Preserving Large Cuts in Fully Dynamic Graphs

by

Omer Wasim
BEng., The University of Hong Kong, 2018

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Preserving Large Cuts in Fully Dynamic Graphs

by

Omer Wasim
BEng., The University of Hong Kong, 2018

Supervisory Committee:

Dr. Valerie King, Supervisor
(Department of Computer Science)

Dr. Bruce Kapron, Departmental Member
(Department of Computer Science)

Supervisory Committee

Dr. Valerie King, Supervisor
(Department of Computer Science)

Dr. Bruce Kapron, Departmental Member
(Department of Computer Science)

ABSTRACT

This thesis initiates the study of the MAX-CUT problem in fully dynamic graphs. Given a graph $G = (V, E)$, we present the first fully dynamic algorithms to maintain a $\frac{1}{2}$-approximate cut in sublinear update time under edge insertions and deletions to G. Our results include the following deterministic algorithms: i) an $O(\Delta)$ *worst-case* update time algorithm, where Δ denotes the maximum degree of G and ii) an $O(m^{1/2})$ amortized update time algorithm where m denotes the maximum number of edges in G during any sequence of updates.

We also give the following randomized algorithms when edge updates come from an oblivious adversary: i) a $\tilde{O}(n^{2/3})$ update time algorithm[1] to maintain a $\frac{1}{2}$-approximate cut, and ii) a $\min\{\tilde{O}(n^{2/3}), \tilde{O}(\frac{n^{3/2+2c_0}}{m^{1/2}})\}$ worst case update time algorithm which maintains a $(\frac{1}{2} - o(1))$-approximate cut for any constant $c_0 > 0$ with high probability. The latter algorithm is obtained by designing a fully dynamic algorithm to maintain a sparse subgraph with sublinear (in n) maximum degree which approximates all large cuts in G with high probability.

[1]Throughout this thesis, $\tilde{O}$ hides a $O(\text{polylog}(n))$ factor.

Contents

List of Figures

ACKNOWLEDGEMENTS

I am greatly indebted to my supervisor, Valerie King without whom this thesis would not be possible. From accepting me to the program and generously supporting me throughout the last two years, to teaching me the ropes of doing research in theory and guiding me through highs and lows, she has done more than what I expected from a mentor. Her crisp and intuitive way to look at problems continuously inspires me to work on fundamental and important problems. I have learnt a lot from our weekly meetings, impromptu drop-ins and corridor walks about randomness, algorithms, career goals and other general topics. Her warm, approachable and positive attitude makes research even more exciting. I would also like to thank her in particular, for encouraging and supporting me to attend CANADAM 2019 and the Swedish Summer School in Theoretical Computer Science (S3CS 2019). I also acknowledge Valerie's NSERC Discovery Grant which supported my research.

I would like to thank Hung Le for his sincere advice on succeeding in graduate school, doing research and writing papers among many other (academia-related) things. Out of many productive discussions, one was quite crucial in obtaining a result on fully dynamic graph sparsifiers in this thesis.

I am grateful to Bruce Kapron for serving on the supervisory committee. I am also thankful to Venkatesh Srinivasan for general advice and writing a letter of recommendation when I applied to several graduate schools.

Wendy Beggs, Kath Milinazzo and Nancy Chan have been instrumental in streamlining my academic experience. I whole-heartedly thank them for helping me navigate through administrative, logistical and financial matters during my time here.

Without the friends I made in Victoria, I would have struggled to maintain a healthy work-life balance. Thanks for agreeing to come out on many nights on short notice.

Finally, I would like to my family for all their prayers and unconditional love. In particular, my parents for always being a constant source of emotional support, and my elder siblings Umaid and Alina for always being out there for me in every way possible.

I dedicate this thesis to my parents who have truly been exceptional in educating, supporting and nurturing me at every stage in life. I can only hope that I am able to mimic their selfless parenting.

Chapter 1

Introduction

Graphs provide a convenient abstraction to model networks in the real world. Fully dynamic graphs model networks which change with time. For a concrete example, consider a social network on n users and let the links between users represent friendship between them. Such a network can be modeled by a graph on n vertices (representing users) and the edges between vertices denoting friendship. Define an update in the social network as a user either befriending or unfriending another user. Suppose now that some property on the updated network which is obtained from the original network via a single update needs to be quickly recomputed (for example, measuring how 'far' users are in the network). A naive and brute-force solution would be to recompute the property on the modified graph as a completely new problem instance i.e., without utilizing any valuable information gleaned during the computation of the property on the original graph. This might seem unnecessary since the new graph, which models the social network after a single update differs from the original graph by a *single* edge. The latter fact is typically exploited by a dynamic graph algorithm to speed up recomputation.

Precisely, a dynamic graph algorithm is required to maintain a property of the underlying graph faster than the best static algorithm under edge and/or vertex updates. Here, a static algorithm is one which executes on a fixed problem instance. In this thesis, we consider only edge updates. Note that a vertex insertion or deletion can be alternately viewed as insertions or deletions of all incident edges to the vertex at once. The performance of a dynamic algorithm is typically measured by the time taken to handle the update, i.e. recomputing the property on the graph following the edge update. The worst-case update time of a dynamic algorithm is the update time incurred after a single update to the graph. Alternatively, the update

time can be measured in an amortized sense, i.e. averaged over *any* sequence of updates. Moreover, dynamic algorithms can be broadly classified as either deterministic or randomized. Deterministic worst-case update time is the strongest and the most pessimistic notion of update time. This is because it does not make any (distributional) assumptions on the input graph or the edge updates, which come from an adversary with unlimited (computational) power. On the other hand, a randomized dynamic algorithm exploits the source of random bits provided to it, to speed up update time. Based on this, a natural (and common) strategy to develop dynamic algorithms is to first design a randomized dynamic algorithm and then, derandomize it and/or consider an adversary with unlimited power to obtain deterministic worst-case guarantees.

Many fundamental problems studied over the past several decades on graphs such as shortest paths, connectivity, minimum spanning trees, topological ordering etc. admit polynomial time algorithms in the static setting, i.e. when the input graph is fixed. Research in dynamic algorithms roughly began in the 1980's and the aforementioned problems were all natural candidates to consider in the dynamic setting. There now exist very fast dynamic algorithms for most of these problems. However, many real world problems involve optimizing an objective function while fulfilling some natural constraints. It is well known that many such optimization problems are NP-hard, i.e. cannot be solved *exactly* in polynomial time. In fact for many NP-hard problems, even obtaining an 'approximately' optimal solution is NP-hard. Research in the static setting for many optimization problems has led to efficient polynomial time approximation algorithms, i.e. those which guarantee a feasible solution with objective value within multiplicative or additive factors of the optimal solution. In fact, for numerous well studied problems the quality of approximation achieved by state-of-the-art algorithms is the best possible under the Unique Games Conjecture (UGC)[26].

Traditionally, research in dynamic algorithms has focused mostly on dynamic variants of well-known problems such as connectivity [31, 20, 23], minimum spanning trees [18, 20, 36], minimum cut [34], etc., all of which admit polynomial time exact algorithms in the static setting.

Following the seminal work of Onak and Rubinfeld [30] who present data structures for maintaining constant factor approximations of maximum matching (and vertex cover), research in dynamic algorithms has broadened to include NP-hard problems. Some natural directions in the domain of such problems include: i) de-

veloping dynamic algorithms with sublinear update time while *approximately* maintaining a feasible solution, ii) investigating the approximability-time tradeoff and iii) proving lower bounds. Over the past few years, a (non-exhaustive) list of problems investigated in the dynamic setting include vertex cover [6, 32, 10, 29], set-cover [16], dominating set [19], graph coloring [9], facility location [15] and maximal independent set [17, 4, 5].

In this thesis, we initiate the study of the MAX-CUT problem in dynamic graphs. MAX-CUT is a well known NP-complete problem [24] which continues to be widely studied. Some of its concrete applications arise in the design of integrated circuits, communication networks and statistical physics. Moreover, it is a natural candidate for an objective function to consider in a clustering and/or graph partitioning problem.

Definitions: Let $G = (V, E)$ be an undirected, unweighted graph $G = (V, E)$ with $n = |V|, m = |E|$. A cut C is a partition of the vertex set V and denoted by $C = (S, \bar{S})$, where $S, \bar{S} \subseteq V$ and $\bar{S} = V \backslash S$. The cut-set $E(S, \bar{S})$, of $C = (S, \bar{S})$, is the set of all edges which have one endpoint in S and the other in $\bar{S}$. A cut edge of C is an edge contained in the cut-set $E(C) = E(S, \bar{S})$. A maximum cut of G is a cut whose cut-set is largest among cut-sets for all possible cuts, i.e. MAX-CUT$(G) = \arg\max_{C=(S,\bar{S}),\, S \subseteq V} |E(S, \bar{S})|$, where $|E(S, \bar{S})|$ denotes the *number* of cut edges. Let OPT denote the size of the cut-set of the maximum cut. A t-approximation algorithm for MAX-CUT yields a cut C, such that the size of its cut-set is at least $t \cdot OPT$ and C is said to be a t-approximate cut. Unlike the minimum cut problem which can be solved in polynomial time, MAX-CUT is NP-hard and under the Unique Games Conjecture (UGC), hard to approximate better than 0.878[27].

1.1 Previous Work

Static algorithms: Johnson's folklore greedy algorithm [21] hereafter referred to as Greedy Max-Cut, finds a $\frac{1}{2}$ approximate cut as follows: Given $G = (V, E)$, where $V = \{v_1, v_2, ..., v_n\}$ it starts with $S = \{v_1\}, \bar{S} = \emptyset$. Each successive vertex v_j, where $j \geq 2$ is added to S or $\bar{S}$ depending on which contains fewer of its neighbors v_i, where $i < j$. Thus, at least half of all edges of the form $\{v_j, v_i\}$ where $i < j$ are contained in the resulting cut. Since each vertex and edge is encountered once, the running time of Greedy Max-Cut is $O(m + n)$.

In a breakthrough paper, Goemans and Williamson (1994) used a semidefinite pro-

gramming (SDP) relaxation [14] and randomized rounding to yield a 0.878-approximation to MAX-CUT. This polynomial-time algorithm runs in super-linear time using state-of-the art numerical methods for solving a semidefinite program. Their general techniques were applied to get significantly improved approximation algorithms for a variety of problems in combinatorial optimization.

Arora and Kale [2] presented a primal dual SDP-based algorithm which runs in $\tilde{O}(m)$ time for d regular graphs and returns a $0.878 - \epsilon$-approximate cut with high probability where the running time depends inversely on ϵ. Trevisan later presented a 0.53 approximation algorithm for MAX-CUT utilizing spectral techniques [35] whose analysis was improved to 0.62 by Soto [33]. In the same paper, Trevisan showed that the primal dual SDP-based algorithm of [2] can be made to run in $\tilde{O}(m)$ time for any degree, via a linear time reduction aimed at reducing the maximum degree to $O(\text{polylog}(n))$. It remained an open question whether any purely combinatorial algorithm could beat Greedy Max-Cut.

