Computing the Nearest Neighbor Transform Exactly with only Double Precision

David L. Millman*, Steven Love*, Timothy M. Chan†, Jack Snoeyink*

* Computer Science, UNC-Chapel Hill, NC, USA, Email: [dave,slove13,snoeyink]@cs.unc.edu
† Computer Science, U Waterloo, Ont, Canada, Email: tmchan@uwaterloo.ca

***Abstract*—The nearest neighbor transform of a binary image assigns to each pixel the index of the nearest black pixel – it is the discrete analog of the Voronoi diagram. Implementations that compute the transform use numerical calculations to perform geometric tests, so they may produce erroneous results if the calculations require more arithmetic precision than is available. Liotta, Preparata, and Tamassia, in 1999, suggested designing algorithms that not only minimize time and space resources, but also arithmetic precision.**

A simple algorithm using double precision can compute the nearest neighbor transform: compare the squared distances of each pixel to all black pixels, but this is inefficient when many pixels are black. We develop and implement efficient algorithms, computing the nearest neighbor transform of an image in linear time with respect to the number of pixels, while still using only double precision.

***Keywords*-Computational geometry; Image processing; Distance transform; Arithmetic precision; Degree-driven analysis of algorithms**

I. Introduction

The *nearest neighbor transform* of a binary $U \times U$ image assigns to every pixel the index of the closest black pixel under the Euclidean metric; this is a discretized Voronoi diagram of the black pixels [**?**]. The closely related *distance transform* assigns the distance to the closest black pixel. Both transforms have a long history of application in image processing [**?**].

Both can be computed in a straightforward manner, especially with the assistance of graphics hardware [**?**]; these algorithms are often used to generate approximations to the Voronoi diagram. It is a challenge, however, to compute the nearest neighbor transform in time that is bounded by the image size $O(U^2)$ only, and not by the number of black pixels; the algorithms that do this are based on computing the true Voronoi diagram and discretizing to a grid [**?**], [**?**], [**?**]. Computing the true Voronoi diagram, however, requires four times the input precision to guarantee correct, consistent output, and this is usually ignored in work on efficient geometric algorithms.

We observed [**?**] that it would be possible to apply Liotta *et al.*'s "degree-driven analysis of algorithms" [**?**] to derive an algorithm to compute the nearest neighbor transform in double precision. In this paper, we provide the full algorithm and show that our degree 2 algorithm takes expected $O(U^2)$ time and $O(U^2)$ space. We also describe a simpler degree 2 algorithm that takes $O(U^2 \log U)$ time and $O(U^2)$ space but is faster in practice. In addition, we report on experiments with our prototype implementation.

In the next section, we precisely define our task and survey previous work. Then, after exploring predicates in Section **??**, we give algorithm details in Section **??**, and experiments in Section **??**.

II. Preliminaries

We begin by defining some notation for the nearest neighbor transform, reviewing previous work, and transforming the problem to that of computing an envelope of lines.

A. Definitions

We assume that we are given the coordinates of n black pixels, or *sites* from a $U \times U$ grid, $\mathbb{U}$. Denote the sites $S = \{s_1, ..., s_n\}$, where each $s_i = (x_i, y_i)$ with $1 \leq x_i, y_i \leq U$. We will be a little lazy in notation, denoting the coordinates of other pixels $p \in \mathbb{U}$ as $p = (x_p, y_p)$.

We assume throughout this paper that the n sites S are listed in order of increasing x-coordinates, with ties broken by y-coordinates. That is, for all $i < j$, either $x_i < x_j$ or ($x_i = x_j$ and $y_i < y_j$). Note that this also forbids duplication of sites. We can create this ordered list easily from pixels in a given image in $O(U^2)$ time and $O(n)$ additional space, or from an unordered list by counting sort in $O(n + U)$ time and space.

The *nearest neighbor transform* finds the closest site for each pixel, breaking ties by site index.

Problem 1 (NNTrans-min): For each pixel $q \in U^2$, find the site $s_i \in S$ such that, for all $j < i$, the distance $\|q - s_i\| < \|q - s_j\|$, and for all $j > i$, the distance $\|q - s_i\| \leq \|q - s_j\|$.

B. Previous work

Many researchers have developed algorithms for computing the nearest neighbor transform on various architectures, including serial and parallel CPUs and GPUs. However, the arithmetic precision of the algorithms are often overlooked, even though all algorithms need exact arithmetic to guarantee a correct output.

Breu *et al.* [?] proposed the first linear time algorithm, which uses divide and conquer to compute and discretize a 2-d Voronoi diagram computation. If you analyze the arithmetic calculations of their geometric predicates, you find that their algorithm requires five times the input precision.

Both Chan and Maurer *et al.* proposed dimension reduction algorithms [?], [?], [?] that compute the nearest neighbor transform from one-dimensional Voronoi diagrams for each column and one-dimensional weighted Voronoi diagrams for each row. Their calculations can be modified to require only three times the input precision. By changing parts of the algorithm, we will see how to reduce that to double precision without affecting the asymptotic $O(U^2)$ bound on the expected running time.

