ connect to $target$ and both of them lie on different shortest paths through $target$ – because the edge cost is the same in both cases – both r and $source$ would be added to $prev[target]$. When the algorithm completes, $prev[]$ data structure will actually describe a graph that is a subset of the original graph with some edges removed.
3. DFS Based Path Finding Algorithm: Its key property will be that is the algorithm is run with some starting node, then every path from that node to every other node in the new graph will be the shortest path between these nodes in the original graph, and all paths of that length in the original graph will be present in the new graph. Then to actually find all the shortest paths between two given nodes, a path finding algorithm like depth first search will be used on the new graph.

Using a Priority Queue

1. Minimum Priority Queue Data Structure: A min$_priority$ queue is an abstract data type that provides three basis operations: $add\_with\_priority$, $decrease\_priority$, and $extract\_min$. As mentioned earlier, using such a data structure can lead to faster

computing times than using a basic queue. Notably, the Fibonacci heap (Fredman and Tarjan (1987)) or Brodal queue offer optimal implementations for these three operations.

2. Gradually Filling up the Queue: Instead of filling up the priority queue with all nodes in the initialization phase, it is also possible to initialize it with only those containing the $source$; then, inside the shortest path examination block, the node must be inserted if not already in the queue, instead of performing a $decrease\_priority$ operation (Mehlhorn and Sanders (2008)).
3. Data Structures with Faster Runtimes: Other data structures can be used to achieve even faster runtimes in practice (Chen, Chowdhury, Ramachandran, Roche, and Tong (2007)).

Proof of Correctness

1. Induction on the Visited Vertexes: Proof of Dijkstra's algorithm is constructed by induction on the number of visited nodes.
2. Invariant Hypothesis for Visited Vertexes: For each visited node v, $dist[v]$ is considered the shortest distance from the $source$ to v; and for each unvisited node u, $dist[u]$ is assumed to be the shortest distance when traveling via visited nodes only, from $source$ to u. This assumption is only considered if a path exists, otherwise the distance is set to infinity. Note that $dist[u]$ is not considered to be the actual shortest distance for unvisited nodes.
3. Triviality of the Inductive Base Case: The base case is when there is just one visited node, namely the initial node $source$, in which case the hypothesis is trivial.
4. Assuming Validity for Arbitrary $n-1$: Assume that the hypothesis is valid for $n-1$ visited nodes. In this case, one chooses an edge vu where u has the least distance $dist[u]$ and the edge vu is such that

$$dist[u] = dist[v] + length(u, v)$$

$dist[u]$ is considered to be the shortest distance from $source$ to u because, if there were a shorter path, and w was the first unvisited node in the alternate path, then by the original hypothesis,

$$dist[w] > dist[u]$$

which creates a contradiction. Similarly, if there were a shorter path to u without using unvisited nodes, and if the last but one node on that path were w, then

$$dist[u] = dist[w] + length(u, w)$$

which is also a contradiction.

5. Final Step of the Induction: After processing u it will still be true that for each unvisited node w, $dist[w]$ will be the shortest distance from $source$ to w using visited nodes only, because if there were a shorter path doesn't go by u if would have been found out previously, and if there were a shorted path using u it would have been updated when processing u.

Running Time

1. Edges/Vertex Based Running Time: Bounds of the running time of a Dijkstra algorithm on a graph with edges E and vertexes V can be expressed as a function of the number of edges denoted $|E|$, and the number of vertexes denoted $|V|$, using the big-$\mathcal{O}$ notation. The complexity bound depends mainly on the data structure used to represent the set Q. In the following, upper bounds can be simplified because $|E|$ is $\mathcal{O}(|V|^2)$ for any graph, but that simplification disregards the fact that in some problems, other upper bounds on $|E|$ may hold.

2. List Based Running Time Estimate: For any data structure for the vertex set Q, the running time is $\mathcal{O}(|E| . T_{dk} + |V| . T_{em})$ where T_{dk} and T_{em} are the complexities of $decrease - key$ and $extract - min$ operations in Q, respectively. The simplest version of Dijkstra's algorithm stores the vertex list as an ordinary linked list or array, and $extract - min$ is simply a linear search through all the vertexes in Q. In this case, the running time is

$$\mathcal{O}(|E| + |V|^2) = \mathcal{O}(|V|^2)$$

3. Adjacency List Based Graph Representation: If the graph is stored as an adjacency list, the running time for a dense graph – where

$$|E| \in \mathcal{O}(|V|^2)$$

is $\mathcal{O}(|V|^2 \log|V|)$ For sparse graphs, that is graphs with far fewer than $|V|^2$ edges, Dijkstra's algorithm can be more efficiently implemented by storing the graphs in the form of adjacency lists using a self-balancing binary search tree, binary heap, pairing heap, or Fibonacci heap as a priority queue to implement extracting minimum efficiently.
4. Times for Different Queue Types: To perform $decrease - key$ steps in a binary heap efficiently, it is necessary to use an auxiliary data structure that maps each vertex to its position in the heap and to keep this structure up to date as the priority queue Q changes. With a self-balancing binary search tree or binary heap, the algorithm requires $\mathcal{O}([|E| + |V|] \log|V|)$ time in the worst case, where log represents the binary logarithm $\log_2(.)$; for connected graphs this can be simplified to $\mathcal{O}(|E| \log|V|)$. The Fibonacci heap improves this to $\mathcal{O}(|E| + |V| \log|V|)$.
5. Binary Heap Average Case Times: When using binary heaps, the average case time complexity is lower than the worst case; assuming that edge costs are drawn independently from a common probability distribution, the expected number of

$decrease - key$ operations is bounded by $\mathcal{O}\left(|V| \log \frac{|E|}{|V|}\right)$, giving a total running time of $\mathcal{O}\left(|E| + |V| \log \frac{|E|}{|V|} \log |V|\right)$ (Mehlhorn and Sanders (2005)).

Practical Optimizations and Infinite Graphs

1. Lazy Initialization of the Priority Queue: In common presentations of the Dijkstra's algorithm, initially all nodes are entered into the priority queue. This is, however, not necessary; the algorithm can start with a priority queue that contains only one item, and insert new items as they are discovered – instead of doing a $decrease - key$, check whether the key is in the queue, and if it is, decrease the key, otherwise insert it (Mehlhorn and Sanders (2008)).This variant has the same worst-case bounds as the common variant, but otherwise maintains a smaller priority queue in practice, speeding up the queue operations (Felner (2011)).
2. Size Benefits of Lazy Initialization: Moreover, not inserting all nodes in the graph makes it possible to extend the algorithm to find the shortest path from a single source to the closest of a set of target nodes on infinite graphs or those too large to represent in memory. The resulting algorithm is called *uniform cost search* in artificial intelligence literature (Nau (1983), Russell and Norvig (2009), Felner (2011)).
3. Alternative Runtime Complexity Estimate: The complexity for the above search can be expressed in an alternate way for very large graphs; when C^* is the length of the shortest path from the start node to any node satisfying the *goal* predicate, each edge has a cost at least ε, and the number of neighbors per node is bounded by b, then the algorithm's worst case time and space complexity is both $\mathcal{O}\left(b^{1+\left\lfloor \frac{C^*}{\varepsilon} \right\rfloor}\right)$ (Russell and Norvig (2009)).
4. Further Optimizations for Specific Situations: Further optimization of the Dijkstra's algorithm for the single-target case includes bi-directional variants, goal directed variants such as the A^* algorithm, graph pruning algorithms to determine which nodes

are likely to form the middle segment of the shortest paths (reach-based routing), and hierarchical descriptions of the input graph that reduce $s - t$ routing to connecting s and t to their respective *transit nodes* followed by the shortest computation between these transit node using a *highway* (Wagner and Willhalm (2007)). Combinations of such techniques may be needed for optimal practical performance on specific problems (Bauer, Delling, Sanders, Schieferdecker, Schultes, and Wagner (2010)).

Specialized Variants

1. Bounded, Small Integer Arc Weights: When arc weights are small integers bounded by a parameter C, a monotone priority queue can be used to speed up the Dijkstra's algorithm. The first algorithm of this type was *Dials' algorithm*, which used a bucket queue to obtain a running time $\mathcal{O}(|E| + diam(G))$ that depends on the diameter of a graph with integer edge weights (Dial (1969)). The use of van Emde Boas tree as a priority queue brings the complexity to $\mathcal{O}(|E| \log\log C)$ (Ahuja, Mehlhorn, Orlin, and Tarjan (1990)). Another interesting variant based on the combination of a new radix heap and the well-known Fibonacci heap runs in time $\mathcal{O}(|E| + |V|\sqrt{\log C})$ (Ahuja, Mehlhorn, Orlin, and Tarjan (1990)). Finally, the best algorithms for this special case are as follows. The algorithm given by Thorup (2000) runs in $\mathcal{O}(|E| + \log\log|V|)$ time, and the algorithm given by Raman (1997) runs in $\left(|E| + |V| \min\left([\log|V|]^{\varepsilon+\frac{1}{3}}, [\log|V|]^{\varepsilon+\frac{1}{4}}\right)\right)$ time.
2. Optimization for Directed Acyclic Graph: Also, for directed acyclic graphs, it is possible to shortest paths from a given starting vertex in linear $\mathcal{O}(|V| + |E|)$ time, by processing the vertexes in a topological order, and calculating the path length of each vertex to be then minimum length obtained by any of its incoming edges (Cormen, Leiserson, Rivest, and Stein (2009)).

3. Undirected Connected Graphs with Integer Weights: In the special case of integer weights and undirected connected graphs, Dijkstra's algorithm can be completely countered with a linear $\mathcal{O}(|E|)$ complexity algorithm, given by Thorup (1999).

Related Problems and Algorithms

1. Ranked List of Suboptimal Solutions: The functionality of Dijkstra's original algorithm can be extended with a variety of modifications. For example, sometimes it is desirable to present solutions that are less than mathematically optimal. To obtain a ranked-list of less-than-optimal solutions, the optimal solution is first calculated. A single edge containing the optimal solution is removed from the graph, and an optimal solution to this new graph is calculated. The secondary solutions are then ranked and presented after the first optimal solution.
2. Usage in Link State Routing Protocols: Dijkstra's algorithm is usually the working principle behind link-state routing protocols, with OSPF and IS-IS being the most common ones.
3. Comparison with Bellman-Ford Algorithm: Unlike Dijkstra's algorithm, Bellman-Ford algorithm can be used on graphs with negative edge weights, as long as the graph contains no negative cycle reachable from the source vertex s. The presence of such cycles means that there is no shortest path, since the total weight becomes lower each time the cycle is traversed. It is possible to adapt Dijkstra's algorithm to handle negative weight edges by combining it with the Bellman-Ford algorithm to remove negative edges and detect negative cycles, such an algorithm is called Johnson's algorithm.
4. Relation to the A^* Algorithm: The A^* algorithm is a generalization of the Dijkstra's algorithm that cuts down on the size of the subgraph that must be explored, if additional information is available that provides a lower bound on the distance to the target. This approach can be viewed from the perspective of linear programming; there is a natural linear program for computing shortest paths, and solutions to its dual

linear programs are feasible if and only if they form a consistent heuristic. This feasible dual/consistent heuristic defines a non-negative reduced cost, and A^* is essentially running Dijkstra with those reduced costs. If the dual satisfies the weaker condition of admissibility, then A^* is instead more akin to the Bellman-Ford program.

5. Relation to Prim's Algorithm: The process that underlies the Dijkstra's algorithm is similar to the greedy process used in Prim's algorithm. Prim's purpose is to find the minimum spanning tree that connects all nodes in a graph; Dijkstra is concerned only with two nodes. Prim does not evaluate the total weight of the path from the starting node, only the individual edges.
6. Relation to Breadth First Search: Breadth-first-search can be viewed as a special case of the Dijkstra algorithm on unweighted graphs, where the priority queue degenerates into a FIFO queue.
7. Relation to Fast-Marching Method: The fast-marching method can be viewed as a continuous version of the Dijkstra's algorithm which computes geodesic distances in a triangular mesh.

Dynamics Programming Perspective

1. SPF Dynamic Programming Functional Equations: From a dynamic programming point of view, Dijkstra's algorithm is a successive approximation scheme that solves the dynamic programming functional equation for the shortest path problem by the *Reaching* method (Denardo (2003), Sniedovich (2006, 2010)).
2. SPF under the Principle of Optimality: Dijkstra's explanation of the logic behind the algorithm is as follows (Dijkstra (1959)): *Find the path of minimal total length between two givens nodes P and Q; This uses the fact that, if R is a node on the minimal path from P to Q, knowledge of this implies the knowledge of the minimal path from P to R.* The above explanation is a paraphrase of Bellman's *Principle of Optimality* in the context of the shortest path problem.

References

- Ahuja, R. K., K. Mehlhorn, J. B. Orlin, and R. E. Tarjan (1990): Faster Algorithms for the Shortest Path Problem *Journal of the ACM* **37 (2)** 213-223
- Bauer, R., D. Delling, P. Sanders, D. Schieferdecker, D. Schultes, and D. Wagner (2010): Combining Hierarchical and Goal-directed Speed-up Techniques for Dijkstra's Algorithm
- Chen, M., R. A. Chowdhury, V. Ramachandran, D. L. Roche, and L. Tong (2007): Priority Queues and Dijkstra's Algorithm *UTCS Technical Report TR-07-54* **University of Texas**
- Cormen, T., C. E. Leiserson, R. Rivest, and C. Stein (2009): *Introduction to Algorithms 3rd Edition* **MIT Press**
- Denardo, E. V. (2003): *Dynamics Programming: Models and Applications* **Dover Publications** Mineola NY
- Dial, R. B. (1969): Algorithm 360: Shortest Path with Topological Ordering [H] *Communications of the ACM* **12 (11)** 632-633
- Dijkstra, E. W. (1959): A Note on Two Problems in Connection with Graphs *Numerische Mathematik* **1** 269-271
- Felner, A. (2011): Position Paper: Dijkstra's Algorithm versus Uniform Cost Search or a Case against Dijkstra's Algorithm *Proceedings of the 4th International Symposium on Combinatorial Search* 47-51
- 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
- Leyzorek, M., R. S. Gray, A. A. Johnson, W. C. Ladew, S. R. Meaker Jr., R. M. Petry, and R. N. Seitz (1957): *Investigation of Model Techniques – First Annual Report – A Study of Model Techniques for Communication Systems* **Case Institute of Technology**
- Mehlhorn, K. W., and P. Sanders (2008): *Algorithms and Data Structures: The Basic Toolbox* **Springer**

- Nau, D. S. (1983): Expert Computer Systems *IEEE Computer* **16 (2)** 63-85
- Raman, R. (1997): Recent Results on the Single-source Shortest Paths Problem *SIGACT News* **28 (2)** 81-87
- Russell, S., and P. Norvig (2009): *Artificial Intelligence: A Modern Approach 3rd Edition* **Prentice Hall**
- Sniedovich, M. (2006): Dijkstra's Algorithm Re-visited; the Dynamic Programming Connection *Journal of Control and Cybernetics* **35 (3)** 599-620
- Sniedovich, M. (2010): *Dynamic Programming: Foundations and Principles* **Taylor and Francis**
- Szczesniak, I., A. Jajszczyk, and B. Wozna-Szczesniak (2019): Generic Dijkstra for Optical Networks *Journal of Optical Communications and Networking* **11 (11)** 568-577
- Thorup, M. (1999): Undirected Single-Source Shortest Paths with Positive Integer Weights in Linear Time *Journal of the ACM* **46 (3)** 362-394
- Thorup, M. (2000): On RAM Priority Queues *SIAM Journal on Computing* **30 (1)** 86-109
- Wagner, D., and T. Willhalm (2007): Speed-Up Techniques for Shortest-Path Computations
- Wikipedia (2020): Dijkstra's Algorithm

Bellman-Ford Algorithm

Overview

1. Focus of Bellman-Ford Algorithm: The Bellman-Ford algorithm is an algorithm that computes the shortest path from a single source vertex to all of the other vertexes in a weighted digraph (Bang-Jensen and Gutin (2008). Wikipedia (2020)). It is slower than Dijkstra's algorithm for the same problem but is more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers.
2. Algorithm's Handling of Negative Cycles: Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm (Sedgewick and Wayne (2011)). If a graph contains a *negative cycle*, i.e., a cycle whose edges sum to a negative value, that is reachable from any source, then there is no *cheapest* path; any path that has a point on a negative cycle can be made cheaper by one more walk around the negative cycle. In such a case, the Bellman-Ford algorithm can detect and report the negative cycle (Bang-Jensen and Gutin (2008), Kleinberg and Tardos (2022)).
3. Characteristics:

Class	Single-source shortest-path problem for weighted directed graphs
Data Structure	Graph
Worst-case Performance	$\Theta(\|V\|\|E\|)$
Best-case Performance	$\Theta(\|E\|)$
Worst-case Space Complexity	$\Theta(\|V\|)$

The Algorithm

1. Similarities with the Dijkstra Algorithm: Like Dijkstra's algorithm, Bellman-Ford proceeds by relaxation, in which approximations to the correct distance are replaced by better ones until they eventually reach the solution. In both algorithms, the approximate distance to each vertex is always an over-estimate of the true distance, and is replaced by the minimum of its old value and the length of the newly found path.
2. Differences with the Dijkstra's Algorithm: However, Dijkstra's algorithm uses a priority queue to select greedily the closes vertex that has not yet been processed, and performs this relaxation on all of its outgoing edges; by contrast, the Bellman-Ford algorithm simply relaxes *all* the edges, and does this $|V| - 1$ times, where $|V|$ is the number of vertexes in the graph.
3. Gradual Approach towards the Solution: In each of these repetitions, the number of vertexes with correctly calculated distances grows, from which it follows that eventually all vertexes will have their correct distances. This method allows the Bellman-Ford algorithm to be applied to a wider class of inputs than Dijkstra.
4. Algorithm's Worst-case Runtime: Bellman-Ford runs in $\mathcal{O}(|V| . |E|)$ time where $|V|$ and $|E|$ are the number of vertexes and edges, respectively.
5. Initializing the Distances from the Source: The algorithm first initializes the distance to the source to zero and all other nodes to infinity. Then for all edges, if the distance to the destination can be shortened by taking the edge, the distance is updated to the new lower value.
6. Shortest Path in i Edges: At each iteration i that the edges are scanned, the algorithm finds all shortest paths of length of at most i edges, as well as some paths longer than i edges.
7. Longest Possible Path without a Cycle: Since the longest possible path without a cycle can be $|V| - 1$ edges, the edges must be scanned $|V| - 1$ times to ensure that the shortest path has been found for all nodes.

