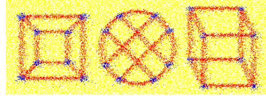
Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 28, no. 1, pp. 27–50 (2024)
DOI: 10.7155/jgaa.v28i1.2926

Linear-algebraic implementation of Fibonacci heap

Danila Demin[1] *Dmitry Sirotkin*[2] *Stanislav Moiseev*[2]

[1]Coleman Services
[2]Huawei Technologies Co.

Submitted: 28 Mar 2023 Accepted: April 2024 Published: May 2024

Article type: Regular Paper Communicated by: U. Brandes

Abstract. In the paper, we demonstrate that modern priority queues can be expressed in terms of linear algebraic operations. Specifically, we showcase one of the arguably most asymptotically faster known priority queues — the Fibonacci heap. By employing our approach, we prove that for the Dijkstra, Prim, Brandes, and greedy maximal independent set algorithms, their theoretical complexity remains the same for both combinatorial and linear-algebraic cases.

1 Introduction

Graphs are a representation of data, which naturally emerges in many different fields of study, from data analytics and route planning to bioinformatics and linguistics. Currently, it's not uncommon for them to consist of millions or even billions of vertices, so we need efficient and scalable algorithms. One of the approaches, used in industry is to represent and work with graphs in terms of linear algebra, which is generally handled by modern hardware well.

The central framework for this task is called GraphBLAS [10]. According to C. Yang, A. Buluç, and J. D. Owens [18], GraphBLAS is a high-performance linear algebra-based graph framework for the GPU that is aimed at achieving the two following goals. The central one is the *performance portability*, i.e. graph algorithms need no modification to have high performance across hardware. The second one is the *concise expression*, which means simplicity of graph algorithms expression. *High-performance* and *scalability* is required to achieve state-of-the-art performance for both small-scale and exascale types of graphs.

GraphBLAS reaches these goals for well-known graph algorithms such as Breadth-first-search [9], PageRank [8], Connected components [14], and Triangle counting [2]. For a wider list of linear algebraic algorithms, see the book [7]. The main idea is to express graph algorithms using linear algebraic operations such as matrix-to-vector or matrix-to-matrix multiplication. For some algorithms, however, their development requires the usage of structures, which do not intrinsically

E-mail addresses: demin.danila-03@yandex.ru (Danila Demin) dmitriy.v.sirotkin@gmail.com (Dmitry Sirotkin) stanislav.moiseev@gmail.com (Stanislav Moiseev)

This work is licensed under the terms of the CC-BY license.

map to linear-algebraic operations. As an example, in [15] authors develop a depth-first search algorithm emulating a stack data structure with linear-algebraic operations.

In 2008 E. Robinson and J. Kepner [13] expressed the Brandes algorithm (see [3]) for the unweighted case in linear algebraic operations and achieved the same time complexity as the combinatorial version of this algorithm. For the weighted case, A. Tumurbaatar and M. J. Sottile [17] in 2021 proposed a linear algebraic algorithm, but it requires solving a single source shortest path (SSSP) problem, and there are no algorithms in GraphBLAS that reach the best-known time complexity problem. For example, Bellman-Ford [1, 5] or Delta-stepping [11] algorithms can be used for solving SSSP problem instead of the Dijkstra algorithm, but they require $O(nm)$ and $O(n + m + d \cdot L)$ time respectively under certain assumptions instead of universal $O(m + n \log n)$ for Dijkstra. Here n is the number of vertices in the graph, m is the number of edges in the graph, d is the maximal degree of vertices, L is the length of the longest of shortest paths from s to any other vertex.

Unfortunately, there are difficulties in the implementation of some algorithms like Dijkstra, Prim [12], Greedy maximal independent set, and Brandes for the weighted case. All of these algorithms require a specific *priority queue* — an additional data structure that can decrease the value in the array (`decreaseKey`) and extract the minimal element from it (`extractMin`). The naive approach requires $O(n)$ time to recalculate a minimal element each time we need to extract it. Up to date, many such structures that reach a better complexity are known. One of them is a Fibonacci heap which requires $O(1)$ time amortized for `decreaseKey` and $O(\log n)$ for `extractMin` (see [4] and [6]).

In this paper, we develop a linear algebraic version of the Fibonacci heap. Our priority queue allows us to express Dijkstra, Prim, Greedy maximal independent set, and Brandes algorithms for the weighted case in terms of GraphBLAS, and for each of them allows us to achieve the best-known time complexity.

Operation	Combinatorial version	Linear algebraic version
Decrease key	$O(1)$	$O(1)$
Extract min	$O(\log n)$	$O(\log n)$
Insert	$O(1)$	not supported
Merge	$O(1)$	not supported

Table 1: Amortized time complexity of Fibonacci heap operations

In Section 2, we will give a brief description of the Fibonacci heap. We will not discuss the dynamic operations in this data structure because for previously mentioned algorithms only `decreaseKey` and `extractMin` are required. In Section 3, we will discuss the linear algebraic method in general and also the idea of *mapping matrices* which will be used many times throughout this paper. The operations of our priority queue will be described in Section 4. For some functions, complexity will be analyzed in the same section, but the general proof of required time complexity will be given in the Section 5. In Section 6 we will discuss previously mentioned algorithms for which our data structure allows us to achieve better time complexity.

2 Combinatorial version of Fibonacci heap

In this section, we give a brief introduction to the Fibonacci heap data structure.

The Fibonacci heap is a data structure, which contains key-values on the nodes of a family of rooted trees. Each tree satisfies the min-heap property (i. e. every child has a greater or equal value with its parent). Each node is structured in a such way as to contain a key-value, pointers to parent, child, left and right sister nodes, its degree, and a boolean value called `marked` that indicates if the node had lost a child node since it became a child of another node. Therefore, the children of each node are connected as a cycle with the left and right sister node pointers. Additionally, the roots of these trees are also connected in the cycle by the left and right sister pointers. In addition, the Fibonacci heap data structure contains the number of nodes and a pointer to the root with the minimum value.

In the general case, Fibonacci heaps support the following operations — `findMin`, `deleteMin`, `insert`, `decreaseKey` and `merge`. For the purposes of the paper, we do not implement `insert` and `merge` operations for two reasons. The former is that they are not needed for the implementation of considered algorithms. The latter is that in a linear-algebraic implementation, these operations require vector concatenation, which is generally not supported.

The `decreaseKey` operation decreases the value of node v to some x, with the value of x being lower than the current value of v. After such a decrease, the min-heap property can be violated. In this case, we start so-called *cascading cut* from this node. We cut this vertex from its parent and make it a root. The parent of its node has lost one child as a result. If it had lost any other child before this operation (i.e. it was marked), we also cut this node from its parent in the same way. We continue this process until we reach the unmarked vertex.

In Figure 1 there is an example of cascading cut.

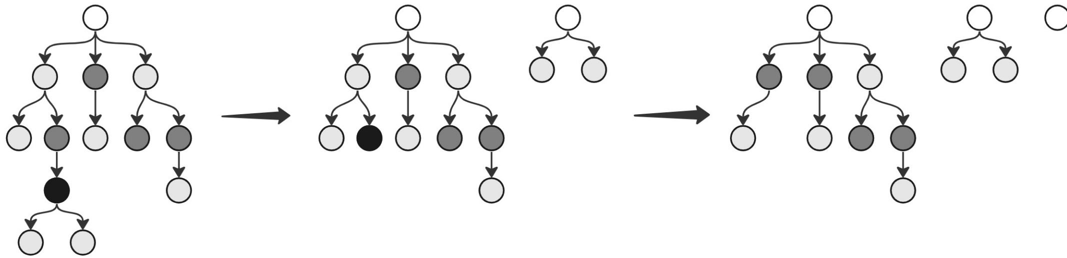

Figure 1: Cascading cut visualization. White vertices are roots, light gray vertices are unmarked, dark gray vertices are marked and black vertices are the ones, which are being cut at the corresponding step.

The `extractMin` operation identifies the node with the minimal value and deletes it from the list of nodes. Because of the min-heap property, the node with minimal value should be a root. During the operation, this root is removed, and therefore all of its children become roots themselves. After that, a so-called `consolidate` operation is performed, which merges some of the trees, presented in the heap together. In particular, we make a list of nodes A with the following property — a node in the position d in the array (e.g. in the $A[d]$) should be both the root and have a degree d. We add roots of the current Fibonacci heap to this list one by one. For each new root x with some degree d, we check if there exists a root y of the same degree (and therefore is the same position) in the list A. If there is no such root, we place node x in $A[d]$. Otherwise, if there exists a root y with the same degree as x and it is already present in A, we choose from x and y a vertex with a lesser key and make the one with a bigger key its child. This results in the appearance of the vertex with the degree on one more than x had, after which we check if $A[d + 1]$ is empty. If it

is so, we place a node in this position in the array. Otherwise, we perform we similar merge and check the $A[d+1]$. We repeat this process until we find an empty position for the node. After all the nodes have been placed in A, the `consolidate` operation is finished.

