Active Data Structures and Applications to Dynamic and Kinetic Algorithms

Kanat Tangwongsan (ktangwon@cs.cmu.edu)

Advisor: Guy Blelloch (guyb@cs.cmu.edu)

May 5, 2006

Abstract

We propose and study a novel data-structuring paradigm, called *active data structures*. Like a time machine, active data structures allow changes to occur not only in the present but at any point in time—including the past. Unlike most time machines, where changes to the past are incorporated and propagated automatically by magic, active data structures systematically communicate with the affected parties, prompting them to take appropriate actions. We demonstrate an efficient maintenance of three active data structures: (monotone) priority queue, dictionary, and compare-and-swap.

These data structures, when paired with the self-adjusting computation framework, create new possibilities in kinetic and dynamic algorithms engineering. Based on this interaction, we present three practical algorithms: a new algorithm for 3-d kinetic/dynamic convex hull, an algorithm for dynamic list-sorting, and an algorithm for dynamic single-source shortest-path (based on Dijkstra). Our 3-d kinetic convex hull is the first efficient kinetic 3-d convex hull algorithm that supports dynamic changes simultaneously.

This thesis provides an implementation for selected active data structures and applications, whose performance is analyzed both theoretically and experimentally.

1 Introduction

Motion is truly ubiquitous in the physical world. Perhaps, equally ubiquitous are contexts in which critical need arises for computation to support motion effectively. The class of computation capable of handling motion is often called *kinetic algorithms (data structures)*. In the simplest form, kinetic algorithms maintain a combinatorial structure as its defining objects undergo a prescribed motion. Over the past decades the algorithmic study on the subject has made significant theoretical advances for many problems, yet leaving many important basic problems open [Gui04]. However, only a small number of these algorithms are successfully implemented and used, mostly because the algorithms are highly intricate.

This thesis was initially motivated by an open problem in computational geometry—kinetic 3-d convex hull—that we later made some progress on. By now, it is hard to overstate the importance of maintaining the convex hull of a set of points. However, while much is known for $d = 2$ [BGH97, Gui04], the problem of maintaining the kinetic convex hull for dimension $d \geq 3$ has remained open. For $d = 3$, the best known algorithm re-computes the convex hull entirely as need arises. Our approach to solving the kinetic 3-d convex hull problem has centered on devising a general-purpose technique for handling motion and other types of changes. We attempted to extend the self-adjusting computation framework [Aca05, ABB+05, ABBT06] to support motion (*i.e.* continuous changes). This study poses two fundamental questions which we answer affirmatively in this thesis:

1. Is there a natural class of data structures that are closely related to the behaviors of incremental computation (or more generally, dynamic/kinetic algorithms) of which the design and analysis of dynamic/kinetic algorithms can take advantage?
2. If these data structures exist, can they be integrated to self-adjusting computation to alleviate the task of dynamic/kinetic algorithms engineering in a natural way?

We propose and study a novel data-structuring paradigm, called *active data structures*. Like a time machine, active data structures allow changes to occur not only in the present but at any point in time—including the past. Unlike most time machines, where changes to the past are incorporated and propagated automatically by magic, active data structures systematically communicate with the affected, prompting them to take appropriate actions. We demonstrate an efficient maintenance of three active data structures: (monotone) priority queue, dictionary, and compare-and-swap.

Given a time machine and an ordinary algorithm, it is not hard to simulate a dynamic/kinetic algorithm. An ordinary algorithm refers to long-familiar algorithms that cannot account for changes that take place on the fly. As an example, consider constructing a dynamic shortest-path algorithm. First, start the time machine. Then, run Dijkstra to the given graph G. This computes the shortest-path for the graph G. To update the graph G, tell the time machine to revert to the beginning of time, update the graph, and tell the time machine to return to the present. The resulting shortest-path now corresponds to the new graph, as desired. Even though no time machine are publicly accessible, self-adjusting computation somewhat mimics that behavior.

Active data structures can naturally interact with self-adjusting computation. This thesis provides an implementation for selected active data structures and applications. We show that active data structures can help simplify and improve many dynamic algorithms. Section 4 illustrates a simple use of active data structures to the dynamic list-sorting problem.

In Section 5, we describe a new practical algorithm for maintaining the kinetic 3-d convex hull based on active data structures. This algorithm greatly improves on the naive method. We verify the effective of our approach both theoretically and empirically. We describe an algorithm for dynamic shortest-path in Section 6. We conclude this paper by discussing about work in progress and pointing out some directions for future work.

2 History and Background

The ability to efficiently maintain the output of a computation as the input undergoes changes has been proven crucial in a countless number of real-world applications. Dynamic changes are discrete changes involving insertions and deletions of objects in the input. Algorithms that handle dynamic changes are called *dynamic algorithms*. Over the past decades, the algorithms community has extensively studied this class of algorithms and made significant theoretical advances. Despite tremendous efforts, many of these algorithms are never successfully implemented, as they are complicated, making the task of implementing and debugging them highly strenuous.

Over the same period of time, the programming languages community has put efforts into developing tools to cope with and combat the implementation challenges of dynamic algorithms. A major line of research has focused on devising techniques for transforming static programs to their dynamic counterparts. Many of these techniques build on the idea of *incremental computation*. Early work on incremental computation [DRT81, PT89, ABH02, Car02] has shown that the technique can deliver competitive performance to handcraft algorithms and are applicable to a broad range of problems.

2.1 Self-Adjusting Computation

Self-adjusting computation is an attempt to bring together techniques in the algorithms community and the programming languages community. The approach provides a technique for semi-automatically transforming static programs to programs that can adjust to changes to their inputs (and internal states). The transformation in the self-adjusting computation framework does not generate a dynamic-algorithm description for the given static algorithm. Instead, think of self-adjusting computation as a special machine that learns about actions that a program (an algorithm) performs and is capable of intelligently re-executing parts of the program as changes occur. We term the process of smart re-execution "change-propagation". A smart re-execution, as opposed a blind re-execution, selectively re-executes parts of computations as needed and reuses results of computations whenever permissible. Theoretical evidence suggests that smart re-execution is highly effective for many classes of problems.

Self-adjusting computation relies on two key ideas: *dynamic dependence graph (or DDG)* and memoization [Aca05, ABH02]. During the execution of a self-adjusting program, the run-time system builds a DDG, which records the relationship between computation and data. After a self-adjusting program completes its execution, the user can change any computation data (*e.g.*, the inputs) and update the output by performing a change propagation. This change-and-propagate step can be repeated. The change-propagation algorithm updates the computation by mimicking a from-scratch execution.

We state some definitions and certain properties of self-adjusting programs. A detailed treatment of the subject can be found in [Aca05, ABB$^+$05, ABBT06, ABH$^+$04].

Definition 1 *Let $\mathcal{I}$ the set of possible start states (inputs). A* class of input changes *is a relation $\Delta \subseteq \mathcal{I} \times \mathcal{I}$. The modification from I to I' is said to conform the class of* input change *Δ if and only if $(I, I') \in \Delta$. For output-sensitive algorithms, Δ can be parameterized according to the output change.*

Definition 2 *A* trace model *consists of a set of possible* traces *$\mathcal{T}$. For a set of algorithms $\mathcal{A}$, a* trace generator *is the function $T : \mathcal{A} \times \mathcal{I} \to \mathcal{T}$, and a* trace distance *is the function $\delta_{tr}(\cdot, \cdot) : \mathcal{T} \times \mathcal{T} \to \mathbb{Z}^+$.*

Let $\mathbf{E}_\phi[X]$ denote the expectation of the random variable X with respect to the probability-density function ϕ, we define expected-case stability as follows.

Definition 3 (Expected-Case Stability) *For a trace model, let P be a randomized program, let Δ_n a class of input changes parametrized n, possibly the size of the input, and let $\phi(\cdot)$ be a* *probability-density function on bit strings* $\{0,1\}^*$. *For all* $n \in \mathbb{N}$, *define*

$$d(n) = \max_{(I,I')\in\Delta_n} \mathbf{E}_\phi[\delta_{tr}\left(tr(P,I), tr(P,I')\right)].$$

We say that P *is expected* S*-stable for* Δ *and* ϕ *if* $d(\cdot) \in S$.

Note that expected $O(f(\cdot))$ stable, $\Omega(f(\cdot))$ stable, and $\Theta(f(\cdot))$ are all valid uses of the stability notation. Worst-case stability is defined by dropping the expectation from the definition.

We define monotonicity of a program as in Acar *et al.* [ABH$^+$04] and state the following theorems, which are useful in the sequel.

Theorem 4 (Stability Theorem [ABH$^+$04]) *If an algorithm is* $O(f(n))$*-stable for a class of input changes* Δ, *then each operation will take at most* $O(f(n)\log n)$.

However, if the discrepancy set has a bounded size, then each operation can be performed in $O(f(n))$.

Theorem 5 (Triangle Inequality for Change Propagation) *Let* P *be a monotone program with respect to the class of changes* Δ_1 *and* Δ_2. *Suppose that* P *is* $O(f(n))$ *and* $O(g(n))$ *stable for* Δ_1 *and* Δ_2, *respectively, for some measure* n. P *is also monotone with respect to the class of changes* $(\Delta_1 \circ \Delta_2)$ *obtained by composing* Δ_1 *and* Δ_2, *then* P *is* $O(f(n)+g(n))$ *stable for* Δ_1 *o* Δ_2.

A proof of this theorem is supplied in the appendix.