Kale and Seshadhri [22] answered the former question in the affirmative and presented an algorithm using random walks to give a $0.5+\epsilon$ approximation with running time depending on ϵ. For $\epsilon = 0.0155$, the running time is $\tilde{O}(n^2)$. As the running time increases, the approximation ratio converges to Trevisan's spectral algorithm [35]. Mathieu and Schudy [28] posed an open question as to whether a random permutation of vertices considered by Greedy Max-Cut could beat the $\frac{1}{2}$-approximation. This was later shown to be false by Costello el al. in [13]. Recently, Censor Hillel et al. [11] presented efficient and improved distributed algorithms for MAX-CUT in the LOCAL and CONGEST models of communication.

To the best of our knowledge, the MAX-CUT problem has not been studied in the fully dynamic setting.

1.2 The Fully-Dynamic Model

In this thesis, we seek to maintain a $\frac{1}{2}$-approximate cut in sublinear update time and handle meaningful queries such as determining whether an edge is in the cut-set, the size of vertex partitions and the cut-set in *constant time*. This is motivated by the fact that Greedy Max-Cut finds a $\frac{1}{2}$-approximate cut in $O(m+n)$ time. Moreover, all known *combinatorial* algorithms which attain better than $(\frac{1}{2}+\delta)$-approximation take super-linear running time where δ is small. For example the algorithm in [22] takes $\tilde{O}(n^{1.6})$ time for $\delta = 0.0051$. We define the Dynamic Max-Cut problem as follows:

Problem 1. *(Dynamic MAX-CUT) Starting with a graph $G = (V, E)$ on n vertices and an empty edge set E, maintain a $\frac{1}{2}$-approximate cut $C = (S, \bar{S})$ for G under edge insertions and deletions to E such that queries of the following form can be handled in constant time: i) Is the edge $\{v_i, v_j\}$ contained in the cut-set $E(S, \bar{S})$? ii) What is the size of the cut-set, $E(C)$? iii) What are the sizes of S and $\bar{S}$?*

Our goal is to update C in $o(m + n)$ time such that $|E(C)| \geq \frac{1}{2}|E|$ to fare better than running Greedy Max-Cut from scratch after every update. Moreover, answers to all queries between any two updates must be consistent with respect to the maintained cut C. An event happens with high probability (w.h.p) if its probability is $1 - \frac{1}{n^c}$ for any $c > 0$. The approximation we seek to maintain is either exact or w.h.p. For our randomized algorithms, we assume that updates come from an oblivious adversary. This is a standard assumption used in the design of many randomized dynamic algorithms. An oblivious adversary is one which cannot choose updates adaptively in response to the answers returned by queries. Thus, updates to the graph can be assumed to be fixed in advance. We assume the existence of an oracle which randomly labels each vertex uniquely using a number in $\{1, ..., n\}$, and to which the adversary is oblivious. This is only used in the algorithm of Theorem 5. The reason why we insist on attaining a deterministic approximation or one which holds w.h.p. is because there is a trivial constant-update time algorithm to maintain a $\frac{1}{2}$-approximate cut *in expectation* when the adversary is oblivious: a random cut initialized before the update sequence stays $\frac{1}{2}$-approximate (*in expectation*) after every update.

Attaining a *deterministic worst-case* update time (i.e. without randomization or amortization) is the holy grail in dynamic algorithms. For the seminal problem of dynamic connectivity, deterministic algorithms beating $O(\sqrt{n})$ update time [12, 25] were only recently discovered after decades of research. For the fundamental problem of maintaining a maximal independent set, known deterministic algorithms [4, 17] only achieve a sublinear (in m) amortized update time and polylogarithmic update time algorithms are yet to be discovered.

1.2.1 On dynamizing known static algorithms

Dynamic algorithms with amortized update time: A natural question to ask is whether maintaining a $(\frac{1}{2} - \epsilon)$-approximate cut is easier than maintaining a $\frac{1}{2}$-approximate cut. We note that for any constant $\epsilon > 0$, a $(\frac{1}{2} - \epsilon)$-approximate cut can be maintained in constant amortized update time. Indeed, starting from a $\frac{1}{2}$-approximate cut one

can run Greedy Max-Cut after every ϵm updates. At any point during those ϵm updates, the cut remains $(\frac{1}{2} - \epsilon)$-approximate. As a result, the amortized update time is $O(m/\epsilon m) = O(1/\epsilon)$. The following observation allows one to obtain dynamic algorithms by using known static algorithms as subroutines.

Observation 2. *Given a $(\frac{1}{2} + \delta)$-approximation algorithm $\mathcal{A}$ for the MAX-CUT problem which runs in time $T(n)$ where $\delta > 0$, there exists a fully dynamic algorithm $\mathcal{D}$ which maintains a $\frac{1}{2}+c\delta$-approximate cut where $0 < c < 1$ in $\frac{1}{(1-c)\delta m}T(n)$ amortized update time.*

Proof. Use $\mathcal{A}$ to obtain a $\frac{1}{2}+\delta$-approximate cut and recompute the cut using $\mathcal{A}$ after every $(1-c)\delta m$ updates. When c, δ are fixed constants, the amortized update time of $\mathcal{D}$ is $\frac{1}{m}T(n)$. □

Based on the above observation, we examine the amortized update times achieved by dynamizing known static algorithms for the MAX-CUT problem. For any constant $0 < \epsilon < 0.0155$, the algorithm of Kale et al. [22] can be used to maintain a $0.5 + c\epsilon$-approximate maximum cut where $c < 1$ is a fixed constant in amortized $O(n^{1/2}+g(\epsilon))$ update time where $g(\epsilon) < n^{\frac{1}{2}}$. This result holds with high probability, and is achieved by first running the algorithm to obtain a $0.5+\epsilon$ approximation, and recomputing after every $(1-c)\epsilon m$ updates. The update time holds for the regime when $m \geq n$ and with small modifications, can be made to work when m is small. For $0.5155 < \epsilon \leq 0.62$, Trevisan's spectral algorithm [35] can be used to maintain a $0.5 + c\epsilon$-approximate maximum cut for a fixed constant $0 < c < 1$ in $\tilde{O}(n)$ amortized update update time by recomputing as above. The algorithm of Arora and Kale [2] near optimally solves the MAX-CUT SDP of [14] for d regular graphs in $\tilde{O}(m)$ time using a primal-dual technique. Trevisan generalized this result to graphs of arbitrary degree by sparsifying the input graph to reduce the maximum degree to $\tilde{O}(1)$, yielding an $\tilde{O}(m)$ algorithm to compute a $(0.878-\delta)$-approximate cut with high probability for any fixed constant $\delta > 0$. By recomputing a cut periodically as above, it is possible to get a dynamic algorithm to maintain a $(0.878 - \delta)$-approximate cut in polylogarithmic amortized update time.

All the static algorithms considered above first sparsify the input graph. We note that recent work on dynamic graph sparsification (see [1] for example) does not directly yield improved fully dynamic algorithms since the cost of periodically recomputing the cut is significant (i.e. at least linear in m).

1.3 Our Contribution

We present the first fully dynamic deterministic and randomized sublinear update time algorithms for the Dynamic MAX-CUT problem. All our algorithms maintain a cut which contains at least $\frac{|E|}{2}$ edges, yielding a $\frac{1}{2}$-approximation. Our results are summarized as follows.

Theorem 3. *There exists a fully dynamic (deterministic) algorithm for the Dynamic MAX-CUT problem with $O(\Delta)$ worst case update time, where Δ denotes the maximum degree of the graph after the update.*

Theorem 4. *There exists a fully dynamic (deterministic) algorithm for the Dynamic MAX-CUT problem with $O(m^{1/2})$ amortized update time, where m is the maximum number of edges in the graph during any arbitrary sequence of updates.*

Theorem 5. *There exists a fully dynamic randomized algorithm for the Dynamic MAX-CUT problem which is always correct and takes $\tilde{O}(n^{2/3})$ worst case update time with high probability.*

For many fundamental problems such as maximum independent set and graph coloring, algorithms in the fully dynamic, distributed and sublinear settings have a dependency on the maximum degree Δ. For problems where locality is exploited in the design of algorithms, a notion of degree sparsification can lead to direct improvements in running times albeit at the cost of randomness or approximation. See [3] for an example of sparsification for $(\Delta + 1)$-vertex coloring.

We study a new variant of graph sparsification, which we call (maximum) degree sparsification. This is relevant for the dynamic MAX-CUT problem where our objective is to reduce the maximum degree of the graph while preserving the weight of all large cuts. A large cut is one whose cut set has size at least $c|E|$ for any constant $c > 0$. We show the existence of sparsifiers preserving the weight of all cuts within $\pm o(1)|E|$ additive error (translating to a $(1 \pm o(1))$ multiplicative error for large cuts), while ensuring that the *maximum* degree of any vertex is sublinear in n. It is important to contrast this objective with another well-studied objective of cut-sparsification which seeks a sparsifier containing a near linear number of edges and preserving the value of all cuts within $(1 \pm \epsilon)$-multiplicative error for any constant $\epsilon > 0$ [7]. However, the maximum degree of such sparsifiers can be $\Omega(n)$. We show that our sparsification process can be made fully dynamic.

Theorem 6. *Given a graph $G = (V, E)$ on n vertices and $m = n^{1+\epsilon}$ edges where $0 < \epsilon \leq 1$, there exists a fully dynamic algorithm to maintain a sublinear (in n) maximum degree sparsifier H of G, which takes $O(n^{1+\epsilon})$ preprocessing time and constant update time. At all times, $\Delta_H = \tilde{O}(n^c)$ where $c = 1 - \frac{\epsilon}{2} + 2c_0$ for any constant $c > 0$ and all large cuts in G are preserved in H within $(1 \pm o(1))$ multiplicative error.*

The algorithm of Theorem 3 is given for unweighted graphs but it gives the same update time when all edge weights are the same, as in the case of H. Combining this with the algorithm of Theorem 5, we obtain our final result.

Theorem 7. *There exists a fully dynamic randomized algorithm which maintains a $(\frac{1}{2} - o(1))$-approximate cut and takes $\min\{\tilde{O}(n^{2/3}), \tilde{O}(\frac{n^{3/2+2c_0}}{m^{1/2}})\}$ worst case update time for any constant $c_0 > 0$ with high probability.*

1.4 Our techniques:

Our techniques utilize combinatorial and structural properties of cuts in graphs. The key insight underlying all our algorithms is the following: in any cut C which is not $\frac{1}{2}$-approximate, there exists a vertex which can be moved across the cut to increase the size of $C's$ cut-set. We show that this vertex can be efficiently found, yielding a simple deterministic $O(\Delta)$ worst case update time algorithm which is used as a subroutine for our other algorithms. We remark that this algorithm is not 'local' in the sense that endpoints of the updated edge need not qualify as vertices which can be moved to increase the number of cut edges. There are several instances where such locality can be useful to obtain fast dynamic algorithms for NP-hard problems such as vertex cover, independent set and coloring.