Hoff and others [?] presented the earliest work on using the GPU to compute the nearest neighbor transform. Their algorithm is dependent on the number of black pixels in addition to the size of the image, and precision is determined by the resolution of the Z-buffer. Approximate GPU algorithms have been proposed proposed [?], [?], [?], but these cannot guarantee an exact result. Cao *et al.* [?] adapted Maurer's algorithm to the GPU, focusing on efficient data structures that take advantage of the memory and the multi-threaded processing power of the GPU. Cao's implementation used triple precision, just like Maurer's.

Liotta, Preparata, and Tamassia [?] suggested that since most geometric predicates are based on the sign of a polynomial, we can analyze the precision of an algorithm by the degree of the polynomials of its predicates. If we assume that our input coordinates are scaled to b-bit integers, and the coefficients of the polynomials are small (which is common for geometric predicates), the sign of a degree d polynomial can be evaluated in $O(db)$ bits (carries from summing can be handled with a compensated sum [?]). By analyzing precision, we can determine if for a given input size, the computer has enough arithmetic bits to carry out intermediate calculations without truncation or rounding errors. Thus, their "degree-driven design of geometric algorithms" [?] incorporates reducing the number of arithmetic bits as an algorithm design criterion. The resulting algorithm can be implemented on any hardware that provides the needed precision, or additonal predicate evaluation schemes such as [?], [?], [?], [?] can be used to further reduce the number of bits required.

As a shorthand to calculate degrees, we frequently just replace polynomial expressions with a circled number indicating their degree. Let us see an example of precision analysis. Given the six coordinates of a query pixel, q, and two pixel sites, s_i and s_j, we can determine the closer site to q by comparing squared distances, which is a degree 2 polynomial in the input coordinates. As a function of q, we could write

$$\begin{aligned} f_{i,j}(q) &= \text{sign}\left(\ \|q-s_i\|^2 - \|q-s_j\|^2\ \right) \\ &= \text{sign}\left(\ x_i^2 + y_i^2 - 2(x_i - x_j)x_q \right. \\ &\qquad \left. -2(y_i - y_j)y_q - x_j^2 - y_j^2\ \right) \\ &= \text{sign}\left(\ ② + ①x_q + ①y_q\ \right). \end{aligned}$$

Let's name this predicate.

Observation 2: Given two sites, s_i and s_j and a query point q with $s_i, s_j, q \in \mathbb{U}$, the predicate $\texttt{Closer}(s_i, s_j, q)$ is degree 2.

In special geometric configurations this may simplify further. For example, if the sites are on the same vertical line so $x_i = x_j$, then there is a factor of $(y_i - y_j)$, so the computation need only determine the signs of two linear terms. In general, however, it is irreducible of degree 2. For more on irreducibility proofs see [?].

Observation 3: Given two sites s_i and s_j on the same vertical line and a query point q with $s_i, s_j, q \in \mathbb{U}$ the predicate $\texttt{Closer1D}(s_i, s_j, q)$ is degree 1.

C. Problem transformations

We can use common transformations [?], [?] to turn the NNTrans-min problem into a series of problems on lines. First, we can work with squared distances when comparing distances from q to sites s_i and s_j, using any of the following equivalent inequalities for $j < i$: (For $i < j$, replace all strict inequalities, so $>$ becomes $\geq$ in the third line below.)

$$\begin{gathered} \|q-s_i\|^2 < \|q-s_j\|^2 \\ q\cdot q - 2q\cdot s_i + s_i\cdot s_i < q\cdot q - 2q\cdot s_j + s_j\cdot s_j \\ 2x_i x_q + 2y_i y_q - x_i^2 - y_i^2 > 2x_j x_q + 2y_j y_q - x_j^2 - y_j^2. \end{gathered}$$

Thus, we can convert the NNTrans-min problem to an equivalent:

Problem 4 (NNTrans-max): For each pixel q, find the site with lowest index $s_i \in S$ that achieves the maximum of $2x_i x_q + 2y_i y_q - x_i^2 - y_i^2$.

NNTrans-max can be solved one row at a time, labeling each pixel along the row $y = Y$ with the index of its closest site. We transform the problem further for the given Y: Let $L_Y(s_i)$ map site s_i to the line $\ell_i^Y : y = a_i x + b_i$ where the slope $a_i = 2x_i = ①$ and intercept $b_i = 2y_i Y - x_i^2 - y_i^2 = ②$. Let $L_Y = L_Y(S)$ denote the set of all lines obtained from sites S. Note that since S is sorted by x with ties broken by y, L_Y is sorted by slope, with ties broken by y-intercept.

Along a fixed row $y = Y$ of the grid, we can phrase the NNTtrans-max problem as determining the highest line at each pixel – the *discrete upper envelope (DUE)* of the lines.

Problem 5 (DUE-Y): For a fixed $1 \leq Y \leq U$, and for each $1 \leq X \leq U$, find the smallest index of a line of L_Y with maximum y coordinate at $x = X$.

