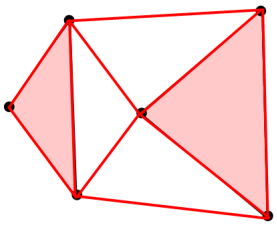
Testing documentation

Most of the code's classes have a set of *unit tests* that are run automatically after compiling. GUI and geometry tools are not tested. Following are the descriptions of all implemented automatic tests, all of which pass everytime.

VertexContainer

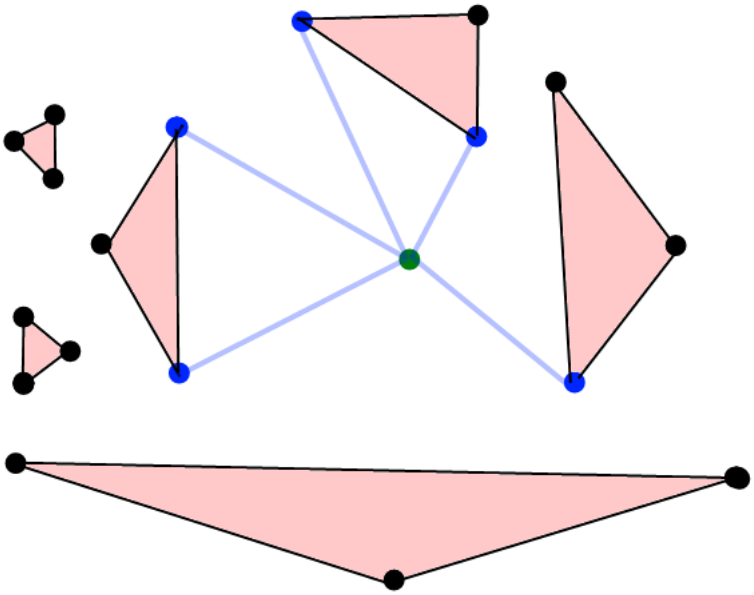
Tests if Vertex and Point are added at coordinates (3,5)
Tests if Vertex and Point can be removed.
Checks if an *edge* between two vertices can be toggled. *toggleEdge* is called two times, after first call edge should exist but after second call it shouldn't. Another test calls *toggleEdge* 1000 times.
Another test calls *buildGraph*, simultaneously testing AngleElimination algorithm. Test places points at static positions, calls *buildGraph* and checks if the graph is correct. Visualisation of the test:



Places three points on a triangle clockwise, which should form a wall polygon. Also places three points counter clockwise, which should form a non-wall polygon.
Another test places points to form a non-wall triangle and then calls *invertShape*, after which triangle should be a wall.

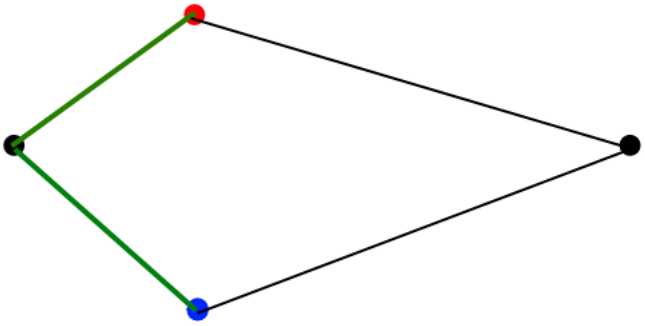
AngleElimination

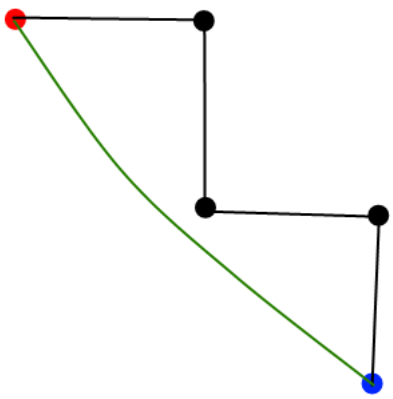
Places a vertex v and several triangles in a specific way, after which AngleElimination is called from the point of view of v. Test checks if algorithm returned a correct set of *unobstructed* points. Visualisation of the test, where v is colored green and unobsturcted points are colored blue:

