

Python program for blobtrees

Listing 1: Initialization

```
5 from collections import defaultdict
6 import math
7 import draw_ipe
8 from random import random
9 import sys
10
11 """The points are (x[0],y[0]) ... (x[n-1],y[n-1]).
12 The drawing programm assumes that they are in the unit square [0,1]x[0,1].
13 """
14
15 beta = 1.0 # cost of tree-edges is multiplied by  $\beta$ 
16
17 TRACE = False
18
19 n=23
20 x=[random() for _ in range(n)]
21 y=[random() for _ in range(n)]
22 if 0: # random points in two clusters:
23     x=[random()*0.4 for _ in range(n)] + [random()*0.4+0.5 for _ in range(n)]
24     y=[random()*0.3 for _ in range(n)] + [random()*0.4+0.6 for _ in range(n)]
25     n=2*n
26 if len(sys.argv)>1: # one filename parameter of a .ipe file
27     import read_ipe
28     x,y = read_ipe.read_ipe(sys.argv[1])
29     n = len(x)
30
31 for i in range(n): print(f"{i}: ({x[i]:6.4f}, {y[i]:6.4f}),")
```

Listing 2: Geometric primitives

```
37 def orientation(i,j,k): # returns 0 if two points are equal
38     return (x[j]-x[i])*(y[k]-y[i]) - (x[k]-x[i])*(y[j]-y[i])
39 def crosses(a,b,u,v):
40     """do the segments ab and uv cross?"""
41     return (len({a,b,u,v}) == 4 and
42             (orientation(a,b,u)>0) != (orientation(a,b,v)>0) and
43             (orientation(u,v,a)>0) != (orientation(u,v,b)>0))
44 def distance(a,b):
45     return math.sqrt((x[a]-x[b])**2 + (y[a]-y[b])**2)
46 def weighted_distance(u,v,s):
47     if s: # tree edge
48         return distance(u,v)*beta
49     return distance(u,v) # blob edge
50
51 ##### ASSUME GENERAL POSITION THROUGHOUT!
52 # checking safety of the data:
53 min_det = min(abs(orientation(i,j,k))
54               for i in range(n)
55               for j in range(i)
56               for k in range(j))
57 print("Degeneracy check: smallest determinant for orientation test =",min_det)
58 if min_det<1e-12:
59     print("Smallest determinant is dangerously small.")
60     print("Aborting.")
61     raise ValueError
62
63 y_x = [(y[i],x[i]) for i in range(n)] # for lexicographic vertical comparison
64 # Equal x-coordinates matter for the L/R-division below A.
65 # We (arbitrarily) assign points with x[u]==x[A] to the LEFT side.
```

Listing 3: Minimum spanning tree

```

71 """Compute MST with the LOWEST point as the root.
72
73 succ[i] is the neighbor of i on the path towards the root.
74 succ[root] == None.
75 children[i] = list of children
76 """
77
78 # Prim-Dijkstra algorithm,  $O(n^2)$  time
79 pointlist = [] # ordered list of non-root vertices s.t. succ[i] is always before i
80 Unfinished = set(range(n))
81 _,root = min((y_x[i],i) for i in range(n)) # lex. min
82 d = [(distance(i,root),root) for i in range(n)]
83 # d is a list of pairs (dist, predecessor)
84 succ=[0]*n
85 succ[root]=None
86 Unfinished.remove(root)
87 while Unfinished:
88     (_,succ_u),u = min((d[i],i) for i in Unfinished)
89     pointlist.append(u)
90     succ[u]=succ_u
91     Unfinished.remove(u)
92     for i in Unfinished:
93         d_new = distance(u,i)
94         if d_new < d[i][0]:
95             d[i] = (d_new,u)
96
97 MST_cost = sum(distance(i,j) for i,j in enumerate(succ) if j is not None)
98 print("MST cost =",MST_cost)
99
100 # now collect lists of children
101 children = [[] for u in range(n)]
102 for u,v in enumerate(succ):
103     if v is not None:
104         children[v].append(u)
105
106 # compute sizes of edge problems:
107 # subtree_size[u] is the size of the subtree rooted at u.
108 # It is the size of the EDGE PROBLEM associated with the edge (u,succ[u])
109 subtree_size = [1]*n
110 for u in reversed(pointlist):
111     subtree_size[succ[u]] += subtree_size[u]
112 assert subtree_size[root] == n
113
114 # buckets for edge problems (plus the root problem)
115 edge_problems = [[] for size in range(n+1)]
116 for u in range(n):
117     edge_problems[subtree_size[u]].append(u)
118
119 print(edge_problems,subtree_size)