Central to our sublinear time algorithms of Result 3 and 4 is a *cut-combining* technique. This allows us to work on induced subgraphs of G and combine their 'locally maintained' cuts to yield a $\frac{1}{2}$-approximate cut on G. However, the update time depends the complexity of maintaining $\frac{1}{2}$-approximate cuts on individual subgraphs and the combining step. We work around this non-trivial dependence. For our algorithm of Theorem 4, we partition vertices based on their degree and only *selectively* update data structures while having stale information in others. We show that selective updating is sufficient for our purpose and refine the vertex partition after sufficiently many updates. This leads to a simple $O(m^{1/2})$ amortized update time algorithm.

To obtain the algorithm of Theorem 5, we construct a random multi-way k-partition of V and use cut-combining to obtain a sublinear in n worst case update time, settling the question of whether there exist sublinear (in n) *worst case* update time algorithms for Dynamic MAX-CUT.

Finally, we obtain new results on graph sparsification relevant to MAX-CUT in the fully dynamic setting. The objective is to maintain a sparse subgraph H, of a graph G undergoing edge updates, which has sublinear (in n) maximum degree and preserves all large cuts within $(1 \pm o(1))$-multiplicative error. With basic probabilistic tools, we first construct sparsifiers with sublinear (in n) maximum degree in the static setting. Then we show that sampling a constant number of edges after every update is sufficient to construct fully dynamic sparsifiers without compromising the quality of sparsification or the worst case update time. Maintaining cuts on those sparsifiers instead of the graph G yields better worst-case update time as compared to the algorithm of Theorem 5 for certain values of m.

We note that prior work on reducing the maximum degree of the input graph in the static setting such as in [35], does not seem applicable to get fast worst-case update time algorithms. This is due to two reasons: i) the sparsifier is not a subgraph of G and, ii) the cost of converting the cut maintained on H to one in G after any update takes $\tilde{O}(m)$ time.

1.5 Organization

In the next chapter, we present an $O(\Delta)$ update time algorithm. In chapter 3, we give a sublinear time algorithm which takes $O(m^{1/2})$ amortized update time. In chapter 4, we give a randomized algorithm which takes $\tilde{O}(n^{2/3})$ worst-case update time. Finally, we give a fully dynamic algorithm to maintain a $\frac{1}{2} - o(1)$-approximate cut via sparsification.

Chapter 2

A deterministic worst-case update time algorithm

2.1 Preliminaries

Starting with an empty graph $G = (V, E)$ where $V = \{v_1, ..., v_n\}$ is fixed, an update to G is either an insertion or a deletion of an edge $\{v_i, v_j\}$ from E. For a cut $C = (S, \bar{S})$ let $\alpha_C(G) = \frac{|E(S,\bar{S})|}{|E|}$ denote the ratio of the sizes of $C's$ cut-set and E. The sizes of sets $S, \bar{S}$ and the cut-set $E(S, \bar{S})$ corresponding to the cut $C = (S, \bar{S})$ are maintained by all algorithms to facilitate queries in constant time. Let $G_k = (V, E_k)$ be the resulting graph after k updates have been made to $G := G_0$ and $m = |E_k|$ denote the number of edges in G. The degree of any vertex v in G_k is denoted by $deg_k(v)$.

The cut on G_0, the empty graph is initialized to $(V, \emptyset)$. Given a $\frac{1}{2}$-approximate cut $C = (S, \bar{S})$, i.e. $\alpha_C(G_{k-1}) \geq \frac{1}{2}$ for some $k \geq 1$, there are a few cases to consider when an edge update $\{v_i, v_j\}$ is made to G_{k-1}. Deletion of a non-cut edge or insertion of a cut edge never decreases the size of $C's$ cut-set. However, C needs to be updated if a cut edge is deleted, or a non-cut edge is inserted since C may cease to be $\frac{1}{2}$-approximate.

2.1.1 A crucial observation

We say that a vertex u is switched (with respect to a cut $C = (S, \bar{S})$) if u is in S (resp. $\bar{S}$) and moved to $\bar{S}$ (resp. S). We leverage the existence of vertices which can be switched to increase the size of the cut-set $|E(S, \bar{S})|$ of C for *any* cut C which is not $\frac{1}{2}$-approximate. Thus, if C ceases to be $\frac{1}{2}$-approximate following any update

there exists a vertex which can be switched to restore the $\frac{1}{2}$-approximation.

Definition 8 (Switching vertex)**.** *For a cut $C = (S, \bar{S})$, let $N_S(u) = \{v \in S | (u, v) \in E\}$ be the neighbors of u in S and $N_{\bar{S}}(u) = \{v \in \bar{S} | (u, v) \in E\}$ be the neighbors of u in $\bar{S}$. Then u is a switching vertex if one of the following two conditions holds: i) $u \in S$ and $|N_S(u)| - |N_{\bar{S}}(u)| \geq 1$ and ii) $u \in \bar{S}$ and $|N_{\bar{S}}(u)| - |N_S(u)| \geq 1$.*

Theorem 9. *Let C be a $\frac{1}{2}$ approximate cut w.r.t. G_{k-1} i.e., $\alpha_C(G_{k-1}) \geq \frac{1}{2}$ and $\{v_i, v_j\}$ be an update. If $\alpha_C(G_k) < \frac{1}{2}$, then there exists a switching vertex u w.r.t. C such that if u is switched, then $\alpha_C(G_k) \geq \frac{1}{2}$.*

Proof. Suppose there does not exist a switching vertex. Then,

$$\alpha_C(G_k) = \frac{1}{2|E|} \sum_{u \in V} \max\{N_S(v), N_{\bar{S}}(v)\} \geq \frac{1}{2|E|} \sum_{v \in V} \frac{1}{2} deg_k(v) = \frac{1}{4|E|} 2|E| = \frac{1}{2}.$$

clearly contradicting our assumption that $\alpha_C(G_k) < \frac{1}{2}$. If a switching vertex u is switched, then the size of $C's$ cut set increases by at least 1 so that $\alpha_C(G_k) \geq \frac{1}{2}$. □

Given the count of a vertex's neighbors in S and $\bar{S}$, it can be decided whether it is switching or not. Maintaining these neighbor counts is necessary to determine a vertex to switch. However, testing all vertices whether they are switching is costly. In the next section we show how to efficiently maintain a set of switching vertices. The following theorem rules out the possibility of using end points of the updated edge as switching vertices.

2.1.2 Endpoints of an updated edge may not be switching

Theorem 10. *Given an edge update $\{v_i, v_j\}$ to G_{k-1} for $k \geq 1$, and a $\frac{1}{2}$-approximate cut C_{k-1} maintained on G_{k-1}, a switching vertex with respect to C_{k-1} need not always be one of v_i, v_j.*

Proof. We refer to Figures 2.1, and 2.2 for the sake of illustration. Let $V = \{v_1, ..., v_9\}$ be the set of vertices such that $S = V, \bar{S} = \emptyset$. Consider the following sequence of edge insertions $\{v_1, v_6\}, \{v_1, v_7\}, \{v_2, v_7\}, \{v_3, v_7\}, \{v_3, v_8\}, \{v_3, v_9\}, \{v_4, v_9\}, \{v_5, v_8\}$ which leads to v_6, v_7, v_8, v_9 moving to $\bar{S}$ in that order, as a result. Next, consider the following non-cut edge insertions in no particular order: $\{v_1, v_3\}, \{v_2, v_3\}, \{v_3, v_4\}, \{v_6, v_7\}, \{v_7, v_8\}, \{$
The latter set of edge insertions does not make any vertex switching, After the edge $\{v_3, v_5\}$ is added v_3 switches to $\bar{S}$. Now consider the insertion of non-cut edges

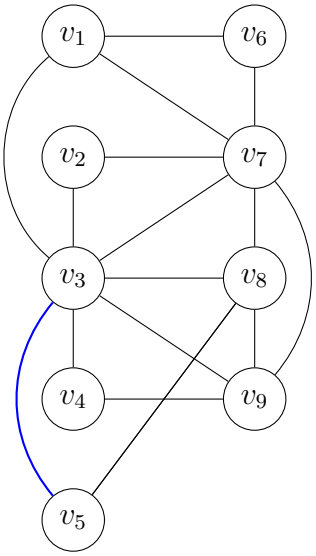

Figure 2.1: The resulting cut immediately after $\{v_3, v_5\}$ is added resulting in a switch.

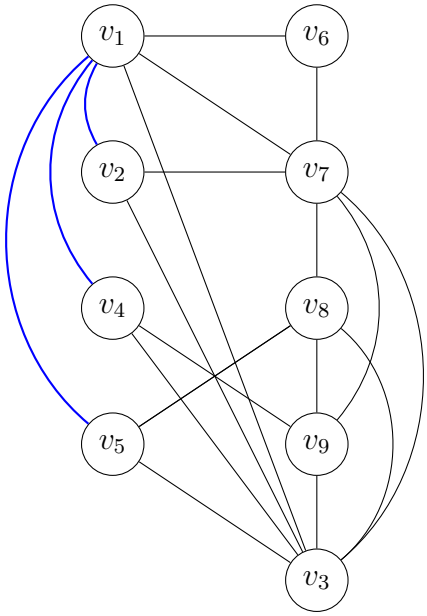

Figure 2.2: A sequence of non cut-edge insertions not making any endpoints switching.

$\{v_1, v_2\}, \{v_1, v_4\}, \{v_1, v_5\}$ so that none of their endpoints namely v_1, v_2, v_4, v_5 become switching. But, $(S, \bar{S})$ is no longer $\frac{1}{2}$-approximate. □

2.2 An $O(\Delta)$ worst-case update time algorithm

In this section, we give a simple fully dynamic algorithm with worst case update time $O(\Delta)$ for Dynamic MAX-CUT.

2.2.1 Data Structures

For each vertex $u \in V$ and a cut $C = (S, \bar{S})$, we maintain the following: i) $N_S(u)$: a list of neighbors of u in S, and its size $|N_S(u)|$, ii) $N_{\bar{S}}(u)$: a list of neighbors of u in $\bar{S}$ and its size $|N_{\bar{S}}(u)|$ and, iii) $flag(u)$: a bit which is 1 if $u \in S$ and -1 if $u \in \bar{S}$.

Definition 11 (Gain of a vertex)**.** *The gain of a vertex u with respect to a cut $C = (S, \bar{S})$ and denoted by $\mathcal{G}(u)$ is given by $\mathcal{G}(u) = flag(u)(|N_S(u)| - |N_{\bar{S}}(u)|)$.*

The gain of a vertex u measures the change in the number of cut edges of C, if u is switched. Note that a vertex is switching if the gain is positive, and non-switching

otherwise.
The following (global) data structures are also maintained:

a. A doubly linked list $\mathcal{L}$, which stores nodes corresponding to switching vertices.

b. An array $\mathcal{P}$ where $\mathcal{P}[i]$ stores the gain of v_i and a pointer. The pointer points to the node in $\mathcal{L}$ corresponding to v_i if $\mathcal{G}(v_i) > 0$ and is $NULL$ otherwise.

