Python program for blobtrees

Listing 1: Initialization

```
from collections import defaultdict
import math
import draw_ipe
from random import random
import sys

"""The points are (x[0],y[0]) ...  (x[n-1],y[n-1]).
The drawing programm assumes that they are in the unit square [0, 1]x[0, 1].
"""

beta = 1.0 # cost of tree-edges is multiplied by β

TRACE = False

n=23
x=[random() for _ in range(n)]
y=[random() for _ in range(n)]
if 0: # random points in two clusters:
    x=[random()*0.4 for _ in range(n)] + [random()*0.4+0.5 for _ in range(n)]
    y=[random()*0.3 for _ in range(n)] + [random()*0.4+0.6 for _ in range(n)]
    n=2*n
if len(sys.argv)>1: # one filename parameter of a .ipe file
    import read_ipe
    x,y = read_ipe.read_ipe(sys.argv[1])
    n = len(x)

for i in range(n): print(f"{i}: ({x[i]:6.4f}, {y[i]:6.4f}),")
```

Listing 2: Geometric primitives

```
def orientation(i,j,k): # returns 0 if two points are equal
    return (x[j]-x[i])*(y[k]-y[i]) - (x[k]-x[i])*(y[j]-y[i])
def crosses(a,b,u,v):
    """do the segments ab and uv cross?"""
    return (len({a,b,u,v}) ≡ 4 and
            (orientation(a,b,u)>0) != (orientation(a,b,v)>0) and
            (orientation(u,v,a)>0) != (orientation(u,v,b)>0))
def distance(a,b):
    return math.sqrt((x[a]-x[b])**2 + (y[a]-y[b])**2)
def weighted_distance(u,v,s):
    if s: # tree edge
        return distance(u,v)*beta
    return distance(u,v) # blob edge

##### ASSUME GENERAL POSITION THROUGHOUT!
# checking safety of the data:
min_det = min(abs(orientation(i,j,k))
              for i in range(n)
              for j in range(i)
              for k in range(j))
print("Degeneracy check: smallest determinant for orientation test =",min_det)
if min_det<1e-12:
    print("Smallest determinant is dangerously small.")
    print("Aborting.")
    raise ValueError

y_x = [(y[i],x[i]) for i in range(n)] # for lexicographic vertical comparison
# Equal x-coordinates matter for the L/R-division below A.
# We (arbitrarily) assign points with x[u]==x[A] to the LEFT side.
```

Listing 3: Minimum spanning tree

```
"""Compute MST with the LOWEST point as the root.

succ[i] is the neighbor of i on the path towards the root.
succ[root] == None.
children[i] = list of children
"""

# Prim-Dijkstra algorithm, O(n^2) time
pointlist = [] # ordered list of non-root vertices s.t. succ[i] is always before i
Unfinished = set(range(n))
_,root = min((y_x[i],i) for i in range(n)) # lex. min
d = [(distance(i,root),root) for i in range(n)]
# d is a list of pairs (dist, predecessor)
succ=[0]*n
succ[root]=None
Unfinished.remove(root)
while Unfinished:
    (_,succ_u),u = min((d[i],i) for i in Unfinished)
    pointlist.append(u)
    succ[u]=succ_u
    Unfinished.remove(u)
    for i in Unfinished:
        d_new = distance(u,i)
        if d_new < d[i][0]:
            d[i] = (d_new,u)

MST_cost = sum(distance(i,j) for i,j in enumerate(succ) if j is not None)
print("MST cost =",MST_cost)

# now collect lists of children
children = [[] for u in range(n)]
for u,v in enumerate(succ):
    if v is not None:
        children[v].append(u)

# compute sizes of edge problems:
# subtree_size[u] is the size of the subtree rooted at u.
# It is the size of the EDGE PROBLEM associated with the edge (u,succ[u])
subtree_size = [1]*n
for u in reversed(pointlist):
    subtree_size[succ[u]] += subtree_size[u]
assert subtree_size[root] ≡ n

# buckets for edge problems (plus the root problem)
edge_problems = [[] for size in range(n+1)]
for u in range(n):
    edge_problems[subtree_size[u]].append(u)

print(edge_problems,subtree_size)
```