8. Check for Negative-weight Cycles: A final scan of all the edges is performed, and if any distance is updated, then a path of length $|V|$ edges has been found which can only occur if at least one negative cycle exists in the graph.

Proof of Correctness

1. Intermediate Vertex Distance Lemma: The correctness of the lemma can be shown by induction. The lemma contains two statements. After i iterations of the main loop:
    a. If $distance(u)$ is not infinity, it is equal to the length of some path from s to u, and:
    b. If there is a path from s to u with at most i edges, then $distance(u)$ is at most the length of the shortest path from s to u with at most i edges.
2. Verifying the Base Inductive Case: For the base case of induction, consider

$$i = 0$$

and the moment before the main loop is executed for the first time. Then, for the source vertex,

$$source.distance = 0$$

which is correct. For other vertexes u

$$u.distance = \infty$$

which is also correct because there is no path from $source$ to u with 0 edges.
3. First Part of the Induction: For the inductive case, the first part is proved first. Applying the inductive assumption when a vertex's distance is updated by

$$v.distance := u.distance + uv.weight$$

it is clear that $u.distance$ is the length of some path from $source$ to u. Then $u.distance + uv.weight$ is the length of the path from $source$ to v that follows the path from $source$ to u and then goes to v.

4. Second Part of the Induction: For the second part, consider a shortest path P – there may be more than one – from source to u with at most i edges. Let v be the last vertex before u on this path. Then, the part of the path from $source$ to v is a shortest path from $source$ to v with at most $i - 1$ edges, and one could then append uv to this path with at most i edges that is strictly shorter than P – a contradiction. By inductive assumption, $v.distance$ after $i - 1$ iterations is at most the length of this path from $source$ to v. Therefore, $uv.weight + v.distance$ is at most the length of P. In the i^{th} iteration, $u.distance$ gets compared with $uv.weight + v.distance$ and is set equal to it if $uv.weight + v.distance$ is smaller. Therefore, after i iterations, $u.distance$ is at most the length of P, the length of the shortest path from $source$ to u that uses at most i edges.
5. Verifying the Negative Weight Cycle: If there are no negative-weight cycles, then every shortest path visits each vertex at most once, so that, finally, no further improvements can be made. Conversely, suppose no improvement can be made. Then, for any cycle with vertexes $v[0], \cdots, v[k-1]$

$$v[i].distance \leq v[i - 1\ (mod\ k)].distance + v[i - 1\ (mod\ k)].v[i].weight$$

Summing around the cycle, the $v[i].distance$ and $v[i - 1\ (mod\ k)].distance$ terms cancel, leaving

$$0 \leq sum\ from\ 1\ to\ k\ of\ v[i - 1\ (mod\ k)].v[i].weight$$

i.e., ever cycle has a non-negative weight.

Finding Negative Weights

When the algorithm is used to find the shortest paths, the existence of negative cycles is a problem, preventing the algorithm from finding a correct answer. However, since it terminates upon finding a negative cycle, the Bellman-Ford algorithm can be used in applications in which this is the target to be sought – for example, in cycle-canceling techniques in network flow analysis (Bang-Jensen and Gutin (2008)).

Applications in Routing

1. Distributed Version of Bellman-Ford: A distributed version of the Bellman-Ford algorithm is used in distance-vector routing protocols, for example, the Routing Information Protocol (RIP). The algorithm is distributed because it involves a number of nodes – routers – within an Autonomous System, a collection of IP networks typically owned by an ISP.
2. Distance Vector Routing Protocol Steps: It consists of the following steps:
    a. Each node calculates the distance between itself and all other nodes within the AS and stores this information in a table.
    b. Each node sends its table to all neighboring nodes.
    c. When a node receives distance table from its neighbors, it calculates the shortest routes to all other nodes and updates its own tables to reflect any changes.
3. Drawbacks of the Bellman-Ford Scheme: The main disadvantages of the Bellman-Ford algorithm in this setting are as follow:
    a. It does not scale well.
    b. Changes in network topology are not reflected quickly since updates are spread node-by-node.

c. Count to infinity occurs if link or node failures render a node unreachable from some set of other nodes, these nodes may spend forever gradually increasing their estimates of the distance to it, and in the meanwhile, there may be routing loops.

Improvements

1. Early Termination of the Main Iterator: The Bellman-Ford algorithm may be improved in practice – although not in the worst-case – by the observation that, if the iteration of the main loop of the algorithm terminates without making any changes, the algorithm can be immediately terminated, as subsequent iterations will not make any more changes. With this early termination condition, the main loop may in some cases uses fewer than $|V| - 1$ iterations, even though the worst of the algorithm remains unchanged.
2. Reducing the Number of Relaxations: Yen (1970) describes two more improvements for a Bellman-Ford algorithm for a graph without negative-weight cycles; again, while making the algorithm faster in practice, they do not change its $\mathcal{O}(|V| . |E|)$ worst-case bound. The first improvement reduces the number of relation steps that need to be performed within each iteration of the algorithm.
3. Vertexes that have finished Updating: If a vertex v has a distance value that has not changed since the last time the edges out of v were relaxed, then there is no need to relax the edges out of v a second time. In this way, as number of vertexes without the correct distance values grows, the number whose outgoing edges that need to be relaxed in each iteration shrinks, leading to a constant-factor savings in time for dense graphs.
4. Vertex Ordering and Edge Partitioning: The second improvement first assigns an arbitrary linear order on all vertexes and then partitions the set of edges into two subsets. The first subset E_f contains all edges (v_i, v_j) such that $i < j$ The second, E_b, contains edges (v_i, v_j) such that $i > j$

5. Position Based Vertex Visitation Order: Each vertex is visited in the order $V_0, V_1, \cdots, V_{|V|-1}$, relaxing each outgoing edge from that vertex in E_f. Each vertex is then visited in the order $V_{|V|-1}, V_{|V|-2}, \cdots, V_0$, relaxing each outgoing edge from that vertex in E_b.
6. Reduction in the Worst-case Iteration Count: Each iteration of the main loop of the algorithm, after the first one, adds at least two edges to the set of edges whose relaxed distances match the correct shortest path distances; one from E_f and the other from E_b. This modification reduces the number of iterations of the main loop of the algorithm from $|V| - 1$ to $\frac{|V|}{2}$ (Cormen, Leiserson, Rivest, and Stein (2009), Sedgewick and Wayne (2011)).
7. Random Permutation of the Vertex Ordering: Another improvement, by Bannister and Eppstein (2012), replaces the arbitrary linear order of the vertexes used in Yen (1970) second improvement by a random permutation. This change makes the worst case of Yen's improvement – in which the edges of a shortest strictly alternate between the two subsets E_b and E_f – very unlikely to happen. With a randomly permuted vertex ordering, the expected number of iterations needed in the main loop is at most $\frac{|V|}{3}$ (Sedgewick and Wayne (2011)).

References

- Bang-Jensen, J., and G. Gutin (2008): *Digraphs: Theory, Algorithms, and Applications 2nd Edition* **Springer**
- Bannister, M. J., and D. Eppstein (2011): Randomized Speedup of the Bellman-Ford Algorithm **arXiV**
- Cormen, T., C. E. Leiserson, R. Rivest, and C. Stein (2009): *Introduction to Algorithms 3rd Edition* **MIT Press**
- Kleinberg, J., and E. Tardos (2022): *Algorithm Design 2nd Edition* **Pearson**
- Sedgewick, R. and K. Wayne (2011): *Algorithms 4th Edition* **Addison Wesley**

- Wikipedia (2020): Bellman-Ford Algorithm
- Yen, J. Y. (1970): An Algorithm for Finding Shortest Routes from all Source Nodes to a given Destination in General Networks *Quarterly of Applied Mathematics* **27 (4)** 526-530

Johnson's Algorithm

Overview

1. Johnson's Algorithm – Purpose and Methodology: Johnson's algorithm is a way to find the shortest paths between all pairs of vertexes in an edge-weighted, directed graph. It allows some of the edge weights to be negative, but no negative weight cycles may exist. It works by using a Bellman-Ford algorithm to compute a transformation of the input graph that removes all negative weights, allowing Dijkstra's algorithm to be used on the transformed graph (Johnson (1977), Black (2004), Cormen, Leiserson, Rivest, and Stein (2009), Wikipedia (2019)).
2. Disjoint Paths of Minimum Length: A similar re-weighting technique is also used in Suurballe's algorithm for finding two disjoint paths of minimum total length between the same two vertexes in a graph with non-negative edges (Suurballe (1974)).
3. Characteristics:

Class	All pairs shortest path problem for weighted graphs
Data Structure	Graph
Worst-case Performance	$\mathcal{O}(\|V\|^2 \log\|V\| + \|V\|\|E\|)$

Algorithm Description

1. New Node with Zero Edge Weights: Johnson's algorithm consists of four steps (Black (2004), Cormen, Leiserson, Rivest, and Stein (2009)). In the first step, a new node q is added to the graph, connected by zero-weight edge to the other nodes.

2. Bellman Ford with the new Node as the Source: The Bellman-Ford algorithm is used in the second step, starting from the new vertex q, to find for each vertex v the minimum height $h(v)$ of a path from q to v. If this step detects a negative cycle, the algorithm is terminated.
3. Re-weighting of the Original Graph: In the next step, the edges of the original graph are re-weighted using the values computed by the Bellman-Ford algorithm; an edge from u to v, having length $w(u,v)$, is given the new length $w(u,v) + h(u) - h(v)$.
4. Removal of the Inserted Node: Finally, q is removed, and Dijkstra's algorithm is used to find the shortest paths from each node s to every other vertex in the re-weighted graph.

Correctness

1. Uniform Weight Adjustment across Pairs: In the re-weighted graph, all paths between a pair s and t of nodes have then same quantity $h(s) - h(t)$ added to them. To see this, let p be an $s - t$ path. Its weight $W(s,t)$ in the re-weighted graph is given by $[w(s,p_1) + h(s) - h(p_1)] + [w(p_1,p_2) + h(p_1) - h(p_2)] + \cdots + [w(p_n,t) + h(p_n) - h(t)]$. Every $h(p_i)$ is cancelled by $-h(p_i)$ in the bracketed expression, and, therefore, the following remains for $W(s,t)$: $[w(s,p_1) + w(p_1,p_2) + \cdots + w(p_n,t)] + [h(s) - h(t)]$. The bracketed expression is the weight of p in the original weighting.
2. Zero Re-weighting of Q Paths: Since the re-weighting adds the same amount to every $s - t$ path, a path is the shortest path in the original weighting if and only if it is a shortest path after re-weighting. The weight if the edges that belong to a shortest path from q to any node is zero, and therefore the lengths of the shortest paths from q to every node becomes zero in the re-weighted graph; however, they still remain shortest paths.
3. Consequence - No Negative Edge Weights: Therefore, there can be no negative edges; if edge uv had a negative weight after the re-weighting, then the zero-length

path from q to u together with this edge would form a negative-length path from q to v, contradicting the fact that all vertexes have zero distance from q.

4. Dijkstra's Algorithm Optimal Path Guarantee: The non-existence of negative edges ensures the optimality of the paths found by the Dijkstra's algorithm. The distances in the original path may be calculated using the distances calculated by the Dijkstra's algorithm in the re-weighted graph by reversing the re-weighting transformation (Cormen, Leiserson, Rivest, and Stein (2009)).

Analysis

The time complexity of this algorithm, using Fibonacci heaps' implementation of the Dijkstra's algorithm, is $\mathcal{O}(|V|^2 \log|V| + |V||E|)$; the algorithm uses $\mathcal{O}(|V||E|)$ for the Bellman-Ford stage, and $\mathcal{O}(|V| \log|V| + |E|)$ for each of the $|V|$ instantiations of the Dijkstra's algorithm. Thus, when the graph is sparse, the total time can be faster than the Floyd-Warshall algorithm, which solves the problem in $\mathcal{O}(|V|^3)$ (Cormen, Leiserson, Rivest, and Stein (2009)).

References

- Black, P. E. (2004): Johnson's Algorithm
- Cormen, T., C. E. Leiserson, R. Rivest, and C. Stein (2009): *Introduction to Algorithms 3rd Edition* **MIT Press**
- Johnson, D. B. (1977): Efficient Algorithms for Shortest Paths in Sparse Networks *Journal of the ACM* **24 (1)** 1-13
- Suurballe, J. W. (1974): Disjoint Paths in a Network *Networks* **14 (2)** 125-145
- Wikipedia (2019): Johnson's Algorithm

A^* Search Algorithm

Overview

1. Focus on A^* and its Limitations: A^* is a graph traversal and path search algorithm, which is often used on account of its completeness, optimality, and optimal efficiency (Russell and Norvig (2018), Wikipedia (2020)). One major practical drawback is its $\mathcal{O}(b^d)$ space complexity as it stores all generated nodes in memory. Thus, in practical travel-routing systems, it is generally out-performed by algorithms that can pre-process the graph to attain better performance (Delling, Sanders, Schultes, and Wagner (2009)), as well as memory bounded approaches; however, A^* is still the best solution in many cases (Zeng and Church (2009)).
2. Heuristics Based Generalization of Dijkstra: First published by Hart, Nilsson, and Raphael (1968), A^* algorithm can be seen as an extension of Dijkstra's algorithm. A^* achieves better performance by using heuristics to guide its search.

Introduction

1. Origin of the A^* Algorithm: A^* was originally created as part of the Shakey project, which had the aim of building a mobile robot that could plan its own actions. The original proposal was to use the Graph Traverser Algorithm (Doran and Michie (1966)) for path planning (Nilsson (2009)).
2. Initial Cut of the Heuristics: Graph Traverser is guided by a heuristic function $h(n)$, the estimated distance from the node n to the goal node; it entirely ignores $g(n)$, the distance from the start node to n. It was suggested later by Bertram Raphael to use the sum $g(n) + h(n)$ (Nilsson (2009)).

3. Admissibility and Consistency of Heuristics: Peter Hart invented the concepts now called admissibility and consistency of heuristic functions. A^* was originally designed for finding least cost paths when the cost of a path is the sum of its edge costs, but it has been shown that A^* can be used for finding optimal paths for any problem satisfying the conditions of a cost algebra (Edelkamp. Jabbar, and Lluch-Lafuente (2005)).
4. Origin of the Algorithm's Consistency: Hart, Nilsson, and Raphael (1968) constructed a theorem that states that no A^*-like algorithm could expand fewer nodes than A^* if the heuristic function is consistent and A^*'s tie-breaking rule is suitably chosen. Here A^*-like means that the algorithm searches by extending paths originating at the start node one edge at a time, just as A^* does. This excludes, for example, algorithms that search backwards from goal or in both directions simultaneously. In addition, algorithms covered by this theorem must be admissible and *not more informed* than A^*.
5. Validation of the Algorithm's Consistency: A *correction* was published later (Hart, Nilsson, and Raphael (1972)) claiming that consistency was not required, but this was shown to be false in Dechter and Pearl's definitive study of A^*'s optimality – called optimal efficiency – which gave an example of A^* with a heuristic that was admissible but not consistent, expanding arbitrarily more nodes than an alternative A^*-like algorithm (Dechter and Pearl (1985)).

Description

1. Least-cost Best-first Search: A^* is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs; starting from a specific starting node of a graph, it aims to find that path to a given goal node having the smallest cost – least distance traveled, shortest time, etc. It does this by maintaining a tree of paths at the starting node and expanding those paths one edge at a time until the termination criterion is satisfied.

2. Selecting the Path to Extend: At each iteration of its main loop, A^* needs to determine which of its paths to extend. It does so based on the cost of the path and an estimate of the cost required to extend the path all the way to the goal.
3. Components of the Cost Heuristic: Specifically, A^* selects the path that minimizes

$$f(n) = g(n) + h(n)$$

   where n is the next node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic function that estimates the cost of the cheapest path from n to goal. A^* terminates when the path it uses to extend is the path from start to goal or if there are no paths eligible to be extended.
4. Admissible Nature of the Heuristic: The heuristic function is problem-specific. If the heuristic function is admissible, meaning that is over-estimates the actual cost to get to the goal, A^* is guaranteed to return a least-cost path from start to goal.
5. Priority Queue Based Edge Choice: Typical implementations of A^*use a priority queue to perform the repeated selection of minimum estimated cost nodes to expand. This priority queue is known as an *open set* or *fringe*.
6. Node with the Lowest Heuristic: At each step of the algorithm, the node with the lowest $f(x)$ value is removed from the queue, the f and the g values of its neighbors are updated accordingly, and these neighbors are added to the queue.
7. Node-to-Target Cost Heuristic: The algorithm continues until a node has a lower f value than any node in the queue, or until the queue is empty. Goal nodes may be passed over multiple times if there remain other nodes with lower f values, as they may lead to a shorter path to the goal. The f value of the goal is then the cost of the shortest path, since h at the goal is zero in an admissible heuristic.
8. Vertexes on the Shortest Path: The algorithm described so far gives only the length of the shortest path. To find the actual sequence of steps, the algorithm can be easily revised so that each node on the path keeps track of its predecessor. After the algorithm is run, the ending node will point to its predecessor, and so on, until a node's predecessor is the start node.

