Fully Dynamic Maintenance of Vertex Cover [1]

(Extended Abstract)

Zoran Ivković[2] Errol L. Lloyd[3]

Abstract

The problem of maintaining an approximate solution for **vertex cover** when edges may be inserted and deleted dynamically is studied. We present a fully dynamic algorithm A_1 that, in an amortized fashion, efficiently accommodates such changes. This algorithm utilizes a special technique, introduced here, of handling the processing of operations (*Insert*s and *Delete*s of edges) in *stages* consisting of a certain predetermined number of such operations. Within each stage, each operation is executed in the same style. The style may be either **Clean** or **Dirty**, depending upon the density of the graph at the beginning of the stage. We further provide for a generalization of this method and present a family of algorithms A_k, $k \geq 1$. The amortized running time of each A_k is $\Theta((v+e)^{\frac{1+\sqrt{1+4(k+1)(2k+3)}}{2(2k+3)}})$ per *Insert/Delete* operation, where e denotes the number of edges of the graph G at the time that the operation is initiated. It follows that this amortized running time may be made arbitrarily close to $\Theta((v+e)^{\frac{\sqrt{2}}{2}})$. Each of the algorithms given here is 2-competitive, thereby matching the competitive ratio of the best existing off–line approximation algorithms for vertex cover.

1 Introduction

Recall that *vertex cover* is a classic problem in combinatorial optimization. The study of vertex cover in computer science has been diverse: it was one of the original *NP*–complete problems [20]; it is often used as a technical tool in performing reductions [13]; various approximation algorithms for vertex cover have been proposed over the past two decades [2, 3, 5, 7, 14, 15, 16, 24]; and, recently, some results in the area of structural complexity theory have deepened our understanding of the difficulties in designing better approximations for vertex cover [4, 22].

Recall also that *fully dynamic* algorithms are aimed at situations where the problem instance is changing (slowly) over time. This situation occurs often in interactive design processes, and fully dynamic algorithms incorporate these incremental changes without any knowledge of the existence and nature of future changes. The objective of course is to develop fully dynamic algorithms that are "competitive" with existing off–line algorithms.

Although the bulk of the existing work on fully dynamic algorithms has been directed toward problems known to be in *P*, e.g. [1, 6, 8, 9, 10, 11, 12, 17, 18, 23, 25], some recent attention has been paid to fully dynamic *approximation* algorithms for problems that are *NP*-complete [19, 21]. In this case, being competitive with off–line algorithms means that the quality of the approximation produced by the fully dynamic approximation algorithm should be as good as that produced by the off–line algorithms. Further, the running time per operation (i.e. change) of the fully dynamic algorithm should be as small as possible.

Thus, in this paper, we consider *fully dynamic approximation algorithms for vertex cover*, where:

- edges are inserted into, and deleted from, the graph in a dynamic fashion, and,
- the vertex cover may be adjusted to accommodate the changes to the instance via insertions and deletions of edges (i.e. vertices may be added to and removed from the vertex cover).

In the subsections that follow we elaborate both on the vertex cover problem itself and on notions associated with fully dynamic approximation algorithms.

[1] Partially supported by the National Science Foundation under Grant CCR–9120731.

[2] Address: Department of Computer and Information Sciences, University of Delaware, Newark, DE, 19711, email: ivkovich@dewey.udel.edu.

[3] Address: Department of Computer and Information Sciences, University of Delaware, Newark, DE, 19711, email: elloyd@cis.udel.edu.

1.1 Vertex cover

In the vertex cover problem, an undirected graph $G = (V, E)$, $v = |V|$ and $e = |E|$, with no multiple edges, and no self–loops is given. The goal is to find a minimum cardinality *vertex cover* V'. That is, a subset of V that covers all edges in E in the following sense: for each edge $uw \in E$, at least one of u and w belongs to V'.

Vertex cover is well-known as one of the earliest *NP*–complete problems [20]. And, as mentioned earlier, vertex cover has a number of polynomial time off–line *approximation algorithms* [2, 3, 5, 7, 14, 15, 16, 24]. For such an approximation algorithm A, it will be convenient to define the **quotient** $Q(A, G) = \frac{A(G)}{OPT(G)}$, where $A(G)$ and $OPT(G)$ denote the cardinality of a vertex cover of a graph G produced by A, and the cardinality of a optimal vertex cover of G. Then, the usual measure of the quality of a solution produced by a vertex cover approximation algorithm A is its **competitive ratio** $R(A)$ defined as $R(A) = \sup\{Q(A, G)\}$. In this case, A is said to be **R(A)–competitive**.

All of the off–line approximation algorithms for vertex cover referenced above are 2–competitive , and no algorithm developed to date has a smaller competitive ratio. Since it is known that vertex cover is approximable in polynomial time only up to some fixed competitive ratio [22], it could be the case that 2 is the best possible competitive ratio (unless, of course $P = NP$). For quotients, better results are possible. In particular, an algorithm from [3] achieves the quotient of $2 - \frac{\log\log v}{2\log v}$. Although this does not improve the competitive ratio, it certainly improves the quotient, and hence the quality of approximations, for graphs on a "small" number of vertices: for graphs with less than 2400, 600000, 10^{12} vertices, the quotient is bounded by 1.75, 1.8, 1.9 respectively [3]. Finally, we note that in regard to running times, the fastest of the above algorithms, e.g. [14, 24], attain the competitive ratio of 2 in time $\Theta(v + e)$.