The head of $\mathcal{L}$, denoted by $\mathcal{L}$.head is $NULL$ if no switching vertex exists. Each node of $\mathcal{L}$ corresponding to a switching vertex v_i stores i as its value.

2.2.2 Algorithm

The algorithm begins with G_0, the empty graph and $C = (S, \bar{S}) = (V, \emptyset)$ on G_0. It maintains a $\frac{1}{2}$ cut on G_{k-1} for any $k \geq 1$ as follows: when an edge update $\{v_i, v_j\}$ to G_{k-1} arrives, $N_S(v_i), N_{\bar{S}}(v_i), N_S(v_j), N_{\bar{S}}(v_j)$ are updated (including their sizes) along with $\mathcal{P}[i]$ and $\mathcal{P}[j]$. If either of v_i, v_j become switching or non-switching, $\mathcal{L}$ is appropriately modified. C is checked if it is $\frac{1}{2}$-approximate. If C ceases to be $\frac{1}{2}$-approximate then a switching vertex v_s is found by accessing the node pointed to by $\mathcal{L}.head$ which stores the value s. This node is removed from $\mathcal{L}$, v_s is switched and $\mathcal{P}[s]$ is updated. Data structures of v_t and $\mathcal{P}[t]$ of all neighbors v_t of v_s are modified to reflect $v_s's$ switch. Thereafter, depending on whether or not $\mathcal{G}(v_t) > 0$ in the updated cut, the node corresponding to v_t in $\mathcal{L}$ is inserted or removed.

2.2.3 Running Time

Updates to data structures of v_i, v_j and $\mathcal{P}[i], \mathcal{P}[j]$ take constant time. Inserting or removing a node from $\mathcal{L}$ also takes constant time. Switching v_s in the case when C is no longer $\frac{1}{2}$-approximate takes time proportional to updating all its neighbors' data structures, their corresponding entries in $\mathcal{P}$ and their corresponding nodes in $\mathcal{L}$. This takes $O(\Delta)$ time. Theorem 3 follows.

The pseudo code of the algorithm is given as follows.

Algorithm 1 Delta-Dynamic Max-Cut($G_{k-1}, \{v_i, v_j\}, C = (S, \bar{S})$)

Update $N_S(v_i), N_S(v_j), N_{\bar{S}}(v_j), N_{\bar{S}}(v_j), \alpha_C(G_k), \mathcal{P}[i], \mathcal{P}[j]$.
for $v_t \in \{v_i, v_j\}$ **do**
    Add(remove) the node corresponding to v_t in $\mathcal{L}$ if v_t becomes switching(non-switching).
if $\alpha_C(G_k) < \frac{1}{2}$ **then**
    $v_s \leftarrow \mathcal{L}.head$. Remove v_s from L.
    Switch v_s and update $C, flag(v_s), N_S(v_s), N_{\bar{S}}(v_s), \mathcal{P}[s]$.
    for $v_t \in N_S(v_s) \cup N_{\bar{S}}(v_s)$ **do**
        Update $N_S(v_t)$ and $N_{\bar{S}}(v_t)$ as appropriate.
        Add(remove) the node corresponding to v_t in $\mathcal{L}$ if v_t becomes switching(non-switching) and update $\mathcal{P}[t]$.
return v_s.

Chapter 3

Achieving sublinear (in m) update time

In the worst case, an adversarially constructed sequence of updates can potentially make many vertices switching and others non-switching. Thus, breaking the $O(\Delta)$ barrier is difficult (in the worst-case) since switching a high degree vertex can lead to many updates. To obtain sublinear update time, new techniques are warranted.

The *high level* idea underlying our algorithm of Theorem 4 involves working with *stale information* and a *carefully constructed partition* of V. The graph G is partitioned into two vertex-disjoint subgraphs G_1 and G_2, on low and high degree vertices respectively. The algorithm of Theorem 3 is applied separately on G_1 and G_2 whenever the updated edge is incident to nodes in the same subgraph. However, when a high degree vertex switches w.r.t. the cut maintained on C_2, updating all its neighbors' data structures is costly. Thus, data structures of only its high degree neighbors are updated leading to stale information stored in the data structures of low degree neighbors. Together with lazy updating of low degree vertex data structures and, re-initializing G_1 and G_2 after sufficiently many updates leads to sublinear amortized updated time.

Given $\frac{1}{2}$-approximate cuts on vertex disjoint induced subgraphs of any graph G, we first show that they can be combined to yield a $\frac{1}{2}$-approximate with respect to G.

Theorem 12 (Cut combining)**.** *Let $G = (V, E)$ be any graph and $C_1 = (S, \bar{S})$ and $C_2 = (T, \bar{T})$ be $\frac{1}{2}$-approximate cuts with respect to the vertex disjoint induced subgraphs $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ of G such that $S \cup \bar{S} = V_1$, $T \cup \bar{T} = V_2$ and $V_1 \cup V_2 = V$. Then one of the following is a $\frac{1}{2}$-approximate cut C of G:*

i) $(S \cup T, \bar{S} \cup \bar{T})$

ii) $(S \cup \bar{T}, \bar{S} \cup T)$.

Proof Idea: Consider the sets, $S, \bar{S}, T$ and $\bar{T}$ to be super-nodes (i.e. those containing multiple vertices). The set E can be partitioned into edges of three types: i) edges in the cut-sets of C_1 and C_2, ii) edges incident to the same supernode and iii) edges of the form (u, v) where $u \in \{S, \bar{S}\}$ and $v \in \{T, \bar{T}\}$. Edges of the first type remain cut-edges of C, which are at least half the number of edges of the first and second type. Considering one of the two possible combinations of the supernodes ensures that the cut-set of C contains at least half the edges of the third type.

Proof. Let $|E(U, V)|$ denote the number of edges between any 2 subsets U, V of G. We show that a $\frac{1}{2}$-approximate cut C of G can be found by considering the following quantities: $|E(S, T)|, |E(S, \bar{T})|, |E(\bar{S}, T)|, |E(\bar{S}, \bar{T})|$. Essentially, if

1. $|E(S, \bar{T})| + |E(\bar{S}, T)| \geq |E(S, T)| + |E(\bar{S}, \bar{T})|$, then $C = (S \cup T, \bar{S} \cup \bar{T})$ is a $\frac{1}{2}$-approximate cut with respect to G.
2. $|E(S, T)| + |E(\bar{S}, \bar{T})| \geq |E(S, \bar{T})| + |E(\bar{S}, T)|$, then $C = (S \cup \bar{T}, \bar{S} \cup T)$ is a $\frac{1}{2}$-approximate cut with respect to G.

We prove the first case since the other is symmetric. Note that since C_1, C_2 are $\frac{1}{2}$-approximate cuts with respect to G_1, G_2, it holds that

$$\begin{aligned} |E| &\leq 2|E(S, \bar{S})| + 2|E(T, \bar{T})| + |E(S, T)| + |E(S, \bar{T})| + |E(\bar{S}, T)| + |E(\bar{S}, \bar{T})| \\ &\leq 2|E(S, \bar{S})| + 2|E(T, \bar{T})| + 2(|E(S, \bar{T})| + |E(\bar{S}, T)|) \\ &= 2|E(S \cup T, \bar{S} \cup \bar{T})| = 2|E(C)| \end{aligned}$$

Dividing the inequalities by 2 yields the result. □

We now describe the data structures required by our algorithm.

3.1 Data Structures

For any $U, V \subseteq V$, let $E(U, V)$ be the set of all edges having one endpoint in U and the other in V. Let G_1 and G_2 denote the induced subgraphs on V_{low} and V_{high}, the set of low and high degree vertices respectively. Denote by $C_1 = (S, \bar{S})$ and $C_2 =$

$(T, \bar{T})$, the $\frac{1}{2}$-approximate cuts on G_1 and G_2 respectively. To use cut combining, the following edge counts need to be maintained: $|E(S,T)|, |E(S,\bar{T})|, |E(\bar{S},T)|, |E(\bar{S},\bar{T})|$. If $|E(S,\bar{T})| + |E(\bar{S},T)| \geq |E(S,T)| + |E(\bar{S},\bar{T})|$, then $C = (S \cup T, \bar{S} \cup \bar{T})$, else we take $C = (S \cup \bar{T}, \bar{S} \cup T)$. Each vertex $v \in V_{low}$ stores a separate list of neighbors in T and $\bar{T}$ denoted by $N_T(v), N_{\bar{T}}(v)$ and similarly every vertex $v \in V_{high}$ stores a list of its neighbors in S and $\bar{S}$ denoted by $N_S(v), N_{\bar{S}}(v)$. This is in addition to the data structures required to maintain $\frac{1}{2}$-approximate cuts on G_{low} and G_{high} using the algorithm of Theorem 3. Note that for any $U, V \subseteq V$ s.t. $U \in \{S, \bar{S}\}$ and $V \in \{T, \bar{T}\}$, we have that $|E(U,V)| = \sum_{u \in U} |N_V(u)|$.

3.2 Algorithm

The algorithm consists of phases. The k^{th} phase for $k \geq 1$ begins with the graph G containing m_k edges and $\frac{1}{2}$-approximate cuts C_1 and C_2 on the induced subgraphs G_1 and G_2 respectively. G_1 and G_2 are induced subgraphs on $V_{low} = \{v \in V | deg(v) \leq m_k^{1/2}\}$ and $V_{high} = V \backslash V_{low}$ respectively. We assume that the first phase starts with a single edge, i.e. $m_1 = 1$. Each phase consists of $m_k^{1/2}$ updates after which a new phase corresponding to the new value of m_k begins. Thereafter, all data structures are reinitialized and $\frac{1}{2}$-approximate cuts are computed for G_1 and G_2 (under the new value of m_k). The total time taken to reinitialize a phase is $O(m_k)$, leading to $O(m_k^{1/2})$ amortized update time.

Note that the number of high degree vertices for any phase beginning with m_k edges is bounded by $|V_{high}| = O(m_k)/\Omega(m_k^{1/2}) = O(m_k^{1/2})$. The $O(\Delta)$-update time algorithm of Theorem 3 is used as a subroutine. Let $\{v_i, v_j\}$ be an edge update during the k^{th} phase for $k \geq 1$. Then,

1. if $v_i \in V_{low}, v_j \in V_{high}$: One of $N_T(v_i), N_{\bar{T}}(v_i)$ and one of $N_S(v_j), N_{\bar{S}}(v_j)$ is updated. Additionally, one of the edge counts $|E(S,T)|$, $|E(S,\bar{T})|$, $|E(\bar{S},T)|$, $|E(\bar{S},\bar{T})|$ depending on the position of v_i and v_j in C_1 and C_2 respectively, is updated.

2. if $v_i, v_j \in V_{low}$: the algorithm of Theorem 3 is used to restore C_1. Let u be a vertex which is switched w.r.t C_1. All data structures of high degree neighbors of $u \in N_T(u) \cup N_{\bar{T}}(u)$ are updated. Moreover, u recomputes the lists of its high degree neighbors $N_T(u), N_{\bar{T}}(u)$. The edge counts $|E(S,T)|$, $|E(S,\bar{T})|$, $|E(\bar{S},T)|$, $|E(\bar{S},\bar{T})|$ are updated.