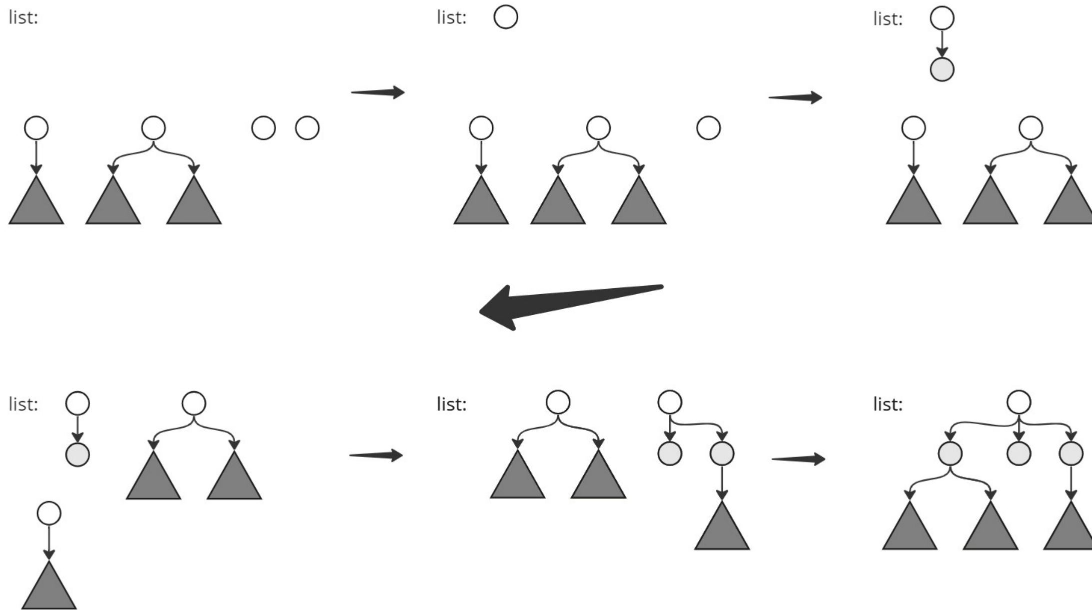

Figure 2: `Consolidate` operation visualisation. White vertices are roots, light gray vertices are unmarked, and triangles are subtrees.

Operations `decreaseKey` and `extractMin` need $O(1)$ and $O(\log n)$ amortized time respectively. It is proved by defining the following potential:

$$\Phi = t + 2m \tag{1}$$

Here variable t refers to the number of trees in the heap and variable m refers to the number of marked vertices. We assume that a unit of potential can be used to do the operations, requiring constant time. It can be proved, that both operations `decreaseKey` and `extractMin` require the amount of both time and potential to take $O(1)$ and $O(\log n)$ amortized time, respectively. In Section 5 similar proof for the linear algebraic version of the Fibonacci heap is presented.

An in-detailed description of the Fibonacci heap data structure can be found in the well-known book [4].

3 Linear algebraic approach

3.1 Sparse vectors and matrices

For real-world graphs, though they can be very different in structure, most of them have one property in common — they are sparse. By that we mean, that the number of edges is somewhat

proportional to the number of nodes. In other words, $m = n^{1+o(1)}$, where n is the number of nodes in the graph and m is the number of edges.

For each graph G, there is a matrix $A(G)$ of dimension $|V| \times |V|$, where each element a_{ij} equals one if there is an edge between vertices i and j, and zero otherwise. This matrix is called the *adjacency* matrix of the graph G. It allows the expression of some graph operations in linear algebraic terms. For example, the i-th component of the vector $A(G)v$ (for a vector v that consists of zeros or ones only) corresponds to the number of neighbors of i such that v_i are non-zero.

Let v be a vector of dimension $|V|$ that consists of zeros and ones. In this paper, we say that vertex i is *included* in v if the i-th component of this vector equals one. Each vector of zeros and ones can be considered a characteristic function of some subset of vertices. So, the word *included* is used in relation to a vertex (or basis vector) and vector means that the vertex included in the set corresponds to the given vector.

3.1.1 Sparse matrix storage format

In practical implementations, sparse $m \times n$ matrix A can be stored in memory in several ways. In this paper, we are using a so-called CSC (compressed sparse by column) format, which was first proposed in [16]. This format uses three arrays `colstart`, `row`, and `value` stored in the memory. Array `value` of size $|M|$ contains values of nonzero elements in the matrix in column-first lexicographic order, i.e. a_{ij} is before a_{kl} if $j < l$ or $j = l$ and $i < k$. Array `row`, also of size $|M|$, contains a row for each of the elements in matrix A in the same order as in `value`. Array `colstart` has length n. The i-th entry in `colstart` indicates the position of the element in the arrays `value` and `row` that appears to be the first non-zero element in the i-th column.

3.1.2 Hyper-sparse matrix storage format

A sparse $m \times n$ matrix M is called *hyper-sparse* if $|M| = o(n)$ [7]. For a family of hyper-sparse $m \times n$ matrices, the CSC format becomes inefficient in the same way as the dense format is inefficient for matrices with $o(mn)$ nonzero elements. In this case, we will assume that the matrix is stored in memory as a list of all nonzero elements with its position in the matrix ordered in column-first lexicographic order. We will call this format — *coordinate format*. Coordinate format with $|M|$ nonzero elements requires $O(|M|)$ memory unlike $O(|M| + n)$ for CSC format.

In this paper, we use both storage formats discussed above for sparse and hyper-sparse matrices for purposes of complexity analysis.

3.1.3 Implementation considerations

Assume we have a matrix A of size $m \times n$ and a vector v of size n, which are stored as two-dimensional and one-dimensional arrays, respectively. For the simplest implementation, we will require $O(mn)$ time to find the result of their multiplication Av. However, if both A and v are sparse, this method of multiplication isn't efficient, since it makes many multiplications, which contain at least one zero. So, for the sparse vectors and matrices, other ways of storing them in memory are more suitable.

We assume that a *sparse* vector v stored in memory as a sorted array of pairs $(i, v[i])$, where i is an index and $v[i]$ is the value of vector in i-th position, we additionally assume that pairs $(i, v[i])$ are sorted in the array by increasing of i. Values on positions that are not listed in the array are assumed to be zero. The number of nonzero elements in vector v and in the array is referred to as $|v|$ in this paper. For matrix M, a notation $|M|$ is used. Non-sparse (e.g. *dense*) are stored as an

array as per normal. As $supp(v)$, we define a sparse vector stored as an array of pairs $(i, 1)$ such that for each i, the pair $(i, v[i])$ is contained in v. Thus, the purpose of $supp(v)$ is to denote the set of indices of non-zero elements of v. In case a vector v contains zeros and ones only, $supp(v) = v$. In this paper, all vectors and matrices are meant to be sparse unless their density is explicitly stated.

For a sparse $m \times n$ matrix A (stored in the CSC format), a sparse vector v, and a dense vector u the in-place operation $u+ = Av$ can be performed in $O(d \cdot |v|)$, where d is the biggest number of nonzero elements in columns of A [18]. Additionally, a sparse vector v can be added in-place to a dense vector u in $O(|v|)$ time. The sum of two sparse vectors v and u can be computed in $O(|v| + |u|)$ time. Any vector $A^T v$ (contained as a sparse vector) with the sparse matrix A and dense vector v can be computed in $O(|A| \log |A|)$ time.

In this paper, we will call $diag(v)$ a diagonal matrix V, which is a matrix with the elements of v on its diagonal. More formally, this is a matrix that satisfies the following condition: $V[i, i] = v[i]$ and $V[i, j] = 0$ for all $i \neq j$. It obviously can be produced in coordinate format from a sparse vector in $O(|v|)$. In addition, we will use binary operations $\geq, \leq, =, \neq, \min$ on vectors of integers. Let u and v be vectors of the same dimension, in which i-th components are u_i and v_i respectively. We define an element-wise $\geq$ operation as follows: the i-th component of the vector $u \geq v$ equals one if $u_i \geq v_i$, and zero otherwise. Other vector operations are defined similarly as the element-wise application of corresponding operations.

3.2 Mapping matrices

For the purposes of providing a linear-algebra-based representation of pointers in the Fibonacci heap, we will need the following definition:

Definition 3.1 *We call a mapping matrix the matrix of zeros and ones such that there exists a single* 1 *element in every column.*

For the mapping matrix F and a sparse vector v, we can compute Fv as a sparse vector in $O(|v| \log |v|)$ time. We will call a vector $e_i = (0, \ldots, 1, \ldots, 0)$ with a single 1 element in i-th position a *basis vector*. It's easy to spot that a result of multiplication F on a sparse basis vector e_i is a sparse basis vector $e_{f(i)}$ that can be computed in $O(1)$ time. Here f is a mapping from $[n] = \{1, \ldots, n\}$ to $[m]$ defined by a mapping matrix F of size $n \times m$.