Listing 4: Preprocessing in $O(n^3)$ time

```
# for each tree edge uv, crossed_wall_problems[u,v] is the list of
# walls (segments) BC for which uv is the only exiting edge.
# uv crosses BC from left to right.
crossed_wall_problems = [[] for size in range(n)]
# Or the exiting edge leaves from a or b:
uncrossed_wall_problems = [[] for size in range(n)]
root_wall_problems = []

# buckets for walls whose weights of crossing MST edges have been accumulated
accumulate_wall_problems = [[] for size in range(n)]

# optimal solutions for edge problem:
```

```
best_edge_value = [None]*(n+1)
best_edge_sol = [None]*(n+1) # for recovering the solution

# optimal solutions for chord problems:
best_chord_value = dict()
best_chord_sol = dict()

cross_left_to_right = defaultdict(list)
# cross_left_to_right[a,b] = list of (startpoints of) all MST edges
# that cross ab from left to right.

chord_problems = [[] for _ in range(n)]
# list of chords (a,b) of appropriate size,

frontside = dict()
# frontside='L' or 'R' is the side containing the root, for every valid chord ab
# frontside='L': left-facing chord
# frontside=None: indicate that no feasible solution exists with this chord.

for A in range(n):
    for B in range(n):
        if y_x[B]<y_x[A]: # lexicographic comparison
            continue
        # Now A is lower than B.
        for u in range(n):
            v = succ[u]
            if v is not None and crosses (u,v, A,B):
                if orientation(A,B,u)>0:
                    cross_left_to_right[A,B].append(u)
                else:
                    cross_left_to_right[B,A].append(u)

for A in range(n):
    for B in range(n):
        if A ≡ B:
            continue
        accumulated_size = sum(
            subtree_size[u] for u in cross_left_to_right[A,B])
        if accumulated_size<n:
            accumulate_wall_problems[accumulated_size].append((A,B))
        if len(cross_left_to_right[A,B]) ≡ 1:
            [u] = cross_left_to_right[A,B]
            crossed_wall_problems[u].append((A,B))
        elif len(cross_left_to_right[A,B]) ≡ 0:
            # tree exit edges from a corner without tree edge crossing.
            # if the exit edge is (A,succ[A]) or (B,succ[B]) it must
            # be the edge with the larger size.
            u = A if subtree_size[A] > subtree_size[B] else B
            uncrossed_wall_problems[u].append((A,B))
            if B ≡ root:
                root_wall_problems.append((A,B))
```

Listing 5: Auxiliary procedures

```
def edge_out_of_left_corner(a,b,c,v):
    return v not in (None,a,b,c) and orientation(a,c,v)<0 and orientation(c,b,v)>0

def edge_out_of_right_corner(a,b,c,v):
    return v not in (None,a,b,c) and orientation(a,b,v)>0 and orientation(b,c,v)<0

def is_exit_triangle(A,B,C):
    """Is ABC a potential exit triangle? Check some necessary conditions"""
    return (orientation(A,B,C) > 0 and
            y_x[A] < min(y_x[B],y_x[C]) and A!=root and
            frontside.get((A,B)) ≡ 'L' and frontside.get((A,C)) ≡ 'R')
```

```

def process_triangle(a,b,c):
    """the cost associated with exit triangle abc, not including the
    exiting edge and the costs of the left/right chord subproblems"""
    incoming_costs = sum(best_edge_value[u] for u in children[c]
                    if edge_out_of_left_corner(a,b,c,u)) + sum(
                  best_edge_value[u] for u in children[b]
                    if edge_out_of_right_corner(a,b,c,u) )
    return distance(b,c) + accumulated_crossings[c,b] + incoming_costs

def process_left_or_right_digon(A,B,side):
    """the cost associated with exit digon AB, not including the exiting edge

    A is the lower point, AB is on the left or right boundary of the blob.
    side = 'left' or 'right' accordingly."""
    if side ≡ 'left':
        l,r = A,B
    else:
        l,r = B,A
    # l,r is the edge in the clockwise direction around the blob

    incoming_costs = sum(best_edge_value[u] for u in children[B]
              if in_upper_left_or_right_wedge(A,B,u, side)) + sum(
                     best_edge_value[u] for u in children[A]
              if in_lower_left_or_right_wedge(A,B,u, side))
    return ( best_chord_value[A,B] + distance(A,B)
             + accumulated_crossings[l,r] + incoming_costs )

### Consider all potential chords (a, b) and determine their frontside and size: ###

def in_lower_left_or_right_wedge(A,B,u, side):
    """u is in the 'left' or 'right' (as determined by side) angular region around A
    when A is the lowest point"""
    if u in (None,B):
        return False
    if y_x[u] > y_x[A]: # lexicographic comparison
        return (orientation(A,B,u)>0)  ≡ (side ≡ 'left')
    else:
        return (x[u]≤ x[A])  ≡ (side ≡ 'left')

def in_upper_left_or_right_wedge(A,B,u, side):
    if u in (None,A):
        return False
    return (orientation(A,B,u)>0)  ≡ (side ≡ 'left')
```

