

Jagiellonian University

Projekt: Ukryty skarb Raport

Maryia Babinskaya
Illia Dovhalenko

June, 2024

Spis treści

1 Problem	3
2 Założenia	3
3 Struktura naszej wyspy	3
3.1 Implementacja class Node w języku Java	3
3.2 Implementacja class Edge w języku Java	4
3.3 Implementacja Graph w języku Java	5
3.4 Implementacja GraphIsland w języku Java	6
4 Rodzaj wysp	8
4.1 SUPER idealna wyspa	8
4.2 Idealna wyspa	8
4.3 REAL wyspa	9
5 Obejścia grafu	9
5.1 Wężykiem	9
5.2 Spiralą od krawędzi do środka	11
5.3 Spiralą od środka do krawędzi	12
5.4 Random	16
5.5 BFS/DFS	17
6 Wynik	20

1 Problem

Wiadomo, że na wyspie znajduje się ukryty skarb, jednak nie wiadomo gdzie.

Cel projektu: zaproponowanie algorytmów poszukiwania takiego skarbu.

2 Założenia

01. Wyspa jest nieznana (czyli nie wiemy gdzie będzie przeszkoda)
02. Graf jest spójny
03. Losujemy ilość wyrzuconych krawędzi od 0 do (ilość wszystkich krawędzi/2)
04. Skarb z większym prawdopodobieństwem jest w centrum

3 Struktura naszej wyspy

Wyspa jest przedstawiona w postaci grafu nieskierowanego ze wszystkimi krawędziami z wagą 1.

Mamy dwie struktury opisujące naszą wyspę: **Graph** oraz **GraphIsland**, gdzie Graph ma wszystkie krawędzi i wszystkie wierzchołki, a GraphIsland ma wszystkie wierzchołki, ale nie wszystkie krawędzi.

3.1 Implementacja class Node w języku Java

```
1 import java.util.*;
2
3 public class Node {
4
5     private Node up, right, down, left;
6     private int x, y;
7     private boolean visited, finalDestination;
8
9     public Node(int x_, int y_) {
10         x = x_;
11         y = y_;
12     }
13
14     public int getX() {
15         return x;
16     }
17
18     public int getY() {
19         return y;
20     }
21
22     public void setUp(Node node){
23         up = node;
24     }
25
26     public void setDown(Node node){
27         down = node;
28     }
29
30     public void setRight(Node node){
31         right = node;
32     }
33
34     public void setLeft(Node node){
35         left = node;
36     }
37 }
```

```

38     public Node getRight(){
39         return right;
40     }
41
42     public Node getDown(){
43         return down;
44     }
45
46     public Node getUp(){
47         return up;
48     }
49
50     public Node getLeft(){
51         return left;
52     }
53
54     @Override
55     public String toString(){
56         return "(" + y + " " + x + ")";
57     }
58
59     public void setFinalDestination(){
60         finalDestination = true;
61     }
62
63     public boolean isFinalDestination(){
64         return finalDestination;
65     }
66
67     public void setVisited(){
68         visited = true;
69     }
70
71     public boolean getVisited(){
72         return visited;
73     }
74
75     public Node getUnVisitedNode(){
76         List<Node> list = new ArrayList<>();
77         if (left != null && !left.visited)
78             list.add(left);
79         if (right != null && !right.visited)
80             list.add(right);
81         if (up != null && !up.visited)
82             list.add(up);
83         if (down != null && !down.visited)
84             list.add(down);
85
86         if(!list.isEmpty()) {
87             Random rndm = new Random();
88             int i = rndm.nextInt(list.size());
89             return list.get(i);
90         } else
91             return null;
92     }
93 }

```

3.2 Implementacja class Edge w języku Java

```

1  class Edge {
2      Node node1, node2;
3      boolean isVertical;
4
5      public Edge(Node node1, Node node2) {
6          this.node1 = node1;

```

```

7         this.node2 = node2;
8         this.isVertical = node1.getX() == node2.getX();
9     }
10
11     public void remove() {
12         if (isVertical) {
13             node1.setDown(null);
14             node2.setUp(null);
15         } else {
16             node1.setRight(null);
17             node2.setLeft(null);
18         }
19     }
20
21     public void restore() {
22         if (isVertical) {
23             node1.setDown(node2);
24             node2.setUp(node1);
25         } else {
26             node1.setRight(node2);
27             node2.setLeft(node1);
28         }
29     }
30 }

```

3.3 Implementacja Graph w języku Java

```

1  import java.util.ArrayList;
2  import java.util.Random;
3  import java.util.Stack;
4
5  public class Graph {
6      int n, m;
7      Node startPoint;
8      ArrayList<ArrayList<Node>> map;
9
10     public Graph(int n_, int m_){
11         n = n_;
12         m = m_;
13         map = new ArrayList<>(n);
14
15         for(int y = 0; y < m; y++){
16             ArrayList<Node> oneRow = new ArrayList<>(n);
17             for (int x = 0; x < n; x++) {
18                 Node newNode = new Node(x, y);
19                 if(x != 0) {
20                     newNode.setLeft(oneRow.get(x - 1));
21                     oneRow.get(x - 1).setRight(newNode);
22                 }
23                 if(y != 0){
24                     newNode.setUp(map.get(y - 1).get(x));
25                     map.get(y - 1).get(x).setDown(newNode);
26                 }
27                 oneRow.add(newNode);
28             }
29             map.add(oneRow);
30         }
31
32         startPoint = map.get(0).get(0);
33         Random random = new Random();
34         int randX = (int) (random.nextGaussian()*200+500);
35         int randY = (int) (random.nextGaussian()*200+500);
36         if(randX>=m) randX=m-1;
37         if(randX<0) randX=0;
38         if(randY>=n) randY=n-1;

```

```

39         if(randY<0) randY=0;
40         map.get(randY).get(randX).setFinalDestination();
41     }
42
43     public void print(){
44         for (ArrayList<Node> row : map) {
45             for (Node node : row) {
46                 System.out.print(node + " ");
47             }
48             System.out.println();
49         }
50     }
51 }