We use mapping matrices in correspondence with various pointers in the combinatorial Fibonacci heap. Let us assume, there exists some set of pointers family $\mathcal{F}$ that contains for every node in the graph a single pointer to some node (maybe itself). This way it generates some mapping f. Every mapping f can also be generated by a mapping matrix F. Therefore, every such set of pointers corresponds to a unique mapping matrix.

For the purposes of this paper, all mapping matrices are stored in the CSC format. Since every column contains a single nonzero element, all elements of the matrix are listed in array `values` in accordance with their column number. The addition of the matrix in CSC format in the general case is a complex operation in general, but it can be completed faster if it converts a mapping matrix to a mapping matrix. To be more precise, if we add a matrix A to the mapping matrix F and the resulting matrix $F + A$ is also a mapping matrix, then this addition requires $O(|A|)$ time.

The change of a pointer for a vertex v corresponds to a change of the image $f(v)$. In terms of matrix F modification, it corresponds to a change of the column which corresponds to the position of the current pointer of v. Let vector e_v be the basis vector that corresponds to the source of the

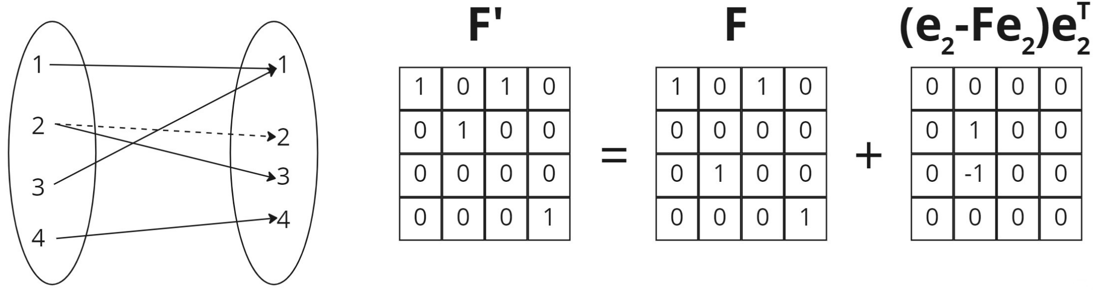

Figure 3: An example of mapping matrix transformation. The matrix F' maps e_2 to e_2 whereas F maps e_2 to e_3. The matrix $(e_2 - Fe_2)e_2^T$ changes the image of e_2 from e_3 to e_2.

existing pointer from v and e_u be a vector that corresponds to the destination of the new pointer from v to u. The new matrix F' can be obtained through the following formula:

$$F' = F + (e_u - Fe_v)e_v^T.$$

This recomputation changes a single element from arrays `row` and `value` and can be computed in $O(1)$ time. Let us check that this formula is correct. Matrix F' is still a mapping matrix because it hasn't changed in any column except for v-th. In v-th, we remove one by the term $-Fe_ve_v^T$ and a new one by the term $e_ue_v^T$. Let f' be a mapping given by the matrix F'. Since F' coincides with F on each column except v-th, $f'(w) = f(w)$ for all vertices $w \neq v$. For vertex v, we have $f'(v) = u$ from the following calculation:

$$F'e_v = (F + (e_u - Fe_v)e_v^T)e_v = Fe_v + (e_u - Fe_v)e_v^Te_v = Fe_v + e_u - Fe_v = e_u.$$

Let F and G be the mapping matrices and s be some vector of zeros and ones. Matrices F and G are interpreted as adjacency matrices of the corresponding graphs and vector s — as the mask. Those entities will be used to construct a joint graph out of two graphs during Fibonacci heap operations. For the purposes of quick pointer changes, we aim to construct a matrix F', which acts like F on basis vectors that correspond to zeros in s and like G on basis vectors that correspond to ones in s. Let us paraphrase it in terms of corresponding mappings f and g defined by the matrices F and G, respectively. We want to construct a mapping f' such that $f'(v) = f(v)$ if the v-th position in vector s is zero, and $f'(v) = g(v)$ if the v-th position in vector s is one. This matrix is defined by the following formula:

$$F' = F(E - diag(s)) + G \cdot diag(s) = F + (G - F)diag(s),$$

where E is the identity matrix, $diag(s)$ is the diagonal matrix with elements of the vector s on its diagonal. The first term in summation corresponds to the positions, where zeros are placed in s, while the second one corresponds to the positions with ones. Therefore $F'e_i = Fe_i$ if e_i is included in the decomposition of v with a zero coefficient and $F'e_i = Ge_i$ otherwise. For the case when F' replaces F in all of the positions, we need to change $O(|v|)$ (with $|v|$ being a number of nonzero elements in vector v) values in arrays `values` and `row`. So, this operation doesn't need to copy F and can be completed in $O(|v|)$ time.

For the purposes of matrix F' computation, we in general case don't have to know both matrices F and G. Due to the nature of matrix $diag(s)$, only those columns in F, which correspond to zeros in the diagonal of $diag(s)$ are needed. The same principle is true for the matrix G (with ones instead of zeros).

If we need to set up a partial mapping, we can define a corresponding matrix with zeros in the columns for which the image isn't defined. Since in mapping matrices, the number of nonzero elements in each column is constant, we can modify a mapping matrix in place, without using extra memory for the copying. We can store partial mappings in the same way as mappings by containing one zero for each column, where the mapping is not defined (unlike classic CSC where all zeros are omitted). Therefore, one element per column will be allocated in memory as normal in CSC format. The resulting zero elements are stored in memory and therefore are considered in complexity analysis.

3.3 Memory model

In this section, we introduce the computational model which we will use for the purposes of algorithm construction. Types of variables are given in Table 2. For each type, initialization from corresponding arrays requires time proportional to the used memory. Same stands for object copying. The notions of vector density, vector sparsity, and CSC matrices were discussed in Section 3.1.

Type	Algebraic object	Memory usage
scalar n	integer or real number	$O(1)$
dense vector `dv`	size n vector	$O(n)$
sparse vector `sv`	size n vector	$O(\lvert\mathtt{sv}\rvert)$
CSC matrix `csc`	size $m \times n$ matrix	$O(\lvert\mathtt{csc}\rvert + n)$
coordinate format `cfm`	size $m \times n$ matrix	$O(\lvert\mathtt{cfm}\rvert)$

Table 2: Types of variables.

Linear algebraic operations which we use are given in Table 3. When we say that "the operation is performed in place", we mean that the result is assigned to the memory of the first argument, and the contents of this argument is therefore rewritten. Otherwise, we allocate new memory for the result of this operation. The time complexity of each operation is provided in Table 3. We assume, that we use a random access machine (RAM) for this. Also, we require an operation that creates a sparse vector e_k with one on k-th position and zeros on others by the number k. It requires $O(1)$ time. In Table 3 this operation is denoted as $k \mapsto e_k$.

Operation $\mathtt{map}+=\mathtt{cfm}$ from Table 3, will be used throughout the paper exclusively in correspondence to cases discussed in Subsection 3.2, i. e. pointer changes. Linear algebraic operations $\mathtt{sv1}+\mathtt{sv2}$, $\mathtt{dv}^{\mathtt{T}}\cdot\mathtt{sv}$, $\mathtt{dv}=\mathtt{sv}\star\mathtt{dv}$, $\mathtt{sv1}\star\mathtt{sv2}$, $\mathtt{sv}\star\mathtt{dv}$, $\mathtt{k}\mapsto\mathtt{e_k}$, $\mathtt{cfm1}+\mathtt{cfm2}$, $\mathtt{cfm}\cdot\mathtt{dv}$, $\mathtt{dv}+=\mathtt{map}\cdot\mathtt{sv}$ are discussed in [7, 18, 19]. For the rest of the operations — namely operations $\mathtt{map}\cdot\mathtt{sv}$, $\mathtt{sv1}\cdot\mathtt{sv2}^{\mathtt{T}}$, $\mathtt{map}\cdot\mathtt{cfm}$, $\mathtt{cfm}^{\mathtt{T}}$, $argmin$ `sv`, $\mathtt{cfm}\cdot\mathtt{sv}$ the required estimations can be obtained trivially. As an example, operation $\mathtt{cfm}\cdot\mathtt{sv}$ can be performed in the following way. One should find all compatible pairs of elements in `cfm` and `sv`, order them by the corresponding position in the resulting vector, and add products of pairs, which has the same order. Please note, that we do not claim this solution to be the best, but it is sufficient for the purposes of this paper. The reason for this is the following — this operation will be applied for matrices `cfm` and for vectors `sv` of size not more than