1.2 Fully dynamic algorithms for vertex cover

The focus of this paper is on fully dynamic approximation algorithms for vertex cover, and the development of fully dynamic algorithms that are competitive with existing off–line methods. We begin this section with a complete description of fully dynamic vertex cover, and follow with a discussion of competitiveness in this context.

1.2.1 Fully dynamic vertex cover

Fully dynamic approximation algorithms for vertex cover process a sequence of *Inserts* and *Deletes* of edges[4]. This is a truly on–line situation, in that the algorithms have no advance knowledge of what future changes (if any) there may be to graph G. As a consequence, in the course of processing *Inserts* and *Deletes*, it may be necessary to change the current vertex cover both in order to maintain a cover, and to insure that the cover is of the appropriate quality. Thus, as the algorithm proceeds, the status of a vertex may change as it is either made part of the cover, or is removed from the cover. Further, such changes may occur arbitrarily often as edges are inserted and deleted.

An important consideration that arises in the fully dynamic context, is just what internal form a "solution" may take. The intent is that the algorithm maintain a form of the solution that is "useful" to the outside world. In this sense, it is natural to require that, in addition to processing *Insert* and *Delete* operations, fully dynamic approximation algorithms for vertex cover should also handle "lookup" queries of the following form:

- *size* – returns in $\mathcal{O}(1)$ time the number of vertices in the current vertex cover,
- *cover* – returns a list of the vertices in the current vertex cover, in time linear in the size of that cover,
- *incover(v)* – returns in $\mathcal{O}(1)$ time the value *true* if v is in the current vertex cover, and *false* otherwise.

These queries may be interspersed in the *Insert/Delete* sequence in any fashion. Of course, since this is an on–line situation, the algorithm has no advance knowledge of where such queries may occur.

[4]Note that the size of the vertex set, $v = |V|$, is fixed.

1.2.2 Competitive fully dynamic approximation algorithms for vertex cover

In this section we discuss the notion of competitiveness in the context of developing fully dynamic approximation algorithms. We begin by noting that with respect to the definitions of *quotient* and *competitive ratio*, there is no need to make a distinction between fully dynamic and off–line algorithms. In each case, these measures reflect the size of the vertex covers produced by the algorithm relative to the size of optimal covers.

In this framework, let A be an (off–line) approximation algorithm for vertex cover, and let B be a fully dynamic approximation algorithm for vertex cover. Then, B is **approximation–competitive** with A if $R(B) \leq R(A)$. Thus, our goal in this paper is to develop fully dynamic approximation algorithms for vertex cover that are 2-competitive.

Note however, that this goal, by itself, is not particularly interesting: such a fully dynamic B can always be produced simply by executing A after every change. This is clearly not what is intended nor required in the incremental context. Rather, the goal is to develop such a B whose running time for any instance G of vertex cover, and any (valid) change of G, call it δ, is $T_B(G, \delta) = o(T_A(\delta(G))$, where $T_B(G, \delta)$ denotes the running time required by B to perform the change δ on G, and $T_A(\delta(G))$ denotes the running time of A on $\delta(G)$, the modified instance.

In the case of vertex cover, since the fastest off–line algorithms require time $\Theta(v + e)$, this translates into developing approximation–competitive fully dynamic algorithms having a running time that is $o(v + e)$ per operation. We say that a fully dynamic approximation algorithm B for vertex cover has **running time** $\mathcal{O}(f(v, e))$ if the time taken by B to process a change to G on v vertices and e edges is $\mathcal{O}(f(v, e))$. If $\mathcal{O}(f(v, e))$ is a *worst case* time bound, then B is **uniform**. If $\mathcal{O}(f(v, e))$ is an *amortized* time bound, then B is **amortized**.

With the preliminaries concluded, the remainder of the paper is organized as follows: In the next section we provide a simple algorithm A_0 for fully dynamic vertex cover. This algorithm is a straightforward extension of a maximal matching based off–line approximation algorithm for vertex cover. Algorithm A_0 is 2–competitive, and its running time is uniform, $\Theta(v)$ per operation. However, as we indicate in section 2, A_0 may not offer any savings over total recomputation for very sparse graphs. We remedy this situation in sections 3 and 4, where we present our main results. We begin in section 3 by giving a fully dynamic 2–competitive algorithm A_1 having an $\Theta((v + e)^{\frac{1+\sqrt{41}}{10}})$ amortized running time per operation. This algorithm features a specific technique that has two essential ingredients. First, we utilize two different methods (**Clean** and **Dirty**) for the processing of *Insert* and *Delete* operations. Second, we partition the processing of operations into *stages*, with a certain number of insertions/deletions handled per stage. Within each stage, all of the operations are handled in an identical fashion (either **Clean** or **Dirty**). The description of A_1 and the techniques used therein lead to the result of section 4. There we generalize A_1 and present a family of fully dynamic approximation algorithms A_k, $k \geq 1$ for vertex cover. For this family of algorithms, we show that, at the expense of some additional effort (for A_k, $k + 1$ "tests"), the asymptotic running time can be made arbitrarily close to $\Theta((v + e)^{\frac{\sqrt{2}}{2}})$ amortized, per *Insert* or *Delete*. Each of these algorithms is 2-competitive, thereby matching the competitive ratio of the best existing off–line algorithms for vertex cover. Finally, in section 5 we furnish some concluding remarks.