```

3.4 Implementacja GraphIsland w języku Java

```

1  import java.util.ArrayList;
2  import java.util.HashSet;
3  import java.util.List;
4  import java.util.Random;
5  import java.util.Set;
6  import java.util.Stack;
7  import java.util.LinkedList;
8  import java.util.Queue;
9
10
11 public class GraphIsland {
12     int rows, cols;
13     Node startPoint;
14     ArrayList<ArrayList<Node>> grid;
15     Random random = new Random();
16
17     public GraphIsland(int rows, int cols) {
18         this.rows = rows;
19         this.cols = cols;
20         grid = new ArrayList<>(rows);
21
22         for (int y = 0; y < rows; y++) {
23             ArrayList<Node> row = new ArrayList<>(cols);
24
25             for (int x = 0; x < cols; x++) {
26                 Node newNode = new Node(x, y);
27
28                 if (x > 0) {
29                     newNode.setLeft(row.get(x - 1));
30                     row.get(x - 1).setRight(newNode);
31                 }
32
33                 if (y > 0) {
34                     newNode.setUp(grid.get(y - 1).get(x));
35                     grid.get(y - 1).get(x).setDown(newNode);
36                 }
37
38                 row.add(newNode);
39             }
40
41             grid.add(row);
42         }
43
44         startPoint = grid.get(0).get(0);
45         setRandomFinalDestination();
46
47         int max = ((rows - 1) * cols + (cols - 1) * rows) / 2;
48         int numberBrokenEdges = random.nextInt(++max);
49         removeEdges(numberBrokenEdges);

```

```

50     }
51
52     private void setRandomFinalDestination() {
53         Random random = new Random();
54         int randX = (int) (random.nextGaussian()*200+500);
55         int randY = (int) (random.nextGaussian()*200+500);
56         if(randX>=cols) {
57             randX=cols-1;
58         }
59         if(randX<0) {
60             randX=0;
61         }
62         if(randY>=rows) {
63             randY=rows-1;
64         }
65         if(randY<0){
66             randY=0;
67         }
68         grid.get(randY).get(randX).setFinalDestination();
69     }
70
71     private void removeEdges(int numberBrokenEdges) {
72         ArrayList<Edge> edges = new ArrayList<>();
73
74         // Collect all edges
75         for (int y = 0; y < rows; y++) {
76             for (int x = 0; x < cols; x++) {
77                 Node node = grid.get(y).get(x);
78                 if (node.getRight() != null) edges.add(new Edge(node, node.getRight()));
79                 if (node.getDown() != null) edges.add(new Edge(node, node.getDown()));
80             }
81         }
82
83         int removedEdges = 0;
84         while (removedEdges < numberBrokenEdges && !edges.isEmpty()) {
85             int index = random.nextInt(edges.size());
86             Edge edge = edges.get(index);
87             edges.remove(index);
88
89             // Temporarily remove edge
90             edge.remove();
91
92             // Check if graph is still connected
93             if (isConnected()) {
94                 removedEdges++;
95             } else {
96                 // Restore edge if graph is not connected
97                 edge.restore();
98             }
99         }
100     }
101
102
103
104     private boolean isConnected() {
105         Set<Node> visited = new HashSet<>();
106         Stack<Node> stack = new Stack<>();
107         stack.push(startPoint);
108
109         while (!stack.isEmpty()) {
110             Node node = stack.pop();
111             if (!visited.contains(node)) {
112                 visited.add(node);
113                 if (node.getRight() != null && !visited.contains(node.getRight()))
114                     stack.push(node.getRight());
115                 if (node.getDown() != null && !visited.contains(node.getDown()))
116                     stack.push(node.getDown());

```

```

117         if (node.getLeft() != null && !visited.contains(node.getLeft()))
118             stack.push(node.getLeft());
119         if (node.getUp() != null && !visited.contains(node.getUp()))
120             stack.push(node.getUp());
121     }
122 }
123
124 // Check if all nodes are visited
125 for (int y = 0; y < rows; y++) {
126     for (int x = 0; x < cols; x++) {
127         if (!visited.contains(grid.get(y).get(x))) {
128             return false;
129         }
130     }
131 }
132 return true;
133 }
134 }

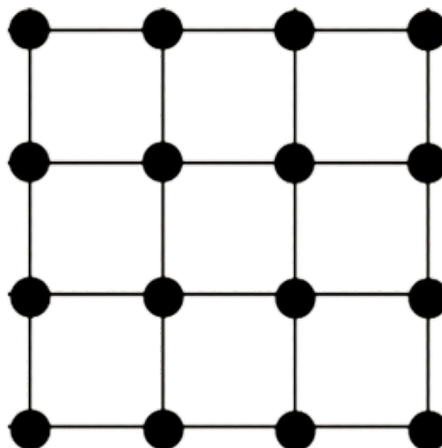
```

