6.897: Advanced Data Structures Spring 2003

Project Presentations — May 7, 12, and 14, 2003

Prof. Erik Demaine *Scribes: Ray Jones and Mihai Pătraşcu*

1 Classical Data Structures

1.1 Partial sums data structures — Mihai Pătraşcu

The problem is to maintain an array of size n, containing b-bit integers ($b \geq \lg n$), subject to the following operations:

update(k, Δ) replaces $A[k] \leftarrow A[k] + \Delta$. Here Δ must be a δ-bit integer, where $\delta \leq b$ is a parameter of the problem.

sum(k) returns the partial sum $\sum_{i=1}^{k} A[i]$.

select(σ) returns the index i such that $\mathrm{sum}(i-1) < \sigma \leq \mathrm{sum}(i)$.

Several new results were presented. First, an upper bound of $\Theta(1 + \lg n / \lg(b/\delta))$ was obtained for the RAM model. Second, a matching lower bound on the cell-probe complexity of *update* and *sum* was derived. Finally, a tight lower bound of $\Omega(\lg n)$ was obtained for the group model of computation.

Open problems include proving a tight bound on the complexity of sequences of *update* and *select* (without the *sum* operation), and analyzing the off-line problem.

Update: This work will appear in the 15th Annual ACM-SIAM Symposium on Discrete Algorithms, 2004.

1.2 Nearest neighbor data structures — Ray Jones, Mihai Pătraşcu

This project investigated the predecessor problem in an algebraic model of computation with random access. It is known that interpolation search achieves an expected $O(\lg \lg n)$ bound if elements are drawn independently from a uniform distribution. Interpolation search trees attain this bound for the dynamic problem, but impose additional restrictions, such as deletes being uniformly random. It can be questioned whether such ideal distributions appear in practice. It would be desirable to obtain good bounds in terms of a more measurable parameter; one such parameter is the spread factor D, defined as the ratio between the maximum and minimum difference between consecutive elements.

The main result of this project is an $O(\lg D \cdot \lg\lg n)$ bound for insert, delete, and predecessor queries. The key idea is to generate random samples from a distribution defined in terms of the input values. These samples are stored in an interpolation search tree; regular binary search trees hold the values between consecutive samples. If the number of samples is appropriately chosen, the binary search trees have polylogarithmic size with high probability.

It would be interesting to analyze the performance of interpolation search in terms of D. It should also be investigated whether regular interpolation search can be used instead of interpolation search trees.

Update: This work (with an improved bound of $O(\lg D)$) will appear in the 15th Annual ACM-SIAM Symposium on Discrete Algorithms, 2004.

1.3 Faster hashing — Vladimir Kiriansky

Hashing is one of the oldest and most important data-structural problems. The collision-chaining technique yields good (optimal) average-case performance, and represents a good reference point for evaluating the performance of alternative implementations. In many contexts, however, we want constant running times in the worst case for lookups. This is achieved for the static case by the classical Fredman-Komlós-Szemerédi perfect hashing scheme, which was later generalized to the dynamic case by Dietzfelbinger et al. Unfortunately, perfect hashing has rather large constant hidden by the asymptotic analysis, especially in the space consumption.

The cuckoo hashing of Pagh and Rodler is another scheme that achieves constant worst-case bounds for lookups; in fact, the lookup routine makes only two memory probes. The space consumption is much lower, roughly $2n$. The space consumption can be reduced even further by d-ary cuckoo hashing, but at the price of increasing the number of memory probes needed during a lookup.

However, experimental evidence suggests that cache performance is more important than the number of memory probes. In this context, it was proposed that every element in the hash table be replaced by a group of k elements that fits in a cache line. This reduces the number of misses in primary search location, and does not decrease the number of cache lines that must be loaded. Experimental evidence found that this scheme is effective. An interesting open problem is to give some theoretical results along these lines.