Listing 6: Labeling of left and right components

```
class Invalid(Exception):
    """Indicate a conflicting label assignment"""
    pass

def labelside(i,s):
    """ s='L' or 'R'.
    Raises exception in case of conflict.
    Otherwise returns True if i was already labeled. """
    assert s in 'LR'
    if sidelabel[i] is None:
        sidelabel[i]=s
        return False
    elif sidelabel[i]!=s:
        raise Invalid()
    return True

for A in range(n):
    for B in range(n):
```

```
        if y_x[B] ≤ y_x[A]: # lexicographic comparison
            continue
        # Now A is below B.
        try:
            sidelabel = [None]*n
            for u in range(n):
                v = succ[u]
                if v is None or {u,v} ≡ {A,B}:
                    continue
                if v in (A,B) or u in (A,B):
                    # one common vertex
                    if v in (A,B):
                        u,v = v,u
                    # Now u ≡ A or u ≡ B
                    if u ≡ B or y[v] > y[A]: # lexicographic comparison not required
                        if orientation(A,B,v)>0:
                            labelside(v,'L')
                        else:
                            labelside(v,'R')
                    else: # u ≡ A and v is below A
                        if x[v] ≤ x[A]:
                            labelside(v,'L')
                        else:
                            labelside(v,'R')
                    continue
                elif crosses(u,v,A,B):
                    if orientation(A,B,u)>0:
                        labelside(u,'L')
                        labelside(v,'R')
                    else:
                        labelside(u,'R')
                        labelside(v,'L')
            # All endpoints of MST edges crossed by AB or incident to AB have
            # been labeled 'L' or 'R'.

            # Now propagate labels up the tree:
            for u in range(n):
                s = sidelabel[u]
                if s is None:
                    continue
                v = succ[u]
                while True:
                    if v is None:
                        root_label = s
                        break
                    if v in (A,B) or crosses(u,v,A,B):
                        break
                    if labelside(v,s):
                        break
                    u,v = v,succ[v]
            # Now propagate labels "down" the tree:
            # search upward from each node to the nearest labeled point
            assert root in (A,B) or sidelabel[root]
            for u in range(n):
                if u!=A and u!=B and sidelabel[u] is None:
                    v = u
                    while sidelabel[v] is None:
                        v = succ[v]
                    s = sidelabel[v]
                    v = u
                    while not labelside(v,s):
                        v = succ[v]
            if A ≡ root:
                root_label = 'L'
                # If A is the root, it is arbitarily assigned to the left side.
```

```
            # store the frontside, indicating that this is a valid chord.
            frontside[A,B] = root_label
            # Now count vertices on the opposite side of the root:
            size = len([u for u in range(n)
                        if u!=A and u!=B and sidelabel[u] != root_label])
            # put the chord problem in the correct bucket:
            chord_problems[size].append((A,B))

        except Invalid:
            pass

print(f"{frontside=}");print(f"{chord_problems=}")
```

