Circular Trapezoidal Decomposition

William K. Mennell

Robert H. Smith School of Business
University of Maryland
College Park, MD 20742
wmennell@rhsmith.umd.edu

Summary. This technical note describes an implementation of the Mulmuley's randomized trapezoidal decomposition algorithm for circles instead of lines. It uses a DAG to store trapezoids (it does not keep a history of all the DAGs built however). It suffers from some of the numerical problems common in computational geometry applications but still does well considering. Some more work is needed before I am willing to put the executable and/or code online, though.
Key words: Directed acyclic graph; history DAG; trapezoidal decomposition; circle decomposition.

1 Introduction

The computational geometry literature abounds with algorithms constrained by provable performance bounds and based on convenient sets of assumptions. Actual implementation of the algorithms and adaptation to real-world settings without all the tidy assumptions is a significant task in its own right, however, and is not generally described except to perhaps briefly describe the standard degeneracies and how to handle them. As a semester-long project for an independent study of computational geometry, we implemented both the standard trapezoidal decomposition algorithm for line segments without almost any assumptions and a similarly generalized algorithm using circles in place of segments. This technical note describes the implementation at a moderate to high level and assumes a prior understanding of the Mulmuley randomized trapezoidal decomposition algorithm (RTDA) and some standard computational geometry terminology. Readers without this knowledge should consult either [2] or [4], together which should suffice. Alternatively, [5] gives a slightly more intuitive description.

The remainder of this note is organized as follows: Section 2 defines the basic terminology used, while section 3 describes the data structures implemented. Section 4 details the main implementation functions, while Section 5 describes the accompanying GUI. Finally, 6 discusses the remaining few issues to fix along with possible resolutions and points out the major strengths and weaknesses of the implementation as it stands. The appendix catalogs numerous small graphs of test cases and the difficulties each one addresses.

2 Terminology and Definitions

Consider a set, S, of lines in the plane. For our purposes, a trapezoid consists of a top and bottom line segment from S of any orientation along with two vertical "bullet paths" connecting the two structures. Slightly abusing the terminology, in the case where a set, C, of circles replaces S, we say that a trapezoid has a top and bottom arc together with the two bullet paths. Note that each vertical segment may be degenerate in the sense that it will have zero length. Figures 1 and 2 show some examples - pay special attention to the circle trapezoids which at first glance may hardly look "trapezoidal".

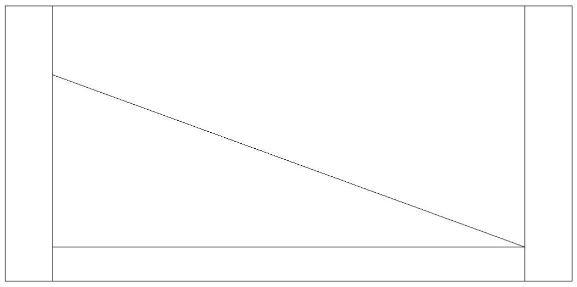


Fig. 1. A trapezoidal decomposition of 2 line segments. Because the two line segments share a common endpoint, one trapezoid has a degenerate right bullet path.

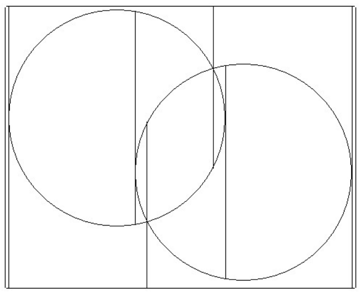


Fig. 2. A trapezoidal decomposition of 2 circles. The left and right-most trapezoids for each circle have one degenerate bullet path each.

3 Data Structures

A directed acyclic graph (DAG) underlies any RTDA. A node in the DAG can have an arbitrary number of parents pointing to it but only two children. Because the end result of Mulmuley's algorithm can be implemented such that multiple copies of the DAG are maintained showing the "history" of its evolution, we refer to it as a *historyDAG* or *hDAG*. Our implementation, however, uses only one copy of the DAG since we have no need for knowing the trapezoids created during the algorithm that are no longer extant. Inside the basic data structure, three kinds of nodes are stored - x, y, and leaf nodes. x nodes correspond to segment endpoints and are used to answer the query, "Is the point I am looking for to the left of or to the right of this current point?" y nodes correspond to line segment or circle objects and answer the same query, only vertically, and leaf nodes correspond to actual trapezoids. Therefore, every generic node in hDAG has a void pointer object called *contents* that is dynamically cast as necessary. A node knows the type (x, y, or trapezoid) of its contents and has access to it. Similarly, each *contents* knows its container node (see 3.2) which comes in handy later on as we will see in 3.2.

3.1 Circles

When moving left-to-right along a line segment or circle arc, we must figure out which trapezoid to enter next after leaving the current one. If exiting the top or bottom, how are we to know which trapezoid to enter next? Since the top and bottom parts of a trapezoid are line segments or circle arcs, this suggests that by storing the leftmost trapezoid bordering each segment/arc on both the top and on the bottom, we can then travel along this segment/arc to find the correct one. Figure 3 shows how this looks. [1]

Therefore, in both the linear and circular cases, it is necessary to store the respective lines and circles as their own objects, with pointers to the leftmost trapezoids above and below. With circles, we implicitly treat the upper half of the circle separately from the lower half and so each half must know the leftmost trapezoids above and below. In 3, the top half of circle X has A as its $aboveLeft$ trapezoid and X1 as its $belowLeft$ whereas the bottom half of X has X1 as its $aboveLeft$ and B as its $belowLeft$. [2]
Additionally, of course, we must also store the coordinates of each circle's origin.

[1]Although in some very dense cases it may be faster to simply query the hDAG to find the next trapezoid, we think that because of the randomized nature of the algorithm, in the expected case, the number of trapezoids to traverse will be less than the depth of the hDAG, thereby making it more efficient. This would have to be proved, but if we are wrong, it is a one-line code change to fix.

[2]The only time that the top half's $belowLeft$ is equal to the bottom half's $aboveLeft$ occurs when an arc bisects a circle exactly through its leftmost point.

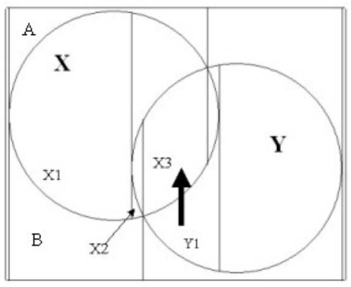


Fig. 3. Imagine a convex arc traveling up from Y1 into X3. How does the algorithm know to proceed to X3 next? Start at circle X's furthest left trapezoid on the upper half of its bottom arc - X1. Follow the arc through X2 until X3 is found using *bottom right neighbors* described in 3.2.

3.2 Trapezoids

The trapezoidal representation is the most complicated data structure in the implementation. We give a brief itemized list for important features of the circular trapezoid; the linear version is essentially a subset of these.

- Left and Right Borders. a. Mulmuley's RTDA for line segments only requires storage of the left and right x coordinates of a given trapezoid and does not explicitly store the y coordinates. This is because of the ease of computation of upper and lower boundary points for each side of a trapezoid, and we follow his convention. It is worth noting that in both the line segment and circle cases, it may be worth storing these coordinates given that storage is not nearly as important in the speed vs. memory tradeoff as it was when this algorithm was created. This is especially important in the circle case where it is more more computationally expensive than in the linear case, but in both cases, these y coordinates are used in finding right neighbors, handling possible corner exits or entries and a few other places, i.e. they are unnecessarily computed many, many times.

b. Left/Right neighbors. As in Mulmuley's implementation, every trapezoid needs to know what borders its lower segment on the left and right side and similarly for the top segment. This is necessary for ease of movement between trapezoids instead of having to query the hDAG every time a new trapezoid is needed. See [5] for definitions and pictures depicting left and right neighbors. However, there are numerous cases where there is no by-definition "neighbor" even though a contiguous trapezoid does exist or there are more than two neighbors. We describe how many of these cases are handled in 4.6. Figures 4 and 5 show the two of the typical cases.

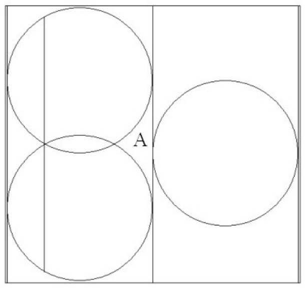


Fig. 4. Trapezoid A has no top or bottom right neighbor, as they are defined in [5].

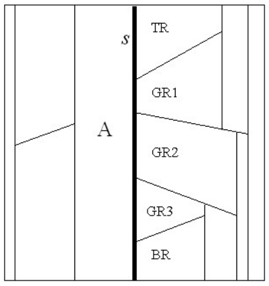


Fig. 5. Trapezoid A has an upper right neighbor, a lower right neighbor, and three *generalright* neighbors. There is no easy an uninvolved algorithmic way to know which is the next one entered so we query the hDAG to find the next one.

b. Container Node. Each node in the hDAG is a *containernode* for either a trapezoid, an arc endpoint, or a circle as described earlier. Conversely, every trapezoid, endpoint, and circle has a pointer to the node which contains it in the hDAG. Once we access a trapezoid, we can simply use its container node pointer to access the its location in the hDAG. [3]

c. Top and Bottom Segments of Circles. Since every trapezoid's top and bottom segments are not bullet paths but rather parts of either a line segment or a circle arc, we therefore need to know who these "originating parent" segments/arcs are, where "originating" is based on each circle having an *origin*. E.g. if we are exiting a trapezoid out of its top segment, we then find the circle it belongs to, and travel from left to right along the appropriate edge until we find the trapezoid we need - see earlier in Figure 3.

[3]Nodes of the hDAG are pulled from a "freelist" implementation that significantly decreases the number of times the **new** operator is called but potentially wastes memory. We should also do this for trapezoids.

d. Top vs. Bottom Half of a Circle. In the circle case, each trapezoid's top and bottom segment is part of either the bottom half of a circle or the top half, see Figure 7. Knowing whether the top/bottom is convex or concave is crucial to correctly compute circle [4] intersections along with many other small but important calculations.

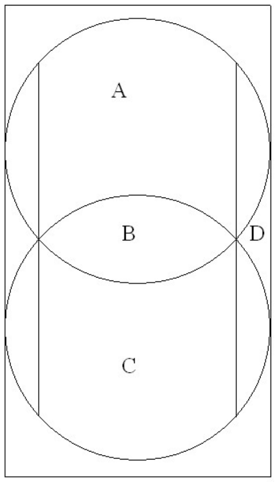


Fig. 6. Trapezoid A has concave top and bottom segments. B has a concave top and convex bottom. C has convex top and bottom, and D has convex top and concave bottom.

e. Merging Information. Frequently, when a new trapezoid is created, it has the same top and bottom originating parents as one of its adjacent neighbors. These two trapezoids then have to be merged. The adjacent trapezoid, *adj*, to be combined with the new one cannot simply be plucked from the hDAG as all of the nodes pointing to *adj* will then be left dangling. Therefore, every time a trapezoid is merged with another, all nodes pointing to it are added to the new trapezoid's list of *mergeparents*, and all necessary updates of neighbors, etc. are carried out. Figure

f. Degenerate Neighbors. Especially in the linear case, it is probably possible to deal with almost all degeneracies in a nice algorithmic way. Figures 4 and 5 previously depicted such cases where knowledge of a "general right neighbor" would maintain the ease of navigation across trapezoids present in non-degenerate cases. There are many cases where knowledge of numerous general neighbors would be necessary but in our experience, the amount and difficulty of properly updating and storing such a list for

[4] nice alliteration, huh?

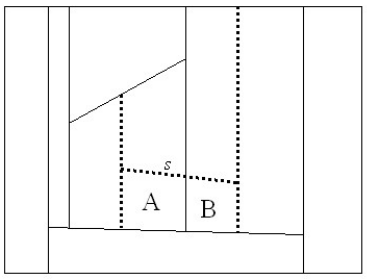


Fig. 7. When segment s is added, note that trapezoid B's left top neighbor is the same as its left bottom neighbor. Therefore, A and B will be merged into one trapezoid.

each trapezoid is not worth the effort. [5] In cases such as these, a query of the hDAG returns the necessary trapezoid. As will be discussed below, the complexity of degeneracies for the circle case seems to be significantly higher and in order to keep the code from blowing up, and for correctness' sake, we currently have shut off the algorithmic handling of degeneracies for circles and simply use hDAG queries.

g. Degenerate Bullet Paths. When two segments/arcs intersect, the intersection forms a degenerate bullet path for one of the new trapezoids. When entering or exiting a trapezoid at a corner point or searching for a general right neighbor, knowledge of this is crucial. Occasionally, *both* the top and bottom right neighbors of a trapezoid, t, will have degenerate left bullet paths meaning that the elephant in the room (the true right neighbor for t except in corner exits) is not stored by t and instead has to be found by an hDAG query.

h. Unique Identifier. Every trapezoid is assigned a unique integer identifier when it is created. This feature is a simple debugging tool very useful when debugging issues in problem instances with thousands or even millions of trapezoids. We can quickly zero in on exactly when the trapezoid in question was created or used improperly in order to deduce the problem.

i. Radius. For now, every circle in the graph has the same fixed radius, r. However, it will not be difficult to allow arbitrary radius for each circle which will then necessitate this attribute.

