Incremental Algorithm for Minimum Cut and Edge Connectivity in Hypergraph

Rahul Raj Gupta and Sushanta Karmakar(✉)

Department of Computer Science and Engineering,
IIT Guwahati, Guwahati, Assam, India
rahulrg.raj@gmail.com, sushantak@iitg.ac.in

Abstract. For an uncapacitated hypergraph $H = (V, E)$ with $n = |V|$, $m = |E|$ and $p = \Sigma_{e \in E}|e|$, and edge connectivity λ, this paper presents an insertion-only algorithm which updates minimum cut and edge connectivity incrementally on addition of a set of hyperedges to an existing hypergraph. The algorithm is deterministic and takes $O(\lambda n)$ amortized time per insertion of a hyperedge. The algorithm can answer queries on edge-connectivity in $O(1)$ time and returns a cut of size λ in $O(n)$ time. First we propose a method to maintain a hypercactus [3] under the addition of a set of hyperedges. It is observed that the time for maintaining a hypercactus on addition of a set U of hyperdeges is $O(n + p_u)$ where $p_u = \Sigma_{e \in U}|e|$. This method is then used as a subroutine in our incremental algorithm for maintaining minimum cut and edge connectivity.

Keywords: Hypergraph · Minimum cut · Edge connectivity · Hypercactus

1 Introduction

Computing the minimum cut (or edge connectivity) is a fundamental problem in graph algorithms. There are many algorithms to compute the edge connectivity in a simple graph [6,12,16,19]. There are also a few algorithms that maintain the edge connectivity in a simple graph under the addition of a few edges and vertices [9,10]. Computing the minimum cut or the edge connectivity for a hypergraph is also an important problem. It has applications in various fields e.g., circuit and chip design, network communication, planning in transportation, circuit partitioning and cluster analysis. There are a few algorithms to compute the edge connectivity for a static hypergraph [2,13,15]. The best algorithm known so far to compute the minimum cut for a static hypergraph is given by Chekuri and Xu [2]. However, to the best of our knowledge, no algorithm exists in the literature that maintains the minimum cut or the edge connectivity for dynamic hypergraphs where a few hyperedges are added to or deleted from a given hypergraph. For dynamic hypergraphs, one straightforward approach is to apply a known algorithm [2,13,15] to compute the minimum cut whenever there is a change in the hypergraph (due to addition or deletion of hyperedges). However this simple

C. J. Colbourn et al. (Eds.): IWOCA 2019, LNCS 11638, pp. 237–250, 2019.
https://doi.org/10.1007/978-3-030-25005-8_20

approach has many drawbacks. For example, even for a single hyperedge insertion, the application of the algorithm by Chekuri and Xu [2] requires $O(p+\lambda n^2)$ time to update the minimum cut and the edge connectivity. In a dynamic environment usually the number of hyperedges being added or deleted is relatively small compared to the overall size of the hypergraph. It is therefore desirable to tolerate a small change (edge or node) in an efficient manner. Also the approach helps answer queries on the minimum cut and the edge connectivity in an online setting. The existing algorithms for computing the minimum cut in a hypergraph suffer from the fact that they do not work efficiently under dynamic or online setting.

In this paper, we present an incremental algorithm that updates the minimum cut and the edge connectivity of an existing hypergraph on addition of a set of hyperedges. The algorithm takes $O(n + p_u)$ time if the edge connectivity does not change in spite of the addition of new hyperedges. Otherwise the algorithm recomputes the edge connectivity and the hypercactus using the method of Chekuri and Xu [2] that requires $O(p + \lambda n^2)$ time. Our algorithm is deterministic and takes $O(\lambda n)$ amortized time per insertion of a hyperedge. The algorithm can answer queries on edge-connectivity in $O(1)$ time and returns a cut of size λ in $O(n)$ time. Note that the claimed bound is a significant improvement over the trivial algorithm that computes everything from scratch. This is because any static algorithm must take $\Omega(p)$ time, where p could be exponential in n.

Our Contribution: In this paper, our contributions are the following:

(i) We present a method to maintain a given hypercactus efficiently under dynamic addition of a few hyperedges. Our method takes $O(n + p_u)$ time compared to $O(p + \lambda n^2)$ time taken by the current best approach (Chekuri and Xu [2]) to update the hypercactus given that the edge connectivity does not change. Here p_u, p, λ and n have usual meanings.

(ii) We present an incremental algorithm that updates the minimum cut and the edge connectivity of an existing hypergraph on addition of a set of hyperedges. We use the aforesaid method of maintaining a hypercactus as a subroutine in this incremental algorithm. We show that our algorithm has an amortized time of $O(\lambda n)$ per insertion of a hyperedge.

Organization: In Sect. 2, we discuss the related work. Section 3 contains some basic preliminaries related to the problem. In Sect. 4 we discuss our algorithm. The proof of correctness of the proposed algorithm is given in Sect. 5. We conclude the work in Sect. 6.