Listing 7: Draw a picture in an ipe-file

```
if 1:
  draw_ipe.open_ipe()
  for i in range(0,min(n-1,10,1),2): # a few random edges
    draw_ipe.start_page()
    draw_ipe.start_frame()
    draw_ipe.draw_tree(x,y,succ)
    if y_x[i+1] > y_x[i]:
        a,b = i,i+1
    else:
        a,b = i+1,i
    draw_ipe.draw_edge(x,y,a,b,'red')
    draw_ipe.end_frame()
    for s,t in enumerate(chord_problems):
        if (a,b) in t:
            draw_ipe.put_text(f"{frontside[a,b]=} size={s}")
            break
    else:
        draw_ipe.put_text("invalid")
    draw_ipe.end_page()
  draw_ipe.close_ipe()
```

Listing 8: Solve an edge problem

```
def process_edge_problem(u,v):
    print(f"edge ({u},{v})")

    # Case 1: u is not in a blob, and all incoming MST edges are used.
    best_u = sum(best_edge_value[v] for v in children[u])
    best_sol = "Tree",u

    for (b,c) in crossed_wall_problems[u]:
        # uv is the crossing exit edge of a triangle abc
        for a in range(n):
            if (is_exit_triangle(a,b,c) and
                  not edge_out_of_left_corner(a,b,c,succ[c]) and
                  not edge_out_of_right_corner(a,b,c,succ[b])):
                value = (process_triangle(a,b,c) +
                      best_chord_value[a,b] + best_chord_value[a,c])
                if value < best_u:
                    best_u = value
                    best_sol = "Exit_triangle",(a,b,c)
        # try if uv can be the crossing exit edge of a digon bc.
        # tree edge uv crosses bc from left to right.
        if y_x[b]<y_x[c]:
            a,hi = b,c # bc can be a right exit digon
            side = 'right'
            if frontside.get((a,hi))!='R':
                continue
        else:
```

```
            a,hi = c,b # bc can be a left exit digon
            side = 'left'
            if frontside.get((a,hi))!='L':
                continue
        if a ≡ root:
            continue
        if in_upper_left_or_right_wedge(a,hi,succ[hi], side):
            continue
        if in_lower_left_or_right_wedge(a,hi,succ[a], side):
            continue
        value = process_left_or_right_digon(a,hi,side)
        if value < best_u:
            best_u = value
            best_sol = "Exit_triangle",(a,hi,side)

    for (b,c) in uncrossed_wall_problems[u]:
        # blob in on the left side of bc
        if u  ≡ c:
            # uv could be an edge out of c for a triangle abc:
            for a in range(n):
                if (is_exit_triangle(a,b,c) and
                  edge_out_of_left_corner(a,b,c,v) and
                  not edge_out_of_right_corner(a,b,c,succ[b])):
                    value = (process_triangle(a,b,c) +
                      best_chord_value[a,b] + best_chord_value[a,c])
                    if value < best_u:
                        best_u = value
                        best_sol = "Exit_triangle",(a,b,c)
        else: # u  ≡ b
            # uv could be an edge out of b for a triangle abc:
            for a in range(n):
                if (is_exit_triangle(a,b,c) and
                  edge_out_of_right_corner(a,b,c,v) and
                  not edge_out_of_left_corner(a,b,c,succ[c])):
                    value = (process_triangle(a,b,c) +
                      best_chord_value[a,b] + best_chord_value[a,c])
                    if value < best_u:
                        best_u = value
                        best_sol = "Exit_triangle",(a,b,c)
        if y_x[b]<y_x[c]:
            side = 'right' # right exit digon
            Lo,Hi = b,c
            if frontside.get((Lo,Hi)) != 'R':
                continue
        else:
            side = 'left' # left exit digon
            Lo,Hi = c,b
            if frontside.get((Lo,Hi)) != 'L':
                continue
        # uv could be an edge out of Hi for a left or right digon bc:
        if u ≡ Hi:
            if not in_upper_left_or_right_wedge(Lo,Hi,v, side):
                continue # uv should go out from Hi
            if in_lower_left_or_right_wedge(Lo,Hi,succ[Lo], side):
                continue # nothing should go out from Lo
        # uv could be an edge out of Lo for a left or right digon bc:
        else: # u ≡ Lo
            if in_upper_left_or_right_wedge(Lo,Hi,succ[Hi], side):
                continue # nothing should go out from Hi
            if not in_lower_left_or_right_wedge(Lo,Hi,v, side):
                continue # uv should go out from Lo
        value = process_left_or_right_digon(Lo,Hi,side)
        if value < best_u:
            best_u = value
            best_sol = "Exit_digon",(Lo,Hi,side)
```

```
    best_edge_value[u] = best_u + distance(u,v)*beta
    best_edge_sol[u] = best_sol
```

