1 Data Structures

We will work with the cell probe model of data structures. There are two components to the model, a CPU and memory. The memory consists of cells, each of which can hold a $w$-bit word. The number of cells is the space used by the memory. We do not place computational restrictions on the processor and just look at the number of accesses (reads or writes) to the memory needed to accomplish a task.

For the whole lecture we will fix the word size $w$. If the problem size is $n$, it is typical to consider $w = \Theta(\log n)$. The external memory model can be approximated by taking $w = B \log n$ for some large constant $B$, where $B$ represents the size of a page.

Example: Partial sums problem. In this problem we have an array $A[1 \ldots n]$ where $A[i] \in [n]$. We want to implement two operations, an update $A[i] \leftarrow x$ which puts the value of $x$ into $A[i]$ and a partial sum query which returns $\sum_{i=1}^{k} A[i]$. We wish to process the array into a data structure so as to minimize the cost (=number of cell accesses) of the update and partial sum query operations.

This is a dynamic problem and we are concerned with the tradeoff between the update and query times.

By creating a binary tree whose leaves are $A[1], \ldots, A[n]$ and whose internal nodes hold the sum of its children, we can implement both the update and query operations with cost $O(\log n)$. The number of cells used is $2n$ which is near optimal as $n$ cells are needed just to store the data.

Example: 2D range counting. In this problem we have $n$ points in the plane. A query is a rectangle and the answer is the number of points contained in this rectangle. This is a very common desired query in a database, the conjunction of two comparisons: How many employees are between the ages of 50 and 60 and have worked for the company for 25 to 40 years?

This is a static problem and we are concerned with the tradeoff between the query time and space used. Let us pick off a couple points on this curve.

If we have $n$ points, we can assume that our axes are labeled $1 \ldots n$ as we just care about the relative position of the points. If we have $n^2$ cells, then we can achieve constant query time. We create cells labeled by $(x, y) \in [n] \times [n]$ and holding the number of points in the rectangle $[0 \ldots x] \times [0 \ldots y]$. Then if we want to know how many points there are in the rectangle $[x_1 \ldots x_2] \times [y_1 \ldots y_2]$ we can add the results from cells $(x_2, y_2)$ and $(x_1, y_1)$ and subtract cells $(x_1, y_2)$ and $(x_2, y_1)$.
At the other end of the spectrum, we can achieve space $O(n)$ and query time $O(n)$ by simply storing the locations of the points and reading all their positions to answer the query.

In a more interesting point of the tradeoff, we can achieve space $O(n \log n)$ and query time $\log n$. How can this be done?

These examples show two different kinds of settings. In dynamic problems like the partial sums problem we have data that is being updated and we want to maintain a data structure that is able to answer a query. Here we usually fix an amount of space to be used and consider the tradeoff between update and query times. In static problems like 2D range counting, we look at the tradeoff between query time and space.

## 2 Static Problems

We begin with lower bounds for static problems. We will be looking at lower bounds coming from communication complexity. The most naive way such a lower bound would work is to associate the CPU with Alice and the memory with Bob.

Suppose there is a cell-probe algorithm for a problem which uses space $s$ and $t$ many queries. This gives a communication protocol for the same problem with communication $t(w + \log s)$—when the processor asks for the contents of a memory cell, this can be done by Alice sending a message of $\log s$ bits indicating the name of the desired cell, and Bob answers with $w$ bits to describe the contents of the cell.

The main weakness of this approach is the presence of the $\log s$ here. If we show a lower bound of $a$ on the communication problem, we obtain $s \geq 2^{a/t - w}$. This bound can be good for constant query time, but degrades quickly with larger $t$.

As a concrete example, with 2D range counting the algorithm using $O(n^2)$ space and constant queries gives an upper bound of $O(\log n)$ on the communication complexity of this problem in this setup.

**Partial Match**  Let us consider another problem, the partial match problem. Here we have a database of $N$ strings in $\{0, 1\}^d$. A query is a string in $\{0, 1, *\}^d$ and we want to know if there are any strings in the database which match the query (with * matching anything).

Let us see how we can use a cell-probe algorithm for the partial match problem to give a communication protocol for the lopsided disjointness problem (LSD). The parameters we will use for LSD are: universe size $mn$, Bob will have a string with $mn/2$ many ones, and Alice will have a string with $n$ many ones. Moreover, we will assume that Alice’s input is “blocked” i.e. she has exactly one one in each block of size $m$. In other words, we can view Alice’s string as $(i_1, \ldots, i_n)$ where each $i_j \in [m]$ indicates the position of her one in the $j^{th}$ block.

Consider an encoding $C : [m] \to \{0, 1\}^{O(\log m)}$ with the property that $\|C(x)\|_H = \|C(y)\|_H$ for all $x, y \in [m]$. That is, all the encoded strings have the same Hamming weight.