9. Euclidean Distance as an Admissible Heuristic: As an example, when searching for a shortest route on a map, $h(x)$ might represent the straight-line distances to the goal, since that is physically the smallest possible distance between any two points.
10. Criterion that makes the Heuristic Consistent: If the heuristic h satisfies the additional condition

$$h(x) \leq d(x, y) + h(y)$$

for every edge (x, y) of the graph – where d denotes the length of the edge – then h is called monotone, or consistent. With a consistent heuristic, A^* is guaranteed to find an optimal path without processing any node more than once, and A^* is equivalent to running Dijkstra's algorithm with the reduced cost

$$d'(x, y) = d(x, y) + h(y) - h(x)$$

11. Implementation Caveats: If a node is reached by one path, removed from the priority queue, and subsequently reached by another path, it will be added to the priority queue again. This is essential to guarantee that the path returned is optimal if the heuristic function is admissible but not consistent. If the heuristic is consistent, when a node is removed from the priority queue, the path to it is guaranteed to be optimal, so the comparison with the updated $g(n)$ will always fail if the node is reached again.

Implementation Details

1. Handling Ties on Heuristic Scores: There are as number of optimization or implementation details that can significantly affect the performance of an A^* implementation. The first detail to note is that the way the priority queue handles ties can have a significant effect on performance is some situations. If the ties are broken

so that the queue behaves in a LIFO manner, A^* will behave like depth-first search among equal cost paths, avoiding exploring more than one equally optimal solution.

2. Retention of the Predecessor Node: When a path is required at the end of a search, it is common to keep with each node a reference to the node's parent. At the end of the search, these nodes can be used to recover the optimal path.
3. Case of Identical Processor Nodes: If these references are being kept, then it can be important that the same node doesn't appear in the priority queue more than once, each entry corresponding to a different path to the node, and each with a different cost.
4. Cost Based Tie Breaking: A standard approach here is to check if a node to be added already appears in the priority queue. If it does, then the priority and the parent pointers are changed to correspond to the lower cost path.
5. Searching Element in a Priority Queue: A standard binary heap based priority queue does not directly support the operation of searching for one of its elements, but it can be augmented with a hash-table that maps elements to their position in the heap, allowing this decrease priority operation to be performed in logarithmic time. Alternatively, a Fibonacci heap can perform the same decrease-priority operations in constant amortized time.

Special Cases

1. Dijkstra's Algorithm – Zero Forward Heuristic: Dijkstra's algorithm, as another example of uniform cost search algorithm, can be viewed as a special case of A^* where

$$h(x) = 0$$

for all x (de Smith, Goodchild, and Longley (2007), Hetland (2010)).

2. DFS Implementation using a Global Counter: General depth-first search can be implemented using A^* by considering that there is a global counter C initialized to a very large value. C is assigned to all of the newly discovered neighbors of a node every time a node is processed. After each single assignment, the counter C is decreased by 1. Thus, the earlier a node is discovered, the higher is its $h(x)$ value.
3. Efficiency of the Zero Forward Heuristic: Both Dijkstra's algorithm and depth-first can be implemented more efficiently without including an $h(x)$ value at each node.

Properties – Termination and Completeness

1. A^* Completeness in Finite Graphs: On finite graphs with non-negative edge weights, A^* is guaranteed to terminate and is *complete*, i.e., it will always find a solution – a path from start to goal – if one exists.
2. Conditional Termination in Infinite Graphs: On infinite graphs with a finite branching factor and edge costs that are bounded away from zero, i.e.,

$$d(x, y) > \varepsilon > 0$$

for some fixed ε, A^* is guaranteed to terminate if and only there exists a solution.

Properties – Admissibility

1. Search Algorithm Admissibility vs. Heuristic Admissibility: A search algorithm is said to be *admissible* if it is guaranteed to return to an optimal solution. If the heuristic function used by A^* is admissible, then A^* itself is admissible. An intuitive *proof* of this is as follows.
2. Optimistic Nature of the Heuristic: When A^* terminates its search, it has found a path from start to goal whose actual cost is lower than the estimated cost of any path from

start to goal through any open node, i.e., the node's f value. When the heuristic is admissible, these estimates are optimistic – under the caveats below – so A^* can safely ignore those nodes because they cannot possibly lead to a cheaper solution than the one it already has. In other words, A^* will never overlook the possibility of a lower cost path from start to goal so it will continue to search until no such possibilities exist.

3. Caveat behind the Optimal Heuristic: The actual proof is a bit more involved, because the f values at the open nodes are not guaranteed to be optimistic even if the heuristic is admissible. This is because the g values at the open nodes are not guaranteed to be optimal, so the sum $g + h$ is not guaranteed to be optimistic.

Properties - Optimal Efficiency

1. Definition of A^* Optimal Efficiency: Algorithm A is optimally efficient with respect to a set of alternative algorithms $\boldsymbol{Alts}$ on a set of problems $\boldsymbol{P}$ if for every problem P in $\boldsymbol{P}$ and every algorithm A' in $\boldsymbol{Alts}$ the set of nodes expanded by A by solving P is a subset of – possibly equal to – the set of nodes expanded by A' in solving P.
2. Impact of Heuristic Admissibility/Consistency: The definitive examination of the optimal efficiency of A^* was done by Dechter and Pearl (1985). They considered a variety of definitions of $\boldsymbol{Alts}$ and $\boldsymbol{P}$ in combination with A^*'s heuristic being merely admissible or being both admissible and consistent.
3. Consistent Heuristic and Admissible Algorithm: The most interesting and positive result they proved is that A^*, with its consistent heuristic, is optimally efficient with respect to all A^*-like search algorithms on all *non-pathological* search problems. Roughly speaking, Dechter and Pearl's notion of non-pathological problems is what is now referred to as *tie-breaking*.
4. Heuristic Admissible but not Consistent: This result does not hold it A^*'s heuristic is admissible but not consistent. In that case, Dechter and Pearl show that there exist A^*-

like algorithms that can expand arbitrarily fewer nodes than A^* on some non-pathological problems.

5. Optimally Efficient Node Set Expansion: Optimal efficiency is about the *set* of nodes expanded, not the *number* of node expansions – the number of iterations of A^*'s main loop. When the heuristic being used is admissible but not consistent, it is possible for a node to be expanded by A^* many times, an exponential number of times in the worst-case (Martelli (1977)). In such circumstances, Dijkstra's algorithm would outperform A^* by a large margin.

Bounded Relaxation

1. Relaxing the Optimal Admissibility Criterion: While the admissibility criterion guarantees an optimal solution path, it also means that A^* must examine all equally meritorious paths to find the optimal path. To compute approximate shortest paths, it is possible to speed up the search at the expense of optimality by relaxing the admissibility criterion.
2. Allowance for ε-admissible Relaxations: Often one wants to bound this relaxation, so there is a guarantee that the solution path is no worse than $(1 + \varepsilon)$ times the optimal path solution. This new guarantee is referred to an ε-admissible. There are a number of ε-admissible algorithms.
3. Weighted A^*/Static Weighting: If $h_a(n)$ is an admissible heuristic function, in the weighted version of A^* search one uses

$$h_w(n) = \varepsilon h_a(n)$$

$$\varepsilon > 1$$

as a heuristic function (Pohl (1970)), and perform A^* search as usual, which eventually happens faster than using h_a, since fewer nodes are expanded. The path

then found by the search algorithm can have a cost of at most ε times that of the least cost path in the graph (Pearl (1984)).

4. Dynamic Weighting: Dynamic weighting (Pohl (1973)) uses the cost function

$$f(n) = g(n) + [1 + \varepsilon\omega(n)]h(n)$$

where

$$\omega(n) = \begin{cases} 1 - \dfrac{d(n)}{N} & d(n) \leq N \\ 0 & otherwise \end{cases}$$

and $d(n)$ is the depth of the search, and N is the anticipated length of the search.

5. Sampled Dynamic Weighting: This uses sampling of the nodes to better estimate and de-bias the heuristic (Koll and Kaindl (1992)).
6. A_ε^*: A_ε^* uses two heuristic functions (Pearl and Kim (1982)). The first is the FOCAL list, which is used to select the candidate nodes, and the second h_F is used to select the most promising node from the FOCAL list.
7. A_ε: $A_\varepsilon$selects nodes with the function $Af(n) + Bh_F(n)$ where A and B are constants (Ghallab and Allard (1983)). If no nodes can be selected, the algorithm will backtrack with the function $Cf(n) + Dh_F(n)$ where C and D are constants.
8. $AlphA^*$: $AlphA^*$ attempts to promote depth-first exploitation by preferring recently expanded nodes (Reese (1999)). $AlphA^*$ uses the cost function

$$f_\alpha(n) = [1 + \omega_\alpha(n)]f(n)$$

where

$$\omega_\alpha(n) = \begin{cases} \lambda & g\big(\pi(n)\big) \leq g(\tilde{n}) \\ \Lambda & otherwise \end{cases}$$

where λ and Λ are constants with

$$\lambda \leq \Lambda$$

and $\tilde{n}$ is the most recently expanded node.

Complexity

1. Worst-case: Unbounded Search Space: The time complexity of A^* depends on the heuristic. In the worst-case of a unbounded search space, the number of nodes expanded is exponential in the depth of the solution – the shortest path - d: $\mathcal{O}(b^d)$, where b is the branching factor, the average number of successors per state (Russell and Norvig (2018)). This assumes that a goal exists at all, and is reachable from the start state; if it is not, and the search space is infinite, the algorithm will not terminate.
2. Heuristics Induced Effective Branching Factor: The heuristic function has a major impact on the practical performance of the A^* search, since a good heuristic allows A^* to prune away many of the b^d nodes that an uninformed search would expand. Its quality can be expressed in terms of the effective branching factor b^*, which can be determined empirically for a problem instance by measuring the number N of the nodes expanded, and the depth of the solution, and then solving (Russell and Norvig (2018))

$$N = 1 + b^* + b^{*2} + \cdots + b^{*d}$$

Good heuristics are those with low effective branching factor, the optimal being

$$b^* = 1$$

3. Criteria for Polynomial Time Search: The time complexity is a polynomial when the search space is a tree, there is a single goal state, and the heuristic function h meets the following condition:

$$|h(x) - h^*(x)| = \mathcal{O}(\log h^*(x))$$

where $h^*(x)$ is the optimal heuristic, the exact cost to get from x to the goal. In other words, the error of $h(x)$ will not grow faster than the logarithm of the perfect heuristic $h^*(x)$ that returns the true distance from x to the goal (Pearl (1984), Russell and Norvig (2018)).

4. A^* Drawback – Space Complexity: The space complexity of A^* is the same as all other graph search algorithms, as it keeps all the generated nodes in memory (Russell and Norvig (2018)). In practice, this turns out to be the biggest drawback of the A^* search, leading to the development of memory-bound heuristic searches, such as Iterative Deepening A^*, memory-bounded A^*, and SMA^*.

Applications

A^* is often used for path-finding in applications such as video games, but was originally designed as a general graph traversal algorithm (Hart, Nilsson, and Raphael (1968)). It finds application in diverse problem, including the problem of parsing using stochastic grammars in NLP (Klein and Manning (2003)). Other cases include informational search with online learning (Kagan and Ben-Gal (2014)).

Relation to Other Algorithms

Some common variants of Dijkstra's algorithm can be viewed as a special case of A^* where the heuristic

$$h(x) = 0$$

for all nodes n (de Smith, Goodchild, and Longley (2007), Hetland (2010)); in turn, both Dijkstra and A^* are special cases of dynamic programming (Ferguson, Likhachev, and Stentz (2005)). A^* itself is a special case of generalization of branch-and-bound (Nau, Kumar, and Kanal (1984)).

Variants

a. Anytime A^* (Hansen and Zhou (2007) or Anytime Repairing A^* (ARA^*) (Likhachev, Gordon, and Thrun (2003))
b. Block A^*
c. D^*
d. Field D^*
e. Fringe
f. Fringe Saving A^* (FSA^*)
g. Generalized Adaptive A^* (GAA^*)
h. Incremental Heuristic Search
i. Informational Search (Kagan and Ben-Gal (2014))
j. Iterative Deepening A^* (IDA^*)
k. Jump-point Search
l. Lifelong Planning A^* (LPA^*)
m. Multiplier-accelerated A^* (MAA^*) (Li and Zimmerle (2019))
n. New Bidirectional A^* (NBA^*) (Pijls and Post (2010))
o. Simplified Memory Bounded A^* (SMA^*)
p. Real-time A^* (Korf (1990))
q. $Theta^*$
r. Time-bounded A^* (TBA^*) (Bjornsson, Bulitko, and Sturtevant (2009))

A^* can also be adapted to a bi-directional search algorithm. Special care needs to be taken for the stopping criterion.

References

- Bjornsson, Y., V. Bulitko, and N. Sturtevant (2009): TBA^*: Time-bounded A^* *Proceedings of the 21st International Joint Conference on Artificial Intelligence* 431-436
- de Smith, M. J., M. F. Goodchild, and P. Longley (2007): *Geo-spatial Analysis: A Comprehensive Guide to Principles, Techniques, and Software Tools* **Troubadour Publishing Limited**
- Dechter, R., and J. Pearl (1985): Generalized Best-first Search Strategies and the Optimality of A^* *Journal of the ACM* **32 (3)** 505-536
- Delling, D., P. Sanders, D. Schultes, and D. Wagner (2009): Engineering Route Planning Algorithms, in: *Algorithms of Large and Complex Networks: Design, Analysis, and Simulation* **Springer**
- Doran, J. E., and D. Michie (1966): Experiments with the Graph Traverser Algorithm *Proceedings of the Royal Society of London A* **294 (1437)** 235-259
- Edelkamp. S., S. Jabbar, and A. Lluch-Lafuente (2005): Cost Algebraic Heuristic Search *Proceedings of the 20th National Conference on Artificial Intelligence* 1362-1367
- Ferguson, D., M. Likhachev, and A. Stentz (2005): A Guide to Heuristic-based Path Planning
- Ghallab, M. and D. Allard (1983): A_ε – An Efficient near-admissible Heuristic Search Algorithm *Proceedings of the 8th Joint International Conference on Artificial Intelligence* 789-791
- Hansen, E. A., and R. Zhou (2007): Anytime Heuristic Search *Journal of Artificial Intelligence Research* **28** 267-297

- Hart, P. E., N. J. Nilsson, and B. Raphael (1968): A Formal Basis for the Heuristic Determination of the Minimum Cost Paths *IEEE Transactions on Systems Sciences and Cybernetics* **4 (2)** 100-107
- Hart, P. E., N. J. Nilsson, and B. Raphael (1972): A Correction to "A Formal Basis for the Heuristic Determination of the Minimum Cost Paths" *ACM SIGART Bulletin* **37** 28-29
- Hetland, M. L. (2010): *Python Algorithms: Mastering Basic Algorithms in the Python Language* **Apress**
- Kagan, E., and I. Ben-Gal (2014): A Group Testing Algorithm with Online Informational Learning *IIE Transactions* **46 (2)** 164-184
- Klein, D., and C. D. Manning (2003): A^* Parsing: Fast Exact Viterbi Parse Selection *Proceedings of the 2003 Human Language Technology Conference of the North America Chapter of the Association for Computational Linguistics* 119-126
- Koll, A., and H. Kaindl (1992): A New Approach to Dynamic Weighting *Proceedings of the 10th European Conference on Artificial Intelligence* 16-17
- Korf, R. E. (1990): Real-time Heuristic Search *Artificial Intelligence* **42 (2-3)** 189-211
- Li, J., and D. Zimmerle (2019): Designing Optimal Network for Rural Electrification using Multiplier-accelerated A^* Algorithm *IEEE PES Asia-Pacific Power and Energy Engineering Conference* 1-5
- Likhachev, M., G. Gordon, and S. Thrun (2003): ARA^*: Anytime A^* with Provable Bounds on Sub-optimality *Proceedings of the 16th International Conference on Neural Information Processing Systems* 767-774
- Martelli, A. (1977): On the Complexity of Admissible Search Algorithms *Artificial Intelligence* **8 (1)** 1-13
- Nau, D. S., V. Kumar, and L. Kanal (1984): General Branch-and-Bound, and its Connection to A^* and AO^* *Artificial Intelligence* **23 (1)** 29-58
- Nilsson, N. J. (2009): *The Quest for Artificial Intelligence* **Cambridge University Press**

- Pearl, J. and J. H. Kim (1982): Studies in Semi-admissible Heuristics *IEEE Transactions on Pattern Analysis and Machine Intelligence* **4 (4)** 392-399
- Pearl, J. (1984): *Heuristics: Intelligent Search Strategies for Computer Problem Solving* **Addison-Wesley**
- Pijls, W. H. L. M., and H. Post (2010): Yet another Bidirectional Algorithm for Shortest Paths
- Pohl, I. (1970): First Results in the Effect of Error in Heuristic Search *Machine Intelligence* **5** 219-236
- Pohl, I. (1973): The Avoidance of (Relative) Catastrophe, Heuristic Competence, Genuine Dynamic Weighting, and Computational Issues in Heuristic Problem-Solving *Proceedings of the 3rd Joint International Conference in Artificial Intelligence* 11-17
- Reese, B. (1999): *AlphA** - An ε-admissible Heuristic Search Algorithm
- Russell, S. J., and P. Norvig (2018): *Artificial Intelligence; A Modern Approach 4th Edition* **Pearson** Boston
- Wikipedia (2020): *A** Algorithm
- Zeng, W., and R. L. Church (2009): Finding Shortest Paths on Real Road Networks; The Case for *A* International Journal of Geographical Information Science* **23 (4)** 531-543

Floyd-Warshall Algorithm

Overview

1. Alternate Names for the Algorithm: Floyd-Warshall algorithm (also called Floyd's algorithm, Roy-Warshall algorithm, Roy-Floyd algorithm, or WFI algorithm) is an algorithm for finding shortest paths in a weighted graph with positive or negative edge weights, but with no negative cycles (Rosen (2003), Cormen, Leiserson, Rivest, and Stein (2009), Wikipedia (2020)).
2. All Pairs Shortest Path Trail: A single execution of the algorithm will find the lengths – summed weights – of shortest paths between all pairs of vertexes. Although it does not return details of the paths themselves, it is possible to reconstruct the paths with simple modifications to the algorithm.
3. Re-purposed Versions of the Algorithm: Versions of the algorithm can also be used for finding the transitive closure of a relation, or – in connection with the Schulze voting system – widest paths between all pairs of vertexes in a weighted graph.
4. Characteristics:

Class	All Pairs Shortest Path Problem (for Weighted Graphs)
Data Structure	Graph
Worst-case Performance	$\Theta(\|V\|^3)$
Best-case Performance	$\Theta(\|V\|^3)$
Average Performance	$\Theta(\|V\|^3)$
Worst-case Space Complexity	$\Theta(\|V\|^3)$

History and Naming

Floyd-Warshall algorithm is an example of dynamic programming, and was published in its currently recognized form by Floyd (1962). However, it is essentially the same as algorithms previously published by Roy (1959), and also by Warshall (1962) for finding a transitive closure of a graph, and is closely related to Kleene's algorithm for converting a deterministic finite automaton into a regular expression (Kleene (1956)). The modern formulation of the algorithm as three nested for-loops was first described by Ingerman (1962).