3. if $v_i, v_j \in V_{high}$: the algorithm of Theorem 3 is used to restore C_2. The edge counts $|E(S,T)|$, $|E(S,\bar{T})|$, $|E(\bar{S},T)|$, $|E(\bar{S},\bar{T})|$ are updated.

3.2.1 Running Time

If an update $\{v_i, v_j\}$ is such that $v_i \in V_{low}, v_j \in V_{high}$, the update time is $O(1)$. If $v_i, v_j \in V_{low}$ the call to the $O(\Delta)$ update time algorithm takes time $O(m_k^{1/2})$ since any vertex in V_{low} has degree at most $2m_k^{1/2} = O(m_k^{1/2})$ throughout the phase, by definition. This holds for the switched vertex u, for which updating the list of its high degree neighbors and their data structures takes $O(m_k^{1/2})$ time. Updating edge counts takes constant time since they are incremented or decremented by constants which can be determined from the size of neighbor lists of the switched vertex.

If $v_i, v_j \in V_{high}$: the call to the $O(\Delta)$ update time algorithm takes $O(m_k^{1/2})$ time since $|V_{high}| = O(m_k^{1/2})$. As in the second case, updating edge counts takes constant time.

Thus, the time taken to handle an edge update during a phase beginning with m_k edges is $O(m_k^{1/2})$. A proof of correctness and the full pseudo code can be found in the next section. Since the amortized cost of re-initialization is $O(m_k^{1/2})$, this gives an $O(m^{1/2})$ amortized update time algorithm where m denotes the maximum number of edges in G during an arbitrary sequence sequence of updates. Theorem 4 follows.

3.3 Pseudo code and Proofs

The algorithm to handle any edge update within a phase is given as follows, whose inputs include the edge update $\{v_i, v_j\}$, and the $\frac{1}{2}$-approximate cuts $C_1 = (S, \bar{S})$ and $C_2 = (T, \bar{T})$ maintained on $G_1 = (V_{low}, E(V_{low}, V_{low}))$ and $G_2 = (V_{high}, E(V_{high}, V_{high}))$ respectively.

Algorithm 2 Sublinear Max-Cut $(\{v_i, v_j\}, C_1 = (S, \bar{S}), C_2 = (T, \bar{T}))$

1: **if** $v_i \in V_{low}$ and $v_j \in V_{high}$ **then**
2: Update $|E(S,T)|, |E(S,\bar{T})|, |E(\bar{S},T)|, |E(\bar{S},\bar{T})|, N_T(v_i), N_{\bar{T}}(v_i), N_S(v_j), N_{\bar{S}}(v_j)$.
3: **else**
4: **if** $v_i, v_j \in V_{low}$ **then**
5: $u \leftarrow$ Delta-Dynamic Max-Cut$(G_1, \{v_i, v_j\}, C_1)$.
6: **for** $w \in N_T(u) \cup N_{\bar{T}}(u)$ **do**
7: Update $N_S(w)$, $N_{\bar{S}}(w)$ to reflect the new position of u in the cut $(S, \bar{S})$.
8: Update $N_T(u)$, $N_{\bar{T}}(u)$, $|E(S,T)|$, $|E(S,\bar{T})|$, $|E(\bar{S},T)|$, $|E(\bar{S},\bar{T})|$.
9: **if** $v_i, v_j \in V_{high}$ **then**
10: $u \leftarrow$ Delta-Dynamic Max-Cut$(G_2, \{v_i, v_j\}, C_2)$.
11: Update $|E(S,T)|$, $|E(S,\bar{T})|$, $|E(\bar{S},T)|$, $|E(\bar{S},\bar{T})|$.

Lemma 13. *Let m denote the number of edges in G at the beginning of the k^{th} phase. Any call to Algorithm 2 during the k^{th} phase runs in $O(m^{1/2})$ time.*

Proof. Let m denote the total number of edges at the beginning of the k^{th} phase. Let $\{v_i, v_j\}$ denote an update during the k^{th} phase. Low degree vertices have at most $O(m^{1/2})$ neighbors throughout the phase since the phase consists of $m^{1/2}$ updates. In the case when $v_i \in V_{low}$ and $v_j \in V_{high}$, the update time is $O(1)$. If $v_i, v_j \in V_{low}$, then lines 5-8 take total time $O(m^{1/2})$; the call to Algorithm 1 with G_1 as argument takes $O(m^{1/2})$ since G_1 only consists of low degree vertices. Lines 6-7 only consider neighbors of some low degree vertex u which might have switched during the call to Algorithm 1. The number of neighbors of a low degree vertex are bounded by $O(m^{1/2})$ by definition. Line 9 takes $O(m^{1/2})$ time-this is because whenever $u's$ high degree neighbors switch w.r.t. C_2, $u's$ data structures $N_T(u)$ and $N_{\bar{T}}(u)$ are not modified. This leads to stale information in $N_T(u)$ and $N_{\bar{T}}(u)$ which is necessary to update the edge counts $|E(S,T)|$, $|E(S,\bar{T})|$, $|E(\bar{S},T)|$ and $|E(\bar{S},\bar{T})|$. The size of V_{high} is bounded by $O(m^{1/2})$ by definition. Hence if both $v_i, v_j \in V_{high}$ then the call to Algorithm 1 takes $O(m^{1/2})$ time. Line 13 takes constant time. Thus, the total time taken by Algorithm 2 is $O(m^{1/2})$. □

Lemma 14. *Algorithm 2 correctly maintains the edge counts $|E(S,T)|, |E(S,\bar{T})|, |E(\bar{S},T)|, |E(\bar{S},\bar{T})|$ where $C_1 = (S, \bar{S}), C_2 = (T, \bar{T})$.*

Proof. Assume that the edge counts $(|E(S,T)|, |E(S,\bar{T})|, |E(\bar{S},T)|, |E(\bar{S},\bar{T})|)$ are accurate before Algorithm 2 is executed to handle the edge update $\{v_i, v_j\}$. For $v_i \in V_{low}$

and $v_j \in V_{high}$ let $X \in \{S, \bar{S}\}$, $Y \in \{T, \bar{T}\}$ be such that $v_i \in X, v_j \in Y$. If $\{v_i, v_j\}$ is an edge insertion, then v_i is added to $N_X(v_j)$, v_j to $N_Y(v_i)$ and $|E(X,Y)|$ is increased by 1. On the other hand, if $\{v_i, v_j\}$ is an edge deletion, v_i is removed from $N_X(v_j)$, v_j from $N_Y(v_i)$ and $|E(X,Y)|$ is decremented by 1. Thus, the edge counts are correctly updated in this case.

In the case when $v_i, v_j \in V_{low}$, Algorithm 1 is called in order to handle the edge update with respect to the induced subgraph G_1. Let $u \in V_{low}$ be a switched vertex and let $X, \bar{X} \in \{S, \bar{S}\}$ be such that $u \in X$ moves to $\bar{X}$ after the switch. Now, u may no longer have an accurate count of its neighbors in T and $\bar{T}$ since when high degree neighbors of u possibly switch in previous updates, the data structures of u namely $N_T(u), N_{\bar{T}}(u)$ are not modified. Thus, lists $N_T(u), N_{\bar{T}}(u)$ are updated and for all high degree neighbors w of u, $N_X(w), N_{\bar{X}}(w)$ are also updated to reflect u's switch. Since u switched from X to $\bar{X}$, the sizes of lists $N_X(w), N_{\bar{X}}(w)$ are modified appropriately. For all neighbors $w \in V_{low}$ of u, their data structures due to $u's$ switch to $\bar{X}$ are already updated in the call to Algorithm 1. Since $u's$ neighbor lists are up-to-date, the counts $|E(X,T)|, |E(X,\bar{T})|, |E(\bar{X},T)|, |E(\bar{X},\bar{T})|$ are correctly modified.

For the case when $v_i, v_j \in V_{high}$, Algorithm 1 is called in order to handle the edge update with respect to the induced subgraph G_2. Let $u \in V_{high}$ be a vertex which switches and let $Y, \bar{Y} \in \{T, \bar{T}\}$ be such that $u \in Y$ before the update and switches to $\bar{Y}$. Vertices in V_{high} are updated to reflect the switch of u with respect to the cut $(T, \bar{T})$ during the call to Algorithm 1. Since u is a high degree vertex, the neighbor lists $N_S(u), N_{\bar{S}}(u)$ are always up-to-date since if u's low degree neighbors switch during any update, $N_S(u), N_{\bar{S}}(u)$ are also updated. Thus, the edge counts $|E(X,T)|, |E(X,\bar{T})|, |E(\bar{X},T)|, |E(\bar{X},\bar{T})|$ are correctly updated. □

Chapter 4

Achieveing sublinear (in n) worst-case update time

In this chapter we give a randomized algorithm for Dynamic MAX-CUT with $\tilde{O}(n^{2/3})$ worst case update time. We obtain the result as follows. First, we design an algorithm with $O(n^{2/3})$ *expected worst-case* update time by maintaining $\frac{1}{2}$-approximate cuts on subgraphs induced by a random k-partition of V. These cuts are then combined to yield a $\frac{1}{2}$-approximate cut for G. We then apply the probability amplification result of Bernstein el al. [8] to get a $\tilde{O}(n^{2/3})$ worst-case update time w.h.p.

Given a graph G on n vertices, a random partition of V into k sets is first constructed using the oracle described in Section 1.2. Let $(V_1, V_2, ..., V_k)$ denote the k-partition of V such that $\bigcup_{i=1}^{k} V_i = V$ and $|V_1| = |V_2| = ... = |V_{k-1}| = \lceil n/k \rceil$, $|V_k| = n - (k-1)\lceil n/k \rceil$. On each subgraph G_i induced by V_i, a $\frac{1}{2}$ approximate cut $C_i = (S_i, \bar{S}_i)$ (where $\bar{S}_i = V_i \backslash S_i$) is dynamically maintained using the algorithm of Theorem 3.

4.1 Data structures

In addition to data structures required by the $O(\Delta)$-update time algorithm, we maintain the following data structures. For each vertex $v \in V$, lists of its neighbors in each S_i, (denoted by $N_{S_i}(v)$) and $\bar{S}_i$ (denoted by $N_{\bar{S}_i}(v)$) for all $1 \leq i \leq k$ are maintained. For all $1 \leq i, j \leq k$, we maintain the edge counts $|E(S_i, S_j)|$, $|E(S_i, \bar{S}_j)|$ $|E(\bar{S}_i, S_j)|$, $|E(\bar{S}_i, \bar{S}_j)|$ for a total of $\binom{2k}{2} = O(k^2)$ counts, where $E(U, V)$ for $U, V \subseteq V$ is defined as in the previous chapter. These counts can be maintained using the size of neighbor

lists maintained for each vertex.