The key to computing the Nearest Neighbor transform in $O(U^2)$ time and degree 2 will be solving DUE-Y in $O(U)$

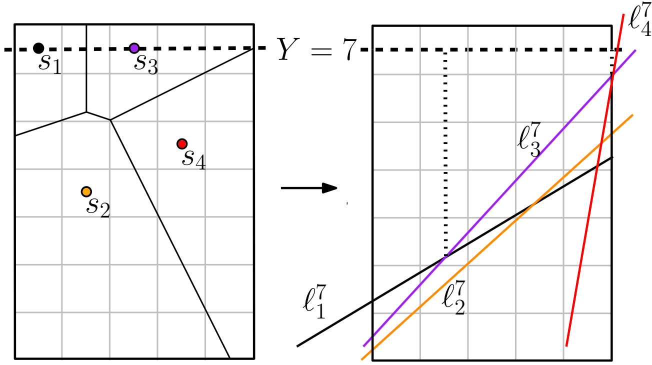

Figure 1. Example transformation from S to $L_Y(S)$ for $Y = 7$. *Left*: sites s_1 through s_4 are shown. *Right*: Each site s_i is transformed to the line ℓ_i^7. Observe that when ℓ_i^7 is the highest line at $x = X$, site s_i is the site with minimal distance to pixel at $(X, 7)$

time and degree 2. As shown in Figure **??**, when ℓ_i^Y is the highest line at $x = X$, site s_i is the site with minimal distance to pixel at (X, Y). Observe that each line in L_Y has the form $y = \textcircled{1}x + \textcircled{2}$. In Section **??** we show how to compute the discrete upper envelope in $O(U)$ expected time with degree 2 and in Section **??** we use this construction to solve NNTrans-max in $O(U^2)$ expected time and degree 2.

D. *Geometric primitives*

Before diving into algorithms, we present some geometric primitives (predicates and constructions) and their precision. Most of these primitives are part of our algorithm; the few that are not are tempting options that we avoid because of their high precision requirement.

A common test in geometric algorithms is determining the orientation of three points, a, b, and c, by evaluating the sign of a determinate whose entries are the homogeneous coordinates of the points. The $\texttt{Orientation}(a, b, c)$ predicate is degree 2 on points with degree 1 coordinates. When the points have coordinate values of $(\textcircled{1}, \textcircled{2})$ (which is the form our points will take after transformations), the predicate has higher degree:

$$\begin{vmatrix} 1 & a_x & a_y \\ 1 & b_x & c_y \\ 1 & c_x & c_y \end{vmatrix} = \begin{vmatrix} \textcircled{0} & \textcircled{1} & \textcircled{2} \\ \textcircled{0} & \textcircled{1} & \textcircled{2} \\ \textcircled{0} & \textcircled{1} & \textcircled{2} \end{vmatrix} = \begin{vmatrix} \textcircled{1} - \textcircled{1} & \textcircled{2} - \textcircled{2} \\ \textcircled{1} - \textcircled{1} & \textcircled{2} - \textcircled{2} \end{vmatrix}$$
$$= \begin{vmatrix} \textcircled{1} & \textcircled{2} \\ \textcircled{1} & \textcircled{2} \end{vmatrix} = \textcircled{1}\textcircled{2} - \textcircled{1}\textcircled{2} = \textcircled{3}$$

Observation 6: Given three points a, b, and c with coordinate precision $(\textcircled{1}, \textcircled{2})$ the predicate $\texttt{Orientation}(a, b, c)$ is degree 3.

Consider two lines, $\ell : y = \ell_m x + \ell_b$ and $h : y = h_m x + h_b$, each with degree 1 slope and degree 2 y-intercept, i.e. $y = \textcircled{1}x + \textcircled{2}$. The construction $\texttt{Intersect}(\ell, h)$ returns the coordinates of the intersection of the two lines q, which are:

$$q_x = \frac{h_b - \ell_b}{\ell_m - h_m} = \frac{\textcircled{2} - \textcircled{2}}{\textcircled{1} - \textcircled{1}} = \frac{\textcircled{2}}{\textcircled{1}}$$
$$q_y = \frac{\ell_m h_b - \ell_b h_m}{\ell_m - h_m} = \frac{\textcircled{2}\textcircled{1} - \textcircled{1}\textcircled{2}}{\textcircled{1} - \textcircled{1}} = \frac{\textcircled{3}}{\textcircled{1}}$$

Observation 7: Given two lines ℓ and h of the form $y = \textcircled{1}x + \textcircled{2}$, the $\texttt{Intersect}(\ell, h)$ construction produces a point with rational x- and y-coordinates of $(\textcircled{2}/\textcircled{1}, \textcircled{3}/\textcircled{1})$.

As our goal is a degree 2 algorithm, the high degree primitives of Observations **??** and **??** are undesirable. Next, we describe the predicates that provide just enough information to compute a nearest neighbor transform with degree 2.

Given two points $a, b \in \mathbb{U}$, the predicate $\texttt{Before}(a, b)$ returns true if a is lexicographically before b. That is, if $a_x < b_x$ (or $a_x = b_x$ and $a_y < b_y$). This test compares degree 1 values only, therefore the predicate is degree 1.

Observation 8: Given two points $a, b \in \mathbb{U}$, the predicate $\texttt{Before}(a, b)$ is degree 1.

Given two lines, ℓ_1 and ℓ_2, of the form $y = \textcircled{1}x + \textcircled{2}$, and a degree 1 vertical line h defined by points satisfying the equation $x - x_0 = 0$, the predicate $\texttt{Above}(\ell_1, \ell_2, h)$ returns true if ℓ_1 is above ℓ_2 on the line h. We evaluate this predicate by plugging x_0 into the slope intercept form of the lines and comparing the result. Thus, we compare the evaluation of two degree 2 polynomials.