[5] To be able to handle the multiple-right-neighbors case, every trapezoid would have to maintain a dynamic list of all its general right neighbors. The updating and other overhead required for maintaining algorithm correctness for just one such neighbor proved so burdensome that the inelegant hDAG query becomes worth the cost.

j. Degree. This attribute stores the number of circles that a given trapezoid lies within. For instance, if we have considered twenty circles so far, and trapezoid t lies completely within five of them, then it has degree 5. Note that by definition of the circle trapezoid, all points in a given trapezoid have the same degree.

4 High Level Description

At the highest level the RTDA is summarized as follows. Repeat until no circles remain to be considered:

a. Obtain next circle.

b. Choose the lower half arc, travel through all intersected trapezoids, iteratively creating the new ones.

c. Choose the upper half arc and do the same.

4.1 Data Input and Storage

It is of course well-known how machine representations of number can cause numerous difficulties and affect behavior egregiously if not properly accounted for. To allow maximum precision [6] all coordinates and computations based on them are stored as long double. Frequently, therefore, a number such as 10.6 is stored as 10.599999999999 which can cause a point on the border of some trapezoid to be considered outside the trapezoid. Or, a check of two numbers for equality may be returned as false when in fact they are equal.

We rectify this situation in two ways:

1. All numbers are input as strings in scientific notation format. A simple routine then converts them to long double format and for some reason, all values are now stored correctly - e.g. 0 is stored as 0 and not as 0.00000000000001. All the built-in Borland C++ formatting routines do not achieve this for some reason.

2. All mathematical operations and less-than/greater-than comparisons are run through a black box that converts them to a form that compares one number with zero. E.g. “if (i ¡ j)” is converted to “if (blackBox(i - j) ¡ 0)”. The black box takes the input argument and compares it with its ceiling and floor function values. If the absolute value of the difference between either is less than epsilon (usually 1e-13) then the ceiling or floor is returned, respectively. [7]

[6] and because our Close-Enough TSP implementation uses this

[7] Every single comparison or mathematical operation that does not use the black box at some point has caused a bug in the program testing so we include it universally. This is at the great expense, however, in execution time. For users needing

Once all segments/circles have been read in, they are sorted in increasing order by x coordinate. All ties are sorted in increasing order of y coordinate.

4.2 Choosing the Next Circle

The structure of the final hDAG is dependent on the order in which segments/circles were added. Therefore, to "prove" correctness, we incorporated a randomized "hat" algorithm from [1] and simply pick segments/circles out of the hat until there is none left. Numbers are chosen using a uniform distribution over the cardinality of the set of segments/circles. Using different seeds, we can eventually ensure that all permutations of segments/circles have been considered. Naturally, for problems larger than roughly ten, this becomes intractable, but the option is at least available.

4.3 Traversing Trapezoids Along the Segment/Arc

The heart of the algorithm is the process of iteratively following the current segment/arc through the trapezoids that it intersects, removing each one and replacing it with the newly trapezoids created, see segment s in Figure 7. We implemented a method called $findIntersectedTraps$ to deal with this. Its core function is to repeatedly find the next intersection point, create the new trapezoids, and move to the next trapezoid, until we reach the end of the current arc. Numerous small details make this not quite so simple a task however. For instance, sometimes the starting point of an arc IS an intersection with another circle, see Figure 32. Or, frequently, the current trapezoid contains the far right point, but we cannot quit yet because there are more intersections in between, see Figure 27. Determining whether we exit the top, right, or bottom of the current trapezoid is determined by comparing the y value of the next intersection with the top and bottom y values of the current trapezoid's right bullet path. As described earlier, it is not computationally cheap to repeatedly calculate these top and bottom y values, but for now, we have not added them as attributes of each trapezoid since Mulmuley's algorithm for line segments does not.

We call method $handleExit$ to take care of the actual exiting process - it determines which new trapezoid we enter, calls the method that creates the new trapezoids, $segInOneTZoid$, ensures that all neighbor relationships are properly updated, and handles all possible merges.

4.4 Obtaining Intersection Coordinates

It is a simple matter to obtain both sets of intersection points, if they exist, of two circles. It is a much more difficult matter, however, to know which is the

less numerical precision, floats could be used and the computational savings should be more than significant, especially for larger problems.

correct one to use. Each time an intersected trapezoid, t, is entered, we must determine where the arc segment will exit it. There are two options: either we exit at an intersection of one of t's top or bottom origin circles with the current arc, or we exit through the right bullet path. Therefore, separate functions are created, called $getCCIntersection$ and $getCBIntersection$, respectively, to determine the next circle intersection and the next bullet path intersection and then the correct one is chosen. To implement the circle intersection correctly, knowledge of the relative angle between the two circles is needed, along with which quadrants their points of intersection are in, and whether or not the current arc is the top or bottom half of the current circle.

4.5 Does a Given Point Lie Within this Trapezoid?

It is frequently necessary to test whether or not a given (x,y) pair lies within trapezoid t. Knowledge of t's top and bottom segments, i.e. convex or concave, is needed along with the upper and lower y-coordinates of the trapezoid at this given x value. This computation is necessary at numerous points of the algorithm and results in a great time expense given that each calculation involves square and square root function calls.

Similarly, we often need to know if the furthest right point for the current circle lies within the current trapezoid. This is not a trivial test, and function $isInTrap$ is implemented to determine this. Note that, as described earlier, even when the answer is yes, this does not guarantee that there are not more intersections to compute before we quit, as Figure 27 depicts.

4.6 Obtaining the Trapezoid to the Right of the Current One

When trapezoid t is exited through its right bullet path, we call a function named $getRtNghbrTrap$ to locate the correct one. Most of the time, this is as simple as choosing between the right top and the right bottom neighbor. However, when degenerate situations are allowed, as mentioned earlier, there will not always be a right neighbor at all. Most of the linear cases can be handled algorithmically using the $generalrightneighbor$ construct, but in the circle case, the degeneracies are significantly more subtle. E.g. what if both top and bottom right neighbors exist but they both have degenerate left bullet paths? Then at least one trapezoid exists in the middle and we have to choose the correct one, see Figure **??**. For all of these cases, we use a function, $ptEpsilonRight$, that moves epsilon distance to the right along the current arc, and then queries the hDAG to locate the correct trapezoid. It is not a $perfect$ solution, of course, given that one might come up with geometric proof of **all** possible degenerate cases and then plan for them, but based on our experience, this quickly became a mess within the code and still was not always correct.

Another subtle wrinkle to this problem occurs when we exit or enter at a top or bottom corner of the new trapezoid. This is handled by a function called

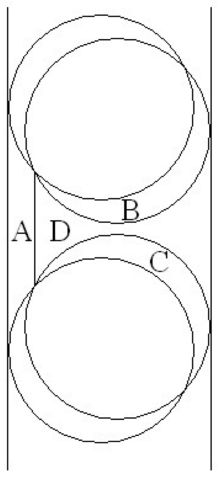


Fig. 8. Trapezoid A's upper right neighbor is B and its lower right neighbor is C. Both have degenerate left bullet paths and so D is the right neighbor that typically will be needed coming from A.

handleCorner which utilizes each circle object's knowledge of its leftmost trapezoids to then iterate from left to right until the correct trapezoid is found. Even **here**, however, it is not just that simple. Sometimes we may be exiting the corner of the far right trapezoid for a given circle and this trapezoid may have no right neighbors. Here again, we use the *ptEpsilonRight* function to locate the correct trapezoid, see Figure 9.

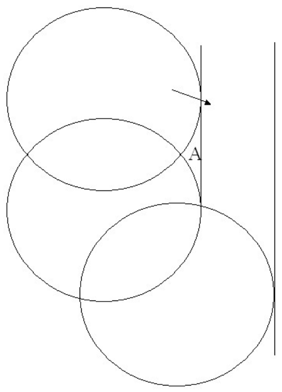
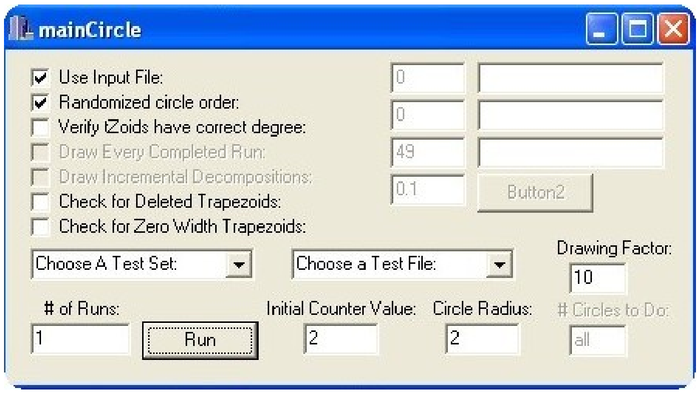


Fig. 9. Trapezoid A is the last trapezoid along the top segment of its bottom circle. Any arc exiting it through its top right corner will have to then query the hDAG.

5 How to Use the GUI

See Figure 10 to follow along with this description. To use the program, an input file or string must be selected (random generation of data will be added eventually). If the user wants to run of the 60 small test examples, uncheck the box "Use Input File" and then select the input string desired from the drop-down box captioned "Choose a Test Set". If instead, an input file is desired, choose one from the drop-down box captioned "Choose a Test File".



- [✓] Use Input File:
- [✓] Randomized circle order:
- [ ] Verify tZoids have correct degree:
- [ ] Draw Every Completed Run:
- [ ] Draw Incremental Decompositions:
- [ ] Check for Deleted Trapezoids:
- [ ] Check for Zero Width Trapezoids:

0
0
49
0.1

Choose A Test Set:
Choose a Test File:
Drawing Factor: 10
# of Runs: 1
Initial Counter Value: 2
Circle Radius: 2
Circles to Do: all

Fig. 10. The GUI interface for the circle decomposition executable.

Pick any real-numbered radius greater than zero using the box titled "Circle Radius". Bear in mind that a very large radius risks creating such a dense problem that execution may take "a while" for large problems. The box entitled "# of runs" allows the user to make multiple execution runs to verify that the same results are produced for different random orderings of the input segments/circles. For each additional run, the initial random seed is incremented by one. The initial random seed value is controlled by the "Initial Counter Value" box. To run the program, simply click "Run". Ignore the greyed-out Button2 and its associated boxes as they are for testing and have not been removed yet.

5.1 Options

The remaining checkboxes all provide various checks on algorithm correctness. If "Randomized circle order" is left unchecked then the circles will be input in order of storage (left to right by x value). "Verify tZoids have correct degree" checks to see after all trapezoids have been created if they all have the correct degree". As discussed below, this is currently beside the point since the degree for all trapezoids is computed at the end of the algorithm. "Check for deleted trapezoids" checks through the entire hDAG each time new trapezoids are added to see if the trapezoid from which they are created is still pointed to anywhere within the hDAG. If it is, then once it is deleted, there will be dangling pointers in the hDAG that will cause errors if accessed. "Check for zero-width trapezoids" lets the user know if any trapezoids are about to be added with width zero or even negative, i.e. the right end of the trapezoid is to the left of the left end. The latter three options add significant time to execution but the user is advised to use them until he is comfortable with the correctness of the implementation. There are also a few code snippets in certain parts of the code that will notify the user that wrong conditions have been met and hang execution. Finally, the "Drawing Factor" is simply a multiplier that makes the displayed graph smaller or larger. Our [8] grasp of graphics is not very advanced and so graphs may not be perfectly contained or centered. Using input data within the [0,100] x [0,100] window should result in a reasonable display.

6 Issues

This section is not meant as quality prose but rather a compendium of various strengths and weaknesses of the implementation.

6.1 Strengths

a. Numerical Stability. Circles as small as 10^-13 are handled without error and all numerical errors are avoided using the black box function added in the code. Although it slows down computation, this important issue causes way too many errors to ignore; almost every single computation where the black box is not used has been the root cause of incorrect behavior in the program.

b. Trapezoid Drawing. Although relatively minor, because each trapezoid can be pointed to by multiple nodes in the hDAG, a traversal of the hDAG can result in redundant redrawings of numerous trapezoids. Adding a flag to each trapezoid that says if it has already been drawn prevents this inefficiency from occurring.

c. Navigation Trick. Look at Figure 11. Note that A1's bottom right neighbor is A2 and not C. Not letting trapezoids such as C be neighbors saves time in numerous cases and other difficulties, but it then poses the problem, how do I find C if I need to? Figure 12 depicts such a case where segment *s* needed to enter C but would instead find A2 initially. Once a test reveals A2 is not the correct trapezoid a kind of ladder-climbing method is used.