```

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

```

125 # for each tree edge uv, crossed_wall_problems[u,v] is the list of
126 # walls (segments) BC for which uv is the only exiting edge.
127 # uv crosses BC from left to right.
128 crossed_wall_problems = [[] for size in range(n)]
129 # Or the exiting edge leaves from a or b:
130 uncrossed_wall_problems = [[] for size in range(n)]
131 root_wall_problems = []
132
133 # buckets for walls whose weights of crossing MST edges have been accumulated
134 accumulate_wall_problems = [[] for size in range(n)]
135
136
137 # optimal solutions for edge problem:

```

```

138 best_edge_value = [None]*(n+1)
139 best_edge_sol = [None]*(n+1) # for recovering the solution
140
141 # optimal solutions for chord problems:
142 best_chord_value = dict()
143 best_chord_sol = dict()
144
145 cross_left_to_right = defaultdict(list)
146 # cross_left_to_right[a,b] = list of (startpoints of) all MST edges
147 # that cross ab from left to right.
148
149 chord_problems = [[] for _ in range(n)]
150 # list of chords (a,b) of appropriate size,
151
152 frontside = dict()
153 # frontside='L' or 'R' is the side containing the root, for every valid chord ab
154 # frontside='L': left-facing chord
155 # frontside=None: indicate that no feasible solution exists with this chord.
156
157 for A in range(n):
158     for B in range(n):
159         if y_x[B]<y_x[A]: # lexicographic comparison
160             continue
161         # Now A is lower than B.
162         for u in range(n):
163             v = succ[u]
164             if v is not None and crosses (u,v, A,B):
165                 if orientation(A,B,u)>0:
166                     cross_left_to_right[A,B].append(u)
167                 else:
168                     cross_left_to_right[B,A].append(u)
169
170 for A in range(n):
171     for B in range(n):
172         if A == B:
173             continue
174         accumulated_size = sum(
175             subtree_size[u] for u in cross_left_to_right[A,B])
176         if accumulated_size<n:
177             accumulate_wall_problems[accumulated_size].append((A,B))
178         if len(cross_left_to_right[A,B]) == 1:
179             [u] = cross_left_to_right[A,B]
180             crossed_wall_problems[u].append((A,B))
181         elif len(cross_left_to_right[A,B]) == 0:
182             # tree exit edges from a corner without tree edge crossing.
183             # if the exit edge is (A,succ[A]) or (B,succ[B]) it must
184             # be the edge with the larger size.
185             u = A if subtree_size[A] > subtree_size[B] else B
186             uncrossed_wall_problems[u].append((A,B))
187         if B == root:
188             root_wall_problems.append((A,B))

```

Listing 5: Auxiliary procedures

```

194 def edge_out_of_left_corner(a,b,c,v):
195     return v not in (None,a,b,c) and orientation(a,c,v)<0 and orientation(c,b,v)>0
196
197 def edge_out_of_right_corner(a,b,c,v):
198     return v not in (None,a,b,c) and orientation(a,b,v)>0 and orientation(b,c,v)<0
199
200 def is_exit_triangle(A,B,C):
201     """Is ABC a potential exit triangle? Check some necessary conditions"""
202     return (orientation(A,B,C) > 0 and
203             y_x[A] < min(y_x[B],y_x[C]) and A!=root and
204             frontside.get((A,B)) == 'L' and frontside.get((A,C)) == 'R')