Type	Time cost	Result	Type	Time cost	Result
$\texttt{map} \cdot \texttt{sv}$	$O(\lvert\texttt{sv}\rvert \log \lvert\texttt{sv}\rvert)$	sv	$\texttt{dv}^{\texttt{T}} \cdot \texttt{sv}$	$O(\lvert\texttt{sv}\rvert)$	number
$\texttt{sv1} + \texttt{sv2}$	$O(\lvert\texttt{sv1}\rvert + \lvert\texttt{sv2}\rvert)$	sv	$\texttt{dv}\star = \texttt{sv}$	$O(\lvert\texttt{sv}\rvert)$	in-place
$\texttt{sv1} \star \texttt{sv2}$	$O(\lvert\texttt{sv1}\rvert + \lvert\texttt{sv2}\rvert)$	sv	$\texttt{map}+ = \texttt{cfm}$	$O(\lvert\texttt{cfm}\rvert)$	in-place
$\texttt{sv1} \cdot \texttt{sv2}^{\texttt{T}}$	$O(\lvert\texttt{sv1}\rvert\lvert\texttt{sv2}\rvert)$	cfm	$\texttt{sv} \star \texttt{dv}$	$O(\lvert\texttt{sv}\rvert)$	sv
$\texttt{map} \cdot \texttt{cfm}$	$O(\lvert\texttt{cfm}\rvert \log \lvert\texttt{cfm}\rvert)$	cfm	$\texttt{k} \mapsto \texttt{e}_{\texttt{k}}$	$O(1)$	sv
$\texttt{cfm}^{\texttt{T}}$	$O(\lvert\texttt{cfm}\rvert \log \lvert\texttt{cfm}\rvert)$	cfm	$argmin$ $\texttt{sv}$	$O(\lvert\texttt{sv}\rvert)$	sv
$\texttt{cfm1} + \texttt{cfm2}$	$O(\lvert\texttt{cfm1}\rvert + \lvert\texttt{cfm2}\rvert)$	cfm	$\texttt{cfm} \cdot \texttt{dv}$	$O(\lvert\texttt{cfm}\rvert)$	sv
$\texttt{cfm} \cdot \texttt{sv}$	$O(\lvert\texttt{cfm}\rvert\lvert\texttt{sv}\rvert \log(\lvert\texttt{cfm}\rvert\lvert\texttt{sv}\rvert))$	sv	$\texttt{dv}+ = \texttt{map} \cdot \texttt{sv}$	$O(\lvert\texttt{sv}\rvert)$	in-place

Table 3: Time cost for linear algebraic operations, `dv` stands for dense vector, `sv`, `sv1`, `sv2` stand for sparse vectors, `cfm`, `cfm1`, `cfm2` stands for coordinate format, `map` stands for mapping matrix and for $\texttt{map}+ = \texttt{cfm}$ we assume that result is also mapping matrix. Symbol $\star$ stands for an arbitrary binary operation (assumed to be computed in constant time). We assume that operation $\texttt{dv}\star = \texttt{sv}$ affects only i-th elements from `dv`, for all i such that $(\texttt{i}, \texttt{sv}[\texttt{i}])$ is contained in `sv`.

two due to the nature of the proposed algorithm. The same stays for the rest of the operations.

Under our assumptions about the computational model, the time complexity of each operation depends exclusively on the number of non-zero elements in sparse vectors and coordinate format matrices. Therefore, for the sparse vectors and coordinate format matrices of size $O(1)$, we can perform all of the operations from Table 3 in $O(1)$ time.

4 Linear algebraic version of Fibonacci heap

In this section, we will rewrite the description of the Fibonacci heap from the Section 2 in terms of linear algebra operations.

Linear algebraic Fibonacci heap represents a set of n nodes and some pointers. It consists of:

- dense vector `values` of real numbers that contain a value for each node;
- a dense vector `marked` of zero and one integers that indicates if a node is marked;
- a dense vector `unmarked` of zero and one integers that indicates if a node is not a root and not marked;
- an integer number `roots` which equals to number of roots in the heap
- a sparse vector `minvertex` of zero and one integers that indicates if a node has a minimal value in the heap. In case of more than one nodes in the heap with a minimal value, only one *True* value can be stored in the vector;
- a dense vector `degree` of integer numbers that contains the number of children of each node;
- a sparse mapping matrix `Parent` corresponding to the mapping of each node to its parent;
- a sparse mapping matrix `Children` corresponding to the mapping of each node to the one of its children;

- a sparse mapping matrix `Left` corresponding to the mapping of each node to its left sister node;
- a sparse mapping matrix `Right` corresponding to the mapping of each node to its right sister node;
- a sparse mapping matrix `Successor` is a transposed Jordan block $J^T_{0,d_{max}}$, where $d_{max} = \lfloor 2.08(\ln n + 3) \rfloor$. This number is an upper boundary for possible degrees of nodes, we will prove this fact in the Section 5. This matrix will be used for `extractMin` operation in Subsection 4.3;

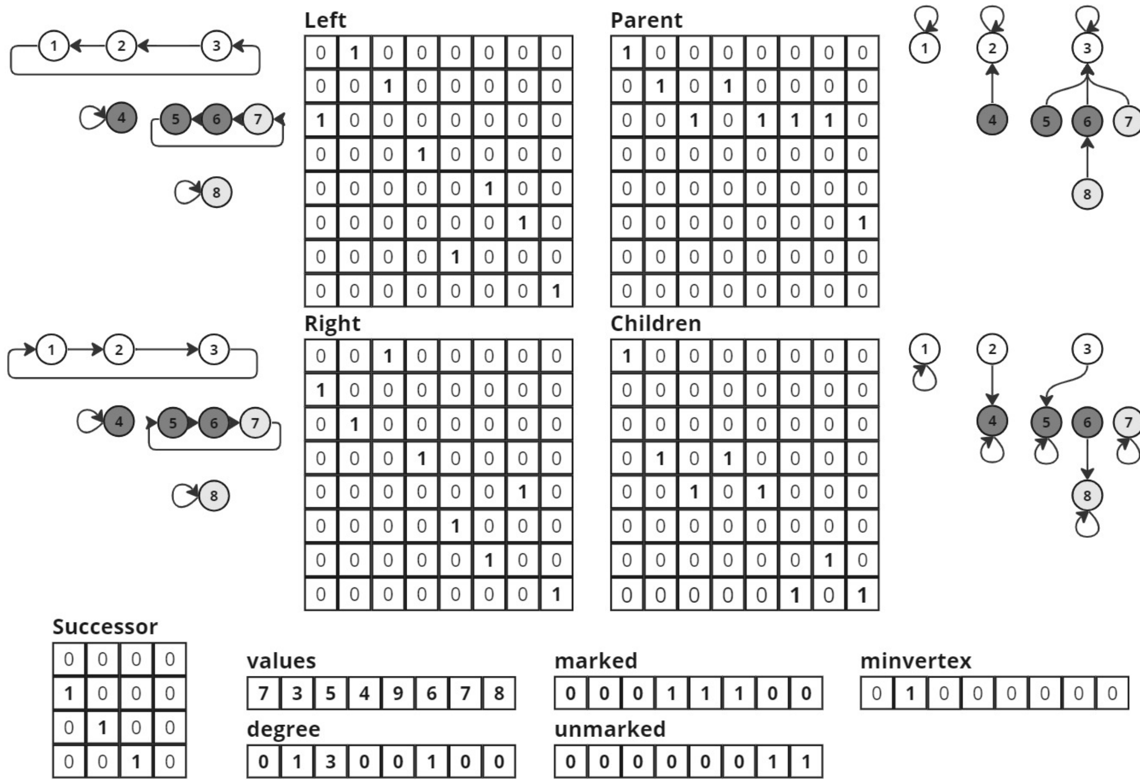

Figure 4: Example of linear algebraic Fibonacci heap. Vertices numbered from left to right from top to bottom. White vertices are roots, dark gray vertices are marked, light gray vertices are unmarked. Numbers in vertices are corresponding values. Edges in each graph correspond to ones in matrices `Parent`, `Children`, `Left`, `Right`.

For `Parent`, `Children`, `Left`, and `Right` in cases when a node doesn't have a parent, child, left sister, or right sister node we assume that it is its own parent, child, left sister, or right sister respectively. For any set of nodes (or a single node), we will use a traditional italic case throughout this paper, for example, v. If we need to use a linear algebraic representation of this entity, we will use the identical designation, but written with a fixed-width font - like this `v`.

Let us describe the initialization process of the Fibonacci heap. Creation of the n-dimensional vector `values` requires $O(n)$ time. We need to initialize vectors `marked`, `unmarked`, and `degree` as vectors of n zeros. Vector `min` has a single non-zero element for the minimum value in `values`, which can be found in $O(n)$. Matrices `Parent`, `Children`, `Left`, `Right` are identity matrices, which can be constructed in $O(n)$ time. Matrix `Successor` can be constructed in $O(d_{max})$ time because it contains ones on positions $(i+1, i)$ and zeros on all others. Number `roots` equals n at the moment of initialization.

We will call any child, which is being mapped with the matrix `Children`, a *pointed* node. If a node is the only child of some other node, we consider it both the left sister and the right sister node of itself.

Note that the time required for the initialization of a CSC matrix or a dense vector is $O(n)$. All new objects that we will initialize are the sparse vectors or listed matrices. The only two exceptions are CSC matrix `RootList` and dense vector `valRootList` of size $n \times d_{max}$ and d_{max} respectively. Both of them will require $O(d_{max}) = O(\log n)$ time for initialization. We will remind you when these objects appear in the Section 4.3.