Algorithm

1. Pair-wise All Paths Comparison: The Floyd-Warshall algorithm compares all possible paths through the graph between each pair of vertexes. It is possible to do this with $\Theta(|V|^3)$ comparisons in a graph, even though there may be up to $\Omega(|V|^2)$ edges in the graph, and every combination of edges is tested. It does so by incrementally improving an estimate on the shortest path between two vertexes, until the estimate is optimal.
2. Pair Path using Intermediate Vertex: Consider a graph G with vertexes numbered 1 through N. Further, consider a function $shortestPath(i, j, k)$ that returns the shortest path from i to j using vertexes only from the set $\{1, 2, \cdots, k\}$ as intermediate points along the way. Now, given this function, the goal is to find the shortest path from each i to each j using *any* vertex in $\{1, 2, \cdots, N\}$.
3. Shortest Path through a Vertex: For each of these pairs of vertexes, the $shortestPath(i, j, k)$ could be either:
    a. a path that does not go through k, i.e., only uses vertexes in the set $\{1, 2, \cdots, k-1\}$, or:
    b. a path that goes through k, from i to k, and then from k to j, both only using intermediate vertexes in $\{1, 2, \cdots, k-1\}$.

4. Mechanism for Constructing a Path: Obviously, the best path from i to j that only uses vertexes 1 through $k-1$ is defined by $shortestPath(i,j,k-1)$, and it is clear that if there were a better path from i through k to j, then the length of this path would be the concatenation of the shortest path from i to k – only using intermediate vertexes in $\{1,2,\cdots,k-1\}$, and the shortest path from k to j – again, only using intermediate vertexes in $\{1,2,\cdots,k-1\}$.
5. Setting up the Recursion Scheme: If $w(i,j)$ is the weight of the edge between vertexes i and j, one can define $shortestPath(i,j,k)$ in terms of the following recursive formula; the base case is

$$shortestPath(i,j,0) = w(i,j)$$

and the recursive case is

$$\begin{aligned} &shortestPath(i,j,k) \\ &\qquad = \min\big(shortestPath(i,j,k-1), shortestPath(i,k,k-1) \\ &\qquad + shortestPath(k,j,k-1)\big) \end{aligned}$$

6. Progressive Scan through Intermediate Vertexes: The formula above is the heart of the Floyd-Warshall algorithm. The algorithm works by first computing $shortestPath(i,j,k)$ for all (i,j) pairs for

$$k = 1$$

then

$$k = 2$$

and so on. This process continues until

$$k = N$$

at which stage the shortest paths for all (i, j) pairs will have been found using any intermediate vertexes.

Behavior with Negative Cycles

1. Algorithm Failure Under Negative Cycles: A negative cycle is a cycle whose edges sum to a negative value. There is no shortest path between any pair of vertexes (i, j) which form part of a negative cycle, because path lengths from i to j can be arbitrarily small, i.e., negative. For numerically meaningful output, the Floyd-Warshall algorithm assumes that there are no negative cycles. Nevertheless, if there are negative cycles, the Floyd-Warshall algorithm can be used to detect them.
2. Scheme for Detecting Negative Cycles:
    a. The Floyd-Warshall algorithm iteratively revises the path lengths between all pairs of vertexes (i, j) including where

$$i = j$$

    b. Initially, the length of the path is (i, i) is zero
    c. A path $[i, k, \cdots, i]$ can only improve upon this if it has length less than zero, i.e., denotes a negative cycle.
    d. Thus, after the algorithm, (i, i) will be negative if there exists a negative length path from i back to i
3. Examining the Floyd-Warshall Path Matrix: Hence, to detect the negative cycles using the Floyd-Warshall algorithm, one can inspect the diagonal of the path matrix, and the presence of a negative number indicates that the graph contains at least one negative cycle. To avoid numerical problems, one should check for negative numbers on the diagonal of the path matrix within the inner *for-loop* of the algorithm

(Hougardy (2010)). Obviously, in an undirected graph, a negative edge creates a negative cycle, i.e., a close walk, involving its incident edges.

Path Reconstruction

The Floyd-Warshall algorithm typically only provides the lengths of the paths between all pairs of vertexes. With simple modifications, it is possible to create a method to reconstruct the actual path between any two vertex endpoints. While one may be inclined to store the actual path from each vertex to each other vertex, this is not necessary, and in fact, is very costly in terms of memory. Instead, the shortest path tree can be calculated for each node in $\Theta(|E|)$ time using $\Theta(|V|)$ memory to store each tree which allows efficient re-construction of a path between any two connected vertexes.

Analysis

Let

$$n = |V|$$

be the number of vertexes. To find all n^2 of $shortestPath(i, j, k)$ for all i and j from those of $shortestPath(i, j, k-1)$ requires $2n^2$ operations. Since one begins with

$$shortestPath(i, j, 0) = w(i, j)$$

and computes the sequence of n matrices $shortestPath(i, j, 1), shortestPath(i, j, 2), \cdots, shortestPath(i, j, n)$, the total number of operations used is

$$n \,.\, 2n^2 = 2n^3$$

The complexity of the algorithm is therefore $\Theta(n^3)$.

Applications and Generalizations

1. Shortest Paths in Directed Graphs: Floyd's algorithm.
2. Transitive Closure of Directed Graphs: This is the Warshall algorithm. In the original formulation, the graph is unweighted and represented by a Boolean adjacency matrix. Then the addition operation is replaced by logical conjunction (AND) and the minimum operation by logical disjunction (OR).
3. Regular Language for Finite Automaton: Finding a regular expression denoting the regular language accepted by a finite automaton – this is Kleene's algorithm, a closely related generalization of the Floyd-Warshall algorithm (Gross and Yellen (2003)).
4. Inversion of Real Matrices: This is the Gauss-Jordan algorithm (Penaloza (2005)).
5. Optimal Routing: In this application, one is interested in finding the path with the maximum flow between two vertexes. This means that, instead of taking the minima in the algorithm above, one instead takes the maxima. The edge weights represent fixed constraints on the flow. Path weights represent bottlenecks; so, the addition operation above is replaced by a minimum operation.
6. Fast Computation of Path Finder Networks.
7. Widest Paths/Maximum Bandwidth Paths.
8. Difference Bound Matrices: Computing canonical form of difference bound matrices (DBM's)
9. Computing the Similarity between Graphs.

Comparisons with other Shortest Path Algorithms

1. Sparse vs. Dense Graph Problems: The Floyd-Warshall algorithm is a good choice for computing paths between all pairs of vertexes in dense graphs, in which most or all pairs of vertexes are connected by edges. For sparse graphs, with non-negative weights, the better choice is to use Dijkstra's algorithm for each possible starting vertex, since the running time for repeated Dijkstra ($\mathcal{O}(|E||V| + |V|^2 \log|V|)$ using Fibonacci heaps) is better than the $\mathcal{O}(|V|^3)$ running time of the Floyd-Warshall algorithm when $|E|$ is significantly smaller than $|V|^2$. For sparse graphs with negative edges, but no negative cycles, Johnson's algorithm can be used, with the same asymptotic running time as the repeated Dijkstra approach.
2. Custom Algorithms for Special Situations: There are also known algorithms using fast-matrix multiplication to speed up all-pairs shortest path computation on dense graphs, but these typically make extra assumptions on the edge weights, such as requiring them to be small integers (Zwick (2002), Chan (2010)). In addition, because of the high constant factors in their running time, they would only provide a speed-up over the Floyd-Warshall algorithm for very large graphs.

References

- Chan, T. M. (2010): More Algorithms for All-Pairs Shortest Paths in Weighted Graphs *SIAM Journal on Computing* **39 (5)** 2075-2089
- Cormen, T., C. E. Leiserson, R. Rivest, and C. Stein (2009): *Introduction to Algorithms 3rd Edition* **MIT Press**
- Floyd, R. W. (1962): Algorithm 97: Shortest Path *Communications of the ACM* **5 (6)** 345
- Gross, J. L., and J. Yellen (2003): *Handbook of Graph Theory* **CRC Press**
- Hougardy, S. (2010): The Floyd-Warshall Algorithm on Graphs with Negative Cycles *Information Processing Letters* **110 (8-9)** 279-291
- Ingerman, P. Z. (1962): Algorithm 141: Path Matrix *Journal of the ACM* **5 (11)** 556

- Kleene, S. C. (1956): Representation of Events in Nerve Nets and Finite Automata, in: *Automata Studies, Shannon, C. E., and J. McCarthy, editors* **Princeton University Press** 3-42
- Penaloza, R. (2005): Algebraic Structures for Transitive Closure
- Rosen, K. H. (2003): *Discrete Mathematics and its Applications 5th Edition* **Addison Wesley**
- Roy, B. (1959): Transitivite et Connexite *Comptes Rendus de l'Academie des Sciences* **249** 216-218
- Warshall, S. (1962): A Theorem on Boolean Matrices *Journal of the ACM* **9 (1)** 11-12
- Wikipedia (2020): Floyd-Warshall Algorithm
- Zwick, U. (2002): All-Pairs Shortest Path using Bridging Sets and Rectangular Matrix Multiplication *Journal of the ACM* **49 (3)** 289-317

Strongly Connected Component

Overview

1. Definition of Strongly Connected Components: In the mathematical theory of connected graphs, a graph is said to be *strongly connected* if every vertex is reachable from every other vertex (Wikipedia (2020)).
2. SCC's of a Directed Graph: The *strongly connected components* (SCC) of an arbitrary directed graph form a partition of subgraphs that are themselves strongly connected.
3. Testing Strong Connectivity in Graphs: It is possible to test the strong connectivity of a graph, or to find its strongly connected components, in linear time, i.e., $\mathcal{O}(|E| + |V|)$.

Definitions

1. Round-trip Path between Nodes: A directed graph is called *strongly connected* if there is a path in each direction between each pair of vertexes of the graph. That is, a path exists from the first vertex in the pair to the second, and another path exists from the second vertex to the first.
2. Strongly Connected Nodes in a Graph: In a directed graph G that may itself not be strongly connected, a pair of vertexes u and v is said to be strongly connected to each other if there is a path in each direction between them.
3. Strongly Connectedness as a Binary Relation: The binary relation of being strongly connected is an equivalence relation, and the induced subgraphs of its equivalence classes are called *strongly connected components*. Equivalently, a *strongly connected component* of a directed graph G is a subgraph that is strongly connected, and is

maximal with this property; no additional edges or vertexes can be included in the subgraph without breaking its property of being strongly connected. The collection of strongly connected components forms a partition of the set of vertexes of G.

4. SCC Contraction into a DAG: If each strongly connected component is contracted to a single vertex, the resulting graph is a directed acyclic graph, a *condensation* of G.
5. SCC Based Definition of DAG: A directed graph is acyclic if any only if it has no strongly connected subgraphs with more than one vertex, because a directed graph is strongly connected and every non-trivial strongly connected component contains at least one directed cycle.

DFS Based Linear Time Algorithms

1. Linear-Time Algorithms using DFS: Several algorithms based on DFS compute strongly connected components in linear time.
2. Kosaraju's Algorithm - First DFS Pass: Kosaraju's algorithm uses two passes of DFS. The first, in the original graph, is used to choose the order in which the outer loop of the second DFS tests vertexes for having been visited already and recursively explores them if not.
3. Kosaraju's Algorithm - Second DFS Pass: The second DFS is on the transpose of the original graph, and each recursive exploration finds a single new strongly connected component (Cormen, Leiserson, Rivest, and Stein (2009), Hong, Rodia, and Olukotun (2013)).
4. Tarjan's Strongly Connected Components Algorithm: Tarjan's (1972) strongly connected components' algorithm performs a single pass of DFS. It maintains a stack of vertexes that have been explored by the search but not yet assigned to a component, and calculates *low numbers* of each vertex – an index number of the highest ancestor reachable in one step from a descendant of the vertex – which it uses to determine when a set of vertexes should be popped off the stack in a new component.

5. Path Based Strong Component Algorithm: The path-based strong component algorithm uses DFS, like Tarjan's algorithm, but with two stacks. One of the stacks is used to keep track of the vertexes not yet assigned to components, while the other keeps track of the current path in the DFS. The linear time version of this algorithm was published by Dijkstra (1976).
6. Comparison of the above Algorithms: Although Kosaraju's algorithm is conceptually simple, Tarjan's algorithm and path-based algorithm require only one DFS rather than two.

Reachability-Based Algorithms

1. Parallelization Provided by Reachability Algorithms: The linear time algorithms shown above are based on DFS, which is generally considered hard to parallelize. Fleischer, Hendrickson, and Pinar (2000) propose a divide-and-conquer approach based on reachability queries, and such algorithms are usually called reachability-based SCC algorithms.
2. Reachability Queries Off of Random Pivots: The idea of this approach is to pick a random pivot vertex and apply forward and backward reachability queries from this vertex. The two queries can partition the vertex into 4 subsets; vertexes reach by both, wither one, or none of the searches.
3. Extracting SCC's from the Subsets: One can show that a strongly connected component has to be contained in one of the subsets. The vertex subset reached by both searches forms a strongly connected component and the algorithm then recurses on the other 3 searches.
4. Expected Running Time of the Algorithm: The expected running time of this algorithm can be shown to be $\mathcal{O}(n \log n)$ - a factor $\mathcal{O}(\log n)$ more than the classical algorithms.
5. Origin of the Parallelism: The parallelism comes from:

   a. the reachability queries can be parallelized more easily, e.g., by a DFS, and it can be fast of the diameter of the graph is small, AND
   b. the independence between the sub-tasks in the divide-and-conquer process.
6. Lack of Theoretical Parallelism Guarantees: This algorithm performs well on real-world graphs (Hong, Rodia, and Olukotun (2013)), but does not have theoretical guarantees on the parallelism, i.e., if a graph has no edges, the algorithm requires $\mathcal{O}(n)$ levels of recursions.
7. Randomly Applying the Reachability Queries: Blelloch, Gu, Shun, and Sun (2016) show that if the reachability queries are applied in a random order, the cost bound of $\mathcal{O}(n \log n)$ still holds. Furthermore, the queries can be batched in a prefix-doubling manner, i.e., 1, 2, 4, 8 queries, and tun simultaneously in one round. The overall span of this algorithm is $\log_2 n$ reachability queries, which is probably the optimal parallelism that can be achieved using the reachability-based approach.

Generating Random Strongly Connected Components

1. Algorithm for Generating Random SCC's: Maurer (2017) describes an algorithm for generating random strongly connected graphs, based on a modification of the Tarjan's algorithm to create a spanning tree and adding the minimum number of edges such that the result becomes strongly connected.
2. Customization of the SCC Structures: When used in conjunction with Gilbert or Erdos-Renyi models with node relabeling, the algorithm is capable of generating any strongly connected graph on n nodes, without restrictions on the kinds of structures that can be generated.

Applications

1. Solving Systems of 2-satisfiability Constraints: Algorithms for finding strongly connected components may be used to solve 2-satisfiability problems, i.e., systems of Boolean variables with constraints on the values of pairs of variables, as shown by Aspvall, Plass, and Tarjan (1979). A 2-satisfiability is unsatisfiable if and only if there is a variable v such that v and its complement are contained in the same strongly connected component of the implication graph of the instance.
2. Classifying Edges of Bipartite Graphs: Strongly connected components are also used to compute the Dulmage-Mendelsohn decomposition, a classification of the edges of a bipartite graph, according to whether or not they can be part of a project matching in the graph (Dulmage and Mendelsohn (1958)).

Related Results

1. Concept of the Ear Decomposition: A directed graph is strongly connected if and only if it has an ear decomposition, a partition of the edges into a sequence of directed paths and cycles such that the first subgraph in the sequence is a cycle, and each subsequent subgraph is either a cycle sharing one vertex with the previous subgraphs, or a path subgraphs its two end points with previous subgraphs.
2. Ear Decomposition on Undirected Graphs: According to Robbins' theorem, an undirected graph may be oriented in such a way that it becomes strongly connected, if and only if it is 2-edge-connected. One way to prove this is result is to find an ear decomposition of the underlying undirected graph and then orient each ear consistently (Robbins (1939)).

