INTERNATIONAL CONFERENCE ON COMBINATORICS, CRYPTOGRAPHY AND COMPUTATION

Iran University of Science &Technology

Proceedings of the 2nd International Conference on Combinatorics, Cryptography and Computation (I4C2017)

Efficient Computation via Delayed Reorganization

Elham Mahdipour
Yazd University, PhD student in Computer Engineering
University Blvd., Safayieh, Yazd, Iran
elham.mahdipour@stu.yazd.ac.ir

Mohammad Ghasemzadeh
Yazd University, Associate Professor, Department of Computer Engineering
University Blvd., Safayieh, Yazd, Iran
m.ghasemzadeh@yazd.ac.ir

ABSTRACT

The main mission of computer science is to invent new data structures and algorithms which can solve some concerned problems effectively. Insertion and deletion are the operations defined almost on every data structure. Effectiveness of these two main operations has a high impact on performance of the data structure and its applications. In most data structures, when insertion is performed, the reorganization of that data structure is also carried out simultaneously. In this paper, we introduce a data structure that unlike most data structures, insertion is performed easily with the lowest cost, and instead, reorganization is performed when a data item is being deleted. This property lets most of the operations on this data structure could be accomplished in $O(1)$ amortized time. This characteristic has made it to become the most suitable data structure in solving some especially important problems like issues related to satellite networks, leader election, routing algorithms, data segmentation and mobile networks.

KEYWORDS: Data structure, Delayed Reorganization**,** Amortized time, Routing Algorithms.

1 INTRODUCTION

Today, in the era of Information Technology from the executive perspective, the design and the choice of data structures and algorithms are the most important process in software design and applications. It is natural to explore the space of possible designs for prevalent data structures. Doing so allows one to consider simpler alternative designs and give more intuition into whether special features of a design are essential or unessential.

In this regard, there are many data structures, such as Queues, Linked lists, Binary trees, Red-Black trees, AVL trees, Heaps and etc., which everyone uses for their applications. In this paper we introduce Fibonacci heap data structure. A Fibonacci heap is a more complicated data structure than a binary heap but may provide potential performance gains. Fibonacci heap is a deterministic data structure implementing a priority queue with optimal amortized operation costs. Fibonacci heap is an ordered collection of rooted trees that obey min-heap property.

According to the studies, research on Fibonacci heap follows one or more of the following goals. Some researchers used Fibonacci heaps because of the speed of performance for their own work; others offered solutions and algorithms to improve the function of Fibonacci heaps.

Therefore, in this paper, we introduce Fibonacci heap data structure and main operations it in Section 2. We review the recent applications of this data structure in Section 3. The results of the research are also presented in Section 4.

2 FIBONACCI HEAP

In computer science, a Fibonacci heap is a tree based data structure, which is a collections of min-heap ordered trees. This data structure developed in 1984 by Fredman and Tarjan (Cormen et al., 2009). Fibonacci heap has a better amortized running time than other priority queue data structures such as binary heap and binomial heap. The operations of Fibonacci heap as follows as (Table 1) (Cormen et al., 2009; Tiwari and Umrao, 2016):

- MAKE-HEAP, INSERT, DECREASE-KEY, UNION and MINIMUM in $O(1)$ amortized time.
- DELETE and EXTRACT-MIN in $O(\log n)$ amortize time, where n is the size of the heap.

Table 1: Comparing the amortized time of operations in Binomial heap and Fibonacci heap

Operation	Type of Heap Tree	
	Binomial Heap	**Fibonacci Heap**
MAKE-HEAP	$O(1)$	$O(1)$
INSERT	$O(\log n)$	$O(1)$
DECREASE-KEY	$O(\log n)$	$O(1)$
UNION	$O(n)$	$O(1)$
MINIMUM	$O(1)$	$O(1)$
DELETE	$O(\log n)$	$O(\log n)$
EXTRACT-MIN	$O(\log n)$	$O(\log n)$

Fibonacci heap is used in priority queues; it can improve the running time important algorithms, such as Dijkstra's algorithm for computing the single-source-shortest path, and prim algorithm for computing minimum spanning tree.