2 A Simple Algorithm for Incremental Vertex Cover

Motivated by the notion of approximation-competitiveness introduced in the preceding section, a natural approach to the development of fully dynamic vertex cover algorithms is to try to adapt off–line methods to a fully dynamic context. Thus, we begin by doing just that.

Specifically, we consider a 2–competitive approximation algorithm for vertex cover that runs in $\Theta(v + e)$ time, and is based on **maximal matching** [14]. This algorithm simply computes a maximal matching in the graph G, and then takes as a vertex cover all of the vertices that are incident to the edges in the matching.

A conversion of this off–line algorithm into a fully dynamic one is given below. The conversion is relatively straightforward, and not particularly interesting. However, the ideas involved in the design of this algorithm, as well as an appreciation of the situations where it performs rather slowly, are important factors that guide the development of the algorithms described in sections 3 and 4.

In what follows we describe both the *Insert* and *Delete* operations. In each case, it is assumed that, immediately before an *Insert/Delete* is processed, there is an existing maximal matching M of the current G (i.e. not including the effect of this operation), and that the graph G is represented with adjacency lists.

- *Insert(a)* – to insert an edge a into G, check whether either of its endpoints is already an endpoint in M. If so, then the set of vertices incident to the edges of M is also a vertex cover of $G+a$, and nothing further needs to be done in this regard. If neither of the endpoints of a is an endpoint of an edge in M, then a is added to M. Clearly $M+a$ is a maximal matching of $G+a$. In either case, we conclude by updating the adjacency lists of both endpoints of a (to reflect the inclusion of a in G). All of these operations can be performed, with a proper implementation, in $\mathcal{O}(1)$ time.

- *Delete(a)* – to delete an edge a from G, check whether a is in M. If it is not, then the set of vertices incident to the edges of M is a vertex cover of $G-a$, and nothing further needs to be done in this regard. If a **is** in M, then, by deleting a from G, we may "uncover" some edges. These "uncovered" edges of G are the ones whose endpoints are **not** adjacent to any endpoints of edges in M, other than a. To cover all such edges in $G-a$, the following will be done for each endpoint of a:

  1. Let u be an endpoint of a. Then the adjacency list of u is scanned until finding a vertex w that is **not** an endpoint of an edge in M, or until the end of the list is encountered.
  2. If such a w is found, add edge uw to M. Otherwise, do nothing, since all of the edges incident on u in $G-a$ are covered by endpoints of edges in $M-a$.

  Clearly, the matching that results from this processing is maximal for $G-a$.

  In either case, the adjacency lists of both endpoints of a are updated to reflect the deletion of a.

  Finally, note that the deletion of a non–matching edge can be performed in $\mathcal{O}(1)$ time, and the deletion of a matching edge may involve scanning an entire adjacency list, and hence takes $\Theta(v)$ time.

Hence, we have the following theorem:

Theorem 1 *A_0 is a fully dynamic approximation algorithm for vertex cover, and it is approximation–competitive with the standard off–line algorithm for computing a maximal matching (that is, A_0 is 2–competitive). Further, Insert operations, as well as Delete operations of non–matching edges, take $\mathcal{O}(1)$ uniform running time, and Delete operations of matching edges take $\Theta(v)$ uniform running time.*

Let us remark on some facts about this simple algorithm: its uniform running time is bounded by $\Theta(v)$, incurred only while deleting matching edges. The operations are fast in all the other cases, taking only constant time per operation in the worst case. A natural question that arises here is whether there is a way to somehow avoid the undesirably high running time of some *Delete* operations by a careful manipulation of the dynamically changing information on G, or perhaps perform an amortized analysis of A_0, and attempt to charge some of the cost of a deletion of a matching edge to some other operation, some vertex or edge. Unfortunately, a situation where an edge (whose endpoints have high degree) "toggles" into, and out of, G and M, a number of times, seems to defeat these ideas.

We do note that for graphs G for which $v = o(e)$, even the use of A_0 is a definite improvement over a total recomputation of a maximal matching after each *Insert* and *Delete*. Unfortunately, many important classes of graphs do not belong to this family, e.g. planar graphs.

An obvious question is whether it is possible to improve upon A_0, and design an algorithm that would guarantee, for each operation performed by that algorithms, savings over recomputation for any graph. To illustrate this, consider the application of A_0 to a very dense graph with $e = \Theta(v^2)$ edges. The first few operations A_0 performs on this graph will offer generous savings, since the running time will be bounded by $\mathcal{O}(v) = \mathcal{O}(\sqrt{e})$. However, suppose that this graph is subjected to a sequence of edge deletions, and eventually becomes quite sparse, with $e = \mathcal{O}(v)$ edges. The operations performed by A_0 on such a graph offer (asymptotically) no savings whatever over recomputation.

It would preferable to have it be the case that, no matter what happens with the graph, whether its number of edges increases towards dense graphs or decreases towards sparse graphs, each and every operation is going to be much faster than recomputation.

In the next section we present an algorithm that accomplishes just that, albeit with an *amortized* running time.

3 An Improved, Amortized Algorithm A_1

We begin this section by describing an algorithm B_1 for fully dynamic vertex cover whose amortized running time is $\Theta((v+e)^{\frac{3}{4}})$ per *Insert/Delete* operation. In our discussion of algorithm B_1, we introduce the technique of **Clean and Dirty stages**. The ideas involved in designing B_1 are then used to obtain another algorithm A_1, with a slightly sharper amortized running time bound of $\Theta((v+e)^{\frac{1+\sqrt{41}}{10}})$ (roughly $\Theta((v+e)^{0.74})$).