Observation 9: Given two lines ℓ_1 and ℓ_2 of the form $y = \textcircled{1}x + \textcircled{2}$ and a degree 1 vertical line h, the predicate $\texttt{Above}(\ell_1, \ell_2, h)$ is degree 2.

We can use the $\texttt{Above}$ predicate in a binary search for the vertical slab in which the ordering of two lines swap. When the vertical slab is a column width, we can return the column containing the lines intersection. Thus,

Lemma 10: Given two lines ℓ_1 and ℓ_2 of the form $y = \textcircled{1}x + \textcircled{2}$, the construction $\texttt{IntersectCol}(\ell_1, \ell_2)$ returns the column containing the intersection of the two lines in $O(\log U)$ time and degree 2.

III. Algorithm Description

We show first how to compute the discrete upper envelope of lines with double precision, then how to use this to compute the nearest neighbor transform.

A. *Discrete Upper Envelope*

Consider the discrete upper envelope (DUE) of a set of lines, which we represent as a minimal length sequence of tuples (s, i, j) where s is the highest line at x for all integers between i and j, and ties in 'highest' are broken by lowest index. Each tuple (s, i, j) determines a cell consisting of all the points on or above s at $i \leq x \leq j$.

For any set of lines with the same slope, only the line with largest y-intercept appears on the upper envelope. Thus, in $O(n)$ time we can throw away any line that shares a slope

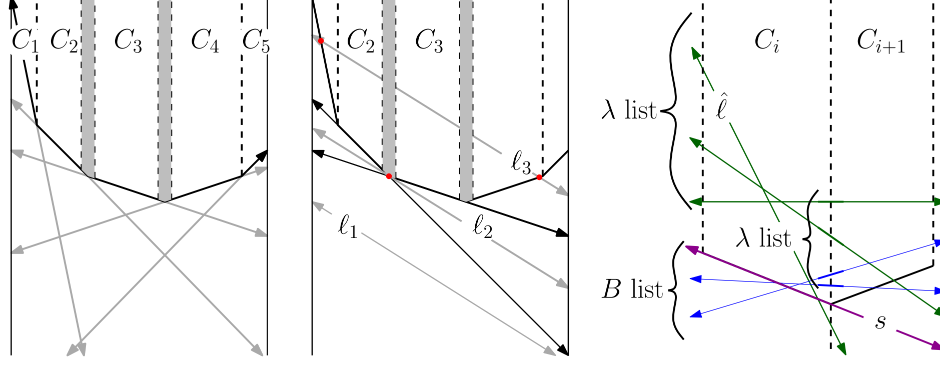

Figure 2. A DUE of m lines L is created by adding a random sample R of $m/2$ lines to the DUE of $L \setminus R$. *Left*: The DUE contains cells C_1 through C_5 defined by lines shown in gray. Each cell is defined by (s, i, j), and can be visualized as the trapezoidal area above line s, between $x = i$, and $x = j$. Sometimes boundaries do not match up and we get a gap between two cells, for example, the gray gap between C_2 and C_3. *Center*: Three examples of identifying the first cell intersected by identically sloped lines ℓ_i. Thanks to the slope ordering of an upper envelope, each ℓ_i would intersect cells C_2 or C_3 if it intersects any cells. Since we are computing the DUE, cells that would be completely inside a grid column, for example a cell formed by ℓ_2, are dropped. As ℓ_3 intersects C_2 and C_3 we test cells to the left until finding the leftmost intersected cell. That cell then adds ℓ_3 to its B list. *Right*: Each cell C_i maintains three lists of lines: s, colored magenta, defines C_i; B, colored blue, are the lines intersecting C_i from the bottom; and λ, colored green, are the lines intersecting C_i from the left. We find cell C_{i+1}'s λ list by throwing away any line of $\ell \in \lambda \cup B$ that is below the defining line of C_{i+1} at its leftmost gridpoint, for example $\hat{\ell}$.

but has smaller y-intercept value. We are left with a set L of $m = O(U)$ lines; for the remainder, we compute the discrete upper envelope of these lines.

We briefly sketch three algorithms for computing discrete upper envelopes, which, in increasing implementation complexity, are: `D3-DUE` is an $O(U)$ algorithm using a degree 3 `Orientation` predicate; `UlgU-DUE` replaces the predicate with `IntersectCol` to reduce to degree 2 at the cost of $\log U$ time for binary search; `U-DUE` achieves $O(U)$ expected time and degree 2 by randomization. We test all three algorithms in our experiments.

First, we sketch the `D3-DUE` procedure, which takes L and produces the upper envelope of L. It is known that computing the upper envelope of a set of lines is equivalent to computing the lower hull of a set of points sorted by x under a transformation that maps each line $y = ax + b$ to point $(a, -b)$ [**?**] . Linear time algorithms for computing the lower hull of a set of points sorted by x use the `Orientation` predicate [**?**]. For our input, the coefficients a and b are degree 1 and 2, respectively, so `Orientation` is degree 3 by Observation **??**; any procedure that uses `Orientation` is at least degree 3. Thus, `D3-DUE` takes $O(U)$ time and degree 3.