2 Related Work

There exist many algorithms for computing the minimum cut and the edge connectivity in a simple graph [6,12,16,19] and in the last few decades, dynamically updating the edge connectivity in a simple graph has been addressed by many

researchers. In 2016, Goranci et al. [9] gave a deterministic incremental algorithm that maintains the edge connectivity in $\widetilde{O}(1)$ amortized time per edge insertion in undirected and unweighted graph. Henzinger [10] has also given an incremental algorithm that maintains the minimum cut and the edge connectivity of a graph G under dynamic addition of a set of edges. It takes $O(\lambda \log n)$ amortized time per insertion of a simple edge. Dinitz and Westbrook proposed a method [5] to maintain a cactus representation [4] which stores all possible minimum cuts in a graph and this method is used as a subroutine in the incremental algorithm given by Henzinger [10]. The current best deterministic algorithm to compute a cactus representation of a graph G is given by Gabow [7] which requires $O(m + \lambda^2 n \log(n/\lambda))$ time.

In case of hypergraphs there are many static algorithms to compute different hypergraph properties like minimum cut, minimum weight hyperpath, transitive closure, rank, independent sets, etc. A careful implementation of the method given by Queyranne [17] to compute the minimum cut in a hypergraph takes $O(np + n^2 \log n)$ time for capacitated hypergraph and $O(np)$ time for an uncapacitated hypergraph. Klimmek and Wagner [13], Mak and Wong [15] independently gave algorithms for computing the minimum cut in a hypergraph having same time bound. The current best algorithm to compute the edge connectivity and the hypercactus is given by Chekuri and Xu [2] that requires $O(p + \lambda n^2)$ time. Ausiello et al. [1] proposed an algorithm to maintain transitive closure for a hypergraph under dynamic addition of hyperedges. Italiano and Nanni [11] proposed an algorithm to maintain minimum rank and minimum gap hyperpath over a batch of hyperedge insertions. In [18], a Dijkstra-like procedure has been proposed for maintaining a weighted shortest path in a *fully-dynamic hypergraph.*

3 Preliminaries

Let $H = (V, E)$ be an uncapacitated hypergraph where V is the set of vertices and E is the set of hyperedges where each hyperedge $e \in E$ is a subset of vertices, $n = |V|$, $m = |E|$ and $p = \Sigma_{e \in E}|e|$ where $|e|$ is the number of vertices in a hyperedge e. A *cut* is the partitioning of V into two non-empty sets A and $V \backslash A$. The set of hyperedges connecting the two sets (also called cut-edges) contribute to the value of the cut. Out of all possible cuts in a hypergraph, any cut whose value is minimum is known as a minimum cut.

We denote the set of hyperedges intersecting both A and $V \backslash A$ with $\delta_H(A)$ and call it a *cut-edge set of H.* A hypergraph $H' = (V', E')$ is said to be a *subhypergraph* of a hypergraph $H = (V, E)$ if $V' \subseteq V$ and there is a bijective mapping $\phi : E \to E'$ where $\phi(e) \subseteq e$ for each $e \in E$. For vertices $u, v \in V$, a (u, v)-walk of length k in H is a sequence $v_0 e_1 v_1 e_2 v_2 \ldots v_{k-1} e_k v_k$ of vertices and hyperedges (possibly repeated) such that $v_0, v_1, \ldots, v_k \in V$, $e_1, \ldots, e_k \in E$, $v_0 = u$, $v_k = v$, and for all $i = 1, 2, \ldots, k$, the vertices v_{i-1} and v_i are adjacent in H via the hyperedge e_i. The vertices $u, v \in V$ are said to be connected in H if there exists a (u, v)-walk in H. The hypergraph H is said to be connected if every pair of distinct vertices is connected in H. The minimum number of

hyperedges whose removal disconnects H is called the *edge connectivity* of H, which is denoted by λ. A cut is *k-cut* if $|\delta_H(A)| = k$. The vertices s and t are said to be *k-edge connected* if there exists no $k^{'}$*-cut*, $k^{'} < k$ that disconnects the pair $\{s, t\}$.

A *cactus tree* [4] is an $O(n)$ sized data structure which compactly represents all possible minimum cuts of size k in a k-edge connected simple graph G. A cactus tree $\tau(G)$ can be represented as (G^*, Φ) where G^* is a simple weighted graph and $\Phi : V(G) \longrightarrow V(G^*)$ such that Φ is a many to one mapping. The properties of $\tau(G)$ are:

(i) $\Phi(u) = \Phi(v)$ if and only if the vertex u and v are not separated by a λ-cut in G.
(ii) If $(X, \bar{X})$ is a λ-cut of G^* for $X \subset V(G^*)$ then $(Y, \bar{Y})$ is a λ-cut of G where $Y = \{\Phi^{-1}(u) | u \in X\}$.
(iii) If λ is odd, G^* is a tree and every edge of G^* has a weight of λ. If λ is even, two simple cycles of G^* have at most one common node, every edge that belongs to a cycle has weight $\lambda/2$ and every edge that does not belong to a cycle has weight λ.

For a given hypergraph H and edge connectivity λ, a *hypercactus* is an $O(n)$ sized data structure which compactly represents all possible minimum cuts of size λ. The hypercactus $\tau(H)$ [3] can be represented as (H^*, Φ) where H^* is a hypergraph and $\Phi : V(H) \rightarrow V(H^*)$ such that Φ is a many to one mapping. The properties of $\tau(H)$ are as follows:

(i) $\Phi(u) = \Phi(v)$ if and only if the vertex u and v are not separated by a λ-cut in H.
(ii) If $(X, \bar{X})$ is a λ-cut of H^* for $X \subset V(H^*)$ then $(Y, \bar{Y})$ is a λ-cut of H where $Y = \{\Phi^{-1}(u) | u \in X\}$.