3 Active Data Structures

We introduce a new data structuring paradigm, called *active data structures.* Akin to the retroactive data structures [DIL04], active data structures allow the data-structure operations to take place not only in the present but also in the past. When an operation is performed, an active data structure, in addition to fixing the internal states, identifies which other operations take on new resulting values and notifies them to adjust to the changes. This capability has an important software-engineering benefit: *composibility.*

Consider a database for bank accounts at a financial company. Suppose that, at 12pm, we discover that an important transaction performed earlier at 9am was erroneous, and we need to correct it. Most traditional systems would need to rollback the transactions to 9am, where we then correct the record and recommit subsequent transactions. Using a retroactive data structure [DIL04], one would, in one-step, magically tell the data structure to correct the operation performed at 9am and rejoice. However, it is often inadequate to make corrections only internally to the data structure. In this example, suppose further that a banker performed a query at 10am, whose result is used in an investment plan recorded to the database at 11am. It is critical to notify the person performing the query at 10am to consider the new value and update the plan.

Comparison to Persistent Data Structures. While persistent data structures and active data structures both consider the notion of time, they are inherently different—both in terms of

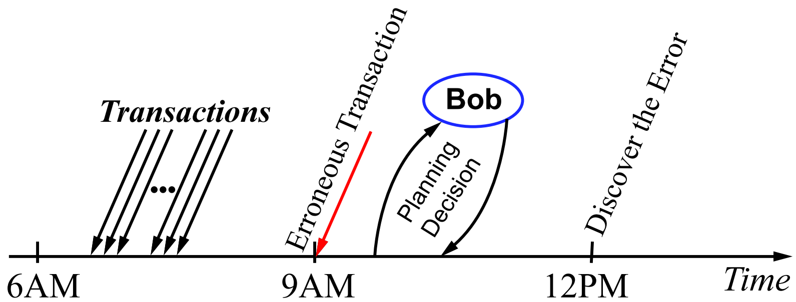

Figure 1: The ability to change the past can be important. Many real-world applications also require composibility.

their functionalities and the underlying ideas. A persistent data structure is characterized by the ability to access any past versions of the data structure. In terms of changes, the simplest form of persistency, called *partial persistency*, allows changes only in the present and queries for any past versions. A *fully persistent* data structure, although allowing changes to occur in the past, does not propagate the effects of the changes to the present. Instead, it creates a new branch in the version tree, as illustrated in Figure 2.

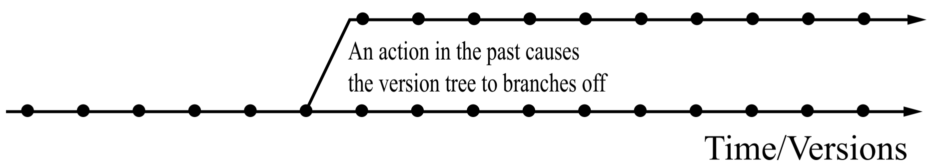

Figure 2: Changes to the past in a fully persistent data structure.

Comparison to Retroactive Data Structures. Even though retroactive data structures and active data structures appear highly related as they both allow changes to occur in the present and in the past, they significantly differ. In a retroactive data structure, operations performed on the data structure are reflected only internally. An outside party who retrieves some data from the data structure has no way of knowing that the data is outdated. This is illustrated by the bank-database example mentioned earlier. In an active data structure, the affected parties will be communicated.

The Notion of Time and Maintaining a Virtual Time Line. In order for the notion of time to make sense, we assume a time line is somehow maintained so that each action can be timestamped with a unique time. In practice, a virtual time line can be maintained efficiently using an order-maintenance data structure [DS87, BCD$^+$02]. These order-maintenance data structures can simulate a virtual time line in amortized $O(1)$. In what follows, we assume a time line is the set of non-negative reals ($\mathbb{R}^+ \cup \{0\}$), and each time-stamp is a unique real number $t \in \mathbb{R}^+ \cup \{0\}$.

3.1 Defining Active Data Structures

We introduce vocabularies for discussing about active data structures. For comparison, consider a usual data structure D with operations $operation_1$, $operation_2$, $operation_3$, …, $operation_k$. In a usual data structure, all operations are performed at the present time, altering the state of the data structure and destroying the previous version. An active data structure enables performing (or undoing) the operations at any time. We define 3 meta-operations that characterize active data structures as follows:

- The meta-operation $\mathsf{perform}(\text{“}operation_i(\cdot)\text{”}, t)$ will perform the operation $operation_i$ at the time t. If $operation_i(\cdot)$ returns a value r_i, then the meta-operation returns the value r_i.
- The meta-operation $\mathsf{undo}(t)$ causes an undo of the operation at the time t.
- The meta-operation $\mathsf{update}(t)$ informs the data structure to "synchronize" up to time t. This operation will become more clear once it appears in context.

In addition, we assume the existence of a *discrepancy set*; this is maintained either by the data structure itself or as a part of another framework (*cf.* self-adjusting computation). The discrepancy set is a list of entities (*e.g.*, a data structure operation, human interaction) that need to take some actions, because the information the entity receives has changed since the last communication. In the banking example, an entity could be Bob, who needs to know that the information he obtained is now outdated. The idea of the discrepancy set is that, as soon as an entity is identify as discrepant, it is inserted to the set. The entries of the discrepancy set are removed and processed in an increasing order of time until the set becomes empty, at which point the data structure is fully synchronized with the current reality. In practice, the discrepancy set can be maintained in a priority queue.

3.2 Active Compare-and-Swap

We begin our discussion of active data structures by introducing a basic data structure, called *compare-and-swap*. Despite its fancy name, a compare-and-swap data structure is a plain-old data structure commonly found in algorithms. Without the notion of time, the data structure has only one operation, `touch`, and maintains a boolean variable b, whose value is initially `false`. If the data structure is *touched*, the variable b changes to `true` and remains `true` for the rest of the time. The `touch` operation returns the current status of b and subsequently updates the value of b.

This simple data structure appears extensively as a tiny component of bigger data structures and algorithms. It is, for example, a part of the bit vector in many implementations of depth-first search, indicating whether or not a node has been visited. The name compare-and-swap emerges from the actions it performs. In many applications, once b is set to `true` no more operation will be performed on the corresponding object.

Similar to a regular compare-and-swap, an active compare-and-swap has only one operation, `touch`, which can be both performed and undone at an arbitrary time. We formally describe an active compare-and-swap as follows. This description outlines the general characteristics of the data structure; an efficient compare-and-swap will be discussed later. Let S be the set of times at which the data structure is *touched*. Initially S is an empty set. For the purpose of this presentation, think of S as a global variable, which is updated as operations are performed on the data structure. Like any active data structures, an active compare-and-swap supports the following:

- **Support for the meta-operation perform.** The operation perform("touch", t)—performing a `touch` at time t—maintains the following invariants. The return value of the operation is `false` if and only if $t < \min S$, with $\min \emptyset = +\infty$. After the operation, the set S is augmented to contain t. That is, $S := S \cup \{t\}$.
- **Support for the meta-operation undo.** The operation undo(t)—undoes a `touch` at time t—updates the set S as $S := S \setminus \{t\}$.
- **Support for the meta-operation update.** The operation checks if the return value of the `touch` operation at that time has changed. If the value is changed, it tells the discrepancy set that the person who retrieves this information is *discrepant*.

We point out that both of these operations may alter the return values of some other `touch` operations. An active compare-and-swap data structure has to be able to identify these operations and re-synchronize accordingly.

Efficient Compare-and-Swap. In the remaining of this section, we describe an efficient maintenance of an active compare-and-swap data structure. For ease, we maintain the set S in a balanced search tree (*e.g.* red-black tree). Maintaining S as a combination of a priority queue and a hash table is an equally viable option and will yield the same asymptotic bounds. The needed basic set operations, including finding the minimum, can be trivially performed on a balanced binary search tree in $O(\log |S|)$.

As mentioned earlier, an operation can affect the return values of other operations. We observe that, in this particular data structure, an operation can affect at most 1 other operation. Performing a `touch` at the time t will cause a side-effect if and only if $t < \min S$, in which case the return value of the old minimum is altered. An undo of the operation at time t causes a side-effect if only if t is the current minimum of S, in which case the return value of the second minimum of S is changed. We note that the number of active `touch`'s T is same as $|S|$ throughout, and establish the following theorem.

Theorem 6 *An active compare-and-swap can be maintained in $O(\log T)$ for all operations, where T is the number of active* `touch`*'s. We say that a* `touch` *operation is active if it has been performed and has not been undone.*

3.3 Active Monotone Priority Queue

We consider the problem of maintaining an active monotone priority queue. The monotone assumption greatly simplifies the problem but suffices for many applications; at the end of this section, we discuss problems with generalizing this to an unrestricted priority queue. Without loss of generality, we assume that the keys (*a.k.a.* priorities) are positive real numbers ($\mathbb{R}^+$) Informally, a monotone priority queue disallows inserting a key smaller than latest minimum removed by the *deletemin* operation. We precisely formulate the problem and formally state this assumption below.

Let DM be the set of times at which a *deletemin* occurs. Let INS be the set of ordered pairs of the form (k,t). Each entry $(k,t) \in \mathsf{INS}$ denotes the insertion point of the key k at the time t. We assume for simplicity that no duplicate keys are inserted.

Definition 7 *Let* $\mathsf{PQ}(t)$ *be the set containing all keys inserted on or before the time* t *that have not been removed by the* deletemin *operation. The set* $\mathsf{PQ}(t)$ *is a hypothetical snapshot of the priority queue at time* t*. Initially* $\mathsf{PQ}(0) = \emptyset$*.*

Definition 8 (Monotone Assumption) *Let* $f(t) := \min \mathsf{PQ}(t)$*. A priority queue is said to be* monotone *if for all* $t_1, t_2 \in \mathsf{DM}$ *with* $t_1 < t_2$*,* $f(t_1) < f(t_2)$*.*

Our priority queue supports two operations: *insert* and *deletemin*. The problem of maintaining active priority queue has the following characteristics:

- **Support for the meta-operation** $\mathsf{perform}$**.** The operation $\mathsf{perform}(\text{"}\mathtt{insert}(k)\text{"}, t)$—performing an insertion of the key k at time t—causes the following changes: $\mathsf{INS} := \mathsf{INS} \cup \{(k,t)\}$. This operation returns nothing.