1.4 Offline linear search — Ilya Baran, Denis Cebikins, Lev Teytelman

Self-organizing linear search is a classic computer-science problem; several data structures for the online version were discussed in class. The offline version of the problem has also been considered. It is known that the problem is NP-hard. Furthermore, a 1.6-approximation algorithm can be obtained by derandomizing the COMB algorithm, which is the best known algorithm for the online problem. This project attempted to give a better approximation algorithm.

It was noted that, in the offline case, there exists an optimal solution using no free swaps. Projective algorithms were discussed and it was remarked that the projective optimum only gives a good approximation of the real optimum for lists of 3 elements. A new algorithm for the offline case was presented. It was shown that it is always as least as good as move-to-front, but unfortunately it only gives a factor 2 approximation in the worst case.

The main open problem is to find an approximation algorithm that improves the 1.6 bound.

2 Text Indexing Data Structures

2.1 String data structures — Alexandr Andoni and Cristian Cadar

While suffix trees give an optimal solution for static text indexing, no such solution is known for the case when characters can be inserted and deleted from the text dynamically. Three solutions for this problem were proposed, having different running times in terms of $|T|$, $|P|$, and k (the number of updates seen so far):

1. The first solution maintains a static suffix tree and uses exhaustive search to find additional matches straddling update points. This gives $O(\lg k)$ updates and $O(k \cdot P)$ searches.
2. The second solution achieves $O(T^{2/3})$ updates, and $O(P^2)$ searches. This is done by maintaining a tree of prefixes and a tree of suffixes, in conjunction with a table that

gives matches defined by two nodes in these trees. The whole structure is rebuilt when $k = \Theta(n^{1/3})$.

3. The third solution achieves $O(T^{2/3})$ updates and $O(P \lg k)$ searches, by augmenting the nodes in the suffix and prefix trees with some additional links.

2.2 Dynamic text indexing — Hans Robertson

The first part of this project concerned text indexing data structures for dynamic texts. It was shown that if the length of searched strings is limited to m, where m is a parameter given in advance, then searches can be supported in optimal time, and updates in $O(m)$ time. The idea is to divide the text in blocks of size $2m$ that overlap adjacent blocks in m positions. All suffixes of these blocks are maintained in a common suffix tree.

The second part of the project concerned the document-listing problem, which was recently solved optimally for the static case by Muthukrishnan. For this problem, the allowable operations are inserting and deleting a whole document. The general approach was to augment the suffix tree with lists of distinct documents under each node, and additional information to help navigate these lists. The bounds achieved are good, but probably not optimal (by a logarithmic factor). Improving these bounds is an interesting open problem.

2.3 Fast and small text indexing — Lance Anderson and Wayland Ni

Due to the high volume of data that must be handled by static text indexing data structures, space consumption becomes an important aspect. We would like a data structure that can search for a pattern in $O(P)$ time, and yet occupy only $O(T)$ bits (for the binary alphabet case). It was shown by Demaine and López-Ortiz that $\Omega(T)$ bits are required for the index, in addition to the actual text. Some known data structures achieve this space bound, but don't quite match the time bound; others give $O(P)$ search time but use superlinear space.

The project concerned finding a data structure that achieves both bounds. The proposed solution was to construct a suffix tree over chunks of size $\lg T$, and recurse to handle these chunks. Unfortunately, this attempt fails to meet the established goals, so the problem of finding an optimal text indexing data structure remains open.

3 Computational Geometry Data Structures

3.1 Nearest-neighbor data structures — Gautam Jayaraman

Many algorithms in machine learning need to construct a neighborhood graph, and thus need a nearest-neighbor data structure. Usually, such a data structure has to handle points from a low-dimensional manifold embedded in a high-dimensional space. Karger and Ruhl presented such a data structure, whose expected performance is logarithmic in the number of points, and exponential in the intrinsic dimensionality. This represents a significant improvement compared to schemes which are exponential in the extrinsic dimension. Their data structure is randomized with a high (constant) probability of correctness, and uses "finger lists" of sample points around each point in the data set.

