A person smiling for the camera

Description automatically generated

**Convex Hull Calculation Using Gift Wrapping**

X Y 10 13 4 10 8 12 15 13 8 14 5 4 12 14 11 1 4 13 11 6 8 8 12 14 2 5 3 6 12 10 7 7 10 8 8 8 12 10 4 6 11 18 11 15 9 14 The Convex Hull Algorithm leads to the smallest convex area (Polygon) that includes all the points in the dataset. In the given table, a list of data points is provided. Extract the points that lie on the vertices of the smallest convex area created by the algorithm described below. The algorithm begins with i=0 and a point p0​ known to be on the convex hull, such as the leftmost point. It selects the point pi+1​ such that all points are to the right of the line pi pi+1​. This point can be found by comparing the polar angles of all points with respect to point pi​ taken as the center of polar coordinates. Letting i=i+1, and repeating until one reaches ph=p0​ again, yields the convex hull in h steps. Write the Power Query M code that will provide a table of the X and Y coordinates of the points on the convex hull for the data provided. Use the above data as the source file for the code.

Thought for 140 seconds

Okay, I need to write a Power Query M code to find the convex hull of the given points using the specified algorithm. Let me think through how to approach this.

First, the algorithm starts with the leftmost point as p0. If there are multiple points with the same leftmost X, then the one with the smallest Y maybe? Or wait, the problem says the leftmost point is known to be on the convex hull. So I need to find the point with the minimum X value. If there's a tie, like two points with the same X, then maybe the one with the lowest Y? Because the leftmost and bottommost would definitely be on the convex hull.

So step one is to find the leftmost point. Let's sort the points by X ascending, then Y ascending, and take the first one. That's p0.

Then, the algorithm uses the gift wrapping method (Jarvis march). The next point is selected such that all other points are to the right of the line formed by p0 and p1. To find this, for each candidate point, we calculate the cross product to determine the orientation. The correct next point is the one that makes the smallest polar angle with p0, but in terms of cross product terms, it's the one that is the most counter-clockwise.

Wait, the algorithm says to select pi+1 such that all other points are to the right of the line pi pi+1. So the next point is the one that, when connected from the current point, forms a line where all other points are on the right side. So, for each candidate point, we check the orientation of every other point relative to the line from current point to candidate. If all other points are to the right (or colinear, but since convex hull includes colinear points only if they're on the edge), then that's the next point.

But that sounds computationally intensive. Alternatively, in Jarvis march, the next point is the one that makes the smallest counter-clockwise turn from the current direction. So for each current point pi, compute the next point pi+1 by checking all other points and selecting the one with the largest polar angle (or the smallest when compared in a certain way). Alternatively, using cross products to determine the orientation.

Wait, maybe the way to do it is to, for each point q (not pi), compute the cross product between the vector from pi to q and the vector from pi to each other point. The correct next point is the one that has all other points to the right of the line pi to q. So for each candidate q, check if all other points are on the right side of the line pi->q. The first such q in the list would be the next point. But how to efficiently compute this.

Alternatively, in the gift wrapping algorithm, for the current point pi, the next point pi+1 is the one that has the smallest polar angle with respect to pi. But how to compute this. Alternatively, for each candidate q, check if for every other point r, the cross product (q - pi) × (r - pi) is >=0. Because that would mean that r is to the right of the line from pi to q.

But that approach for each candidate q would require checking all other points r. The candidate q for which this is true for all r is the next point on the convex hull.

So the steps are:

1. Start with the leftmost point (p0).

2. For the current point pi, find the next point pi+1 such that for all other points r, the cross product (pi+1 - pi) × (r - pi) <=0 (assuming we're moving counter-clockwise). Wait, the cross product sign depends on the order. Let's think: The cross product (q - p) × (r - p) tells us the direction. If positive, r is to the left of the line p to q; if zero, colinear; if negative, to the right.

But the algorithm requires that all points are to the right of the line pi pi+1. So for a candidate q, we need that for every other point r, (q - pi) × (r - pi) >=0. Because then r is to the right or colinear. Wait, but this would mean that the line pi to q has all other points on or to the right of it. So that's the correct candidate.