3.1 What is Clean and what is Dirty?

Here we give an intuitive introduction to our technique. Based on the concepts developed here, we sketch B_1. In later sections we build on the ideas from this section, developing more and more efficient algorithms. All of the forthcoming algorithms are closely related to B_1, and the description of B_1 given here will provide for an easy appreciation of A_1, as well as the algorithms A_k, $k > 1$ described in section 4.

The algorithm B_1 is also based on the computation of a maximal matching to provide a vertex cover. For the purposes of fully dynamic approximation of vertex cover, this method is appealing because whether or not an edge is in the matching depends only on the "local properties" of that edge, i.e. on the *status* of the endpoints of the edge under consideration. Here, the *status* of a vertex is either *matched*, i.e. being an endpoint of some matching edge, or *non–matched*, i.e. not being an endpoint of some matching edge in the current maximal matching.

We proceed by reiterating the difficulties encountered in attempting to speed up A_0. The weak spot of this algorithm is that a *Delete* of a matching edge may involve scanning as many as $\Theta(v)$ elements of the adjacency lists of the two endpoints of that edge, searching for an unmatched vertex. This being the case, it would seem reasonable to do the following: For each vertex u, separate the adjacency list of u into two sublists, with one sublist containing the matched neighbors of u, and the other sublist containing the non-matched neighbors of u. Such a partition would make it trivial to locate an unmatched neighbor of u. Unfortunately, the maintenance of such a partition is quite costly. In particular, the sequence of *Inserts* and *Deletes* may be such that a vertex of high degree "toggles" between being and not being an endpoint of a matching edge. This forces each neighbor w of that vertex to continually register these changes by moving the vertex between the two parts of the adjacency list of w. If such vertices have as many as $\Theta(v)$ neighbors, there are $\Theta(v)$ changes to be performed per operation.

3.1.1 The Dirty strategy

Despite the difficulties just discussed, the idea of separating adjacency lists into two sublists in order to facilitate the location of unmatched neighbors is an appealing one, and is one that we utilize, in a modified form, in all of the remaining algorithms of this paper. To attempt to deal with the "toggling" problem discussed above, we relax the stringent condition that *all* of the changes of status be recorded immediately. The idea is to proceed with the computation, with the awareness that some of the information about the status of the neighbors of a vertex may be false. Thus, the algorithm tolerates some "inaccuracy" in the sublists. To make this work, there needs to be an efficient mechanism for testing whether the information read from the adjacency list (two sublists) is actually correct. This can be easily done in $\mathcal{O}(1)$ uniform running time: in a record associated with each vertex, one bit of information will suffice to record whether or not that vertex is an endpoint in the current maximal matching; and, each entry of an adjacency list (two sublists) may have an additional pointer that points to the record associated with the respective vertex.

It is also critical that there be a bound on the number of operations that the algorithm performs while tolerating inaccuracy. After this number of operations, call it s, the algorithm "cleans up" all of the inaccurate

information, and then again proceeds in the described fashion until the next "clean–up", and so on. The "clean–up" may take longer than $\Theta(v+e)$, but if it is performed rather rarely, the cost may be amortized by charging to the operations performed between the two successive "clean–ups". *Insert* and *Delete* operations between two successive "clean–ups" may require $\Theta(s)$ time in order to deal with $\mathcal{O}(s)$ inaccurate information that may be present.

This is the essence of the **Dirty** strategy. As will be seen later, if G is sparse, the **Dirty** strategy provides significant savings; whereas, if G is dense, then the **Dirty** strategy will not be beneficial.

3.1.2 The Clean strategy

The **Clean** strategy performs *Insert* and *Delete* operations in precisely the manner described in the discussion of A_0, along with the additional requirement that, for each vertex, both adjacency sublists are maintained correctly from operation to operation. As noted before, A_0 provides generous savings if G *is* dense.

3.2 Incorporating both strategies

Recall that our goal is to offer savings over recomputation, regardless of whether G is dense or sparse. And, as noted earlier, the density of G may change in the course of performing many operations. Thus performing only one of the two strategies will not do, since a dense graph may well become sparse and vice versa. Therefore, in the algorithms that follow, we incorporate a certain level of supervision, such that, from time to time, the algorithm will pause and determine whether the next several operations will be performed in the **Clean** or **Dirty** way. This algorithm operates in stages, processing a stream of on–line requests for *Inserts/Deletes* and queries (*size, cover, incover(u)*).

We now proceed with a description of B_1. Here, we define a **stage** in the execution of B_1 as:

- the execution of two tests that will determine whether all of the *Insert* and *Delete* operations in that stage will be done in the **Clean** or **Dirty** manner;
- the execution of v^a, $a = \frac{3}{4}$ *Insert/Delete* operations, in an on–line fashion;
- in case the chosen strategy was **Dirty**, the "clean–up" of the inaccurate information in the adjacency sublists.

Next, we give the details of the tests performed at the beginning of each stage, and give a thorough description of the **Clean** and **Dirty** modes of operation.

- Tests – two tests are performed. If either of the two tests succeeds, then the **Clean** strategy will be used. Conversely, if both tests fail, then the **Dirty** strategy will be used.