An experimental analysis of this scheme was presented. The evidence suggested that the proposed dimension of the finger lists was unnecessarily high. It was demonstrated that smaller finger lists yield significantly better performance. At the same time, the probability of correctness does not decrease significantly, since it quickly enters a linear region as finger lists increase. An open problem is to give a theoretical analysis confirming these observations.

3.2 Higher-dimensional van Emde Boas — Sid Sen and Shantonu Sen

The project was to define a 2D predecessor/successor structure in a fixed, two-dimensional universe. There are several ways to define successor in higher dimensions. For example, a particular direction might be chosen, and the successor of a point would be defined as the next point along a ray in that direction. Successor could also be defined as the nearest neighbor, in which case the two closest points are each other's successor.

This project considers a compromise between the ray successor and the nearest neighbor, in which a cone is defined at each point, in a particular direction and with a given angle. The successor within a cone is the closest (under the Euclidean metric) point in the cone. There are real world problems that behave like this, such as limited viewing angle of human vision.

A specific instance of this problem is when the query point is on the main diagonal (through the origin, with a slope of 1), and the search direction along the diagonal. For this case, the other points can be projected onto the main diagonal and a 1D vEB search performed. The answer is approximate, with error $1/\cos(\alpha/2)$, where α is the angle of the cone. This error comes from using perpendicular projection to sort points, rather than true Euclidean distance. In the worst case, all of the points might lie outside of the cone, but project to near the query point. In this case, the successor query takes $O(n)$ time.

The limited case can be extended to a more general case, in which the query point is allowed to be anywhere but the query direction is still 45°. Points are projected to the main diagonal, and vEB searches performed on overlapping groups. Within a group, a second vEB is performed in the perpendicular direction. Again, the worst case time might be $O(n)$ if all points lie just outside the cone.

This case can be generalized to use an arbitrary query direction. However, the number of points along a an arbitrary diagonal of slope m is pU, where $p = \frac{m\sqrt{m^2+1}}{m^2+1}$ and U is the universe size. To perform vEB searches on the projected diagonal, the granularity of the universe may have to be increased by a factor of p, which might be quite large.

Runtimes are poor in the worst case, but can be improved by searching denser data sets, limiting the distance of the search so that far away points are not considered, and iteratively updating the search based on failures in the successor search (i.e., points that lie outside the cone).

3.3 Proximate point location — Jeff Lindy

The problem this project addresses is querying a point set for whether a particular point is in that set or not. In addition, a goal is to have the queries have the dynamic finger property, so that the time for searching for point i and then point j is $O(\lg |d(i,j)|)$, where $d(i,j)$ is the difference in "rank" between the two points.

In 1D, rank can be defined by sorting the points. However, for 2D the definition is not as obvious.

A method by Demaine, Iacono, and Langerman defines rank difference as the number of points lying within a convex shape containing both points. In addition, the shape must contain a disk around the second point, and an arc through the second point centered on the first.

Given this metric, a structure with the dynamic finger property can be defined. Each point splits space into six triangles with a vertex at that point and an angle of 60°. Six lists are stored with each point, linking it to "stepping stone" points in a particular direction, chosen such that the metric between the central point and the stepping stone grows exponentially.

Search proceeds by jumping from stepping stone to stepping stone.

An open problem is whether a similar method works in 3D. The direct extension is not possible, because the 2D version relies on equilateral triangles tiling the plane, which is not the case for regular tetrahedra in 3-space. One possibility is to use cubes to define the metric, rather than tetrahedra.

3.4 Planar point location with fusion trees — Glenn Eguchi, Hana Kim

The problem being attacked in this project is planar point location: preprocess a polygonal planar subdivision with n edges to support queries of, given a point q, determine which polygon contains q.

The problem can be broken down into locating which of n vertical slabs contains the point and then which polygon (clipped to that slab) contains the point. Slabs are defined such that slab boundaries occur at (and only at) vertices of the subdivision. There are n slabs, in this case, with all clipped polygons having three or four edges. Building a fusion tree on the x-coordinates of the slab gives $O(\frac{\lg n}{\lg \lg n})$ query time for the vertical slab. The problem then becomes searching for which horizontal trapezoid contains the point.