The main difference between a cactus and a hypercactus is that a hypercactus can have hyperedges (see Fig. 2 for example) in addition to simple edges and cycles whereas a cactus can have only simple edges and cycles. The weight of each non cycle edge and each hyperedge in $\tau(H)$ is λ whereas the weight of each cycle edge in $\tau(H)$ is $\lambda/2$. Figure 2 represents a hypercactus $\tau(H)$ for a hypergraph H as shown in Fig. 1 whose edge connectivity is 2. In Fig. 2, $x1 = x2 = \phi$ and $\Phi(1) = \Phi(2) = \Phi(3) = \Phi(4) = a$. For every other vertex u of H, $\Phi(u) = u$ in $\tau(H)$ as shown in the figure. The edge weights are also shown in the figures. In addition to structural differences between a cactus and a hypercactus, there are also operational differences between them. For example, we can apply *shrinking* and *squeezing* operations on a cactus structure whereas we can apply *tuning* operation in addition to *shrinking* and *squeezing* operations in a hypercactus.

Dinitz and Westbrook [5] defined *shrinking* of nodes and *squeezing* of cycles as follows. In *shrinking* a subset of vertices $W \subseteq V$, the operation replaces all vertices in W by a single vertex w, deletes all edges whose both end points lie in W. For any edge (x, y) where $x \in W$ and $y \notin W$, the operation replaces (x, y) with (w, y).

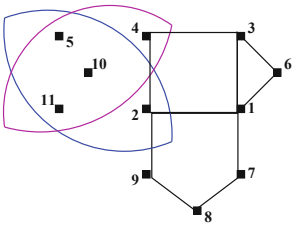

Fig. 1. H $(\lambda = 2)$

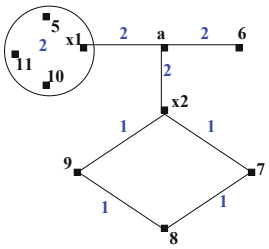

Fig. 2. $\tau(H)$

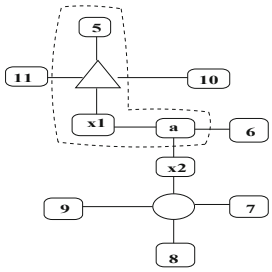

Fig. 3. $T(H)$

Let $C = (v_1, v_2, ..., v_k, v_1)$, $k \geq 2$ be a cycle. Then *squeezing* C at v_i and v_j, $i < j$, results in *shrinking* v_i and v_j. The *squeezing* results in two new cycles: $(w, v_{j+1}, ..., v_k, v_1, ..., v_{i-1}, w)$ and $(w, v_{i+1}, ..., v_{j-1}, w)$. If the length of a resulting cycle is 2, the cycle gets replaced by a simple edge. In this paper we introduce a new operation named *tuning* of a hyperedge which is defined in Subsect. 4.2.

4 The Proposal

In this section we first briefly discuss the work of Dinitz and Westbrook [5] for maintaining the cactus tree $\tau(G)$ of a graph G. This is used as a subroutine for maintaining the minimum cut and the edge connectivity in a simple graph under dynamic addition of edges (Henzinger [10]). Next we propose our method to maintain a hypercactus $\tau(H)$ of a hypergraph H on dynamic addition of hyperedges. We use this hypercactus maintenance method as a subroutine for designing the incremental algorithm for maintaining the minimum cut and the edge connectivity in a hypergraph, which is described in Subsect. 4.3.

4.1 Cactus Maintenance

Let n be the number of vertices in G. The data structure used by Dinitz and Westbrook takes $O(n + m + q)$ time to perform m number of *Insert-Edge(u,v)* operations and q number of *Same-k-Class(u,v)* queries. The *Insert-Edge(u,v)* operation inserts a new edge (u, v) dynamically to a graph G. The *Same-k-Class(u,v)* query returns *true* if vertex u and v are k-edge connected and returns $false$ otherwise. The algorithm takes $O(m+k^2 n \log(n/k))$ preprocessing time in order to construct the initial data structure where m is the number of edges in G. Previously the algorithms for maintaining all possible minimum cuts of size 1, 2 and 3 in a graph G has been described in [8,14]. The algorithm by Dinitz and Westbrook [5] is a generalization for an increasing value of k.