Listing 9: Solve a chord problem

```
def process_chord_problem(A,B_new):
    s = frontside[A,B_new] # != None at this point
    best_value = None
    for B_old in range(n):
        if y_x[B_old] ≤ y_x[A] or B_old ≡ B_new:
            # the tree root is also excluded. Can only move TOWARD the root.
            continue
        if frontside.get((A,B_old))!=s:
            continue
        if s ≡ 'L':
            B,C = B_old,B_new # right-to-left triangle ABC
        else:
            B,C = B_new,B_old # left-to-right triangle ABC
        # now ABC should be positively oriented (counterclockwise).
        if orientation(A,B,C)≤ 0:
            continue
        if (cross_left_to_right[B,C] or # exiting edge exists:
            edge_out_of_right_corner(A,B,C,succ[B]) or
            edge_out_of_left_corner(A,B,C,succ[C])):
            continue
        # triangle ABC:
        value = process_triangle(A,B,C) + best_chord_value[A,B_old]
        if best_value is None or value<best_value:
            best_value = value
            best_sol = "Triangle",B_old
    # non-exit digon AB ("starting" digon):
    B = B_new
    if s ≡ 'R': # blob lies on the right side of the digon
        side = 'left' # digon on the left of the blob
        l,r = A,B # clockwise order around the blob
    else:
        side = 'right'
        l,r = B,A
    if (not cross_left_to_right[r,l] and
    # check outgoing edges from A and B
      not in_upper_left_or_right_wedge(A,B,succ[B], side) and
      not in_lower_left_or_right_wedge(A,B,succ[A], side) ):
        value = (distance(A,B) + accumulated_crossings[l,r]
            + sum( best_edge_value[u] for u in children[B]
                 if in_upper_left_or_right_wedge(A,B,u, side))
            + sum( best_edge_value[u] for u in children[A]
                 if in_lower_left_or_right_wedge(A,B,u, side)) )
        if best_value is None or value<best_value:
            best_value = value
            best_sol = "Digon",side

    if best_value is not None:
        best_chord_value[A,B_new] = best_value
        best_chord_sol[A,B_new] = best_sol
    else: # This can happen,
        # for example if their is a tree path from A to B and it goes below A.
        print(f"*** setting {A,B_new} to None! *** {size=}")
        frontside[A,B_new] = None # indicate that the chord is not usable, although valid
```

Listing 10: Process the subproblems according to size

```
accumulated_crossings = dict()
# accumulated_crossings[A,B] = sum of opt. solutions for edges crossing AB from
# left to right
```

```
for size in range(n):
    print(f"*** {size=}")
    ## EDGE PROBLEMS ##
    for u in edge_problems[size]:
        v = succ[u]  # we know that v=succ[u] exists, because size<n.
        process_edge_problem(u,v)
    ## ACCUMULATE EDGES CROSSING A SIDE ##
    for a,b in accumulate_wall_problems[size]:
        accumulated_crossings[a,b] = sum(
            best_edge_value[u] for u in cross_left_to_right[a,b])
    ## CHORD PROBLEMS ##
    for (A,B_new) in chord_problems[size]:
        process_chord_problem(A,B_new)
```

Listing 11: Solve the root problem

```
# Case 1: root is not in a blob, and all incoming MST edges are used.
best_value = sum(best_edge_value[v] for v in children[root])
best_sol = "Tree",root

# Case 2: The root is the lower corner A of a left digon AB.
for B,A in root_wall_problems:
    # We already know: no MST edge crosses AB from right to left.
    # A ≡ root, below B
    if frontside.get((A,B)) != 'L':
        continue
    if in_upper_left_or_right_wedge(A,B,succ[B], 'left'):
        continue
    if 1:
        value = process_left_or_right_digon(A,B,'left')
        if value < best_value:
            best_value = value
            best_sol = "Exit_digon",(A,B,"left")
print(f"{best_value=} {best_sol=}")
```