  1. The first test asks whether there are at least v^{b_2}, $b_2 = \frac{2}{3}$ vertices of degree at least v^{c_2}, $c_2 = \frac{2}{3}$. If so, the **Clean** strategy should be used throughout the stage (i.e. for the processing of v^a, $a = \frac{3}{4}$ *Insert/Delete* operations). While performing this test, label the vertices meeting the degree requirement as *high*, and the other vertices as *low*. These labels will, in case both tests fail and the **Dirty** strategy is adopted, provide the information needed to properly execute the slightly inaccurate operations and the "clean–up".
  2. The second test asks whether there are at least v^{b_1}, $b_1 = \frac{1}{2}$ vertices of degree at least v^{c_1}, $c_1 = \frac{5}{6}$. If so, then the **Clean** strategy should be used throughout the stage.

  These tests may be implemented in time $\Theta(v)$. This does not increase the running time of the algorithm in either the **Clean** or **Dirty** mode of operation.

- **Clean** – as noted before, we perform the operations in a manner almost identical to that of A_0, with one exception: since each vertex now has two adjacency sublists, recording the neighbors currently in the matching and not in the matching respectively, this information needs to be maintained rigorously, and upon the completion of each operation it should be correct. Note that the running time is $\Theta(v)$ per

Insert/Delete operation, but in the case of **Clean**, this is permissible: $\Theta(v) = \Theta(e^{\frac{3}{4}})$. In addition, the relatively small number of operations in the stage cannot significantly change the fact that G contains a relatively high number of edges: in case the first test succeeded, $v^a \ll v^{b_2+c_2}$, i.e. $\Theta(v^{b_2+c_2} \pm v^a) = \Theta(v^{b_2+c_2})$ (similar reasoning for the second test). The remaining issue of accounting for the running time required to perform the test(s) is easily resolved by assigning the entire cost ($\Theta(v)$) to the first operation in the stage.

- **Dirty** – the very fact that **Dirty** is being executed implies that both tests failed. This provides some useful information. In particular, consider the following classification of vertices of G at the beginning of such a stage:

  - Vertices of degree at least v^{c_1}. We call the set of such vertices XL (extra large), and note that there are fewer than v^{b_1} such vertices at the beginning of the stage. While performing the amortized analysis we will conservatively assume that at the end of the stage, all XL vertices are of degree v.
  - Vertices with degree at least v^{c_2}, but less than v^{c_1}. We call the set of such vertices L (large), and note that there are less than v^{b_2} such vertices[5] at the beginning of the stage. While performing the amortized analysis we will conservatively assume that at the end of this stage, all L vertices are of degree $v^{c_1} + v^a = \Theta(v^{c_1})$.

    Note that the XL and L vertices are precisely the vertices that are labeled as *high* while performing the first test.

  - All other vertices. We call the set of such vertices S (small), and note that these are vertices with degree less than v^{c_2} at the beginning of the stage, thus labeled as *low*. Note that the degree of such vertices may grow within the stage to at most $\mathcal{O}(v^a)$.

  With these preliminaries concluded, we proceed with a description of a **Dirty** stage. As noted, in a **Dirty** stage, v^a *Inserts/Deletes* are performed. Both *Inserts* and *Deletes* depend upon the information about the status of the endpoints of an edge. That status (recall: matched/non–matched) is always maintained correctly in the record associated with each of the endpoints of the edge. The aforementioned "inaccuracy" pertains to the fact that some elements might belong to the wrong adjacency sublist.

  To simplify our description of a **Dirty** stage, we divide *Inserts* and *Deletes* into two categories:

  Operations that do not change the status of any vertices – this category includes both the *Inserts* of edges having at least one endpoint in the current maximal matching, and *Deletes* of non–matching edges. Since our goal is simply to maintain a maximal matching, in these cases no vertices should change their status. It follows that we need only add (for *Inserts*) or remove (for *Deletes*) one endpoint, call it u, to/from the other endpoint's, call it w, adjacency sublists and vice versa. In the case of insertion, u should be added into the appropriate sublist according to the status of u. This needs to be done in a specific way: if u is a *high* vertex, the algorithm insists that u should be added at the front of the appropriate sublist (matched/non–matched); if u is a *low* vertex, and it needs to be added to the non–matched sublist of w, then u is added at the front of the sublist; whereas if it needs to be added to the matched sublist of w, then the algorithm insists that u should be added at the end of the sublist.

  In addition, we maintain an *update list* (doubly linked), for each vertex x, that identifies of all the occurrences of x in the adjacency lists throughout G. So, before completing the *Inserts* and *Deletes* in this category, the algorithm will add the newly created adjacency list entries to (remove the deleted adjacency list entries from) u's and v's respective update lists.

  All of these actions may be performed in constant time.

[5] Note that this a very loose bound, since we did not exclude the XL vertices. It will turn out that this generosity does not affect the analysis, and we chose to keep matters simple.

Operations that change the status of one or more vertices (at most two vertices per operation) – this category includes both the *Inserts* of edges neither of whose endpoints are in the current maximal matching, and *Deletes* of matching edges.

First, we consider the *Insert* of an edge h neither of whose endpoints (u and w) is in the current maximal matching. Hence, h needs to be added to the matching. To reflect the inclusion of h in G, u and w are added into each other's adjacency sublists, and update lists. We do not however *necessarily* update the entries for u and w in their neighbor's sublists (to reflect that h is now in the matching). This is the essence of the **Dirty** technique: first for u and then for w, the algorithm checks whether u (w) is labeled *low*. If so, the entire update list associated with u (w) is scanned, and the status information at all of the neighbors' adjacency sublists is updated. If, on the other hand, u (w) is labeled *high*, the sublists of the neighbors of u (w) will not be updated. This inaccuracy in some adjacency sublists is the essential ingredient that leads to a reduced running time per operation. Finally, it follows from the vertex degrees that an *Insert* requires $\Theta(v^a)$ time.