  The operation $\mathsf{perform}(\text{"}\mathtt{deleteMin}\text{"}, t)$ calculates the return value as $\min \mathsf{PQ}(t)$ and alters DM to $\mathsf{DM} \cup \{t\}$. We note that this description does not describe how to efficiently maintain such a structure.

- **Support for the meta-operation** undo**.** The operation $\mathsf{undo}(t)$ for an `insert` operation simply removes the corresponding $(*, t)$ from INS. Similarly, the operation $\mathsf{undo}(t)$ for an `deleteMin` operation removes t from DM.

* * * * * * * *

Efficient Monotone Priority Queue. We give a solution to the problem of efficiently maintaining an active monotone priority queue. Our data structure maintains 2 balanced binary search trees (*e.g.* red-black, AVL tree): T_{DM} and T_{INS}, corresponding to the sets DM and INS, respectively. The tree T_{DM} is naturally ordered by time. The other tree T_{INS} is indexed by first the key then the the time. That is, $(k_1, t_1) < (k_2, t_2)$ if and only if $(k_1 < k_2)$ or $((k_1 = k_2) \wedge (t_1 < t_2))$. Each `deletemin`—a node of T_{DM}—is linked with a node of T_{INS} whose key is the result of that `deletemin`. Each node of T_{INS} is vice versa linked with its corresponding `deletemin` when applicable.

As an active data structure, our active priority queue is responsible for identifying the operations whose return values are affected, in addition to maintaining the invariants stated earlier. The problem of identifying the affected operations are discussed later. We will now focus on the problem of the maintaining the aforementioned invariants:

- The operation $\mathsf{perform}(\text{"}\mathtt{insert}(k)\text{"}, t)$ results in an insertion of the key (k,t) to the tree T_{INS}. The undo operation for an `insert` is handled vice versa by removing the key (k,t) from the tree.

- The operation $\mathsf{perform}(\text{"}\mathtt{deletemin}\text{"}, t)$ adds an entry t to the tree T_{DM}, and its undo operation removes t from T_{DM}. The return value of $\mathsf{perform}(\text{"}\mathtt{deletemin}\text{"}, t)$ is derived as follows.

Compute $t^- := \max\{t' \in \mathsf{DM} : t' < t\}$, with $\max \emptyset = -\infty$. If $t^- = -\infty$, report the minimum key in T_{INS}. Otherwise, query T_{DM} for the corresponding value the delete; call this value k^-. Make another query to T_{INS} for $k^* := \min\{k' \in \mathsf{Keys}(\mathsf{INS}) : k' > k^-\}$, where $\mathsf{Keys}(\mathsf{INS})$ contains all the keys existed in INS. Report k^*. It is trivial to fix the cross-links between the two trees.

* * * * * * * *

A Ray-shooting Game and Discrepancy Detection. We introduce a "ray-shooting" game, which illustrates the processing of finding the affected operations. Every sequence of operations in a retroactive data structure can be visually represented as follows. Consider a two-dimensional diagram, whose horizontal axis represents time and vertical axis represents the keys of the elements of interest. Recall that each key is a positive real number. An example of such diagram is shown in Figure 3 (left). Each horizontal line represents the lifespan of its corresponding key. It originates where the key is inserted and extends to the right either indefinitely if it is never removed by a `deletemin` or until it vanishes at the time when removed by a `deletemin`. Each vertical line represents a `deletemin` operation. It always originates from the horizontal axis and extends upward up to the height of the key that the `deletemin` operation returns.

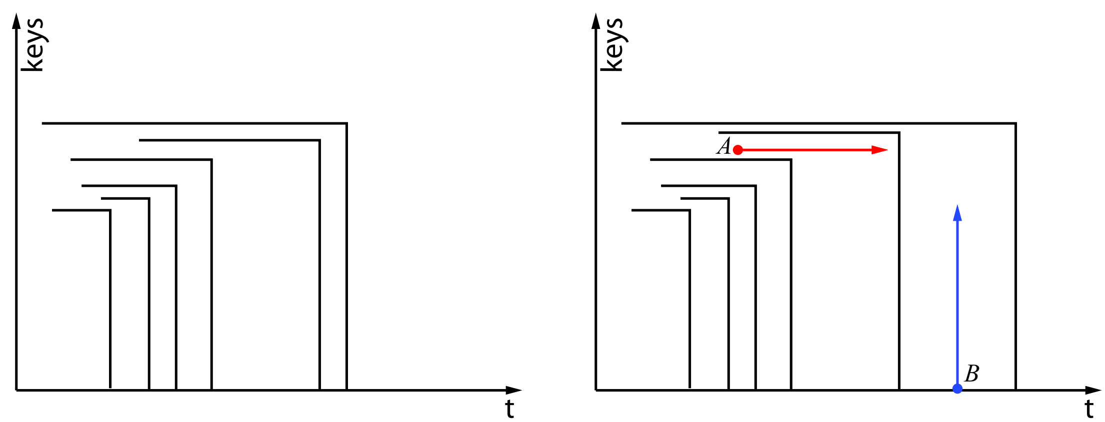

Figure 3: Visualizing an Active Priority Queue using Ray-shooting

The task of identifying the affected operations can thought of as two cases of ray shooting: rightward and upward. Inserting a key k at time t is equivalent to ray-shooting rightward from the point (t, k), as depicted in the point A in Figure 3 (right). We say that a line segment is *occupied* if it meets with another line segment of a different orientation. Removing a key will "free" a vertical segment; such a segment is affected. A new return value for that `deletemin` can be calculated by performing an upward ray-shooting as depicted at point B in Figure 3 (right). The two other operations can be handled similarly. With the monotonicity assumption, these ray-shooting operations can be efficiently performed on the two trees in $O(\log T)$, where T is the total number of active operations on the data structure. As before, an operation is active if it has been performed but not undone.

Theorem 9 *An active monotone priority queue can be maintained in an amortized $O(\log T)$ for* *all operations, where T is the number of active operations.*

We point out that an active full priority queue is much more involved, as it demands a point-location data structure. In particular, the trick that allows us to compute the current minimum will not generalize, because it relies on the monotonicity assumption. We conclude the discussion of active priority queues by listing a number of applications that admit the monotone assumption. In Dijkstra, the distance of the nodes as maintained in a priority is monotone. The same is true for Prim's algorithm for computing a minimum spanning tree. A monotone priority queue can also be used to ensure that certain operations are performed in order. Two real-world applications that illustrate such use are discussed in a subsequent section.

3.4 Active Dictionary

A dictionary data structure supports three basic operations: *insert*($key, data$), *lookup*(key), and *delete*(key). Let U be the universe of keys the dictionary supports. We assume that either U is totally ordered or there exists a universal hash function $h\colon U \to [m]$ for $m \in \mathbb{N}$. This assumption is realistic and allows an easy maintenance of an active dictionary.

Like other active data structures we discuss, an active dictionary has the same three operations—*insert*, *lookup*, and *delete*—wrapped in the meta-operations perform, undo, and update. In order to support these operations, an active dictionary maintains the following structures internally. Each key $k \in U$ is associated with three sets: $\mathsf{INS}(k)$, $\mathsf{DEL}(k)$, $\mathsf{QUERY}(k)$. First, the set $\mathsf{INS}(k)$ keeps track of all *insert* operations performed on the key k. Entries of this set take the form (t, d), where t denotes the time and d is the data associated with that insertion. Second, the set $\mathsf{DEL}(k)$ contains the times at which *delete* operations are performed on the key k. Finally, the set $\mathsf{QUERY}(k)$ records the times at which queries to the key k take place. These sets can be trivially maintained for the required data structure operations.

Efficient Active Dictionary. The problem of maintaining an active dictionary can be reduced to a simpler problem in the following manner. When an operation about a key k is received, the data structure refers to the three sets—$\mathsf{INS}(k)$, $\mathsf{DEL}(k)$, and $\mathsf{QUERY}(k)$—on which actions are performed. We note that it suffices to consider only these three sets to support any operations regarding the key k. Because of our assumption about the universe of keys, locating the corresponding three sets can be easily accomplished by a hash table or a (balanced) binary search tree.

We now turn our focus to maintaining the three sets for each key and how they can be used to support all dictionary operations. Therefore, when the context is clear, we drop the mention of key. Consider Figure 4. The sets INS and DEL combine to a set of intervals in which the key is present in the dictionary. Formally, let

$$d_>(t) := \{t' \in \mathsf{DEL} : t' > t\}$$

and

$$\mathsf{IV} := \{(\alpha, \inf d_>(\alpha)) : \alpha \in \mathsf{TIME}(\mathsf{INS})\},$$

where (α, β) denotes the open-interval from α to β (exclusive), and $\inf(\cdot)$ is the infimum. We note that at most one interval is unbounded, corresponding to an insertion which is never deleted. A lookup at time t finds the key (and its corresponding data) if and only t lies in one of the intervals

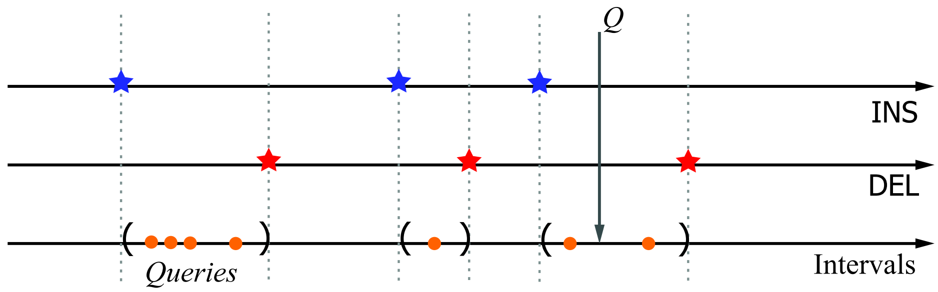

Figure 4: The sets INS and DEL combine to a set of intervals in which the key is present in the dictionary.

of IV. If a lookup at time t satisfies $t \in (\alpha, \beta)$ for $(\alpha, \beta) \in$ IV, then the *lookup* operation finds the key, where the data is the data of the insertion at time α.