[8] *my*

A2's top segment's *aboveLeft* trapezoid is used to then move to the right until either the correct trapezoid is found or we repeat the process. In this case, C *is* A2's top segment's *aboveLeft* but often this is not the case. When t extends further left, this is not as efficient and in fact there are probably many highly contrived and degenerate cases where it will be more efficient to simply query the hDAG but for most instances similar to this, it provides a nice, quick method for finding the (usually one) trapezoid inbetween A1 and A2.

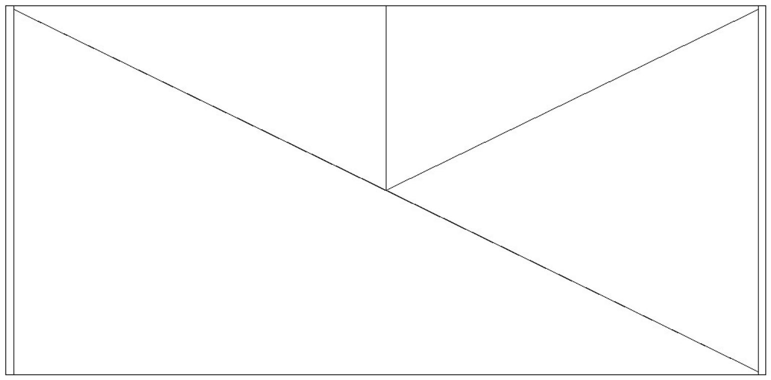


Fig. 11. Trapezoid A1's bottom right neighbor is defined as A2 here since C only is a neighbor for exactly one point. Typically A2 will be the needed trapezoid; only when a point intersects into C directly at the corner point does this become an issue.

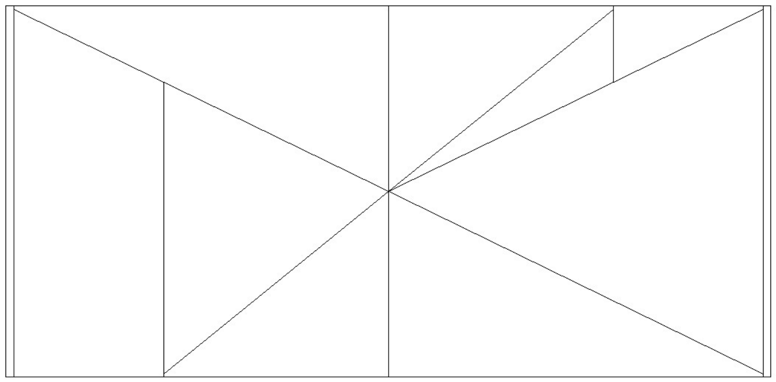


Fig. 12. Here we see that A2's top segment is t. Its *aboveLeft* trapezoid is C, and so we are done.

d. Degeneracy Handling. The decomposition algorithm is not very difficult or complicated with all the assumptions in the literature. Although there are still cases to be correct, see below, a significant number of subtle and difficult cases have already been handled; the circle case is rife with very subtle ones that are difficult to see without being involved in programming them, and we have not kept a meticulous record of every single case encountered.

6.2 Weaknesses

a. Line Segment Decomposition. Aside from some fairly absurd degeneracies such as when a segment completely overlays more than two other disjoint segments, there are two major cases to implement. 1. Multiple Right Neighbors - a simple fix of adding a boolean when this is the case and then calling *ptEpsilonRt* to find the correct one. 2. k lines intersect at point p and an additional m segments have p as their left endpoint, see Figure 13.

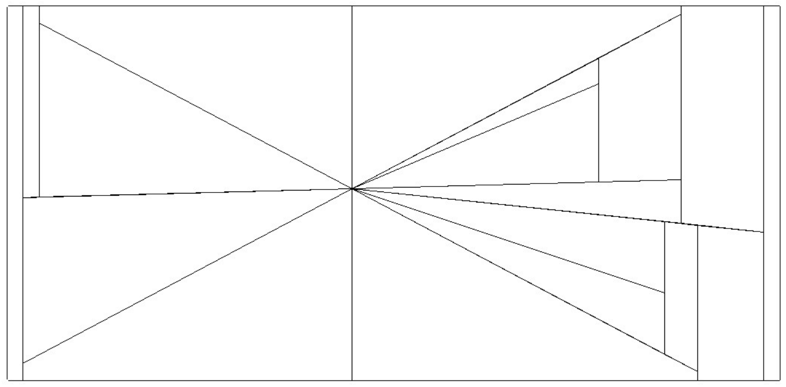


Fig. 13. This does not work yet for most permutations of the input segments.

b. Circle Decomposition. We have not yet added the capability of recognizing and handling the case where k ¿ 4 circles all intersect at some point p. Although the odds of this happening randomly using long double precision is extremely low, a user could define a cases with 100 or even more circles all sharing one intersection.

c. Hanging Pointers. Some large input files for the circle decomposition still produce cases of hanging pointers in the hDAG or trapezoids with ¡ 0 width, both of which imply errors. We do not know yet if these indicate still more degeneracies that have not been accounted for or if they are the result of programming errors.

d. Random Generation. The user should be able to generate n random line segments or circles (with n random radii if desired) to decompose allowing the user to play with the algorithm and further prove its "correctness".

e. Memory Management. We [9] have botched the memory-management aspects of the code, not withstanding a computer science background. Each time all dynamically allocated objects are destroyed, much of the memory is *not* returned to the heap, according to Windows Task Manager. We must start from scratch and trace exactly where the memory is lost so that we can create an executable truly stable across an arbitrary number of runs. As it now stands, running an extremely large decomposition, i.e. millions of trapezoids, would use up all RAM after about four iterations.

f. Storage of Corner Y Values. As discussed in the text, we should add top left and right and bottom left and right y coordinate storage to the trapezoid object. Although Mulmuley specifically does not do this, the circle decomposition calculates these values so often that it should be well worth the extra storage to achieve noticeable speed gains.

g. No Black Box for Numerical Difficulties. It would be interesting to include some sort of flag which redefines all functions and data as floats, instead of long double, and then shut off the black box that prevents numerical issues to see what the *actual* time difference is.

h. Trapezoid Degree Algorithm. As it currently stands, the algorithm finishes calculating all trapezoids and then applies an O(kn) algorithm to determine each trapezoid's degree, where k is the number of trapezoids and n is the number of segments/circles. For large problems this inefficiency becomes noticeable. So far, we have tried three different methods to calculate the degree as the algorithm progresses but none works properly yet. The first method tries to figure things out when each set of new trapezoids is created - very unlikely to work easily or be the best solution since it requires foreknowledge of how certain things will end up. The second and most promising, in our opinion, is to wait until circle c has been finished and then use a recursive algorithm to travel through all these new trapezoids updating as needed. This quickly becomes inordinately complex (some trapezoids involved with other circles may also need incrementing) and liable to infinite loops (visiting the same trapezoid or set of trapezoids in a loop without recognizing this). However, we think that an extremely rigorous approach to the problem to determine all possible situations, combined with judicious use of flags (the third method simply added use of

[9] I

flags) to determine if a trapezoid has already been visited, might result in a correct algorithm.

i. Arbitrary Radius. The motivation for this project is my work on the Close-Enough TSP where a relatively easy but important extension is to allow each circle to have its own radius. Therefore, this capability also needs to be added here. It should not take much as each trapezoid object has its own radius but might involve working out a few kinks here and there in the code.

j. Software Engineering. Although the code utilizes objects and compartmented functions well enough, there are some extremely ugly if statements in $findIntersectedTraps$. Determining whether or not more intersections need to be calculated even though the furthest right point lies in the current trapezoid, see Figures 27 and Figure 28, is not easy but there must be a more elegant and compact solution than the ugly if statements currently used.

Acknowledgement

Thanks to Dr. David Mount for patience as I implemented this algorithm up to its current state and for the helpful advice regarding the never ending degeneracies and numerical issues to consider.

References

1. AngleWyrm. http://home.comcast.net/ anglewyrm/hat.h.htm //
2. K. Mulmuley. A Fast Planar Partitition Algorithm, I. *Journal of Symbolic Computation*, 10:253–280, 1990.
3. K. Mulmuley. A Fast Planar Partitition Algorithm, II. *Journal of the ACM*, 38:74–103, 1991.
4. K. Mulmuley. Incremental Algorithms, *Computational Geometry: An Introduction Through Randomized Algorithms*, 84–93, Prentice Hall, 1994.
5. M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf. Point Location. *Computational Geometry: Algorithms and Applications*, 121–144, Springer, 1998.

7 Appendix

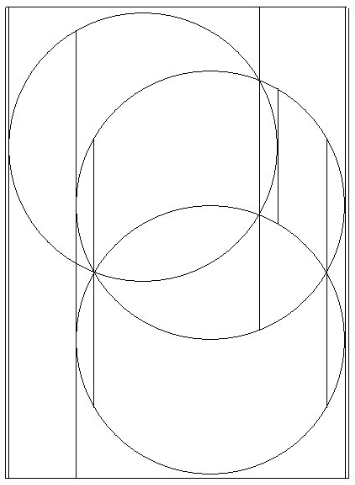


Fig. 14. 3 points share nearly the exact same intersection point.

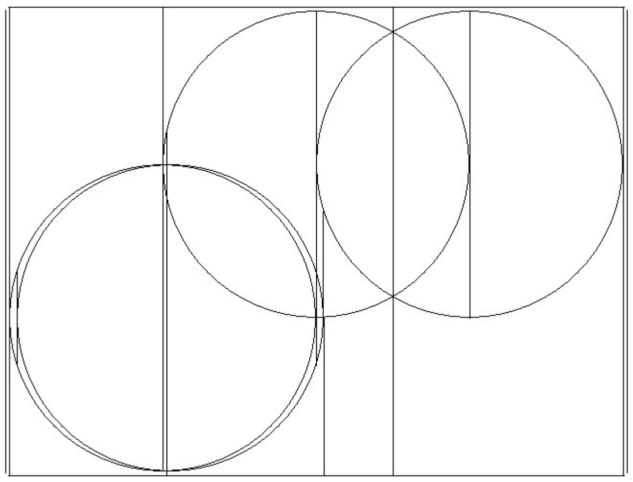


Fig. 15. 2 circles share the same y value making a difficult case for future right trapezoids.

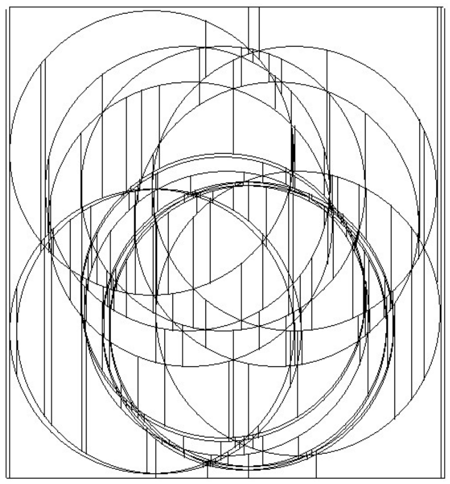


Fig. 16. An 18-node example.

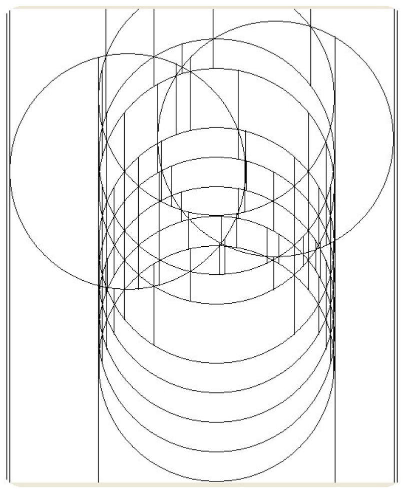


Fig. 17. Multiple vertical circles with both left and right side circle intersecting.

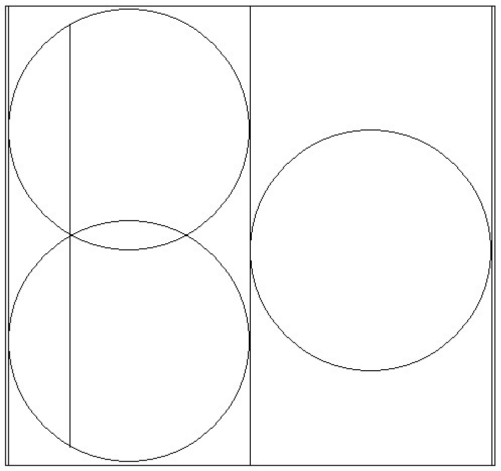


Fig. 18. Illustration of having no right neighbors.

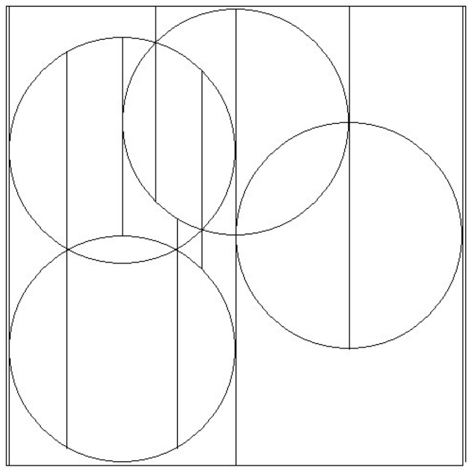


Fig. 19. The trapezoid with no right neighbors is intersecting exactly through the midpoint on the right.