References

- Aspvall, B., M. F. Plass, and R. E. Tarjan (1979): A Linear-time Algorithm for Testing the Truth of certain Quantified Boolean Formulas *Information Processing Letters* **8 (3)** 121-123
- Blelloch, G. E., Y. Gu, J. Shun, and Y. Sun (2016): Parallelism in Randomized Incremental Algorithms *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures* 467-478
- Cormen, T., C. E. Leiserson, R. Rivest, and C. Stein (2009): *Introduction to Algorithms 3rd Edition* **MIT Press**
- Dijkstra, E. (1976): *A Discipline of Programming* **Prentice Hall** NJ
- Dulmage, A. L., and N. S. Mendelsohn (1958): Coverings of Bipartite Graph *Canadian Journal of Mathematics* **10** 517-534
- Fleischer, L. K., B. Hendrickson, and A. Pinar (2000): On Identifying Strongly Connected Components in Parallel
- Hong, S., N. C. Rodia, and K. Olukotun (2013): On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-world Graphs *SC '13: Proceedings of the International Conference on High-performance Computing, Networking, Storage, and Analysis* 1-11
- Maurer, P. M. (2017): Generating Strongly Connected Random Graphs *International Conference on Modeling, Simulation, and Visualization Methods*
- Tarjan, R. E. (1972): Depth-first Search and Linear Graph Algorithms *SIAM Journal on Computing* **1 (2)** 146-160
- Robbins, H. E. (1939): A Theorem on Graphs, with an Application to a Problem on Traffic Control *American Mathematical Monthly* **46 (5)** 281-283
- Wikipedia (2020): Strongly Connected Component

Kosaraju's Algorithm

Overview

1. Focus of the Kosaraju-Sharir Algorithm: *Kosaraju's algorithm* – also known as the *Kosaraju-Sharir algorithm* – is a linear time algorithm to find the strongly connected components of a directed graph (Sharir (1981), Aho, Hopcroft, and Ullman (1983), Wikipedia (2020)).
2. SCC Property used by the Algorithm: The algorithm makes use of the fact that the transpose graph, i.e., the same graph with the direction of every edge reversed, has exactly the same strongly connected components as the original graph.

The Algorithm

1. Graph Operations Used: The primitive graph operations that the algorithm uses are to enumerate the vertexes of the graph, to store data per vertex – if not in the graph data structure itself, then in some table that can use the vertexes as indexes, to enumerate the out-neighbors of a vertex, i.e., traverse edges in the forward direction, and to enumerate the in-neighbors of the vertex, i.e., traverse edges in the backward direction. However, the last step can be done away with, at the cost of constructing a representation of the transpose graph during the forward traversal phase.
2. Extra Data Structures used by the Algorithm: The only additional data structure needed by the algorithm is an ordered list L of the graph vertexes, that will grow to contain each vertex once.
3. SCC's Root Vertex: If strong components are to be represented by appointing a separate root vertex for each component, and assigning to each vertex the root vertex of each component, then Kosaraju's algorithm can be stated as follows.

4. Step #1 - Initializing u and L: For each vertex u of the graph, mark u as unvisited. Let L be empty.
5. Step #2 - u Visitation and L Update: For each vertex u of the graph, do $Visit\ (u)$ where $Visit\ (u)$ is the following recursive subroutine:
    a. If u is unvisited, then:
        i. Mark u as visited.
        ii. For each out-neighbor v of u, do $Visit\ (v)$
        iii. Prepend u to L.
    b. Otherwise do nothing.
6. Step #3 – Root Vertex Assignment: For each element u of L in order, do $Assign\ (u,u)$ where $Assign\ (u,root)$ is the following recursive subroutine.
    a. If u has not been assigned to a component, then:
        i. Assign u as belonging to the component whose root is $root$
        ii. For each in-neighbor v of u, do $Assign\ (v,root)$
    b. Otherwise, do nothing.
7. Alternative Variations of the Algorithms: Trivial variations are to instead assign a component number to each vertex, or to construct per-component lists of vertexes that belong to it. The visited/unvisited indication may share storage location with the final assignment of the root of a vertex.
8. Search Tree Post-Order Prepending: The key point of the algorithm is that during the first, i.e., the forward, traversal of the graph edges, the vertexes are prepended to the list L in post-order relative to the search tree being explored.
9. Identification of the Forward Path: This means that it does not matter whether vertex v was first visited because it appeared in the enumeration of all vertexes or because it was the out-neighbor of another vertex u that got visited; either way, v will be prepended to L before u is, so if there is a forward path from u to v then v will appear before u on the final list L, unless u and v both belong to the same strong component; in which case their relative order in L is arbitrary.

10. DFS vs BFS for Graph Navigation: As given above, the algorithm employs DFS for simplicity, but it could just as well use BFS as long as the post-order property is preserved.
11. Reachability by Forwards/Backwards Traversal: The algorithm can be understood as identifying the strong component of a vertex u as the set of vertexes which are reachable from u by both forwards and backwards traversal. Writing $F(u)$ for the set of vertexes reachable from u by forward traversal, $B(u)$ for the set of vertexes reachable from u by backwards traversal, and $P(u)$ for the set of vertexes which appear strictly before u on the list L on the phase 2 of the algorithm, the string component containing a vertex u appointed as root is

$$B(u) \cap F(u) = B(u) \backslash [B(u) \backslash F(u)] = B(u) \backslash P(u)$$

12. Optimization achieved using Ordered List: Set introspection is computationally costly, but it is logically equivalent to a double set difference, and since

$$B(u) \backslash F(u) \subseteq P(u)$$

it becomes sufficient to test whether a newly encountered element of $B(u)$ has already been assigned to a component or not.

Complexity

1. Linear Running Time in $|V|$ and $|E|$: Provided the graph is described using a adjacency list, Kosaraju's algorithm performs two complete traversals of the graph and so runs in $\mathcal{O}(|V| + |E|)$, i.e., linear time, which is asymptotically optimal because there is a matching lower bound, which is that any algorithm must examine all vertexes and edges.

2. Comparison to Tarjan's and Path-based Algorithms: Kosaraju's algorithm is conceptually the simplest efficient algorithm, but it is not as efficient in practice as Tarjan's strongly connected components algorithm and the path-based string component algorithm, which perform only one traversal of the graph.
3. Adjacency Matrix Graph Representation: If the graph is represented as an adjacency matrix, the algorithm requires $\mathcal{O}(|V|^2)$ time.

References

- Aho, A. V., J. E. Hopcroft, and J. D. Ullman (1983): *Data Structures and Algorithms* **Addison Wesley**
- Sharir, M. (1981): A Strong-connectivity Algorithm and its Applications to Data Flow Analysis *Computer and Mathematics with Applications* **7 (1)** 67-72
- Wikipedia (2020): Kosaraju's Algorithm

Selection Algorithm

Overview

1. k^{th} Order Statistic Selection Algorithm: A selection algorithm is an algorithm for finding the k^{th} smallest number in a list or an array; such a number is called the k^{th} *order statistic*. This includes the cases of finding the minimum, the maximum, and the median elements (Wikipedia (2019)).
2. Linear/Sublinear Worst-case Time: There are $\mathcal{O}(n)$ time, i.e., worst-case linear time, selection algorithms, and sublinear performance is possible for structured data; in the extreme, $\mathcal{O}(1)$ for an array of sorted data.
3. Super-problems Employing Selection Algorithms: Selection is a sub-problem of more complex problems like the nearest neighbor and the shortest path problems.
4. Selection as Generalization of Sorting: Many selection algorithms are derived by generalizing a sorting algorithm, and conversely some sorting algorithms can be derived as repeated applications of selection.
5. Minimum/Maximum Finding Selection Algorithms: The simplest case of a selection algorithm is finding a minimum – or maximum – element by iterating through the list, keeping track of the running minimum/maximum – the minimum/maximum so far – and can be seen as being related to the selection sort.
6. Selection Algorithms for Median Finding: Conversely, the hardest case of a selection algorithm is finding the median. In fact, a specialized median selection algorithm can be used to build a general selection algorithm, as in median of medians.
7. Quickselect - Average/Worst-case Performance: The best-known selection algorithm is quickselect, which is related to quicksort; like quicksort, it has asymptotically optimal average performance, but poor worst-case performance, though it can be modified to give optimal worst-case performance as well.

Selection by Sorting

1. Selection Algorithm using Full Sorting: By sorting the list or array and the selecting the desired element, selection can be reduced to sorting. This method is inefficient for selecting a single element, but is efficient when many selections need to be made from an array, in which case only one initial, expensive sort is needed, followed by many cheap selection operations - $\mathcal{O}(1)$ for an array, though selection is $\mathcal{O}(n)$ in a linked list, even if sorted, due to lack of random access. In general, sorting requires $\mathcal{O}(n \log n)$ time, where n is the length of the list, although a lower bound is possible with non-comparative sorting algorithms like radix sort and counting sort.
2. Selection from Partially Sorted List: Rather than sorting the whole list or array, one can use instead partial sorting to select the k largest or k smallest elements. The k^{th} smallest – resp. the k^{th} largest – element is then the largest – resp., smallest – element of the partially sorted list; this then takes $\mathcal{O}(1)$ to access in an array and $\mathcal{O}(k)$ to access in a list.

Unordered Partial Sorting

1. Unordered Partially Sorted Element List: If partial sorting is relaxed so that the k smallest elements are returned, but not in order, the factor of $\mathcal{O}(k \log k)$ can be eliminated. An additional maximum selection taking $\mathcal{O}(k)$ time is required, but since

$$k \leq n$$

this still yields an asymptotic complexity of $\mathcal{O}(n)$. In fact, partition-based selection algorithms yield both the k^{th} smallest element itself and the k smallest elements, with the other elements not in order. This can be done in $\mathcal{O}(n)$ time – average complexity of quickselect, and the worst-case complexity of refined partition-based algorithms.

2. Selection Induced Unordered Partial List: Conversely, given a selection algorithm, one can easily get an unordered partial sort, i.e., k smallest elements, not in order – in $\mathcal{O}(n)$ time by iterating through the list and recording all the elements less than the k^{th} element. If this results in fewer than $k-1$ elements, any remaining elements equal the k^{th} element. Care must be taken, due to the possibility of equality of the elements; one must not include all elements less than *or equal to* the k^{th} element, as elements greater than the k^{th} element may also be equal to it.
3. Selection vs. Unordered Partial Sorting: Thus, unordered partial sorting – lowest k elements, but not ordered – and selection of the k^{th} element are very similar problems. Not only do they have the same asymptotic complexity $\mathcal{O}(n)$ but a solution to either one can be converted into a solution to the other by a straightforward algorithm, e.g., finding a maximum of k elements, of filtering elements of a list below the cutoff of the value of the k^{th} element.

Partial Selection Sort

1. Minimum/Maximum Partial Selection Sort: A simple example of selecting by partial sorting is to use a partial selection sort. The obvious linear time algorithm to find the minimum – resp. maximum – iterating over the list and keeping track of the minimum – resp. maximum – element so far can be seen as a partial selection sort that selects the 1 smallest – resp. largest – element. However, many other partial sorts also reduce this algorithm for the case $k-1$, such as partial heap sort.
2. Partial Sorting Solution - Trivial Implementation: More generally, a partial selection sort yields a simple selection algorithm which takes $\mathcal{O}(kn)$ time. This is asymptotically inefficient, but can be sufficiently if k is small, and is easy to implement. Specifically, one simply finds the extremum value and moves it to the beginning, repeating on the remaining lists until k elements have been accumulated, and then return the k^{th} element.

Partition-Based Selection

1. Comparison between Quickselect and Quicksort: Linear performance can be achieved by a partition-based selection algorithm, most basically quickselect. Quickselect is a variant of quicksort – in both one chooses a pivot and partitions the data by it, but while quicksort recurses on both sides of the partition, quickselect only recurses on one side, namely the side on which the desired k^{th} element is.
2. Time Complexity of Quickselect Algorithm: As with quicksort, this has optimal performance, in this case linear, but poor worst-case performance, in this case quadratic. This occurs, for instance, when taking the first element as the pivot and searching for the maximum element, if the data is already sorted. In practice, this can be avoided by choosing a random element as the pivot, which yields almost certain linear performance. Alternatively, a more careful deterministic pivot strategy can be used, such as median of medians.
3. Optimization by using the Hybrid Introselect Algorithm: These are combined in the hybrid introselect algorithm, analogous to introsort, which starts with quickselect, but falls back to median of medians if the progress is slow, resulting in both fast average performance and average worst-case performance of $\mathcal{O}(n)$.
4. Space Complexity of the Algorithm: The partition-based algorithms are generally done in place, which thus results in partially sorting the data. They can be done out of place, not changing the original data, at the cost of $\mathcal{O}(n)$ additional space.

Median Selection as Pivot Strategy

1. Using Median as the Pivot: A median-selection algorithm can be used to yield a general solution or sorting algorithm, by applying it as the pivot strategy in quickselect or quicksort; if the median-selection algorithm is asymptotically optimal, i.e., linear time, the resulting selection or sorting algorithm is as well. In fact, exact

median is not necessary – an approximate median is sufficient. In the median of medians selection algorithm, the pivot strategy computes an approximate median and uses this as the pivot, recursing on a smaller set to compute this pivot. In practice, the overhead of the pivot computation is significant, so these algorithms are generally not used, but this technique is of theoretical interest in related selection and sorting algorithms.

2. Optimization achieved using Median Pivot: Specifically, given a median-selection algorithm, one can use it as a pivot strategy in quickselect, thereby obtaining a selection algorithm. If the median-selection algorithm is optimal, meaning $\mathcal{O}(n)$, then the resulting general selection algorithm is also optimal, again meaning linear. This is because quicksort is a divide-and-conquer algorithm, and using a median at each pivot means that at each step the search set decreases by half in size, so the overall complexity is a generic series times the time complexity of a single step, in fact

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

times, summing the whole series.

3. Median Pivot Based Sorting Algorithm: Similarly, given a median-selection algorithm or general selection algorithm applied to find the median, one can use it as a pivot strategy in quicksort, obtaining a sorting algorithm. If the selection algorithm is optimal, meaning $\mathcal{O}(n)$, then the resulting sorting algorithm is optimal, meaning $\mathcal{O}(n \log n)$. The median is the best pivot for sorting, as it evenly divides the data, and thus guarantees optimal sorting, assuming the selection algorithm is optimal. A sorting analog to median-of-medians exists, using the pivot strategy, i.e., approximate median, in quicksort, and similarly yields an optimal quicksort.

Incremental Sorting by Selection

1. Sorting by Repeated Selection: Converse to selection by sorting, one can incrementally sort by repeated selection. Abstractly, selection only yields a single element – the k^{th} element. However, practical selection algorithms frequently involve partial sorting, or can be modified to do so.
2. Benefits Gained by Partial Sorting: Selecting by partial sorting naturally sorts the array, sorting the elements up to k, and selecting by partitioning also sorts some elements; the pivots are sorted to their correct positions, with the k^{th} element being the final pivot, and the elements between the pivots have values between the pivot values.
3. Partition Based on Selection vs. Sorting: The difference between partition-based selection and partition-based sorting, as in quickselect versus quicksort, is that in selection one recurses on only one side of each pivot, sorting only on pivots – on average $\log n$ pivots are used – rather than recursing on both sides of the pivot.
4. Partial Selection Amortizes Sorting Costs: This can be used to speed up subsequent selections on the same data; in the extreme, a fully sorted array allows for $\mathcal{O}(1)$ selection. Further, compared with doing a full sort, incrementally sorting by repeated selection amortizes the cost over multiple selections.
5. Leveraging Previously Sorted Data Ranges: For partially sorted data – up to k – so long the partially sorted data and the index k up to which the data is sorted are recorded, subsequent selections of j less than k can simply select the j^{th} element, as it is already sorted, while selections of j greater than k only need to sort elements above the k^{th} position.
6. Benefits of Storing Pivot Lists: For partitioned data, if the list of pivots is stored – for example, in a sorted list of indexes – then subsequent selections only need to select the index between the two pivots – the nearest pivots below and above. The biggest gain is from the top-level pivots, which eliminate costly large partitions; a single pivot near the middle of the data cuts the time for future selections in half. The pivot list will grow over subsequent selections, as the data becomes more sorted, and can even be passed to a partition-based sort as the basis of a full sort.

Using Data Structures to Select in Sublinear Time

1. Organized Data Sublinear Access Times: Given an unorganized list of data, linear time $\Omega(n)$ is needed to find the minimum element, because every element has to be examined. If the list is organized, for example, by keeping it sorted at all times, then selecting the k^{th} largest element is trivial, but then insertion requires linear time, as do other operations such as combining two lists.
2. Tree Structures and Frequency Tables: The strategy to find an order statistic in sublinear time is to store the data in an organized fashion using suitable data structures that facilitate the selection. Two such data structures are tree-based structures and frequency tables.
3. Heap Structure Operations Time Requirements: When only the minimum or the maximum is needed, a good approach is to use a heap, which is able to find the minimum or the maximum element in constant time, while all other operations, including insertion, are $\mathcal{O}(\log n)$ or better.
4. Self-Balancing Binary Search Tree: More generally, a self-balancing binary search tree can easily be augmented to make it possible to both insert an element and find the k^{th} largest element in $\mathcal{O}(\log n)$ time; this is called an *order statistic tree*. Each node simply stores a count of how many descendants it has, and uses this to determine which path to follow. The information can be updated efficiently since adding a node only affects the counts of its k^{th} largest element in $\mathcal{O}(\log n)$ ancestors, and the tree rotation only affects the count of nodes involved in the rotation.
5. Data Range-aware Hashtable: Another simple strategy is based on some of the same concepts as the hashtable. When the range of values is known beforehand, one can divide the range into h sub-intervals and assign these to h buckets. When an element is inserted, it is added to the bucket corresponding to the interval it falls in.
6. Locating the k^{th} Order Statistic: To find the minimum or the maximum element, one scans from the beginning or the end looking for the first non-empty bucket and find the minimum of the maximum element in the bucket. In general, to find the k^{th}

element, one maintains a count of the number of elements in each bucket, then scans the buckets from left to right adding up the counts until the bucket containing the desired element is found, then uses the expected linear-time algorithm to find the correct element in that bucket.

7. Maintenance Overhead with Hashtables: If one chooses h of roughly size $\sqrt{n}$, and the input is close to being uniformly distributed, this scheme can perform selections in expected $\mathcal{O}(\sqrt{n})$ time. Unfortunately, this strategy is also sensitive to clustering of elements in a narrow interval, which may result in buckets with large numbers of elements; clustering can be eliminated through a good hash function, but finding the element with the k^{th} has value is not very useful. Additionally, like hash-tables, this structure requires table re-sizings to maintain efficiency as elements are added and n becomes much larger than h^2.
8. Frequency Tables for Descriptive Statistics: A useful case of this is finding an order statistic in a finite range of data. Using the above table with bucket interval of 1 and maintaining counts in each of the buckets is much superior to other methods. Such hashtables are like frequency tables used to classify data in descriptive statistics.