Next, we consider a *Delete* of a matching edge f. Similarly to above, the adjacency sublists, and the update lists of u and w (the endpoints of f) are updated to reflect the deletion of f from G. In addition, in order to maintain the matching, up to two edges may need to be added to the matching, and, of course, f needs to be removed. To do this we first check if any of the remaining edges incident to u and w can be added to the matching. Checking for such edges is a somewhat involved, since adjacency sublists may contain some inaccurate information. Thus, let x be first u and then w. The algorithm scans the adjacency sublists of x. Since the goal is to locate a non–matched vertex adjacent to x, the sublist of x containing non–matched vertices is scanned first. This scan proceeds until a vertex y is found that both claims to be, and truthfully is, a non–matched vertex. Note that when scanning x's non–matched sublist, the only "liars" are *high* vertices, because *low* vertices are updated after each operation. It follows that one of two cases will occur: either a truthfully non–matched neighbor of x will be found after scanning $\mathcal{O}(v^a)$ elements of the non–matched sublist of x, or the scanning of the entire non–matched sublist of x will not yield a non–matched neighbor. Note that, in the latter case, the sublist is rather short ($\mathcal{O}(v^a)$). However, this does **not** mean that there is not a non–matched neighbor of x. There could be such a neighbor who is sitting in the matched sublist of x, but is a "liar". Here, our discipline for inserting *high* and *low* vertices into adjacency sublists comes to the rescue: *high* vertices are at the front of the matched sublist. Thus, the algorithm proceeds by scanning the matched sublist until either a "liar" is found, or a *low* vertex is encountered, or the entire sublist is scanned. It follows that no more than $\mathcal{O}(v^a)$ elements in both sublists need to be scanned until either a non–matched vertex has been found, or until it becomes certain that there is no non–matched neighbor of x.

If a non–matched neighbor y of x is located, then the edge xy needs to be added to the matching by updating the status of x and y. This is done precisely as described above for dealing with the insertion of an edge, neither of whose endpoints is in the matching. This means that if x (y) is a *low* vertex, then a full update of the appropriate sublists is performed, and if x (y) is a *high* vertex, then nothing further is done. Therefore, the total running time for operations of this kind is bounded by $\Theta(v^a)$.

We conclude our description of the processing in a **Dirty** stage, by considering the "clean–up" to be done at the end of the stage. Recall, that in the process of executing the operations, accurate information was always maintained for *low* vertices (i.e. some pains were taken to always have a *low* vertex on the appropriate sublists). Therefore, the only "clean–up" that needs to be done in this regard is associated with *high* vertices. This "clean–up" is simple: for each *high* vertex u, it suffices to scan the update list of u, and using the entries on that list, to place u in the appropriate sublist of each of its neighbors (placed at either the front or end of the sublist according to the status and the degree of u). The time required is proportional to the degree of u.

The overall time for this "clean–up" is thus: $|\mathrm{XL}|\mathcal{O}(v) + |\mathrm{L}|\mathcal{O}(v^{c_1}) \leq v^{b_1}\mathcal{O}(v) + v^{b_2}\mathcal{O}(v^{c_1}) = \mathcal{O}(v^{2a})$.

There is yet another "clean–up" that needs to be performed. Namely, in the process of executing insertions and deletions within a stage, a certain number of vertices ($\mathcal{O}(v^a)$) may, although initially labeled as *low*, actually become of high degree, and should be labeled as *high*. While this is going to

happen at the beginning of the next stage, the resulting data structure may become degenerate: recall that the algorithm insists upon a very particular arrangement of *high* and *low* vertices in matching sublists. Thus, this needs to be rectified by "cleaning–up", i.e. placing the transient (*low* $\rightarrow$ *high*) vertices at the appropriate end of all the matched sublists where they occur. There can be $\mathcal{O}(v^a)$ such vertices. The situation is analogous for vertices that started initially as *high*, and eventually had their degree reduced to *low*. The overall time required for this "clean–up" in a **Dirty** stage[6] is again $\mathcal{O}(v^{2a})$.

Note that we can charge the time required to perform the tests by simply adding it to the time for "clean–up", since $2a > 1$, i.e. $\mathcal{O}(v^{2a} + v) = \mathcal{O}(v^{2a})$.

The cost of the "clean–up", i.e. the incurred running time, can be charged to the preceding v^a *Insert/Delete* operations, allocating a charge of $\mathcal{O}(v^a)$ to each operation, so the amortized running time of each operation in a **Dirty** stage of B_1 is $\mathcal{O}(v^a)$.

Summarizing, each operation in a **Clean** stage requires $\Theta(v)$ time, which is, due to the large number of edges, $\Omega(e^a)$, $a = \frac{3}{4}$. Also, each operation in a **Dirty** stage requires amortized time $\Theta(v^a)$, $a = \frac{3}{4}$. Hence, operations performed in both **Clean** and **Dirty** fashion achieve significant savings over recomputation. The same will be true of all the subsequent algorithms. Thus we have the following theorem:

Theorem 2 *B_1 is a 2–competitive fully dynamic approximation algorithm for vertex cover (thus, it is approximation–competitive with the standard off–line algorithm). Further, both Insert and Delete operations require $\Theta((v+e)^{\frac{3}{4}})$ amortized running time.*

3.3 A slight improvement – Algorithm A_1

Our description of B_1 involved the constants a, b_1, c_1, b_2, and c_2. The values of these constants were carefully chosen to produce a time bound of $\Theta(v^a)$. In this section we describe algorithm A_1, a variation of B_1, that achieves a slightly improved running time through a different selection of the values of those constants. Further, we can show that this particular choice of the values of the constants is the best possible. That is, there are no other values of the constants that produce a superior running time. In some sense this says that A_1 is "the best in its class"[7].