```

```

205
206 def process_triangle(a,b,c):
207     """the cost associated with exit triangle abc, not including the
208     exiting edge and the costs of the left/right chord subproblems"""
209     incoming_costs = sum(best_edge_value[u] for u in children[c]
210                          if edge_out_of_left_corner(a,b,c,u)) + sum(
211                          best_edge_value[u] for u in children[b]
212                          if edge_out_of_right_corner(a,b,c,u) )
213     return distance(b,c) + accumulated_crossings[c,b] + incoming_costs
214
215 def process_left_or_right_digon(A,B,side):
216     """the cost associated with exit digon AB, not including the exiting edge
217
218     A is the lower point, AB is on the left or right boundary of the blob.
219     side = 'left' or 'right' accordingly."""
220     if side == 'left':
221         l,r = A,B
222     else:
223         l,r = B,A
224     # l,r is the edge in the clockwise direction around the blob
225
226     incoming_costs = sum(best_edge_value[u] for u in children[B]
227                          if in_upper_left_or_right_wedge(A,B,u, side)) + sum(
228                          best_edge_value[u] for u in children[A]
229                          if in_lower_left_or_right_wedge(A,B,u, side))
230     return ( best_chord_value[A,B] + distance(A,B)
231             + accumulated_crossings[l,r] + incoming_costs )
232
233 ### Consider all potential chords (a,b) and determine their frontside and size: ###
234
235 def in_lower_left_or_right_wedge(A,B,u, side):
236     """u is in the 'left' or 'right' (as determined by side) angular region around A
237     when A is the lowest point"""
238     if u in (None,B):
239         return False
240     if y_x[u] > y_x[A]: # lexicographic comparison
241         return (orientation(A,B,u)>0) == (side == 'left')
242     else:
243         return (x[u] ≤ x[A]) == (side == 'left')
244
245 def in_upper_left_or_right_wedge(A,B,u, side):
246     if u in (None,A):
247         return False
248     return (orientation(A,B,u)>0) == (side == 'left')

```

Listing 6: Labeling of left and right components

```

254 class Invalid(Exception):
255     """Indicate a conflicting label assignment"""
256     pass
257
258 def labelside(i,s):
259     """ s='L' or 'R'.
260     Raises exception in case of conflict.
261     Otherwise returns True if i was already labeled. """
262     assert s in 'LR'
263     if sidelabel[i] is None:
264         sidelabel[i]=s
265         return False
266     elif sidelabel[i]!=s:
267         raise Invalid()
268     return True
269
270 for A in range(n):
271     for B in range(n):

```

```

272     if y_x[B] ≤ y_x[A]: # lexicographic comparison
273         continue
274     # Now A is below B.
275     try:
276         sidelabel = [None]*n
277         for u in range(n):
278             v = succ[u]
279             if v is None or {u,v} == {A,B}:
280                 continue
281             if v in (A,B) or u in (A,B):
282                 # one common vertex
283                 if v in (A,B):
284                     u,v = v,u
285                 # Now u ≡ A or u ≡ B
286                 if u ≡ B or y[v] > y[A]: # lexicographic comparison not required
287                     if orientation(A,B,v)>0:
288                         labelside(v,'L')
289                     else:
290                         labelside(v,'R')
291                 else: # u ≡ A and v is below A
292                     if x[v] ≤ x[A]:
293                         labelside(v,'L')
294                     else:
295                         labelside(v,'R')
296                 continue
297             elif crosses(u,v,A,B):
298                 if orientation(A,B,u)>0:
299                     labelside(u,'L')
300                     labelside(v,'R')
301                 else:
302                     labelside(u,'R')
303                     labelside(v,'L')
304             # All endpoints of MST edges crossed by AB or incident to AB have
305             # been labeled 'L' or 'R'.
306
307         # Now propagate labels up the tree:
308         for u in range(n):
309             s = sidelabel[u]
310             if s is None:
311                 continue
312             v = succ[u]
313             while True:
314                 if v is None:
315                     root_label = s
316                     break
317                 if v in (A,B) or crosses(u,v,A,B):
318                     break
319                 if labelside(v,s):
320                     break
321                 u,v = v,succ[v]
322         # Now propagate labels "down" the tree:
323         # search upward from each node to the nearest labeled point
324         assert root in (A,B) or sidelabel[root]
325         for u in range(n):
326             if u!=A and u!=B and sidelabel[u] is None:
327                 v = u
328                 while sidelabel[v] is None:
329                     v = succ[v]
330                 s = sidelabel[v]
331                 v = u
332                 while not labelside(v,s):
333                     v = succ[v]
334         if A == root:
335             root_label = 'L'
336             # If A is the root, it is arbitrarily assigned to the left side.