So the way to find the next point is, for each candidate q (excluding pi), check if all other points r (excluding pi and q) satisfy cross product (q.x - pi.x, q.y - pi.y) × (r.x - pi.x, r.y - pi.y) <=0. Wait, cross product here is (qx - pix)\*(ry - piy) - (qy - piy)\*(rx - pix). If this is >=0, then the point r is on the left side of the line from pi to q. But the algorithm requires that all points are to the right, so for a candidate q to be the next point, all other r must be on the right side of piq. So the cross product (q - pi) × (r - pi) should be <=0. So for candidate q, we check for all r: ( (qx - pix) \* (ry - piy) - (qy - piy)\*(rx - pix) ) <=0. If any r violates this (i.e., cross product >0), then q is not the next point.

So for each candidate q (starting with the next possible point), we need to check all other r points. The first q where all r's cross product is <=0 is the next point.

But this could be computationally expensive for each candidate. But in practice, for the gift wrapping algorithm, the next point is found by comparing angles. Alternatively, you can compute for each candidate q, the cross product with all other r points, and find the q that is the "most counter-clockwise" possible. Wait, perhaps there's a smarter way. Let's think again.

In the gift wrapping approach, the next point is selected by choosing the point that makes the largest angle in the counter-clockwise direction. To find this, for each candidate q, compute the cross product of the vector from pi to q and from pi to r (for all other r). The correct q is the one where for all r, the cross product (pi to q, pi to r) is >=0 (i.e., r is on or to the left of the line pi to q). So the algorithm needs to find the q where for all r, the cross product between (q - pi) and (r - pi) is <=0. Wait, maybe I'm getting the direction reversed.

Alternatively, here's the standard approach for Jarvis march:

For each step, after selecting pi, the next point pi+1 is initialized to the first candidate point. Then, for every other point r, check if the turn from pi to pi+1 to r is a right turn. If it is, then pi+1 is updated to r. Alternatively, in terms of cross product:

For the current candidate pi+1, compare with each r. Compute the cross product (pi+1 - pi) × (r - pi). If this cross product is positive, then r is to the left of the line from pi to pi+1, so pi+1 is not the correct next point. So, if any r has a positive cross product with pi+1, then we need to select a new candidate.

Wait, perhaps the correct way is to iterate through all possible points to find the one that is the "most counter-clockwise" from pi. That is, for each candidate q, check if for any other point r, the cross product (q - pi) × (r - pi) is positive. If so, then q is not the correct next point. The correct next point is the q for which there are no such r's. However, this would involve for each q, checking all r's. But that's O(n^2) time for each step.

Alternatively, the standard Jarvis march approach for each step is to select the next point by comparing each candidate point and keeping track of the one that forms the largest angle (i.e., the most counter-clockwise). This is done by initializing the next point as a possible candidate, and then for each other point, checking whether it makes a larger angle. The cross product can be used for this comparison.

So for example:

Start with p0. Then, for the next point, start with the first candidate (say, the first point in the list that's not p0). For each other point q, compute the cross product of (current candidate - p0) and (q - p0). If the cross product is positive, then q is to the left of the line p0 to current candidate, so current candidate can't be the next point. So we update the candidate to q. If the cross product is zero, then they are colinear, and we pick the farthest one.

So the process is:

current = p0

hull = [current]

do:

next = first point in the list (not current)

for each point q in the dataset:

if q is current, skip

cross = cross\_product(next - current, q - current)

if cross < 0: # q is to the right of current-next, so next remains

continue

elif cross > 0: # q is to the left, so next is replaced by q

next = q

else: # colinear, check the distance

if distance(q, current) > distance(next, current):

next = q

add next to hull

current = next

until current == p0

Wait, but in this approach, during each step, we find the next point by comparing all points and selecting the one that is the most counter-clockwise (i.e., the one that when compared to the current candidate, forms a left turn). This way, after checking all points, the next point is the one that is the farthest in the counter-clockwise direction. This would correctly build the convex hull.

