Algorithms Seminar: A Linear-Space Data Structure for Answering Distance Queries for Marked Points on a Grid (draft)

Sam McCauley

11/9/12

1 Introduction

We examine a variant of the clasic nearest neighbor problem, where given a set of k points, we preprocess the points so that we can quickly answer the distance to the nearest neighbor to a query point. We specifically focus on the case where all points—both the k original points as well all query points—are confined to an $n \times n$ integer grid in the plane. Furthermore, we only require the distance to the nearest point to be reported (i.e. not the point itself) in the L_2 metric. We hope to extend this to higher dimensions and different metrics in the future, but we always require integer coordinates for all points. Our result is a data structure that requires $O(N^{1.5})$ preprocessing time and $O(N)$ space that can answer queries in $O(1)$ time.

2 Problem Definition

We are given an $n \times n$ grid of unit squares in the real plane, some of which may be marked with a $*$. We are given time to preprocess the points, and then must answer queries of the form "What is the distance from square (x, y) to the closest marked square?" We assume here that the distance refers to the L_2 metric, but our results can be extended to any distance metric. Our results are given in terms of $N = n^2$, the number of squares in the grid.

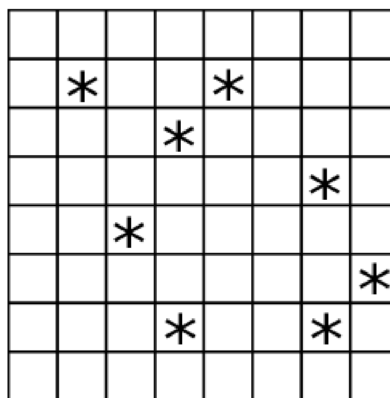

Figure 1: The original problem instance, a set of marked squares in a grid.

3 Naive Solution

We can immediately give a data structure that has:

- $O(N^2)$ preprocessing
- $O(N \log N)$ space
- $O(1)$ query time

To do this, we find the distances from all points to the closest $*$ with a brute-force algorithm, checking all pairwise distances. This takes $O(N^2)$ time. Then, we can store these distances (each of which has size $O(\log n)$) in $O(N \log N)$ space. To query, we can look up the point in our array which takes $O(1)$ time. This is shown in Figure 2.

2	1	2	2	1	2	3	4
1	0	1	1	0	1	2	3
2	1	1	0	1	2	3	4
3	2	1	1	2	3	4	4
2	1	0	1	2	3	4	5
3	2	1	2	3	4	5	6
3	3	2	3	3	4	5	6
4	4	3	4	4	5	5	6

Figure 2: The most basic solution stores all distances.Note that the squares marked with a $*$ are now marked 0 (as the distance to the closest $*$ is zero) and are highlighted in red.

We can improve the preprocessing time of this data structure by slowly "growing" out from each marked point, as in Djikstra's algorithm or a heat map, as follows:

1: **while** All points are not yet marked
2:     **for** $i = 1$ to N
3:         **if** a neighbor of i has a recorded distance **then**
4:             $x \leftarrow$ neighbor of i with nearest neighbor closest to i
5:             Record the nearest neighbor of x in i
6:         **end if**
7:     **end for**
8: **end while**

Clearly, each iteration of the inner **for** loop is $O(N)$. The largest possible distance is $O(\sqrt{2N})$, so the outer **while** loop can only run $O(\sqrt{N})$ times, giving a total preprocessing cost of $O(\sqrt{N})$. The pointers to the nearest neighbors can be replaced with the corresponding distances in $O(N)$ time if desired (the resulting data structure is $O(N \log N)$ space either way). Note that the correctness of this method depends in part on Lemma 1.

4 Getting To Linear Space

To improve the bounds, we first point out that adjacent squares must either have the same distance to the closest $*$, or they may differ by one. This is in particular true for the L_2 metric, but it is in fact implied by the triangle inequality.

Lemma 1. *For any distance metric such that $d(x, x') = 1$ if x and x' are adjacent points on the integer grid, if y and y' are the closest marked points to x and x' respectively, $|d(x, y) - d(x', y')| \leq 1$.*

Proof. If $y = y'$, we have $d(x, y) \leq d(x', y) + d(x', x) = d(x', y) + 1$ by the triangle inequality. Similarly, we can find $d(x', y) \leq d(x, y) + 1$. Since both of these are true simultaneously, we get $|d(x, y) - d(x', y')| \leq 1$.