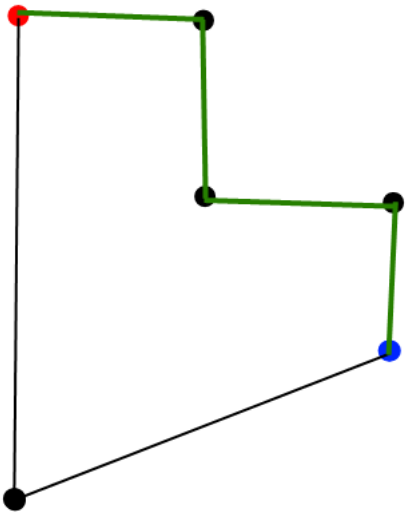


Dijkstra

Three different graphs are built with end points, after which test checks if Dijkstra's returned path is the shortest one. Visualisations of the test, where starting vertex is colored red, ending vertex blue, and shortest path is colored green:







Heap

Checks if Heap always *pops* the smallest element. 1000 random integers are *inserted* into the heap, after which heap is popped until it's *empty*. Test passes if next popped integer is never smaller than previously popped integer.

Checks if Heap is empty when *clear* is called. Before calling *clear* Heap is filled with random integers.

1337 random integers are added to the Heap, after which *size* is called, which should return 1337.
Checks if calling *pop* will reduce Heap's *size* by one.
Heap's original array size is 15. Test inserts 14 elements, after which array size shouldn't change. After inserting one more element, array size should be doubled.
Following integers are inserted in following order: 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 6. Heap's array should now be in the following order: 1, 3, 5, 6, 9, 11, 13, 7, 17, 19, 21, 23, 25, 27, 29, 15.
Integers from 1-15 are inserted to the Heap in ascending order. Heap is popped, after which Heap's array should be in the following order: 2, 4, 3, 8, 5, 6, 7, 15, 9, 10, 11, 12, 13, 14.
Following integers are inserted in following order: 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29. Then, 19's value is changed to 2. Heap's array should now be in the following order: 1, 2, 5, 7, 3, 11, 13, 15, 17, 9, 21, 23, 25, 27, 29.
Following integers are inserted in following order: 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29. Then, 5's value is changed to 26. Heap's array should now be in the following order: 1, 3, 11, 7, 9, 23, 13, 15, 17, 19, 21, 26, 25, 27, 29.

Note about changing values of existing integers: Integers had to be reimplemented in order to have two identifiable attributes. One is its value, which is what is tested and represented in this document. Other is its ID, which can never be changed. This is needed in order to find the correct integer from the TreeMap with indexes.

1337 integers are added to the heap, one of them is 1, which is the smallest integer in the Heap. *peek* is called 1337 times. Returned value should be 1 every time, and Heap's size shouldn't change.
TreeMap's contents are checked. For every added element e, TreeMap should return an index i, where $array[i] = e$. *Note: test works correctly if all integers in the Heap are unique.*

LinkedList

Several elements are *added* and then *iterated*. Test checks if correct elements are returned in correct order. Also, after all elements are iterated, *getNext* should return null.
1337 elements are added, after which *size* should return 1337.
1337 elements are added and *clear* is called. *isEmpty* should return true.

clear is called, after which *getNext* should return null.
1337 elements are added, *clear* is called and 13 elements are added after that. Now *size* should return 13.
1337 elements are added and *getNext* is called 1337 times. *getNext* should never return null.
1337 elements are added and *hasNext* is called 1337 times paired with *getNext*. It should never be false, except if called for the 1338'th time.
4 elements are added and iterated. After calling *reset*, LinkedList should be able to be iterated another time.
1338 elements are added and *getNext* is called 478 times. After calling *reset*, *getNext* should return the first element.

Point

Point is placed at (0,0). Left and right neighbours are then placed at several locations, while forming a 180° angle. Point's angle should return 180° every time.
Point is placed, whose angle is less than 180° and another with more than 180° angle. First point should return false for *isVertex*, while other should return true.

Queue

For every test, four elements are *enqueued*.
dequeue is called five times. Each time *dequeue* should return a correct value, the fifth time should return null.
dequeue is called three times, after which new element is *enqueued*. Now *dequeue* should return the fourth element, and next *dequeue* should return the newly added element. After that *dequeue* should return null.
After calling *clear isEmpty* should return true.
clear is called and a new element is enqueued. Now *dequeue* should return the new element. Another *dequeue* returns null.
Queue is cleared and elements are queued a random r amount of times. Then elements are dequeued a random b amount of times, where b < r. *size* should return r - b. This test is then repeated 10000 times.