Lower Bounds

1. Lower Bound on the Comparisons: Knuth (1997) discusses a number of lower bounds for the number of comparisons required to locate the t smallest entries of an unorganized list of n items, using only comparisons. There is a trivial lower bound of $n - 1$ for the maximum or the minimum entry.
2. Comparisons Lower Bound - Explicit Expression: The story becomes more complex for other indexes. Defining $W_t(n)$ as the minimum number of comparisons required to find the t smallest values, Knuth (1997) references S. S. Kislitsyn to show an upper bound on this value:

$$W_t(n) \leq n - t + \sum_{n+t-1<j\leq n} \lceil \log_2 j \rceil$$

for

$$n \geq t$$

the bound is achievable for

$$t = 2$$

but better, more complex bounds are known for larger t.

Space Complexity

The required space complexity of selection is $\mathcal{O}(1)$ additional storage, in addition to storing the array in which the selection is being performed. Such space complexity can be achieved while preserving optimal $\mathcal{O}(n)$ time complexity.

Online Selection Algorithm

1. k^{th} Extremum of a Stream: Online selection may refer narrowly to computing the k^{th} smallest element of a stream, inn which case the partial sorting algorithms – with $k + \mathcal{O}(1)$ space for the k smallest elements so far – can be used, but partition-based algorithms cannot be.
2. Probabilistic Selection of k^{th} Extremum: Alternatively, selection itself may be required to be online, that is, an element can only be selected from a sequential input at the instant of observation, and each selection – or refusal – is irrevocable. The

problem is to select, under these constraints, a specific element of the input sequence – for example, the largest or the smallest value – with the largest probability. This problem can be tackled by the Odds algorithm, which yields an optimal selection under an independence condition; it is also optimal itself as an algorithm with the number of computations being linear in the length of the output.

3. Secretary Problem - Probability Maximizing Selection: The simplest example is the secretary problem of choosing the maximum with high probability, in which case the optimal strategy on random data is to track the running maximum on the first $\frac{n}{e}$ elements and reject them, and then select the first element that is higher than this maximum.

Related Problems

One may generalize the selection problem to apply to ranges within a list, yielding the problem of range queries. The question of range median queries – computing the medians of multiple ranges – has also been analyzed.

References

- Knuth, D. (1997): *The Art of Computer Programming 3rd Edition* **Addison-Wesley**
- Wikipedia (2019): Selection Algorithm

Quickselect

Overview

1. k^{th} Order Statistic Selection Algorithm: *Quickselect* is a selection algorithm to find the k^{th} smallest element in an unordered list. It is related to the quicksort sorting algorithm; like quicksort, it was developed by Hoare (1961), and thus is also known as *Hoare's selection algorithm* (Wikipedia (2019)). Like quicksort, it is efficient in practice and has good average-case performance, but has poor worst-case performance. Quickselect and its variants are the selection algorithms most often used in efficient real-world implementations.
2. Relation to the Quicksort Algorithm: Quickselect uses the same overall approach as quicksort, choosing one element as the pivot and partitioning the data into two based on the pivot, accordingly as less than or greater than the pivot. However, instead of recursing in both sides, as in quicksort, quickselect only recurses on one side – the side with the element it is searching for. This reduces the average complexity from $\mathcal{O}(n \log n)$ to $\mathcal{O}(n)$, with a worst-case of $\mathcal{O}(n^2)$.
3. In-place Partially Soring Algorithm: As with quicksort, quickselect is generally implemented as an in-place algorithm, and beyond selecting the k^{th} element, it also partially sorts the data.
4. Quickselect Characteristics:

Class	Selection Algorithm
Data Structure	Array
Worst-case Performance	$\mathcal{O}(n^2)$
Best-case Performance	$\mathcal{O}(n)$
Average Performance	$\mathcal{O}(n)$

Algorithm

1. Linear-time Pivot Based Partitioning: Typical implementations of quicksort contain a partition procedure that can, in linear time, group a list – ranging from indexes left to right – into two parts; those less than a certain element, and those greater than or equal to that element. A variation of the partition scheme, known as the Lomuto partitioning scheme, is simpler but slightly less efficient than Hoare's original partition scheme.
2. One-sided Single Recursive Call: In quicksort, both branches are recursively sorted, leading to the best case $\mathcal{O}(n \log n)$ time. However, when doing selection, the partition in which the desired element lies in is already known, since the pivot is in its final sorted position, with all those preceding it in an unsorted order and all those following it in an unsorted order. Therefore, a single recursive call locates the desired element in the correct position, and this can be built upon for quickselect.
3. Origin of Linear Time Performance: Note the resemblance to quicksort: just as the minimum-based selection algorithm is a partial selection sort, quickselect is also a partial quicksort, generating and partitioning only $\mathcal{O}(\log n)$ of its $\mathcal{O}(n)$ partitions. This simple procedure has expected linear performance, and, like quicksort, has quiet good average performance in practice. It is also an in-place algorithm, requiring only a constant memory overhead if tail call optimization is available, or if the tail recursion can be eliminated with a loop.

Time Complexity

1. Sensitivity to the Pivot Choice: Like quicksort, quickselect has good average performance, but is sensitive to the pivot that is chosen. If good pivots are chosen, meaning ones that consistently decrease the search set by a given fraction, the search

set decreases in size exponentially, and by induction – or summing the geometric series – the performance can be seen to be linear, as each step is linear and the overall time is a constant times this – depending on how quickly the search set reduces.

2. Consistent Choice of Bad Pivots: However, if bad pivots are consistently chosen, such as decreasing by only a single element each time, the worst-case performance is quadratic - $\mathcal{O}(n^2)$. This occurs for example in searching for the maximum element of a set, using the first element as a pivot, and having sorted data.

Variants

1. Median-of-3 Pivot Strategy: The easiest solution is to choose a random pivot, which yields almost certain linear time. Deterministically, one can choose a median-of-3 pivot strategy – as in quicksort – which yields linear performance on partially sorted data, as is common in the real world. However, contrived sequences can still cause worst-case complexity; David Musser describes a *median-of-3 killer* sequence that allows an attack against that strategy, which was one motivation for his introselect algorithm.
2. Introselect - Robust Fallback Augmented Quickselect: One can ensure linear performance even in the worst-case by using a more sophisticated pivot strategy; this is done in the median-of-medians algorithm. However, the overhead of computing the pivot is high, and this is generally not used in practice. One can combine basic quickselect with median-of-medians as fallback to get both fast average case performance and linear worst-case performance; this is done introselect.
3. Improvement in the Time Complexity: Finer computations of average time complexity yield a worst-case of

$$n\big(2 + 2\log 2 + o(1)\big) \leq 3.4n + o(n)$$

for random pivots in the case of median; other k are faster (Eppstein (2007)). The constant can be improved to $\frac{3}{2}$ by a more complicated pivot strategy, yielding the Floyd-Rivest algorithm, which has an average complexity of $1.5n + \mathcal{O}(\sqrt{n})$ for median, with other k being faster.

References

- Eppstein, D. (2007): Blum-style Analysis of Quickselect
- Hoare, C. A. R. (1961): Algorithm 65: Find *Communications of the ACM* **4 (1)** 321-322
- Wikipedia (2019): Quickselect

Median of Medians

Overview

1. Median-of-Medians Algorithm: The *median-of-medians* is an approximate median selectin algorithm, frequently used to supply a good pivot for an exact selection algorithm, mainly the quickselect, that selects the k^{th} largest element of an initially unsorted array (Wikipedia (2020)).
2. Benefit of this Pivot Selector: Median of medians finds an approximate median in linear-time only, which is a limited but additional overhead for quickselect. When this approximate median is used as an improved pivot, the worst-case complexity of quickselect reduces significantly from quadratic to *linear*, which is also the asymptotically optimal worst-case complexity of any selection algorithm.
3. Approximate Median Selection Algorithm: In other words, median-of-medians is an approximate median-selection algorithm that helps building an asymptotically optimal, exact selection algorithm, especially in the sense of worst-case complexity, by producing good pivot elements.
4. Practical Deployment of the Algorithm: Median of medians can also be used as a pivot strategy in quicksort, yielding an optimal algorithm, with a worst-case complexity of $\mathcal{O}(n \log n)$. Although this approach optimizes the asymptotic worst-case quite well, it is typically outperformed in practice by instead choosing random pivots for its average $\mathcal{O}(n)$ complexity for solution and average $\mathcal{O}(n \log n)$ complexity for sorting, without any overhead of computing the pivot.
5. Guaranteed Linear Worst-case Performance: Similarly, median-of-medians is used in the hybrid *introselect* algorithm as a fallback for pivot selection until the k^{th} largest is found. This again ensures a worst-case linear performance, in addition to average-case linear performance; introselect starts with quickselect, with random pivot as default, to obtain good performance, and then falls back to modified quickselect with

the pivot obtained from median-of-medians with pivot obtained from median of medians if the progress is too slow. Even though asymptotically similar, such hybrid algorithm will have a lower complexity than a straightforward introselect by up to a constant factor – both in the average-case as well as the worst-case – at any finite length.

6. Blum-Floyd-Pratt-Rivest-Tarjan: The algorithm was published in Blum, Floyd, Pratt, Rivest, and Tarjan (1973), and thus is called **BFPRT** after the last names of the authors. In the original paper, the algorithm was referred to as **PICK**, referring to quickselect as **FIND**.
7. Algorithm Characteristics:

Class	Selection Algorithm
Data Structure	Array
Worst-case Performance	$\mathcal{O}(n)$
Best-case Performance	$\mathcal{O}(n)$
Worst-case Space Complexity	$\mathcal{O}(\log n)$ auxiliary

Outline

1. Time Complexity of Naïve Quickselect: Quickselect is linear time on average, but it can require quadratic time with poor pivot choices. This is because quickselect is a divide-and-conquer algorithm, with each step taking time that is linear in the size of the remaining search set. If the search set decreases exponentially quickly in size, i.e., by a fixed proportion, this yields a geometric series times the $\mathcal{O}(n)$ factor of a single step, and thus overall linear time. However, if the search set decreases slowly in size, such as linearly – by a fixed number of elements, in the worst-case only reducing one element each time, then a linear sum of linear steps yields quadratic overall time – formally, triangular numbers grow exponentially. For example, the worst-case occurs when pivoting on the smallest element at each step, such as applying quickselect to

the maximum element of an already sorted data and taking the first element as the pivot each time.

2. Optimization using Good Pivoting Strategy: If one instead chooses consistently *good* pivots, this is avoided and one always gets linear performance even in the worst case. A *good* pivot is one for which one can establish that a constant proportion of elements fall above and below it, as then the search set decreases by at least a constant proportion each step, hence exponentially quickly, and the overall time remains linear. The median is a good pivot – the best for sorting, and the best overall for selection – decreasing the search set by half at each step. Thus, if one can compute the median in linear time, this only adds linear time to each step, and thus the overall complexity of the algorithm remains linear.
3. Median of Medians Performance Improvement: The median-of-medians algorithm computes an approximate median, namely a point that is guaranteed to be between the 30^{th} and the 70^{th} percentiles, i.e., in the middle 4 deciles. Thus, the search set decreases by at least 30%. The problem is reduced to 70% of the original size, which is a fixed proportion smaller. Applying the same algorithm on the now smaller set recursively until only one or two elements remain results in a cost of

$$\frac{n}{1-0.7} \approx 3.3333n$$

Algorithm

1. General Template of the Selection Algorithm: As stated earlier, median-of-medians is used as a pivot selection strategy in the quickselect algorithm. Typical implementations carefully handle the left and the right indexes, as well as the n; it is better to use the same index for the left, the right, and n to avoid handle index conversion.
2. Median-of-Median Partition Function: Typical implementations also contain a partition routine that can, in linear time, group the list of indexes ranging from left to

right into three parts – those less than a certain element, those equal to it, and those greater than the element, in a three-way problem. The grouping into three parts ensures that the media-of-medians maintains a linear execution time in the case of many or all coincident elements.

3. Median-of-medians Pivot Selector: The Pivot function is the actual median-of-medians algorithm. It divides its input – a list of length n – into groups of at most 5 elements, computes the median of each of these groups (see below), then recursively computes the true median of the $\frac{n}{5}$ medians found in the previous step (Cormen, Leiserson, Rivest, and Stein (2009)). Pivot function calls the select function above; this is an instance of mutual recursion.
4. Median-of-medians - Subset Median Calculator: The median selector of a group of at most 5 elements is often built into its own function; an easy way to implement this is insertion sort (Cormen, Leiserson, Rivest, and Stein (2009)). It can also be implemented as a decision tree.

Properties of the Pivot

Of the $\frac{n}{5}$ groups, half the number of groups, i.e.,

$$\frac{1}{2} \times \frac{n}{5} = \frac{n}{10}$$

have their median less than the pivot, the median-of-medians. Another half the number of groups, again

$$\frac{1}{2} \times \frac{n}{5} = \frac{n}{10}$$

have their median greater than the pivot. In each of the $\frac{n}{10}$ groups less than the pivot, there are two elements smaller than their respective medians, which are smaller than the pivot. Thus, each of the $\frac{n}{10}$ groups have at least 3 elements that are smaller than the pivot. Similarly, in each of the $\frac{n}{10}$ groups with median greater than the pivot, there are two elements that are greater than their respective medians, which are greater than the pivot. Thus, each of the $\frac{n}{10}$ groups have at least 3 elements that are greater than the pivot. Hence, the pivot is less than $\frac{3n}{10}$ elements and greater than another $\frac{3n}{10}$ elements. Therefore, the chosen median splits the elements somewhere between $30\%/70\%$ and $70\%/30\%$, which assumes worst-case linear behavior of the algorithm.

Proof of $\mathcal{O}(n)$ Running Time

1. Expression for the Time Taken: Let $T(n)$ be the time it takes to run a median-of-medians quickselect algorithm on any array of size n. From the analysis above, this time is

$$T(n) \leq T\left(\frac{n}{5}\right) + T\left(n \cdot \frac{7}{10}\right) + c \cdot n$$

2. Components of the Time Taken:
    a. The $T\left(\frac{n}{5}\right)$ part is for finding the *true* median of the $\frac{n}{5}$ medians, by running an independent quickselect on them, since finding a mean is just a special case of selecting a k-largest element.
    b. The $\mathcal{O}(n)$ term $c \cdot n$ is for the partitioning work to create the two sides, one of which quickselect will recurse on. Each element is visited a constant number of times in order to form them into $\frac{n}{5}$ groups and take each median in $\mathcal{O}(1)$ time.

c. The $T\left(n \cdot \frac{7}{10}\right)$ part is for the actual quickselect recursion fort the worst case, in which the k^{th} element is in the bigger partition that may be of size $n \cdot \frac{7}{10}$ maximally.

3. Proof of Linear Time Consumption: From the above, using induction one can easily show that

$$T(n) \leq c \cdot n \in \mathcal{O}(n)$$

Analysis

1. Origin of the Linear Runtime: The key step is reducing the problem to selecting two links whose total length is shorter than the original list, plus a linear factor for the reduction step. This allows a simple induction to show that the overall running time is linear.
2. Choice of Five-Element Groups: The specific choice of the group of five elements is explained as follows. First, computing the median of an odd list is faster and simpler; while one could use an even list, this would require taking the average of the two middle elements, which is slower than simply selecting the exact middle element. Second, five is the smallest odd number such that the median-of-medians works. With groups of only three elements, the resulting list of medians to search in is of length $\frac{n}{3}$, and this reduces the list to recurse into length $\frac{2n}{3}$. Thus, this still leaves n elements to search in, not reducing the problem sufficiently. The individual lists are shorter, however, and one can bound the resulting complexity to $\mathcal{O}(n \log n)$ by the Akra-Bazzi method, but it does not prove linearity.
3. Increasing the Size of Groups: Conversely, one may instead group by 7, 9, or more elements (g), and this does work. This reduces the size of the list of

medians to $\frac{n}{g}$, and the size of list to recurse asymptotes at $\frac{3n}{4}$ (75%), as the quadrants approximate at 25%, since the size of the overlapping lines decreases proportionally. This reduces the scaling factor from 10 asymptotically down to 4, but accordingly raises the c term for the partitioning work. Finding the median of a larger group takes longer, but it is a constant factor that depends only on g, and thus does not change the overall performance as n grows. In fact, considering the number of comparisons in the worst-case, the constant factor is $\frac{2g(g-1)}{g-3}$.

4. Fixed Number of Lists in the Array: If one instead groups the other way, say dividing the n element list into 5 lists, computing the median of each, and then computing of these – grouping by a constant fraction, not a constant number - one does not as clearly reduce the problem, since it requires computing 5 medians, each in a list of $\frac{n}{5}$ elements, and then recursing on a list of length at most $\frac{7n}{10}$. As with grouping by 3, the overall length is not shorter – in fact, longer – and one can thus only prove super-linear bounds. Grouping into a square of $\sqrt{n}$ lists of length $\sqrt{n}$ is similarly complicated.

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
- Cormen, T., C. E. Leiserson, R. Rivest, and C. Stein (2009): *Introduction to Algorithms 3rd Edition* **MIT Press**

- Wikipedia (2020): Median of Medians

Floyd-Rivest Algorithm

Overview

1. Focus of the Floyd-Rivest Algorithm: The *Floyd-Rivest algorithm* is a selection algorithm that has an optimal expected number of comparisons within lower-order terms (Wikipedia (2019)). It is functionally equivalent to quickselect, but runs faster in practice than average (Floyd and Rivest (1975a, 1975b)). It has an expected running time of $\mathcal{O}(n)$ and an expected number of comparisons of $n + \min(k, n-k) + \mathcal{O}(\sqrt{n})$.
2. Algorithm Characteristics:

Class	Selection Algorithm
Data Structure	Array
Average Performance	$n + \min(k, n-k) + \mathcal{O}(\sqrt{n})$

Algorithm

1. Structure of the Floyd-Rivest Algorithm: The Floyd-Rivest algorithm is a divide-and-conquer algorithm, sharing many similarities with quickselect. It uses sampling to help partition the list into three sets. It then recursively selects the k^{th} smallest element from the appropriate set.
2. Drawing a Small Random Sample: Select a small random sample S from the list L.
3. Selection of the Partition Pivots: From S recursively select two elements u and v such that

$$u < v$$

These two elements will be the *pivots* for the partition and are expected to contain the k^{th} smallest element of the entire list between them – in a sorted list.