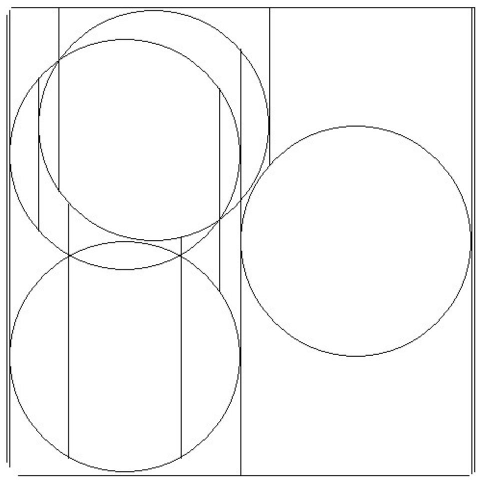


Fig. 20. Trapezoid with no right neighbors is intersected to get upper neighbor to the right.

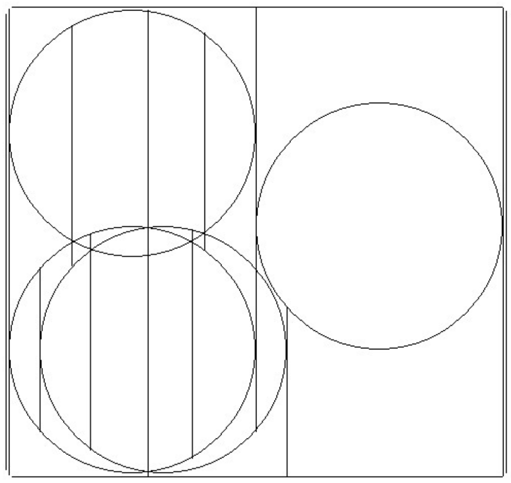


Fig. 21. Trapezoid with no right neighbors is intersected to get lower neighbor to the right.

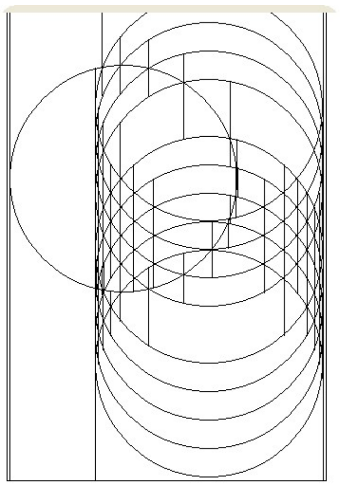


Fig. 22. Multiple vertical circles with one left intersecting circle.

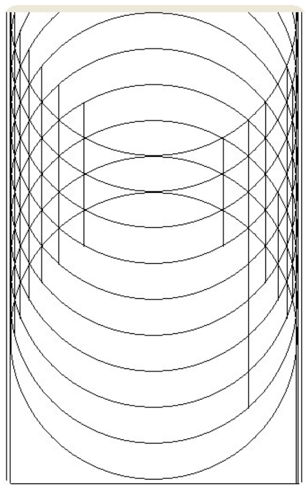


Fig. 23. Multiple vertical circles showing the numerous general right or left neighbors cases and circles touching at exactly one point so bullet paths created get merged.

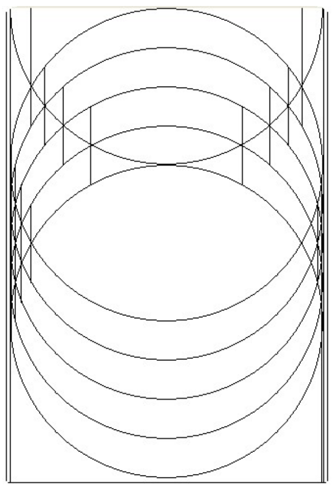


Fig. 24. Multiple vertical circles with one intersecting above.

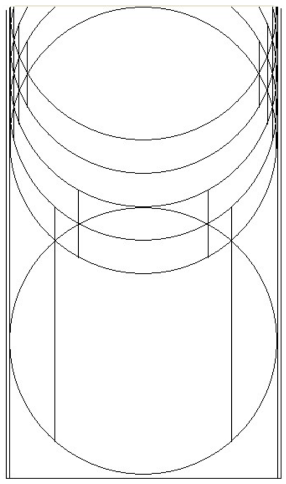


Fig. 25. Multiple vertical circles with one intersecting below.

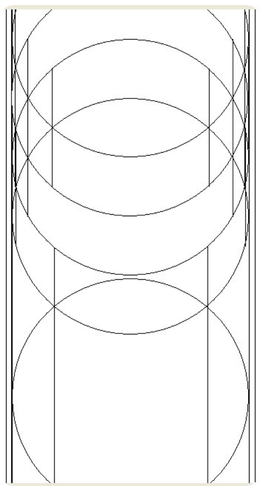


Fig. 26. Very nearly exactly vertical circles but not quite so - a few extremely thin trapezoids.

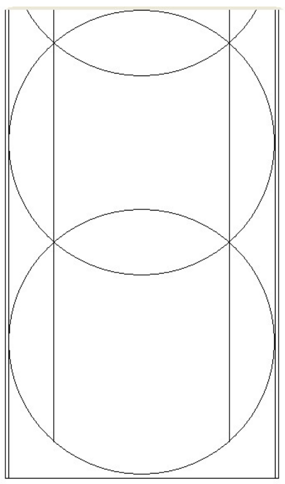


Fig. 27. Here the right end point of one circle may be in the current trapezoid but there are still more trapezoids to locate. This becomes much uglier when multiple circles also intersect the lens-shaped region.

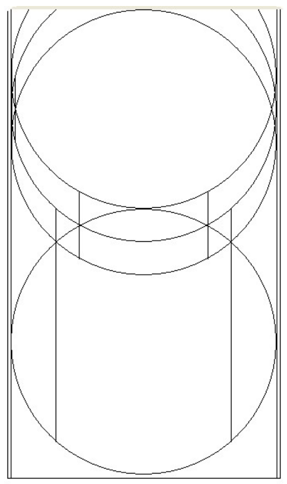


Fig. 28. Here the right end point of one circle may be in the current trapezoid but there are still more trapezoids to locate.

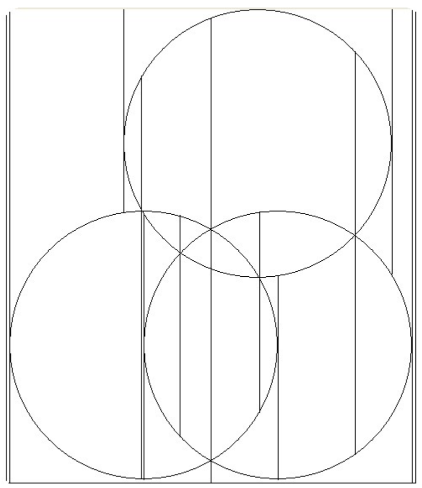


Fig. 29. Two circles share the same y value but more complicated middle case.

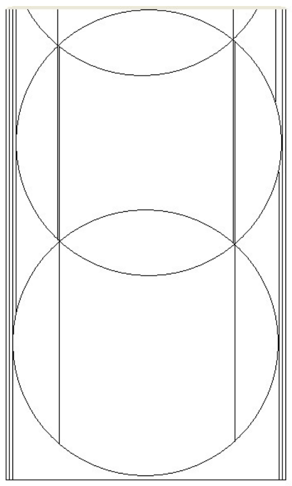


Fig. 30. Three circles are almost vertical - shows the narrow trapezoids created.

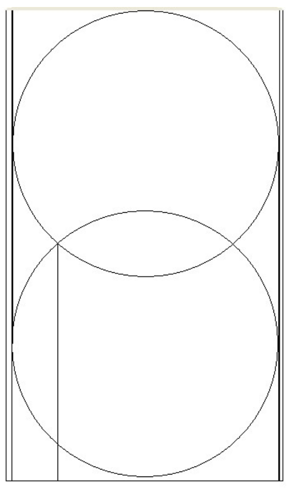


Fig. 31. THIS ONE HAS BECOME MESSED UP - MUST FIX!! Some trapezoids here are 10ˆ8th wide.

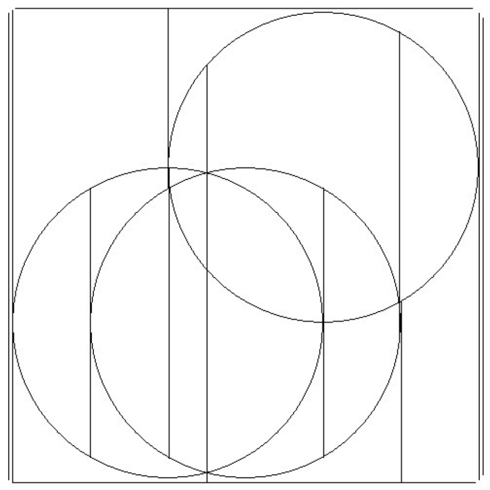


Fig. 32. The far left point of one circle IS an intersection - very difficult to handle.

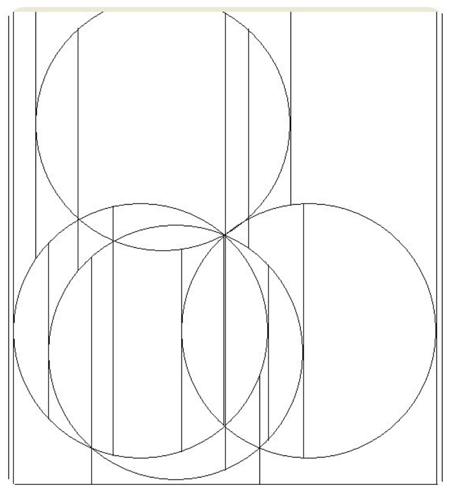


Fig. 33. Four circles intersect at exactly the same spot. We still need to get k circles to share same intersection and work.

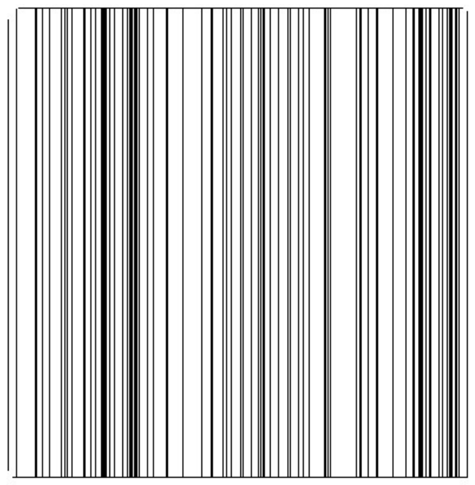


Fig. 34. Radius is 10ˆ9th so 100 node problem looks like a barcode.

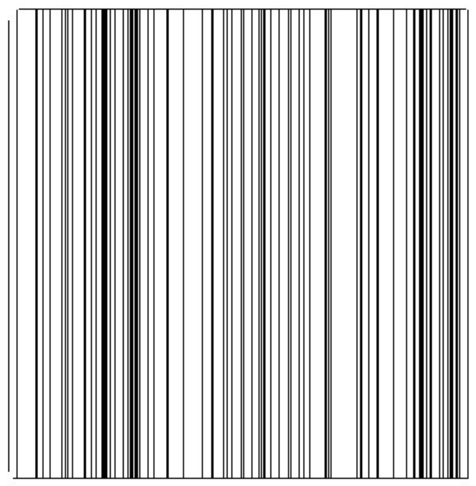


Fig. 35. Radius is just above machine epsilon for our purposes 10ˆ12th, so 100 nodes look like barcode.

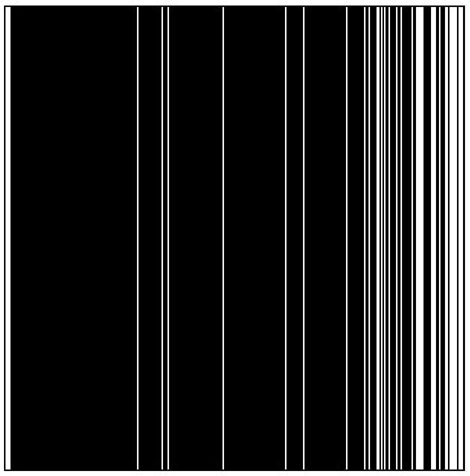


Fig. 36. Same barcode effect but using a 1000-node problem.

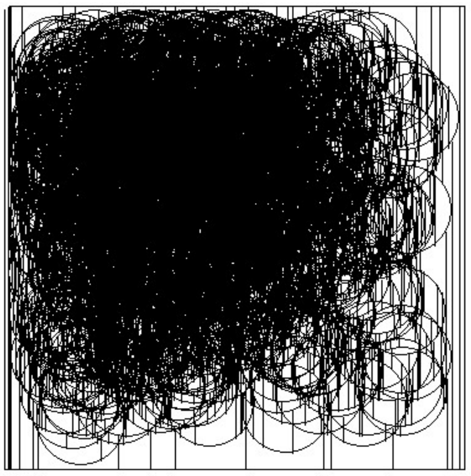


Fig. 37. Dr. Golden's 1000-node instance with radius 12 - produces 658469 trapezoids.