TreeMap

Three key-value pairs are *put* into the TreeMap for every test.
Calling *get* for every *key* returns a appropriate *value*.
If *put* is called for existing key, that key should return the new value.
If *get* is called for a key that doesn't exist in the TreeMap, null should be returned.

Tree

1024 integers are *added* in ascending order. Tree's height should be less than 20.
1024 integers are added in descending order. Tree's height should be less than 20.
1024 integers are added in random order. Tree's height should be less than 20.
2048 integers are added in ascending order and 1024 first integers are *removed*. Tree's height should be less than 20.
2048 integers are added in random order and 1024 random integers are removed. Tree's height should be less than 20.
2048 integers are added and *clear* is called. *isEmpty* should return true.
2048 integers are added in ascending order and *toLinkedList* is called. Returned LinkedList should contain all integers from 1 to 2048.
Integer 0 is added among with 2048 random integers, which are bigger than 0. *getMin* should return 0.
Tree is cleared. Now *getMin* should return null.
Tree is cleared and integers are added in ascending order a random r amount of times. Then elements are removed in ascending order a random b amount of times, where b < r. *size* should return r - b. This test is then repeated 10000 times.
Tree is cleared. 1000 integers are added in ascending order. Tree should now contain an integer, value of which is between 1 and 1000. This test is then repeated 1000 times.
Integers from 1 to 1000 are added. *contains(-1)* should return false.

Vertex

Two vertices are placed. One vertex is *added* for the other as *adjacent*. Now vertices should have each other in their *adjacent tree*.
One vertex is placed and the same vertex is added as adjacent. Adjacent tree should be empty.
Two vertices are placed. One vertex is added for the other as adjacent. Then, one vertex is removed from other's adjacent tree. Now both adjacent trees should be empty.
A vertex is placed. 1000 vertices are added to the vertex as adjacents. After calling *removeAllAdjacents*, adjacent tree must be empty.
One vertex is placed at (0,0), other is placed at (3,4). When measuring distance from either vertex to the other, distance should be 5.
One vertex v_1 is placed at (0,0), other v_2 is placed at (1,1). Direction from v_1 to v_2 must be $45°$, and $-135°$ vice versa.

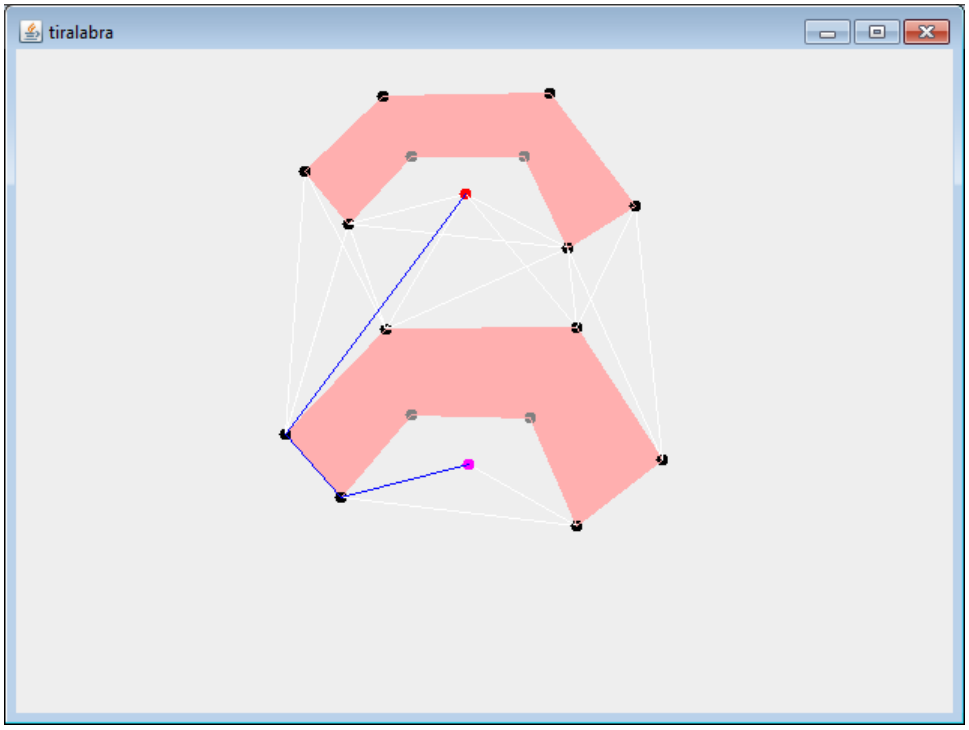
src is placed at (0,0), *v* is placed at (1,0), *left* at (1,-1) and *right* at (1,1). Now, *src* should recognise that *v* is between *left* and *right*.
src is placed at (0,0), *v* is placed at (-1,0), *left* at (-1,1) and *right* at (-1,-1). Now, *src* should recognise that *v* is between *left* and *right*.
src is placed at (0,0), *v* is placed at (-1,0), *left* at (1,-1) and *right* at (1,1). Now, *src* should recognise that *v* **isn't** between *left* and *right*.
src is placed at (0,0), *v* is placed at (1,0), *left* at (-1,1) and *right* at (-1,-1). Now, *src* should recognise that *v* isn't between *left* and *right*.