```

```

337
338     # store the frontside, indicating that this is a valid chord.
339     frontside[A,B] = root_label
340     # Now count vertices on the opposite side of the root:
341     size = len([u for u in range(n)
342                if u!=A and u!=B and sidelabel[u] != root_label])
343     # put the chord problem in the correct bucket:
344     chord_problems[size].append((A,B))
345
346     except Invalid:
347         pass
348
349 print(f"{frontside=}");print(f"{chord_problems=}")

```

Listing 7: Draw a picture in an ipe-file

```

355 if 1:
356     draw_ipe.open_ipe()
357     for i in range(0,min(n-1,10,1),2): # a few random edges
358         draw_ipe.start_page()
359         draw_ipe.start_frame()
360         draw_ipe.draw_tree(x,y,succ)
361         if y_x[i+1] > y_x[i]:
362             a,b = i,i+1
363         else:
364             a,b = i+1,i
365         draw_ipe.draw_edge(x,y,a,b,'red')
366         draw_ipe.end_frame()
367         for s,t in enumerate(chord_problems):
368             if (a,b) in t:
369                 draw_ipe.put_text(f"{frontside[a,b]=} size={s}")
370                 break
371         else:
372             draw_ipe.put_text("invalid")
373         draw_ipe.end_page()
374     draw_ipe.close_ipe()

```

Listing 8: Solve an edge problem

```

380 def process_edge_problem(u,v):
381     print(f"edge ({u},{v})")
382
383     # Case 1: u is not in a blob, and all incoming MST edges are used.
384     best_u = sum(best_edge_value[v] for v in children[u])
385     best_sol = "Tree",u
386
387     for (b,c) in crossed_wall_problems[u]:
388         # uv is the crossing exit edge of a triangle abc
389         for a in range(n):
390             if (is_exit_triangle(a,b,c) and
391                 not edge_out_of_left_corner(a,b,c,succ[c]) and
392                 not edge_out_of_right_corner(a,b,c,succ[b])):
393                 value = (process_triangle(a,b,c) +
394                         best_chord_value[a,b] + best_chord_value[a,c])
395                 if value < best_u:
396                     best_u = value
397                     best_sol = "Exit_triangle", (a,b,c)
398         # try if uv can be the crossing exit edge of a digon bc.
399         # tree edge uv crosses bc from left to right.
400         if y_x[b] < y_x[c]:
401             a,hi = b,c # bc can be a right exit digon
402             side = 'right'
403             if frontside.get((a,hi)) != 'R':
404                 continue
405         else:

```

```

406         a,hi = c,b # bc can be a left exit digon
407         side = 'left'
408         if frontside.get((a,hi))!='L':
409             continue
410     if a == root:
411         continue
412     if in_upper_left_or_right_wedge(a,hi,succ[hi], side):
413         continue
414     if in_lower_left_or_right_wedge(a,hi,succ[a], side):
415         continue
416     value = process_left_or_right_digon(a,hi,side)
417     if value < best_u:
418         best_u = value
419         best_sol = "Exit_triangle",(a,hi,side)
420
421 for (b,c) in uncrossed_wall_problems[u]:
422     # blob in on the left side of bc
423     if u == c:
424         # uv could be an edge out of c for a triangle abc:
425         for a in range(n):
426             if (is_exit_triangle(a,b,c) and
427                 edge_out_of_left_corner(a,b,c,v) and
428                 not edge_out_of_right_corner(a,b,c,succ[b])):
429                 value = (process_triangle(a,b,c) +
430                         best_chord_value[a,b] + best_chord_value[a,c])
431                 if value < best_u:
432                     best_u = value
433                     best_sol = "Exit_triangle",(a,b,c)
434     else: # u == b
435         # uv could be an edge out of b for a triangle abc:
436         for a in range(n):
437             if (is_exit_triangle(a,b,c) and
438                 edge_out_of_right_corner(a,b,c,v) and
439                 not edge_out_of_left_corner(a,b,c,succ[c])):
440                 value = (process_triangle(a,b,c) +
441                         best_chord_value[a,b] + best_chord_value[a,c])
442                 if value < best_u:
443                     best_u = value
444                     best_sol = "Exit_triangle",(a,b,c)
445     if y_x[b]<y_x[c]:
446         side = 'right' # right exit digon
447         Lo,Hi = b,c
448         if frontside.get((Lo,Hi)) != 'R':
449             continue
450     else:
451         side = 'left' # left exit digon
452         Lo,Hi = c,b
453         if frontside.get((Lo,Hi)) != 'L':
454             continue
455     # uv could be an edge out of Hi for a left or right digon bc:
456     if u == Hi:
457         if not in_upper_left_or_right_wedge(Lo,Hi,v, side):
458             continue # uv should go out from Hi
459         if in_lower_left_or_right_wedge(Lo,Hi,succ[Lo], side):
460             continue # nothing should go out from Lo
461     # uv could be an edge out of Lo for a left or right digon bc:
462     else: # u == Lo
463         if in_upper_left_or_right_wedge(Lo,Hi,succ[Hi], side):
464             continue # nothing should go out from Hi
465         if not in_lower_left_or_right_wedge(Lo,Hi,v, side):
466             continue # uv should go out from Lo
467     value = process_left_or_right_digon(Lo,Hi,side)
468     if value < best_u:
469         best_u = value
470         best_sol = "Exit_digon",(Lo,Hi,side)

```

```

471     best_edge_value[u] = best_u + distance(u,v)*beta
472     best_edge_sol[u] = best_sol
473