Listing 12: Construct the solution by backtracking

```
def include_in_solution(a,b, text=""):
    global solution
    solution.append((a,b))
    if TRACE: print(" "*32,"sol",a,b, text)

def solution_triangle(A,B,C):
    """what comes in across the top edge and into B or C"""
    include_in_solution("blob edge",(B,C))
    for u in children[C]:
        if edge_out_of_left_corner(A,B,C,u):
            backtrack_sol_tree(u)
    for u in children[B]:
        if edge_out_of_right_corner(A,B,C,u):
            backtrack_sol_tree(u)
    for u in cross_left_to_right[C,B]:
        backtrack_sol_tree(u)

def backtrack_sol_tree(u):
    if succ[u] is not None:
        include_in_solution("tree edge", (u,succ[u]),
          f"{best_edge_value[u]}, len = {distance(u,succ[u])}")
    backtrack_sol(best_edge_sol[u])

depth = 0
def backtrack_sol(x):
    global depth
    if TRACE: print(" "*depth+"backstart:",x)
```

```
    depth += 1
    solution_type, data = x
    if solution_type ≡ "Exit_triangle":
        A,B,C = data
        if C in ('left','right'):
            side = C
            if side ≡ 'left':
                l,r = A,B
            else:
                l,r = B,A
            # (l,r) is in the clockwise direction around the blob
            include_in_solution("blob edge",(r,l))
            backtrack_sol(("Chord_"+side, (A,B)))
            for u in cross_left_to_right[l,r]:
                backtrack_sol_tree(u)
            for u in children[B]:
                if in_upper_left_or_right_wedge(A,B,u, side):
                    backtrack_sol_tree(u)
            for u in children[A]:
                if in_lower_left_or_right_wedge(A,B,u, side):
                    backtrack_sol_tree(u)
        else:
            solution_triangle(A,B,C)
            backtrack_sol(("Chord_left",(A,B))) # left-facing chord
            backtrack_sol(("Chord_right",(A,C)))
    elif solution_type ≡ "Exit_digon":
        A,B,side = data # A below B
        if side ≡ 'left':
            l,r = A,B
        else:
            l,r = B,A
        include_in_solution("blob edge",(r,l))
        for u in cross_left_to_right[l,r]:
            backtrack_sol_tree(u)
        for u in children[B]:
            if in_upper_left_or_right_wedge(A,B,u, side):
                backtrack_sol_tree(u)
        for u in children[A]:
            if in_lower_left_or_right_wedge(A,B,u, side):
                backtrack_sol_tree(u)
        backtrack_sol(("Chord_"+side, (A,B)))
    elif solution_type ≡ "Chord_left": # intermediate type (not stored)
        A,B = data
        t2,data2 = best_chord_sol[A,B]
        #side = frontside[A,B]
        if TRACE: print(" "*depth,"- best = ",(t2,data2),
                        "value =", best_chord_value[A,B] )
        if t2 ≡ "Triangle":
            C = data2
            solution_triangle(A,C,B)
            backtrack_sol(("Chord_left",(A,C)))
        else: # t2 ≡ "Digon", data2 is redundant
            include_in_solution("blob edge",(A,B))
            for u in cross_left_to_right[B,A]:
                backtrack_sol_tree(u)
            for u in children[B]:
                if in_upper_left_or_right_wedge(A,B,u, 'right'):
                    backtrack_sol_tree(u)
            for u in children[A]:
                if in_lower_left_or_right_wedge(A,B,u, 'right'):
                    backtrack_sol_tree(u)
    elif solution_type ≡ "Chord_right":
        A,B = data
        t2,data2 = best_chord_sol[A,B]
        if TRACE: print(" "*depth,"- best = ",(t2,data2),
```

```
                        "value =", best_chord_value[A,B] )
        if t2 ≡ "Triangle":
            C = data2
            solution_triangle(A,B,C)
            backtrack_sol(("Chord_right",(A,C)))
        else: # t2 ≡ ”Digon”, data2 is redundant
            include_in_solution("blob edge",(B,A))
            for u in cross_left_to_right[A,B]:
                backtrack_sol_tree(u)
            for u in children[B]:
                if in_upper_left_or_right_wedge(A,B,u, 'left'):
                    backtrack_sol_tree(u)
            for u in children[A]:
                if in_lower_left_or_right_wedge(A,B,u, 'left'):
                    backtrack_sol_tree(u)
    elif solution_type ≡ "Tree":
        v = data
        for u in children[v]:
            backtrack_sol_tree(u)
    else:
        raise ValueError

    depth -= 1
    if TRACE: print(" "*depth+"backend:",x)
```