4.2 Algorithm

4.2.1 Cut combining

We first describe how to combine $\frac{1}{2}$-approximate cuts C_i on G_i for $1 \leq i \leq k$ to get a $\frac{1}{2}$-approximate cut C, on G. Initially, $C = (S_1, \bar{S_1})$. Whenever considering cut $C_i = (S_i, \bar{S_i})$ for $2 \leq i \leq k$ to combine with C, the edge counts $|E(S_i, S_j)|$, $|E(S_i, \bar{S_j})|$, $|E(\bar{S_i}, S_j)|$, $|E(\bar{S_i}, \bar{S_j})|$, for $1 \leq j \leq i-1$ are used to compute the edge counts $|E(S, S_i)|$, $|E(S, \bar{S_i})|$, $|E(\bar{S}, S_i)|$, $|E(\bar{S}, \bar{S_i})|$. Depending on the combination which maximizes $|E(S, \bar{S})|$, either S_i (resp. $\bar{S_i}$) is added to S (resp. $\bar{S}$) or S_i (resp. $\bar{S_i}$) is added to $\bar{S}$ (resp. S). Computing the edge counts takes $O(k)$ time, yielding $O(k^2)$ time to compute C.

4.2.2 Update algorithm

Let $\{v_i, v_j\}$ be an edge update. Then,

1. if $v_i \in V_p$ and $v_j \in V_q$ s.t. $p \neq q$: Only the lists $N_{S_q}(v_i), N_{\bar{S_q}}(v_i), N_{S_p}(v_j), N_{\bar{S_p}}(v_j)$ and edge counts $|E(S_p, S_q)|, |E(S_p, \bar{S_q})|, |E(\bar{S_p}, S_q)|, |E(\bar{S_p}, \bar{S_q})|$ are updated which takes $O(1)$ time.

2. if $v_i, v_j \in V_p$ for some p: the cut C_p is updated using the $O(\Delta)$ update time algorithm. Let u be the switched vertex w.r.t C_p. The lists $N_{S_p}(w), N_{\bar{S_p}}(w)$ of all neighbors w of u are updated to reflect $u's$ switch. For all $1 \leq q \leq k$ such that $N_{S_q}(u) \cup N_{\bar{S_q}}(u) \neq \emptyset$, edge counts of the form $|E(S_p, S_q)|$, $|E(S_p, \bar{S_q})|$, $|E(\bar{S_p}, S_q)|$, $|E(\bar{S_p}, \bar{S_q})|$ are also updated. This can be done by using the values of $|N_{S_q}(u)|$ and $|N_{\bar{S_q}}(u)|$.

Following this, the cuts $C_1, ..., C_k$ are combined to yield C.

Running Time

For the case when $v_i \in V_p$ and $v_j \in V_q$ s.t. $p \neq q$ updating the edge counts takes constant time and combining cuts takes $O(k^2)$ time. Note that a single update can cause the cuts to combine differently in order to compute a $\frac{1}{2}$-approximate cut on G, and this is why the combining cost is incurred in both cases.

For the case when $v_i, v_j \in V_p$ for some p, the algorithm of Theorem 3 takes $O(n/k)$ time. Let u be the switched vertex w.r.t. C_p. Updating the neighbor lists of all neighbors of u takes $O(\Delta)$ time. The total update time in this case is $O(\Delta + \frac{n}{k} + k^2) = O(\Delta + k^2)$.

We now give the pseudo-code of the algorithm and analyze its running time.

4.3 Pseudo-code and analysis of running time

Algorithm 3 Randomized Sublinear MAX-CUT $(\{v_i, v_j\}, G_1, ..., G_k, C_1, ..., C_k)$

1: **if** $v_i \in V_p, v_j \in V_q$ s.t. $p \neq q$ **then**
2: Update $N_{S_q}(v_i), N_{\bar{S}_q}(v_i), N_{S_p}(v_j), N_{\bar{S}_p}(v_j)$.
3: Update $|E(S_p, S_q)|, |E(\bar{S}_p, S_q)|, |E(S_p, \bar{S}_q)|, |E(\bar{S}_p, \bar{S}_q)|$ as required.
4: **else**
5: $u \leftarrow$ Delta-Dynamic Max-Cut$(G_p, \{v_i, v_j\}, C_p)$.
6: **for** all neighbors v of u where $v \in V_r$ for any $1 \leq r \leq k$ **do**
7: Update $N_{S_r}(u), N_{\bar{S}_r}(u), N_{S_p}(v), N_{\bar{S}_p}(v)$.
8: Update $|E(S_p, S_r)|, |E(\bar{S}_p, S_r)|, |E(S_p, \bar{S}_r)|, |E(\bar{S}_p, \bar{S}_r)|$ as required.
9: $S = S_1, \bar{S} = \bar{S}_1$.
10: **for** $t = 2, ..., k$ **do**
11: **if** $|E(S \cup S_t, \bar{S} \cup \bar{S}_t)| \geq |E(S \cup \bar{S}_t, \bar{S} \cup S_t)|$ **then**
12: $S = S \cup S_t$, $\bar{S} = \bar{S} \cup \bar{S}_t$.
13: **else**
14: $S = S \cup \bar{S}_t$, $\bar{S} = \bar{S} \cup S_t$.

We note that the combining step does not need to be explicitly implemented as in lines 10-14 of Algorithm 3 in which sets S and $\bar{S}$ are incrementally constructed using S_t and $\bar{S}_t$. The only information required to determine how to combine the cut $(S_t, \bar{S}_t)$ with $(S, \bar{S})$ in each iteration of the for loop is the position of all $S_i, \bar{S}_i$ for all $i \leq t-1$ in $(S, \bar{S})$. Thus, computing the edge counts $|E(S \cup S_t, \bar{S} \cup \bar{S}_t)|$, $|E(S \cup \bar{S}_t, \bar{S} \cup S_t)|$ can be done in $O(k)$ time. Consequently, the combining cost after a single update is $O(k^2)$.

Lemma 15. *The expected running time of Algorithm 3 is $O(\frac{\Delta}{k} + k^2)$. For $k = \Theta(n^{1/3})$, this yields $O(n^{2/3})$ time in expectation.*

Proof. Let $\{v_i, v_j\}$ be an edge update. The probability that this update is of the second type, i.e. $v_i, v_j \in V_p$ for some $1 \leq p \leq k$ is at most $1/k$. The expected update time, denoted by $E[T(n,k)]$ can be written as,

$$\begin{aligned}
E[T(n,k)] &= \Pr[v_i, v_j \in V_p]O(\Delta + k^2) + \Pr[v_i \in V_p, v_j \in V_q, p \neq q]O(k^2) \\
&= \Pr[v_i, v_j \in V_p]O(\Delta + k^2) + (1 - \Pr[v_i, v_j \in V_p])O(k^2) \\
&= \frac{1}{k}O(\Delta) + O(k^2) \\
&= O(\frac{\Delta}{k} + k^2) \\
&= O(\frac{n}{k} + k^2).
\end{aligned}$$

The value of k which minimizes $E[T(n,k)]$ is $\Theta(n^{1/3})$ yielding $O(n^{2/3})$ expected worst case update time. □

Bernstein et al. [8] give a general technique to convert a fully dynamic data structure with expected worst-case update time to a worst-case update time with high probability. Given an algorithm A to maintain a fully dynamic data structure D with expected worst-case update time α such that the number of elements stored in the data structure are bounded by a polynomial of n, they show how to obtain an algorithm A' which processes each update to the data structure in $\alpha log^2(n)$ worst case update time w.h.p. See [8] for technical details.

By using their technique as a black-box, we can thus convert our randomized algorithm taking $O(n^{2/3})$ expected worst-case update time to one taking $O(n^{2/3}log^2(n)) = \tilde{O}(n^{2/3})$ update time with high probability. Theorem 5 follows.

Chapter 5

Fully Dynamic Degree Sparsifiers

Many static algorithms for MAX-CUT depend on the maximum degree of the input graph G. Indeed, Trevisan's linear time reduction [35] to convert the input graph $G = (V, E)$ into $G' = (V', E')$ with $\tilde{O}(1)$ maximum degree is used as a subroutine in many static algorithms for MAX-CUT including those in [22, 2]. However, the number of vertices in G' can be $2|E|$ due to which it is not applicable in the dynamic setting. This is because in the static setting, a large cut is first computed on G' and then converted to one on G (see [35]). This does not affect the asymptotic time complexity in the static setting as such but it is expensive in the dynamic setting to convert the cut after every update.

We tackle this problem by constructing a sparsifier $H = (V, E')$ which in contrast to Trevisan's reduction, is a *subgraph* of G with maximum degree sublinear in n while preserving the weight of all large cuts (i.e. cuts C s.t. $|E(C)| \geq c|E|$ for any constant $c > 0$) within $(1 \pm o(1))$-multiplicative error. The upside of our method is that a large cut in G stays a large cut in H w.h.p, so no conversion is required. We use the $O(\Delta)$-update time algorithm of Theorem 3 to maintain a $\frac{1}{2}$-approximate cut on H, and this cut is used to answer queries.

Let $G = (V, E)$ be the an undirected and sufficiently dense graph such that $m = |E| = n^{1+\epsilon}$ for some $0 < \epsilon \leq 1$. First we show that there exists a (weighted) subgraph H of G such that the maximum degree of H denoted by Δ_H is $\tilde{O}(n^c)$ for some $\frac{1}{2} < c < 1$ which depends on ϵ. Second, we show that there is a simple process to maintain these sparsifiers in the fully dynamic setting. The following theorems summarize the sparsification result.

Theorem 16 (Degree Sparsifiers)**.** *Given a graph $G = (V, E)$ on n vertices and*

$m = n^{1+\epsilon}$ edges where $0 < \epsilon < 1$, there exists a sublinear (in n) maximum degree sparsifier H of G, which can be computed in $O(m)$ time. H satisfies the following properties:

1. *$\Delta_H = \tilde{O}(n^c)$ where $c = 1 - \frac{\epsilon}{2} + 2c_0$ for any constant $c_0 > 0$.*
2. *The weight of all cuts in G is preserved in H within $\delta|E|$ additive error with probability at least $1 - \frac{1}{n^d}$ for any constant $d > 0$ where $\delta \geq \frac{(d+1)^4}{n^{c_0}}$. For n sufficiently large, this implies that all large cuts are preserved within $(1 \pm o(1))$ multiplicative error w.h.p.*

Theorem 6. *Given a graph $G = (V, E)$ on n vertices and $m = n^{1+\epsilon}$ edges where $0 < \epsilon \leq 1$, there exists a fully dynamic algorithm to maintain a sublinear (in n) maximum degree sparsifier H of G, which takes $O(n^{1+\epsilon})$ preprocessing time and constant update time. At all times, $\Delta_H = \tilde{O}(n^c)$ where $c = 1 - \frac{\epsilon}{2} + 2c_0$ for any constant $c > 0$ and all large cuts in G are preserved in H within $(1 \pm o(1))$ multiplicative error.*

If $c_0 = \frac{1}{2000}\epsilon$ in the result above, then $c = 1 - 0.4999\epsilon$, so as $c_0 \to 0$, then $c \to 1 - \frac{\epsilon}{2}$. For some representative values of $\epsilon = 0.25, 0.5, 0.75, 0.9$ and 0.95, the update times (up to three decimal places) with the above choice of c_0 are $\tilde{O}(n^{0.875})$, $\tilde{O}(n^{0.750})$, $\tilde{O}(n^{0.626})$, $\tilde{O}(n^{0.550})$ and $\tilde{O}(n^{0.525})$. For $\epsilon > 2/3 + 4c_0$, sparsification yields an algorithm with update time better than the algorithm of Theorem 5. Theorem 7 follows.

