1 Description the exact algorithm

The algorithm works in a Dijkstra like fashion, it means that it computes shortest paths by using a priority queue, ie at each step it chooses the closest point to compute the distance to its neighbors. But here it is not that simple: we first have to choose what objects we are propagating. We choose windows which are a distance field over an edge associated to a pseudosource. Here is the pseudocode for the algorithm.

Input : Graph **G** and source **s**
Output: For each halfedge, a TreeSet of windows
for $v \in Neighbors(s)$ **do**
  $e \longleftarrow edge(u, v)$
  $l \longleftarrow length(e)$
  $Q \longleftarrow Window(0, l, 0, l, e, \sigma = 0)$
end
while $Q \neq \emptyset$ **do**
  $w \longleftarrow Q.pop$
  $e_1 = w.e.next.opposite$
  $e_2 = w.e.next.next.opposite$
  for $i \longleftarrow 1$ **to** 2 **do**
    W_i = propagation of w on e_i
    $w_i First$ = leftmost window on e_i to intersect with W_i on e_i
    $w_i Last$ = rightmost window on e_i to intersect with W_i on e_i
    for w' *on* e_i *with* $w' >= w_i First$ *and* $w' <= w_i Last$ **do**
      if W_i *"wins the fight" on an interval* $\neq \emptyset$ **then**
        Update w'
        Add W_i with correct parameters
        $Q \longleftarrow W_i$
      end
    end
  end
end

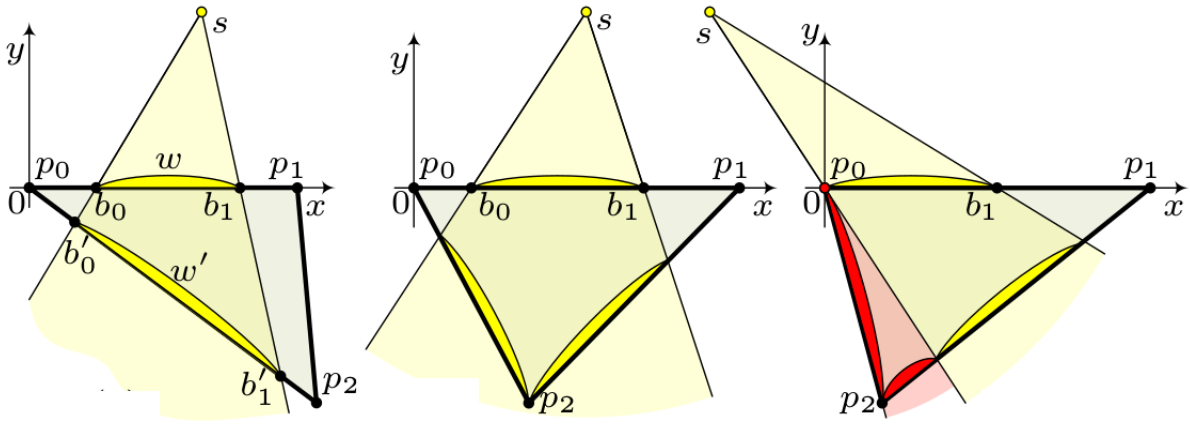
Algorithm 1: Exact Algorithm for computing shortest paths

Some operations still have to be explained a bit more accurately. The first one is :

$$Q \longleftarrow Window(e, 0, l, 0, l, \sigma = 0)$$

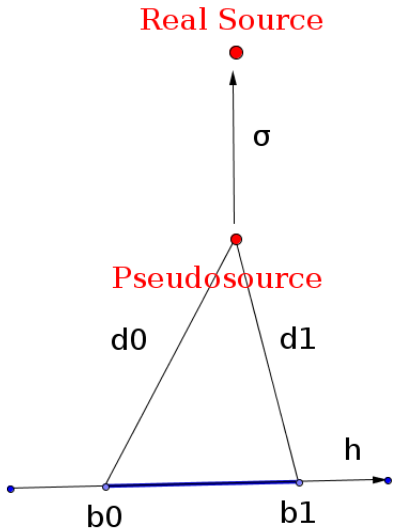
Here Q is a priority queue containing windows with the order : $w_1 \leq w_2$ if and only if the minimal distance from w_1 to its pseudosource plus the distance from the pseudosource to the real source is $\leq$ to the same real number for w_2.

Then we pick the "closest" window according to our distance and we propagate it over the two opposite halfedges (orientation is important to know on which side of the edge is the pseudosource). We have these three (plus two with symetries) cases :



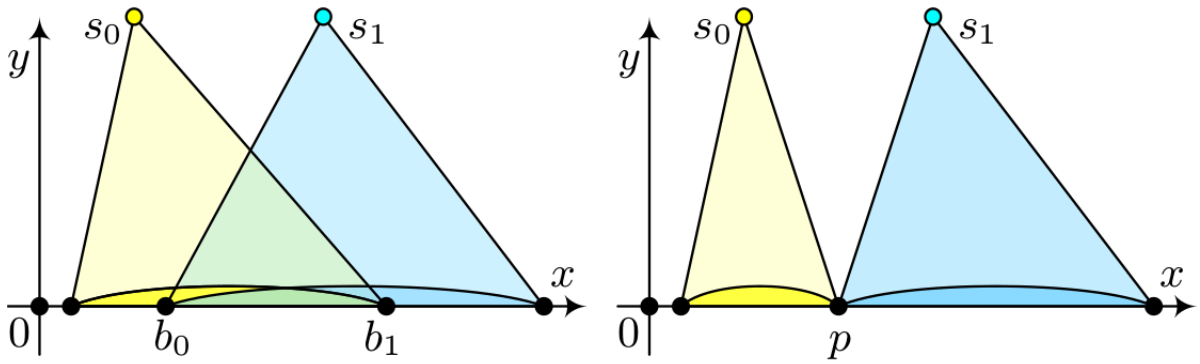
The yellow line is the propagated window and the red ones are the propagated windows when the blue segment touches one of the edge ends. These new windows have the red dot as a new pseudosource.