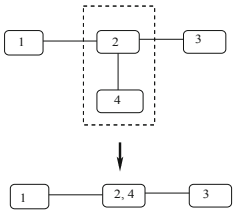

Fig. 4. Square nodes

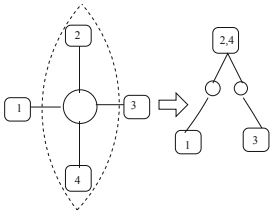

Fig. 5. Round nodes

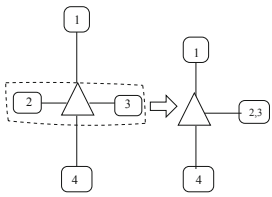

Fig. 6. Triangle nodes

The author introduced the definition of the *auxiliary tree* which is an extension of the *cactus tree*. The auxiliary tree $T(G)$ consists of two types of nodes: *square node*, one for each node in $\tau(G)$ and *round node*, one for each cycle in $\tau(G)$. Two square nodes in $T(G)$ are connected by a simple edge if corresponding nodes in $\tau(G)$ are connected by a simple edge. The square nodes in $T(G)$ which are a part of a cycle C in $\tau(G)$ are made adjacent to the round node in $T(G)$ representing C. The square nodes connected to a round node in $T(G)$ follow the order in which the corresponding nodes in $\tau(G)$ are connected in a cycle. A leaf node is created for each vertex v in G. These leaf nodes are made children of the corresponding square nodes in $T(G)$. The query *Same-(k+1)-Class(u,v)* returns *true* only if vertices u and v have same parent in $T(G)$.

On inserting a new edge, the algorithm first finds a unique path between the corresponding square nodes in $T(G)$. It modifies the path such that it correctly reflects the effects of *squeezing* cycles and *shrinking* nodes in $\tau(G)$. If two square nodes in the path are connected by a simple edge in $T(G)$ then the square nodes are merged in $T(G)$ as shown in Fig. 4 and *shrinking* of corresponding nodes in $\tau(G)$ are applied. If two square nodes in the path are connected to a round node in $T(G)$ then the modification of $T(G)$ is done as shown in Fig. 5 and *squeezing* of corresponding cycle in $\tau(G)$ is applied. There will be leaf nodes connected to the square nodes in $T(G)$, however we omit them for the clarity of the figures.

4.2 Hypercactus Maintenance

In this subsection we propose a method to maintain a hypercactus on dynamic addition of a set U of hyperedges which takes $O(n + p_u)$ time. This method inherits the ideas proposed by Dinitz and Westbrook [5] described in Subsect. 4.1. The method is described using the following cases:

(i) $\lambda = 0$: In this case the given hypergraph H is disconnected. On adding a new hyperedge $e = \{u_1, u_2, ..., u_l\}$, the method updates H using a fast disjoint set-union data structure [20] which takes $O(q\alpha(q, n))$ time to perform any sequence of q number of *union* and *find* operations. Here α is the functional inverse of the Ackermann's function (practically, α is a constant). The method first creates a set of each connected component of H.

The *find(v)* operation returns the name of a component containing the vertex v. The *union* operation takes two connected components as the input, merges the two components and returns the merged component. The query *Same-*$(\lambda+1)$*-Class(u,v)* returns true if $find(u) = find(v)$. The method iterates through each vertex in e. If $find(u_i) = find(u_{i+1})$ for $1 \leq i < l$ then the method does not change anything. Otherwise *union* operation on disjoint sets $find(u_i)$ and $find(u_{i+1})$ is applied. The method follows the same procedure for each hyperedge $e \in U$. Note that total $O(p_u)$ number of $find$ and *union* operations are called in this case and thus the time complexity is $O(p_u \alpha(p_u, n))$. After a certain number of hyperedge insertions if the hypergraph has a single component then the hypergraph is no more disconnected. In this situation the method computes the edge connectivity and the hypercactus using the algorithm of Chekuri and Xu [2] and applies the technique given in the following case to maintain the hypercactus.

(ii) $\lambda \geq 1$: In this case the hypergraph is connected. In case of a graph, a cactus consists of simple edges and cycles. In case of a hypergraph, a hypercactus can have hyperedges in addition to simple edges and cycles. The method extends the definition of the auxiliary tree $T(H)$ used by Dinitz and Westbrook for the case of hypercactus. This auxiliary tree is used in the proposed method for hypercactus maintenance. In addition to *square* nodes and *round* nodes, another type of node called *triangle* node is introduced in $T(H)$, one for each hyperedge in $\tau(H)$. Square nodes and round nodes follow the same rules as described earlier in the case of cactus maintenance. Here the rules for triangle nodes are described. The square nodes in $T(H)$ which are the part of a hyperedge e in $\tau(H)$ are made adjacent to the triangle node in $T(H)$ corresponding to e. Unlike the case of round nodes, the square nodes can be connected to a triangle node in any order. A leaf node is created for each vertex v in H. These leaf nodes are made children of the corresponding square nodes in $T(H)$. Figure 3 represents the auxiliary tree $T(H)$ for a hypercactus $\tau(H)$ shown in Fig. 2. The query *Same-*$(\lambda+1)$*-Class(u,v)* returns *true* if vertices u and v belong to same parent in $T(H)$. The $find(v)$ operation for a vertex $v \in e$ returns the corresponding square node in $T(H)$. Similarly, the $union(x, y)$ operation merges the nodes x and y in $T(H)$.

On inserting a hyperedge $e = \{u_1, u_2, ..., u_k\}$, the method iterates through each vertex in e. Let $x = find(u_i)$ and $y = find(u_{i+1})$. If $x = y$ then the method does not change anything. Otherwise the method finds the unique path between the nodes x and y in $T(H)$. There can be three types of nodes in the path: square nodes, round nodes and triangle nodes. This path is then modified in such a way that it correctly reflects the effects of *shrinking* nodes, *squeezing* cycles and *tuning* hyperedges in $\tau(H)$. The *tuning* of a hyperedge is defined as follows.