So, to implement this in Power Query M, I need to:

1. Find the leftmost point (p0). If there are multiple points with the same X, choose the one with the smallest Y.

2. Initialize the convex hull list with p0.

3. Iterate, finding each next point using the cross product method.

4. Stop when the next point is back to p0.

But how to do this in M, considering that loops are not straightforward. Power Query is functional, so recursion is possible, but there's a limit to recursion depth.

Alternatively, perhaps using a loop with a termination condition.

But first, let's outline the steps.

First, get the data as a list of records, each with X and Y.

Then, find the starting point: leftmost (min X), then min Y if there are ties.

Then, in each iteration, find the next point based on the current point.

The cross product between two vectors (a, b) and (c, d) is a\*d - b\*c.

So, for two points current and next, and a candidate point q, the cross product is (next.X - current.X)\*(q.Y - current.Y) - (next.Y - current.Y)\*(q.X - current.X).

If this cross product is positive, q is to the left of the line from current to next. So during the selection of next, if any q makes this cross product positive, then the current candidate next is not the correct one, and we need to select a new candidate.

Wait, no. In the standard Jarvis march, for each current point, the next point is determined by finding the point that is the most counter-clockwise. That is, for the current point, iterate through all other points and select the one that makes the smallest angle in the counter-clockwise direction. The way to do this is by comparing points using the cross product: if the cross product between the current-next vector and current-q vector is positive, then q is more counter-clockwise than next, so next becomes q.

So the steps for each iteration are:

current\_point = last point added to the hull

next\_candidate = first point in the list (not current)

for each q in points:

if q == current\_point, skip

compute the cross product between (current\_point to next\_candidate) and (current\_point to q)

if cross < 0: next\_candidate remains

else if cross > 0: next\_candidate becomes q

else (cross is 0):

compute the distance from current\_point to next\_candidate and to q. If q is farther, replace next\_candidate with q.

So, for each q, if it is more counter-clockwise (cross product positive), we update the next\_candidate. If colinear (cross zero), pick the farthest one.

This way, after checking all points, next\_candidate is the correct next point on the hull.

So, the algorithm proceeds by adding next\_candidate to the hull, and then using it as the current\_point for the next iteration.

Now, implementing this in M.

First, let's get the data into a table.

Let me see, the sample data is given as X and Y columns. So the first step is to read that into a list of points.

Then, find the leftmost point (min X, then min Y).

Then, loop until we return to the starting point.

But Power Query doesn't have loops, but you can use recursion or a list accumulator.

But recursion depth can be a problem. However, for a convex hull, the number of steps is equal to the number of points on the hull, which for 22 points could be up to 22 steps, but likely much less. Power Query allows recursion up to a certain depth, maybe 100, which should be sufficient.

So, here's the plan:

- Read the source data as a list of records.

- Find the starting point (leftmost, then min Y).

- Initialize the hull list with the starting point.

- Use a recursive function to find the next points until the starting point is revisited.

Now, the code.

First, the source data is in a table with X and Y columns.

Read that into a list:

let

Source = Table.FromRows({{10,13}, {4,10}, ... }, {"X", "Y"}),

// Or the actual data entries

...

But for brevity, assuming the data is already loaded as a table.

Then, convert the table to a list of records:

points = Table.ToRecords(Source)

Then, find the leftmost point:

leftmost = List.Sort(points, (a, b) =>

if a[X] < b[X] then -1

else if a[X] > b[X] then 1

else if a[Y] < b[Y] then -1

else 1

){0}

So, sorted by X ascending, then Y ascending, take the first.

Initialize the hull as a list: hull = {leftmost}

Then, define a recursive function that, given the current point and the hull, finds the next point and appends it until the starting point is found again.

But in Power Query, functions can't be recursive in the same way as in other languages. So perhaps using a loop with a List.Generate or another approach.

Alternatively, let's think of using a loop with a list that accumulates the hull points. We can use a loop that continues until the next point is the starting point.

But in M, you can't have a while loop. So the way to do this is via recursion.