The first method of performing this search is to again build a fusion tree on horizontal sub-slabs, where a slab is defined for each crossing of an edge with the boundaries between vertical slabs. Locating a point requires first finding the horizontal slab containing the point, then which polygon overlapping the sub-slab contains it. Unfortunately, in the worst case this could be all of the polygons.

This can be addressed by rotating the slabs, then building horizontal slabs within them. The rotation is chosen to make the median slope of the crossing lines horizontal. This approach can be applied recursively. If edge slopes are uniformly distributed, then after only a few applications all horizontal slabs will overlap a constant number of polygons.

A different approach is to perform finer subdivisions in the first step, when defining the vertical slabs. If a particular boundary for a vertical slab would cause two polygons in the slab to overlap in their horizontal projection, then a vertical subdivision where one edge's endpoint projects to the other edge produces horizontal sub-slabs that contain no more than two Repeatedly applying this procedure gives a subdivision into vertical and horizontal slabs with no sub-slabs containing more than two polygons. However, the number of subdivisions might be as high as u, the universe size.

3.5 Faster planar point location — Alan Leung

This project was concerned with planar point location in a vertical slab with n non-crossing edges. The method uses a divide-and-conquer approach to find which sub-polygon the query point lies in. Empirical tests suggest that the time to locate a point in n subdivisions is $O(\lg n \lg \lg u)$.

4 Distributed Data Structures

4.1 Amorphous data structures — Jake Beal

This project was concerned with implementing a dictionary (a.k.a. association memory), on an amorphous computer network. Such a network can support spatially local communication, is only partially synchronous, and allows for point and region failures. The proposed distributed data structure requires $O(d)$ time and messages per operation (where d is the diameter), requires a background communication density of $O(\lg d)$ and guarantees a maximum load per area of $O(d^{\epsilon} \cdot n/A)$. Furthermore, the data structure has good failure bounds. The proposed data structure was simulated with positive results.

4.2 Distributed data structures — Ben Leong

This project was also about distributed dictionaries, but on more common network of computers. It was argued that the routing path length for a query can be logarithmic, even if each node has a constant number of neighbors, which was an improvement over existing schemes. A simple idea is to organize computers in a tree with constant branching factor; unfortunately, this leads to exponentially higher traffic at higher levels and is therefore impractical. The proposed solution was to upgrade and downgrade nodes in the tree according to the hit rate of the data they hold. Experimental evidence suggested the routing length grows logarithmically with the number of nodes, and the traffic experienced by nodes higher in the tree is comparable to that at lower levels.

4.3 Distributed data structures — Rui Fan and Sayan Mitra

This project gives a general transformation from a distributed data structure to a fault-tolerant distributed data structure that is guaranteed to be failsafe. The model assumes that there is an upper bound f on the number of computers that can fail in one tick; furthermore, the data structure consists of "nodes" containing independent information. The strategy is to encode each node using an erasure code, and divide it on $2f$ computers. During each tick, $3f$ random computers are probed; if one has failed, the data can still be recovered from adjacent computers, with high probability. To guarantee successful recovery in all cases, a more costly failsafe procedure is employed. Unfortunately, this scheme puts some information regarding a node on each of the n computers, which increases the cost of writes to n messages. Since all other bounds are polylogarithmic, improving writes remains the most significant direction for future research.

5 RAMBO Data Structures

5.1 RAMBO data structures — Loizos Michael

The project was concerned with speeding up dynamic prefix problems using a RAMBO. A dynamic prefix problem asks to maintain an array $A[1..n]$ under element update, and prefix queries, which return the composition of $A[1..k]$ under a monoid operation for a given value of k.

To support interesting operations efficiently, an extension to the RAMBO architecture was proposed: in addition to the usual registers which represent root-to-leaf paths, the machine can also read dual registers, which are the "complement" of a normal path. Using this architecture, it was shown how to support prefix boolean AND and parity in constant time per operation (sometimes requiring precomputed tables in RAM). Some possible extensions to multiple dimensions were discussed, as well as possible applications to image processing.