Tuning Operation: Let $e = (v_1, v_2, ..., v_k)$, $k > 2$ be a hyperedge (for $k = 2$, e is a simple edge). Then *tuning* e at v_i and v_j, $i < j$, results in *shrinking* v_i and v_j. This results in a new hyperedge: $(v_1, v_2, ..., v_{i-1}, w, v_{i+1}, ..., v_{j-1}, v_{j+1}, ..., v_k)$. Here w denotes the supervertex obtained after merging

nodes v_i and v_j throughout the entire hypergraph. If the size of the resulting hyperedge is 2, the hyperedge is replaced by a simple edge.
If two square nodes in the path are connected by a simple edge in $T(H)$ then the *union* operation is applied on the square nodes in $T(H)$ as shown in Fig. 4 and the *shrinking* operation on corresponding nodes in $\tau(H)$ is applied. If two square nodes in the path are connected to a round node in $T(H)$ then the nodes are modified as shown in Fig. 5 and the *squeezing* operation is applied on the corresponding cycle in $\tau(H)$. Similarly, if two square nodes in the path are connected to a triangle node in $T(H)$ then the nodes are modified as described below and the *tuning* operation is applied on the corresponding hyperedge in $\tau(H)$.
Triangle Node Modification: Let s_1 and s_2 be the square nodes connected to a triangle node t i.e., (s_1, t) and (t, s_2) are the edges in the path between nodes x and y in $T(H)$. The method merges s_1 and s_2 into a supernode w and connects this w with t. All the edges connected to s_1 and s_2 get connected to w. Rest other edges connected to t remain the same. After the modification, if t has 2 square nodes connected to it then t is deleted and these two square nodes get connected with a direct edge. An example of triangle node modification is shown in Fig. 6.

Theorem 1. *Let $\tau(H) = (H^*, \Phi)$ be the hypercactus representation of a hypergraph H whose edge connectivity is λ. Under dynamic addition of a set U of hyperedges to H, $\tau(H)$ can be maintained in $O(n + p_u)$ time, where n is the number of vertices in H and $p_u = \Sigma_{e \in U} |e|$.*

Proof. The method described in Subsect. 4.2 to maintain $\tau(H)$ under dynamic addition of a set of hyperedges uses the method similar to Dinitz and Westbrook [5] described in Subsect. 4.1 with additional case of handling hyperedges. From the construction of the auxiliary tree $T(H)$ it is clear that each hyperedge of size k has $k + 1$ number of nodes and k number of edges in $T(H)$. From Cheng [3], we know that $|V(H^*)| = O(|V(H)|)$ and $|E(H^*)| = O(|V(H)|)$. Thus, the construction of $T(H)$ corresponding to $\tau(H)$ can be done in linear time. Under dynamic addition of a set of hyperedges, the method applies *find* and *union* operations for each hyperedge iteratively. The merging of two square nodes in the path between two nodes in the auxiliary tree takes $O(1)$ time. Correspondingly, *shrinking* an edge and *squeezing* a cycle in $\tau(H)$ takes $O(1)$ time. The method introduces a new technique called *tuning* which modifies a hyperedge in $\tau(H)$. In *tuning*, the method *shrinks* the two nodes and as a result a new hyperedge forms. The operation can be done in $O(1)$ time using the method of disjoint set union-find operation as used for *shrinking* and *squeezing* technique. On inserting a hyperedge e, the method uses $O(|e|)$ number of find and union operations to update $T(H)$ and $\tau(H)$. Thus, total $O(p_u)$ number of *find* and *union* operations are used under the insertion of a set U of hyperedges. The total time taken by *find* and *union* operation is $O(p_u \alpha(p_u, n))$. The total running time to maintain $\tau(H)$ is $O(n + p_u \alpha(p_u, n))$ time. In practical scenario, $\alpha \leq 6$. Thus, the total cost to maintain a hypercactus under dynamic addition of a set U of hyperedges is $O(n + p_u)$. □

Algorithm 1. The Incremental Algorithm

1: Compute λ, τ(H) using Chekuri and Xu [2]
2: $N \leftarrow \phi$
3: **while** there is ≥ 1 mincut of size λ **do**
4: **if** next operation is **query-size()**
5: print λ
6: **if** next operation is **query-mincut()**
7: print the mincut //Refer to point 2 in Subsection 4.3
8: **if** next operation is **Same-(λ+1)-Class**(u, v)
9: print "true" or "false"
10: **if** next operation is **InsertHyperedges(U)**
11: update $\tau(H)$ //Refer to Subsection 4.2
12: $N \leftarrow N \cup U$
13: Recompute λ and $\tau(H)$ with $H = (V, E^{'})$ where $E^{'} = E \cup N$
14: Goto step 2

4.3 The Incremental Algorithm

In this subsection we describe an incremental approach to maintain the minimum cut and the edge connectivity of an uncapacitated hypergraph H under dynamic addition of a set of hyperedges. The psuedocode of the proposed algorithm is given in Algorithm 1. In line 1, the algorithm first computes λ and $\tau(H)$ using the algorithm of Chekuri and Xu [2]. The algorithm checks in $O(1)$ time if there exist at least one minimum cut of size λ in $\tau(H)$ by asserting that $\tau(H)$ has more than one node. In the algorithm we discuss four queries considering that the hypercactus is modified and then these queries come:

(i) **query-size()**: It returns the current value of λ. Since λ is always known, we can return the result in $O(1)$ time.