If Alice receives input $(i_1, \ldots, i_n)$ in the LSD problem, we will make a query in the partial match problem the string $C(i_1), \ldots, C(i_n)$, where all ones have been turned to *. The dimension $d$ of this query is $d = O(n \log m)$. 

2
Now we use Bob’s string $y \in \{0, 1\}^{mn}$ to populate the database. We refer to positions in $y$ by pairs $(i, j)$ where $i \in m$ and $j \in n$. For each one in $y$, say in position $(i, j)$, we create an element of the database which has zeros everywhere except in the $i^{th}$ block, where it has $C(j)$.

Now if $x$ and $y$ intersect, then we will find a match—as we changed ones to stars in the query, any zeros in the database strings will be matched. On the other hand if $x$ and $y$ do not intersect, there will not be a match as all codewords have the same Hamming weight there will be a place where the string in the database has a one and the query has a zero.

With this setup in the LSD problem we know that either Alice has to send $\Omega(n \log m)$ bits, or Bob has to send $\Omega(nm^{1-\epsilon})$ many bits.

The communication on Alice’s side is $t \log s$ and on Bob’s side $tw$. This gives us that either $t \log s = \Omega(n \log m) = \Omega(d)$ or $tw = \Omega(nm^{1-\epsilon}) = N^{1-\epsilon}$. The weaker condition is most likely the first one which gives $s = 2^{\Omega(d/t)}$. This means that with constant time, the space required is $2^{\Omega(d)}$. This proof does not give a very good bound, however, for growing time.

2D range counting Let’s go back to the 2D range counting problem. We want to try to show a lower bound for linear space. Remember that the problem before was the fact that Alice’s communication was only logarithmic in $s$. One idea to get around this is to use a direct sum type of argument. That is, we imagine $k$ many processors operating on the database. We get some gain as the amount of communication Alice needs to simulate this does not go up by a factor of $k$—Alice can use $\log \binom{\log s}{k} \approx k \log \log n$.

Now if we want to show a lower bound for space $s = n \log n$, we can set $k = n/\log n$. This means a single message of Alice costs $k \log(s/k) = \Theta(k \log \log n)$. Alice’s input can now be described with $k \log n$ bits, so the best query lower bound we can hope for is $tk \log \log n \geq k \log n \Rightarrow t \geq \log n/\log \log n$.

The reduction goes through the Butterfly graph. The degree $b$ butterfly graph is a graph with $d+1$ layers each with $b^d$ many vertices. The vertices in layer zero are sources and those in level $d$ sinks. If we label the vertices in each layer by strings in $[b^d]$ then there is an edge from $x_i$ in layer $i$ to $x_{i+1}$ in layer $i+1$ if and only if $x_i$ and $x_{i+1}$ agree outside of the $i^{th}$ bit. Notice that there is a unique path from every source vertex to every sink vertex.

Problem X: Given a subgraph of the butterfly graph and a source and sink, is there a path from source to sink?

Let us first see what happens when we remove an edge from the butterfly graph, say between $x_i$ and $x_{i+1}$. The source vertices that could potentially use this edge are those that agree with $x_{i+1}$ on all indices $i+1, \ldots, d$. In other words all source vertices in the interval where the high order bits are fixed by $x_{i+1}$ and all low-order bits (positions $1, \ldots, i$) can be arbitrary. A similar thing happens with the sink vertices, only now those affected differ in have low order bits which agree with $x_{i+1}$ and differ on high order bits. By reversing the labels of the sink vertices, turning high order bits into low order bits, we can again make this an interval. Thus killing an edge affects any sink source pairs that lie in a product sets of two intervals—a rectangle!

This means that given a set of rectangles and a query point, if we can answer if this point
lies in a rectangle, then we can solve Problem X. This problem is known as 2D stabbing. Finally, with (weighted) 2D range counting one can solve 2D stabbing. One replaces a rectangle \([a_1, b_1] \times [a_2, b_2]\) by four points, \((a_1, b_1)\) and \((a_2, b_2)\) with weight +1 and \((a_1, b_2)\) and \((a_2, b_1)\) with weight −1. To test whether \((q_1, q_2)\) lies in a rectangle, one queries the sum in the range \([0, q_1] \times [0, q_2]\). If the point is inside a rectangle, the lower left hand corner contributes +1 to the count. If it is outside, the corners cancel out.

Now we sketch the lower bound for solving Problem X by making \(k\) queries in parallel via a reduction to the lopsided disjointness problem. The basic idea is that Bob will identify the set of ones in his string with removed edges in the butterfly graph. Alice will identify the ones in her string with edges in the butterfly graph forming paths from source to sink. The key desired property is that Alice’s set of edges form edge disjoint paths from source to sink. Intuitively, when this happens each parallel query has “full power” in answering the disjointness problem on a substring of length \(d\), the length of a path from source to sink. For full details see [1].

References