Here's an outline:

hull = {p0}

current = p0

next\_point = FindNextPoint(current, points)

while next\_point != p0:

hull = hull & {next\_point}

current = next\_point

next\_point = FindNextPoint(current, points)

So, the key is the FindNextPoint function.

Implementing FindNextPoint:

function FindNextPoint(current, allPoints) as record:

next\_candidate = first point in allPoints that is not current

for each q in allPoints:

if q == current then skip

cross = (next\_candidate.X - current.X) \* (q.Y - current.Y) - (next\_candidate.Y - current.Y) \* (q.X - current.X)

if cross > 0 then:

next\_candidate = q

else if cross == 0 then:

distance\_current\_to\_next = (next\_candidate.X - current.X)^2 + (next\_candidate.Y - current.Y)^2

distance\_current\_to\_q = (q.X - current.X)^2 + (q.Y - current.Y)^2

if distance\_current\_to\_q > distance\_current\_to\_next:

next\_candidate = q

return next\_candidate

But in M, how to implement this.

In M code, perhaps:

FindNextPoint = (current as record, points as list) as record =>

let

// Filter out the current point

filtered = List.Select(points, each \_[X] <> current[X] or \_[Y] <> current[Y]),

// Initialize next\_candidate as the first point in filtered

initialCandidate = filtered{0},

// Iterate through all points to find the correct candidate

finalCandidate = List.Accumulate(

filtered,

initialCandidate,

(state, q) =>

let

cross = (state[X] - current[X]) \* (q[Y] - current[Y]) - (state[Y] - current[Y]) \* (q[X] - current[X]),

distanceState = Number.Power(state[X] - current[X], 2) + Number.Power(state[Y] - current[Y], 2),

distanceQ = Number.Power(q[X] - current[X], 2) + Number.Power(q[Y] - current[Y], 2)

in

if cross > 0 then

q

else if cross = 0 then

if distanceQ > distanceState then q else state

else

state

)

in

finalCandidate

Wait, but List.Accumulate takes a list and a seed (initial state), then applies a function to each element, updating the state. Here, the seed is the initialCandidate. Then, for each q in filtered (excluding current), we compare the current state (the current candidate) with q.

But the cross product here is (state.X - current.X) \* (q.Y - current.Y) - (state.Y - current.Y) \* (q.X - current.X). If this is positive, then q is to the left of the line from current to state, which means state is not the right candidate. So, in this case, the new candidate should be q. Otherwise, if cross is zero, check the distance. Otherwise, keep the current candidate.

Wait, but the cross product in the standard Jarvis march is (q.X - current.X) \* (state.Y - current.Y) - (q.Y - current.Y) \* (state.X - current.X). Wait, maybe the order is reversed.

Wait, the cross product between two vectors (from current to state) and (from current to q) is:

vector1 = (state.X - current.X, state.Y - current.Y)

vector2 = (q.X - current.X, q.Y - current.Y)

cross = vector1.X \* vector2.Y - vector1.Y \* vector2.X

Which is (state.X - current.X) \* (q.Y - current.Y) - (state.Y - current.Y) \* (q.X - current.X)

If cross is positive, then vector2 is counter-clockwise from vector1. So, in this case, q is a better candidate than the current state. So, in the accumulation, if cross >0, we replace the state with q. Because this would mean that q is more counter-clockwise than the current state. So, the code above is correct.

Wait, no: if the cross product is positive, then the angle from vector1 to vector2 is counter-clockwise. So in this case, q is "more to the left" than the current candidate. So for Jarvis march, we want to select the q that is the most counter-clockwise. So during the accumulation, whenever we find a q with a higher cross product (positive), we update the candidate.

So, the code in the List.Accumulate is:

for each q in filtered:

compute cross between current candidate (state) and q.

if cross > 0: q is better, so next candidate is q.

else if cross is 0: check distance.

else: keep current candidate.

So the code as written is correct.

Then, once the finalCandidate is found, return it.

Once we have this function, the loop can be built using recursion.

But in Power Query, how to build the list of hull points.

We can create a recursive function that appends points to the hull list until the next point is the starting point.