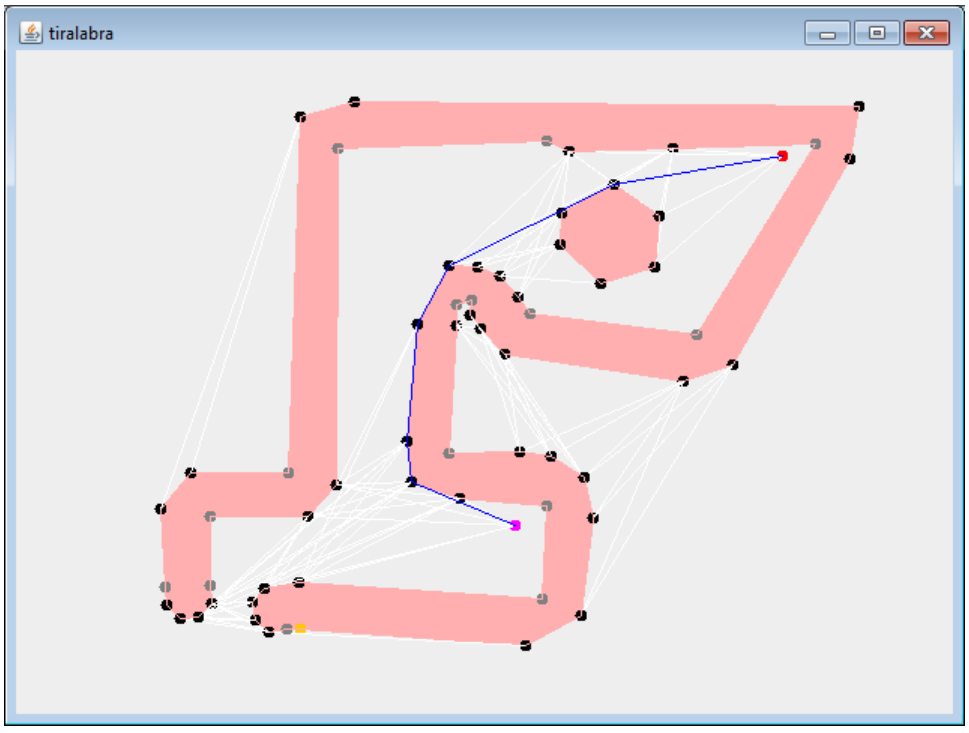
Tools

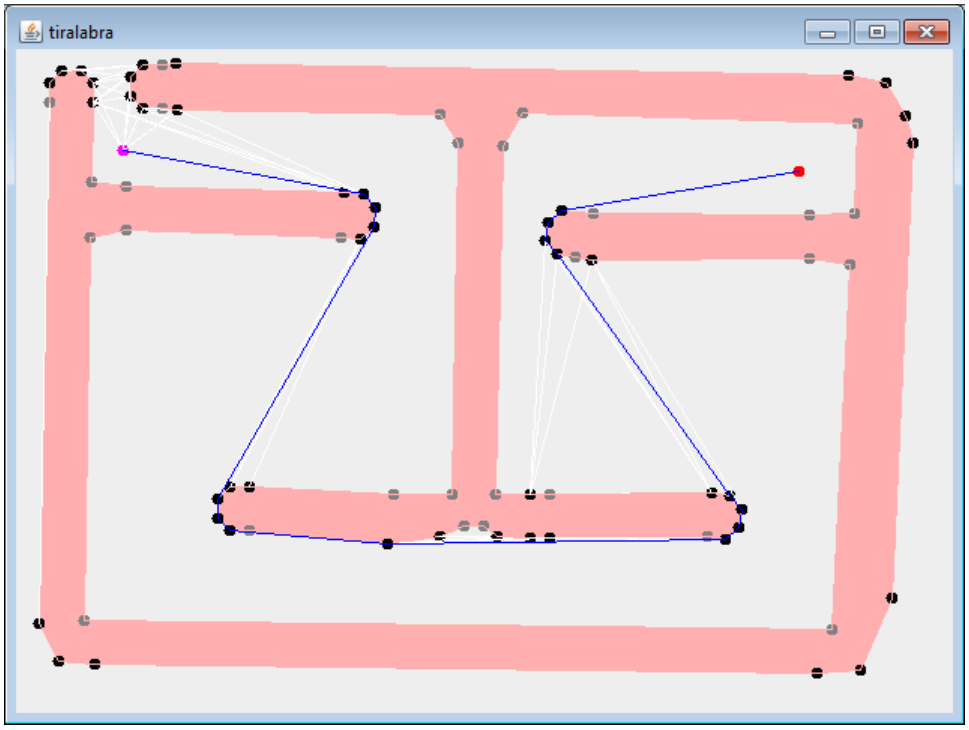
round is tested with several parameters. Specific parameters should be evident from test's code.
$0°$ should be between $-45°$ and $45°$
$180°$ should be between $135°$ and $-135°$
$-180°$ shouldn't be between $-45°$ and $45°$
$0°$ shouldn't be between $135°$ and $-135°$