If $y \neq y'$, without loss of generality, $d(x, y) < d(x, y')$ and $d(x', y') \leq d(x', y)$. [1] Then $d(x, y) < d(x, y') \leq d(x, x') + d(x', y') = 1 + d(x', y')$ by the Triangle Inequality. We also have $d(x', y') \leq d(x', y) \leq d(x, x') + d(x, y) = 1 + d(x, y)$. To satisfy these simultaneously, $|d(x, y) - d(x', y')| \leq 1$. □

Our preprocessing occurs as normal; we grow our solution incrementally in $O(\sqrt{N})$ time. Our solution takes the resulting $O(N \log N)$ space data structure and reduces it to linear space while retaining constant query time.

The first step to reduce the space is to divide the grid into tiles of size $\sqrt{\frac{1}{c} \log n} \times \sqrt{\frac{1}{c} \log n}$. We will assign a constant value to c later. From now on, we consider these smaller tiles, as shown in Figure 3.

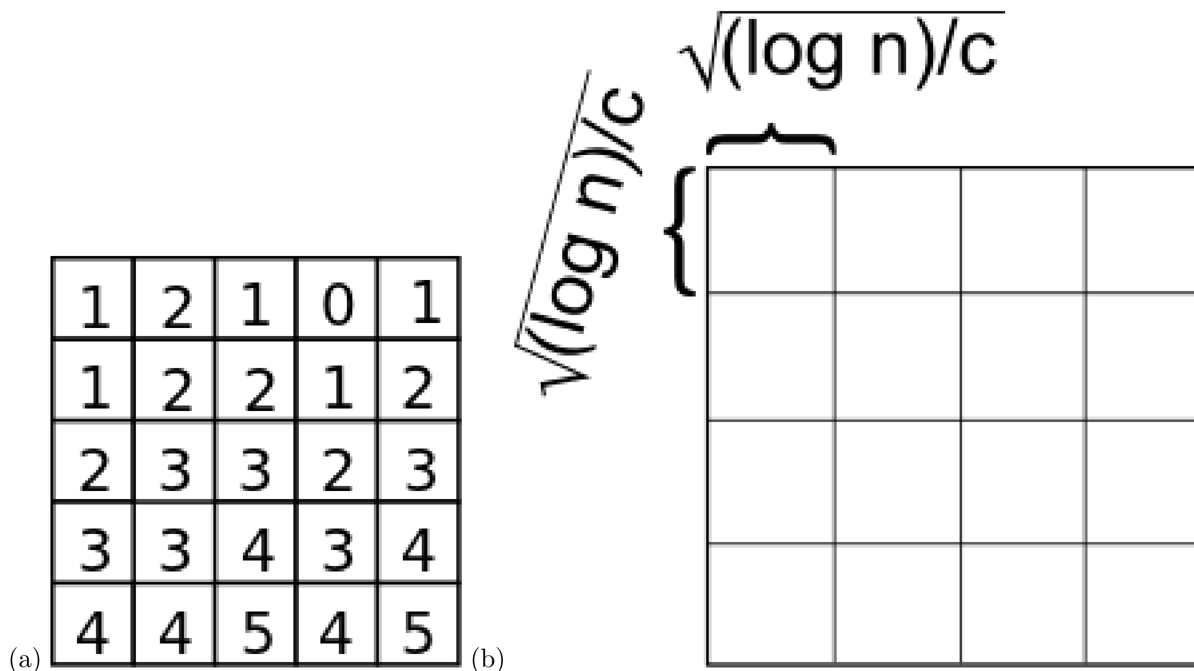

Figure 3: (a) A tile, filled with the distances we found using brute force (b) We divide into tiles of size $\sqrt{\frac{1}{c} \log n} \times \sqrt{\frac{1}{c} \log n}$.

Let's use Lemma 1 to represent this slightly differently. Instead of storing all distances, replace all squares except those on the boundary with arrows pointing to an adjacent grid square that's closer to a $*$. Not all squares have a smaller neighbor—all its neighbors could have the same or greater distance, or it could be that the square itself is marked. We color the arrows red to show that the distance does not decrease when the path is followed. Furthermore, we mark all $*$ on the non-boundary points with a special marker. This is shown in Figure 4(a).

[1] We cannot have equality, as if $d(x, y) = d(x, y')$ and $d(x', y') = d(x, y)$ we can just pick $y = y'$ and we have case 1, so one must not be correct. Without loss of generality we assume it fails for x. We cannot have $\geq$ because we find the minimum point.