5.2 Higher-dimensional RAMBO data structures — Chris Collier

This project's goal is to build multidimensional RAMBO data structures to solve the nearest-neighbor problem. RAMBO structures can be used to enhance searches in the k-nearest neighbor problem, via projection search. Sorting time is decreased to $O(dn)$, and query time to $O(d \lg u + k)$.

RAMBO structures can also be used to create faster multidimensional range trees. In the usual multidimensional range query tree, a range tree is built for the first dimension. Each node in the tree links to a range tree in the next dimension for the points lying within the range summarized by that node. This leads to a $O(\lg^d n + k)$ query time to return k points. In the Yggdrasil extension, we can represent ranges with two leaves. An $O(1)$ query on the prefixes of the leaves summarizes the range. Again, for each node, we can store a recursive structure on the points within that range. This gives a $O(d+k)$ query time.

An interesting open problem is to extend these sorts of structures to kd-trees.

5.3 RAMBO data structures — Jonathan Brunsman and Sonia Garg

The project was concerned with reducing memory consumption using the RAMBO architecture. For the predecessor problem, the known constant time solution of Brodnik et al. uses $\Theta(u)$ space; this can be quite bad if n is significantly smaller than u. It was remarked that the RAMBO memory contains many zero bits in this case, and that space can be reduced by not maintaining intervals of zeroes on the same level of the tree. Unfortunately, such a scheme leads to a very unstructured memory and slow updates. Another idea was to improve the space consumption of fusion trees, by not storing zero bits generated by the sketch function. Unfortunately, this only leads to a marginal improvement, and creates significant complications in hardware.

An interesting open problem is to improve the space required by the Brodnik et al. data structure, or prove a lower bound.

6 Applications of Data Structures

6.1 XML-related data structures — David Malan

The XML document format has caused significant excitement and is becoming an important standard for document storage and exchange. Investigating the performance of processing XML documents is therefore a well-motivated problem.

One inefficiency arises from the fact that a document has to be read and processed completely, even though typical queries can be answered by examining only a small portion of the file. It was proposed that a summary structure be added to the beginning of the document to make it possible to read only the interesting portions of the file. The proposed structure was the balanced-parentheses representation of the document we discussed in class, which makes it possible to find the starting position of interesting fields efficiently. Experimental results confirmed that this idea is effective.

Another inefficiency in answering queries using standard approaches is that a lot of edges have to be traversed to get to interesting nodes, even though these nodes are typically cousins on the same level. A "summary DOM" approach was proposed, which mitigated this deficiency by adding level links to the classical representation. Experimental findings supported this idea.

An interesting challenge for research is how to represent the DOM efficiently in a cache-oblivious setting. This becomes important considering the large volume of information usually stored in this format.

6.2 Financial data structures — William Kwong

Detecting statistical arbitrage is an important problem for real-time trading systems. The problem is complicated by the high volume of data, and the fact that the evolution of prices is best described as a continuous rather than discrete signal.

It was proposed that kinetic data structures, normally studied in the context of computational geometry, can be used for real-time trading systems. The idea is to assign a "certificate" to each piece of information that gives an expiration time. Based on an assumed continuity in the prices, the certificates form a guarantee that recent changes do not significantly alter the result computed by the algorithm. Whenever a certificate expires, more recent information can be used to recalculate relevant values. It was argued that this general approach can be used to improve the running time of financial algorithms under normal operating conditions.

6.3 Minimum spanning hypertrees — Percy Liang

This project was about finding the minimum spanning hypertree in a given hypergraph, which has applications to machine learning. Unfortunately, the problem of determining the minimum spanning hypertree is NP-hard. Several approximation and heuristic algorithms are known, and this project focused on the data structure needed for their efficient implementation. This work lead to a generalization of the classic union-find data structure. A good implementation was presented. One contribution of the project was the development of a framework in which new approximation algorithms for the minimum spanning hypertree can be benchmarked.