So, if we want to reach $O(1)$ and $O(\log n)$ amortized time complexity, we can't initialize a dense vector of size n or a CSC matrix of size $n \times n$.

Please note that the memory requirements for all of the above vectors and matrices are $O(n)$. In all of the further algorithms, this limit is not exceeded for any of the resulting vectors and matrices.

4.1 Auxiliary operations

In this subsection, we will describe two auxiliary operations that are commonly used to modify the Fibonacci heap.

4.1.1 Operation connectVertices

The operation `connectVertices`(e, f) requires nodes e and f to be two root nodes. This operation makes vertex f a child of e. It uses a vector representation of nodes `e` and `f` which are trivial to get in $O(1)$ time. Connection of vertices is required for Fibonacci heap only in the consolidation stage of `extractMin` operation. At this stage we connect root nodes only, so the restriction that e and f are roots is admissible.

As we showed in Section 3.2, the addition of matrix $(\mathtt{e} - \mathtt{f}) \cdot \mathtt{f}^T$ to `Parent` changes the parent of f from f to e. The initial parent of f is f itself since f is a root vertex.

Since f has to be non-root vertex after the execution of the `connectVertices`(e, f) operation, we perform two updates of matrices `Left` and `Right` in lines 13-14: (1) we connect left and right sisters of f to restore the cycle of roots; (2) we temporarily make f its own left and right sister. Then, for the matrix `Children`, we made f a new child of e, which is also pointed.

After the check, that the degree (number of children) of e wasn't equal to zero, we modified vectors `Left` and `Right` by connecting children nodes of e and f in a cycle. To be more precise, let ce be the pointed child of e, let rce be its right sister. We restore the cycle of children of e by deleting connection between ce and rce and adding connections between ce and f, f and rce.

In the case of the degree (number of children) of e being zero, we don't need to change the left and right sister nodes of f.

Every operation in the code is performed in $O(1)$ time, therefore a `connectVertices`(e, f) operation also requires $O(1)$ time.

Algorithm 1 connectVertices(e, f)

```
 1: unmarked += f                                  ▷ make f unmarked
 2: degree += e                                    ▷ increase degree of e by one
 3: Parent += (e - f) · f^T                        ▷ make e a new parent of f
 4:                                                ▷ restore root cycle
 5: lf = Left · f
 6: rf = Right · f
 7: Left += (lf - f) · rf^T + (f - lf) · f^T
 8: Right += (rf - f) · lf^T + (f - rf) · f^T
 9: ce = Children · e                              ▷ ce is a child of e
10: rce = Right · ce                               ▷ rce is a right sister of ce
11: Children + = (f - ce) · e^T                    ▷ connect all children of e in a cycle
12: if e^T · degrees > 1 then
13:     Left + = (f - ce) · rce^T + (ce - f) · f^T
14:     Right + = (f - rce) · ce^T + (rce - f) · f^T
15: end if
16: roots − = 1
```

4.1.2 Operation cutVertex

In the operation `cutVertex`(e) basis vector e indicates the vertex that will be cut. This operation cuts vertex encoded by a vector e from its parent and restores the structure of the heap to the correct state. Operation `cutVertex`(e) doesn't process marked vertices that lost a child, it will be handled by the operation `decreaseKeys`.

The algorithm starts with updating of the number `roots` and vectors `marked`, `unmarked`, and `degree`. Let pe be the parent of e and cpe be the pointed child of e. Next, the matrix `Children` is restored. The term $(\mathtt{cpe}^T \cdot \mathtt{e}) \cdot (\mathtt{le} - \mathtt{e}) \cdot \mathtt{pe}^T$ in line 9 makes le the pointed child of pe if $cpe = e$. The term $(\mathtt{le}^T \cdot e) \cdot (\mathtt{pe} - e) \cdot \mathtt{pe}^T$ in line 10 makes pe the pointed child of pe if e is the only child of pe. Then, the cycle of children of pe is restored in lines 11-12 by connecting left and right sisters of e. Note that these two lines do not change matrices `Left` and `Right` in the case when e is the only child of pe.

To satisfy all conditions for e being a root vertex, we update its parent and connect to the cycle of roots in lines 13-16.

Similarly, as for the `connectVertices`(e, f) operation, the operation `cutVertex`(e) is performed in $O(1)$ time.

4.1.3 Operation cutChildren

Later, we will need to cut all of the children of some vertex e. Since the matrix `Children` contains only one child for each vertex, we need a special operation that cuts the vector of children for a vertex given by vector `e`.

Suppose, vertex e has exactly d children. Every step of the `for` loop can be done in constant time and there are d steps of the `for` loop. Therefore, all children of a vertex can be cut in $O(d)$ time.

Algorithm 2 cutVertex(e)

```
1: roots += 1
2: marked -= e ·_ew marked - unmarked ·_ew (Parent · e)
3: unmarked -= e ·_ew unmarked + unmarked ·_ew (Parent · e)    ▷ update roots, marked, and unmarked vertices
4: degree -= Parent · e                                        ▷ decrease degrees of cut vertices' parents
5: pe = Parent · e
6: cpe = Children · pe
7: le = Left · e
8: re = Right · e
9: Children += (cpe^T · e) · (le - e) · pe^T
10: Children += (le^T · e) · (pe - e) · pe^T
11: Left += (le - e) · re^T
12: Right += (re - e) · le^T                                   ▷ restore matrices for children and sister nodes
13: Parent + = (e - Parent · e) · e^T                          ▷ update matrix for parents
14: lmv = Left · minvertex
15: Left += (lmv - le) · e^T + (e - lmv) · minvertex^T
16: (minvertex - re) · e^T + (e - minvertex) · lmv^T
```

Algorithm 3 cutChildren(e)

```
1: child = Children · e
2: for n in range degree^T · e do
3:     lc = Left · child
4:     cutVertex(child)
5:     child = lc
6: end for
```

4.2 Operation decreaseKey

We remind the reader, that the `decreaseKey` operation decreases the key for one vertex.

Algorithm 4 decreaseKey

```
1: values min= w · e                                   ▷ decrease values of the node
2: minvertex = argmin (min(minvertex ·ew values, w · e))  ▷ find such a basis vector minvertex,
   that the expression // min(minvertex ·ew values, w · e) is minimal
3: parvalue = (Parent · e)^T · values
4: if parvalue > w then
5:     cut = e                                         ▷ find vertices that need to be cut
6:     while cut is nonempty do
7:         newcut = marked ·ew (Parent · cut)  ▷ find vertex that need to be cut on the next step
8:         cutVertex(cut)
9:         cut = newcut                                ▷ update vertices that need to be cut
10:    end while
11: end if
```

Operation `decreaseKey` gets a basis vector `e`, a new value w, and decreases the value contained by the vertex corresponding to `e`. This operation can be expressed as a change of each element from vector `values` to a minimum of corresponding elements from vectors `values` and $w \cdot e$. In addition, we recalculate the minimum element of the heap by taking less of the current minimum value and a minimal element from a vector $w \cdot e$. In the listing, operation $argmin(v)$ returns a basis vector e_{min} for which $e_{min}^T v$ is minimal.

After this decrease, we need to restore Fibonacci heap properties. Firstly, we check if the vertex encoded by e has a value less than its parent. The vector `cut` corresponds to the vertex which will be cut on the current step of the `while` loop in lines 6-10. Vector (`Parent` · `cut`) is a vector of indicators, which vertices have a cut child. This allows obtaining vector `newcut` that indicates vertices that will be cut on the next step. This is done by uniting marked vertex that lost a child.

4.3 Operation extractMin

Operation `extractMin` returns the minimal element and deletes it from the heap.

At the beginning of the algorithm, we cut the children of the minimal node, make them the roots, and delete this minimal node. After this, we perform the `consolidate` operation. Matrix `RootList` corresponds to the array A from Section 2 and represents the list of processed roots. Suppose vector e_i has a single nonzero element on i-th position. Also, let j = `degree`$[i]$. Then the left multiplication of matrix `RootList` on the vector d_j which has a single nonzero element on j-th position results in the vector e_i. In other words, matrix `RootList` maps d to the root with degree d in the list. Every iteration of the `for` loop in lines 7-25 corresponds to the addition of some root to the array `RootList` and every step of `while` loop in lines 11-24 corresponds to one check if in `RootList` exists a node with the same degree as the added one. Let us remind that `RootList` and `valRootList` are contained as a CSC matrix of size $n \times d_{max}$ and dense vector of size d_{max} respectively. Its initialization requires $O(d_{max}) = O(\log n)$ time.