Due to space constraints, we furnish without proof:

Theorem 3 *A_1 is a 2–competitive fully dynamic approximation algorithm for vertex cover (thus, it is approximation–competitive with the standard off–line algorithm). Further, Insert operations, as well as Delete operations require $\Theta((v+e)^{\frac{1+\sqrt{41}}{10}})$ amortized running time. The choice of constants in A_1, i.e. $a = \frac{1+\sqrt{41}}{10}$, $b_1 = 2a - 1$, $c_1 = \frac{a+1}{2}$, $b_2 = \frac{3a-1}{2}$, and $c_2 = a$, is optimal in the sense that no other choice of a, ... , c_2 can improve the running time.*

4 Further Improvements – Algorithms A_k

In this section we develop a family of fully dynamic algorithms A_k, $k \geq 1$, each of which improves upon the running time of the basic algorithm A_1.

The difference between A_1 and any A_k, $k > 1$, is only in the number of tests performed prior to the execution of a stage. In general, A_k will perform $k+1$ tests of the form, $k+1 \geq i \geq 1$:

Are there at least v^{b_i} vertices of degree at least v^{c_i}?

As before, an affirmative answer to any of the $k+1$ questions implies the execution of a **Clean** stage, whereas a negative answer to all of the questions implies the execution of a **Dirty** stage. As in A_1, during

[6]Note that this "clean–up" is not necessary in a **Clean** stage, since there is sufficient time to perform the updates of this nature within each *Insert/Delete* operation.

[7]Note that this does not at all suggest that A_1 is the best among all algorithms. In the next section, we present a family of algorithms A_k, $k \geq 1$, all of which, except of course A_1 itself, are superior to A_1.

the computation of the first test, each vertex is labeled *high/low*. Aside from the use of the additional tests, each of these algorithms proceeds in a manner identical to that of A_1.

Our results are summarized in the following theorem:

Theorem 4 *For each integer $k > 0$, A_k is a 2–competitive fully dynamic approximation algorithm for vertex cover (thus, it is approximation–competitive with the standard off–line algorithm). Further, Insert operations, as well as Delete operations performed by A_k require $\Theta((v+e)^{\frac{1+\sqrt{1+4(k+1)(2k+3)}}{2(2k+3)}})$ amortized running time. The choice of constants in A_k, is optimal in the sense that no other choice of a, ... , c_{k+1} can improve the running time. Here, the constants are $a = \frac{1+\sqrt{1+4(k+1)(2k+3)}}{2(2k+3)}$, b_1, ... , c_{k+1}, where b_i's and c_i's satisfy the following equations:*

$$b_1 + c_1 = \ldots = b_{k+1} + c_{k+1} = \frac{1}{a}, \qquad b_{k+1} + c_k = b_k + c_{k-1} = \ldots = b_1 + 1 = 2a, \qquad \textit{and} \qquad c_{k+1} = a.$$

Before giving a brief sketch of the proof, we provide an example: For A_2, the value of a is $a = \frac{1+\sqrt{85}}{14} \doteq 0.73$, so the amortized running time of A_2 is bounded by roughly $\Theta((v+e)^{0.73})$. The three tests are performed with the following values: $b_1 = 2a - 1 \doteq 0.46$, $c_1 = 1 + 1/a - 2a \doteq 0.91$; $b_2 = 4a - 1 - 1/a \doteq 0.55$, $c_2 = 1 + 2/a - 4a \doteq 0.82$; and $b_3 = 6a - 1 - 2/a \doteq 0.64$, $c_3 = a \doteq 0.73$.

Sketch of proof: We first note that c_{k+1} may not exceed a, for otherwise c_{k+1} would dominate the running time. To see this, we set $c_{k+1} = a - \epsilon$. We now solve the outlined system of equations. By a standard inductive argument, we obtain that the value of a is:

$$a = \frac{(1+\epsilon) + \sqrt{(1+\epsilon)^2 + 4(k+1)(2k+3)}}{2(2k+3)}$$

Clearly, the value of a is minimized for $\epsilon = 0$. This gives the stated value of a and finishes the proof.

□

Corollary 1 *For any α, $\frac{\sqrt{2}}{2} < \alpha$ there is an amortized fully dynamic approximation algorithm C_α for vertex cover that is 2–competitive. Further, Insert and Delete operations performed by C_α require $\mathcal{O}((v+e)^\alpha)$ amortized running time.*

5 Conclusion

We have studied the problem of maintaining an approximate solution for **vertex cover** when edges may arrive and depart dynamically. The main result of our paper is the family of fully dynamic 2–competitive approximation algorithms for vertex cover A_k, $k \geq 1$.

Algorithms A_k feature appealing theoretical running time bounds, and achieve significant savings over recomputation: for example, A_5 ($a = 0.7212$) needs to look into at most (roughly) 146 entries in the adjacency lists of a graph on 1000 vertices, 767 entries for graphs on 10000 vertices, 4037 entries for graphs on 10^5 vertices, and 12885 entries for graphs on $5 \cdot 10^5$ vertices. Furthermore, the algorithms do not employ complicated data structures, and should perform very well in practice.