2.1 Structures of Fibonacci Heap

A Fibonacci Heap Structure is a set of a forest of trees. The structure of each node in the Fibonacci heap such as follow where H is a Fibonacci heap and roots of the rooted trees are linked to form a doubly linked list termed as Root list (Cormen et al., 2009):

- One pointer (*x.p*) to the parent key
- One pointer (*x.chid*) to any one of offsprings. The offsprings of x are linked together in a circular, doubly linked list, which call the *child list* of x.
- *y.left* and *y.right* pointers are y's left and right siblings, respectively, for each child y in a child list.
- *x.degree* point to the number of offsprings in the child list of node x.
- *x.mark* is a Boolean valued attribute that indicates whether node x has lost a child since the last time x was made the child of another node.
- One pointer (*H.min*) to the root of the tree with a minimum key in Root list.
- One pointer (*H.n*) that show the number of nodes currently in H.

Using of doubly linked list in the Fibonacci heap cause to first node can remove from a circular in $O(1)$ time and concatenated two lists in $O(1)$ time (Tiwari and Umrao, 2016).

2.2 Operations in Fibonacci Heap

Generally used of MAKE-FIB-HEAP() procedure to create an empty Fibonacci heap so-called as H.

For insert node x into a Fibonacci heap H, used of FIB-HEAP-INSERT() procedure; assuming that the node has been memory allocated already and that x.*key* has already been filled in (Cormen et al, 2009). Insert a node has three steps:

1. Initialize pointers for new node such as *x.degree*= 0, *x.p* = NIL, *x.child* = NIL, *x. mark* = FALSE.
2. Insert *x* into the H's root list and update *H. min* if necessary.
3. Increment the nodes number in the Fibonacci heap (*H.n*) to return the additions of the new node.

```
FIB-HEAP-INSERT(H ,x)
    x.degree= 0;
    x.p = NIL;
    x.child = NIL;
    x.mark = FALSE;
    if (H.min == NIL)
            create a root list for H containing just x;
            H.min = x;
    else
          insert x into H's root list
          if (x.key < H.min.key)
                  H.min= x;
     H.n = H.n + 1;
```

The FIBO-HEAP-MIN (H) procedure finds the minimum node of a Fibonacci heap H is given by the pointer *H.min*.

FIB-HEAP-UNION() procedure concatenated the root lists of H1 and H2 and then determines the new minimum node. This operation, called as union Fibonacci heaps H1 and H2, destroying H1 and H2 in the process. The union operation has three steps as follow:

1. Union the root lists;
2. Set up a new *H.min* = min {*H1.min*, *H2.min*};
3. Reset *H.n* = *H1.n* +*H2.n* and return H;

The process of extracting the minimum node is the most complex operation. Delayed work of consolidating trees in the root list is finally occurring by this operation, which is divided into two steps:

1. Remove the minimum and add every child to the root list.
2. Consolidate the trees with the same degree (Cormen et al, 2009).

```
CONSOLIDATE(H)
  Let A[0] ..D(H.n) be a new array
  for i = 0 to D(H.n)
      A[i]=NIL;
      for each node w in the root list of H
        x =w;   d = x.degree;
        while A[d]≠ NIL
          y = A[d]  // another node with the same degree as x
          if (x.key > y.key)  exchange x with y;
          FIB-HEAP-LINK(H, y, x);// remove y from the root list of H, make y a child of x, incrementing x.degree.
               A[d]= NIL;   d = d +1;
        A[d]= x;
  H.min = NIL;
  for i = 0 to D(H.n)
      if (A[i]≠ NIL)
        if (H.min == NIL)  create a root list for H containing just A[i]; H.min = A[i];
        else insert A[i] into H's root list;
             if (A[i].key < H.min.key)   H.min = A[i];
```

The FIB-HEAP-DECREASE-KEY() procedure used for the decrease key operation.