All the vectors in the algorithm belong to one of two types. One as before is indexed by nodes, and the other is indexed by degrees (number of children), i. e. each basis vector corresponds to some vertex and its corresponding degree. These vectors have different dimensions of n and d_{max} respectively. Matrix `AddedRoot` expresses a single root that is added to the list at the current stage

Algorithm 5 extractMin

1: lmv = Left · minvertex
2: cutChildren(minvertex) ▷ cut children of the minimum
3: values -= values $\cdot_{ew}$ minvertex ▷ delete the minimal value
4: RootList = 0 ▷ initialize the list of roots for consolidation
5: valRootList = ∞
6: e = vector that consists of ones and dimension d_{max}
7: **for** n in range roots **do** ▷ the consolidate operation
8: dlmv = vector with one on position degree^T · lmv and dimension d_{max}
9: AddedRoot = lmv · dlmv^T ▷ initialize the added root
10: valAddedRoot = AddedRoot^T · values
11: **while** AddedRoot is nonempty **do**
12: aRMore = (valRootList $<$ valAddedRoot)
13: aRLess = (valRootList $\geq$ valAddedRoot) ▷ find coincide degrees and the vertex with less value
14: less = RootList · aRMore + AddedRoot · aRLess
15: more = RootList · aRLess + AddedRoot · aRMore
16: connectVertices(less, more) ▷ connect vertices of the same degree
17: RootList += AddedRoot
18: RootList -= RootList · $diag$(aRMore + aRLess)
19: valRootList = min(valAddedRoot, valRootList)
20: valRootList -= (aRMore + aRLess) $\cdot_{ew}$ valRootList ▷ update values of the list of the roots
21: AddedRoot = less · (Successor · (aRLess + aRMore))^T ▷ find a new added root
22: valAddedRoot = AddedRoot^T · values ▷ update value of the added root
23: roots += 1 - 2(aRMore + aRLess) $\cdot_{ew}$ e ▷ update the number of roots
24: **end while**
25: **end for**
26: minvertex = $argmin$ (root $\cdot_{ew}$ values) ▷ find a new minimal element

of the algorithm. This matrix consists has a single element 1 on the position ($\texttt{degree}[s], s$) and zeros in all the other positions. It means that for a vector v of dimension d_{max} vector $\texttt{AddedRoot} \cdot v$ equals to $v_{d_s} e_s$, where e_s is the s-th basis vector, d_s is the degree of s, and v_{d_s} is the d_s-th component of v. So, matrices `RootList` and `AddedRoot` has sizes $d_{max} \times n$ and corresponds to partial mappings from $[d_{max}]$ to $[n]$. Matrix `AddedRoot` is stored in CSC as in the general case. For `RootList` we use additional zeros as for partial mapping matrix (see Section 3.2).

Let us describe a single step of `while` loop in lines 11-24 in detail. In pair of vectors `aRMore` and `aRLess`, there is at most one nonzero vector. It has a single nonzero element 1 on the position which corresponds to the added root. These vectors allow writing the analysis of all cases without repeating similar steps for these cases. If the considered vector is a zero vector, we are considering the other case. Vector `aRMore` is nonzero when the added root has a greater value than the root with the same degree in `RootList`. Vector `aRLess` is nonzero when the added root has a less or equal value than the root with the same degree in `RootList`. It can be found by comparison of corresponding elements in vectors `valRootList` and `valAddedRoot` (which contain values of corresponding roots).

Suppose we have a pair of roots of the same degree, with one of them being attached to another during the consolidation process. Vectors `less` and `more` correspond to such roots with smaller and larger values respectively. Let us analyze the formula $\texttt{RootList} \cdot \texttt{aRMore} + \texttt{AddedRoot} \cdot \texttt{aRLess}$ then. If the term $\texttt{RootList} \cdot \texttt{aRMore}$ is nonzero, then it results in a basis vector, indicating the root from `RootList` which has the same degree as the added root. We have the following from the definition of `aRMore` — if the term $\texttt{RootList} \cdot \texttt{aRMore}$ is nonzero, then it equals a basis vector which indicates one of the paired roots with a smaller value. The term $\texttt{AddedRoot} \cdot \texttt{aRLess}$ similarly corresponds to another case. The sum of two vectors indicates the vertex with a smaller value in both cases. The same holds for the vector `more`.

After computing `less` and `more` we attach the vertex with the greater value to the one with the lesser value. Vector $\texttt{aRMore} + \texttt{aRLess}$ is nonzero if there is a root in the list with the same degree as the added root. The addition of the matrix $\texttt{AddedRoot} - \texttt{RootList} \cdot diag(\texttt{aRMore} + \texttt{aRLess})$ corresponds to the addition or extraction of the root depending on the case. Values in the list are updated in a similar way.

The matrix `Successor` expresses the partial mapping that compares to each degree d the number $d + 1$. The matrix $\texttt{less} \cdot (\texttt{Successor} \cdot (\texttt{aRLess} + \texttt{aRMore}))^{\texttt{T}}$ corresponds to the partial mapping, which compares to the number d, one of the paired roots with a lesser value if $d - 1$ equals the degree of added root. Since we know that the degree of added root increases by one after attaching a new vertex, this partial mapping correctly expresses the adding root on the next step of the `while` loop in lines 11-24. The value of the added root is also updated.

After that, all that is left is to update vectors `root` and `minvertex` and this completes the `extractMin` operation.

5 Complexity analysis

As discussed in Section 2 the combinatorial version of the Fibonacci heap performs `decreaseKey` and `extractMin` operations in $O(1)$ and $O(\log n)$ time amortized respectively. In this section, we will prove a similar result for the linear algebraic version. We will follow the general structure of the proof for the combinatorial Fibonacci heap.

Amortized time estimates will be given using the method of potentials. We remind that we defined potential as $\Phi = t + 2 \cdot m$, where t is the number of different trees in the heap, and m is the number of marked vertices. For the operation `extractMin`, we will prove that it requires

$O(\Delta\Phi + \log n)$ time, where $\Delta\Phi$ is the decrease of the potential after the operation (if potential was increased then $\Delta\Phi$ equals zero). For the `decreaseKey`(e, w) operation, we will prove an estimate $O(\Delta\Phi + 1)$.

Complexity analysis of the Fibonacci heap requires the usage of Fibonacci numbers, i.e. numbers F_n defined recursively by formula $F_n = F_{n-1} + F_{n-2}$ for $n \geq 3$ and $F_1 = F_2 = 1$. Let us review some well-established facts about Fibonacci heaps that one can find in [4].

- $F_n = 1 + \sum_{m=1}^{n-2} F_m$ for all $n \geq 3$
- $F_n = \frac{\varphi^n - (-\varphi)^{-n}}{\sqrt{5}}$, where $\varphi = \frac{1+\sqrt{5}}{2}$

Lemma 1 *In the Fibonacci heap, for every vertex v of degree d for each $k \in [0, d-1]$, at least $d - k - 1$ of its children having degrees at least k.*

Lemma 2 *In the Fibonacci heap, every tree with the root of degree d has at least F_{d+2} vertices.*

Lemma 3 *In the Fibonacci heap, every vertex has a degree less or equal to $d_{max} = \lfloor 2.08(\ln n + 3) \rfloor$.*

5.1 Operation decreaseKey complexity

We repeat the listing of the `decreaseKey` operation together with time complexity estimates for operations on each string.

Algorithm 6 decreaseKey

```
1:  values min= w · e                                                    ▷ O(1)
2:  minvertex = argmin (min(minvertex ·_ew values, w · e))               ▷ O(1)
3:  parvalue = (Parent · e)^T · values                                   ▷ O(1)
4:  if parvalue > w then
5:      cut = e                   ▷ find vertices that need to be cut    ▷ O(1)
6:      while cut is nonempty do
7:          newcut = marked ·_ew (Parent · cut)                          ▷ O(1)
8:          cutVertex(cut)                                               ▷ O(1)
9:          cut = newcut                                                 ▷ O(1)
10:     end while
11: end if
```

In this subsection, we prove that `decreaseKey` operation requires $O(1)$ time amortized. So one `decreaseKey` operation can be done in $O(1)$ time amortized, just like for the combinatorial version of the Fibonacci heap.

The first three lines of the algorithm require $O(1)$ time. The vector `cut` after these operations can have at most one nonzero element. Every line of a step in the `while` loop in lines 7-10 has a time complexity $O(1)$.

We prove that the total decrease of potential equals at least $l - 2$, where l is the number of steps of the `while` loop in lines 7-10. Also, l — is the number of cut vertices in the `while` loop, and at least $l - 1$ of them were marked. Therefore, we get l new roots and lose at least $l - 1$ marked vertex. So, the decrease of the potential equals at least $2(l-1) - l = l - 2$.

Hence, we can complete all steps using current potential and extra $O(1)$ time and potential. So, this proves the following theorem:

Theorem 1 *Operation* `decreaseKey`(e, w) *requires $O(1)$ time amortized.*

5.2 Operation extractMin

At the beginning of the algorithm, we delete a minimal element and make its children the roots. Since the degree of each vertex is $O(|\log n|)$, all these actions require $O(|\log n|)$ time and increase potential on at most $O(|\log n|)$ by making the children of this minimal element roots. Note that the matrix `RootList` is a partial mapping matrix and might contain some zeros as values in CSC format. Since the maximal degree of each vertex is $O(\log n)$, we need logarithmic time to initialize this matrix.