The orientation of halfedges is important. By convention, we decided that the source was on your left when you are oriented like the halfedge (for instance, in the five example the top halfedge with the blue segment is oriented from left to right). Therefore a window is described as a six-tuple: $w = (d_0, d_1, b_0, b_1, h, \sigma)$
Where d_0, d_1, b_0, b_1, σ are defined like the article suggested. We only use h the halfedge instead of τ the orientation because every time we pop a window out of Q we need to know where it is in the graph.



Then we need to explain : "$w_i First$ = leftmost window on e_i to intersect with W_i on e_i"
This sentence implies two things : given an edge, we can find the windows on it and we can perform a quick search on these windows to find the right ones. Therefore we chose to implement that with an **ArrayList of TreeSets**. The ArrayList is indexed by halfedges' indexes and TreeSets contain the windows on one halfedge.
But we still need to specify an order on the windows to be able to perform a binary search. We chose to sort the windows according to their b_0. This is ok because at every iteration of the while loop, windows on each edge are not overlapping so we have for instance $w_i First$ as the last window whose b_0 is $\leq$ to the propagated window's b_0.
Therefore we have $O(log(n))$ for inserting, removing or searching operations.

Finally, we can explain "W_i "wins the fight" on an interval $\neq \emptyset$"
This means that we need to compute intervals on which the new propagated window and old windows are overlapping. If they are, we then need to compute the interval on which the new window offers a shorter path and the interval on which the old one is better (ie we have to find the right p on the figure).
We can do this in $O(1)$ with geometry formulas.



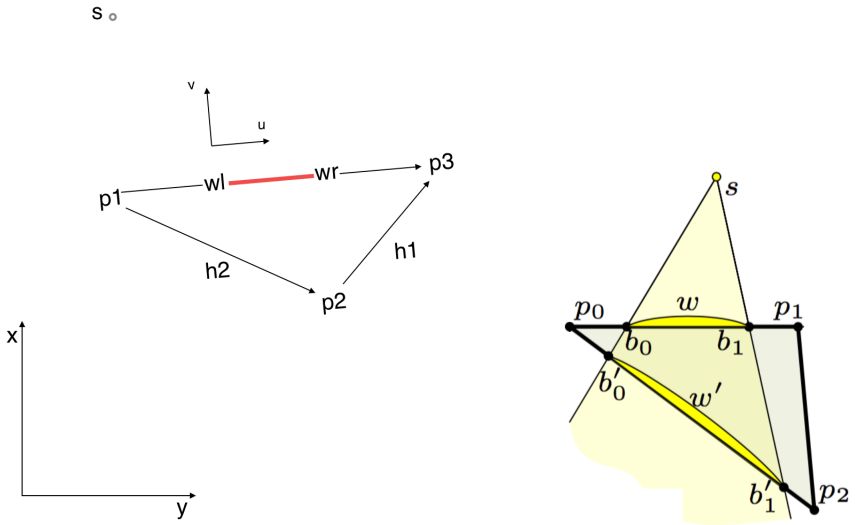
During this step we have to be careful with the orientation of the edges because if the propagated window goes downwards we don't want to compare it with windows going upwards.

Finally, we push something in the priority queue Q if and only if some distances have been updated.

2 Details about our implementation and subtle problems

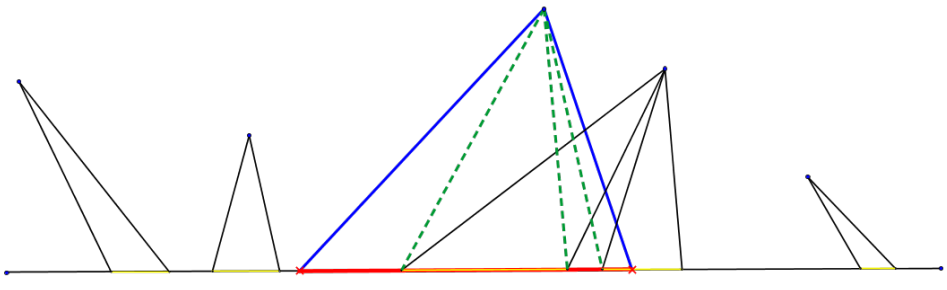
The theory is beautiful but practice is another story.

At every step, when we pick a window and we propagate it, we need to compute the propagated windows on the opposite edges. In order to avoid heavy formulas, we decided to implement a class Rotation which simply computes the rotation that would rotate the normal vector of the considered face to have it colinear to (Oz). With this trick we could just ignore the z coordinate to find opposite windows so we are in this case. And the function FindIntersection enables us to find b'_0 and b'_1.



Once we got these opposite windows, we have to compare them to windows already existing. So we used the function tailset of TreeSet that gives the highest element lower than a given element.

So we have to cut the propagated windows in order to match already existing windows ends. Here already existing windows are in yellow and the new one in red. It is split in several windows (green) to match yellow ends.



Once it is done we then compute the intersection for each overlap and we update both the TreeSet and the PriorityQueue. But how ?

3 What about complexity ?

The article argues that according to another result there are at most $O(n)$ windows per edge so $O(n^2)$ windows in total. Therefore complexity is $O(n^2 log(n))$ because of the binary search.

But here we can visit a window more than once and we change at every step windows.