4 Rodzaj wysp

01. SUPER idealna wyspa
02. Idealna wyspa
03. REAL wyspa

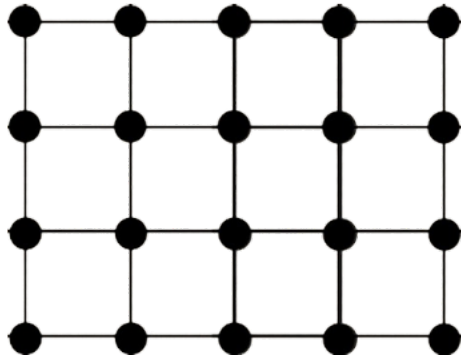
4.1 SUPER idealna wyspa

- 1) Kwadratowa
- 2) Mamy wszystkie krawędzi



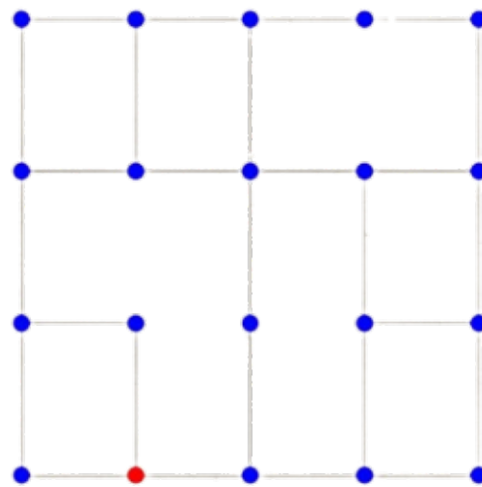
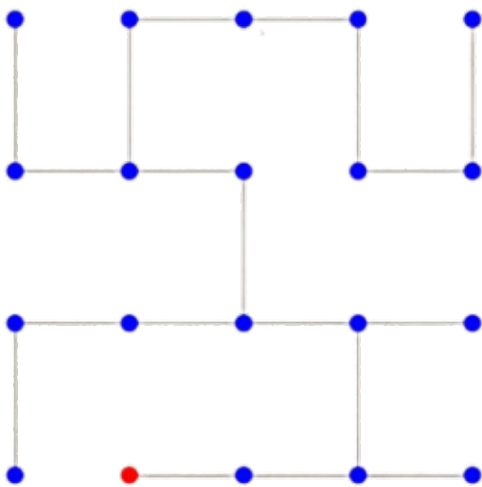
4.2 Idealna wyspa

- 1) Prostokąt
- 2) Mamy wszystkie krawędzi



4.3 REAL wyspa

- 1) Prostokąt/kwadrat
- 2) Mamy nie wszystkie krawędzie (możemy mieć rzeki lub góry, które mogą przeszkadzać w obejściu wyspy)

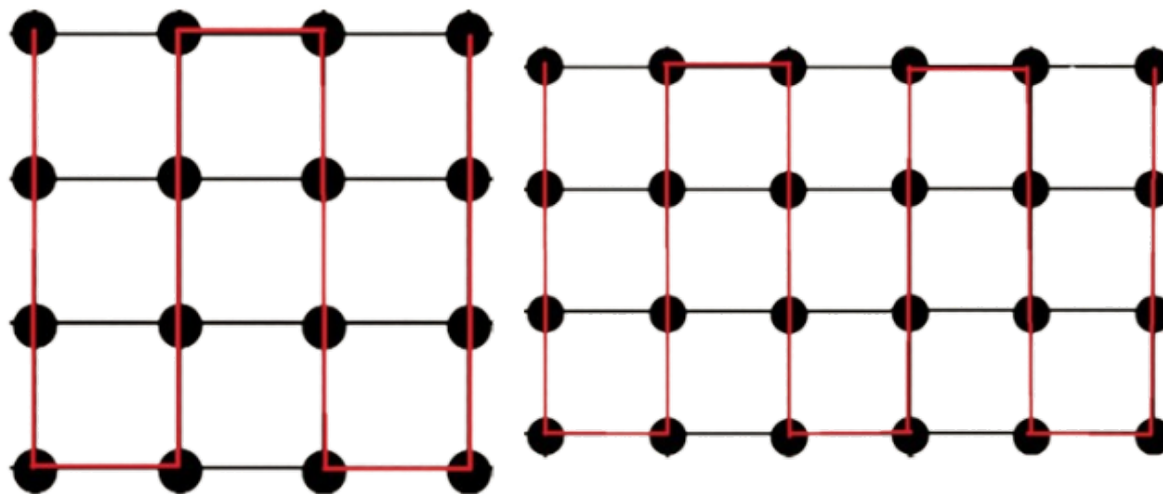


5 Obejścia grafu

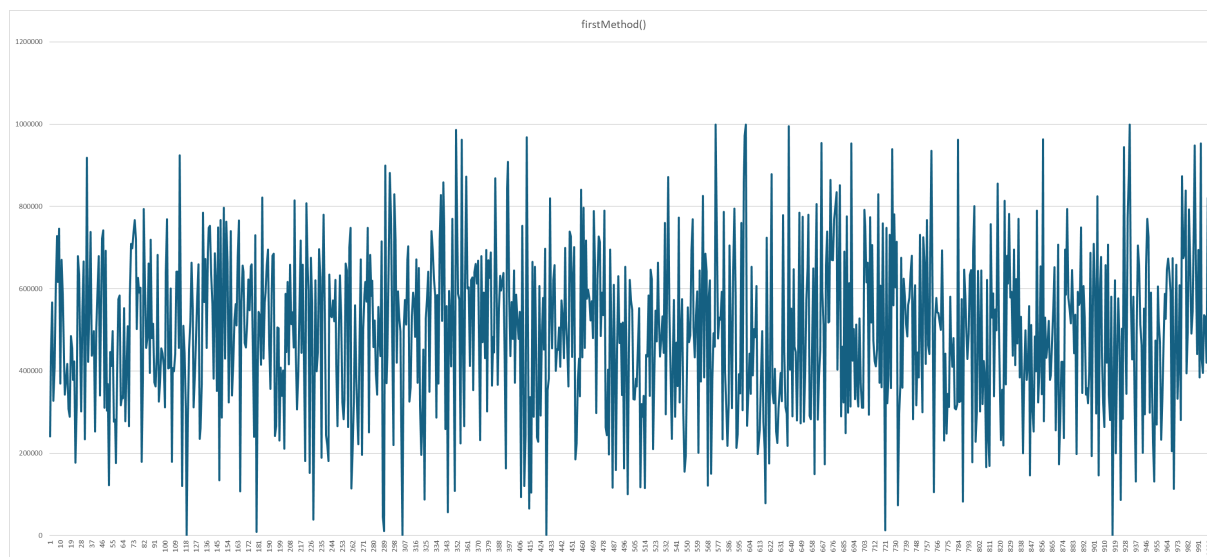
01. Wężymkiem
02. Spirala od krawędzi do środka
03. Spirala od środka do krawędzi
04. Random
05. BFS/DFS

5.1 Wężymkiem

Startujemy z punktu (0,0) i idziemy maksymalnie w dół, potem na jeden krok w prawo i maksymalnie do góry, zatem znów krok w prawo. Robimy taki cykl do końca przejścia naszego grafu.