The FIB-HEAP-DELETE() procedure deletes a node from an n-node Fibonacci heap and assumes that there is no key value of -infinity currently. This procedure decreases key x to –infinity firstly, then extract a minimum of Fibonacci heap.

3 FIBONACCI HEAP APPLICATIONS

In this section we review Fibonacci heap applications in recently years.

Gueunet et al. presented a new algorithm for the fast, shared memory multi-core computation of augmented merge trees on triangulations (Gueunet et al., 2017). In contrast to most subsisting parallel algorithms, their technique computes augmented trees. This augmentation is required to enable the full extent of merge tree based applications, including data segmentation. Their approach completely revisits the traditional, sequential merge tree algorithm to re-formulate the computation as a set of independent local tasks based on Fibonacci heaps. This results in superior time performance in practice, in sequential as well as in parallel thanks to the OpenMP task runtime. In the context of augmented contour tree computation, they show that a direct usage of their merge tree procedure also results in superior time performance overall, both in sequential and parallel. They reported performance numbers that compare their approach to reference sequential and multi-threaded implementations for the computations of augmented merge and contour trees. These experiments demonstrated the runtime efficiency of their approach as well as its scalability on common workstations. They demonstrated the efficiency of their approach in data segmentation applications. They also provided a lightweight VTK-based C++ implementation of them approach for reproduction purposes.

The Hollow Heap data structure proposed with the same amortized efficiency as the Fibonacci heap (Hansen et al., 2017). All heap operations except delete and delete-min take $O(1)$ time, worst case as well as amortized; delete and delete-min take $O(\log n)$ amortized time on a heap of n items. Hollow heaps are the simplest structure to achieve these limits. Hollow heaps combine two novel ideas: the use of lazy deletion and re-insertion to do decrease-key operations and the use of a dag (directed acyclic graph) instead of a tree or set of trees to represent a heap.

Tiwari and Umrao used on a Fibonacci heap structure to find the leader in lesser amount of time and messages (Tiwari and Umrao, 2016). Their main challenge is to find the new leader in lesser time with a minimum number of message communications in Mobile Ad hoc network. Mobile Ad hoc network is a self-configured network of devices connected using a wireless medium. Ad hoc network is a temporary network connection created for a specific purpose. MANET can be seen as a distributed computing environment, where Leader Election mechanism is used, for the purpose of synchronization. Election algorithms are used to find the leader for Distributed System. The better time complexity of operations using Fibonacci heap structure makes it suitable for the leader election in Mobile Ad Hoc Network, as compared to other tree structures. Fibonacci heap results good by their advanced time complexities. The design of Fibonacci heap suits on wireless networks, and especially on Mobile Ad Hoc Networks. Where network changes dynamically, and Fibonacci heap give the efficient way to change the structure in amortized time.

The problem of Fibonacci heaps is that they must maintain a “mark bit” unfortunately, that serves only to make sure efficiency of heap operations incorrectly. Karger conjectured that this data structure has expected amortized cost $O(\log s)$ for delete-min, where s is the number of heap operations. Li et al. give a tight analysis of Karger’s randomized Fibonacci heaps, resolving Karger’s conjecture (Li and Peebles, 2015). Specifically, they obtain matching upper and lower bounds of $O(\log^2 s / \log\log s)$ for the runtime of delete-min. They also prove a tight lower bound of $\Omega(\sqrt{n})$ on delete-min in terms of the number of heap elements n. The request sequence used to prove this bound also solves an open problem of Fredman on whether cascading cuts are necessary. Finally, they give a simple additional modification to these heaps which yields a tight runtime $O(\log^2 n / \log\log n)$ for delete-min.