Theorem 7. *There exists a fully dynamic randomized algorithm which maintains a $(\frac{1}{2} - o(1))$-approximate cut and takes $\min\{\tilde{O}(n^{2/3}), \tilde{O}(\frac{n^{3/2+2c_0}}{m^{1/2}})\}$ worst case update time for any constant $c_0 > 0$ with high probability.*

5.1 Construction in the static setting

The main ideas of the construction are as follows. Given a graph $G = (V, E)$ on $m = n^{1+\epsilon}$ edges, each edge is sampled independently in H with probability $p = \frac{\delta}{n^{1-c}}$ and given a weight $1/p$ where $\delta > 0$ measures the quality of cut approximation in H. Thus, the expected degree of any vertex in H is bounded by δn^c and by a standard Chernoff bound, it follows that $\Delta_H = \tilde{O}(n^c)$ w.h.p.

We prove that the weight of any cut in H is within $\pm\delta|E|$ of its expectation. The expected weight of any cut C in H is $|E(C)|$ since the expected weight of any edge in H is 1. The proof consists of two steps. First, we fix a cut C and show that the

weight of C in H is well concentrated within $\delta|E|$ additive error of its expectation using a Chernoff bound. Second, we take a union bound over all exponentially many cuts to conclude that the result holds for all cuts w.h.p. Lastly, we show that $\delta = o(1)$ (for sufficiently large n) whenever $c = 1 - \frac{\epsilon}{2} + 2c_0$ for any constant $c_0 > 0$. We now give a proof of Theorem 16.

Proof. The sparsification process is remarkably simple: every edge $e \in E$ is independently sampled with probability $p = \frac{\delta}{n^{1-c}}$ and given a weight $w = 1/p$ in H, where c is a constant depending on ϵ. Let Δ_H denote the maximum degree of H. Then it follows that, $E[\Delta_H] \leq \frac{\delta}{n^{1-c}} n = \delta n^c$ and it is easy to check $\Delta_H = \tilde{O}(n^c)$ with high probability. We now show that the weight of *every* cut in H is concentrated within $\pm\delta|E|$ of its expectation w.h.p by using the following additive version of the Chernoff bound.

Fact 17. *(Additive Chernoff Bound) Let X be a random variable which is a sum of random variables X_i such that $X = \sum_{i=1}^{n} X_i$ where each $X_i \in [a_i, b_i]$, then the following holds for any $t > 0$:*

$$Pr[X \notin (E[X] - t, E[X] + t) \leq 2exp(\frac{-2t^2}{\sum_{i=1}^{n}(a_i - b_i)^2})$$

We first fix a cut C and show that the weight of C in H is concentrated around its expectation. Taking a union bound over all (exponentially many) cuts concludes the result. Let X_e be the indicator random variable corresponding to edge e which is $\frac{1}{p} = \frac{n^{1-c}}{\delta}$ with probability $p = \frac{\delta}{n^{1-c}}$ and 0 otherwise. Let $X_H(C), X_G(C)$ denote the random variable denoting the weight of the cut C in H and the number of cut edges in the cut set of C in G, respectively. Then, $X_H(C) = \sum_{eE(C)} X_e$ and by linearity of expectation, $E[X_H(C)] = \sum_{e \in E(C)} E[X_e] = X_G(C)$. The squared range in the denominator of the expression on the RHS of the Chernoff bound is at most $|E|(\frac{1}{p})^2 = \frac{|E|n^{2-2c}}{\delta^2}$ which upper bounds the probability. Using the fact that $|E| = n^{1+\epsilon}$, we get:

$$\begin{aligned} Pr[X_H(C) \notin (E[X_H(C)] - \delta|E|, E[X_H(C)] + \delta|E|)] &\leq 2exp(\frac{-2\delta^2|E|^2}{|E|\frac{n^{2-2c}}{\delta^2}}) \\ &= \frac{2}{exp(2\delta^4 n^{2c+\epsilon-1})} \end{aligned}$$

Taking a union bound over all 2^{n-1} cuts, we get that,

$$Pr[\exists C, X_H(C) \notin (E[X_H(C)] - \delta|E|, E[X_H(C)] + \delta|E|)] \leq \frac{2^n}{exp(2\delta^4 n^{2c+\epsilon-1})} \leq \frac{1}{2^{2\delta^4 n^{2c+\epsilon-1}-n}}$$

If this probability is desired to be at most $\frac{1}{n^d}$ for some constant d, then $2^{2\delta^4 n^{2c+\epsilon-1}-n} \geq n^d$, from which it follows that $\delta^4 \geq \frac{d\log n+n}{2n^{2c+\epsilon-1}}$. Since the latter expression is at most $\frac{(d+1)n}{n^{2c+\epsilon-1}}$, we have that $\delta \geq \frac{(d+1)^{1/4}}{n^{\frac{2c+\epsilon-2}{4}}}$. For δ to approach zero as n grows large, we require that $\frac{2c+\epsilon-2}{4} = c_0$ for a constant $c_0 > 0$ so that $c = 1 - \frac{\epsilon}{2} + 2c_0$. □

5.2 Maintaining sparsifiers dynamically

We now describe the ideas behind the process of making the sparsification process fully dynamic. The results hold when updates comes from an oblivious adversary. The update sequence is divided into phases. A phase starts with the input graph having $m_i = n^{1+\epsilon}$ edges for some $0 < \epsilon < 1$. A phase ends (and a new phase is started) when the number of edges in G becomes either $\frac{1}{2}m_i$ or $2m_i$. For this section we assume that the number of updates in each phase are polynomial i.e., bounded by n^k for sufficiently large k. Later we show how exponential number of updates can be handled. First, we consider how to handle updates within a single phase. The following algorithm is executed when (u, v) is an edge update to G where G has $m = O(n^{1+\epsilon})$ edges and c is a constant depending on ϵ.

Algorithm 4 Update-Sparsifier $(H, (u, v))$

```
if (u, v) is an edge deletion then
    if (u, v) ∈ E_H then
        Delete (u, v) from E_H.
else                                          ▷ (u, v) is an edge insertion
    Add (u, v) to E_H with probability δ/n^{1−c} and weight n^{1−c}/δ.
```

To prove that Algorithm 4 preserves the weight of any cut within $\pm o(1)|E|$ additive error of its expectation during *any* phase which terminates after a polynomial number of updates, we need the following lemma.

Lemma 18. *Fix a set of updates U and let G and H denote the input graph and sparsified subgraph of G respectively at the beginning of a phase, such that H satisfies*

the properties given in Theorem 16. If Algorithm 4 is used to modify H for updates in U then the expected weight of any cut C with respect to H, is within $\pm o(1)|E|$ additive error of the number of edges in the cut set of C with respect to G (after applying the updates in U) w.h.p.

Proof. We remark that the following argument can be made only when the update sequence U can be fixed in advance. Let G' and H' denote the resulting graph and sparsifier after the set of updates U are applied to G and H respectively and $E_G, E_{G'}, E_H, E_{H'}$ denote their edge sets. Fix a cut C and let $X_G(C), X_H(C)$ as before and define $X_{H'}(C), X_{G'}(C)$ analogously. We first prove that $E[X_{H'}(C)] = X_{G'}(C)$ before arguing for concentration. The execution of Algorithm 4 on the entire update sequence U can be modelled as the following random process where X_e is a random variable corresponding to the updated edge in U. X_e is defined as follows:

1. if e is an edge insertion: $X_e = \frac{n^{1-c}}{\delta}$ with probability $\frac{\delta}{n^{1-c}}$ and 0 otherwise.
2. if e is an edge deletion: $X_e = -\frac{n^{1-c}}{\delta}$ with probability $\frac{\delta}{n^{1-c}}$ and 0 otherwise.

The definition of X_e when e is an edge deletion is valid due to the following reason: the probability that e is deleted from H while executing Algorithm 4 on updates in U is equal to the probability that it was originally sampled in H. Moreover if $e \in E_H$ it is surely deleted. Applying linearity of expectation over the sequence of updates in U yields that $E[X_{H'}(C)] = X_{G'}(C)$. Intuitively, any edge e in the resulting graph after t updates for any $t \leq |U|$ is contained in the sparsifier with probability $\frac{\delta}{n^{1-c}}$. Since the sampling probabilities for edges in G and the update sequence U are the same, by Theorem 16 we conclude that with high probability (for a suitable choice of constants) for any fixed set of updates U such that $|U| \leq n^k$, H' preserves cuts within $\pm o(1)|E|$ error. Indeed, the process of applying the updates to H to yield H' can be equivalently seen as *statically* constructing the sparsifier on G'. □

Lemma 18 holds for a fixed set of updates U. The number of possible update sequences to consider in U are only n^k. Applying yet again, a union bound on all these sequences of updates yields the theorem below.

Theorem 19. *Let G and H denote the input graph and the sparsifier respectively which satisfy the properties in Theorem 16. If the number of updates are polynomial in n, there exists a fully dynamic algorithm to maintain H in constant update time such that the weight of all cuts in H is preserved within $\pm o(1)|E|$ additive error.*

5.2.1 Maintaining sparsifiers between different phases

In this section, we show how sparsifiers can be maintained between different phases as ϵ changes. We assume that a phase terminates after a polynomial number of updates. Let $m_i = n^{1+\epsilon}$ be the number of edges in G at the beginning of a phase where $0 < \epsilon < 1$, m_c be the current number of edges in G and m_f be the number of edges in G at the end of the phase. We maintain an active and passive instance of the sparsifier denoted by H_a and H_p respectively. A passive instance H_p becomes an active instance in the next phase, and is constructed in parallel during the current phase. All queries to the data structure are answered using the active instance, H_a on which a $\frac{1}{2}$-approximate cut is dynamically maintained using Algorithm 1. H_a is modified using Algorithm 4. The passive instance is constructed as follows: If $m_c = m_i$, H_p is empty. If $m_c < m_i$, each edge in G is sampled with probability $\frac{\delta}{n^{1-c'}}$, where $c' = 1 - \frac{\epsilon'}{2} + 2c_0$, $\epsilon' = \epsilon - \frac{1}{log\, n}$ and $c_0 > 0$ is a fixed constant. On the other hand, if $m_c > m_i$ each edge in G is sampled with probability $\frac{\delta}{n^{1-c'}}$, where $c' = 1 - \frac{\epsilon'}{2} + 2c_0$, $\epsilon' = \epsilon + \frac{1}{log\, n}$ and $c_0 > 0$ is a fixed constant. The sampling process is simple: after every update, the edge involved in the update is sampled or deleted from H_p and at most one additional edge which has not been previously sampled in H_p since its last insertion in G is sampled in H_p. So when a phase terminates, all graph edges found in G at that point have been sampled. This ensures that H_p is a sparsifier fulfilling the properties in Theorem 16 and can be used as an active instance in the next phase.