Let T_k be the number of operations on which the key k. The sets INS, DEL, and QUERY can be trivially represented with balanced binary search trees, where needed operations take at most $O(\log T_k)$ time.

Theorem 10 *For a fixed key k, an active dictionary can be maintained in an amortized $O(\log T_k)$ for all operations, where T_k is the number of active operations on the key k. Let n the total number of keys. Assume that a balanced binary search tree is used to locate the right triple of sets; then, every operation in the active dictionary runs in $O(\log n + \log T_k)$.*

4 Dynamic Heap Sort

We demonstrate how an active data structure can be used in conjunction with the self-adjusting library to implement a dynamic algorithm. We consider a simple problem of maintaining a sorted list: Given a list L, the algorithm maintains `sort`(L) under possible changes to L—insertions/deletions of elements. In the static scenario, this can be easily accomplished by a heap as in Algorithm 1.

Algorithm 1 Simple Heap Sort on a List L

$PQ \leftarrow \emptyset$
foreach $x \in L$ **do** $PQ \leftarrow PQ \cup \{x\}$
$L' \leftarrow$ `nil`
while PQ is not empty **do**
  $x \leftarrow$ `deleteMin`(PQ)
  Append x to L'
end while

The Standard ML code in Figure 5 shows a dynamic (self-adjusting) heap sort, using an active priority queue. The function **feed** inserts the list elements of L into the priority queue, and the function **reel** assembles a sorted list based on the `deleteMin` operation.

```
fun feed l =
  l ⊸→ (fn c ⇒
    (case c of
    ML.NIL ⇒ ()
    | ML.CONS(h, t) ⇒ t ⊸→ (fn ct ⇒ liftFeed ([Box.indexOf h],ct) (fn t ⇒
      let val () = PQ.insert mpq h
      in feed t
      end))))
fun reel () =
  (PQ.deleteMin mpq) ⊸→ (fn vopt ⇒
    case vopt of
    NONE ⇒ ML.write ML.NIL
    | SOME(h) ⇒ liftReel ([Box.indexOf h])(fn () ⇒
      let val tail = C.modref(reel ())
      in ML.write (ML.CONS(h, tail))
      end))
```

Figure 5: Dynamic Heap Sort in Standard ML

5 Dynamic and Kinetic Convex Hull in 3-d

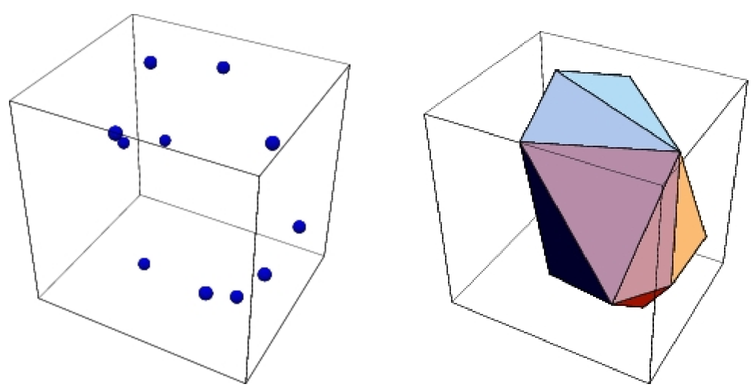

Figure 6: 3-d convex hull of a set of points

Informally, the *convex hull* of a set of points S, denoted $\mathsf{CH}(S)$, is the smallest polyhedron enclosing these points. An example of the convex hull of a set of points is given in Figure 6. Formal treatments of the material can be found in a standard computational geometry text [BY98, dBSvKO00]. The problem of maintaining convex hulls has been studied extensively both in the ordinary and the dynamic (incremental) settings [Gra72, OvL81, Ove83, Mul91c, dBSvKO00, BJ02, Cha06]. We discuss some essential preliminaries here.

5.1 Preliminaries

Let $\mathcal{O}$ be the universe of points. For the purpose of this discussion, $\mathcal{O}$ is a finite subset of $\mathbb{R}^3$. As common in literature, a convex hull is defined by means of its boundary, which consists of *faces*.

For simplicity, each *face* of a convex hull is an oriented triangle, consisting of three directed *edges*. Faces and edges are presented as ordered tuples of points. This is illustrated in Figure 7.

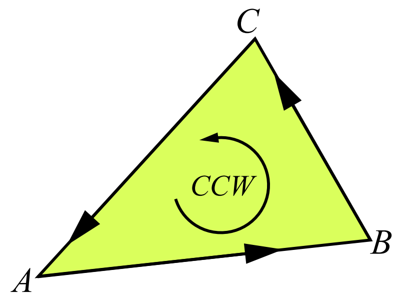

Figure 7: An Oriented Face

Let the face in the figure be called f. We write $\texttt{edges}(f)$ to refer to the set of edges of a face. For example, the edge (A, B) represents an edge that goes from A to B, and the face (A, B, C) represents a face where the points A, B and C are ordered counter-clockwise in that order. The edges of the face (A, B, C) is the set $\texttt{edges}(f) = \{(A, B), (B, C), (C, A)\}$. We write $\texttt{flip}(e)$ to denote the edge that runs in the reverse direction of e, i.e. $\texttt{flip}((A, B)) = (B, A)$.

We employ the simplicial complex representation for storing the convex hull. In the simplest form, a convex hull is maintained in a dictionary D_H. The dictionary supports three operations: *insert*$(key, data)$, *lookup*$(key$, and *delete*(key). We assume that no duplicate keys exist. The dictionary maintains a mapping from edges to faces. By carefully choosing the orientation of the edges, this mapping is sufficient for traversing the convex hull. Each face (p_1, p_2, p_3) corresponds to three directed edges $(p_1, p_2), (p_2, p_3), (p_3, p_1)$. We note that even though (p_1, p_2, p_3) is not a unique representation of the corresponding face, the set of edges $\{(p_1, p_2), (p_2, p_3), (p_3, p_1)\}$ provides a unique representation of the face.

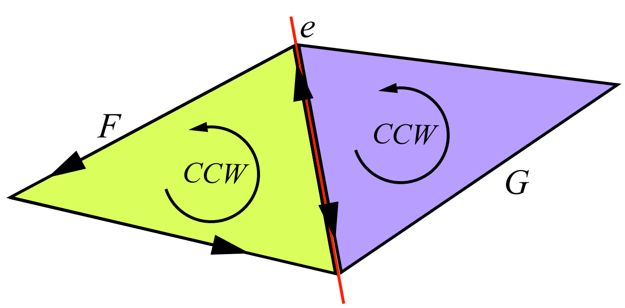

Figure 8: Adjacent faces F and G share an edge e.

We observe that if two adjacent faces F and G share an edge e, one direction of e belongs to F, and the other belongs to G. This is illustrated in Figure 8. This observation implies that the neighbors of a face f with directed edges $\overrightarrow{e_1}, \overrightarrow{e_2}, \overrightarrow{e_3}$ are the faces associated to $\texttt{flip}(\overrightarrow{e_1})$, $\texttt{flip}(\overrightarrow{e_2})$, $\texttt{flip}(\overrightarrow{e_3})$.

Facts about Randomized Computation in Computational Geometry. We introduce more notations and definitions and reiterate a lemma of Clarkson and Shor for bounding the number of objects at conflict with a particular region. Assume that each region is defined by at most b objects for some $b \in Z^+$. Let $\mathcal{F}_j^i(S)$ denote the subset of regions of $\mathcal{F}(S)$ containing exactly the regions that are defined by precisely i objects of S and are at conflict with precisely j objects of S. We further define

$$\mathcal{F}_{\le j}^i(S) = \bigcup_{k \le j} \mathcal{F}_k^i(S) \quad \text{and} \quad \mathcal{F}_j(S) = \bigcup_{k \in \mathbb{Z}^+} \mathcal{F}_j^k(S)$$

A *r-random sample* of a finite set of objects S is an r-subset of S chosen at random (*i.e.* with probability $1/\binom{n}{r}$). We note that the model of equal chance mentioned earlier always yields an $|S_i|$-random sample at the i-th stage.

Let $f_j(r, S)$ be the expected size of $\mathbf{E}[|\mathcal{F}_j(R)|]$, where the expectation is taken over r-random samples of S.

Lemma 11 (Clarkson and Shor [CS89])

$$\mathbf{E}[|\mathcal{F}_{\le j}^i(S)|] = O\left(j^i f_0(\lfloor n/j \rfloor, S\right)$$

Proof: We supply a proof due to Clarkson and Shor [CS89] for completeness. The proof uses the random-sampling technique. Let R be an r-random sampling of S. First, we note that a region $v \in \mathcal{F}_j^i(S)$ is free of conflicts with objects in a set $R \subseteq \mathcal{O}$ if and only if all i objects defining v belong to R while all j objects at conflict with v lie elsewhere. That is, $\mathbf{Pr}(v \in \mathcal{F}_0(R))$ is given by:

$$\mathbf{Pr}(v \in \mathcal{F}_0(R)) = \frac{\binom{n-i-j}{r-i}}{\binom{n}{r}} = \frac{(n-i-j)!r!(n-r)!}{(n-r-j)!(r-i)!n!} \ge \frac{1}{4} \frac{r \cdots (r-i+1)}{n \cdots (n-i+1)}$$

The expectation $\mathbf{E}[|\mathcal{F}_0^i(R)|]$ is computed as follows.

$$\begin{aligned}
\mathbf{E}[|\mathcal{F}_0^i(R)|] &= \sum_{j=0}^{n-i} \sum_{v \in \mathcal{F}_j^i(S)} \mathbf{Pr}(v \in \mathcal{F}_0(R)) \ge \sum_{k=0}^{j} \sum_{v \in \mathcal{F}_j^i(S)} \mathbf{Pr}(v \in \mathcal{F}_0(R)) \\
&\ge \left(\sum_{k=0}^{j} \left|\mathcal{F}_k^i(S)\right| \right) \frac{1}{4} \frac{r \cdots (r-i+1)}{n \cdots (n-i+1)} = \left|\mathcal{F}_{\le j}^i\right| \frac{1}{4} \frac{r \cdots (r-i+1)}{n \cdots (n-i+1)}
\end{aligned}$$

Therefore, $\left|\mathcal{F}_{\le j}^i\right| = O(j^i f_0(\lfloor n/j \rfloor, S))$. ∎

Facts about 3-d Convex Hulls. Few properties of convex hulls in the three-dimensional space are worth mentioning. Using the notation presented earlier, we point out that, in the context of 3D convex hulls, $\mathcal{F}_0(R)$ is the set of faces of the convex hull of the set R. Thus, $f_0(r, S)$ is the expected number of faces on a r-random sample of S. The following lemma is well-known; we state it without a proof.