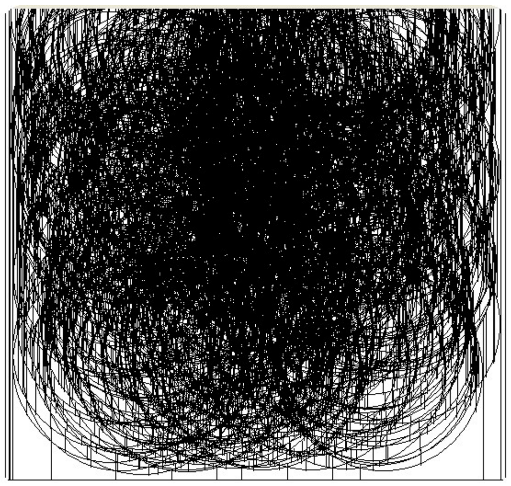


Fig. 38. Dr. Golden's 500-node instance with radius 27 - produces 460301 trapezoids.

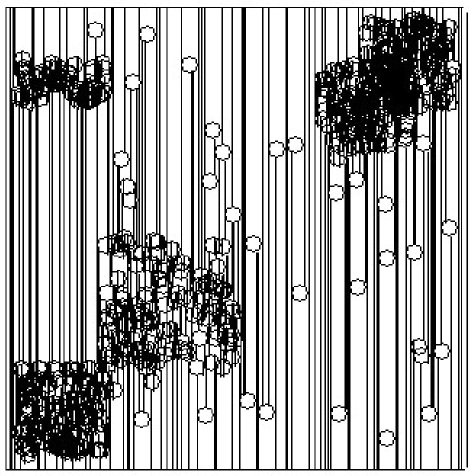


Fig. 39. Dr. Golden's 499-node instance with radius 2 - produces 17447 trapezoids.

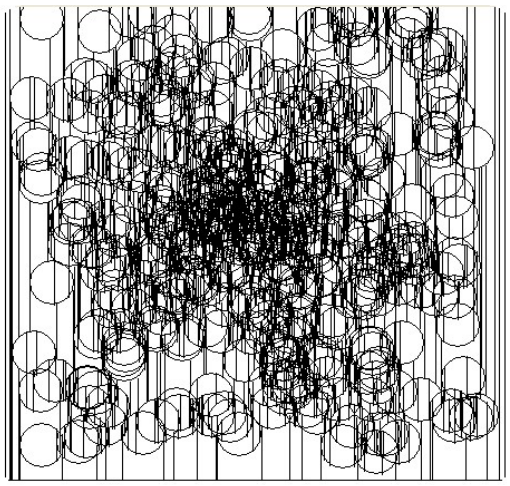


Fig. 40. Dr. Golden's 400-node instance with radius 5 - produces 21767 trapezoids.

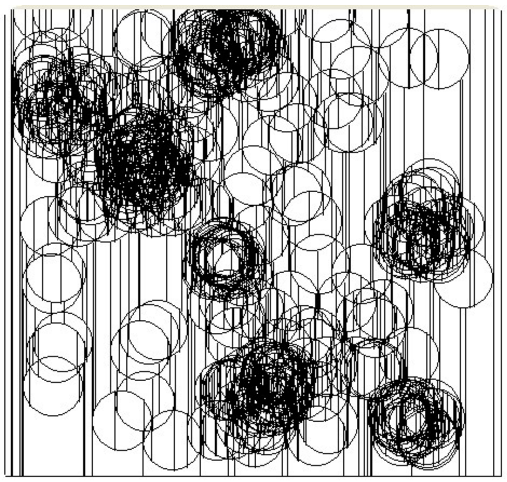


Fig. 41. Dr. Golden's 300-node instance with radius 7 - produces 27304 trapezoids.

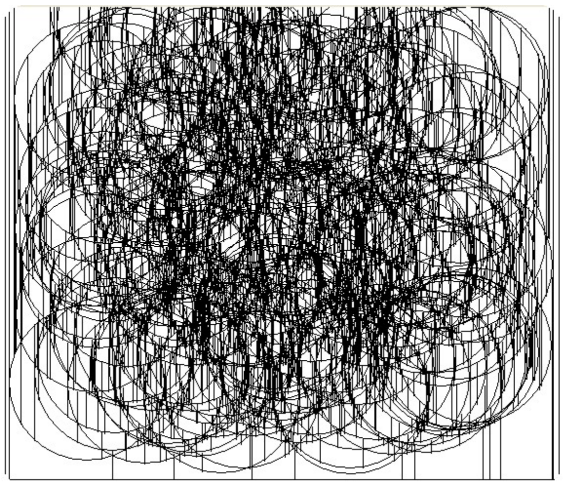


Fig. 42. Dr. Golden's 200-node instance with radius 20 - produces 42543 trapezoids.

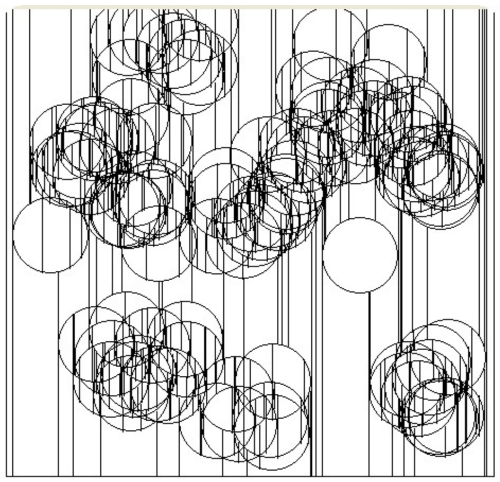


Fig. 43. Dr. Golden's 100-node instance with radius 9 - produces 4049 trapezoids.

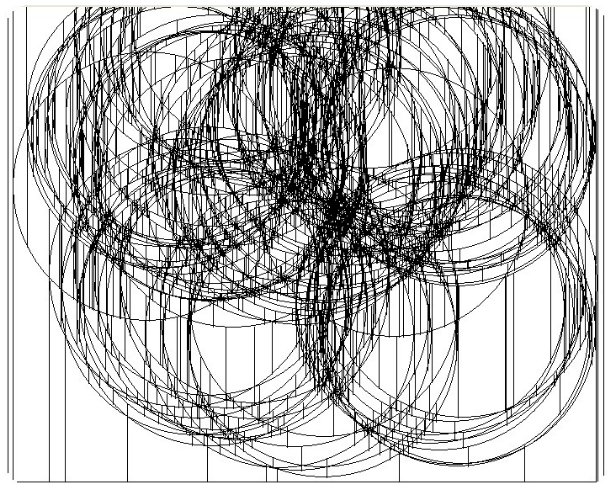


Fig. 44. Dr. Golden's 100-node instance with radius 40 - produces 26279 trapezoids.

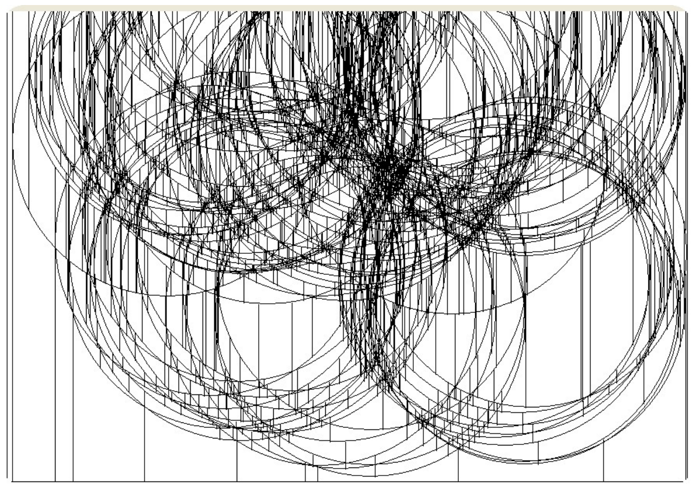


Fig. 45. Dr. Golden's 100-node instance with radius 40 magnified - produces 26279 trapezoids.

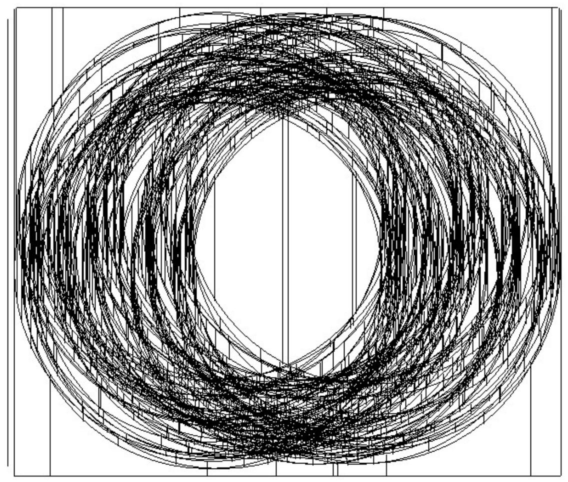


Fig. 46. TSPLIB's kroD100 with radius 12 - produces 29501 trapezoids.

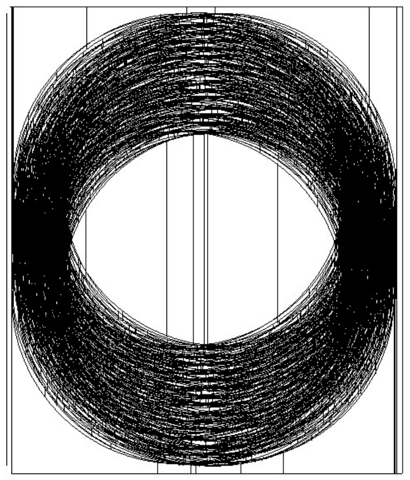


Fig. 47. TSPLIB's rad195 with radius 40 - produces 113077 trapezoids.

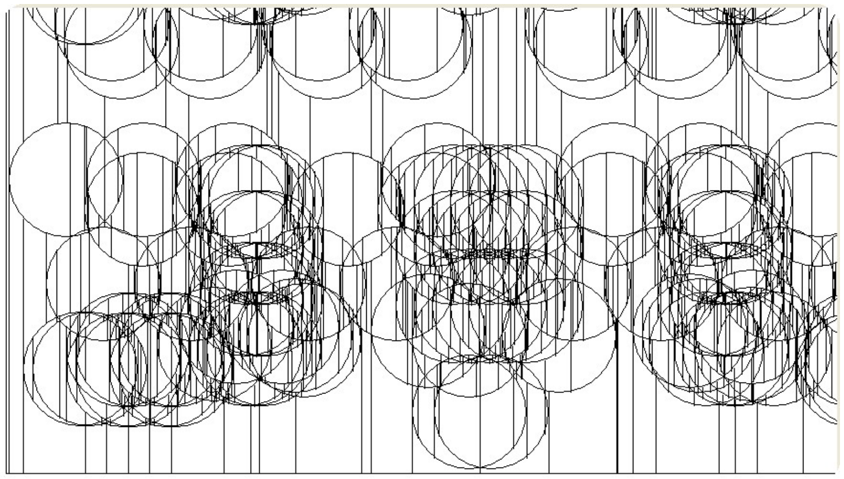


Fig. 48. TSPLIB's lin318 with radius 20 - produces 14597 trapezoids.

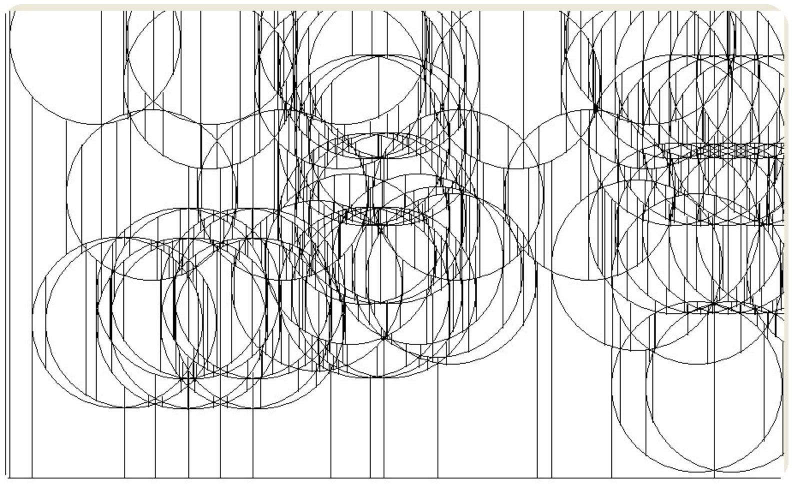


Fig. 49. TSPLIB's lin318 with radius 20 - produces 14597 trapezoids.