A $\frac{1}{2}$-approximate cut on H_p is also concurrently maintained using Algorithm 1. This is in addition to maintaining a $\frac{1}{2}$-approximate cut on the active instance H_a, which is used to answer queries during the current phase. Since at most a constant number of edges– one in H_a and at most two in H_p– are sampled or deleted, the total update time is $O(n^{c'})$.

5.2.2 Handling exponential number of updates

In this section we show that an exponential number of graph updates can be handled while ensuring the high probability guarantee on cut concentration. So far, a phase was defined as containing only a polynomial number of updates, thus yielding a high probability guarantee by taking a union bound over all possible sequences of updates. To handle exponentially many updates, a phase is now terminated if *either* the number of edges grows or shrinks by a factor of two *or* the number of updates since the beginning of the phase exceed n^k for a fixed sufficiently large constant k. In

particular, let $k = log(m_i)/log(n)$ so that the number of updates are m_i, the number of edges in G at the beginning of a phase.

The high level ideas to handle exponential updates are as follows. We maintain two passive instances of the sparsifier H_{p_1}, H_{p_2} and one active instance H_a as before. The active instance is modified under edge updates to G using Algorithm 3. The first passive instance H_{p_1} is maintained as described in the previous section, based on the assumption that the current phase terminates as a result of the number of edges changing by factor two. On the other hand, H_{p_2} is maintained (in parallel) solely to ensure that the high probability argument on cut concentration holds after m_i updates such that $m_f \in (\frac{1}{2}m_i, 2m_i)$. One of H_{p_1}, H_{p_2} is made the active instance in the next phase.

Edges are sampled in H_{p_2} as follows: Following any edge update involving e, e is either sampled in H_{p_2} or deleted if it was previously sampled. An additional edge currently present in G is sampled in H_{p_2}. This edge is one which is in G at the beginning of the phase and not involved in any update during the current phase. This is because any edge updated in the current phase is already sampled or deleted from H_{p_2}. Deciding the additional edges to sample can be done by considering all edges in G at the beginning of the current phase and excluding those that have been involved in any update in the current phase. Moreover, $\frac{1}{2}$-approximate cuts are also maintained on H_{p_1} and H_{p_2} using Algorithm 1. The total update time is thus bounded by $O(n^{c'})$.

We are now ready to prove Theorem 6.

Theorem 6. *Given a graph $G = (V, E)$ on n vertices and $m = n^{1+\epsilon}$ edges where $0 < \epsilon \leq 1$, there exists a fully dynamic algorithm to maintain a sublinear (in n) maximum degree sparsifier H of G, which takes $O(n^{1+\epsilon})$ preprocessing time and constant update time. At all times, $\Delta_H = \tilde{O}(n^c)$ where $c = 1 - \frac{\epsilon}{2} + 2c_0$ for any constant $c > 0$ and all large cuts in G are preserved in H within $(1 \pm o(1))$ multiplicative error.*

Proof. Given a graph $G = (V, E)$ on $n^{1+\epsilon}$ edges we first construct H which takes $O(n^{1+\epsilon})$ time marking the beginning of the first phase. Passive instances are maintained in $\tilde{O}(n^c)$ worst case update time as explained in the previous section. By virtue of the sampling probabilities with which edges are sampled in the active and passive instances, the maximum degree of the active instance is $\tilde{O}(n^c)$ w.h.p. Theorem 19 and Lemma 18 guarantee cut concentration in active and passive instances, concluding the proof. □

Chapter 6

Conclusion

MAX-CUT is a fundamental problem in combinatorial optimization which quite suprisingly has received scant attention by researchers working in fully dynamic algorithms. The goal of this thesis was to initiate the study of the problem in the fully dynamic setting and obtain sublinear update time algorithms. This is more challenging as compared to the static setting. It seems unclear how the primal dual framework of Arora and Kale [2], which achieves a near optimal cut in almost linear time can be modified to handle updates in sublinear update time. In particular, our lack of understanding comes from the fact that their algorithm has certain disparate subroutines for which a fully dynamic generalization is seemingly hard. On the other hand, a general theoretical framework for primal dual semidefinite programming in the dynamic setting may lead to dynamic algorithms for a class of optimization problems for which Arora and Kale give very fast static approximation algorithms [2]. This would be akin to utilizing the primal dual framework used to solve linear programs to design fully dynamic algorithms [10].

We hope the work in this thesis encourages researchers to improve upon the approximation and update time of our algorithms. One concrete open problem which naturally arises from this thesis is determining whether or not there exists a fully dynamic algorithm for MAX-CUT which obtains better than $\frac{1}{2}$-approximation in sublinear update time. Another possibility would be to extend our work to maintain a $\frac{1}{2}$-approximate cut when the graph is weighted. It would also be interesting to investigate the trade-offs between update-time and approximability for the MAX-CUT problem in the dynamic setting. We believe that unconditional or even conditional lower bounds based on popular complexity conjectures would be crucial in informing the design of dynamic algorithms for MAX-CUT.

Bibliography

[1] I. Abraham, D. Durfee, I. Koutis, S. Krinninger, and R. Peng. On fully dynamic graph sparsifiers. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 335–344, Oct 2016.

[2] Sanjeev Arora and Satyen Kale. A combinatorial, primal-dual approach to semidefinite programs. *J. ACM*, 63(2):12:1–12:35, May 2016.

[3] Sepehr Assadi, Yu Chen, and Sanjeev Khanna. Sublinear algorithms for (Δ+ 1) vertex coloring. *CoRR*, abs/1807.08886, 2018.

[4] Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. Fully dynamic maximal independent set with sublinear update time. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2018, pages 815–826, New York, NY, USA, 2018. ACM.

[5] Sepehr Assadi, Krzysztof Onak, Baruch Schieber, and Shay Solomon. Fully dynamic maximal independent set with sublinear in n update time. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '19, pages 1919–1936, Philadelphia, PA, USA, 2019. Society for Industrial and Applied Mathematics.

[6] S. Baswana, M. Gupta, and S. Sen. Fully dynamic maximal matching in o (log n) update time. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 383–392, Oct 2011.

[7] Andras Benczur and David R. Karger. Randomized approximation schemes for cuts and flows in capacitated graphs, 2002.

[8] Aaron Bernstein, Sebastian Forster, and Monika Henzinger. A deamortization approach for dynamic spanner and dynamic maximal matching. *ArXiv*, abs/1810.10932, 2018.

[9] Sayan Bhattacharya, Deeparnab Chakrabarty, Monika Henzinger, and Danupon Nanongkai. Dynamic algorithms for graph coloring. *CoRR*, abs/1711.04355, 2017.

[10] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Dynamic algorithms via the primal-dual method. *Inf. Comput.*, 261(Part):219–239, 2018.

[11] Keren Censor-Hillel, Rina Levy, and Hadas Shachnai. Fast distributed approximation for max-cut. *CoRR*, abs/1707.08496, 2017.

[12] Julia Chuzhoy, Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, and Thatchaphol Saranurak. A deterministic algorithm for balanced cut with applications to dynamic connectivity, flows, and beyond, 2019.

[13] Kevin P. Costello, Asaf Shapira, and Prasad Tetali. Randomized greedy: New variants of some classic approximation algorithms. In *Proceedings of the Twenty-second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '11, pages 647–655, Philadelphia, PA, USA, 2011. Society for Industrial and Applied Mathematics.

[14] Michel X. Goemans and David P. Williamson. .879-approximation algorithms for MAX CUT and MAX 2sat. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing, 23-25 May 1994, Montréal, Québec, Canada*, pages 422–431, 1994.

[15] Gramoz Goranci, Monika Henzinger, and Dariusz Leniowski. A tree structure for dynamic facility location, 2019.

[16] Anupam Gupta, Ravishankar Krishnaswamy, Amit Kumar, and Debmalya Panigrahi. Online and dynamic algorithms for set cover, 2016.

[17] Manoj Gupta and Shahbaz Khan. Simple dynamic algorithms for maximal independent set and other problems. *CoRR*, abs/1804.01823, 2018.

[18] Monika R. Henzinger, Valerie King, and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *J. ACM*, 46(4):502–516, July 1999.

[19] Niklas Hjuler, Giuseppe F. Italiano, Nikos Parotsidis, and David Saulpic. Dominating sets and connected dominating sets in dynamic graphs. In *STACS*, 2019.

[20] Jacob Holm, Kristian de Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM*, 48:723–760, 07 2001.

[21] David S. Johnson. Approximation algorithms for combinatorial problems. *J. Comput. Syst. Sci.*, 9(3):256–278, December 1974.

[22] Satyen Kale and C. Seshadhri. Combinatorial approximation algorithms for maxcut using random walks. *CoRR*, abs/1008.3938, 2010.

[23] Bruce Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1131–1142, 01 2013.

[24] Richard M. Karp. Reducibility among combinatorial problems. In *50 Years of Integer Programming*, 1972.

[25] Casper Kejlberg-Rasmussen, Tsvi Kopelowitz, Seth Pettie, and Mikkel Thorup. Deterministic worst case dynamic connectivity: Simpler and faster. *CoRR*, abs/1507.05944, 2015.

[26] Subhash Khot. On the power of unique 2-prover 1-round games. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, pages 767–775, 2002.

[27] Subhash Khot, Guy Kindler, Elchanan Mossel, and Ryan O'Donnell. Optimal inapproximability results for max-cut and other 2-variable csps? In *45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings*, pages 146–154, 2004.

[28] Claire Mathieu and Warren Schudy. Yet another algorithm for dense max cut: go greedy. In *SODA '08*, 2008.

[29] Ofer Neiman and Shay Solomon. Simple deterministic algorithms for fully dynamic maximal matching, 2012.

[30] Krzysztof Onak and Ronitt Rubinfeld. Maintaining a large matching and a small vertex cover. pages 457–464, 09 2010.

[31] Yossi Shiloach and Shimon Even. An on-line edge-deletion problem. *J. ACM*, 28(1):1–4, January 1981.

[32] Shay Solomon. Fully dynamic maximal matching in constant update time. *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 325–334, 2016.

[33] José A. Soto. Improved analysis of a max cut algorithm based on spectral partitioning. *CoRR*, abs/0910.0504, 2009.

[34] Mikkel Thorup. Fully-dynamic min-cut. In *Proceedings on 33rd Annual ACM Symposium on Theory of Computing, July 6-8, 2001, Heraklion, Crete, Greece*, pages 224–230, 2001.

[35] L. Trevisan. Max cut and the smallest eigenvalue. *SIAM Journal on Computing*, 41(6):1769–1786, 2012.

[36] Christian Wulff-Nilsen. Fully-dynamic minimum spanning forest with improved worst-case update time, 2016.