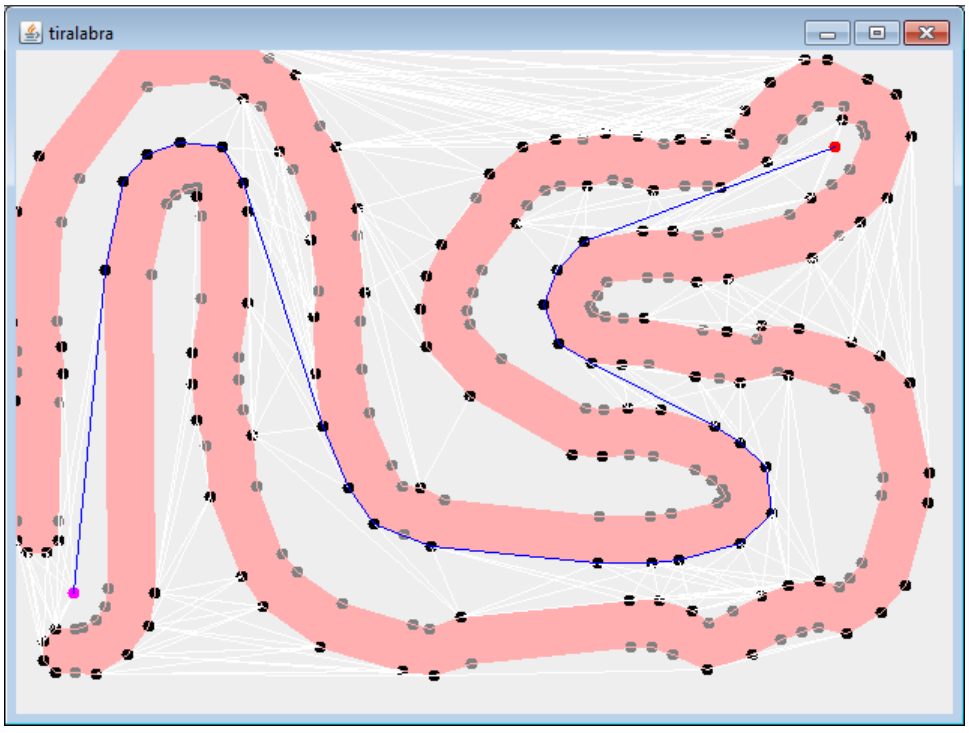
Empirical tests

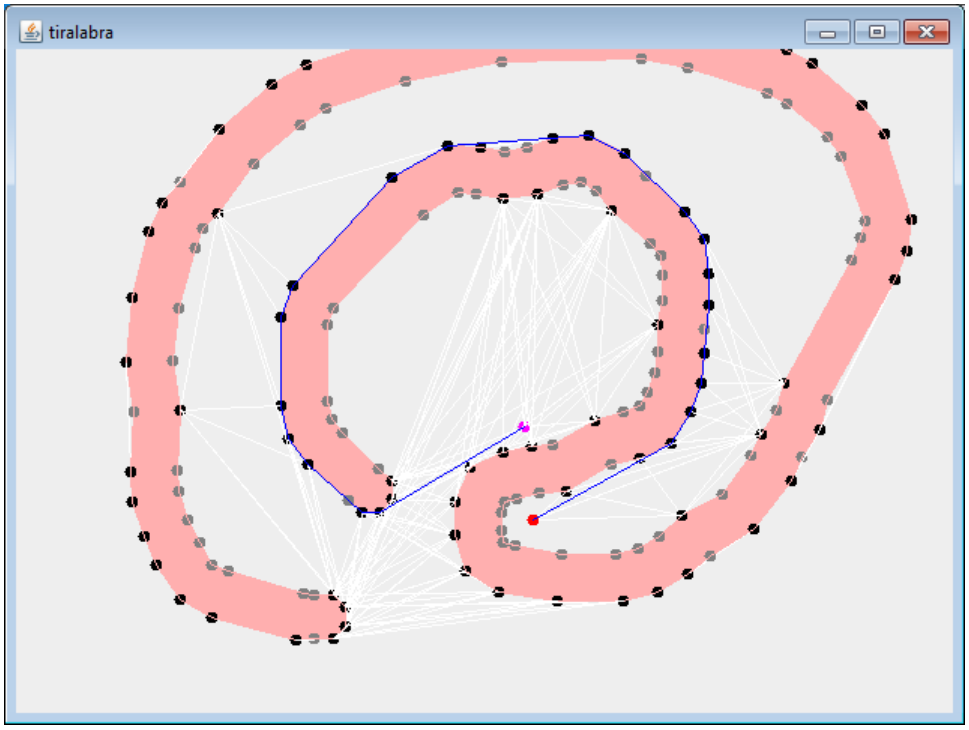
Because of the complexity of the problem, many elements of the program where many systems are integrated are better tested by empirical means. The following are screenshots of the program. Start and end vertices are colored red and magenta, walls are colored pink. Points with reflex angles are colored black, other points are gray. Graph is colored white, shortest path is colored blue.