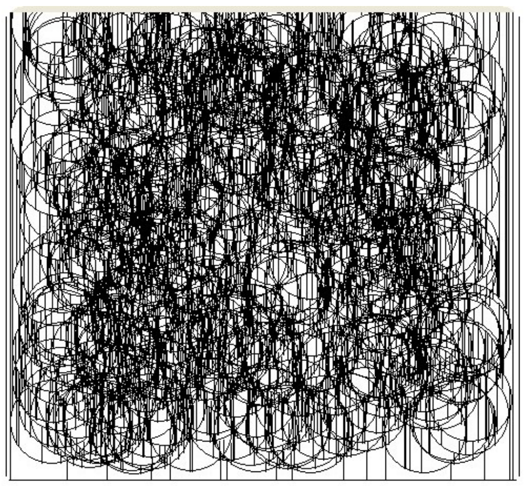


Fig. 50. TSPLIB's rd400 with radius 10 - produces 51241 trapezoids.

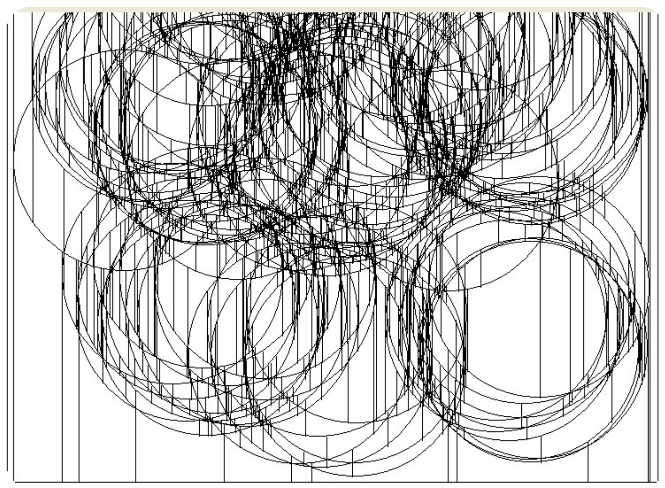


Fig. 51. TSPLIB's pcb442 with radius 25 - produces 14843 trapezoids.

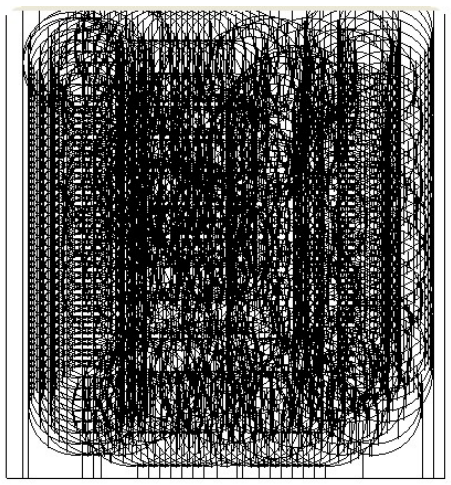


Fig. 52. TSPLIB's pcb442 with radius 4.346 - produces 94251 trapezoids. This is a *drilling* problem from TSPLIB - these are definitely the most difficult to decompose as degeneracies abound.

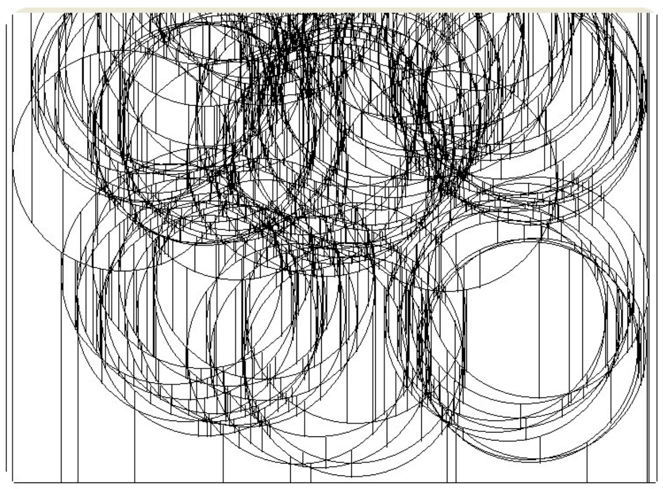


Fig. 53. TSPLIB's pcb442 with radius 25 - produces 14843 trapezoids.

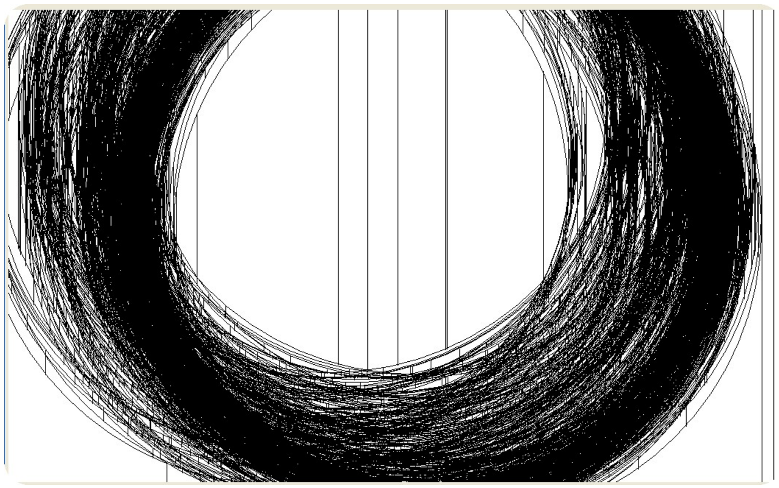


Fig. 54. TSPLIB's d493 with radius 40 - produces 725555 trapezoids.

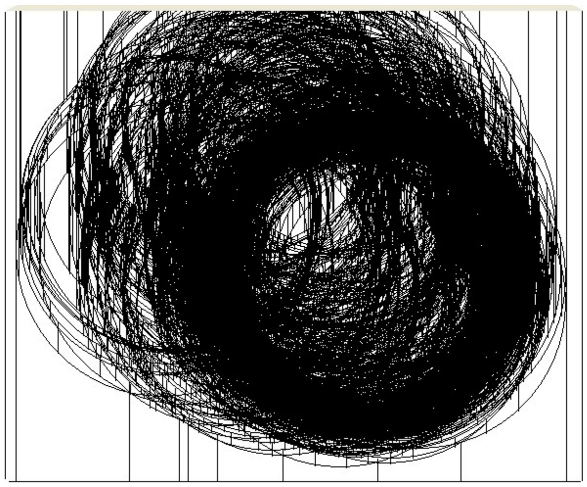


Fig. 55. TSPLIB's d493 with radius 15 - produces 725947 trapezoids.

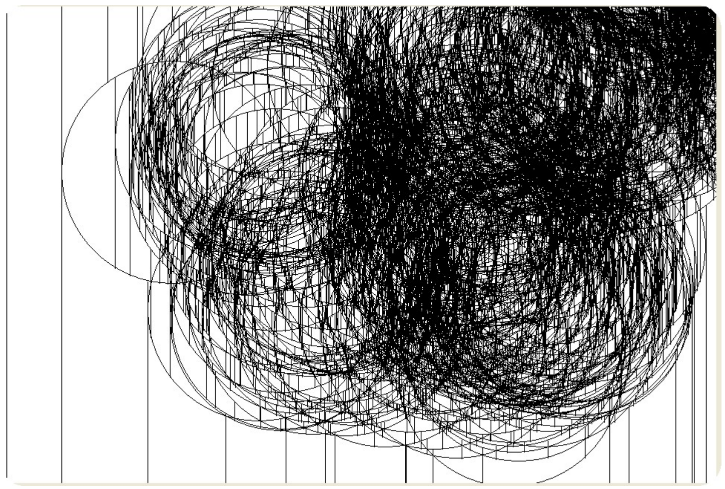


Fig. 56. TSPLIB's dsj1000 with radius 15 - produces 695120 trapezoids.

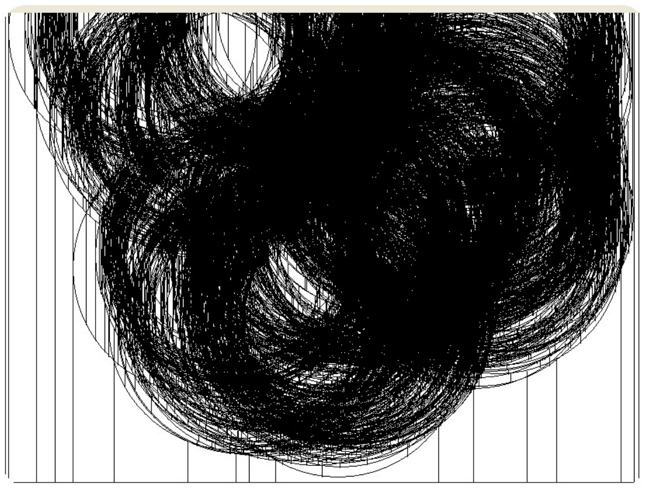


Fig. 57. TSPLIB's dsj1000 with radius 40 - produces 2308090 trapezoids.

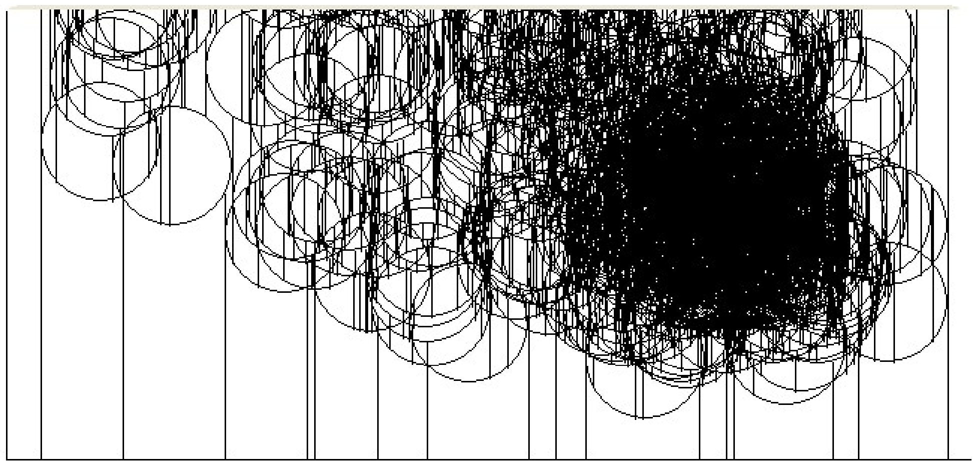


Fig. 58. TSPLIB's d493 with radius 2 - produces 171341 trapezoids.

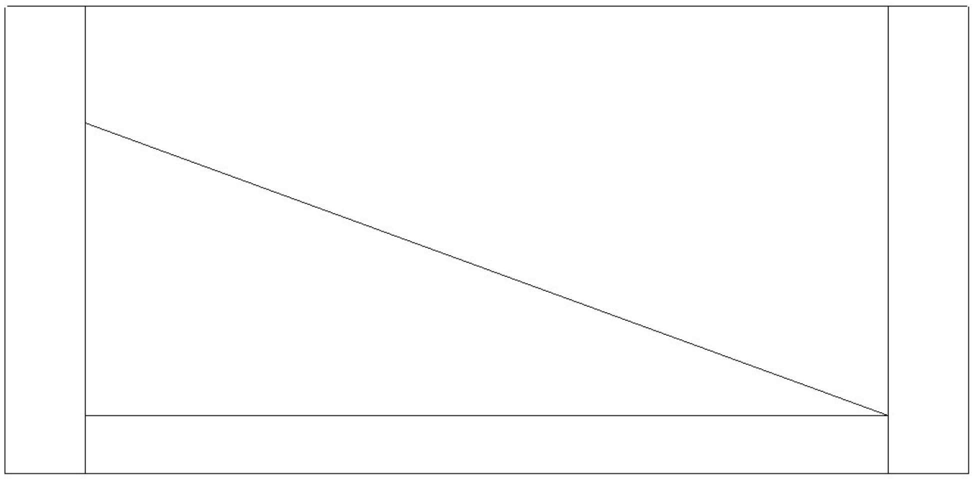


Fig. 59. Allowing segments to have same x coordinate allows for multiple right neighbors and no left neighbors.

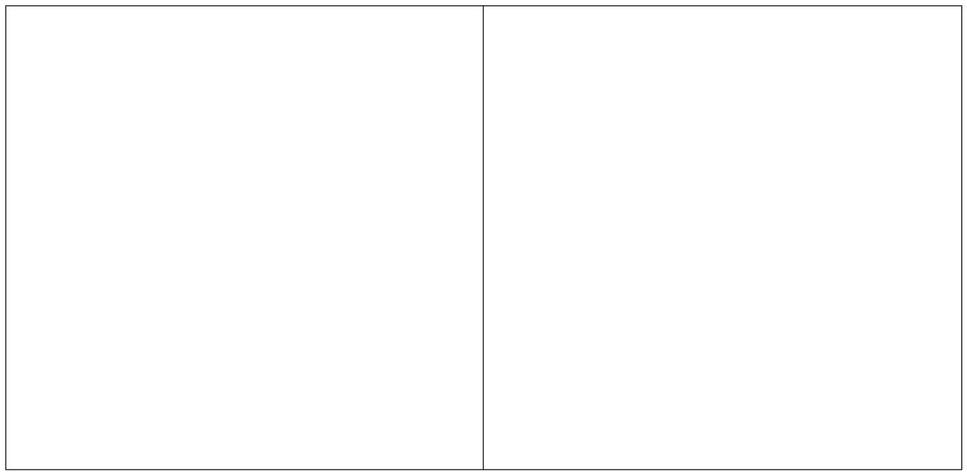


Fig. 60. Here, two disjoint, vertical segments with same x coords, resulting in only two trapezoids created.