Dla grafu $1000 * 1000$ mieliśmy 1000 iteracji. Możemy na poniższym wykresie zobaczyć wyniki:



Tak jak nasz skarb z większym prawdopodobieństwem będzie na środku, to średnia liczba potrzebnych kroków do znajdowania skarbu dążyć do 500 tys.. W naszym przykładzie ta liczba wynosi 494109.

Poniżej przedstawiamy fragment kodu tej metody:

```

1 public int firstMethod(){
2     int numberOfSteps = 0;
3     boolean down = true;
4     Node currentNode = startPoint;
5     if(currentNode.isFinalDestination()) return numberOfSteps;
6     while (currentNode != null){
7         while((down ? currentNode.getDown() : currentNode.getUp()) != null){
8             if(currentNode.isFinalDestination()) return numberOfSteps;
9             currentNode = (down ? currentNode.getDown() : currentNode.getUp());
10            numberOfSteps++;
11        }
12        if(currentNode.isFinalDestination()) return numberOfSteps;
13        currentNode = currentNode.getRight();
14        down = !down;

```

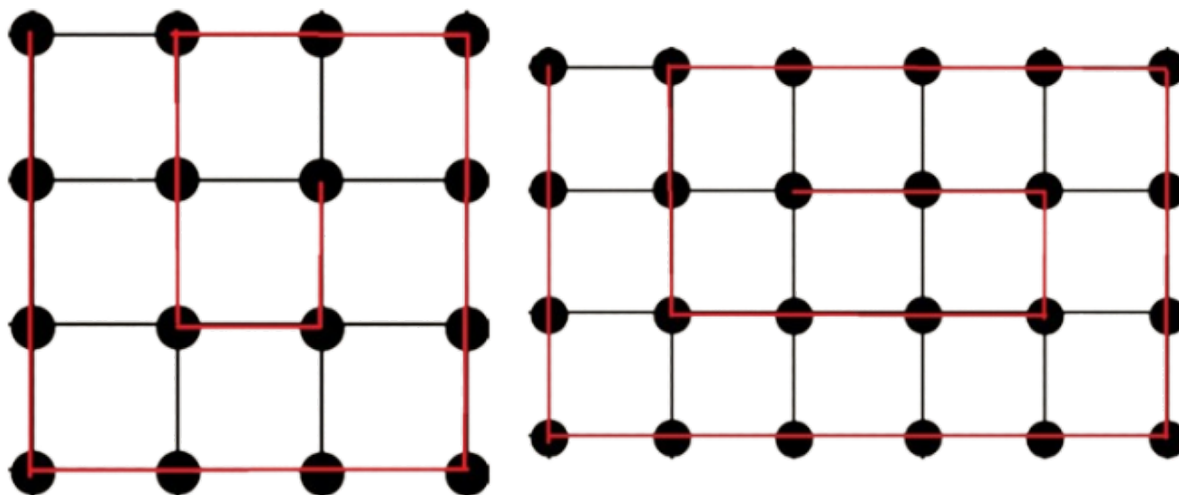
```

15         numberOfSteps++;
16     }
17     return numberOfSteps;
18 }

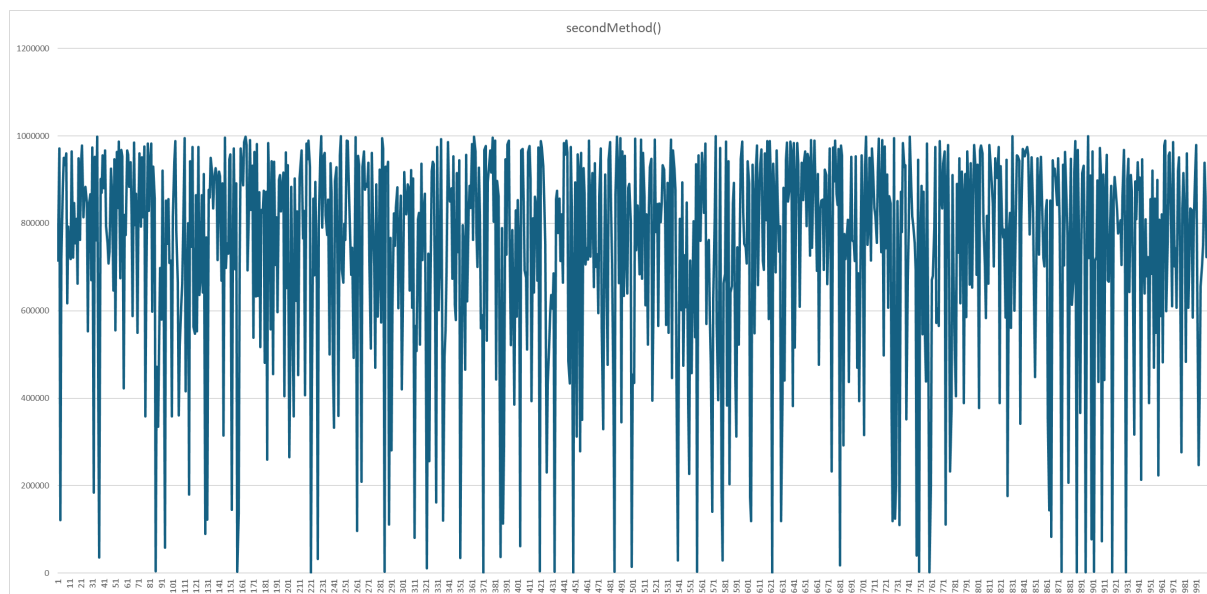
```

5.2 Spirala od krawędzi do środka

Startujemy z punktu (0,0) i idziemy maksymalnie w dół, potem maksymalnie w prawo i maksymalnie do góry, zatem maksymalnie w lewo(odwiedzamy wierzchołki, które jeszcze nie były odwiedzone). Robimy taki cykl do końca przejścia naszego grafu. Końcowy punkt będzie w samym środku.