(ii) **query-mincut()**: It returns a minimum cut of the form $(A, \bar{A})$ such that $A \subset V$. The auxiliary tree $T(H)$ is first split by any edge (u, v). The DFS (Depth First Search) method is then applied on nodes u and v to get two connected components $(X, \bar{X})$ such that $u \in X$ and $v \in \bar{X}$. Then $(\Phi^{-1}(X), \Phi^{-1}(\bar{X}))$ is the resulting minimum cut. The number of vertices and edges in $T(H)$ are of size $O(n)$ (due to Cheng [3]), therefore the query takes $O(n)$ time.

(iii) **Same-($\lambda + 1$)-Class**(u, v)**:** It returns true if vertices u and v have edge connectivity greater than λ.; here both u and v have same parent node in $T(H)$. If $find(u) = find(v)$ the algorithm returns true otherwise it returns false. The time complexity for this query is $O(1)$.

(iv) **InsertHyperedges(U)**: This query inserts a set U of hyperedges into H. Due to the addition of new hyperedges some cuts of size λ in H may no more remain a minimum cut. In order to update all the minimum cuts of size λ the algorithm applies the method described in Subsect. 4.2 that maintains hypercactus $\tau(H)$ on insertion of a set U of hyperedges in $O(n + p_u)$ time.

Let N denote the set of all hyperedges added so far such that no minimum cut of size λ exists in $\tau(H)$. If the algorithm keeps receiving single edge insertions at any point of time i.e, $U = \{e\}$ then the algorithm sets $\lambda = \lambda + 1$ at line 13 of Algorithm 1 instead of actually recomputing λ. Otherwise it recomputes λ. The algorithm recomputes the hypercactus $\tau(H)$ for updated λ using the method of Chekuri and Xu [2] that takes $O(p + \lambda n^2)$ time. The algorithm goes to line 2 to continue the incremental process.

Theorem 2. *The amortized time to maintain the minimum cut and the edge connectivity for a dynamic hypergraph H is $O(\lambda n)$ per hyperedge insertions.*

Proof. Let λ_0 be the initial edge connectivity in line 1 of Algorithm 1. It takes $O(p_0 + \lambda_0 n^2)$ time to compute λ_0 and $\tau(H)$ in line 1 of Algorithm 1. During the execution of Algorithm 1, let λ assume the values $\lambda_0, \ldots, \lambda_f$ in an increasing order. Phase i consists of all steps executed while $\lambda = \lambda_i$. Let U_i denote the set of hyperedges inserted in Phase i. In Phase i, we compute the new edge connectivity λ_i and $\tau(H)$ in line 13 and maintain $\tau(H)$ in line 11. The time to compute λ_i and the corresponding hyercactus $\tau(H)$ in line 13 is $O(p+\lambda_i n^2)$ where p is calculated in the modified hypergraph. From Theorem 1, the time taken to maintain $\tau(H)$ is $O(n + p_i)$ where $p_i = \Sigma_{e \in U_i} |e|$. The total time spent in executing Phase i is $O(n + p_i + p + \lambda_i n^2)$. The maximum number of phases can be λ. Thus the total time to execute all phases is asymptotically $O(\lambda p + \lambda^2 n^2)$ where $p = \Sigma_{e \in E \cup U} |e|$. The amortized cost of a hyperedge insertion is $O(\lambda + \lambda^2 n^2/p)$. For a hypergraph H with edge connectivity λ, $p = \Omega(\lambda n)$ Thus the amortized insertion time is $O(\lambda + \lambda n) = O(\lambda n)$. □

4.4 Analysis

Let H be the given uncapacitated hypergraph whose edge connectivity is λ. Let $\tau(H)$ be the corresponding hypercactus. Let U be the set of hyperedges that is dynamically inserted to H. In order to compute λ and $\tau(H)$, we can apply the static algorithm (Chekuri and Xu) which takes $O(p_0 + p_1 + \lambda' n^2)$ time where $p_0 = \Sigma_{e \in E} |e|$, $p_1 = \Sigma_{e \in U} |e|$ and λ' is the new egde connectivity. With our proposed incremental algorithm, if the value of λ does not change then updating $\tau(H)$ takes $O(n + p_u)$ time. Otherwise, computing the new edge connectivity λ' and the corresponding hypercactus takes $O(\lambda' n)$ amortized time (Theorem 2). Hence the cost of the proposed algorithm is better than the static algorithm.

Now we show the probabilistic approach to compute the cost of updating λ and $\tau(H)$ using the proposed incremental algorithm. Let f denote the probability that the value of λ gets changed under dynamic addition of a set U of hyperedges to H. Then the total cost to update λ and $\tau(H)$ is asymptotically $O(f(p_0 + p_1 + \lambda n^2) + (1 - f)(n + p_1))$. Thus, if the value of f is very small, then the proposed incremental algorithm requires less computation time than the static method.