Second, the `UlgU-DUE` procedure also takes $L = \{\ell_1, \ldots, \ell_m\}$, sorted by slope, and produces the discrete upper envelope directly from the lines. Even here we have an obstacle; in Observation **??** we saw that computing intersection points requires too much precision. Fortunately, for the discrete upper envelope it is enough to identify the grid column containing the intersection of a pair of lines, which is degree 2 and $O(\log U)$ time by Lemma **??**. We use this as follows.

We maintain a stack to represent the prefix of the DUE computed so far, much like a convex hull algorithm. Instead of using an orientation test to decide to pop the stack on the insertion of a new line ℓ_k, we use the `Above` test to pop all cells (s, i, j) where ℓ_k is above s at i and at j. Let line ℓ be the line defining the last cell of the DUE, we find the beginning column of the cell for ℓ_k with `IntersectCol`(ℓ_k, ℓ). The procedure `UlgU-DUE` uses only the degree 2 predicates `Above` and `IntersectCol` and takes $O(U \log U)$ time.

Third, the `U-DUE` procedure solves the DUE-Y problem in expected $O(U)$ time and degree 2 by random sampling and one-sided recursion. We use a stable random partitioning of the set L of m lines to create a set R of $m/2$ lines that maintains slope ordering. In order to find the DUE of all m lines, we (recursively) find Q, the DUE of the lines not in R, and then 'add' the lines of R into Q.

Here is an overview of the input, output, and invariants for the 'adding' procedure in `U-DUE`, depicted in Figure **??**:

Input: The sampled lines R, ordered by slope with no duplicate slopes, and
Q: the DUE of rest of the lines, represented as an array of cells in order of increasing x: Each cell stores its defining line and leftmost gridpoint; the sentinel leftmost cell has left end at $-\infty$.

Output: The DUE of all the lines, in the same representation as an array of cells as the input.

Data Structure: Each cell is given the head of a slope-sorted linked list of lines that "start in the cell;" each line of R is put in the list for the first cell it intersects (by increasing x).

Processing: The new DUE is built by processing the old cells in increasing x, i.e. from left to right.

Invariant 1: Three slope-sorted lists of lines contribute to

the DUE inside a cell: s, the single line defining the cell; B, the list of lines that start in the cell; and λ, the list of lines that cross over from the previous cell.

Invariant 2: The algorithm uses a stack to maintain the output DUE of the set of lines from the lists of Invariant 1 up to a chosen slope.

We initialize and maintain invariants as follows.

First, we identify, for each line ℓ of R, the cell that ℓ starts in. We do so in two sub-steps: first we identify a cell intersected by ℓ (if one should exist). Since R and the lines of an upper envelope are ordered by slope, we can do this by merging Q and R in $O(m)$ time. Next, we walk through cells to the left to find the first cell intersected by each line ℓ. We bound the expected time of this operation later. We do these sub-steps for the lines of R in reverse order, inserting lines at the head of the linked list associated with the cell, so that the lines starting in each cell remain in order of increasing slope.

Next we process cells from left to right. To begin, any lines that start in the sentinel are checked to see if they continue into the first finite cell. Those that do not are discarded. This gives the three lists for the first cell.

In processing a cell, we maintain a stack to represent the DUE, as in `UlgU-DUE`. The only difference is that the binary searches of `IntersectCol` is over the boundary of the cell (not the entire grid).

Lemma 11: Given a set S of n lines sorted by slope of the form $y = ①x + ②$, we can compute the discrete upper envelope of S in $O(U + n)$ expected time.

Proof: As mentioned above, in $O(n)$ we produce L of size m, where each lines has a unique slope. We can upper bound the processing for a cell of width U_i that is intersected by m_i lines as $cm_i \log U_i$, where c is a constant. Clarkson's results on random sampling [**?**, Theorem 3.7] say that $E[\sum_i m_i^2] = O(m)$, so each cell intersects not too many lines on average. Using the Cauchy-Schwartz inequality, and the fact that $\sum_i U_i \le U$, we can upper bound the total cost of processing all cells:

$$E\left[\sum_i cm_i \log U_i\right] \le c\sqrt{\mathrm{E}\left[\sum_i m_i^2\right] \cdot \sum_i \log^2 U_i} \le c\sqrt{O(m) \cdot m \log^2 \frac{U}{m}} = O\left(m \log \frac{U}{m}\right).$$

We build DUEs recursively on random samples of half the lines, so the total expected time is bounded by the following recurrence:

$$T(m,U) = T\left(\frac{m}{2}, U\right) + O(m) + O\left(\mathrm{E}\left[\sum_i m_i \log U_i\right]\right) = T\left(\frac{m}{2}, U\right) + O\left(m \log\left(\frac{U}{m}\right)\right) \implies$$

$$T(m,U) = O\left(m \log\left(\frac{U}{m}\right)\right)$$

Since $m \le U$, we have $T(m,U) = O(U)$. Thus, in total it takes $O(n+U)$ time in expectation to compute the discrete upper envelope. ■

