CS 330: Homework 4 Extra Credit

Michael Lin

November 4, 2015

Problem 1 Extra Credit

Algorithm 1 Functions for finding outer tangents

```
1: function TANGENT(polygon P[1,...,n], polygon Q[1,...,m])
2:     i = 1, j = 1
3:     p = P[i], q = Q[j]
4:     a = HELPER(p, q), b = HELPER(q, p)
5:     while a ≠ (L, R) and b ≠ (R, L) do
6:         if a = (L, L) or (R, L) then
7:             j = j + 1 mod m
8:         else if a = (R, R), (R, C), or (C, R) then
9:             j = j − 1 mod m
10:        if b = (R, R) or (L, R) then
11:            i = i + 1 mod n
12:        else if b = (L, L), (L, C), or (C, L) then
13:            i = i − 1 mod n
14:        p = P[i], q = Q[j]
15:        a = HELPER(p, q), b = HELPER(q, p)
16:    T_1 = (p, q)
17:
18:    i = 1, j = 1
19:    p = P[i], q = Q[j]
20:    a = HELPER(p, q), b = HELPER(q, p)
21:    while a ≠ (R, L) and b ≠ (L, R) do
22:        if a = (R, R) or (L, R) then
23:            j = j + 1 mod n
24:        else if a = (L, L), (L, C), or (C, L) then
25:            j = j − 1 mod n
26:        if b = (L, L) or (R, L) then
27:            i = i + 1 mod m
28:        else if b = (R, R), (R, C), or (C, R) then
29:            i = i − 1 mod m
30:        p = P[i], q = Q[j]
31:        a = HELPER(p, q), b = HELPER(q, p)
32:    T_2 = (p, q)
33:
34:    return T_1, T_2
```

Algorithm 2 Helper functions for above main function

function HELPER(point q, vertex $P[i]$)
    $a_i = \text{DIRECTION}(q, P[i-1 \mod n], P[i \mod n])$
    $b_i = \text{DIRECTION}(q, P[i \mod n], P[i+1 \mod n])$
    return (a_i, b_i)

function DIRECTION(a, b, c)
    $d = \det \begin{pmatrix} 1 & a_x & a_y \\ 1 & b_x & b_y \\ 1 & c_x & c_y \end{pmatrix}$
    if $d > 0$ **then return** L ▷ L for left, R for right, C for collinear
    if $d < 0$ **then return** R
    if $d = 0$ **then return** C

Proof of correctness. The algorithm relies on the same principle as discussed for problem 1b. Starting at the left-most points $P[1], Q[1]$ and based on their direction with respect to each other, we make decisions to march along each polygon in a specified direction. In general, tangents are characterized by boundaries where direction between a point and the tangent changes either from left to right (L, R) or from right to left (R, L). In the case of tangents between two polygons, one outer tangent can be characterized as (L, R) going from tangent point $P[T_1]$ to tangent point $Q[T_1]$ and (R, L) going from tangent point $Q[T_1]$ to tangent point $P[T_1]$. The other outer tangent is just the opposite. In choosing the left-most points to begin the search, we will ensure that each polygon will, for the most part, march in the same direction (e.g. polygon P marches clockwise, polygon Q marches counterclockwise). Eventually, we will reach the desired outer tangent. The same argument is true to the find the other outer tangent. □

Proof of runtime. The algorithm for the most part will only march in the same direction until reaching the desired outer tangent. This ensures that each pair of points will be compared only once, and each point will always change unless it arrives at the outer tangent. Therefore, we have at most $O(n + m)$ pairs of points to check. Each pair of points takes a constant amount of time to check. Therefore, the algorithm runs in $O(n + m)$ time. □

Problem 3 Extra Credit

Algorithm 3 Algorithm to compute the set of maximal points of S

```
1: function MAXIMAL(set S = {p_1, ..., p_n})
2:     X = sort S by x coordinate
3:     S_x = ∅                                              ▷ S_x, S_y are stacks
4:     for i in 1 to n do
5:         PUSH(X[i], S_x)
6:     C_x = ∅                                              ▷ C_x is a set where each value appears only once
7:     x_a = POP(S_x), x_b = POP(S_x)
8:     while S_x ≠ ∅ do
9:         if y(x_a) ≥ y(x_b) and z(x_a) ≥ z(x_b) then      ▷ y(·), z(·) are the y, z coordinates of the point
10:            x_b = POP(S_x)
11:        else
12:            ADD(x_a, C_x)
13:            x_a = POP(S_x)
14:    ADD(x_a, C_x)
15:    if y(x_a) ≤ y(x_b) then
16:        ADD(x_b, C_x)
17:
18:    Y = sort C_x by y coordinate
19:    S_y = ∅
20:    for i in 1 to n do
21:        PUSH(Y[i], S_y)
22:    C_y = ∅                                              ▷ C_y is a set where each value appears only once
23:    y_a = POP(S_y), y_b = POP(S_y)
24:    while S_y ≠ ∅ do
25:        if x(y_a) ≥ x(y_b) and z(y_a) ≥ z(y_b) then      ▷ x(·), z(·) are the x, z coordinates of the point
26:            y_b = POP(S_y)
27:        else
28:            ADD(y_a, C_y)
29:            y_a = POP(S_y)
30:    ADD(y_a, C_y)
31:    if x(y_a) ≤ x(y_b) then
32:        ADD(y_b, C_y)
33:
34:    Z = sort C_y by z coordinate
35:    S_z = ∅
36:    for i in 1 to n do
37:        PUSH(Z[i], S_z)
38:    C_z = ∅                                              ▷ C_z is a set where each value appears only once
39:    z_a = POP(S_z), z_b = POP(S_z)
40:    while S_z ≠ ∅ do
41:        if x(z_a) ≥ x(z_b) and y(z_a) ≥ y(z_b) then      ▷ x(·), y(·) are the x, y coordinates of the point
42:            z_b = POP(S_z)
43:        else
44:            ADD(z_a, C_z)
45:            z_a = POP(S_z)
46:    ADD(z_a, C_z)
47:    if x(y_a) ≤ x(y_b) then
48:        ADD(z_b, C_z)
49:
50:    return C_z
```

Proof of correctness. First, consider the points sorted by x coordinates. In descending order, we check if one point dominates another point with smaller or the same x coordinate. If so, the point with smaller or the same x coordinate cannot be in the set of maximal points. If not, then both points may still be in the set of maximal points. Continuing in this manner until all points are exhausted will give all points where smaller x coordinates corresponds to larger y coordinates. We then consider remaining candidates C_x sorted by y coordinates. Finally, we consider remaining candiates C_y sorted by z coordinates. This takes into account the fact that some points may have either the same x, y, or z coordinates. C_z is a subset of the candidates C_x as well as a subset of the candidates $C_y \subseteq C_x$, which will give the set of maximal points. □

Proof of runtime. Sorting the set S in x, y, and z coordinates each takes $O(n \log n)$ time. Adding sorted values into a stack takes $O(n)$ time. The while-loop (for x, y, and z) makes comparisons whether a point dominates another point. There are at most $O(n)$ comparisons, and each comparison takes a constant amount of time. Thus, the 2 while-loops take $O(n)$ time. Thus, the algorithm overall takes $O(n \log n)$ time. □