5 Proof of Correctness

Lemma 1. *A hypercactus has the following properties:* ***(a)*** *No two hyperedges can have more than one node in common.* ***(b)*** *A hyperedge and a cycle can have*

at most one node in common. ***(c)*** *No two cycles can have more than one node in common.*

Proof. Let there be two hyperedges that have more than one node in common in a hypercactus. Each hyperedge in the hypercactus has a weight of λ. From the hypercactus construction, we have one minimum cut of the form $(A, \bar{A})$ such that at least one common node belongs to A and the other common nodes belong to $\bar{A}$. But to get such a minimum cut, we need to remove two hyperedges since the common nodes belong to both the hyperedges. Thus, the size of the cut becomes 2λ which is not minimum. This is a contradiction. Hence two hyperedges in $\tau(H)$ can not have more than one node in common. Using similar arguments we can proof properties **(b)** and **(c)**. □

Lemma 2. *On introducing a triangle node into the auxiliary tree $T(H)$, it remains a tree.*

Proof. We first claim that $T(H)$ constructed after introducing a triangle node is connected i.e., there is a path between any two nodes in $T(H)$. Let us assume that there exists two nodes in $T(H)$ between which no path exists. This can be possible only if $T(H)$ is disconnected. It means each square node connected to the triangle node has no edge with other square nodes or round nodes in $T(H)$. But $T(H)$ is constructed from a hypercactus, it implies that the hypercactus is disconnected. This is a contradiction. Thus the auxiliary tree $T(H)$ is always connected.

Now we claim that $T(H)$ has no cycles. Let us assume that $T(H)$ have a cycle after introducing a triangle node. It means that the square nodes connected to a triangle node forms a cycle with other square nodes or round nodes in $T(H)$. But if this is the case then in the hypercactus the corresponding hyperedge forms a cycle with the cycle edges or simple edges which is a contradiction as per Lemma 1. Thus, the auxiliary tree can not have any cycle after introduction of a triangle node. Hence the auxiliary tree remains a tree on introducing a triangle node into it. □

Lemma 3. *On applying a tuning operation, the hypercactus preserves it's properties, i.e., it preserves all the minimum cuts of size λ.*

Proof. Dinitz and Westbrook [5] gave the definition of *shrinking* and *squeezing* operations. On applying a *shrinking* or *squeezing* operation to the hypercactus, the updated hypercactus preserves it's properties. In this paper, the *tuning* operation is introduced. Let $\tau(H)$ be the hypercactus corresponding to the hypergraph H. Let $Z = \{a_1, a_2, ..., a_k\}$ be a hyperedge in $\tau(H)$. Let $\tau(H^{'})$ be the updated hypercactus after applying a tuning operation on nodes a_i and a_j in Z, $i \neq j$. We prove that the $\tau(H^{'})$ preserves it's properties using three cases:

- Case $|Z| = 2$: As per the definition of tuning operation, the hyperedge of size 2 is treated as a simple edge. This simple edge gets the same weight as of hyperedge Z i.e., λ. The shrinking operation between the nodes a_i and a_j is applied in this case and thus preserves the properties of $\tau(H^{'})$. This follows from the work of Dinitz and Westbrook.

- Case $|Z| = 3$: In this case, the shrinking operation on nodes a_i and a_j is first applied. The size of updated hyperedge $Z^{'}$ becomes 2. This $Z^{'}$ is replaced with a simple edge with same weight λ. In a hypercactus the weights of each hyperedge and each simple edge is always λ. Thus from the construction, all the mincuts of size λ are preserved in the updated hypercactus.
- Case $|Z| \geq 4$: In this case, the shrinking operation on nodes a_i and a_j is first applied. The size of hyperedge Z gets reduced by 1. The updated hyperedge $Z^{'}$ is still a hyperedge with same weight λ. Thus, all the mincuts of size λ are preserved in the updated hypercactus. □

Lemma 4. *An update on $T(H)$ eventually leads to a corresponding update on $\tau(H)$.*

Proof. On insertion of a new hyperedge $e = \{v_1, v_2, .., v_k\}$ to the given hypergraph H, the method first finds a unique path P in $T(H)$ between the two square nodes s_1 and s_2 which corresponds to the nodes v_i and v_{i+1} in e respectively. In the path P the two consecutive square nodes can either be directly connected, connected by a round node or connected by a triangle node. If in the path P, no triangle node is involved i.e, every consecutive square nodes are either directly connected or connected to a round node then the modification technique to update $T(H)$ and $\tau(H)$ is exactly the same as Dinitz and Westbrook [5]. Hence for this case the updated $T(H)$ corresponds to the updated $\tau(H)$.