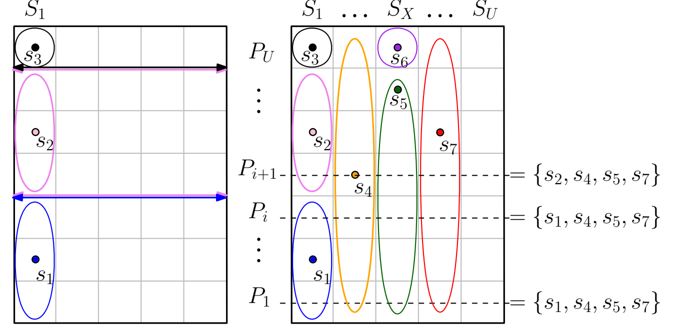

Figure 3. *Left*: Ovals drawn around sites depicting pixels to which they are nearest neighbor. The Voronoi diagram of sites in the same column S_X are horizontal lines. Thus, all pixels in row Y are always closer to the same site in S_X than any other sites in S_X. So, only one site per column can ever be the nearest neighbor to a pixel, and vertical comparisons suffice to find out which one is nearest$_X(Y)$, leaving us with only $O(U)$ sites to consider for each row. *Right*: Illustration of *possible lists* and their generation. Ovals show vertical nearest neighbors. After determining which sites are the highest in their column, (s_3, s_4, s_6, and s_7), we only need to perform a few tests for each site $s_i \in P_i$ to generate P_{i+1}. Each site s_i exhibits one of three cases. Case 1: s_i is the highest in its column, so it is in P_{i+1}. Case 2: s_i is still nearest$_X(Y)$. Case 3: s_{i+1} is now nearest$_X(Y)$ so it replaces s_i in P_{i+1}.

B. NNTrans Algorithm

Assume that we are given a set of n sites $S = \{s_1, \ldots, s_n\}$ on a $U \times U$ pixel grid. We compute the Nearest Neighbor transform of S by using DUE-Y in each row in order, which takes in $O(U^2)$ expected time, $O(U^2)$ space and degree 2. Our algorithm has the following three steps.

In the first step, if S is not already sorted, then we sort the sites by increasing x-coordinate (breaking ties by y-coordinate) using counting sort by y and then by x. For convenience, we assume that the sites of S are labeled in their sorted order.

Let S_X denote the set of sites of S with given x-coordinate value X – sites in a given grid column, as seen in Figure **??**. Each S_X is a contiguous sublist of S, thanks to the sorting. The Voronoi diagram of just the sites in S_X is formed by the horizontal lines bisecting adjacent sites in S_X.

If we now consider a given grid row $y = Y$, then from each non-empty S_X at most one site can possibly

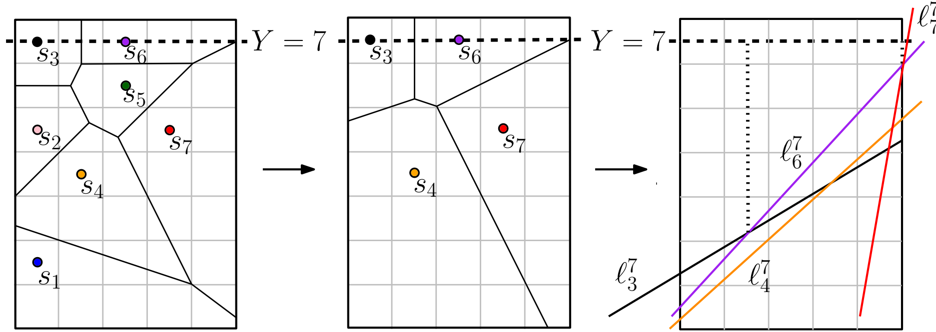

Figure 4. The three steps of the NNTrans algorithm: (Steps one and two show the Voronoi diagram of the sites to illustrate that it does not change along Y when sites are removed.) *Step one*: Sites are sorted by x-coordinate breaking ties by y-coordinate. *Step two*: For each row Y, sites are reduced to the possible list P_Y. *Step three*: Each site $s_i \in P_Y$ is transformed to the line ℓ_i^Y and we compute the DUE of the lines.

contribute to the Voronoi diagram – for non-empty S_X, let $\text{nearest}_X(Y)$ = the site of S_X nearest to (X, Y), breaking ties by lowest index. The *possible list*, $P_Y = \{\text{nearest}_X(Y) \mid \forall 1 \leq x \leq U\}$, contains the at most U sites that can be nearest neighbors of a pixel in row $y = Y$.

In the second step, depicted in Figure **??**, we produce a data structure, which we call the *possible list generator*, that iterates from $Y = 1, \ldots, U$ producing each P_Y. This data structure simply maintains a pointer to the last site used in each ordered list S_X. These pointers advance as Y crosses the bisector between adjacent elements in each list S_X, generating all possible lists P_Y in sequence.

In the third step, we transform each possible list into lines, and compute the discrete upper envelope of the lines $L_Y(P_Y)$ from Problem **??** by Lemma **??**. Each DUE-Y constructs one row of the Nearest Neighbor transform.

Theorem 12: Given n sites on a $U \times U$ pixel grid, we can compute the Nearest Neighbor transform of the sites in $O(U^2)$ expected time, $O(U^2)$ space, and degree 2.

Proof: The procedure to compute the Nearest Neighbor transform of the sites has three steps, shown in Figure **??**.

The first step sorts the n sites with two counting sort in two passes over S and makes degree 1 comparisons of coordinates. Thus, the step uses $O(U)$ space, $O(n)$ time and degree 1.