Lemma 12 *Assume general position of points. For 3D convex hulls,*

$$f_0(r, \mathcal{O}) = O(r).$$

Since each face is defined by 3 points, it immediately follows from this lemma and Lemma 11 that the expected number of faces at conflict with at most j points is linear in the number of input points.

* * * * * * * *

A Model of Equal Chance. We describe a randomized model for sequences of insertions and deletions. The model is a variant of the models introduced and studied by Mulmuley [Mul91a, Mul91c, Mul91b], Boissonnat*et al.* [BDS+92], and Schwarzkopf [Sch91].

In this model, an adversary chooses the universe $\mathcal{O}$ and a sequence $\delta \in \{+,-\}^*$ of a finite length. The entry δ_i denotes the action to take place at the i-th step: $+$ for an insert, and $-$ for a delete. The algorithm starts with $S_0 = \emptyset$ and goes through stages. There are two possible transitions from S_i to S_{i+1}, depending on the value of δ_i:

- For an insertion, the algorithm picks a point $p \in \mathcal{O} \backslash S_i$ uniformly at random, resulting in $S_{i+1} = S_i \cup \{p\}$.
- For a deletion, the algorithm picks a point $p \in S_i$ uniformly at random, resulting in $S_{i+1} = S_i \backslash \{p\}$.

This description implies that S_i is equally likely to be any one of the $|S_i|$-subsets of $\mathcal{O}$.

* * * * * * * *

Kinetic Setting. Instead of stationery points, the locations of points in the kinetic setting change as a function time. The reader is referred to the survey papers of Guibas [Gui04] for a comprehensive treatment of the subject. In a nutshell, the location of a point p is determined as $(x_p(t), y_p(t), z_p(t))$. The combinatorial structure of interest (convex hull, in this case) is maintained as the time progresses.

5.2 An Algorithm for Maintaining 3-d Convex Hulls

This section outlines an algorithm for maintaining 3D convex hulls that we dynamize and kinetize. The algorithm is a slight modification of the standard incremental convex hull. Our variant of incremental convex hull uses the simplicial complex representation and maintains the conflict graph somewhat differently from the version described in Schwarzkopf *et al.* [dBSvKO00]. We provide a pseudo-code of the algorithm in Algorithm 2.

As typical with most incremental algorithms, the main function (`hull`) takes a list of points L and constructs the convex hull by incrementally inserting each point. The algorithm assumes the list L is pre-permuted.

In order to compute the hull efficiently, the algorithm maintains various relations between points, edges, and faces. As described earlier, the convex hull is maintained as a mapping from edges to faces. In addition, the algorithm maintains a *conflict map* and a *visibility map.* A conflict map associates each face with a list of points (not yet inserted) that conflicts with the face. A *visibility hint* provides a partial mapping for which point sees which face. This is a hint rather than a complete mapping for reasons that will soon be apparent.

We say that a point p is at a *direct conflict* with the face f if the ray $\overrightarrow{cp}$ penetrates the face f. Immediately before the insertion of the point with rank i, the following three invariants hold: 1) the hull is the convex hull of the points of all points with rank less than i, 2) for any point p_j with $j > i$, V maps p_j to a face, 3) and each face f of the hull has a conflict list consisting of all remaining points with direct conflicts with the face f.

These functions are represented using three different dictionaries: the convex hull dictionary (H), the conflict dictionary (C), and the visibility dictionary (V). The pseudo-code assumes that the hull, the visibility and conflict maps are initialized to contain the convex hull of the first three points and the resulting conflict and visibility maps. In addition, the algorithm uses a priority queue data structure that supports `insert` and `deleteMin` operations.

To insert a point p on the hull, the algorithm first finds a face f that is visible to the point p being inserted. The algorithm then checks if f is in the hull. If not, then p is inside the hull, in which case the hull remains the same. Otherwise the algorithm calls the `rip` function to remove all the faces that are visible from p by using one of the edges of the face f. The function returns the set of points π that conflict with the removed faces, the boundary edges $\mathcal{E}$ of the hull that are adjacent to ripped out faces, and the partial hull H. The region removed from the hull is then tented at p by calling the `tent` function.

Given a point p, a set of boundary edges $\mathcal{E}$, and a set of points π, the `tent` function extends the hull by inserting the point p to the hull. This requires making a face from each boundary edge and the point p. For each new face f, the function determines the points π' that conflict with f and updates the conflict dictionary. The algorithm then identifies the point p_m with the minimum rank that conflicts with f and updates the visibility map by mapping the p_m to f.

A dynamic algorithm is derived from applying syntactic transformation techniques of self-adjusting computation. By pairing the dynamic algorithm with our kinetic library [ABTV06], we obtain a kinetic algorithm for 3-d convex hull that also supports dynamic changes.

5.3 Analysis of the Algorithm

We present a few facts (and their proofs) about the algorithm, and informally argue about its efficiency. The input to the algorithm is a permutation of points drawn from a finite universe $\mathcal{O}$. We assume that dynamic changes obey the model of equal chance (described in the preliminaries). Let $\tau(n)$ denote the expected number of faces of the convex hull with n points. It is well-known that $\tau(n) = O(n)$.

Lemma 13 (Constant Degree) *Record all edges ever created through the lifetime of an Incremental Hull 3D. Create a graph G, whose nodes are the points of $\mathcal{O}$ and whose edges are those edges. Let $\overline{d}$ be the average degree of this graph (i.e. $\overline{d} = \frac{1}{n}\sum_{v \in V(G)} \deg_G(v)$. Then,*

$$\mathbf{E}[\overline{d}] = O(1)\,.$$

Proof: Consider the degree sum $D = \sum_{v \in V(G)} \deg_G(v)$. For the point being inserted at the i-th step to the hull, let D_i be the increase in the degree. We find that $D = \sum_{i=1}^{n} D_i$. The lemma

Algorithm 2 Modified Incremental Hull 3D

```
function hull (L) =
  while (L ≠ nil) do
    p ← first (L)
    case lookup_V(p) of
      Found(f):
        (e, _, _) ← edges(f)
        case lookup_H(e) of
          Found f:
            (π, ℰ) ← rip (p, e)
            tent (p, π, ℰ)
        NotFound:  delete_V(p)
      NotFound:
    L ← next (L)

function rip(p, e) =
  ℰ ← ∅
  π ← ∅
  Q ← flipe
  while (Q ≠ ∅) do
    e ← deleteMin(Q)
    case lookup_H(flipe) of
      Found(f):
        if isVisible(p, f) then
          π ← π ∪ lookup_C(f)
          (e1, e2, e3) ← edgesf
          delete_H(e1, f)
          delete_H(e2, f)
          delete_H(e3, f)
          delete_C(f)
          Q ← Q ∪ {e1, e2, e3} \ {e}
        else ℰ ← ℰ ∪ {e}
      NotFound:
  return (π, ℰ)

function tent(p, π, ℰ) =
  for each e = (p1, p2) ∈ ℰ do
    f ← (p2, p1, p)
    π' ← {p ∈ π : →(p_c p) ∩ f ≠ ∅}
    (e1, e2, e3) ← edgesf
    insert_H(e1, f)
    insert_H(e2, f)
    insert_H(e3, f)
    insert_C(f, π')
    π ← π\π'
    p_m ← arg min_{q∈π'} rank(q)
    delete_V(p_m)
    insert_V(p_m, f)
```

follows, as $\mathbf{E}[D_i] = O(1)$. ■

Lemma 14 *Assume that rank$(p^*) < r$. Let F be the set of faces "ripped" during the insertion of the point p_r. Let F' be the set of faces "ripped" during the insertion of the point p_r under the presence of p^*. Then, $\mathbf{E}[|F \Delta F'|] \leq C/r$ for some constant $C \in \mathbb{R}^+$.*

Proof: First, we bound the size of $F \backslash F'$. A face f is ripped initially but is not ripped when p^* is present if and only if p^* is at conflict with f. Thus, we have $F\backslash F' = \{f \in \mathsf{CH}(S_{r-1}) : \{p_r, p^*\} \subseteq \mathcal{F}(f)\} = \{f \in \mathcal{F}_2(S_{r-1} \cup \{p_r, p^*\}) : \mathcal{F}(f) = \{p_r, p^*\}\}$. According to the model of equal chance, we establish

$$\mathbf{E}[|F\backslash F'|] = \frac{1}{\binom{|\mathcal{O}|}{r+1}} \sum_{\substack{R \subseteq \mathcal{O} \\ |R|=r+1}} \frac{1}{r+1} \sum_{p^* \in R} \frac{1}{r} \sum_{\substack{p_r \in R \\ p_r \neq p^*}} |\{f \in \mathcal{F}_2(R) : \mathcal{F}(f) = \{p_r, p^*\}\}|$$

$$\leq \frac{1}{r^2} \frac{1}{\binom{|\mathcal{O}|}{r+1}} \sum_{\substack{R \subseteq \mathcal{O} \\ |R|=r+1}} |\mathcal{F}_2(R)| \leq \frac{O(\tau(r+1))}{r^2} \leq C_1/r$$

for a suitable $C_1 \in \mathbb{R}^+$.