```

Listing 9: Solve a chord problem

```

479 def process_chord_problem(A,B_new):
480     s = frontside[A,B_new] # != None at this point
481     best_value = None
482     for B_old in range(n):
483         if y_x[B_old] ≤ y_x[A] or B_old ≡ B_new:
484             # the tree root is also excluded. Can only move TOWARD the root.
485             continue
486         if frontside.get((A,B_old))!=s:
487             continue
488         if s == 'L':
489             B,C = B_old,B_new # right-to-left triangle ABC
490         else:
491             B,C = B_new,B_old # left-to-right triangle ABC
492             # now ABC should be positively oriented (counterclockwise).
493             if orientation(A,B,C) ≤ 0:
494                 continue
495             if (cross_left_to_right[B,C] or # exiting edge exists:
496                 edge_out_of_right_corner(A,B,C,succ[B]) or
497                 edge_out_of_left_corner(A,B,C,succ[C])):
498                 continue
499             # triangle ABC:
500             value = process_triangle(A,B,C) + best_chord_value[A,B_old]
501             if best_value is None or value<best_value:
502                 best_value = value
503                 best_sol = "Triangle",B_old
504             # non-exit digon AB ("starting" digon):
505             B = B_new
506             if s == 'R': # blob lies on the right side of the digon
507                 side = 'left' # digon on the left of the blob
508                 l,r = A,B # clockwise order around the blob
509             else:
510                 side = 'right'
511                 l,r = B,A
512             if (not cross_left_to_right[r,l] and
513                 # check outgoing edges from A and B
514                 not in_upper_left_or_right_wedge(A,B,succ[B], side) and
515                 not in_lower_left_or_right_wedge(A,B,succ[A], side)):
516                 value = (distance(A,B) + accumulated_crossings[l,r]
517                     + sum( best_edge_value[u] for u in children[B]
518                         if in_upper_left_or_right_wedge(A,B,u, side))
519                     + sum( best_edge_value[u] for u in children[A]
520                         if in_lower_left_or_right_wedge(A,B,u, side)) )
521                 if best_value is None or value<best_value:
522                     best_value = value
523                     best_sol = "Digon",side
524
525             if best_value is not None:
526                 best_chord_value[A,B_new] = best_value
527                 best_chord_sol[A,B_new] = best_sol
528             else: # This can happen,
529                 # for example if there is a tree path from A to B and it goes below A.
530                 print(f"*** setting {A,B_new} to None! *** {size=}")
531                 frontside[A,B_new] = None # indicate that the chord is not usable, although valid

```

Listing 10: Process the subproblems according to size

```

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

```



```

540
541 for size in range(n):
542     print(f"*** {size=}")
543     ## EDGE PROBLEMS ##
544     for u in edge_problems[size]:
545         v = succ[u] # we know that v=succ[u] exists, because size<n.
546         process_edge_problem(u,v)
547     ## ACCUMULATE EDGES CROSSING A SIDE ##
548     for a,b in accumulate_wall_problems[size]:
549         accumulated_crossings[a,b] = sum(
550             best_edge_value[u] for u in cross_left_to_right[a,b])
551     ## CHORD PROBLEMS ##
552     for (A,B_new) in chord_problems[size]:
553         process_chord_problem(A,B_new)

```

Listing 11: Solve the root problem

```

559 # Case 1: root is not in a blob, and all incoming MST edges are used.
560 best_value = sum(best_edge_value[v] for v in children[root])
561 best_sol = "Tree",root
562
563 # Case 2: The root is the lower corner A of a left digon AB.
564 for B,A in root_wall_problems:
565     # We already know: no MST edge crosses AB from right to left.
566     # A ≡ root, below B
567     if frontside.get((A,B)) != 'L':
568         continue
569     if in_upper_left_or_right_wedge(A,B,succ[B], 'left'):
570         continue
571     if 1:
572         value = process_left_or_right_digon(A,B,'left')
573         if value < best_value:
574             best_value = value
575             best_sol = "Exit_digon", (A,B,"left")
576 print(f"{best_value=} {best_sol=}")

```

Listing 12: Construct the solution by backtracking

```

582 def include_in_solution(a,b, text=""):
583     global solution
584     solution.append((a,b))
585     if TRACE: print(" *32,\"sol\",a,b, text)
586
587 def solution_triangle(A,B,C):
588     """what comes in across the top edge and into B or C"""
589     include_in_solution("blob edge", (B,C))
590     for u in children[C]:
591         if edge_out_of_left_corner(A,B,C,u):
592             backtrack_sol_tree(u)
593     for u in children[B]:
594         if edge_out_of_right_corner(A,B,C,u):
595             backtrack_sol_tree(u)
596     for u in cross_left_to_right[C,B]:
597         backtrack_sol_tree(u)
598
599 def backtrack_sol_tree(u):
600     if succ[u] is not None:
601         include_in_solution("tree edge", (u,succ[u]),
602             f"{best_edge_value[u]}, len = {distance(u,succ[u])}")
603         backtrack_sol(best_edge_sol[u])
604
605 depth = 0
606 def backtrack_sol(x):
607     global depth
608     if TRACE: print(" *depth+\"backstart:\",x)