If two consecutive square nodes in the path P are connected by a triangle node then we update $T(H)$ as described above in **Triangle Node Modification** and apply *tuning* operation between the two corresponding vertices in $\tau(H)$. We show that the updated $T(H)$ corresponds to the updated $\tau(H)$. Let (s_1, t) and (t, s_2) be the consecutive edges in the path P. Here, s_1 and s_2 denote the two consecutive square nodes connected to the triangle node t. After modifying the path P in $T(H)$, let s denotes the square node in $T(H)$ after merging s_1 and s_2. Similarly, let w_1 and w_2 denote the vertices in $\tau(H)$ corresponding to the square nodes s_1 and s_2 in $T(H)$ respectively. After applying the tuning operation on w_1 and w_2, let w denotes the merged node in $\tau(H)$. This w in $\tau(H)$ should correspond to s in $T(H)$. For the sake of contradiction, let us assume that w and s do not correspond to each other. It means that w either maps to some other square node $s^{'} \neq s$ in $T(H)$ or it maps to empty. From the construction of $T(H)$ we create one square node for each vertex in $\tau(H)$. Therefore w can not map to empty. Let us consider the case where w maps to $s^{'}$. In the modification of path P, $s^{'}$ is not touched. Thus there must exist a vertex $w^{'}$ in $\tau(H)$ that maps to $s^{'}$. From our assumption w maps to $s^{'}$ which means $w = w^{'}$. But w is formed after merging w_1 and w_2. This leads to a contradiction that $w = w^{'}$. Thus w in $\tau(H)$ maps to s in $T(H)$. This proves that the updated $T(H)$ corresponds to the updated $\tau(H)$. □

6 Conclusion

Under dynamic addition of hyperedges, when the edge connectivity changes the proposed incremental algorithm relies on Chekuri and Xu's approach to recom-

pute hypercactus. It may be worth investigating a method to recompute the hypercactus efficiently using the structures behind the Chekuri and Xu's static algorithm, instead of recomputing everything from scratch. Similarly it would be interesting to recompute or update the hypercactus efficiently in the deletion case. It may help in designing an efficient decremental algorithm to maintain the edge connectivity and the minimum cut under dynamic deletion of hyperedges. All the contributions made in this paper are for uncapacitated hypergraph. It will be worth investigating a method for the capacitated case.

References

1. Ausiello, G., Nanni, U., Italiano, G.F.: Dynamic maintenance of directed hypergraphs. Theor. Comput. Sci. **72**(2–3), 97–117 (1990)
2. Chekuri, C., Xu, C.: Computing minimum cuts in hypergraphs. In: Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 1085–1100. Society for Industrial and Applied Mathematics (2017)
3. Cheng, E.: Edge-augmentation of hypergraphs. Math. Program. **84**(3), 443–465 (1999)
4. Dinits, E.A., Karzanov, A.V., Lomonosov, M.V.: On the structure of the system of minimum edge cuts in a graph. In: Investigations in Discrete Optimization, pp. 290–306 (1976) (in Russian)
5. Dinitz, Y., Westbrook, J.: Maintaining the classes of 4-edge-connectivity in a graph on-line. Algorithmica **20**(3), 242–276 (1998)
6. Gabow, H.N.: A matroid approach to finding edge connectivity and packing arborescences. In: Proceedings of the Twenty-third Annual ACM Symposium on Theory of Computing, pp. 112–122. ACM (1991)
7. Gabow, H.N.: The minset-poset approach to representations of graph connectivity. ACM Trans. Algorithms **12**(2), 24:1–24:73 (2016)
8. Galil, Z., Italiano, G.F.: Maintaining the 3-edge-connected components of a graph on-line. SIAM J. Comput. **22**(1), 11–28 (1993)
9. Goranci, G., Henzinger, M., Thorup, M.: Incremental exact min-cut in polylogarithmic amortized update time. In: 24th Annual European Symposium on Algorithms, ESA 2016, 22–24 August 2016, Aarhus, Denmark, pp. 46:1–46:17 (2016)
10. Henzinger, M.R.: A static 2-approximation algorithm for vertex connectivity and incremental approximation algorithms for edge and vertex connectivity. J. Algorithms **24**(1), 194–220 (1997)
11. Italiano, G.F., Nanni, U.: On-line maintenance of minimal directed hypergraphs. In: Italian Conference on Theoretical Computer Science, pp. 335–349 (1989)
12. Kawarabayashi, K.I., Thorup, M.: Deterministic global minimum cut of a simple graph in near-linear time. In: Proceedings of the Forty-seventh Annual ACM Symposium on Theory of Computing, pp. 665–674. ACM (2015)
13. Kilmmek, R., Wagner, F.: A simple hypergraph min cut algorithm. Technical Report B 96–02, Department of Mathematics and Computer Science, Freie Universität Berlin (1996)
14. La Poutré, J.A., van Leeuwen, J., Overmars, M.H.: Maintenance of 2-and 3-edge-connected components of graphs. Discrete Math. **114**(1–3), 329–359 (1993)
15. Mak, W.K., Wong, D.: A fast hypergraph min-cut algorithm for circuit partitioning. Integr. VLSI J. **30**(1), 1–11 (2000)

16. Nagamochi, H., Ibaraki, T.: Computing edge-connectivity in multigraphs and capacitated graphs. SIAM J. Discrete Math. **5**(1), 54–66 (1992)
17. Queyranne, M.: Minimizing symmetric submodular functions. Math. Program. **82**(1–2), 3–12 (1998)
18. Ramalingam, G., Reps, T.: An incremental algorithm for a generalization of the shortest-path problem. J. Algorithms **21**(2), 267–305 (1996)
19. Stoer, M., Wagner, F.: A simple min-cut algorithm. J. ACM (JACM) **44**(4), 585–591 (1997)
20. Tarjan, R.E., Van Leeuwen, J.: Worst-case analysis of set union algorithms. J. ACM (JACM) **31**(2), 245–281 (1984)