Dla grafu 1000 * 1000 mieliśmy 1000 iteracji. Możemy na poniższym wykresie zobaczyć wyniki:



Tak jak nasz skarb z większym prawdopodobieństwem będzie na środku, to średnia liczba potrzebnych kroków do znajdowania skarbu będzie dążyć do 1 mln.. W naszym przykładzie ta liczba wynosi 733887. Poniżej przedstawiamy fragment kodu tej metody:

```

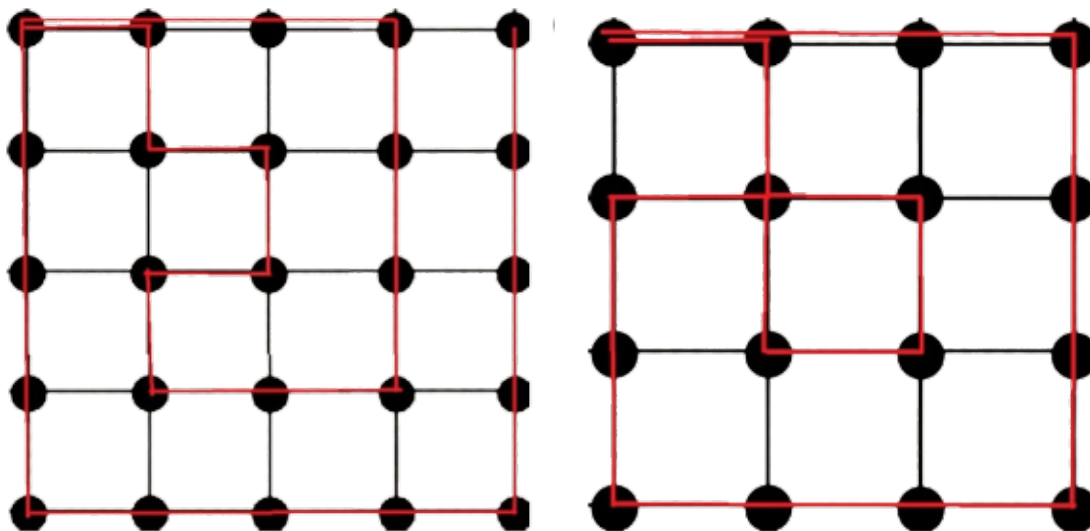
1 public int secondMethod(){
2     int numberOfSteps = 0;
3     Node currentNode = startPoint;
4     int direction = 0;
5     if(currentNode.isFinalDestination()) return numberOfSteps;
6     while(!currentNode.isFinalDestination()){
7         switch (direction % 4){
8             case 0 -> {
9                 while(currentNode.getDown() != null && !currentNode.getDown().getVisited()){
10                     currentNode.setVisited();
11                     currentNode = currentNode.getDown();
12                     numberOfSteps++;
13                     if(currentNode.isFinalDestination()) return numberOfSteps;
14                 }
15                 direction++;
16             }
17             case 1->{
18                 while(currentNode.getRight() != null && !currentNode.getRight().getVisited()){
19                     currentNode.setVisited();
20                     currentNode = currentNode.getRight();
21                     numberOfSteps++;
22                     if(currentNode.isFinalDestination()) return numberOfSteps;
23                 }
24                 direction++;
25             }
26             case 2 ->{
27                 while(currentNode.getUp() != null && !currentNode.getUp().getVisited()){
28                     currentNode.setVisited();
29                     currentNode = currentNode.getUp();
30                     numberOfSteps++;
31                     if(currentNode.isFinalDestination()) return numberOfSteps;
32                 }
33                 direction++;
34             }
35             case 3->{
36                 while(currentNode.getLeft() != null && !currentNode.getLeft().getVisited()){
37                     currentNode.setVisited();
38                     currentNode = currentNode.getLeft();
39                     numberOfSteps++;
40                     if(currentNode.isFinalDestination()) return numberOfSteps;
41                 }
42                 direction++;
43             }
44         }
45     }
46     return numberOfSteps;
47 }