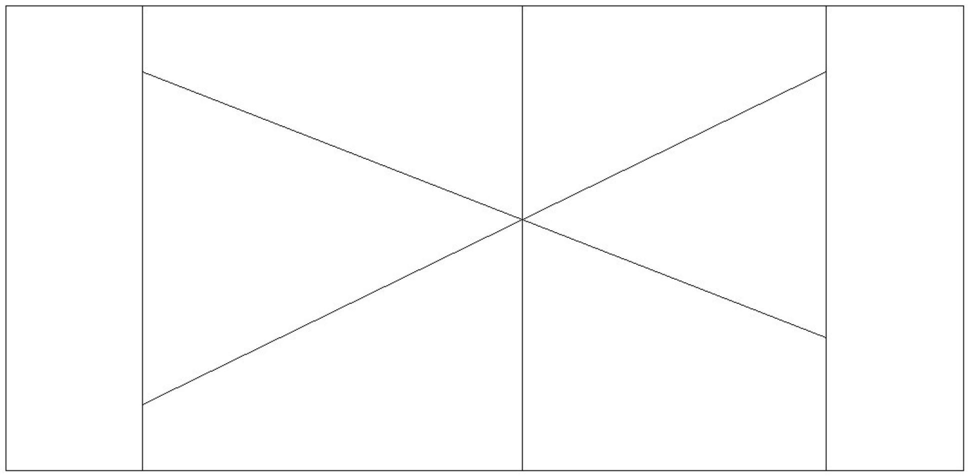


Fig. 61. Intersecting line segments can cause problems.

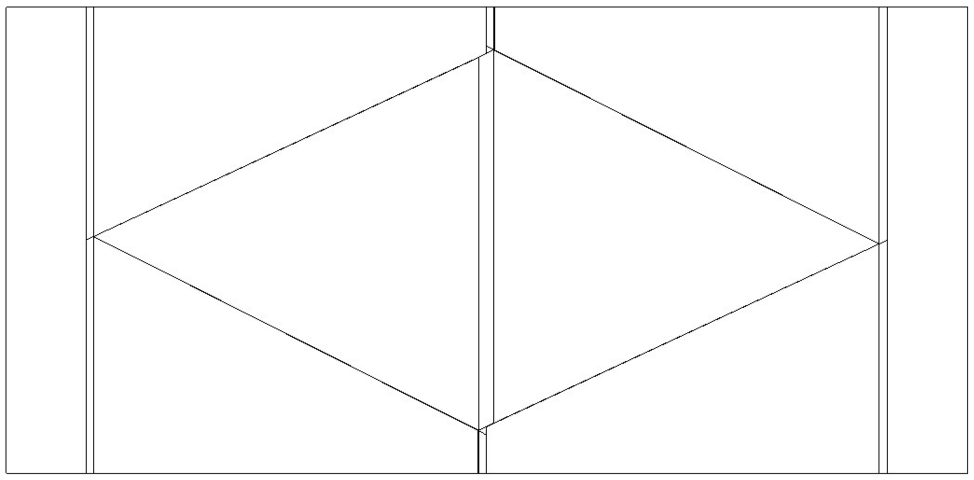


Fig. 62. There's no degeneracy here, but an interesting pattern, nonetheless.

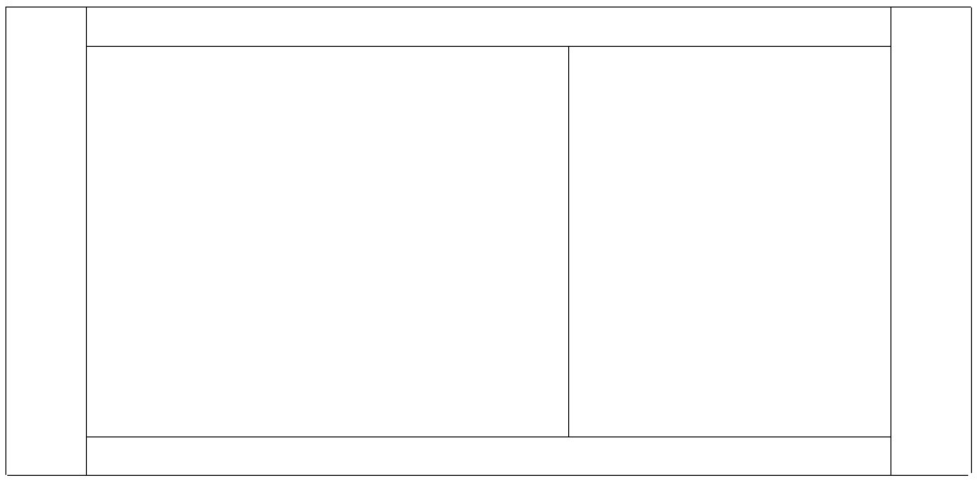


Fig. 63. Horizontal segments with same left and right x coordinates and a vertical segment.

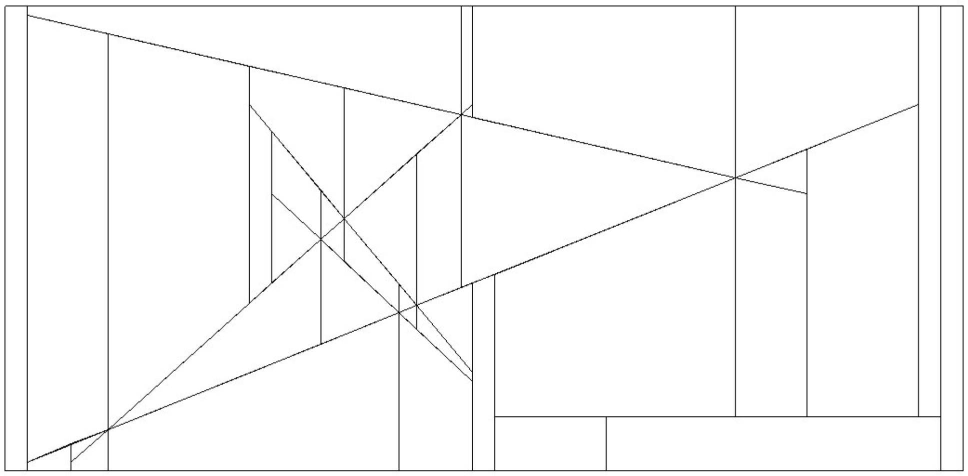


Fig. 64. 2 vertical segments (one overlaps a bullet path) interspersed in 8 segments.

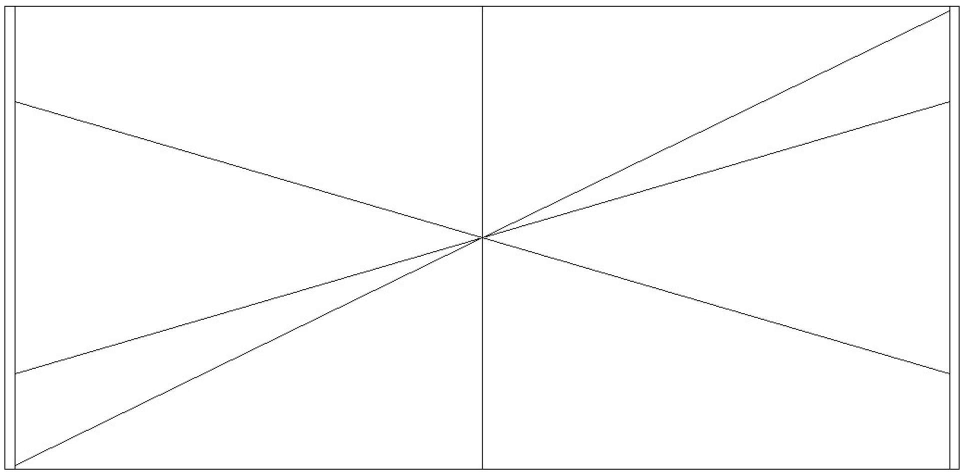


Fig. 65. Four segments intersect at the same spot, but one of them is redundant and so is dropped.

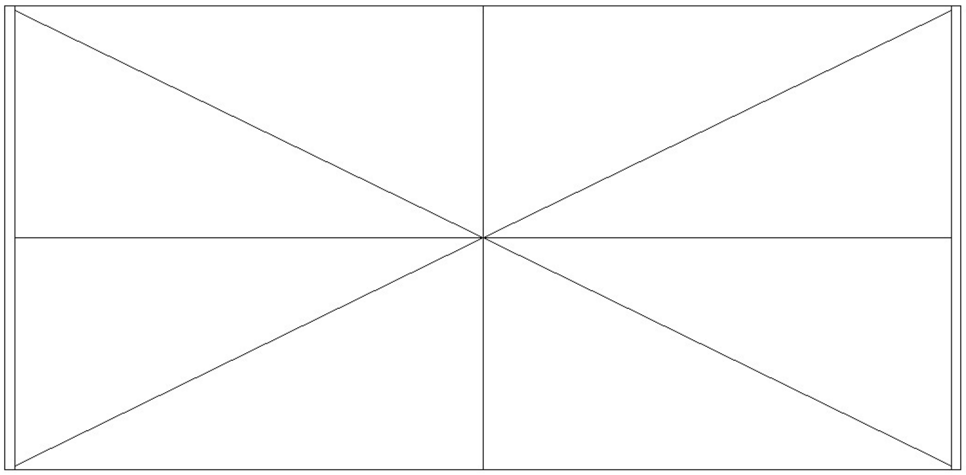


Fig. 66. Four segments intersect at same point - the “vertical” segment is actually as close to vertical as possible without being vertical so there are a few trapezoids here too narrow to see, but they are there.

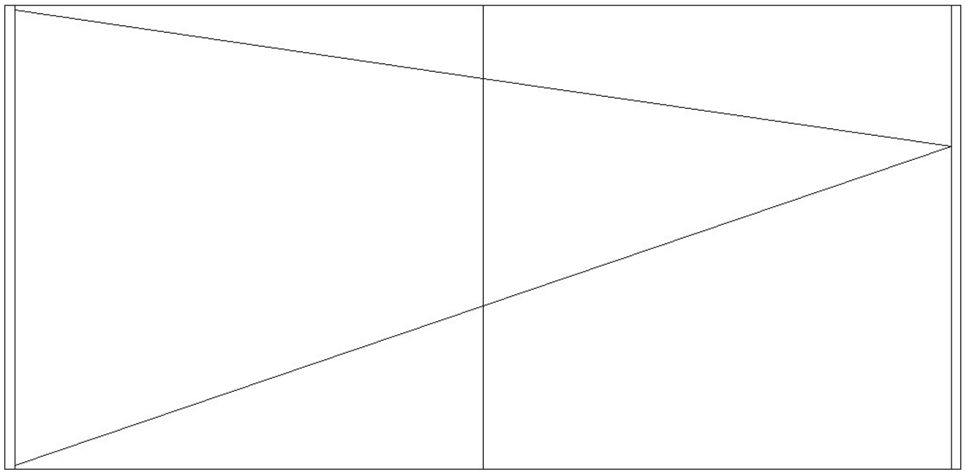


Fig. 67. The vertical line is a segment that extends slightly above the upper non-vertical segment. Its bullet path finished the vertical line all the way to the top.

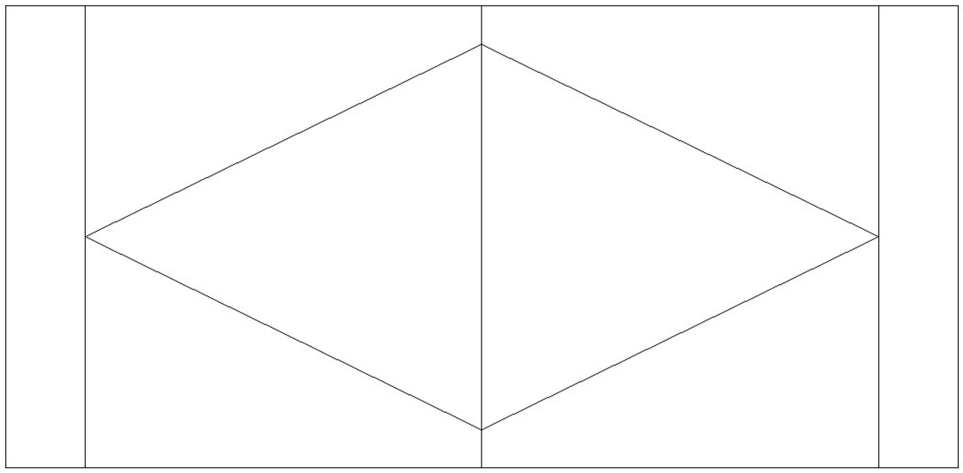


Fig. 68. A diamond figure has multiple same right and left x coordinates to deal with.