Qu et al. proposed a fast Isomap algorithm based on Fibonacci heap (Qu et al., 2015). For the slow operational speed problem of Isomap algorithm in which the Floyd-Warshall algorithm is applied to finding shortest paths, an improved Isomap algorithm is proposed based on the sparseness of the adjacency graph. In the improved algorithm, the runtime for shortest paths is reduced by using Dijkstra's algorithm based on Fibonacci heap, and thus the Isomap operation is speeded up. The experimental results on several data sets show that the improved version of Isomap is faster than the original one.

Kaplan et al. explore the design space of the Fibonacci heap data structure and proposed a version with the following improvements over the original (Kaplan et al., 2014): (i) Each heap is represented by a single heap-ordered tree, instead of a set of trees. (ii) Each decrease-key operation does only one cut and a cascade of rank changes, instead of doing a cascade of cuts. (iii) The outcomes of all comparisons done by the algorithm are explicitly represented in the data structure, so none are wasted. They also give an example to show that without cascading cuts or rank changes, both the original data structure and the new version fail to have the desired efficiency, solving an open problem of Fredman. Finally, they illustrated the richness of the design space by proposing several alternative ways to do cascading rank changes, including a randomized strategy related to one way proposed by Karger. They leave the analysis of these alternatives as intriguing open problems.

Mozes et al. combined two techniques for efficiently computing shortest paths in directed planar graphs (Mozes et al., 2014). The first is the linear-time shortest-path algorithm and the second is Dijkstra's algorithm on the dense distance graph (DDG). They modified Fakcharoenphol and Rao's FR-Dijkstra algorithm and the Monge heap data structure. They used Fibonacci heap in implementations and decrease all costs of modifying FR-Dijkstra to $O(1)$. They developed new techniques that would lead to faster, possibly linear-time, algorithms for problems such as minimum-cut, maximum-flow, and shortest paths with negative arc lengths. As immediate applications, they show how to compute maximum flow in directed weighted planar graphs in $O(n \log p)$ time, where p is the minimum number of edges on any path from the source to the sink. They also show how to compute any part of the DDG that corresponds to a region with r vertices and k boundary vertices in $O(r \log k)$ time, which is faster than has been previously known for small values of k.

Wang et al. proposed a novel routing algorithm design of time evolving graph based on pairing heap for MEO satellite network (Wang et al., 2014). As a tradeoff of GEO and LEO, MEO satellite system has more acceptable service performance and it is more appropriate to provide global mobile communications. A MEO satellite system model communicating according to time slots is constructed in the paper. Moreover, in order to improve comprehensive performance of the network, a novel routing algorithm applying Time evolving graph based on pairing heap is proposed. The time evolving graph is employed to analysis the dynamic topology of the network and the pairing heap is applied in the Dijkstra algorithm to reduce the time complexity. By contrast, Fibonacci heap is also used to optimize Dijkstra algorithm. Finally simulation results show that routing algorithm applying time evolving graph based on pairing heap can perform better and reduce the time complexity obviously, and at the same time, pairing heap works better than Fibonacci heap when the number of nodes grows bigger.

Chan proposed Quake Heaps that are a data structure as theoretical performance as Fibonacci heaps (Chan, 2013). Quake Heaps supported decrease-key operations in $O(1)$ amortized time and delete-min operations in $O(\log n)$ amortized time. The data structure is simple to explain and analyze, and may be of pedagogical value.

Jain and Sharma presented a novel approach towards leader election using Fibonacci heap by electing minimum node as leader (Jain and Sharma, 2012). Fibonacci heaps offer a good example of data structure designed with amortized analysis in mind. However the resulting structure is a little complicated, but it can be made useful in practical cases of leader election algorithms. The design is suitable in wireless networks in spite of the fact that they are unstable and prone to faults. They concluded that higher the system is immune to the faults, the better our design works. The lesser complexity in message passing exhibited by this method had been justified through obtained simulation results.