But since each step depends on the previous, and recursion is limited, perhaps using a helper function.

Alternatively, here's an approach:

Start with the initial hull (hull = {leftmost}).

current = leftmost.

loop:

next\_point = FindNextPoint(current, points)

if next\_point == hull{0}, exit.

else append next\_point to hull.

current = next\_point.

But how to implement this loop.

Alternatively, use List.Generate to generate the sequence until the next\_point is the first one.

But List.Generate can be used to create a list of next points until the termination condition.

Alternatively, here's a possible code structure:

hull = List.Generate(

() => [Point = leftmost, Prev = null],

each [Point] <> hull{0} or [Prev] = null,

each [

NextPoint = FindNextPoint([Point], points),

Prev = [Point],

Point = NextPoint

],

each [Point]

)

Wait, List.Generate syntax is:

List.Generate(

initial value as a function,

condition as a function (continues while true),

next as a function (generates next state),

transform as a function (selects the value to output)

)

But in our case, the initial state is the leftmost point. But we need to generate the next points until we return to the leftmost.

But the first point is leftmost. The next points are generated until the next\_point is leftmost again.

Wait, the first step is to add leftmost, then generate next points until we loop back.

So perhaps:

Start with the leftmost as the first point.

The next points are generated by repeatedly applying FindNextPoint, starting from the current point.

But to avoid infinite loops, we need to ensure that when the next\_point is the leftmost, we stop.

So, let's see:

List.Generate(

() => leftmost,

(state) => state <> null and (state <> leftmost or state is the first iteration),

(state) =>

let next = FindNextPoint(state, points)

in if next == leftmost then null else next,

(state) => state

)

But this is a bit unclear.

Alternatively, perhaps use a helper list to track the hull.

Another approach is to use recursion with a function that accumulates the hull points.

But given the limitations of Power Query, perhaps the best way is to use a custom function.

Let me try to outline the code.

Full code steps:

1. Convert the source table to a list of records.

2. Find the leftmost point (min X, then min Y).

3. Initialize the hull list with the leftmost point.

4. Find the next points using the FindNextPoint function until we return to the leftmost.

Now, the code:

let

Source = #table(type table [X = number, Y = number], {

{10,13},

{4,10},

{8,12},

{15,13},

{8,14},

{5,4},

{12,14},

{11,1},

{4,13},

{11,6},

{8,8},

{12,14},

{2,5},

{3,6},

{12,10},

{7,7},

{10,8},

{8,8},

{12,10},

{4,6},

{11,18},

{11,15},

{9,14}

}),

points = Table.ToRecords(Source),

// Find the leftmost point (min X, then min Y)

sorted = List.Sort(points, (a, b) =>

if a[X] < b[X] then -1

else if a[X] > b[X] then 1

else if a[Y] < b[Y] then -1

else 1

),

leftmost = sorted{0},

// Function to find the next point in the convex hull

FindNextPoint = (current as record, allPoints as list) as record =>

let

filtered = List.Select(allPoints, each \_[X] <> current[X] or \_[Y] <> current[Y]),

initialCandidate = if List.IsEmpty(filtered) then null else filtered{0},

finalCandidate = List.Accumulate(

filtered,

initialCandidate,

(state, q) =>

let

cross = (state[X] - current[X])\*(q[Y] - current[Y]) - (state[Y] - current[Y])\*(q[X] - current[X]),

distanceState = Number.Power(state[X] - current[X], 2) + Number.Power(state[Y] - current[Y], 2),

distanceQ = Number.Power(q[X] - current[X], 2) + Number.Power(q[Y] - current[Y], 2)

in

if cross > 0 then

q

else if cross = 0 then

if distanceQ > distanceState then q else state

else

state

)

in

finalCandidate,

// Recursive function to build the hull

BuildHull = (current as record, hull as list) as list =>

let

nextPoint = FindNextPoint(current, points),

newHull = if nextPoint = hull{0} then hull else BuildHull(nextPoint, hull & {nextPoint})

in

newHull,

// Start building the hull

initialHull = {leftmost},

hullList = BuildHull(leftmost, initialHull),

// Convert the hull list to a table

hullTable = Table.FromRecords(hullList)

in

hullTable

Wait, but the recursive function BuildHull may hit recursion limits. For example, if the convex hull has many points, Power Query could throw a stack overflow error. But given that the sample data has 22 points, and the convex hull likely has around 10 points, this might work.