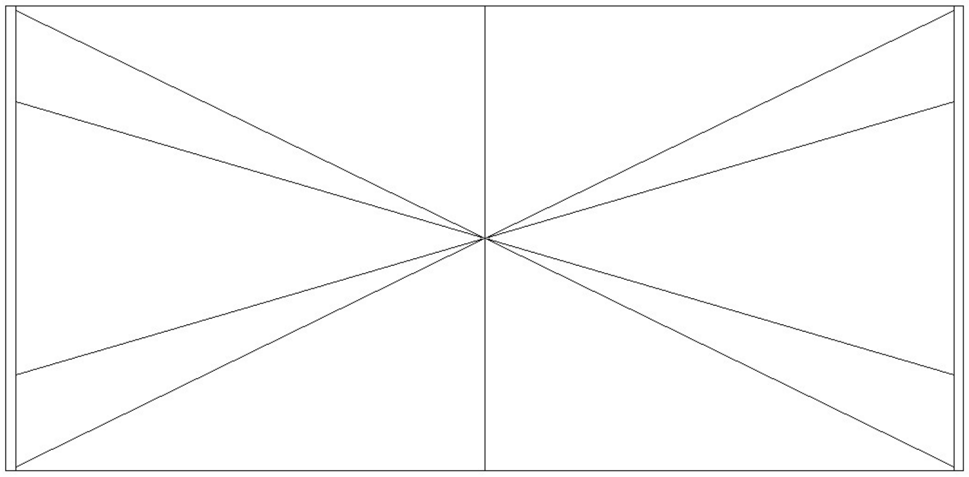


Fig. 69. TSPLIB's d493 with radius 2 - produces 171341 trapezoids.

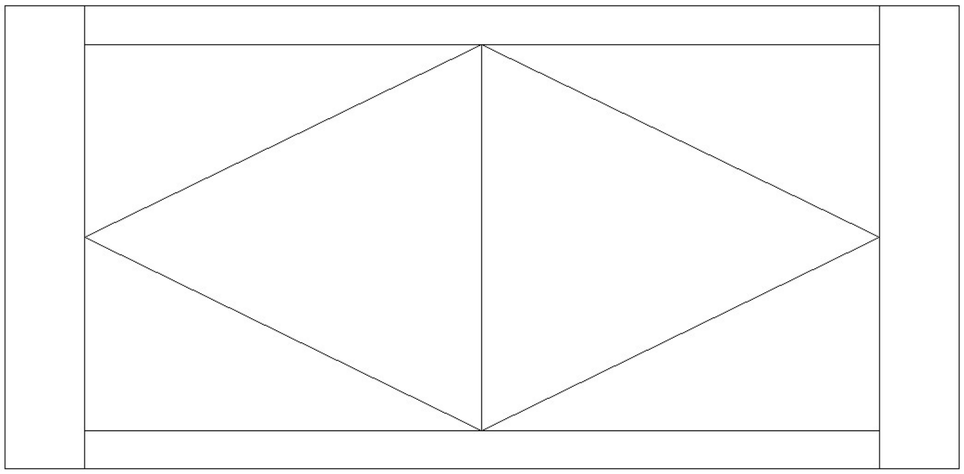


Fig. 70. Diamond shape with added horizontal lines with same x coords.

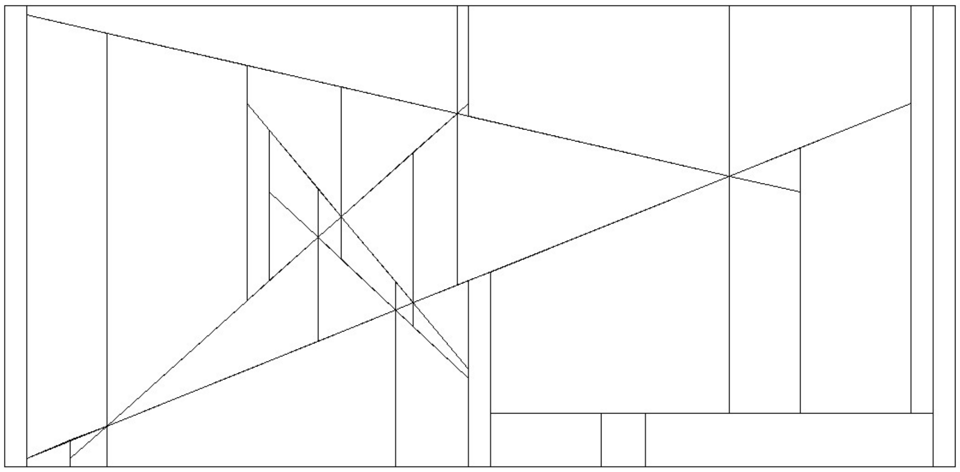


Fig. 71. Same as earlier example but both vertical segments are visible and both have top y coordinate same as a horizontal segment.

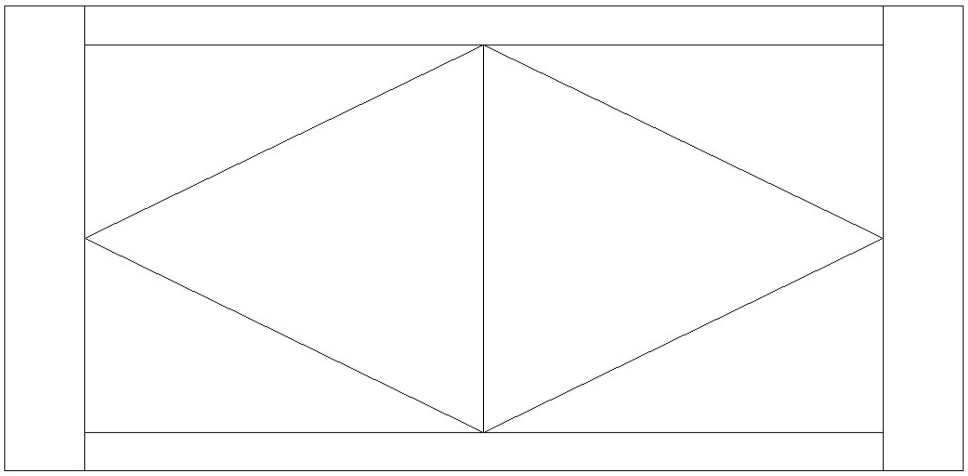


Fig. 72. Same as earlier but the middle bullet path is also a vertical segment.

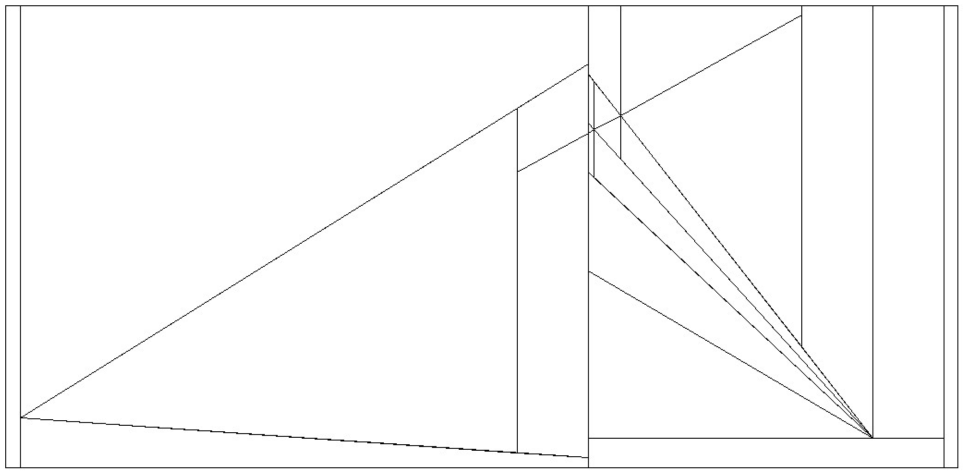


Fig. 73. This positive slope segment in the middle has to choose amongst multiple general right neighbors.

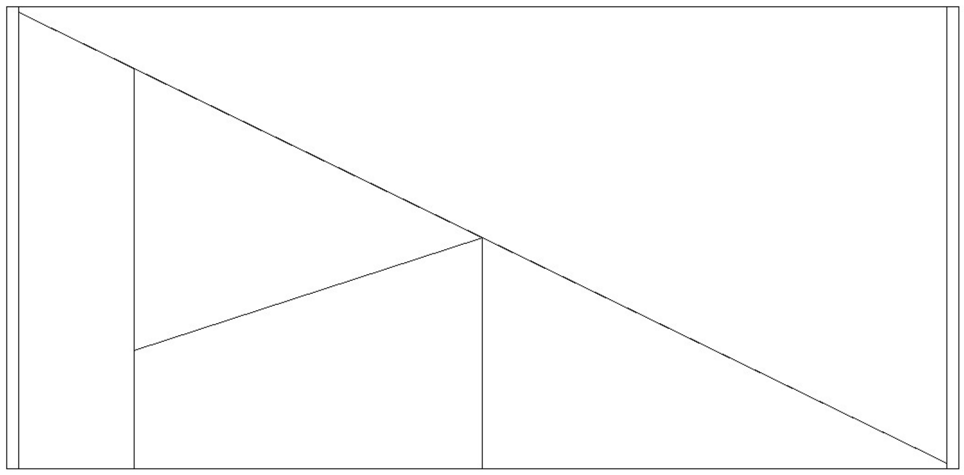


Fig. 74. Sometimes a segment ends right at its intersection with another line - can be tricky.

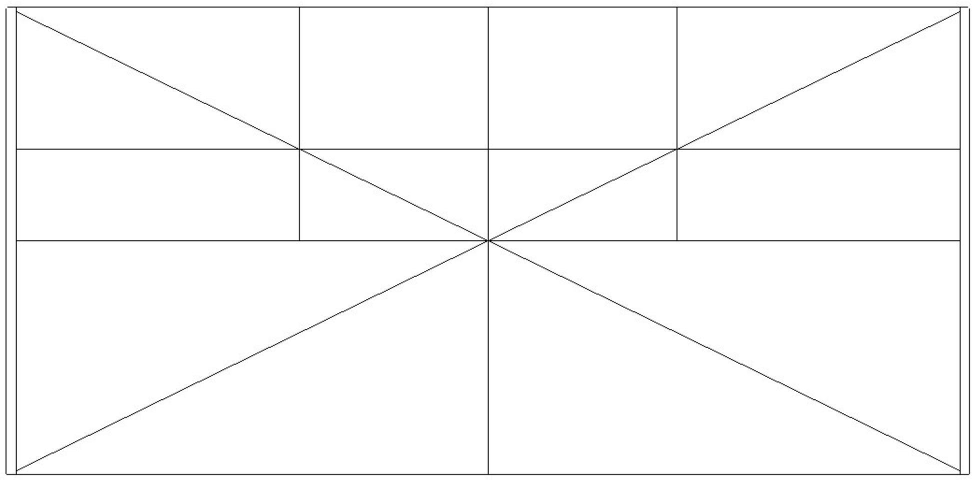


Fig. 75. Multiple segments intersect in the same point, have the same left and right x coordinates and have a vertical segment to contend with, addtionally.

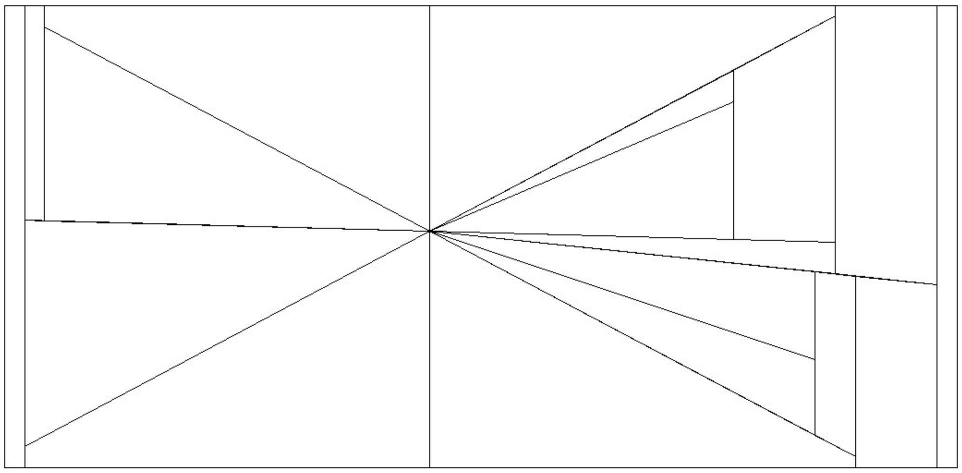


Fig. 76. Same as Figure 13 but one segment has negative instead of positive slope.