Then, we approximate the size of $F' \backslash F$ by observing that $f \in F' \backslash F$ if and only if f is at conflict with p_r and $p^* \in \mathsf{Defn}(f)$. Thus, $F' \backslash F = \{f \in \mathsf{CH}(S_{r-1} \cup \{p^*\}) : p^* \in \mathsf{Defn}(f) \text{ and } p_r \in \mathcal{F}(f)\}$. It follows that

$$\begin{aligned}\mathbf{E}[|F'\backslash F|] &= \frac{1}{\binom{|\mathcal{O}|}{r+1}} \sum_{\substack{R\subseteq\mathcal{O}\\|R|=r+1}} \frac{1}{r+1}\sum_{p_r\in R}\frac{1}{r}\sum_{\substack{p^*\in R\\p^*\neq p_r}} |\{f\in \mathsf{CH}(R\backslash\{p_r\}) : p^*\in \mathsf{Defn}(f) \text{ and } p_r\in\mathcal{F}(f)\}| \\ &\leq \frac{1}{\binom{|\mathcal{O}|}{r+1}} \sum_{\substack{R\subseteq\mathcal{O}\\|R|=r+1}} \frac{1}{r+1}\sum_{p_r\in R}\frac{3}{r}|\{f\in \mathsf{CH}(R\backslash\{p_r\}) : p_r\in\mathcal{F}(f)\}| \leq C_2/r\end{aligned}$$

We remark that $\{f\in \mathsf{CH}(R\backslash\{p_r\}) : p_r\in\mathcal{F}(f)\} \subseteq \{f\in\mathcal{F}_{\leq 2}(R) : p_r\in\mathcal{F}(f)\}$. ■

Lemma 15 *Let G be the set of faces "tented" during the insertion of the point p_r. Let G' be defined similarly except in the presence of p^*. Then, $|G\Delta G'| \leq C/i$ for some constant $C\in\mathbb{R}^+$.*

The proof is similar to that of the previous lemma and is omitted.

Theorem 16 *The algorithm for maintain convex hulls in 3-d outlined in Algorithm 2 is $O(logn)$-stable with respect to dynamic (and kinetic) changes in the model of equal chance.*

Sketch of Proof: We argue that the trace differences for each point being inserted can be thought of as structural differences at the particular step. Following from two previous lemmas, we conclude that the total trace differences is at most $\sum_{r=1}^{n}(C_1+C_2)/r = O(\log n)$.

We further argue that every kinetic change can be simulated by deleting the relevant point and re-inserting it. An actual kinetic change is much cheaper than this simulation. Applying Theorem 5, we establish that the algorithm is also $O(\log n)$-stable for kinetic changes (assuming that each point is equally likely to participate in a kinetic event). ■

5.4 Implementation and Experimental Evaluation

We implemented a variant of the algorithm described earlier. The implementation differs from the description in Algorithm 2, as we find that, in practice, the algorithm can perform well even if the program-monotonicity assumption is relaxed. The implementation also stores the convex hull in a dynamized treap—an equivalence of an active dictionary. We note that even though Treaps allow for non-exact queries, in the scope of our use, Treaps and active dictionaries provide the same interface and functionalities. We suspect that an active dictionary will have less overhead than a Treap, since the dynamic behaviors of our Treap have to pay the overhead of self-adjusting computation.

We evaluate the effectiveness of our algorithm through a set of benchmarks, of which two are reported and discussed:

Time for initial run: This experiment measures the total time it takes to run the dynamic version on a given input. In order to determine the overhead our techniques, we divide this time by the time for running the static version of the same program.

Average time for a deletion/insertion: This experiment mimics a dynamic change in the model of equal chance discussed earlier. It measures the average time taken to perform an insertion/deletion. We start by running a self-adjusting application on a given input list. We then delete the first element in the input list and perform a change propagation. Next, we insert the element back into the list and perform a change propagation. We perform this delete-propagate-insert-propagate operation for each element. Note that after each delete-propagate-insert-propagate operation, the input list is the same as the original input. We compute the average time for an insertion/deletion as the ratio of the total time to the number of insertions and deletions ($2n$ for an input of size n).

Figure 9 (left) shows the initial run graph, and the one on the right shows an average time for a dynamic change.

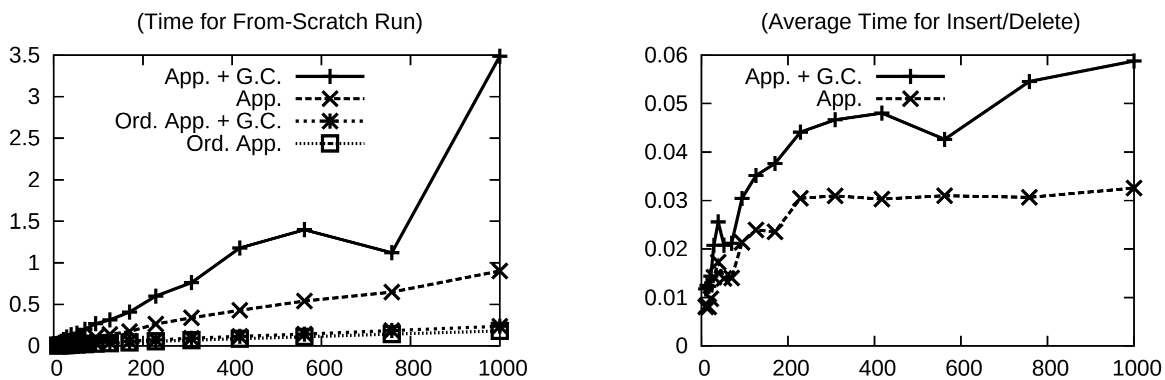

Figure 9: Experimental Results for the Convex Hull in 3D

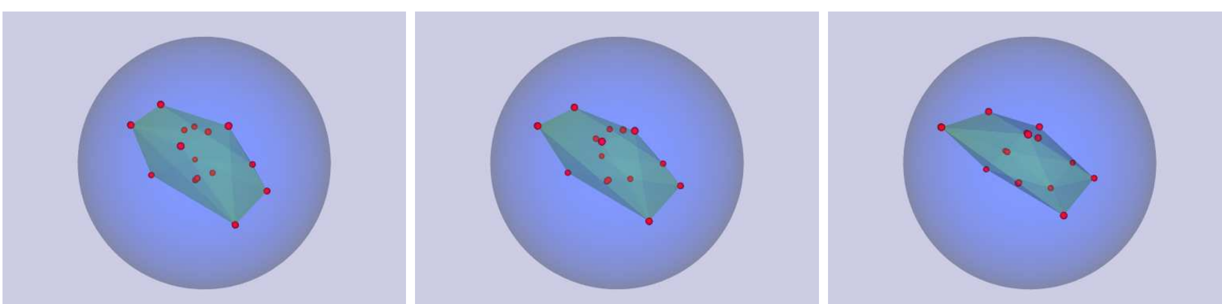

Figure 10: Frames from kinetic simulation. The convex hull is maintained by our code, but the figures were generated with Povray.

Kinetic Simulation. We run a kinetic simulation for the following scenario and produce a video clip; few frames of the clip are displayed in Figure 10. Consider a perfectly elastic unit sphere with a number of gas molecules inside of it. As the gas molecules hit the surface of the sphere, they bounce off without losing any energy (*i.e.* angle of incident is the same as angle of reflection, the velocity is maintained). Further development on the display engine may allow a real-time display of this simulation.

6 Dynamic Single-Source Shortest-Path

We study the Dijkstra algorithm for computing the single-source shortest-path in a graph. As pointed out earlier, the distance of the node removed from the priority admits the monotone property. Our study shows that, with an appropriate graph representation, the combination of self-adjusting computation and retroactive priority queue can yield a simple implementation of dynamic single-source shortest-path with competitive performance to that of Ramalingam and Reps [RR96] if theirs used a binary heap.

We have implemented the graph represetation and dynamized the Dijkstra algorithm. Our algorithm is $O(\|\delta\| \log \|\delta\|)$-stable, where δ is the sum of degrees of nodes whose distances change. We further note that our dynamic single-source shortest-path algorithm can be trivially used to devise a dynamic all-pair shortest-path algorithm. The performance, however, may not be comparable to the best asymptotic bounds to date.

7 Discussions and Conclusion

In this thesis, we present a study of a new data-structuring paradigm and demonstrate some practical use of it. We show that a number of data structures can be efficiently maintained, delivering tremendous impacts on the design, analysis, and implementation of dynamic and kinetic algorithms. The approach has been proven effective in practice, as supported by the experimental evidence.

We mention some work in progress and point out certain directions that this work can be extended. There are many other data structures that we have yet to investigate. Even though a priority queue can simulate a queue, we wonder if it is possible to construct an active queue that beats $O(\log T)$ runtime. In general, it is natural to find a non-trivial lower-bound for this class of data structure. Another interesting data structure to consider is union-find. An efficient active union-find data structure may enable the development of dynamic minimum spanning tree algorithm based on Prim's algorithm.