```

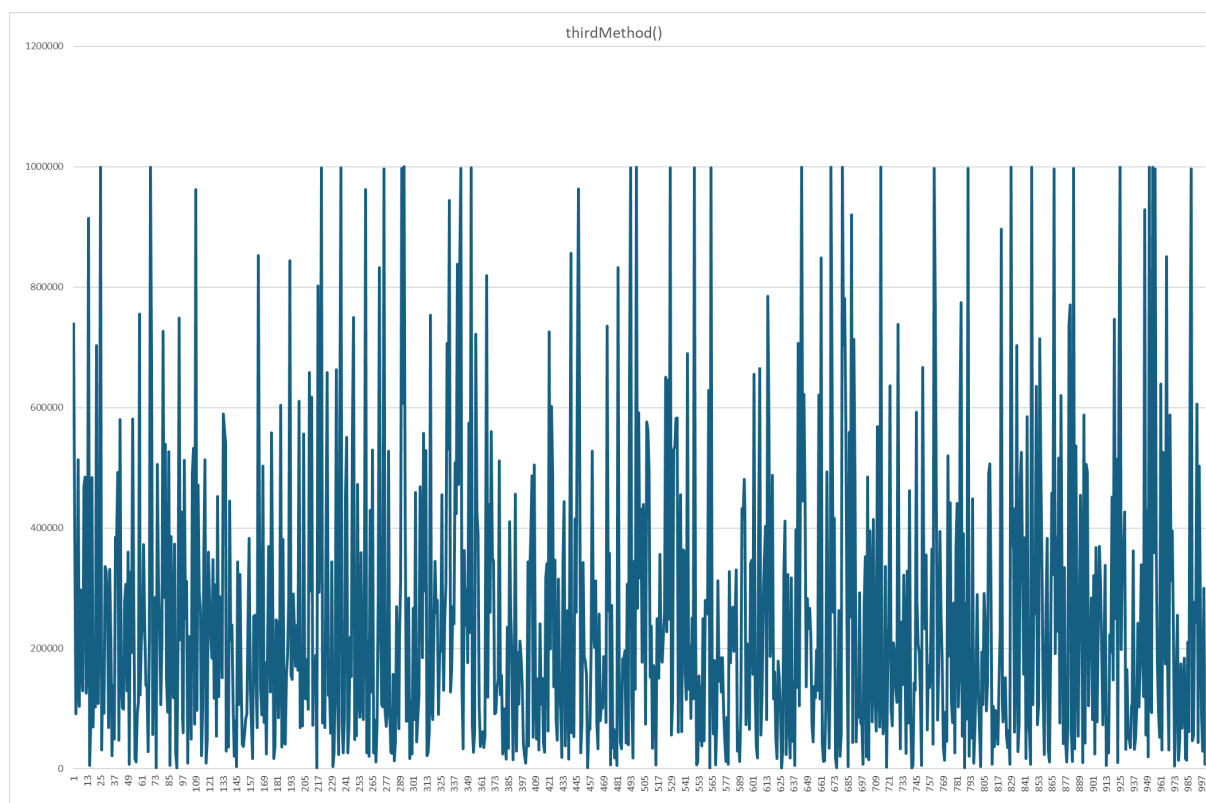
5.3 Spirala od środka do krawędzi

- 1) Tylko dla SUPER wyspy
- 2) Trochę bez sensu (bo nie wiemy gdzie jest środek), z założenia wyspa jest nieznana na samym początku

Startujemy z punktu (0,0) i idziemy w środek (jeden punkt dla nieparzystego n (długość boku), i 4 punkty w środku dla parzystego). Z tego środka idziemy spiralą do punktu startowego.



Dla grafu $1000 * 1000$ mieliśmy 1000 iteracji. Możemy na poniższym wykresie zobaczyć wyniki:



Tak jak nasz skarb z większym prawdopodobieństwem będzie na środku, to średnia liczba potrzebnych kroków do znajdowania skarbu będzie mniej niż 500 tys.. W naszym przykładzie ta liczba wynosi 257727. Poniżej przedstawiamy fragment kodu tej metody:

```

1
2 public int thirdMethod(){
3     int numberOfSteps = 0;
4     Node currentNode = startPoint;

```

```

5     Node centerNode;
6     boolean odd = false;
7     if( n % 2 == 0) centerNode = map.get(n / 2 - 1).get(n / 2 - 1);
8     else{
9         centerNode = map.get(n / 2).get(n / 2);
10        odd = true;
11    }
12    currentNode.setVisited();
13    if(currentNode.isFinalDestination()) return numberOfSteps;
14    while(currentNode != centerNode){
15        currentNode = currentNode.getRight();
16        currentNode.setVisited();
17        numberOfSteps++;
18        if(currentNode.isFinalDestination()) return numberOfSteps;
19        currentNode = currentNode.getDown();
20        currentNode.setVisited();
21        numberOfSteps++;
22        if(currentNode.isFinalDestination()) return numberOfSteps;
23    }
24    if(odd) {
25        for (int i = 1; i < n; i++) {
26            if (i == n - 1) {
27                for (int j = 1; j <= i; j++) {
28                    currentNode = currentNode.getRight();
29                    if(!currentNode.getVisited()){
30                        currentNode.setVisited();
31                        numberOfSteps++;
32                    }
33                    if(currentNode.isFinalDestination()) return numberOfSteps;
34                }
35                for (int j = 1; j <= i; j++) {
36                    currentNode = currentNode.getUp();
37                    if(!currentNode.getVisited()){
38                        currentNode.setVisited();
39                        numberOfSteps++;
40                    }
41                    if(currentNode.isFinalDestination()) return numberOfSteps;
42                }
43                for (int j = 1; j <= i; j++) {
44                    currentNode = currentNode.getLeft();
45                    if(!currentNode.getVisited()){
46                        currentNode.setVisited();
47                        numberOfSteps++;
48                    }
49                    if(currentNode.isFinalDestination()) return numberOfSteps;
50                }
51            } else if (i % 2 != 0) {
52                for (int j = 1; j <= i; j++) {
53                    currentNode = currentNode.getLeft();
54                    if(!currentNode.getVisited()){
55                        currentNode.setVisited();
56                        numberOfSteps++;
57                    }
58                    if(currentNode.isFinalDestination()) return numberOfSteps;
59                }
60                for (int j = 1; j <= i; j++) {
61                    currentNode = currentNode.getDown();
62                    if(!currentNode.getVisited()){
63                        currentNode.setVisited();
64                        numberOfSteps++;
65                    }
66                    if(currentNode.isFinalDestination()) return numberOfSteps;
67                }
68            } else {
69                for (int j = 1; j <= i; j++) {
70                    currentNode = currentNode.getRight();
71                    if(!currentNode.getVisited()){

```

```

72         currentNode.setVisited();
73         numberOfSteps++;
74     }
75     if(currentNode.isFinalDestination()) return numberOfSteps;
76 }
77 for (int j = 1; j <= i; j++) {
78     currentNode = currentNode.getUp();
79     if(!currentNode.getVisited()){
80         currentNode.setVisited();
81         numberOfSteps++;
82     }
83     if(currentNode.isFinalDestination()) return numberOfSteps;
84 }
85 }
86 }
87 }
88 else{
89     for (int i = 1; i < n; i++) {
90         if (i == n - 1) {
91             for (int j = 1; j <= i; j++) {
92                 currentNode = currentNode.getRight();
93                 if(!currentNode.getVisited()){
94                     currentNode.setVisited();
95                     numberOfSteps++;
96                 }
97                 if(currentNode.isFinalDestination()) return numberOfSteps;
98             }
99             for (int j = 1; j <= i; j++) {
100                 currentNode = currentNode.getUp();
101                 if(!currentNode.getVisited()){
102                     currentNode.setVisited();
103                     numberOfSteps++;
104                 }
105                 if(currentNode.isFinalDestination()) return numberOfSteps;
106             }
107             for (int j = 1; j <= i; j++) {
108                 currentNode = currentNode.getLeft();
109                 if(!currentNode.getVisited()){
110                     currentNode.setVisited();
111                     numberOfSteps++;
112                 }
113                 if(currentNode.isFinalDestination()) return numberOfSteps;
114             }
115         }
116         else if(i == 1){
117             currentNode = currentNode.getRight();
118             if(!currentNode.getVisited()){
119                 currentNode.setVisited();
120                 numberOfSteps++;
121             }
122             if(currentNode.isFinalDestination()) return numberOfSteps;
123             currentNode = currentNode.getDown();
124             if(!currentNode.getVisited()){
125                 currentNode.setVisited();
126                 numberOfSteps++;
127             }
128             if(currentNode.isFinalDestination()) return numberOfSteps;
129             currentNode = currentNode.getLeft();
130             if(!currentNode.getVisited()){
131                 currentNode.setVisited();
132                 numberOfSteps++;
133             }
134             if(currentNode.isFinalDestination()) return numberOfSteps;
135             currentNode = currentNode.getUp();
136             if(!currentNode.getVisited()){
137                 currentNode.setVisited();
138                 numberOfSteps++;