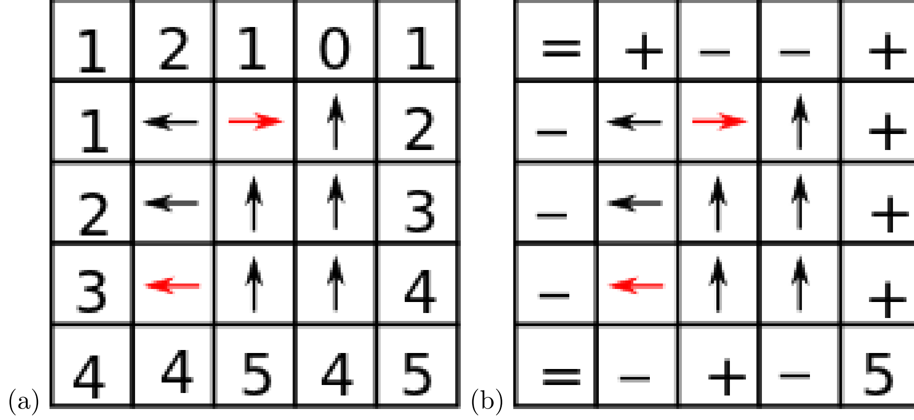

Figure 4: (a) The inner squares are replaced with colored arrows showing the direction we should go to get a better distance (or at least not a worse one); red indicates that the distance does not actually decrease.(b) The boundary points have been replaced with $+$, $-$, or $=$

We can also store the boundary more efficiently, by starting in the bottom-right corner, and marking whether the previous boundary point was larger, smaller, or the same. We go clockwise around the boundary, replacing numbers with $+$, $-$, or $=$. This is shown in Figure 4(b). Note that except for the bottom-right corner, each element of the tile can be stored in a constant number of bits. For inner elements these bits represent the arrow direction and color, as well as whether or not it's a $*$. For outer elements, two bits are enough to denote $+$, $-$, or $=$. We formalize this in Lemma 2:

Lemma 2. *Each element of a tile can be stored in four bits. Therefore, there are only* $4^{\frac{1}{c}\lg n} = n^{\frac{1}{c \lg_4 2}}$ *distinct tiles. If we set* $c = 1/8$*, there are* $O(\sqrt{n})$ *types of tiles.*

Proof. For the inner elements, two bits can represent the direction of the arrow, one can represent the color, and one represents whether or not it's a $*$. For the boundary elements, two bits suffice to denote $+$, $-$, or $=$. Since all tiles can be represented in this way, the number of distinct tiles is bounded by the number of representations, which is given in the lemma. □

Now that we have bounded the number of configurations, we create our $O(n)$ space data structure that supports $O(1)$ queries. First, divide the array into tiles of size $\sqrt{\frac{1}{8}\log n} \times \sqrt{\frac{1}{8}\log n}$. We calculate the arrows as above, but we do not store the arrows themselves. Instead, we only store the actual distance in the bottom-right corner of the tile, and a pointer representing the type of tile (one of the $O(\sqrt{n})$ configurations of arrows, $+$,$-$,$=$, and $*$).

More specifically, this pointer points to an actual array; there will be one array for each type of tile. The array containsan entry for each square in the tile which states the difference between its answer (the distance from this square to the nearest $*$) and that of the bottom-right distance. Note that this set of differences is determined entirely by the arrows, so even though we do not store the arrows, two tiles of the same type will both be able to use this array to answer queries.

There are $O(\sqrt{n})$ arrays, one for each type of tile. Each array has $\frac{1}{c}\log n$ elements, and each entry in the array has a distance which is $O(\log\log n)$ (since the distance between any point and the bottom-right point cannot be more than $\log n$). In total, then, the set of arrays takes $O(\sqrt{n}\log n \log\log n) = o(n)$ space.

We must also store, for each tile, the distance in the lower-right hand corner, as well as a pointer to the tile type. The distance in the right-hand corner requires $O(\log N)$ bits to store (as all distances in an $N \times N$ grid are $O(N)$ under our assumptions), and the pointer to the tile requires $O(\log\sqrt{N})$ bits. There are $cN/\log N$ tiles, so in total, storing these tiles with the distances and pointers requires $O(N)$ space.

In total, this data structure requires $O(N)$ space. Furthermore, we can answer queries in $O(1)$ time. Given a query point, we find which tile it occurs in. We then find the pointer associated with that tile, and

look up the point's position in the tile in the corresponding array. We add the distance at this address and the distance stored as the value of the bottom-right corner of the tile, and we obtain our answer.

5 Future Work

The results we found here have the potential to be extended considerably. First, we have only verified that this works for the limited model of the L_2 metric on a two-dimensional grid. It seems that the data structure can be generalized to relax both assumptions—in particular, by chosing a different c that possibly depends on the dimension d, it seems we can achieve similar bounds for a d-dimensional grid for the L_1, L_2, and L_∞ metrics. It may be possible to generalize it further.

However, these extensions do not appear to require significant differences in the data structure. There are several goals that may require more changes. It may be that it is possible to report not only the distance, but in fact the location of the closest point, while achieving the same bounds. This may require handling the boundaries differently (i.e. something more descriptive than the $+,-,=$ solution), but the rest of the data structure seems capable of handling these queries. We are also interested in getting a data structure that uses sublinear space and answers questions with a bounded error probability.