A The Triangle Inequality Theorem

Theorem 17 (Triangle Inequality for Change Propagation) *Let P be a monotone program with respect to the class of changes Δ_1 and Δ_2. Suppose that P is $O(f(n))$ and $O(g(n))$ stable for Δ_1 and Δ_2, respectively, for some measure n. P is also monotone with respect to the class of changes $(\Delta_1 \circ \Delta_2)$ obtained by composing Δ_1 and Δ_2, then P is $O(f(n)+g(n))$ stable for Δ_1 o Δ_2.*

Proof: Let T_0, T_1, and T_2 be the traces of P with some inputs I_0, I_1, and I_2, respectively, such that $I_1 = \Delta_1(I_0)$, and $I_2 = \Delta_2(I_1)$. Note that $I_2 = (\Delta_2 \circ \Delta_1)(I_0)$.

To prove the theorem, we will show that the distance $\delta_{tr}(T_2, T_0)$ between T_2 and T_0 is bounded by $\delta_{tr}(T_2, T_1) + \delta_{tr}(T_1, T_0)$. Let $\mathbf{w}(\cdot)$ denote the weight of a vertex. We know by definition that the following hold.

$$\delta_{tr}(T_2, T_0) = \sum_{v \in T_2 \setminus T_0} \mathbf{w}(v) + \sum_{v \in T_0 \setminus T_2} \mathbf{w}(v)$$

$$\delta_{tr}(T_2, T_1) + \delta_{tr}(T_1, T_0) = \Big(\sum_{v \in T_2 \setminus T_1} \mathbf{w}(v) + \sum_{v \in T_1 \setminus T_2} \mathbf{w}(v) \Big) + \Big(\sum_{v \in T_0 \setminus T_1} \mathbf{w}(v) + \sum_{v \in T_1 \setminus T_2} \mathbf{w}(v) \Big).$$

It therefore suffices to show that $T_2 \setminus T_0 \subseteq (T_2 \setminus T_1) \cup (T_1 \setminus T_0)$ and $T_0 \setminus T_2 \subseteq (T_0 \setminus T_1) \cup (T_1 \setminus T_2)$. This follows directly from the following basic property of sets: for any three sets A, B, C, it is true that $(A \setminus C) \subseteq (A \setminus B) \cup (B \setminus C)$.

We conclude that $\delta_{tr}(T_2, T_0) \leq \delta_{tr}(T_2, T_1) + \delta_{tr}(T_1, T_0)$. Since P is $O(f(n))$ and $O(g(n))$ stable for Δ_1 and Δ_2, respectively, we have $\delta_{tr}((,T)_1, T_0) \in O(f(n))$ and $\delta_{tr}((,T)_2, T_1) \in O(g(n))$. It therefore follows that P is $O(f(n) + g(n))$ stable for the changes $\Delta_2 \circ \Delta_1$.

■

The theorem implies that composing (batching) a constant number of changes does not change the asymptotic complexity of change propagation with respect to the maximum of the change propagations with respect to each change.

* * * * * * * *

Remarks. We note that triangle inequality does not hold if the program is not monotone with respect to the change obtained by composition. For example, if the changes are insertions and deletions, then they may swap the position of two elements in a list (unless of course they are restricted to affect the same location). In this case, the program may not be monotone with respect to a swap, and the theorem will not hold. In fact, it is easy to construct examples of such programs. For example, consider some program that takes the input list $[1, 2, 3]$. Suppose that the program traverses the list from head to tail, and performs large amount of work for the item 2 but performs constant work for all other elements. Now, if we delete 3 perform a change propagation and insert it back before 2 and run change propagation, then both change propagation will take constant time (i.e., the inputs are $[1, 2, 3], [1, 2], [1, 3, 2]$). If instead, we delete 2 and insert it in front of 3 and perform change propagation, then the result for 3 will be found in the memo, but the result for 2 will not be found—since 3 comes after 2 re-using the result for 3 will delete the result for 2. Thus the result for 2 will have to be recomputed requiring possibly non-constant time.

B Code for Active Priority Queue

```
(*
 * Active Monotone Priority Queue
 *
 * Kanat Tangwongsan
 *
 *)

functor HAPriorityQueue (structure Item : PQUEUE_ITEM) : PQUEUE =
struct

  exception NYI
  exception delOpsTreapScrewedUp
  exception InconsistentPQ
  structure C = Comb

  (* define deleteMinClosure *)
  structure deleteMinClosure =
  struct
    type t = TimeStamps.t*(unit Modref.t)*
     ((TimeStamps.t*Item.t) option Modref.t)
  end

  structure KeysNode =
  struct
    type vtype = Item.keyt*TimeStamps.t
    type data = Item.datat*(deleteMinClosure.t option ref)

    fun compare((x,xt),(y,yt)) =
     (case Item.compare(x, y) of
  EQUAL => TimeStamps.compare(xt, yt)
| x => x)

    fun toString _ = raise NYI
  end

  structure DelOpsNode =
  struct
    (* store a bunch of deleteMinClosures *)
    type vtype = TimeStamps.t
    type data = deleteMinClosure.t

    val compare = TimeStamps.compare

    fun toString _ = raise NYI
  end

  (* Definitions of the two traps *)
  structure KeysTreap = BinaryTree (KeysNode)
  structure DelOpsTreap = BinaryTree (DelOpsNode)
```

```
  type pqt =  ((KeysNode.data KeysTreap.t)*
   (DelOpsNode.data DelOpsTreap.t)) ref
  type t = pqt

  type keyt = Item.keyt
  type datat = Item.datat
  type eltt = Item.t

  val dPrint = fn _ => () (*print*) (* fn _ => ()  *)

  fun new () : pqt = ref (KeysTreap.empty, DelOpsTreap.empty)


(*********************************************************************
  Utility functions
 *********************************************************************)

  fun pickUpClosure (qr:pqt) key =
    let val (o_kt, _) = !qr
    in case KeysTreap.lookup o_kt key of
 NONE => NONE
      | SOME(_,clsRef) => !clsRef
    end

  fun eliminateKey (qr:pqt) key =
    let val (o_kt,o_dt) = !qr
val kt' = KeysTreap.delete o_kt key
    in qr := (kt', o_dt)
    end

  fun scheduleWakeUp cls =
    let
      val (ts,synA,_) = cls
    in case TimeStamps.compare (ts, !Modref.now) of
 LESS => ()
      | _ => C.iwrite' (fn _ => false) synA ()
    end

  fun doRemoveInsert (qr:pqt) x () =
    let val wCls : deleteMinClosure.t option = pickUpClosure qr x
(*        val () = print "eliminating an insert\n"  *)
val () = eliminateKey qr x
    in case wCls of
   NONE => ()
 | SOME(cls) => scheduleWakeUp cls
    end

  (* NOTE: fetchMinAtTime has a side-effect of removing all "future"
   * insertions whose keys are bigger the smallest element not yet
```

```
   * used
   *)
  fun fetchMinAtTime (qr:pqt) time : (KeysNode.vtype*KeysNode.data) option =
    let
      fun findMin pvMinItem pvTs : (KeysNode.vtype*KeysNode.data) option =
let val (o_kt, o_dt) = !qr
          val curMin =
let val (lastKey, _) = Item.explode pvMinItem
in KeysTreap.minStrictGT o_kt (lastKey, pvTs)
end
        in case curMin of
     NONE => NONE
   | SOME(wKey as (k,curMinTs),(v,_)) =>
     (case TimeStamps.compare (curMinTs, time) of
LESS => curMin
      | _ =>
let val () = dPrint "fetchMinAtTime: deleting out of time key\n"
in  (doRemoveInsert qr wKey ();
     findMin (Item.implode(k,v)) curMinTs)
end)
(*       | _ => curMin*)
end

      val (o_kt, o_dt) = !qr
      val prevDelMin : (DelOpsNode.vtype*DelOpsNode.data) option =
  DelOpsTreap.maxStrictLT o_dt time
    in
      case prevDelMin of
 (*(dPrint ("no prev dM\n");KeysTreap.findMinOpt o_kt)*)
NONE => (case (KeysTreap.findMinOpt o_kt) of
  NONE => NONE
| SOME(wMin as (wKey as (_,ts),_)) =>
  (case TimeStamps.compare (ts, time)  of
     LESS => SOME(wMin)
   | _ => (doRemoveInsert qr wKey ();
   fetchMinAtTime qr time)))

      | SOME(_,(_,_,dm)) =>
(case (Modref.deref dm) of
     NONE => NONE
   | SOME(ts,lastMin) => findMin lastMin ts)
    end

  fun findBiggerKey (qr:pqt) key =
    let val (o_kt,o_dt) = !qr
(*        val tl = map (fn (x,_) => x) (KeysTreap.toListPre o_kt)
        val str = foldr (fn (x,s) => (Item.keyToString x)^" "^s) "" tl
        val () = dPrint (str^"\n")*)
    in case KeysTreap.minStrictGT o_kt key of
 NONE => (* find the NONE value *)
 let
```

```
   val wCls =
       case (KeysTreap.findMaxOpt o_kt) of
 NONE => DelOpsTreap.findMinOpt o_dt
       | SOME(_,(_,dmc)) =>
 (case (!dmc) of
    NONE => NONE
  | SOME(ts,_,_) => DelOpsTreap.minStrictGT o_dt ts)
 in
   case wCls of
     NONE => NONE
   | SOME(_,dmc) => (SOME dmc)
 end
       | SOME(_,(_,dmc)) => !dmc
    end

  fun nextClosure (qr:pqt) ts =
    let val (_,o_dt) = !qr
    in case (DelOpsTreap.minStrictGT o_dt ts) of
 NONE => NONE
       | SOME(_,v) => SOME(v)
    end

  fun eliminateClosure (qr:pqt) ts =
    let val (o_kt, o_dt) = !qr
val dt' = DelOpsTreap.delete o_dt ts
    in qr := (o_kt, dt')
    end

  fun writeEqOption (a, b) =
      case (a,b) of
   (NONE, NONE) => true
         | (SOME x, SOME y) => Item.writeEq (x, y)
 | _ => false
  fun writeClosureEqOption (a,b) =
      case (a,b) of
  (NONE, NONE) => true
| (SOME(xt,x),SOME(yt,y)) =>
  (TimeStamps.compare(xt,yt)=EQUAL)
  andalso Item.writeEq(x,y)
| _ => false


  fun disassociateKey qr c2 =
    let
      val (kt,_) = !qr
    in
      case (Modref.deref c2) of
NONE => () (* dPrint ("disassoc: found nothing\n") *)
      | SOME(ts,elt) =>
let val (k,_) = Item.explode elt
        val key = (k,ts)
```

```
in case (KeysTreap.lookup kt key) of
     NONE => ()
   | SOME(_,mr) =>
     (case (!mr) of
NONE => ()
      | SOME(_) => mr := NONE)
end
    end


  fun grabKey (qr:pqt) (kk:KeysNode.vtype) (nc:deleteMinClosure.t)  =
    let
      val (kt,_) = !qr
(*      val () = dPrint "grabKey.. init\n" *)
    in
      case (KeysTreap.lookup kt kk) of
NONE => raise InconsistentPQ
      | SOME(_,mr) =>
(case (!mr) of
   NONE => mr := SOME(nc)
 | SOME(cls) =>
   let  (* val () = dPrint "grabKey: SOME/cls\n"   *)
(*              val () = disassociateKey cls*)
            val () = scheduleWakeUp cls
       val () = mr := (SOME nc)
   in ()
   end)
    end

(*********************************************************************)
  fun c2reader c2 =
      (case (Modref.deref c2) of
NONE => "NONE"
      |SOME (_,d) => let val (k,v) = Item.explode d
    in "SOME("^(Item.keyToString k)^")"
    end) handle _ => "undef."

  fun clsToHolder cls =
    let val (_,_,c2) = cls
    in
      c2reader c2
    end

(********************************************************************
  Action components
 ********************************************************************)

  fun action_removeInsert (qr:pqt) x () = doRemoveInsert qr x ()

  fun action_removeDelMin (qr:pqt) ts () =
    let val wCls : deleteMinClosure.t option = nextClosure qr ts
```

```
(*        val () = print "eliminating a delMin\n" *)
val () = eliminateClosure qr ts
    in  case wCls of
   NONE => ()
 | SOME(cls) => scheduleWakeUp cls
    end

  fun action_addInsert (qr:pqt) x =
    let
      val (o_kt, o_dt) = !qr
      val (k, v) = Item.explode x
(*      val () = dPrint ("insert: "^(Item.keyToString k)^"\n") *)
      val insTime = Modref.insertTime ()
      val keyPack = (k, insTime)
      val () = TimeStamps.setInv (insTime, action_removeInsert qr keyPack)

      val () = (case (findBiggerKey qr keyPack) of
  NONE =>  () (* dPrint ("none to wake up\n") *)
| SOME(cls) =>
  let
(*    val () = dPrint ("call swaking up..who's holding"^(clsToHolder cls)^"\n") *)
  in scheduleWakeUp cls
  end)

      val hv = (v, ref NONE)
      val kt' = KeysTreap.insert o_kt (keyPack,hv)
    in
      qr := (kt', o_dt)
    end

  fun action_addDelMin (qr:pqt) =
    let
      val (synA,synB,outputM) = (Modref.new (), Modref.empty (), Modref.empty())
      val delMinTime = Modref.insertTime ()
      val () = TimeStamps.setInv (delMinTime, action_removeDelMin qr delMinTime)

      val (kt,dt) = !qr

      val closure = (delMinTime,synA,synB)
      val key = delMinTime
      val dt' = DelOpsTreap.insert dt (key,closure)
      val () = qr := (kt, dt')

      fun fwrap_pre () =
C.iread synA (fn () =>
         let
   (* precond: *)
(*   val () = dPrint "i am up\n" *)
(*        val (keys,_)= !qr
        val tl = map (fn ((x,_),_) => x) (KeysTreap.toListPre keys)
        val str = foldr (fn (x,s) => (Item.keyToString x)^" "^s) "" tl
```

```
        val () = dPrint (str^"\n") *)

   val curMinCls : (TimeStamps.t*Item.t) option =
       (case (fetchMinAtTime qr delMinTime) of (* should be delMinTime not now*)
    NONE => NONE
  | SOME((k,ts),(v,_)) =>
    let (*val () = dPrint ("curmin is"^(Item.keyToString k)^"\n") *)
      val () = if TimeStamps.compare(ts, delMinTime) = LESS then
 ()
       else raise Crap
    in SOME(ts, Item.implode (k,v))
    end)
(*    val () = dPrint ("before writing to c2: originally c2 = "^(c2reader c2)^"\n")*)
   val () =
       (if (writeClosureEqOption (curMinCls, Modref.deref synB)) then
 ()
       else disassociateKey qr synB) handle _ => ()
 in
           C.iwrite' writeClosureEqOption synB curMinCls
 end)

      fun fwrap_post () =
C.iread synB (fn tvalue =>
          let
(*     val () = dPrint "c2 invoked\n"  *)
    val closure = (delMinTime,synA,synB)
    val () = (case tvalue of
 NONE => ()
       | SOME(ts,kk) =>
 let val (k',_) = Item.explode kk
 in grabKey qr (k',ts) closure
 end)
(*     val () = dPrint "postc2:y\n"  *)
    val value = case tvalue of
    NONE => NONE
  | SOME(_,x) => SOME(x)
(*     val () = dPrint "transfering to c3\n" *)
          in
    C.iwrite' writeEqOption outputM value
  end)
      val _ = fwrap_pre ()
      val _ = fwrap_post ()
    in
      outputM
    end


  fun insert qr x = action_addInsert qr x

  fun deleteMin (qr:pqt) = action_addDelMin qr
 end
```