Jekovec et al. give a brief overview of 4 comparison-based priority queues published before 1993: the original binary heap, Binomial heap, Fibonacci heap and the Weak-Heap (Jekovec et al., 2012). The review focuses on the theoretical worst-case, average-case and amortized time complexity of element comparisons and justifies it by providing necessary implementation details. Their review results show that the Fibonacci heap with its lazy melding and cascading cut is the best general-purpose comparison-based heap. It achieves the best amortized performance in comparison to the binary heap or the binomial heap. Weak heap on the other hand is specialized on fast inline sorting and is the fastest comparison-based heap sort to date.

Brodal et al. presented the first pointer-based heap implementation with time bounds matching those of Fibonacci heaps in the worst case (Brodal et al., 2012). They support make-heap, insert, find-min, meld and decrease-key in worst-case $O(1)$ time, and delete and delete-min in worst-case $O(\log n)$ time, where n is the size of the heap. The data structure uses linear space. A previous, very complicated, solution achieving the same time bounds in the RAM model made essential used of arrays and extensive used of redundant counter schemes to maintain balance. Their solution used neither. Their key simplification is to discard the structure of the smaller heap when doing a meld. They used the pigeonhole principle in place of the redundant counter mechanism. All these variations can be combined, each solution implying different bounds on the maximum degrees, constants in the time bounds, and complexity in the reduction transformations. In the presented solution they aimed at reducing the complexity in the description, whereas the constants in the solution were of secondary interest.

Liu et al. introduced the dijkstra algorithm based on the Fibonacci heap (Liu et al., 2011). Compared to traditional dijkstra algorithm, the main advantage of this algorithm is that it has a good time bound that can efficiently work out the shortest path between two points. Under the circumstance that algorithm efficiency has been guarantee, the satellites and connections are turned into a directorial graph as the input of the algorithm in the Walk constellation topology program. In such case, a static routing program with smaller communication overhead and processing power to work out the optimal path is obtained, which adapts to the rapid dynamic changes of the satellite topology network. This routing program has been simulated and it turns out that the performance fully meets the design requirements. But the program also has many deficiencies, such as it requires a considerable amount of system overhead.

The application of the data structure in the external memory algorithms is studied for the need of using external memory algorithms in the computer programming (Li et al., 2011).According to the Fibonacci heaps' characteristics in internal memory, a new data structure for external memory algorithm is proposed. Then, the time complexity of its operations is analyzed. It is proved that the operations can be finished with unit number of page transfers except for the operations of delete-min and decrease-key. Finally, the feasibility and availability of the data structure is illustrated by the application of Fibonacci heap in the Dijkstra algorithm.

Elmasry proposed a priority queue that achieves the same amortized bounds as Fibonacci heaps (Elmasry, 2010). They claim that resolved this issue by introducing a priority queue that called as violation heap. The Violation heaps had the same amortized bounds as Fibonacci heaps, and expected to perform in practice in a more efficient manner than other Fibonacci-like heaps and compete with pairing heaps. Namely, find-min requires $O(1)$ worst-case time, insert, merge and decrease-key require $O(1)$ amortized time, and delete-min requires $O(\log n)$ amortized time. The main idea behind their construction is to propagate rank updates instead of performing cascaded cuts following a decrease-key operation, allowing for a relaxed structure.

4 CONCLUSION

In computer science, the designing and selecting of data structures and algorithms is the most important process in software design and applications from an operational point of view. In this paper, we introduced an efficient data structure called as the Fibonacci heap. The Fibonacci heap data structure developed in 1984 by Fredman and Tarjan. The Fibonacci heap data structure that unlike most data structures, insertion is performed easily with the lowest cost, and instead, reorganization is performed

when a data item is being deleted. It is a type of heap that supports union, insert, decrease-key and find the minimum in $O(1)$ amortized time, also extract-min and delete in $O(\log n)$ amortized time. In general, we can use the Fibonacci heap to improve the speed of the algorithm, because the delayed reorganization characteristic transforms it into the most appropriate data structure to solve some of the most important problems. According to this study, some researchers used of Fibonacci heap for satellite networks, leader election, routing algorithms, data segmentation, mobile networks, and so on. Also other researches improve the Fibonacci heap structure and increase the speed of the algorithm, or defined a new data structure same as Fibonacci heap.

REFERENCES

Brodal, Gerth Stølting, George Lagogiannis, and Robert E. Tarjan. (2012). "Strict fibonacci heaps". In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pp. 1177-1184. ACM.

Cormen Thomas H., Leiserson Charles E., Rivest Ronald L. and Stein Clifford. "Introduction To Algorithms", Third edition, *The MIT Press Cambridge*, Massachusetts. ISBN 978-0-262-03384-8, pp: 505-530. (2009).

Chan, Timothy M. "Quake heaps: a simple alternative to Fibonacci heaps." In *Space-Efficient Data Structures, Streams, and Algorithms*, pp. 27-32. Springer Berlin Heidelberg, (2013).

Elmasry, Amr. (2010). "The violation heap: A relaxed Fibonacci-like heap." *Discrete Mathematics, Algorithms and Applications* 2, no. 04 (2010): 493-503.

Gueunet, Charles, Pierre Fortin, Julien Jomier, and Julien Tierny. (2017). "Task-based augmented merge trees with Fibonacci heaps." In *IEEE Symposium on Large Data Analysis and Visualization 2017*.

Hansen, Thomas Dueholm, Haim Kaplan, Robert E. Tarjan, and Uri Zwick. "Hollow heaps." *ACM Transactions on Algorithms (TALG)* 13, no. 3 (2017): 42.

Jain, Arihant Kumar, and Sharma Ramashankar. (2012). "Leader Election Algorithm in Wireless Environments Using Fibonacci Heap Structure." *International Journal of Computer Technology and Applications* 3, no. 3.

Jekovec, Matevz, and Andrej Brodnik. (2012). "Review of 4 comparison-based priority queues: Binary, Binomial, Fibonacci and Weak heap: Technical report LUSY-2012/01."

Kaplan, Haim, Robert E. Tarjan, and Uri Zwick. (2014). "Fibonacci heaps revisited." *arXiv preprint arXiv:1407.5750*.

Li, Jerry, and Peebles John. "Replacing mark bits with randomness in Fibonacci heaps." In *International Colloquium on Automata, Languages, and Programming*, pp. 886-897. Springer, Berlin, Heidelberg, (2015).

Liu, Yanyun, Wenquan Feng, Hua Sun, Jia Yin, and Zhiyuan Zheng.(2011). "An algorithm design of inter-satellite routing based on Fibonacci Heap." In *Computational Intelligence and Design (ISCID), 2011 Fourth International Symposium on*, vol. 2, pp. 51-54. IEEE.

Li Peng, Zhang Yuan-ping, Li Li. (2011). "Fibonacci heap and its application in external memory algorithms", Computer Engineering and Design.

Mozes, Shay, Yahav Nussbaum, and Oren Weimann. (2014). "Faster shortest paths in dense distance graphs, with applications." *arXiv preprint arXiv:1404.0977*.

Qu, Taiguo, and Zixing Cai. (2015). "A Fast Isomap Algorithm Based on Fibonacci Heap." In *International Conference in Swarm Intelligence*, pp. 225-231. Springer, Cham.

Tiwari Kshama and Umrao Brajesh Kumar. (2016). "Leader Election Algorithm using Fibonacci Heap Structure in Mobile Ad hoc Network." *IJCA Proceedings on Technical Symposium on Emerging Technologies in Computer Science*, TSETCS 2016(2):5-8.

Wang, Yupeng, Gong Zhang, Zhuqing Jiang, Chengkai Huang, Xueyang Wang, Aidong Men, Bo Yang, and Kaifeng Qi. (2014). "A novel routing algorithm design of time evolving graph based on pairing heap for MEO satellite network." In *Vehicular Technology Conference (VTC Fall), 2014 IEEE 80th*, pp. 1-5. IEEE.