Listing 13: Show the solution and check its value

```
def clean_underscore(s):
    return "".join((x if x != '_' else r'\_') for x in str(s))

val = 0
solution = []
backtrack_sol(best_sol)
for s,(a,b) in solution:
    d = distance(a,b)
    val += weighted_distance(a,b, s ≡ "tree edge")
    print(s,(a,b), f"{d:5.3f}", "-----" if s ≡ "tree edge" else "")
print("total length", val, "**DISCREPANCY**" if abs(val-best_value)>1e-10 else "")
print(f"{best_value=} {best_sol=} {MST_cost=}")
```

Listing 14: Draw the solution picture in the ipe-file

```
draw_ipe.open_ipe("blobsol.ipe")
draw_ipe.start_page()
draw_ipe.start_frame()
for s,(i,j) in solution:
    draw_ipe.draw_edge(x,y,i,j, 'red' if s ≡ "blob edge" else 'blue',
                                extras = ' pen="fat"' )
draw_ipe.draw_tree(x,y,succ,point_labels = True)
draw_ipe.end_frame()
draw_ipe.put_text(f"optimum solution")
draw_ipe.end_page()
for i in range(0,min(n-1,20),2): # a few random subproblems
    #### some chord problem solution #####
    draw_ipe.start_page()
    draw_ipe.start_frame()
    draw_ipe.draw_tree(x,y,succ)
    if y_x[i+1]>y_x[i]:
        a,b = i,i+1
    else:
        a,b = i+1,i
    draw_ipe.draw_edge(x,y,a,b,'red',extras = ' pen="heavier" dash="dashed" ' )
    sol = best_chord_sol.get((a,b))
    val = 0
    if sol:
```

```
        if frontside[a,b] == 'L':
            sol = ("Chord_left",(a,b))
        else:
            sol = ("Chord_right",(a,b))
        solution = []
        backtrack_sol(sol)
        for s,(u,v) in solution:
            val += weighted_distance(u,v, s == "tree edge")
            draw_ipe.draw_edge(x,y,u,v,'red' if s == "blob edge" else 'blue',
                               extras = ' pen="fat"' )
    draw_ipe.end_frame()
    for s,t in enumerate(chord_problems):
        if (a,b) in t:
            draw_ipe.put_text(
                f"Chord {(a,b)=}, {frontside[a,b]=}, subproblem size={s}, "
                + ("no solution" if (a,b) not in best_chord_sol else
                     f"cost={best_chord_value[a,b]:5.4f}, " +
                     f"total length={val:5.4f}, "+
                     f"solution={clean_underscore(best_chord_sol[a,b])}"))
            break
    else:
        draw_ipe.put_text(f"{a,b} invalid chord")
    draw_ipe.end_page()

    #### some edge problem solution ####
    j = succ[i]
    if j is None:
        continue
    draw_ipe.start_page()
    draw_ipe.start_frame()
    draw_ipe.draw_tree(x,y,succ)
    draw_ipe.draw_edge(x,y,i,j,'blue',extras = ' pen="ultrafat" dash="dashed" ' )
    sol = best_edge_sol[i]
    val = distance(i,j)*beta
    solution = []
    backtrack_sol(sol)
    for s,(u,v) in solution:
        val += weighted_distance(u,v,s == "tree edge")
        draw_ipe.draw_edge(x,y,u,v,'red' if s == "blob edge" else 'blue',
                           extras = ' pen="fat"' )
    draw_ipe.end_frame()
    draw_ipe.put_text(f"Tree edge {(i,j)}, subproblem size={subtree_size[i]}, " +
             f"cost={best_edge_value[i]:5.4f}" +
             f", total length={val:5.4f}, solution={clean_underscore(sol)}")
    draw_ipe.end_page()
draw_ipe.close_ipe()
```