Testing the code with sample data.

Looking at the sample data, let's manually check a few points.

The leftmost point is (2,5) (X=2, Y=5), which is present in the data.

Then, the next point would be the one such that all other points are to the right of the line from (2,5) to nextPoint.

So, applying the FindNextPoint function.

For current = (2,5), initialCandidate is the first point in filtered list (which is all points except (2,5)). Let's see:

The filtered list is all points except (2,5). The initialCandidate is the first in the filtered list after sorting. Wait, in the code, the filtered list is List.Select(points, ...), which removes the current point, but the order of the list is the original order of the points. So initialCandidate is the first point in the original list that is not current.

Wait, in the code, after converting the points to a list of records, the sorted list is only used to find the leftmost. The 'points' variable is Table.ToRecords(Source), which preserves the original order. Then, in the FindNextPoint function, the 'filtered' list is all points except current. The initialCandidate is filtered{0}, which is the first point in the original list that is not current.

But this can be problematic because the order of the points affects which initial candidate is chosen, but in the List.Accumulate, all points are checked. So even if the initial candidate is not the correct one, the Accumulate will iterate through all points and update the candidate accordingly.

Thus, the order of the points in the filtered list may not affect the final candidate, as all points are considered.

So the code should work.

But when implementing this, there's a possible error in the cross product calculation. Let's recheck:

cross = (state.X - current.X)\*(q.Y - current.Y) - (state.Y - current.Y)\*(q.X - current.X)

This is correct. For vectors state - current and q - current.

If this cross product is positive, then q is more counter-clockwise than state. So we replace state with q.

If zero, we check the distance. The farther one is selected.

So the code should correctly find the next point.

Now, testing with sample data.

For example, the leftmost point is (2,5). The next point should be the next one in the convex hull. Let's see.

The next point in the convex hull after (2,5) is likely (3,6), but not sure. Or maybe (4,6), (4,10), etc.

But without manual calculation, it's hard to tell. However, the code is supposed to compute this correctly.

Another potential issue is handling colinear points. For example, if multiple points are colinear, the code should include the farthest ones and skip the intermediate ones.

But since the algorithm checks for colinear points and selects the farthest, this should be handled.

Once the hullList is built, it's converted to a table, which is the output.

Now, possible edge cases: all points colinear. Then the convex hull is the two endpoints and any points in between that are colinear but not in between.

But the code should handle that.

Another edge case: all points forming a convex shape. The code should select all points.

Another edge case: duplicates. But in the code, when selecting the next point, if two points are the same (duplicates), they would be considered as one.

But in the given data, are there duplicates? For example, (12,14) appears twice, (8,8) appears twice. But since the code uses Table.ToRecords, duplicates would be treated as separate records. But when the code checks for equality in List.Select, if two points have the same X and Y, they would be considered the same.

Wait, no. For example, if there are two points (8,8), when current is (8,8), the filtered list would exclude all points with X=8 and Y=8. But if there are multiple entries, the code would process them as separate points. However, the convex hull algorithm should not be affected by duplicates, as the code will process each occurrence. But in reality, duplicates don't contribute to the convex hull. So, if there are multiple points at the same location, they should be considered as a single point. So perhaps the code should first remove duplicate points.

In the given data, (12,14) appears twice, (8,8) appears twice. So those duplicates could be removed.

But the original problem statement says "extract the points that lie on the vertices of the smallest convex area". So if a point is present multiple times but is on the convex hull, should it appear multiple times in the output?

Probably not. So the code should first remove duplicates.

So, adding a step to remove duplicates.

In the code:

after converting to points, do:

points = Table.Distinct(Source)

But in the code above, the Source is the original table. So perhaps:

Source = ... (the original data),

points = Table.Distinct(Source),

...