4. Partition into Three Subsets: Using u and v partition S into three subsets; A, B, and C. A will contain elements with values less than u, B will contain elements with values between u and v, and C will contain the elements with values greater than v.
5. Assign $L - S$ Elements into Appropriate Subsets: Partition the remaining elements in L, that is, the elements in $L - S$, by comparing them to u or v and placing them into the appropriate set. If k is smaller than half the number of elements in L rounded up, the remaining elements should be compared to v first and then only to u if they are smaller than v. Otherwise, the remaining elements should be compared to u first and only to v if they are greater than u.
6. Recursively Extracting the Order Statistic: Based on the value of k, apply the algorithm recursively to the appropriate set to select the k^{th} smallest element in L.

References

- Floyd, R. W., and R. L. Rivest (1975a): Expected Time Bounds for Selection *Communications of the ACM* **18 (3)** 165-172
- Floyd, R. W., and R. L. Rivest (1975b): Algorithm 489: The Algorithm SELECT – for finding the i^{th} smallest of n Elements *Communications of the ACM* **18 (3)** 173
- Wikipedia (2019): Floyd-Rivest Algorithm

Introselect

Overview

1. Idea behind the Introselect Algorithm: *Introselect* – short for *introspective selection* – is a selection algorithm that is a hybrid of quickselect and median of medians which has fast average performance and optimal worst-case performance (Wikipedia (2020)).
2. Relation to the Introsort Algorithm: Introselect is related to the introsort sorting algorithm: these are analogous refinements to basic quicksort and quickselect algorithm, in that they both start with the quick algorithm, which has good optimal performance and low overhead, but fall back to an optimal worst-case algorithm – with higher overhead – if the quick algorithm does not progress rapidly enough.
3. Optimal Average and Worst Cases: Both algorithms were designed by Musser (1997) with purpose of providing generic algorithms for the C++ Standard Library that have both fast performance and optimal worst-case performance, thus allowing the performance requirements to be highlighted.
4. Algorithm Characteristics:

Class	Selection Algorithm
Data Structure	Array
Worst-case Performance	$\mathcal{O}(n)$
Best-case Performance	$\mathcal{O}(n)$

Algorithm

1. Methodology of the Introsort Algorithm: Introsort achieves practical performance comparable to quicksort while preserving $\mathcal{O}(n \log n)$ worst-case behavior by creating a hybrid of quicksort and heapsort. Introsort starts with quicksort, so it achieves performance similar to quicksort if quicksort works, and falls back to heapsort – which has optimal worst-case performance – if the quicksort does not progress quickly enough.
2. Methodology of the Introselect Algorithm: Introselect works by optimistically starting out with quickselect and switching to a linear time worst-case selection algorithm – the Blum-Floyd-Pratt-Rivest-Tarjan median-of-medians algorithm – if it recurses too many times without making sufficient progress. The switching strategy is the main technical content of this algorithm. Simply limiting the recursion to constant depth is not good enough, since this would make the algorithm switch on all sufficiently large lists. Musser (1997) discusses a couple of simple approaches.
3. Tracking Sized of the Processed Sub-partitions: Keep track of the list of sizes of the sub-partitions processed so far. If at any point k recursive calls have been made without halving the list size, for some positive small k, switch to the worst-case linear algorithm.
4. Tracking Cumulative Sizes of the Generated Partitions: Sum the sizes of all the partitions generated so far. If this exceeds the list size times some small positive constant k switch to the worst-case algorithm. The sum is easy to track on a single scalar variable.
5. Guaranteed Limit on Recursion Depth: Both approaches limit the recursion depth to

$$k\lceil \log n \rceil = \mathcal{O}(\log n)$$

and the total running time to $\mathcal{O}(n)$.

References

- Musser, D. R. (1997): Introselect Sorting and Selection Algorithms *Software: Practice and Experience* **27 (8)** 983-993
- Wikipedia (2020): Introselect

Order Statistic Tree

Overview

1. Order Statistic Tree Operations Supported: An *order-statistic tree* is a variant of a binary search tree – or, more generally, a B-tree – that supports two additional operations beyond insertion, lookup, and deletion:
    a. $Select\ (i)$ => Finds the i^{th} smallest element stored in the tree
    b. $Rank\ (x)$ => Finds the rank of element x in the tree, i.e., its index in the sorted list of elements of the tree
2. Time Complexity of the Extra Operations: Both operations can be performed in $\mathcal{O}(\log n)$ worst-case time when a self-balancing tree is used as the base data structure.

Augmented Search Tree Implementation

1. Augmenting the Regular Search Tree: To turn a regular search tree into an order-statistic tree, the nodes of the tree need to store one additional value, which is the size of the sub-tree rooted at that node, i.e., the number of nodes below it.
2. Invariants Mandated during Tree Modifications: All operations that modify the tree must adjust this information to the preserve the invariant

$$size[x] = size\big[left[x]\big] + size\big[right[x]\big] + 1$$

where

$$size[x] = \emptyset$$

by definition.

3. Implementation of Select and Rank: Implementation of $select$ and $rank$ is fairly trivial, as is illustrated in Cormen, Leiserson, Rivest, and Stein (2009) and Wikipedia (2019).
4. Additional Augmentations for Specific Trees: Order-statistic trees can be further amended with book-keeping information to maintain balance, i.e., the tree height can be added to get an order statistic AVL tree, or a color bit to get a red-black order statistic tree. Alternatively, the size field can be used in conjunction with w weight-balancing scheme at no additional storage cost (Roura (2001)).

References

- Cormen, T., C. E. Leiserson, R. Rivest, and C. Stein (2009): *Introduction to Algorithms 3rd Edition* **MIT Press**
- Roura, S. (2001): A New Method for Balancing Binary Search Trees, in: *International Colloquium on Automata, Languages, and Programming* 469-480
- Wikipedia (2019): Order Statistic Tree

Maximum Sub-array Problem

Overview

1. Maximum Sum Subarray Problem: The *maximum sum subarray problem* is the task of finding a contiguous subarray with the largest sum within a given one-dimensional array $A[1, \cdots, n]$ of numbers. Formally, the task is to find indexes i and j with

$$1 \leq i \leq j \leq n$$

   such that the sum $\sum_{x=i}^{j} A[x]$ is as large as possible (Wikipedia (2020)). Some formulations of the problem also allow the empty subarrays to be considered; by convention, the sum of all values of the empty subarray is zero. Each member in the input array A can be positive, negative, or zero (Bentley (1989)).
2. Maximum Sum Subarray Problem Characteristics: Some properties of the problem are:
   a. If the array contains all negative numbers, then the problem is trivial; the maximum subarray is the entire array.
   b. If the array contains all non-positive numbers, then a solution is any subarray of size 1 containing the maximal value of the array, or the empty subarray, if it is permitted.
   c. Several different subarrays may have the same maximum sum.
3. Approaches to Solving the Problem: As discussed in Bentley (1989), this problem can be solved using several different algorithmic techniques, including brute-force, divide-and-conquer, dynamic programming, and reduction to shortest paths.

Background

1. Digitized Images Simplified MLE Model: The maximum subarray problem was proposed by Ulf Grenander in 1977 as a simplified model for the maximum likelihood estimation of patterns in digitized images (Bentley (1984)).
2. Rectangular Subarray with Maximum Sum: Grenander was looking to find a rectangular subarray with maximum sum in a 2D array of real numbers. A brute-force approach to the 2D problem runs in $\mathcal{O}(n^6)$ time; because this was prohibitively slow, Grenander proposed the 1D problem to gain insight into the structure.
3. $\mathcal{O}(n^2)$ Solution for the 1D Problem: Grenander derived an algorithm that solves the problem in $\mathcal{O}(n^2)$ time, improving the brute-force running time of $\mathcal{O}(n^3)$. He did this by using a pre-computed table of cumulative sums

$$S[k] = \sum_{x=i}^{k} A[x]$$

to compute the subarray sum

$$\sum_{x=i}^{j} A[x] = S[j] - S[j-1]$$

in constant time.
4. Improvements by Shamos and Kadane: Michael Shamos devised an $\mathcal{O}(n \log n)$ divide-and-conquer algorithm for it. Jay Kadane designed a $\mathcal{O}(n)$ algorithm (Gries (1982), Bentley (1984, 1989)), which is as fast as possible, since every algorithm must at least scan the array once, which already takes $\mathcal{O}(n)$ time.
5. $\mathcal{O}(n)$ Solutions of Gries/Bird: Gries (1982) obtained the same $\mathcal{O}(n)$-time algorithm by applying Dijkstra's *standard strategy*; Bird (1989) derived it purely by algebraic manipulation of the brute-force algorithm using the Bird-Meertens formalism.

6. 2D Distance-Matrix Multiplication Approaches: Grenander's 2D generalization can be solved in $\mathcal{O}(n^3)$ time either by using Kadane's algorithm as a sub-routine, or through a divide-and-conquer approach. Slightly faster algorithms based on distance-matrix multiplication have been proposed by Tamaki and Tokuyama (1998) and Takaoka (2002).
7. Optimality of the above Solutions: There is some evidence that no significantly faster algorithm exists; an algorithm that solves the 2D maximum subarray problem in $\mathcal{O}(n^{3-\varepsilon})$ time, for any

$$\varepsilon > 0$$

would imply a similarly fast algorithm for the all-pairs shortest paths problem (Backurs, Dikkala, and Tzamos (2016)).

Applications

1. Usage in Genomic Sequence Analysis: Genomic sequence analysis employs maximum subarray algorithms to identify important biological segments of protein sequences. These problems include conserved segments, GC-rich regions, tandem-repeats, low- complexity filters, DNA binding domains, and regions of high charge.
2. Brightest Areas in Bitmap Images: In computer vision, maximum subarray algorithms are used on bitmap images to detect the brightest area in an image.

Kadane's Algorithm

1. Basic Thrust of the Algorithm: Kadane's algorithm scans the given array $A[1, \cdots, n]$ from left to right. At the j^{th} step, it computes the subarray with the largest sum ending at j; the sum is maintained in the variable $current\_sum$. Moreover, it

computes the subarray with the largest sum anywhere in $A[1, \cdots, j]$, maintained in the variable $best\_sum$, and easily obtained as the maximum of all values of $current\_sum$ so far.

2. Running Update of the $current\_sum$ Variable: As a loop invariant, in the j^{th} step, the old value of $current\_sum$ holds the maximum over all

$$i \in \{1, \cdots, j\}$$

of the sum $A[i] + \cdots + A[j-1]$ – note that the sum is 0 when

$$i = j$$

corresponding to the empty subarray $A[j, \cdots, j-1]$. Therefore, $current\_sum + A[j]$ is the maximum over all

$$i \in \{1, \cdots, j\}$$

of the sum $A[i] + \cdots + A[j]$. To extend the latter maximum to cover also the case

$$i = j + 1$$

it is sufficient also to consider the empty subarray $A[j+1, \cdots, j]$. This is done by assigning $\max(0, current\_sum + A[j])$ as the new value of $current\_sum$, which after that holds the maximum over all

$$i \in \{1, \cdots, j+1\}$$

of the sum $A[i] + \cdots + A[j]$.

3. Disallowing the Empty Subarray Inputs: The above version of the algorithm will return 0 if the input contains no positive elements – including when the input is

empty. For a variant of the problem that allows empty subarrays, $best\_sum$ should be initialized to $-\infty$ instead (Bentley (1989)). Further, $current\_sum$ should be updated as $\max(x, current\_sum + x)$ In the case where the input contains no positive element, the returned value is that of the largest element, i.e., the least negative value, or $-\infty$ if the input array is empty.

4. Starting/Ending Maximum Subarray Indexes: The algorithm can be modified to keep track of the starting and the ending subarray indexes of the maximum subarray as well.
5. Trivial Instance of Dynamic Programming: Because of the way the algorithm uses optimal structures, i.e., the maximum subarray ending at each position is calculated in a simple way from a related but smaller and overlapping problem – the maximum subarray ending at the previous position – this algorithm can be viewed as a simple/trivial example of dynamic programming.
6. Runtime Complexity of the Algorithm: The runtime complexity of Kadane's algorithm is $\mathcal{O}(n)$ (Gries (1982), Bentley (1989)).

Generalizations

1. Higher Dimension Maximum Subarray Sums: Similar problems may be posed for higher dimensional arrays, but their solutions are more complicated, see, e.g., Takaoka (2002).
2. k Largest Subarray Sums in 1D: Brodal and Jorgensen (2007) show how to find the k largest subarray sums in a 1D array, in the optimal time bound $\mathcal{O}(n + k)$.
3. Maximum Sum k Disjoint Subarrays: The maximum sum k disjoint subarrays can also be computed in optimal time bound $\mathcal{O}(n + k)$ (Bengtsson and Chen (2007)).

References

- Backurs, A., N. Dikkala, and C. Tzamos (2016): Tight Hardness Results for Maximum Weight Rectangles *Proceedings of the 43th International Colloquium on Automate, Languages, and Programming* 1-13
- Bengtsson, F., and J. Chen (2007): Computing Maximum-scoring Segments Optimally
- Bentley, J. (1984): Programming Pearls: Algorithm Design Techniques *Communications of the ACM* **27 (9)** 865-873
- Bentley, J. (1989): *Programming Pearls 2nd Edition* **Addison-Wesley** Reading MA
- Bird, R. S. (1989): Algebraic Identities for Program Calculation *Computer Journal* **32 (2)** 122-126
- Brodal. G. S., and A. G. Jorgensen (2007): A Linear Time Algorithm for the k Maximal Sums Problem
- Gries, D. (1982): A note on a Standard Strategy for developing Loop Invariants and Loops *Science of Computer Programming* **2 (3)** 207-214
- Takaoka, T. (2002): Efficient Algorithms for the Maximum Subarray Problem by Distance Matrix Multiplication
- Tamaki, J., and T. Tokuyama (1998): Algorithms for the Maximum Subarray Problem based on Matrix Multiplication *Proceedings on the 9th Symposium on Discrete Algorithms* 446-452
- Wikipedia (2020): Maximum Subarray Problem

Subset Sum Problem

Overview

1. Statement of the Subset Sum Problem: The *subset sum problem* is an important decision problem in complexity theory and crystallography. There are equivalent formulations of the problem. One of them is: given a set – or multi-set – of integers, is there a non-empty subset whose sum is zero? The problem is **NP**-complete, meaning that while it is easy to confirm whether a proposed solution is valid, it may be inherently prohibitive to determine in the first place whether any solution exists (Wikipedia (2020)).
2. Relationship to Knapsack and Partition Problems: The problem can be equivalently formulated as follows: Given the integers or natural numbers $w_1, \cdots, w_n$ does any subset of them sum precisely to W (Kleinberg and Tardos (2022))? Subset sum can be thought of as a special case of the knapsack problem (Martello and Toth (1990)). One interesting special case of the subset sum is the partition problem, in which W is half the sum of all the elements in the set, i.e.,

$$W = \frac{1}{2}(w_1 + \cdots + w_n)$$

Complexity

1. Parameters Determining the Problem Complexity: The complexity of the subset sum problem can be viewed as depending on two parameters - N, the number of decision variables, and P, the precision of the problem, stated as the number of binary digit values that it takes to state the problem.

2. Exponential Complexity on the Smaller Parameter: The complexity of the best-known algorithms is exponential in smaller of the two parameters N and P. Thus, the problem is most difficult if N and P are of the same order. It only becomes easy if either N or P becomes very small.
3. Solutions for Small N/P: If N – the number of variables – is small, then an exhaustive search for the solution is practical. If P – the number of digit values – is a small fixed number, then there are dynamic programming algorithms that can solve it exactly. Efficient algorithms for both small N and small P cases are given below.

Exponential Time Algorithm

1. Naïve Exponential in N Approach: There are several ways to solve the subset sum problem in time exponential in N. The most naïve algorithm would be to cycle through all subsets of N numbers, and, for every one of them, heck if the subset sums to the correct target. The running time is of the order $\mathcal{O}(2^N.N)$, since there are 2^N subsets, and, to check each subset, one needs to sum at most N elements.
2. Sum Lists of Split Subsets: A better exponential time algorithm is known which works in time $\mathcal{O}\left(2^{\frac{N}{2}}\right)$. The algorithm splits arbitrarily the N elements into two sets of $\frac{N}{2}$ each. For each of the two sets, it stores a list of the sums of all $2^{\frac{N}{2}}$ possible subsets of its elements.
3. Sorted List of Subset Sums: Each of these two lists is then sorted. Using a standard comparison sorting algorithm for this step would take time $\mathcal{O}\left(2^{\frac{N}{2}}.N\right)$.
4. Sorting/Merging an Extra Element: However, given a sorted list of sums for k elements, the list can be expanded into two sorted lists with the introduction of a $(k+1)^{th}$ element, and these two sorted lists can be merged in time $\mathcal{O}(2^k)$. Thus, each list can be generated in sorted form in time $\mathcal{O}\left(2^{\frac{N}{2}}\right)$.

5. Sum across Two Sorted Lists: Given the two sorted lists, the algorithm can check if the element of the first array and an element of the second array sum up to s in time $\mathcal{O}\left(2^{\frac{N}{2}}\right)$. To do that, the algorithm passes through the first array in decreasing order – starting at the largest element – and the second array in increasing order – starting at the smallest element. Whenever the sum of the first element in the current array and the current element in the second array is more than s, the algorithm moves to the next element in the first array. If it is less than s, the algorithm moves to the next element in the second array. If two elements that sum to s are found, it stops (Horowitz and Sahni (1974)).