```

```

139         }
140         if(currentNode.isFinalDestination()) return numberOfSteps;
141         currentNode = currentNode.getLeft();
142         if(!currentNode.getVisited()){
143             currentNode.setVisited();
144             numberOfSteps++;
145         }
146         if(currentNode.isFinalDestination()) return numberOfSteps;
147     }
148     else if(i == 2){
149         for (int j = 1; j <= i; j++) {
150             currentNode = currentNode.getDown();
151             if(!currentNode.getVisited()){
152                 currentNode.setVisited();
153                 numberOfSteps++;
154             }
155             if(currentNode.isFinalDestination()) return numberOfSteps;
156         }
157     }
158     else if (i % 2 == 0) {
159         for (int j = 1; j <= i; j++) {
160             currentNode = currentNode.getLeft();
161             if(!currentNode.getVisited()){
162                 currentNode.setVisited();
163                 numberOfSteps++;
164             }
165             if(currentNode.isFinalDestination()) return numberOfSteps;
166         }
167         for (int j = 1; j <= i; j++) {
168             currentNode = currentNode.getDown();
169             if(!currentNode.getVisited()){
170                 currentNode.setVisited();
171                 numberOfSteps++;
172             }
173             if(currentNode.isFinalDestination()) return numberOfSteps;
174         }
175     } else {
176         for (int j = 1; j <= i; j++) {
177             currentNode = currentNode.getRight();
178             if(!currentNode.getVisited()){
179                 currentNode.setVisited();
180                 numberOfSteps++;
181             }
182             if(currentNode.isFinalDestination()) return numberOfSteps;
183         }
184         for (int j = 1; j <= i; j++) {
185             currentNode = currentNode.getUp();
186             if(!currentNode.getVisited()){
187                 currentNode.setVisited();
188                 numberOfSteps++;
189             }
190             if(currentNode.isFinalDestination()) return numberOfSteps;
191         }
192     }
193 }
194 }
195 return numberOfSteps;
196 }

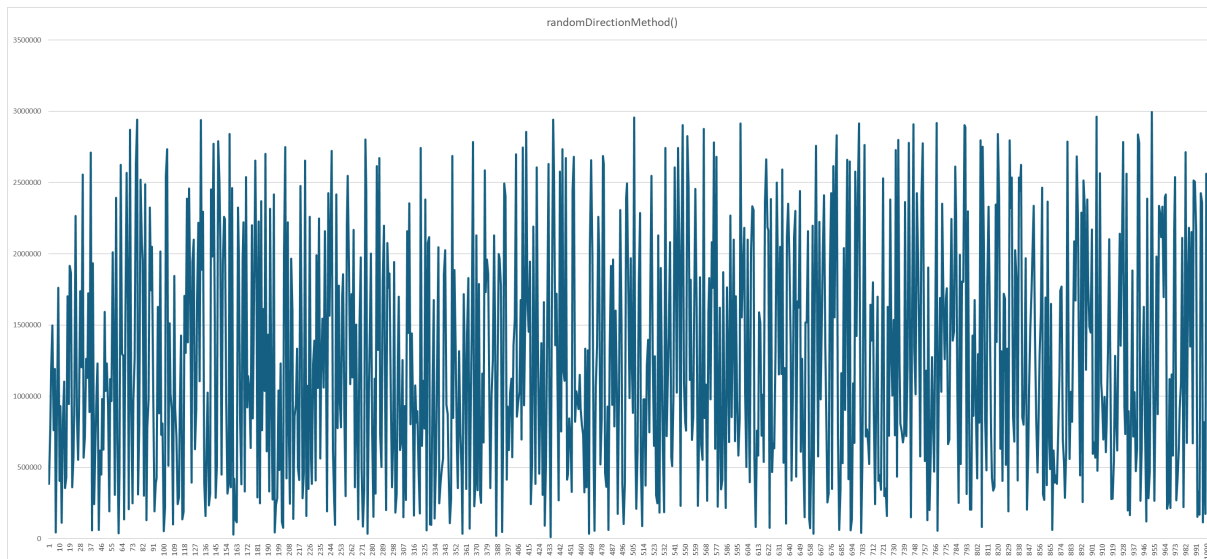
```

5.4 Random

- 1) Wybiera losowo
- 2) Wybiera nieodwiedzone wierzchołki

Startujemy z punktu (0,0) i wybieramy losowo w jaką stronę pójść. Wybieramy zawsze nieodwiedzone wierzchołki. W przypadku gdy jesteśmy w punkcie otoczonym odwiedzonymi wierzchołkami to idziemy drogą, którą przeszliśmy dopóki nie znajdziemy wierzchołek z nieodwiedzonym sąsiadem.

Dla grafu $1000 * 1000$ mieliśmy 1000 iteracji. Możemy na poniższym wykresie zobaczyć wyniki:



Tak jak nasza metoda jest losowa, ciężko przewidzieć ile wynosi średnia liczba potrzebnych kroków do znajdowania skarbu. W naszym przykładzie ta liczba wynosi 1255954.