But how is the original data read. If the data is read as a table, then Table.Distinct would remove duplicate rows.

Yes, that's important. Because in the given data, duplicates like (12,14) and (8,8) would be considered as the same point. So we need to deduplicate first.

So modifying the code:

let

Source = ...,

// Remove duplicate points

DistinctSource = Table.Distinct(Source),

points = Table.ToRecords(DistinctSource),

... rest of the code ...

]

This way, duplicate points are removed before processing.

This is crucial, as otherwise, the code may process duplicates and include them in the hull, which would be incorrect.

So adding that step.

Another thing: the initial code may have leftmost as the first point, but after deduplication, it's possible that the data has fewer points.

Now, after deduplication, the code should correctly find the convex hull.

So putting it all together, the final code includes:

- Reading the data.

- Removing duplicates.

- Finding the leftmost point.

- Building the convex hull using Jarvis march.

Now, the final M code should be:

let

Source = #table(type table [X = number, Y = number], {

{10,13},

{4,10},

{8,12},

{15,13},

{8,14},

{5,4},

{12,14},

{11,1},

{4,13},

{11,6},

{8,8},

{12,14},

{2,5},

{3,6},

{12,10},

{7,7},

{10,8},

{8,8},

{12,10},

{4,6},

{11,18},

{11,15},

{9,14}

}),

// Remove duplicate points

DistinctSource = Table.Distinct(Source),

// Convert to list of records

points = Table.ToRecords(DistinctSource),

// Find the leftmost point (min X, then min Y)

sorted = List.Sort(points, (a, b) =>

if a[X] < b[X] then -1

else if a[X] > b[X] then 1

else if a[Y] < b[Y] then -1

else 1

),

leftmost = sorted{0},

// Function to find the next point in the convex hull

FindNextPoint = (current as record, allPoints as list) as record =>

let

// Filter out the current point

filtered = List.Select(allPoints, each \_[X] <> current[X] or \_[Y] <> current[Y]),

// If no points left, return null (should not happen)

initialCandidate = if List.IsEmpty(filtered) then null else filtered{0},

// Iterate through all points to find the correct next candidate

finalCandidate = List.Accumulate(

filtered,

initialCandidate,

(state, q) =>

let

// Compute the cross product

cross = (state[X] - current[X])\*(q[Y] - current[Y]) - (state[Y] - current[Y])\*(q[X] - current[X]),

// Compute distances squared for colinear points

distanceState = Number.Power(state[X] - current[X], 2) + Number.Power(state[Y] - current[Y], 2),

distanceQ = Number.Power(q[X] - current[X], 2) + Number.Power(q[Y] - current[Y], 2)

in

if cross > 0 then

q

else if cross = 0 then

// Choose the farther point

if distanceQ > distanceState then q else state

else

state

)

in

finalCandidate,

// Recursive function to build the hull

BuildHull = (current as record, hull as list) as list =>

let

nextPoint = FindNextPoint(current, points),

// Check if we've returned to the starting point

isDone = nextPoint[X] = hull{0}[X] and nextPoint[Y] = hull{0}[Y],

// Append nextPoint to hull unless it's the starting point

newHull = if isDone then hull else BuildHull(nextPoint, hull & {nextPoint})

in

newHull,

// Start building the hull

initialHull = {leftmost},

hullList = BuildHull(leftmost, initialHull),

// Convert the hull list to a table

hullTable = Table.FromRecords(hullList)

in

hullTable

This code should correctly compute the convex hull.

Testing it with the given data.

The leftmost point is (2,5). The next points are determined by the algorithm.

But to verify, perhaps the hull includes points like (2,5), (11,1), (15,13), (11,18), (9,14), etc. (This is just a guess.)

The final output table will have the points on the convex hull.

Note that the code removes duplicate points first, so the duplicate (12,14), (8,8), etc., are processed once.

Therefore, this should correctly compute the convex hull vertices.

To solve this problem, we need to determine the convex hull of a given set of points using the Gift Wrapping Algorithm (Jarvis March). The convex hull is the smallest convex polygon that can enclose all the given points. The algorithm starts with the leftmost point and iteratively selects the next point such that all other points lie to the right of the line formed by the current and next points.