The second step constructs the possible list generator. Each possible list contains at most U sites. Thus, the size of the data structure is $O(U)$. Since S is sorted we can initialize P_1 in $O(U)$ time by identifying adjacent sites with different x-coordinates, which is degree 1. To update from P_Y to P_{Y+1}, we perform at most U evaluations of the `Closer1D` predicate, which by Observation **??** is degree 1. Thus, initializing the possible list generator takes $O(U)$ time, $O(U)$ space and degree 1 and generating P_{Y+1} from P_Y takes $O(U)$ time and degree 1.

The third step generates the possible list P_Y for each $Y = 1, \ldots U$. The sites are transformed into lines $L_Y(P_Y)$ sorted by slope. Each line in L_Y has the form ①x+② and we compute the discrete upper envelope of L_Y. Since L_Y has at most U lines, by Lemma **??**, processing each Y takes $O(U)$ expected time, $O(U)$ space and degree 2. Therefore, processing all possible sites takes $O(U^2)$ expected time, $O(U^2)$ space and degree 2. ■

In the proof above we use the procedure `U-DUE` to compute the nearest neighbor transform in $O(U^2)$ time and degree 2. However, we could have instead used `UlogU-DUE` to get an $O(U^2 \log U)$ degree 2 algorithm or `Deg3-DUE` to get an $O(U^2)$ degree 3 algorithm. In the next section we will compare the experimental running times of implementations of the three algorithms and MATLAB's implementation of Maurer's algorithm.

IV. Experiments

In this section we investigate robustness and timings of four implementations of algorithms that compute the nearest neighbor transform of an image. We will see that the theoretical advantage of removing a $\log U$ factor is outweighed by the cost of generating and maintaining random samples. On the positive side, however, our degree 2 algorithms, which are guaranteed correct in double precision, are notably faster than the implementation of Maurer's algorithm used to perform MATLAB's `bwdist` function.

Here are the four implementations:

- `Usq` is our implementation of the expected $O(U^2)$ time and degree 2 algorithm, discussed in Section **??**;
- `UsqLgU` is our implementation of the $O(U^2 \log U)$ time and degree 2 algorithm, discussed at the end of Section **??**;
- `Deg3` is our implementation of the (U^2) and degree 3 algorithm, also discussed at the end of Section **??**; and
- `Maurer` is the MATLAB `bwdist` function, which is a compiled implementation of Maurer's algorithm (after version of 2009a).

Recall that `Usq` and `UsqLgU` are degree 2 and `Deg3` and Maurer's algorithm are degree 3.

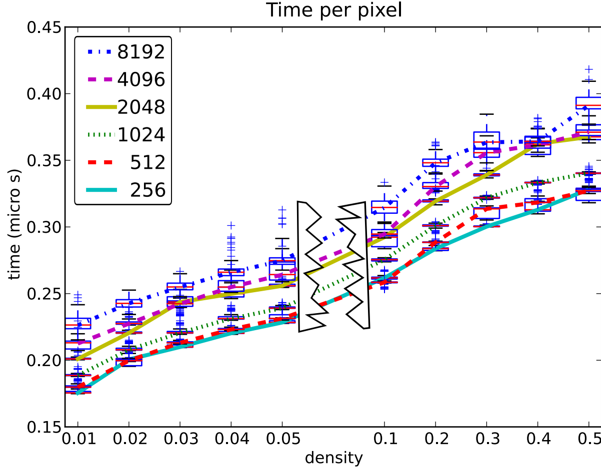

Figure 5. Time-per-pixel box plots of 100 runs of `Usq` on densities between .01 and .5 and grid sizes between 256 and 8192.

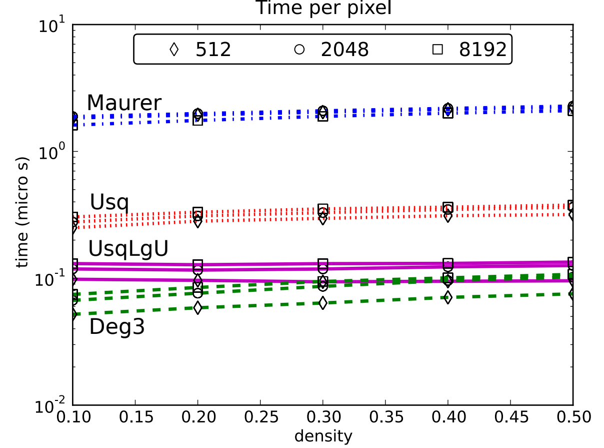

Figure 6. Time-per-pixel on a logarithmic scale for four implementations on grid sizes ranging from 512 to 8192 with densities ranging from .1 to .5.

Timings were taken on a 3.2GHz Intel Xeon processor with 12GB RAM running Ubuntu 10.04. Our implementations were were written in C++, compiled with gcc toolchain and `bwdist` was run using MATLAB 7.0.10 (R2010a).

In our timing experiments we run the four implementations on varying grid sizes and site densities. The tuple of an algorithm A, grid size U, and density δ form a *run*. For each run, we generate a set of sites S by selecting if a pixel contains a site uniformly at random with probability δ, then we execute A on input S for enough iterations so that it takes over one second and report the average run length of an iteration of the time of the run. For runs on algorithm `Usq`, the random number generator is reset to a fixed seed before each iteration to maintain the same execution path.