Pseudo-polynomial Time Dynamic Programming Solution

1. Setup of Dynamic Programming Framework: The problem can be solved in pseudo-polynomial time using dynamic programming. Suppose the sequence is $x_1, \cdots, x_N$ sorted in the increasing order, and one wishes to determine if there is a non-empty subset that sums to 0.
2. Dynamic Programming Boolean Decision Function: Define the boolean valued function $Q(i, s)$ to be the value – *true* or *false* – of: "there is a non-empty subset of $x_1, \cdots, x_N$ which sums to s". Thus, the solution to the problem "Given a set of interest, is there a non-empty subset whose sum is zero?" is $Q(N, s)$.
3. Extremum Sums of Positive/Negative: Let A be the sum of the negative values and B be the sum of the positive values. Clearly,

$$Q(i, s) = false$$

if

$$s < A$$

or

$$s > B$$

So, these values do not need to be stored or computed.

4. Array of Decision Function Values: Create an array to hold the values $Q(i, s)$ for

$$1 \leq i \leq N$$

and

$$A \leq s \leq B$$

Set

$$Q(1, s) := (x_1 == s)$$

Then, for

$$i = 2, \cdots, N$$

set

$$Q(i, s) = Q(i-1, s) \; \boldsymbol{or} \; (x_i == s) \; \boldsymbol{or} \; Q(i-1, s - x_i)$$

for

$$A \leq s \leq B$$

5. Analysis of the Dynamic Programming Step: For each assignment, the values of Q on the right are already known, either because they were stored in the table for the previous value of i or because

$$Q(i-1, s-x_i) = false$$

if

$$s - x_i < A$$

and

$$s - x_i < B$$

Therefore, the total number of arithmetic operations is $\mathcal{O}(N[A-B])$ For example, if all the values are $\mathcal{O}(N^k)$ for some k, then time required is $\mathcal{O}(N^{k+2})$. This algorithm is easily modified to return the subset with sum 0 if one such subset exists.

6. Non-Polynomial in the Represented Bits: The dynamic programming solution has a runtime of $\mathcal{O}(sN)$ where s is the sum one wants to find in the set of N numbers. This solution does not count as polynomial time in complexity theory because $B-A$ is not polynomial in the *size* of the problem, which is the number of bits used to represent it. This algorithm is polynomial in the values of A and B, which are exponential in their numbers of bits.
7. Arrays with Positive, Bounded Elements: For the case that each x_i is positive and bounded by a fixed constant C, Pisinger (1999) found a linear time algorithm having a complexity of $\mathcal{O}(NC)$ – note that this is for the version of the problem where the target sum is not necessarily zero, otherwise the problem would be trivial (Pisinger (1999)). Koiliaris and Xu (2015) found a deterministic $\mathcal{O}\left(s\sqrt{N}\right)$ algorithm for the subset problem where s is the sum one needs to find. Later, Bringmann (2017) found a randomized $\mathcal{O}(s)$ time algorithm.

Polynomial-Time Approximate Algorithm

1. Approximate Subset Sum – Problem Statement: An approximate version of the subset sum problem would be: given a set of N numbers $x_1, \cdots, x_N$ and a number s, output:
    a. Yes, if there is a subset that sums up to s.
    b. No, if there is no subset summing up to a number between $(1 - c)s$ and s for some small

$$c > 0$$

    c. Any answer, if there is a subset summing up to a number between $(1 - c)s$ and s but no subset summing up to s.
2. Approximate Subset Sum – Problem Complexity: If all numbers are non-negative, the approximate subset sum is solvable in time polynomial in N and $\frac{1}{c}$.
3. Exponential Complexity on the Number of Bits: The solution for subset sum also provides the solution for the original subset sum problem in the case where the numbers are small – again, for non-negative numbers. If any sum of the numbers can be specified with at most p bits, then solving the problem approximately with

$$c = 2^{-P}$$

    is equivalent to solving it exactly. Then, the polynomial time algorithm for the approximate subset sum becomes an exact algorithm with running time polynomial in N and 2^P, i.e., exponential in P.
4. Approximate Subset Sum Polynomial Algorithm: The algorithm for the approximate subset sum problem is presented in Wikipedia (2020).

5. Origin of the Polynomial Time Complexity: The algorithm is polynomial time because the intermediate lists used always remain of size polynomial in N and $\frac{1}{c}$ and, as long as they are of polynomial size, all operations on them can be done in polynomial time. The size of the lists is kept polynomial in the trimming step, in which a given number is only included if it is greater than the previous one by $\frac{cs}{N}$ and not greater than s.
6. Limits on the Number of Elements in the List: This ensures that each element in the process list is smaller than the next one by at least $\frac{cs}{N}$ and does not contain elements greater than s. Any list with that property consists of no more than $\frac{N}{c}$ elements.
7. Correctness of the Algorithm: The algorithm is correct because each step introduces an additive error of at most $\frac{cs}{N}$ and N steps together introduce an error of at most cs.

References

- Bringmann, K. (2017): A near-linear Pseudo-polynomial Time Algorithm for Subset Sums *Proceedings of the 28th Annual ACM SIAM Symposium on Discrete Algorithms* 1073-1084
- Horowitz, E., and S. Sahni (1974): Computing Partitions with Applications to the Knapsack Problem *Journal of the ACM* **21 (2)** 277-292
- Kleinberg, J., and E. Tardos (2022): *Algorithm Design 2nd Edition* **Pearson**
- Koiliaris, K., and C. Xu (2016): A Faster Pseudo-polynomial Time Algorithm for Subset Sum **arXiV**
- Martello, S., and P. Toth (1990): *Knapsack Problems: Algorithms and Computer Interpretations* **Wiley-Interscience**
- Pisinger, D. (1999): Linear-time Algorithms for Knapsack Problems with Bounded Weights *Journal of Algorithms* **33 (1)** 1-14
- Wikipedia (2020): Subset Sum Problem

3SUM

Overview

1. Statement of the 3SUM Problem: The *3SUM* problem asks if a given set of n real numbers contains 3 elements that sum to 0. A generalized version, $kSUM$, asks the same question on k numbers. 3SUM can be easily solved in $\mathcal{O}(n^2)$ time, and matching $\Omega\left(n^{\lceil \frac{k}{2} \rceil}\right)$ lower bounds are known in some specialized models of computation (Erickson (1999, Wikipedia (2020)).
2. Time Complexities of Various Approaches: It was conjectured that any deterministic algorithm for the 3SUM requires $\Omega(n^2)$ time. The original 3SUM conjecture was refuted by Gronlund and Pettie (2014) who gave a deterministic algorithm that solves 3SUM in $\mathcal{O}\left(\frac{n^2}{(\log n/\log\log n)^{\frac{2}{3}}}\right)$ time. Additionally, they showed that the 4-linear decision tree complexity of 3SUM is $\mathcal{O}\left(n^{\frac{3}{2}}\sqrt{\log n}\right)$. These bounds were subsequently improved in Freund (2017), Gold and Sharir (2018), and Chan (2018). The current best-known algorithm for 3SUM runs in $\mathcal{O}\left(\frac{n^2[\log\log n]^{o(1)}}{\log^2 n}\right)$ time (Chan (2018)). Kane, Lovett, and Moran (2018) showed that the 6-linear decision tree complexity of 3SUM is $\mathcal{O}(n\log^2 n)$. The latter bound is tight, up to a logarithmic factor. It is still conjectured that 3SUM is unsolvable in $\mathcal{O}\left(n^{2-\Omega(1)}\right)$ expected time (Kopelowitz, Pettie, and Porat (2014)).
3. Positive/Negative Range-Bound Array: When the elements are in the range $[-N, \cdots, +N]$, 3SUM can be solved in $\mathcal{O}(n + N\log N)$ time by representing the input S as a bit vector, computing the set $S + S$ of all pairwise sums as a discrete convolution using a Fast-Fourier transform, and finally comparing this to $-S$ (Cormen, Leiserson, Rivest, and Stein (2009)).

Quadratic Algorithm

1. Hashtable-Based $\mathcal{O}(n^2)$ Solution: Suppose the input array is $S[0, \cdots, n-1]$. In integer, i.e., word RAM, models of computing, 3SUM can be solves in $\mathcal{O}(n^2)$ time on average by inserting each number $S[i]$ into a hashtable, and then for each index i and j, checking whether the hashtable contains the integer $-(S[i] + S[j])$.
2. Sorting-Based $\mathcal{O}(n^2)$ Solution: It is also possible to solve the problem in the same time using a comparison-based model of computation or real RAM, for which hashing is not allowed. The algorithm first sorts the input array and then tests all possible pairs in a careful order that avoids the need to binary search for the pairs in the sorted list, achieving the worst-case $\mathcal{O}(n^2)$ time.
3. Correctness of the Algorithm: The correctness of the algorithm can be seen as follows. Suppose there is a solution

$$a + b + c = 0$$

Since the pointers move in only one direction, the algorithm can be run until the left-most pointer points to a. The algorithm is then run until one of the remaining pointer points to b or c, whichever occurs first. Then the algorithm is run until the last pointer points to the remaining term, giving the affirmative solution.

Variants

1. Non-zero Sum: Instead of looking for numbers whose sum is 0, it is possible to look for numbers whose sum is any constant C in the following way:
    a. Subtract $\frac{C}{3}$ from all elements of the input array.

b. In the modified array, find 3 elements whose sum is 0.

One could also simply modify the original algorithm to search the hashtable for the integer $C - (S[i] + S[j])$.

2. 3-Array Sum Variant: Instead of searching for 3 numbers in a single array, one can search for them in three different arrays, i.e., given 3 arrays X, Y, and Z, find three numbers

$$a \in X$$

$$b \in Y$$

and

$$c \in Z$$

such that

$$a + b + c = 0$$

Call the 1-array sum variant $3SUM \times 1$ and the 3-array variant $3SUM \times 3$.

3. Approach for the $3SUM \times 3$ Problem: Given a solver for $3SUM \times 1$, the $3SUM \times 3$ problem can be solved in the following way, assuming all elements are integers:

a. For every element in X, Y, and Z, set

$$X[i] \leftarrow X[i] \times 10 + 1$$

$$Y[i] \leftarrow Y[i] \times 10 + 2$$

$$Z[i] \leftarrow Z[i] \times 10 - 3$$

b. Let S be a concatenation of the elements X, Y, and Z

c. Use the $3SUM \times 1$ Oracle to find three elements

$$a' \in X$$

$$b' \in Y$$

and

$$c' \in Z$$

such that

$$a' + b' + c' = 0$$

d. Return

$$a \leftarrow \frac{a' - 1}{10}$$

$$b \leftarrow \frac{b' - 1}{10}$$

$$c \leftarrow \frac{c' - 1}{10}$$

4. Correctness Guarantee for $3SUM \times 3$: By the way the arrays are transformed, it is guaranteed that

$$a \in X$$

$$b \in Y$$

and

$$c \in Z$$

5. Convolution Sum: Instead of looking for arbitrary elements of the array such that

$$S[k] = S[i] + S[j]$$

the *convolution 3SUM* - $Conv3SUM$ – looks for elements in specific locations (Patrascu (2010))

$$S[i + j] = S[i] + S[j]$$

Reduction from $Conv3SUM$ to $3SUM$

1. Transformation into the $3SUM \times 1$ Problem: Given a solver for $3SUM$, the $Conv3SUM$ problem can be solved in the following way (Patrascu (2010)):
    a. Define new array T such that for every index i

$$T[i] = 2nS[i] + i$$

where n is the number of elements in the array, and the indexes run from 0 to $n - 1$.

    b. Solve $3SUM$ on the array T.
2. Forward Equivalence between S and T: If in the original array there is a triple with

$$S[i + j] = S[i] + S[j]$$

then

$$T[i+j] = 2nS[i+j] + i + j = (2nS[i] + i) + (2nS[j] + j) = T[i] + T[j]$$

Thus, this solution will be found by $3SUM$ on T.

3. Equivalence of the Convex Setup: Conversely, if in the new array there is a triple with

$$T[k] = T[i] + T[j]$$

then

$$2nS[k] + k = 2n(S[i] + S[j]) + (i + j)$$

Because

$$i + j < 2n$$

necessarily

$$S[k] = S[i] + S[j]$$

and

$$k = i + j$$

Thus, this is a valid solution for $Conv3SUM$ on S.

Reduction from $3SUM$ to $Conv3SUM$

1. Definition of Linear Hash Function: Given a solver for $Conv3SUM$, the $3SUM$ problem can be solved in the following way (Patrascu (2010), Kopelowitz, Pettie, and Porat (2014)). The reduction uses a hash function. As a first approximation, one assumes that there exists a linear hash function, i.e., a function h such that

$$h(x + y) = h(x) + (y)$$

2. Reduced Set of Hash Buckets: Suppose that all elements are integers in the range $0, \cdots, N - 1$, and that the function h maps each element to an element in the smaller range of indexes $0, \cdots, n - 1$. Create a new array T and send each element of S to its hash-value in T, i.e., for every x in S

$$T[h(x)] = x$$

3. Unique Hash Mappings - $Conv3SUM$ Solution: Initially, suppose that the mappings are unique, i.e., each cell in T accepts only a single element from S. Solve $Conv3SUM$ on T. Then:
4. $Conv3SUM$ Solution using a Linear Hash: If there is a solution for the $3SUM$

$$z = x + y$$

then

$$T[h(z)] = T[h(x)] + T[h(y)]$$

and

$$h(z) = h(x) + h(y)$$

thus, this solution will be found by the $Conv3SUM$ solver on T.

5. Permutative Equivalence between S and T: Conversely, if $Conv3SUM$ is found on T, then obviously it corresponds to a $3SUM$ solution on S since T is just a permutation of S.
6. Handling Collisions of Hash Values: This idealized doesn't work, because any hash function might map several distinct elements of S to the same cell of T. The trick is to create an array T^* by selecting a single random element from each cell of T, and run $Conv3SUM$ on T^*. If a solution is found, then it is a correct solution for $3SUM$ on S. If no solution is found, then create a different random T^* and try again. Suppose there are at most R elements in each cell of T. Then the probability of finding a solution – if a solution exists – is the probability that the random selection will select the correct element from each cell, which is $\frac{1}{R^3}$. By running $Conv3SUM$ R^3 times, the solution will be found with a high probability.
7. Imperfect (Almost) Linear Hash Function: Unfortunately, there is no perfect linear hashing, so one has to use an almost linear has function, i.e., a function h such that

$$h(x + y) = h(x) + h(y)$$

or

$$h(x + y) = h(x) + h(y) + 1$$

8. $Conv3SUM$ Solution using Almost Linear Hashing: This requires duplicating the elements of S when copying them into T, i.e., put every element

$$x \in S$$

both in $T[h(x)]$ as before, and in $T[h(x)] - 1$. Thus, each cell will have $2R$ elements, and one will have to run $Conv3SUM$ $(2R)^3$ times.

3SUM-Hardness

1. Definition of 3$SUM - Hard$ Problems: A problem is called 3$SUM - Hard$ if solving it in sub-quadratic time implies a sub-quadratic time algorithm for 3SUM. The concept of 3$SUM - Hardness$ was introduced by Gajentaan and Overmars (1995). They proved that a large class of problems in computational geometry is 3$SUM - Hard$.
2. Equivalent 3$SUM - Hard$ Computational Geometry Problems:
    a. Given a set of lines in a plane, are there three that meet in a point?
    b. Given a set of non-intersecting axis-parallel line segments, is there a line that separates them into two non-empty subsets?
    c. Given a set of infinite strips in a plane, do they fully cover a given rectangle?
    d. Given a set of triangles in a plane, compute their measure.
    e. Given a set of triangles in a plane, does their union have a hole?
3. Visibility and Motion Planning Problems:
    a. Given a set of horizontal triangles in space, can a particular triangle be seen from a particular point?
    b. Given a set of non-intersecting axis-parallel line segment obstacles in a plane, can a given rod be moved by translations and rotations between a starting and an ending position pair without colliding on the obstacles?
4. Decision Version of Dual Set Sorting: The decision version of $X + Y$ sorting in another example: given sets of numbers X and Y with n elements each, are there n^2 distinct $x + y$ for

$$x \in X$$

and

$$y \in Y$$

(Demaine, Erickson, and O'Rourke (2006))?

References

- Chan, T. M. (2018): More Logarithmic Factor Speedups for 3SUM, (median+)-Convolution, and some Geometric 3SUM-Hard Problems *Proceedings of the 29th Annual ACM-SIAM Symposium on Discrete Algorithms* 881-897
- Cormen, T., C. E. Leiserson, R. Rivest, and C. Stein (2009): *Introduction to Algorithms 3rd Edition* **MIT Press**
- Demaine, E., J. Erickson, and J. O'Rourke (2006): Problem 41: Sorting X + Y (Pairwise Sums)
- Erickson, J. (1999): Lower Bounds for Linear Satisfiability Problems *Chicago Journal of Theoretical Computer Science* **8** 1-30
- Freund, A. (2017): Improved Sub-quadratic 3SUM *Algorithmica* **44 (2)** 440-458
- Gajentaan, A., and M. H. Overmars (1995): On a Class of $\mathcal{O}(n^2)$ Problems in Computational Geometry *Computational Geometry: Theory and Applications* **5 (3)** 165-185
- Gold, O., and M. Sharir (2017): Improved Bounds for 3SUM, $kSUM$, and Linear Degeneracy *Proceedings of the 25th Annual European Symposium on Algorithms* 1-13
- Gronlund, A., and S. Pettie (2014): Threesomes, Degenerates, and Love Triangles **arXiV**
- Kane, D. M., S. Lovett, and S. Moran (2018): Neat-optimal Linear Decision Trees for $kSUM$ and Related Problems *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing* 554-563
- Kopelowitz, T., S. Pettie, and E. Porat (2014): Higher Lower Bounds from the 3SUM Conjecture **arXiV**

- Patrascu, M. (2010): Towards Polynomial Lower Bounds for Dynamic Problems *Proceedings of the 42nd ACM Symposium on Theory of Computing* 603-610
- Wikipedia (2020): 3SUM