```

```

609     depth += 1
610     solution_type, data = x
611     if solution_type == "Exit_triangle":
612         A,B,C = data
613         if C in ('left', 'right'):
614             side = C
615             if side == 'left':
616                 l,r = A,B
617             else:
618                 l,r = B,A
619             # (l,r) is in the clockwise direction around the blob
620             include_in_solution("blob edge", (r,l))
621             backtrack_sol(("Chord_" + side, (A,B)))
622             for u in cross_left_to_right[l,r]:
623                 backtrack_sol_tree(u)
624             for u in children[B]:
625                 if in_upper_left_or_right_wedge(A,B,u, side):
626                     backtrack_sol_tree(u)
627             for u in children[A]:
628                 if in_lower_left_or_right_wedge(A,B,u, side):
629                     backtrack_sol_tree(u)
630         else:
631             solution_triangle(A,B,C)
632             backtrack_sol(("Chord_left", (A,B))) # left-facing chord
633             backtrack_sol(("Chord_right", (A,C)))
634     elif solution_type == "Exit_digon":
635         A,B,side = data # A below B
636         if side == 'left':
637             l,r = A,B
638         else:
639             l,r = B,A
640         include_in_solution("blob edge", (r,l))
641         for u in cross_left_to_right[l,r]:
642             backtrack_sol_tree(u)
643         for u in children[B]:
644             if in_upper_left_or_right_wedge(A,B,u, side):
645                 backtrack_sol_tree(u)
646         for u in children[A]:
647             if in_lower_left_or_right_wedge(A,B,u, side):
648                 backtrack_sol_tree(u)
649         backtrack_sol(("Chord_" + side, (A,B)))
650     elif solution_type == "Chord_left": # intermediate type (not stored)
651         A,B = data
652         t2,data2 = best_chord_sol[A,B]
653         #side = frontside[A,B]
654         if TRACE: print(" *depth, - best = ", (t2,data2),
655                        "value =", best_chord_value[A,B] )
656         if t2 == "Triangle":
657             C = data2
658             solution_triangle(A,C,B)
659             backtrack_sol(("Chord_left", (A,C)))
660         else: # t2 == "Digon", data2 is redundant
661             include_in_solution("blob edge", (A,B))
662             for u in cross_left_to_right[B,A]:
663                 backtrack_sol_tree(u)
664             for u in children[B]:
665                 if in_upper_left_or_right_wedge(A,B,u, 'right'):
666                     backtrack_sol_tree(u)
667             for u in children[A]:
668                 if in_lower_left_or_right_wedge(A,B,u, 'right'):
669                     backtrack_sol_tree(u)
670     elif solution_type == "Chord_right":
671         A,B = data
672         t2,data2 = best_chord_sol[A,B]
673         if TRACE: print(" *depth, - best = ", (t2,data2),

```

```

674         "value =", best_chord_value[A,B] )
675     if t2 == "Triangle":
676         C = data2
677         solution_triangle(A,B,C)
678         backtrack_sol(("Chord_right", (A,C)))
679     else: # t2 == "Digon", data2 is redundant
680         include_in_solution("blob edge", (B,A))
681         for u in cross_left_to_right[A,B]:
682             backtrack_sol_tree(u)
683         for u in children[B]:
684             if in_upper_left_or_right_wedge(A,B,u, 'left'):
685                 backtrack_sol_tree(u)
686         for u in children[A]:
687             if in_lower_left_or_right_wedge(A,B,u, 'left'):
688                 backtrack_sol_tree(u)
689     elif solution_type == "Tree":
690         v = data
691         for u in children[v]:
692             backtrack_sol_tree(u)
693     else:
694         raise ValueError
695
696     depth -= 1
697     if TRACE: print(" *depth+"backend:",x)