Algorithm 7 extractMin

```
 1: lmv = Left · minvertex                                        ▷ O(1)
 2: cutChildren(minvertex)                                        ▷ O(log n)
 3: values -= values ·_ew minvertex                               ▷ O(log n)
 4: RootList = 0                                                  ▷ O(log n)
 5: valRootList = ∞                                               ▷ O(log n)
 6: e = vector that consists of ones and dimension d_max          ▷ O(log n)
 7: for n in range roots do                         ▷ the consolidate operation
 8:     dlmv = vector with one on position degree^T · lmv and dimension d_max
 9:     AddedRoot = lmv · dlmv^T                                  ▷ O(1)
10:     valAddedRoot = AddedRoot^T · values
11:     while AddedRoot is nonempty do
12:         aRMore = (valRootList < valAddedRoot)                 ▷ O(1)
13:         aRLess = (valRootList ≥ valAddedRoot)                 ▷ O(1)
14:         less = RootList · aRMore + AddedRoot · aRLess         ▷ O(1)
15:         more = RootList · aRLess + AddedRoot · aRMore         ▷ O(1)
16:         connectVertices(less, more)                           ▷ O(1)
17:         RootList += AddedRoot                                 ▷ O(1)
18:         RootList -= RootList · diag(aRMore + aRLess)          ▷ O(1)
19:         valRootList = min(valAddedRoot, valRootList)          ▷ O(1)
20:         valRootList -= (aRMore + aRLess) ·_ew valRootList     ▷ O(1)
21:         AddedRoot = less · (Successor · (aRLess + aRMore))^T  ▷ O(1)
22:         valAddedRoot = AddedRoot^T · values                   ▷ O(1)
23:         roots += 1 - 2(aRMore + aRLess) ·_ew e                ▷ O(1)
24:     end while
25: end for
26: minvertex = argmin (root ·_ew values)                         ▷ O(log n)
```

All of the matrices `RootList`, `AddedRoot`, `Successor`, and any diagonal matrix have at most one nonzero element in every row and every column. Vectors `aRMore`, `aRLess`, and `valAddedRoot` have at most one nonzero element. Then, each matrix on vector multiplications and binary vector operations in `while` loop in lines 12-24 requires a constant time. Also, vector $\texttt{AddedRoot}^{\texttt{T}} \cdot \texttt{values}$ can be found in constant time because `AddedRoot` has only one nonzero element.

For each step of `while` loop in lines 12-24, we add a new root to the list or extract a root from the list. For each extraction step, the number of roots decreases by one. So, the potential decreases by one. This unit of potential can be used for adding and extracting a single root. After consolidation, the number of roots becomes $O(\log n)$ because all of the roots have different degrees which are bound by $O(\log n)$. Therefore, processing of these roots requires $O(\log n)$ time. So, we

have

Theorem 2 *Operation* `extractMin` *requires* $O(\log n)$ *time amortized.*

6 Fibonacci heap usage

In many cases, the main bottleneck in the transfer of some combinatorial algorithms to the linear algebraic language is the need for a priority queue. In this section, we will provide some examples of such algorithms and prove for them that they have a linear algebraic analog with the same time complexity.

6.1 Dijkstra algorithm

Let G be a weighted graph. The single source shortest path problem is a task of finding the length of the shortest path from some fixed vertex s to all others. Combinatorial Dijkstra algorithm solves this problem in $O(m + n \log n)$ time and uses a priority queue. The linear algebraic version of this algorithm reaches the same time complexity by using the linear algebraic priority queue, proposed in this paper while the trivial approach requires $O(n^2)$ time.

In the Dijkstra algorithm, on each step we choose the nearest vertex u to s that hasn't been chosen before, update distances to any other vertex v by counting paths from s to v with u as the penultimate vertex in the path, and mark u as chosen. On each step, for any chosen vertex, the length of the shortest path is already found.

Let us denote the adjacency matrix of graph G as `G`, let `s` be a vector that indicates source vertex s and `dist` be a vector to be filled with distances from s to other vertices. Then the Dijkstra algorithm can be written as follows:

Algorithm 8 Dijkstra(G, s)

1: Fibonacci heap FH with values from s
2: **while** FH is nonempty **do**
3: dist += values · s
4: **for** each e in G · s **do**
5: decreaseKey(e, e^T (G · s) + $values^T$ s)
6: **end for**
7: s = extractMin(FH)
8: **end while**

Dijkstra algorithm requires n `extractMin` operations and m `decreaseKey`. So, we have

Corollary 6.1 *There exists a linear algebraic Dijkstra algorithm that works in* $O(m+n \log n)$ *time and* $O(n + m)$ *space.*

6.2 Prim algorithm

A minimal spanning tree for the connected graph G is a tree that is a subgraph of G, contains all vertices of G, and has the least possible sum of edge weights. The problem of finding this tree is solved by the Prim algorithm [12] in $O(m + n \log n)$ time. We propose a linear algebraic version that reaches the same time complexity, while the trivial approach requires $O(n^2)$ time.

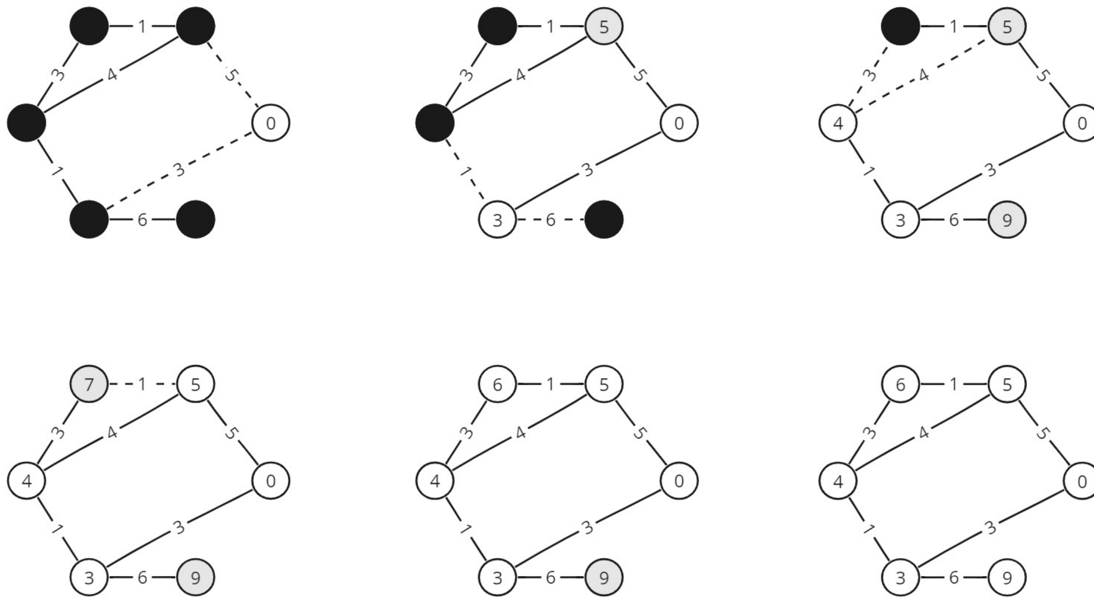

Figure 5: Dijkstra algorithm visualization. Weights of edges are written on them. Numbers on vertices are the minimum found lengths from the source to the corresponding vertex, for white vertices these numbers are exact minimum lengths, for light gray vertices these numbers can be decreased in the next steps.

The Prim algorithm starts constructing a tree from some vertex s and on each step connects the nearest vertex to the tree with the edge with the smallest possible weight. An example of the work of this algorithm is in Figure 6.

Like in the previous subsection `G` is the adjacency matrix of graph G, `s` is a vector that indicates the source vertex s. The `Nearest` is a matrix that corresponds to the mapping of vertices to their nearest vertex in the tree in the tree for other vertices, and `T` is the adjacency matrix of the tree.

This algorithm outputs a matrix of an oriented tree `T`, which, if required, can be made symmetrical in linear time. There are n uses of `extractMin` and m `decreaseKey` operations. So, we have

Corollary 6.2 *There exists a linear algebraic Prim algorithm that works in $O(m + n \log n)$ time and $O(n + m)$ space.*

6.3 Maximum independent set

The exact polynomial algorithm existence for the problem of searching for the maximal independent set is unknown since the problem is **NP**-hard. So we will focus on the heuristic algorithm. The simplest one is a greedy algorithm that requires $O(m + n \log n)$ time in the combinatorial version. As with the previous two algorithms, it includes a priority queue and requires $O(n^2)$ time for the trivial approach.