Poniżej przedstawiamy fragment kodu tej metody:

```

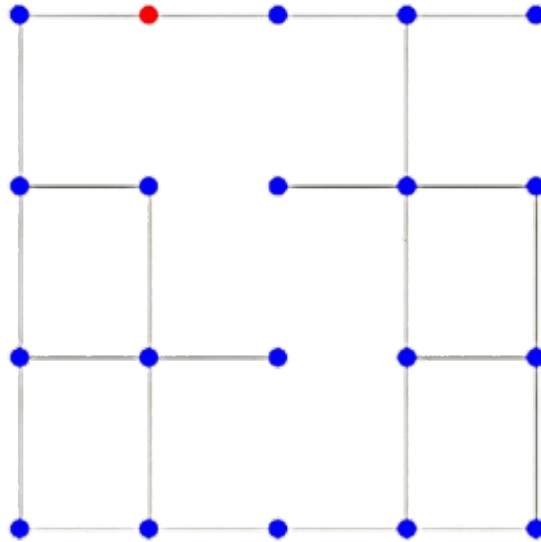
1 public int randomDirectionMethod(){
2     int numberOfSteps = 0;
3     Node currentNode = startPoint;
4     Stack<Node> stack = new Stack<>();
5     if(currentNode.isFinalDestination()) return numberOfSteps;
6     stack.push(currentNode);
7     while (!stack.isEmpty() && !currentNode.isFinalDestination()){
8         currentNode.setVisited();
9         currentNode = stack.peek().getUnvisitedNode();
10        numberOfSteps++;
11        if(currentNode == null){
12            currentNode = stack.pop();
13            numberOfSteps++;
14        }else{
15            stack.push(currentNode);
16        }
17    }
18    return numberOfSteps;
19 }

```

5.5 BFS/DFS

1) BFS nie będzie działać na naszym grafie
(bo nie możemy przeskakiwać po przekątnej)

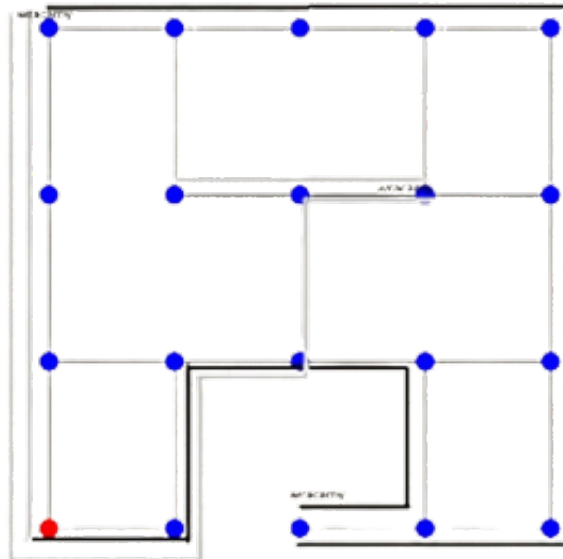
Używając BFS będziemy mieć taką drogą: $(0,0) \rightarrow (0,1)$ i $(1,0)$



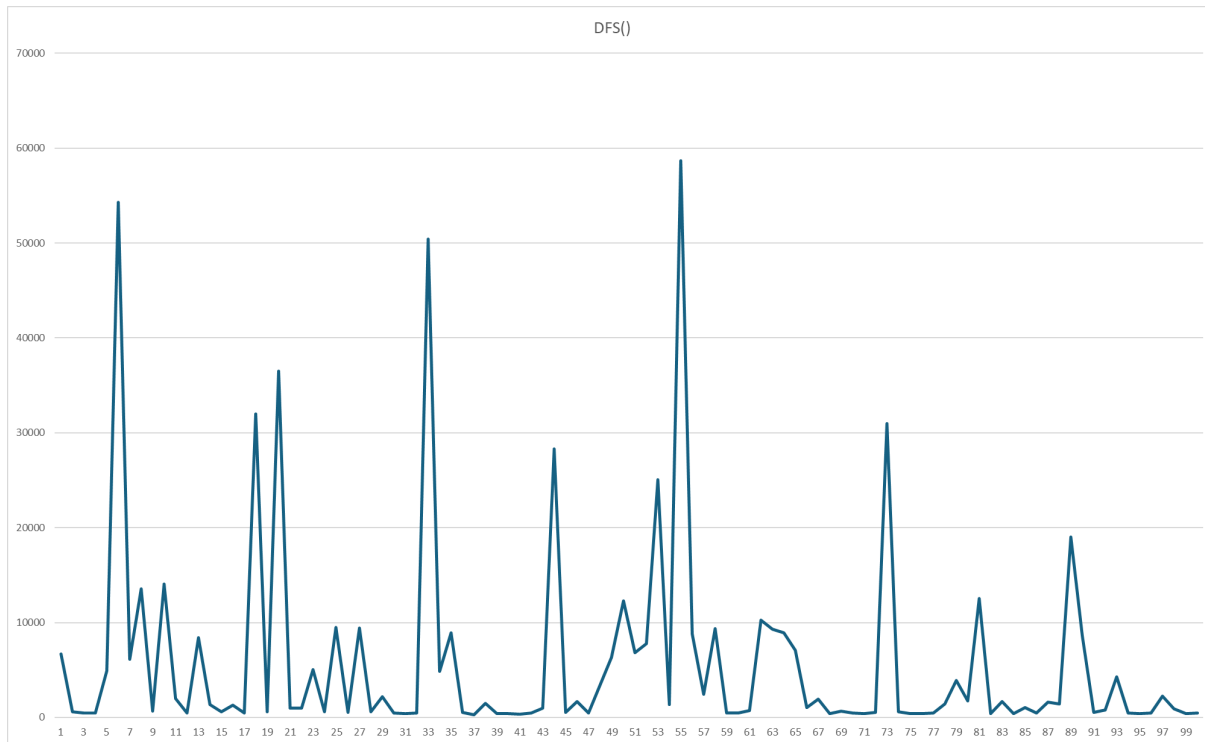
2) DFS dla prawdopodobnej wyspy

Pierwszeństwo: prawo, dół, lewo, góra

Startujemy z punktu (0,0) i idziemy, dopóki możemy w prawo, zatem w dół, w lewo i do góry. