The Fast Euclidean Distance Transform

Robotics 102: Introduction to AI & Programming

In lecture, we learned a fast algorithm for computing the Manhattan distance transform of a binary image. The Euclidean distance is a better measure of how far away an obstacle is from the robot at any cell, since our omnidrive robots can move in any direction. These notes cover a fast algorithm to calculate the distance transform using Euclidean distance. This algorithm was published in a paper by Felzenswalb et al. [1] in 2012.

1 1D Euclidean Distance Transform

The trick of the fast Euclidean distance transform is to define a parabola at each cell. We will use the *vertex* form of a parabola:

$$y = a(x - h)^2 + k$$

The vertex is the lowest point of the parabola, at (h, k).[1] For each cell of our image, we will draw one parabola where h is the *index* of the cell. For occupied cells, we will set $k = 0$. For free cells, we will set $k = \infty$. For all the parabolas, we will set $a = 1$. The *lower envelope* of the parabolas gives the squared distance transform.

Let's look at an example to see how it works. Figure 1a shows a 1D binary image, where 0 indicates a *free* cell and 1 indicates an *occupied* cell. Figure 1b shows the three parabolas which come from the occupied cells, with h values 1, 3, and 7 (see Figure 1b). The parabolas from free cells are at infinity, so they do not show up on the graph.

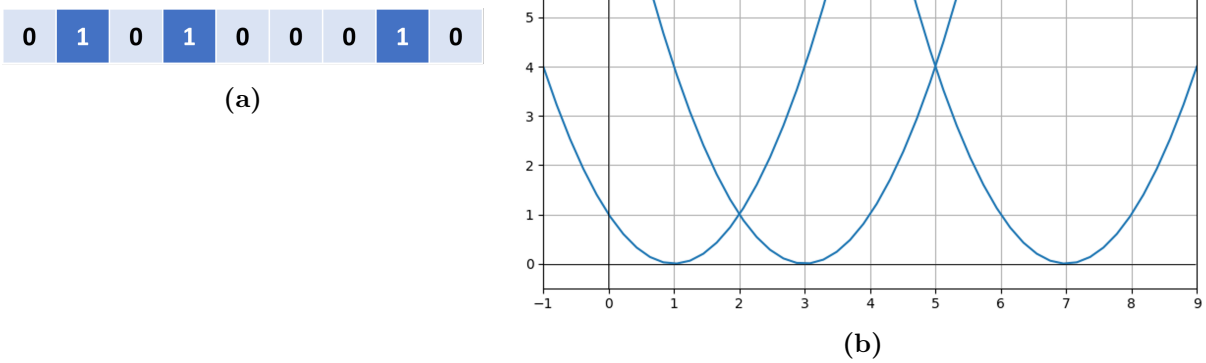


Figure 1: The parabolas for the binary image in Figure 1a. The parabolas from the occupied cells are given by the functions $(x - 1)^2$, $(x - 3)^2$, and $(x - 7)^2$.

[1] You might be used to seeing a parabola written like this: $y = ax^2 + bx + c$. We can turn one form into the other with some algebra.

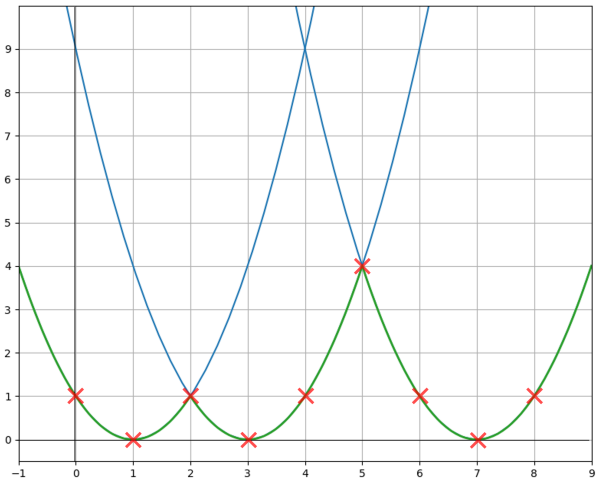


Figure 2: The lower envelope of the parabolas for the image in Figure 1a is drawn in green. The red X's indicate the value of the lower envelope at each index, which is equal to the squared distance transform ($DT[0] = 1$, $DT[1] = 0$, $DT[5] = 4$, etc.).

1.1 Lower Envelope of Parabolas

The **lower envelope** of the parabolas is the function you would get if you traced along the lowest points of the function. We can get the *squared* distance transform by evaluating the lower envelope at each index.[2] The lower envelope for our example is shown in Figure 2

The lower envelope is a *piecewise function*, which we can represent by a list of parabolas and the range of each one. To represent it in code, we will use two vectors: `paras` will contain the index of the parabolas in the lower envelope. `ranges` will represent the horizontal range that each parabola acts over. The parabola at `paras[i]` will be active from `ranges[i]` to `ranges[i+1]`.

This is illustrated for our example in Figure 3. The lower envelope vectors are:

$$\texttt{paras} = \begin{bmatrix} 1 & 3 & 7 \end{bmatrix}, \quad \texttt{ranges} = \begin{bmatrix} -\infty & 2 & 5 & \infty \end{bmatrix}$$

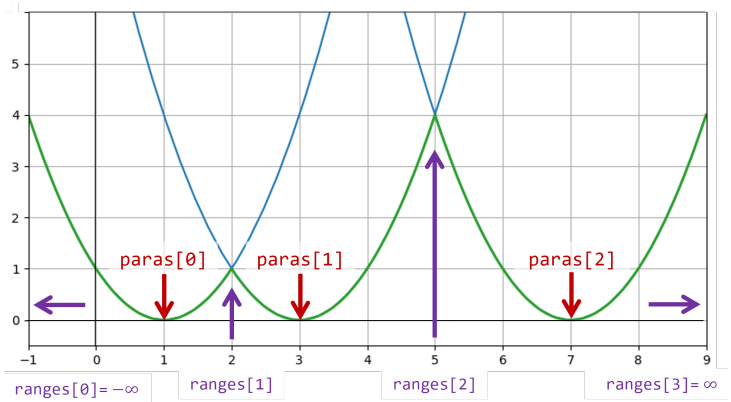


Figure 3: The lower envelope of the parabolas is represented by a vector of the parabola indices, `paras`, and a vector of their active ranges, `ranges`.

[2] If we want the normal distances rather than the squared distances, we simply take the square root.

The distance transform at each index i can be evaluated using the piecewise function that describes the lower envelope:

$$DT[i]^2 = \begin{cases} (i-1)^2 & i < 2 \\ (i-3)^2 & 2 \leq i < 5 \\ (i-7)^2 & i \geq 5 \end{cases}$$

1.2 1D Euclidean Distance Transform Algorithm

The algorithm for computing the distance transform has two steps:

1. Calculate the lower envelope of the parabolas.
2. Fill in the distance transform by evaluating the lower envelope at each index.

Before we introduce the algorithm, we will define a vector f, with the same length as the 1D image, and the values:[3]

$$f[i] = \begin{cases} 0 & \text{if } i \text{ is occupied} \\ \infty & \text{if } i \text{ is free} \end{cases} \tag{1}$$

The function for the parabola from index i is:

$$y = (x-i)^2 + f[i] \tag{2}$$

Algorithm 1 The 1D Euclidean Distance Transform

```
1:  function DistanceTransform1D(f)
2:      N = length of f
3:      paras = [0]                    ▷ Initialize lower envelope with the parabola at 0
4:      ranges = [−∞, ∞]
5:      for i = 1 to N-1 do            ▷ Step 1: Compute lower envelope
6:          k = length of paras - 1
7:          s = intersection between parabolas from i and paras[k]
8:          while s ≤ ranges[k] do     ▷ Case (i): Remove unneeded parabolas
9:              remove last element in paras
10:             k = length of paras - 1
11:             s = intersection between parabolas from i and paras[k]
12:         append i to paras          ▷ Case (ii): Add new parabola
13:         ranges[k] = s
14:         ranges[k+1] = ∞
15:     DT = vector of N zeros         ▷ Initialize distance transform
16:     k = 0
17:     for i = 0 to N-1 do            ▷ Step 2: Update distance transform
18:         while ranges[k+1] < i do
19:             k += 1
20:         DT[i] = (i - paras[k])^2 + f[paras[k]]
21:     return DT
```

[3]This is how we initialized the Manhattan distance transform in the algorithm we saw in class.

The main part of the algorithm is the computation of the lower envelope. We consider the parabolas from one index at a time, starting at index 0. When we consider a new parabola at index i, we will compute its intersection with the rightmost parabola in the current lower envelope.

The horizontal component of the intersection between two parabolas can be obtained with algebra. Consider two parabolas, coming from indices p and q, $(i-p)^2 + f[p]$ and $(i-q)^2 + f[q]$. Their intersection is:

$$s = \frac{(f[p] + p^2) - (f[q] + q^2)}{2p - 2q} \tag{3}$$

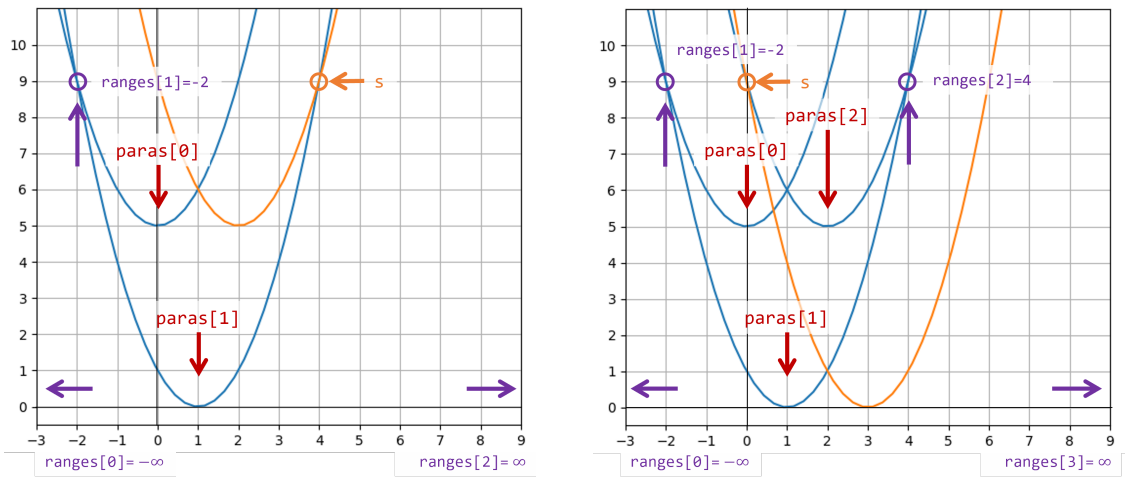
Let's say the rightmost parabola comes from index k. There are two cases to consider:

(i) $s >$ `ranges[k]`: The intersection for this new parabola comes *after* the beginning of the range of the rightmost parabola in the current envelope. The new parabola is added to the envelope, with range starting at s.

(ii) $s \leq$ `ranges[k]`: The intersection comes *before* the beginning of the range of the rightmost parabola. This means that the rightmost parabola should be removed from the lower envelope.

After considering each index and computing the lower envelope, the distance transform is updated by evaluating the lower envelope values.

Example: Let's look at the example image in Figure 1a. To make all the parabolas visible, we will use an f value of 5 for free cells, instead of infinity.[4] Consider the cell at index 2 of the image, which is a free cell. When $i = 2$, we will have added two parabolas, at 0 and 1, to the lower envelope already, shown in blue in Figure 4a. Our parabola vector is `paras` = `[0,1]`, and the ranges vector is `ranges` = `[-∞,-2,∞]`.



(a) **(b)**

Figure 4: An illustration of the lower envelope computation for cell $i = 2$ (a) and cell $i = 3$ (b).

[4] This is a bad choice, but 5 happens to be higher than the highest squared distance (which is 4) in this particular example, so it will still work.

The parabola being considered is drawn in orange. The rightmost parabola in the envelope is the parabola from `paras[1]` = 1, with range starting at `ranges[1]` = -2. First, we calculate its intersection between the new parabola and the rightmost, $s = 4$. This is an example of case (i), since s > `ranges[1]`. We add the new parabola to the envelope starting at $s = 4$, making `paras` = `[0,1,2]` and `ranges` = `[-`∞`,-2,4,`∞`]`.

Next, we consider the parabola from $i = 3$, shown in orange in Figure 4b. The intersection with the rightmost parabola, from `paras[2]` = 2, is at $s = 0$. This is an example of case (ii), since $s \leq$ `ranges[2]` = 4. The parabola at $i = 2$ is no longer a part of the lower envelope, so we remove it, making `paras` = `[0,1]`. The new rightmost parabola is at `paras[1]` = 1, which intersects with the current parabola at $s = 2$. This is case (i), since s > `ranges[1]`, so we add the new parabola to the envelope, making `paras` = `[0,1,3]` and `ranges` = `[-`∞`,-2,2,`∞`]`.

1.3 Why it works

It's not a coincidence that we can find the distance transform by calculating the lower envelope of parabolas. We can see why that is by writing down the problem the distance transform more formally.

For each index, i, the distance transform is the distance to the nearest occupied cell. The Euclidean distance between cell i and an occupied cell at index p is $d(i, p) = \sqrt{(i-p)^2}$. Squaring both sides gives us the formula for a parabola with vertex at $(p, 0)$:

$$d(i, p)^2 = (i - p)^2 \tag{4}$$

The squared distance transform is the smallest value of $d(i, p)^2$ for all occupied cells $[p_1, p_2, \ldots, p_m]$. We can write this mathematically:

$$DT[i]^2 = \min\left((i - p_1)^2, (i - p_2)^2, \ldots, (i - p_m)^2\right) \tag{5}$$

For the example image in Figure 1a, there are three occupied cells: $p_1 = 1$, $p_2 = 3$, and $p_3 = 7$. Notice that each squared distance function is a parabola in vertex form. The distance transform in Equation (5) is just the formula for the lower envelope of the parabolas![5]

2 2D Euclidean Distance Transform

It turns out that we can compute the 2D distance transform using the 1D distance transform algorithm on each row of the map, then on each column. The algorithm for the 2D transform has three steps:

1. Initialize the distance transform using the same equation as for the 1D case:

$$DT[i, j] = \begin{cases} 0 & \text{if index is occupied} \\ \infty & \text{if index is free} \end{cases} \tag{6}$$

2. Perform the 1D distance transform over each row.
3. Perform the 1D distance transform over each column.

[5] We can also write the distance transform in terms of *all* cells instead of just occupied cells, by adding $f[p]$, as defined in Equation (1) to each parabola.

Algorithm 2 The 2D Euclidean Distance Transform

```
1: function DistanceTransform2D(map)
2:     H, W = height and width of map
3:     DT = vector of H×W zeros                          ▷ Initialize distance transform
4:     for i = 0 to H-1 do
5:         for j = 0 to W-1 do
6:             if map[i, j] is occupied then
7:                 DT[i, j] = ∞
8:     for i = 0 to H-1 do                    ▷ Perform 1D distance transform on rows
9:         f = row i of DT
10:        row i of DT = DistanceTransform1D(f)
11:    for j = 0 to W-1 do                 ▷ Perform 1D distance transform on columns
12:        f = column j of DT
13:        column i of DT = DistanceTransform1D(f)
14:    return DT
```

See the pseudocode in Algorithm 2. When we iterate over the rows in step (2), we use the values of the distance transform for the current row as function f. The parabola for index j in row r is:

$$y = (x - j)^2 + DT[r, j]$$

The values of $DT[r, j]$ will either be 0 or ∞. That is the same case as the 1D distance transform. The difference is that when we apply the algorithm to the columns, we will use the updated distance transform which contains the squared 1D distance transform for that row. The parabola for an index i in column c will be:

$$y = (x - i)^2 + DT[i, c]^2$$

2.1 Example

To illustrate how the algorithm works, let's use the small example in Figure 5. Recall that the Euclidean distance transform will compute the *squared* distance to the nearest

0	0	0	0	0	1
0	1	0	0	0	1
0	1	0	0	0	0
0	1	1	0	0	0
0	1	1	0	0	0
0	0	0	0	0	0

Figure 5: A 2D binary image.

occupied cell. By inspection, the (squared) distance transform for this image is:

$$DT^2 = \begin{bmatrix} 2 & 1 & 2 & 4 & 1 & 0 \\ 1 & 0 & 1 & 4 & 1 & 0 \\ 1 & 0 & 1 & 2 & 2 & 1 \\ 1 & 0 & 0 & 1 & 4 & 4 \\ 1 & 0 & 0 & 1 & 4 & 9 \\ 2 & 1 & 1 & 2 & 5 & 10 \end{bmatrix}$$

After step 2, the algorithm will give us the 1D distance transform over each row:

$$DT^2 = \begin{bmatrix} 25 & 16 & 9 & 4 & 1 & 0 \\ 1 & 0 & 1 & 4 & 1 & 0 \\ 1 & 0 & 1 & 4 & 9 & 16 \\ 1 & 0 & 0 & 1 & 4 & 9 \\ 1 & 0 & 0 & 1 & 4 & 9 \\ \infty & \infty & \infty & \infty & \infty & \infty \end{bmatrix}$$

In step 3, we will update the column values, using the existing row values. For our example, we will update the values in the first column using $f = [25, 1, 1, 1, 1, \infty]$, which are the values of the first column in DT^2. The first index, $i = 0$, of the first column is given by:

$$\begin{aligned} DT[0,0]^2 &= \min\big((0-0)^2 + 25, (0-1)^2 + 1, (0-2)^2 + 1, (0-3)^2 + 1, \\ &\qquad (0-4)^2 + 1, (0-5)^2 + \infty\big) \\ &= \min(25, 2, 5, 10, 17, \infty) \end{aligned}$$

This gives us $DT[0,0]^2 = 2$, which is the value we got by inspection.

Let's take a look at the parabolas for the fifth column, shown in Figure 6. The last parabola has a vertex at infinity, so we can't see it on the graph. Evaluating the lower envelope at each index gives us the correct values of the distance transform for the column!

2.2 Why it works

For a 2D map, we need to define the distance transform with respect to the 2D Eculidean distance. We will also include the vector f, which is we get from Equation (6). For an image with N rows and M columns, the squared distance transform at cell (i, j) is defined by:

$$DT[i,j]^2 = \min_{0 \le p < N,\, 0 \le q < M} \big((i-p)^2 + (j-q)^2 + f[p,q]\big) \tag{7}$$

This means that $DT[i,j]^2$ is equal to the minimum Euclidean distance from the cell (i, j) to each cell in the graph, (p, q), plus the value of f at (p, q).[6] The first two terms in the function in Equation (7) are independent of each other, so we can rewrite the function like this:

$$DT[i,j]^2 = \min_{0 \le p < N} \left((i-p)^2 + \underbrace{\min_{0 \le q < M} \big((j-q)^2 + f[p,q]\big)}_{\text{1D distance transform for row } p} \right) \tag{8}$$

[6]Remember that f will be infinity for the free cells, so this is the same as saying that the distance transform is the distance to the nearest occupied cell.

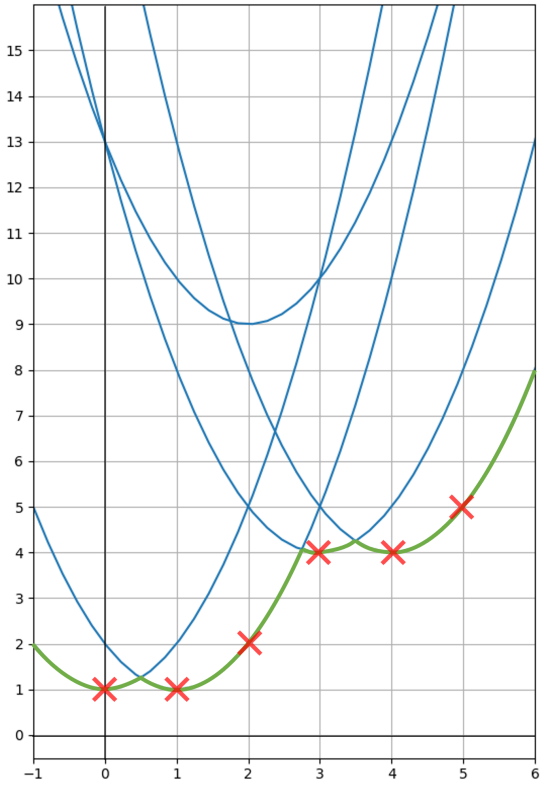


Figure 6: The parabolas for column 4 (the fifth column) of the test image. The lower envelope is indicated in green, and the values of the lower envelop evaluated at each index are marked with red X's.

That means we can write our 2D distance transform in terms of two 1D distance transforms:

$$DT[i,j]^2 = \min_{0 \le p < N} \left((i-p)^2 + DT_p^2 \right) \tag{9}$$

where DT_p^2 is the (squared) 1D distance transform for row p.

References

[1] P. F. Felzenszwalb and D. P. Huttenlocher, "Distance transforms of sampled functions," *Theory of computing*, vol. 8, no. 1, pp. 415–428, 2012.