We consider this algorithm for an unoriented graph. We find the vertex of the minimal degree

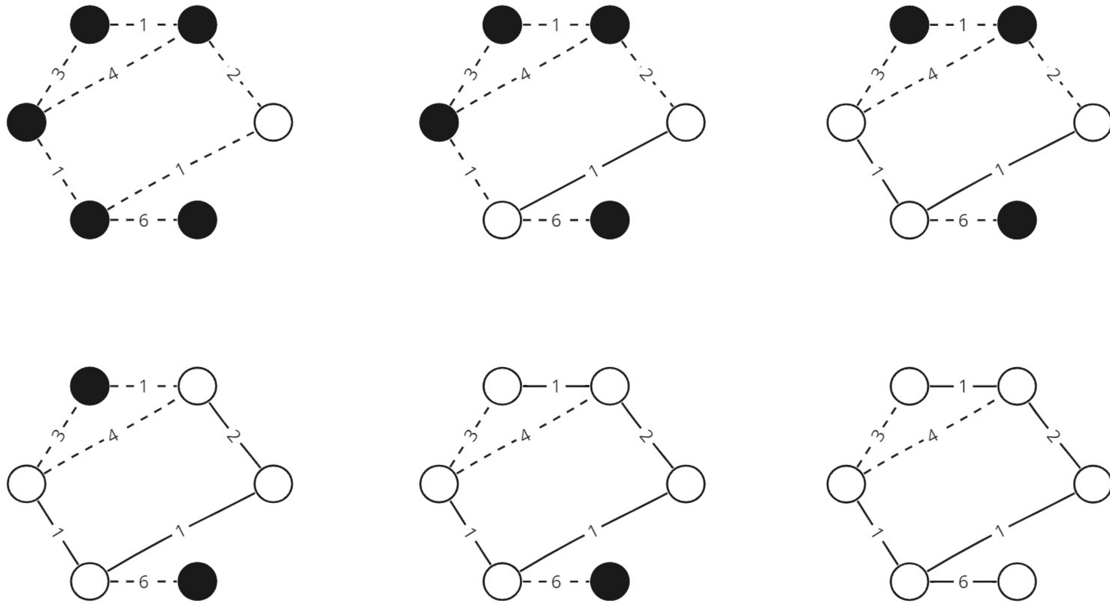

Figure 6: Prim algorithm visualization. Weights of edges are written on them, solid edges are the edges of the constructed tree, and white vertices are the vertices of the constructed tree.

v, put it in the independence set, and delete it with its neighbors from the list of vertices. We repeat this operation until the list of vertices is empty.

Note that to delete vertices that correspond to nonzero elements of vector `v` we can perform `decreaseKeys`(`e`, $-$`1`) for each basis vector `e` included in `v` and |`v`| times use `extractMin`. It deletes `v` in $O(|\mathtt{v}| \log n)$ amortized. Therefore, we have an operation a `delete`(`v`) that deletes all elements that correspond to nonzero elements in vector `v`.

Let `G` be the adjacency matrix of graph G, `set` be a vector that indicates constructed independent set, `mindegree` be a vector that indicates a vertex of minimal degree, and `ones` be a vector that consists of n ones.

This algorithm counts degrees of all vertices in $O(m)$ time, deletes n vertices in $O(n \log n)$ time, and does $O(m)$ operations `decreaseKeys`. This proves

Corollary 6.3 *There exists a linear algebraic greedy algorithm for the maximal independent set that works in $O(m + n \log n)$ time and $O(m + n)$ space.*

6.4 Brandes algorithm

Betweenness centrality is one of the measures that expresses how central and important the vertex is. It equals $\sum_{s,t} \frac{\sigma_{st}(v)}{\sigma_{st}}$, where $\sigma_{st}(v)$ is a number of shortest paths from s to t through v and σ_{st} is a total number of shortest paths from s to t. Paper [3] presents an algorithm that calculates the betweenness centrality for all vertices in a given graph in $O(mn)$ time for the unweighted case and $O(mn + n^2 \log n)$ time for the weighted case. This algorithm uses the breadth-first search for the

Algorithm 9 Prim(G, s)

1: create Fibonacci heap FH with values from s
2: **while** FH is nonempty **do**
3: T += Nearest · *diag*(s)
4: Nearest -= Nearest · *diag*(values ¿ G · s)
5: Nearest += s · (values > G · s)^T
6: **for** each e in G · s **do**
7: decreaseKey(e, e^T (G · s))
8: **end for**
9: s = extractMin(FH)
10: **end while**

Algorithm 10 GreedyMIS(G)

1: degrees = G · ones
2: create Fibonacci heap FH with values from degrees
3: **while** FH is nonempty **do**
4: mindegree = extractMin(FH)
5: set += mindegree
6: delete(G · mindegree)
7: decreaseDegrees = degrees · (G · (G · mindegree) $\neq$ 0)
8: decreaseDegrees -= (G · (G · mindegree))
9: Nearest += s · (values > G · s)^T
10: **for** each e in G · s **do**
11: decreaseKey(e, e^T decreaseDegrees)
12: **end for**
13: **end while**

unweighted case and the Dijkstra algorithm for the weighted case correspondingly. The essential part of this algorithm is n runs of the BFS or Dijkstra algorithm. Its complexity equals $O(nf(n))$, where $O(f(n))$ is the time required by the BFS or Dijkstra correspondingly. So, it requires $O(n^3)$ time for a trivial approach.

According to [17], Brandes algorithm can be expressed in linear algebraic terms without increasing time complexity except for this run of the single source search algorithm for the weighted case. To be more precise, it can be expressed in linear algebraic terms with time complexity $O(nf(n))$ if it uses a single source search algorithm that works in $O(f(n))$. Since we presented a linear algebraic version of the Dijkstra algorithm, we have

Corollary 6.4 *There exists a linear algebraic Brandes algorithm that works in $O(mn + n^2 \log n)$ time and $O(n + m)$ space.*

7 Conclusion

In this paper, we propose a priority queue implementation in terms of linear algebraic operations. Namely, we describe one of the more powerful priority queues — a Fibonacci heap. In addition, we show its application for several well-known algorithms, improving its computational complexity for the linear algebraic case to the theoretical boundaries. The authors hope, that this result will lead to the development of faster algorithms, which use a linear algebra representation.

The same approach, which was used by the authors, can be used for the development of other implementations of the priority queue, which looks like a promising direction for future research.

References

[1] R. Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958. doi:10.1090/qam/102435.

[2] M. Bisson and M. Fatica. High performance exact triangle counting on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 28(12):3501–3510, 2017. doi:10.1109/TPDS.2017.2735405.

[3] U. Brandes. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001. doi:10.1080/0022250X.2001.9990249.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2022. doi:10.5555/1614191.

[5] L. R. Ford and D. R. Fulkerson. Flows in networks. In *Flows in Networks*. Princeton university press, 2015. doi:10.1515/9781400875184.

[6] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34(3):596–615, jul 1987. doi:10.1145/28869.28874.

[7] J. Kepner and J. Gilbert. *Graph algorithms in the language of linear algebra*. SIAM, 2011. doi:10.1137/1.9780898719918.

[8] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. Cusha: vertex-centric graph processing on gpus. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 239–252, 2014. doi:10.1145/2600212.2600227.

[9] H. Liu and H. H. Huang. Enterprise: breadth-first graph traversal on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015. doi:10.1145/2807591.2807594.

[10] T. Mattson, D. Bader, J. Berry, A. Buluc, J. Dongarra, C. Faloutsos, J. Feo, J. Gilbert, J. Gonzalez, B. Hendrickson, J. Kepner, C. Leiserson, A. Lumsdaine, D. Padua, S. Poole, S. Reinhardt, M. Stonebraker, S. Wallach, and A. Yoo. Standards for graph algorithm primitives. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–2, 2013. doi:10.1109/HPEC.2013.6670338.

[11] U. Meyer and P. Sanders. δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003. doi:10.1016/S0196-6774(03)00076-2.

[12] R. C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957. doi:10.1002/j.1538-7305.

[13] E. Robinson and J. Kepner. Array based betweenness centrality. In *SIAM Conference on Parallel Processing for Scientific Computing*, 2008.

[14] J. Soman, K. Kishore, and P. Narayanan. A fast gpu algorithm for graph connectivity. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8. IEEE, 2010. doi:10.1109/IPDPSW.2010.5470817.

[15] D. G. Spampinato, U. Sridhar, and T. M. Low. Linear algebraic depth-first search. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, ARRAY 2019, page 93–104, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3315454.3329962.

[16] W. Tinney and J. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. Nov. 1967. doi:10.1109/PROC.1967.6011.

[17] A. Tumurbaatar and M. J. Sottile. Algebraic algorithms for betweenness and percolation centrality. *Journal of Graph Algorithms and Applications*, 25(LLNL-JRNL-818857), 2021. doi:10.7155/jgaa.00558.

[18] C. Yang, A. Buluç, and J. D. Owens. Graphblast: A high-performance linear algebra-based graph framework on the gpu. *ACM Transactions on Mathematical Software (TOMS)*, 48(1):1–51, 2022. doi:10.1145/3466795.

[19] A. Yzelman, D. Di Nardo, J. Nash, and W. Suijlen. A c++ graphblas: specification, implementation, parallelisation, and evaluation. *Preprint*, 2020. doi:10.1145/3561652.