**Approach**

1. **Remove Duplicates**: Ensure there are no duplicate points in the dataset.
2. **Identify the Leftmost Point**: The leftmost point (with the smallest X-coordinate, and smallest Y-coordinate in case of ties) is the starting point of the convex hull.
3. **Gift Wrapping Algorithm**: For each point on the convex hull, determine the next point such that all other points are to the right of the line formed by the current and next points. This is done using cross products to check the orientation of points relative to the line.
4. **Handle Collinear Points**: If multiple points are collinear with the current point, select the farthest one to ensure the hull is the smallest convex polygon.

**Solution Code**

powerquery

Copy

let

Source = #table(type table [X = number, Y = number], {

{10,13},

{4,10},

{8,12},

{15,13},

{8,14},

{5,4},

{12,14},

{11,1},

{4,13},

{11,6},

{8,8},

{12,14},

{2,5},

{3,6},

{12,10},

{7,7},

{10,8},

{8,8},

{12,10},

{4,6},

{11,18},

{11,15},

{9,14}

}),

// Remove duplicate points

DistinctSource = Table.Distinct(Source),

// Convert to list of records

points = Table.ToRecords(DistinctSource),

// Find the leftmost point (min X, then min Y)

sorted = List.Sort(points, (a, b) =>

if a[X] < b[X] then -1

else if a[X] > b[X] then 1

else if a[Y] < b[Y] then -1

else 1

),

leftmost = sorted{0},

// Function to find the next point in the convex hull

FindNextPoint = (current as record, allPoints as list) as record =>

let

// Filter out the current point

filtered = List.Select(allPoints, each \_[X] <> current[X] or \_[Y] <> current[Y]),

// If no points left, return null (should not happen)

initialCandidate = if List.IsEmpty(filtered) then null else filtered{0},

// Iterate through all points to find the correct next candidate

finalCandidate = List.Accumulate(

filtered,

initialCandidate,

(state, q) =>

let

// Compute the cross product

cross = (state[X] - current[X])\*(q[Y] - current[Y]) - (state[Y] - current[Y])\*(q[X] - current[X]),

// Compute distances squared for colinear points

distanceState = Number.Power(state[X] - current[X], 2) + Number.Power(state[Y] - current[Y], 2),

distanceQ = Number.Power(q[X] - current[X], 2) + Number.Power(q[Y] - current[Y], 2)

in

if cross > 0 then

q

else if cross = 0 then

// Choose the farther point

if distanceQ > distanceState then q else state

else

state

)

in

finalCandidate,

// Recursive function to build the hull

BuildHull = (current as record, hull as list) as list =>

let

nextPoint = FindNextPoint(current, points),

// Check if we've returned to the starting point

isDone = nextPoint[X] = hull{0}[X] and nextPoint[Y] = hull{0}[Y],

// Append nextPoint to hull unless it's the starting point

newHull = if isDone then hull else @BuildHull(nextPoint, hull & {nextPoint})

in

newHull,

// Start building the hull

initialHull = {leftmost},

hullList = BuildHull(leftmost, initialHull),

// Convert the hull list to a table

hullTable = Table.FromRecords(hullList)

in

hullTable

**Explanation**

1. **Removing Duplicates**: The Table.Distinct function ensures each point is unique.
2. **Finding the Leftmost Point**: Sorting the points by X and then Y coordinates gives the leftmost point as the starting point of the convex hull.
3. **Gift Wrapping Algorithm**: The FindNextPoint function uses cross products to determine the orientation of points relative to the current line segment. Points to the right are ignored, while points to the left or collinear (farthest ones) are considered.
4. **Building the Hull**: The recursive BuildHull function iteratively finds the next point on the hull until it returns to the starting point, forming the convex hull.

This approach efficiently computes the convex hull using the Gift Wrapping Algorithm, ensuring all points on the hull are included in the smallest convex polygon.

New chat

DeepThink (R1)

Search

AI-generated, for reference only