References

[ABB+05] Umut A. Acar, Guy E. Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. A library for self-adjusting computation. In *ACM SIGPLAN Workshop on ML*, 2005.

[ABBT06] Umut A. Acar, Guy E. Blelloch, Matthias Blume, and Kanat Tangwongsan. An experimental analysis of self-adjusting computation, 2006. To Appear in the Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.

[ABH02] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. In *Proceedings of the 29th Annual ACM Symposium on Principles of Programming Languages*, pages 247–259, 2002.

[ABH+04] Umut A. Acar, Guy E. Blelloch, Robert Harper, Jorge L. Vittes, and Shan Leung Maverick Woo. Dynamizing static algorithms, with applications to dynamic trees and history independence. In *SODA '04: Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 531–540, 2004.

[ABTV06] Umut A. Acar, Guy E. Blelloch, Kanat Tangwongsan, and Jorge Vittes. Kinetic algorithms via self-adjusting computation. Technical Report CMU-CS-06-115, Department of Computer Science, Carnegie Mellon University, March 2006.

[Aca05] Umut A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.

[BCD+02] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In *Lecture Notes in Computer Science*, pages 152–164, 2002.

[BDS+92] Jean-Daniel Boissonnat, Olivier Devillers, Ren Schott, Monique Teillaud, and Mariette Yvinec. Applications of random sampling to on-line algorithms in computational geometry. *Discrete & Computational Geometry*, 8:51–71, 1992.

[BGH97] Julien Basch, Leonidas J. Guibas, and John Hershberger. Data structures for mobile data. In *SODA'97*, pages 747–756, 1997.

[BJ02] Gerth Stolting Brodal and Riko Jacob. Dynamic planar convex hull. In *Proceedings of the 43rd Annual IEEE Symposium on Foundations of Computer Science*, pages 617–626, 2002.

[BY98] Jean-Daniel Boissonnat and Mariette Yvinec. *Algorithmic Geometry*. Cambridge University Press, 1998.

[Car02] Magnus Carlsson. Monads for incremental computing. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 26–35. ACM Press, 2002.

[Cha06] Timothy M. Chan. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 1196–1202, New York, NY, USA, 2006. ACM Press.

[CS89] Kenneth L. Clarkson and Peter W. Shor. Applications of random sampling in computational geometry,II. *Discrete and Computational Geometry*, 4(1):387–421, 1989.

[dBSvKO00] Mark de Berg, Otfried Schwarzkopf, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2000.

[DIL04] Erik D. Demaine, John Iacono, and Stefan Langerman. Retroactive data structures. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, pages 274–283, New Orleans, Louisiana, January 11–13 2004.

[DRT81] Alan Demers, Thomas Reps, and Tim Teitelbaum. Incremental evaluation of attribute grammars with application to syntax directed editors. In *Proceedings of the 8th Annual ACM Symposium on Principles of Programming Languages*, pages 105–116, 1981.

[DS87] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th ACM Symposium on Theory of Computing*, pages 365–372, 1987.

[Gra72] R. L. Graham. An efficient algorithm for determining the convex hull of a finete planar set. *Information Processing Letters*, 1:132–133, 1972.

[Gui04] L. Guibas. Modeling motion. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 1117–1134. Chapman and Hall/CRC, 2nd edition, 2004.

[Mul91a] Ketan Mulmuley. Randomized multidimensional search trees (extended abstract): dynamic sampling. In *Proceedings of the seventh annual symposium on Computational geometry*, pages 121–131. ACM Press, 1991.

[Mul91b] Ketan Mulmuley. Randomized multidimensional search trees: Further results in dynamic sampling (extended abstract). In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 216–227, 1991.

[Mul91c] Ketan Mulmuley. Randomized multidimensional search trees: Lazy balancing and dynamic shuffling (extended abstract). In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 180–196, 1991.

[Ove83] Mark H. Overmars. *The Design of Dynamic Data Structures*. Springer, 1983.

[OvL81] Mark H. Overmans and Jan van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166–204, 1981.

[PT89] William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 315–328, 1989.

[RR96] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph algorithms. *Theoretical Computer Science*, 158(1–2):233–277, 1996.

[Sch91] O. Schwarzkopf. Dynamic maintenance of geometric structures made easy. In *In Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, pages 197–206, 1991.