```

Listing 13: Show the solution and check its value

```

703 def clean_underscore(s):
704     return "".join((x if x != '_' else r'\_') for x in str(s))
705
706 val = 0
707 solution = []
708 backtrack_sol(best_sol)
709 for s,(a,b) in solution:
710     d = distance(a,b)
711     val += weighted_distance(a,b, s == "tree edge")
712     print(s,(a,b), f"{d:5.3f}", "-----" if s == "tree edge" else "")
713 print("total length", val, "**DISCREPANCY**" if abs(val-best_value)>1e-10 else "")
714 print(f"{best_value=} {best_sol=} {MST_cost=}")

```

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

```

720 draw_ipe.open_ipe("blobsol.ipe")
721 draw_ipe.start_page()
722 draw_ipe.start_frame()
723 for s,(i,j) in solution:
724     draw_ipe.draw_edge(x,y,i,j, 'red' if s == "blob edge" else 'blue',
725                       extras = ' pen="fat" ' )
726 draw_ipe.draw_tree(x,y,succ,point_labels = True)
727 draw_ipe.end_frame()
728 draw_ipe.put_text(f"optimum solution")
729 draw_ipe.end_page()
730 for i in range(0,min(n-1,20),2): # a few random subproblems
731     ##### some chord problem solution #####
732     draw_ipe.start_page()
733     draw_ipe.start_frame()
734     draw_ipe.draw_tree(x,y,succ)
735     if y_x[i+1]>y_x[i]:
736         a,b = i,i+1
737     else:
738         a,b = i+1,i
739     draw_ipe.draw_edge(x,y,a,b,'red',extras = ' pen="heavier" dash="dashed" ' )
740     sol = best_chord_sol.get((a,b))
741     val = 0
742     if sol:

```

```

743     if frontside[a,b] == 'L':
744         sol = ("Chord_left", (a,b))
745     else:
746         sol = ("Chord_right", (a,b))
747     solution = []
748     backtrack_sol(sol)
749     for s, (u,v) in solution:
750         val += weighted_distance(u,v, s == "tree edge")
751         draw_ipe.draw_edge(x,y,u,v, 'red' if s == "blob edge" else 'blue',
752                             extras = ' pen="fat"')
753     draw_ipe.end_frame()
754     for s,t in enumerate(chord_problems):
755         if (a,b) in t:
756             draw_ipe.put_text(
757                 f"Chord {(a,b)=}, {frontside[a,b]=}, subproblem size={s}, "
758                 + ("no solution" if (a,b) not in best_chord_sol else
759                   f"cost={best_chord_value[a,b]:5.4f}, " +
760                   f"total length={val:5.4f}, "+
761                   f"solution={clean_underscore(best_chord_sol[a,b])}")
762             break
763     else:
764         draw_ipe.put_text(f"{a,b} invalid chord")
765     draw_ipe.end_page()
766
767     ##### some edge problem solution #####
768     j = succ[i]
769     if j is None:
770         continue
771     draw_ipe.start_page()
772     draw_ipe.start_frame()
773     draw_ipe.draw_tree(x,y,succ)
774     draw_ipe.draw_edge(x,y,i,j, 'blue', extras = ' pen="ultrafat" dash="dashed" ')
775     sol = best_edge_sol[i]
776     val = distance(i,j)*beta
777     solution = []
778     backtrack_sol(sol)
779     for s, (u,v) in solution:
780         val += weighted_distance(u,v, s == "tree edge")
781         draw_ipe.draw_edge(x,y,u,v, 'red' if s == "blob edge" else 'blue',
782                             extras = ' pen="fat"')
783     draw_ipe.end_frame()
784     draw_ipe.put_text(f"Tree edge {(i,j)}, subproblem size={subtree_size[i]}, " +
785                       f"cost={best_edge_value[i]:5.4f}" +
786                       f", total length={val:5.4f}, solution={clean_underscore(sol)}")
787     draw_ipe.end_page()
788     draw_ipe.close_ipe()

```