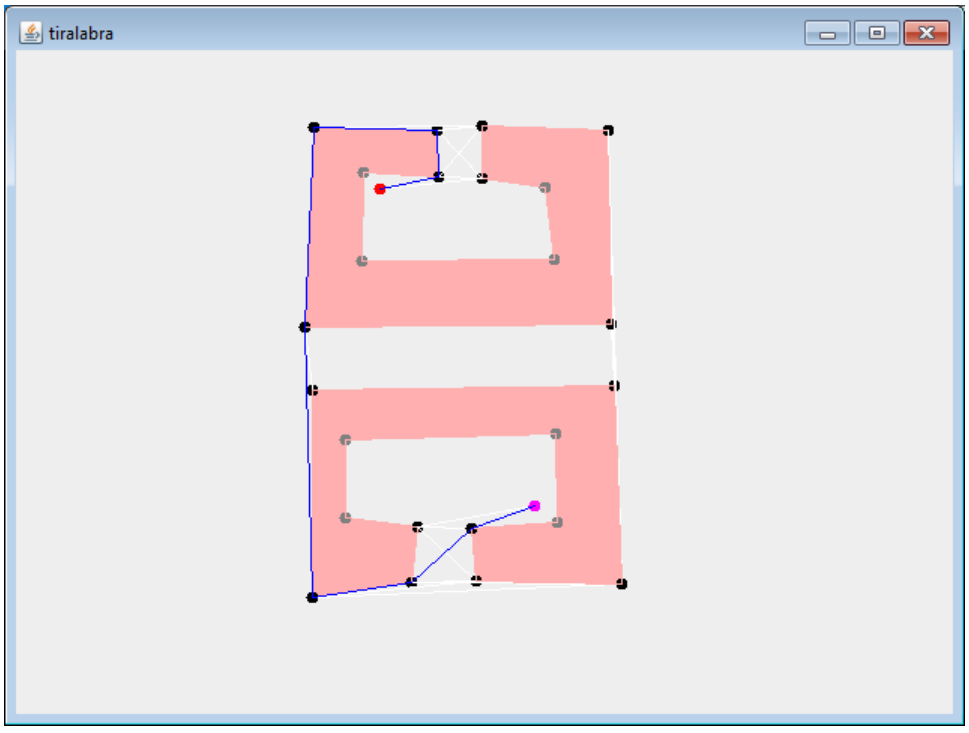

tiralabra

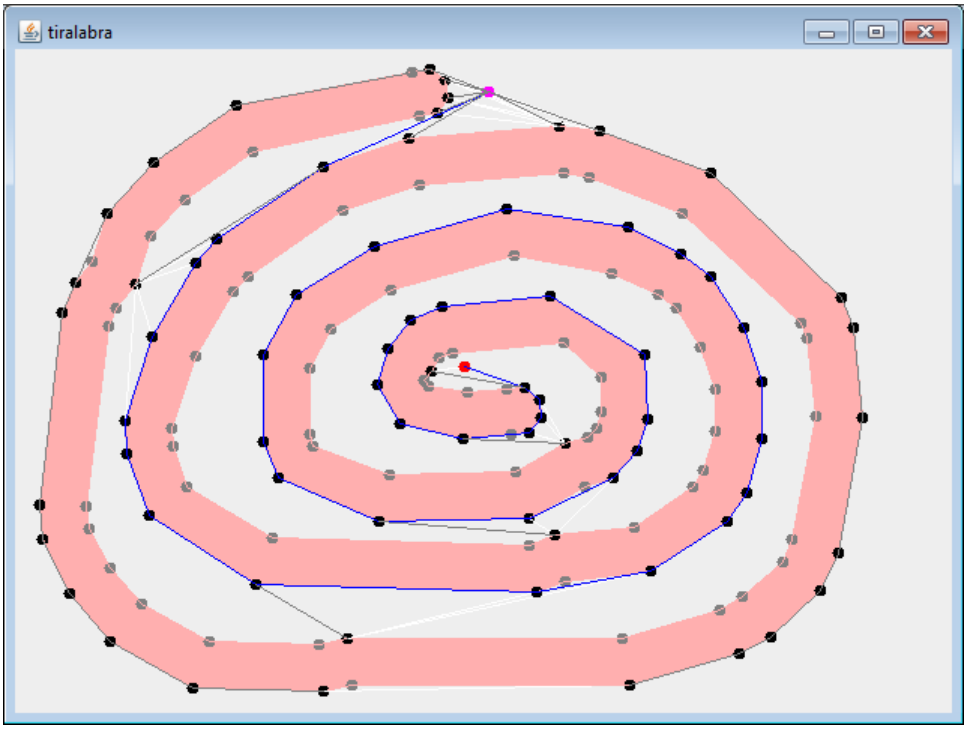

tiralabra


tiralabra

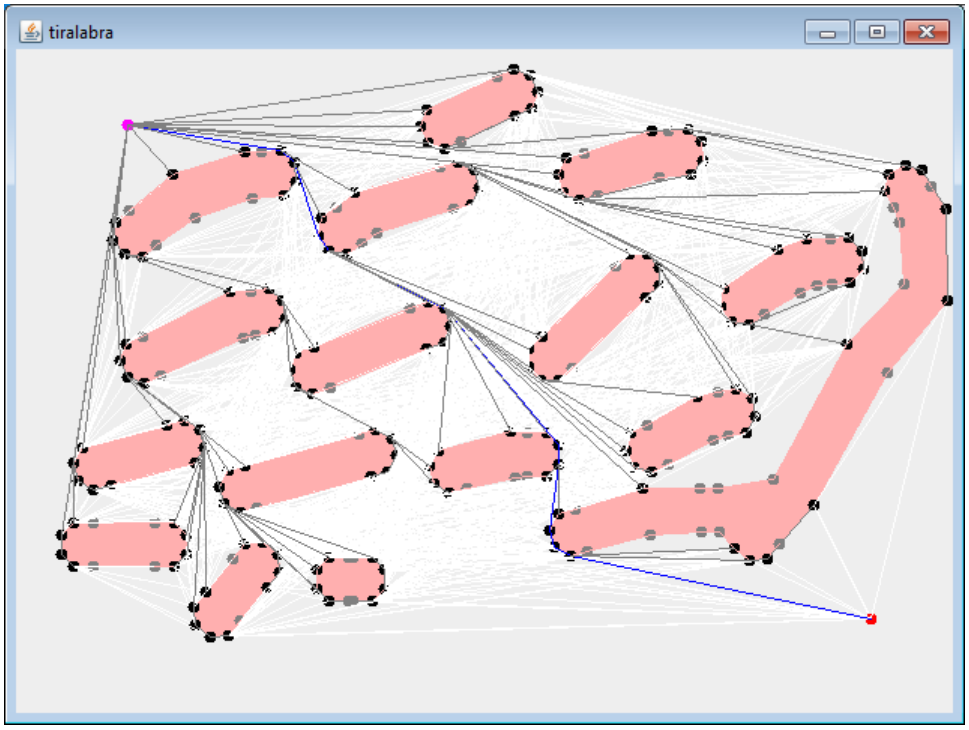