The unresolved issues are whether there exist efficient uniform fully dynamic algorithms for maximal matching, and whether the running time bound can be further improved for amortized fully dynamic approximation algorithms for maximal matching. One possibility might be to consider a variety of **Dirty** approaches that would represent a scale of strategies, each one employed in the case of a certain density of G, for a certain number of operations, not necessarily identical for all the **Dirty** strategies.

Finally, the major unresolved issue is whether there are efficient fully dynamic (uniform and/or amortized) algorithms for vertex cover that are not based on maximal matching, and whose running time improves upon those of the family of algorithms A_k.

References

[1] G. Ausiello, G. F. Italiano, A. M. Spaccamela, and U. Nanni. (1991). Incremental Algorithms for Minimal Length Paths. *Journal of Algorithms* **12**, pp. 615–638.

[2] R. Bar–Yehuda and S. Even. (1981). A Linear–Time Approximation Algorithm for the Weighted Vertex Cover Problem. *Journal of Algorithms* **2**, pp. 198–203.

[3] R. Bar–Yehuda and S. Even. (1985). A Local–Ratio Theorem for Approximating the Weighted Vertex Cover Problem. *Annals of Discrete Mathematics* **25**, pp. 27–46.

[4] R. Bar–Yehuda and S. Moran. (1984). On Approximation Problems Related to the Independent Set and Vertex Cover Problems. *Discrete Applied Mathematics* **9**, pp. 1–10.

[5] V. Chavatal. (1979). A Greedy Heuristic for the Set–Covering Problem. *Mathematics of Operations Research* **4**(3), pp. 233-235.

[6] S. W. Cheng and R. Janardan. (1991). Efficient Maintenance of the Union of Intervals on a Line, with Applications. *Journal of Algorithms* **12**, pp. 57–74.

[7] K. L. Clarkson. (1983). A Modification of the Greedy Algorithm for Vertex Cover. *Information Processing Letters* **16**(1), pp. 23–25.

[8] D. Eppstein, Z. Galil, G. F. Italiano, and A. Nissenzweig. (1992). Sparsification – A Technique for Speeding up Dynamic Graph Algorithms. *Proceedings of the 33rd IEEE Symposium on Foundations of Computer Science*, pp. 60–69.

[9] G. Frederickson. (1985). Data Structures for On–Line Updating of Minimum Spanning Trees, with Applications. *SIAM Journal on Computing* **14**(4), pp. 781–798.

[10] Z. Galil and G. F. Italiano. (1991). Fully Dynamic Algorithms for edge Connectivity Problems. *Proceedings of the 23rd ACM Symposium on Theory of Computing*, pp. 317–327.

[11] Z. Galil and G. F. Italiano. (1991). Maintaining Biconnected components of Dynamic Planar Graphs. *Proceedings of the 18th International Colloquium on Automata, Languages, and Programming*, pp. 339–350.

[12] Z. Galil, G. F. Italiano, and N. Sarnak. (1992). Fully Dynamic Planarity Testing. *Proceedings of the 24th ACM Symposium on Theory of Computing*, pp. 495–506.

[13] M. R. Garey and D. S. Johnson. (1979). *Computers and Intractability: A Guide to the Theory of NP–Completeness.* Freeman, San Francisco.

[14] F. Gavril. (1974). See [13, pp. 134]

[15] D. S. Hochbaum. (1982). Approximation Algorithms for the Set Covering and Vertex Cover Problems. *SIAM Journal on Computing* **11**(3), pp. 555–556.

[16] D. S. Hochbaum. (1983). Efficient Bounds for the Stable Set, Vertex Cover and Set Packing Problems. *Discrete and Applied Mathematics* **6**, pp. 243–254.

[17] T. Ibaraki and N. Katoh. (1983). On–Line Computation of Transitive Closures of Graphs. *Information Processing Letters* **16**, pp. 95–97.

[18] G. F. Italiano. (1988). Finding Paths and Deleting Edges in Directed Acyclic Graphs. *Information Processing Letters* **28**, pp. 5–11.

[19] Z. Ivković and E. L. Lloyd. (1992). Fully Dynamic Algorithms for Bin Packing. *CIS Technical Report No. 92–26.* University of Delaware.

[20] R. M. Karp. (1972). Reducibility among Combinatorial Problems. In *Complexity of Computations* (R. E. Miller and J. W. Thatcher, Eds.), pp. 85–103. Plenum, New York.

[21] P. N. Klein and S. Sairam. (1993). Fully Dynamic Approximation Schemes for Shortest Path Problems in Planar Graphs. Manuscript.

[22] C. H. Papadimitriou and M. Yannakakis. (1991). Optimization, Approximation, and Complexity Classes. *Journal of Computer and System Sciences* **43**, pp. 425–440.

[23] M. Rauch. (1992). Fully Dynamic Biconnectivity in Graphs. *Proceedings of the 33rd IEEE Symposium on Foundations of Computer Science*, pp. 50–59.

[24] C. Savage. (1982). Depth–First Search and the Vertex Cover Problem. *Information Processing Letters* **14**(5), pp. 233–235.

[25] M. Smid. (1991). Maintaining the Minimal Distance of a Point Set in Polylogarithmic Time. *Proceedings of the 2nd Annual ACM–SIAM Symposium on Discrete Algorithms*, pp. 1–6.