Our first experiment investigates timings and variance of `Usq`. The implementation was run on increasing grid sizes $\{265, 512, \ldots, 8192\}$, and densities $\{.01, .02, \ldots, .05\}$, and $\{.1, .2 \ldots, .5\}$. In Figure **??**, we see a box and whiskers plot of the time-per-pixel for 100 runs of different seeds and the medians of runs on the same grid size are connected with lines. We plot time-per-pixel as it allow detail to be seen at both small and large grid sizes.

The first observation about Figure **??** is that the boxes are small, about 2 microsecond width on average, indicating that `Usq`'s expected $O(U^2)$ running time comes with a small variance. This is not surprising, as `Usq` makes U calls to a randomized algorithm with expected $O(U)$ time, which smooths out the variance.

One might expect that since `Usq` has an expected $O(U^2)$ running time, the time-per-pixel should be constant across varying grid sizes and densities. In fact, one would even expect higher per-pixel-times for smaller grid sizes since overheads are amortized over fewer pixels.

First, to explain why the per-pixel times increase with the grid size, we fixed the density, varied the grid size, and used Valgrind [**?**] to capture the number of calls to each line. The per-pixel line counts were equal (or slightly higher for smaller grid sizes, as expected). Thus, we believe that the time increase on larger problems can be attributed to different memory usage patterns, e.g., at data sizes that no longer fit into L1 or L2 cache.

Second, to explain why per-pixel times increase with higher densities, recall that each discrete upper envelope calculation produces a run length encoding of the nearest neighbor transform along each horizontal line and that the sites are uniformly distributed. At density δ, we would expect that a row would intersect $\sqrt{\delta U^2}$ Voronoi cells, for a total output size of $\delta^{\frac{1}{2}} U^2$. The experimental times are consistent with a large fraction of the running time being proportional to the output size.

Our second experiment compares the per-pixel time of the four implementations on the three grid sizes, $\{512, 2048, 8192\}$ and the five densities $\{0.1, \ldots, 0.5\}$. In Figure **??**, we see a plot of density verses time-per-pixel. Each algorithm is shown as a different line style and the markers correspond to grid size. For every run, `Maurer` is the slowest followed by `Usq` (5–7× faster than `Maurer`), `UsqLgU`(15–23× faster than `Maurer`), and `Deg3` (20–33× faster than `Maurer`). First, notice that even though the computational complexity of `Usq` is less than `UsqLgU`, the clock time of `UsqLgU` is faster than `Usq` (by about 3×). The unexpected speed difference is because $\log U$ is not large, and `UsqLgU` maintains almost no data structure while `Usq` keeps track of a lot of data structures, and random number generation is slow. Next it is interesting to note that even though `Maurer` is degree 3 it is still slower than the degree 2 algorithms in `Usq` and `UsqLgU`. Finally, while the `Deg3` is the fastest implementation (at about 1.5× faster than `UsqLgU`) it uses degree 3.

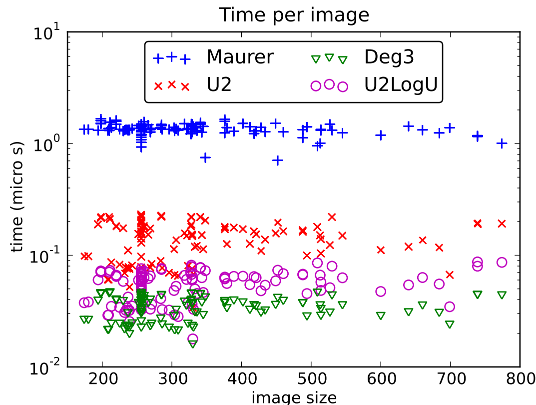

Figure 7. Time-per-pixel on a logarithmic scale for four implementations on boundaries extracted from 120 images from the MPEG7 data set.

Our third experiment compares per-pixel time of the four implementations on 120 extracted boundaries of images from the MPEG7 CE Shape-1 Part B data set. We used the first 20 images from the fish, frog, lizard, lmfish, octopus, and turtle sets, which range in size from 174^2 to 774^2 pixels. Since the boundaries are closed curves or a small number closed curves, an image of U^2 pixels has $O(U)$ sites. In Figure **??**, we see a scatter plot of image size verses time-per-pixel. Each algorithm is depicted with a different marker. As is expected, the per-pixel times of the all four algorithms are similar to the previous experiment, even though the grid sizes are smaller and site densities are substantially reduced.

V. Conclusions

We presented a $O(U^2)$ expected time degree 2 algorithm for computing the Nearest Neighbor transform of a $U \times U$ pixel image. The key step was in computing the discrete upper envelope of at most U lines in $O(U)$ expected time and degree 2. It seems possible that the discrete upper envelope could be computed deterministically by divide-and-conquer with more programming complexity, however, any non-randomized degree 2 algorithm that we attempted had an extra $\log U$ factor. Can the $\log U$ factor be removed, while still using degree 2, without randomization? If it could not, it would be interesting.

We also wonder what other problems are accessible when using degree-driven algorithm design? The algorithms presented in this paper generalize to higher dimensions, but can we create low degree algorithms for the nearest neighbor transform in other norms? L_1 and L_∞ are particularly interesting, as they each have a degree 1 distance comparison.

Acknowledgment

The authors would like to acknowledge their funding sources: Dr. Chan is partially supported by NSERC, Dr. Snoeyink is partially supported by an NSF research grant, and Mr. Love by an NSF REU supplement.