tiralabra

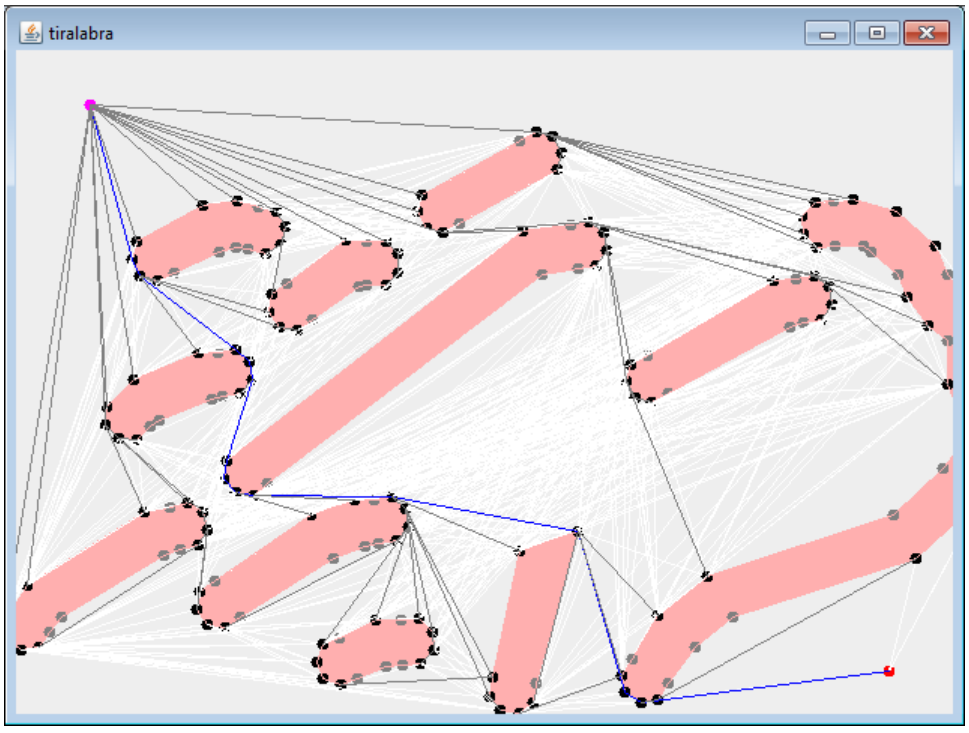

tiralabra


tiralabra



Next screenshots visualize the A* algorithm, where not all edges in the graph are processed while finding the shortest path. Processed edges are colored gray.


tiralabra


tiralabra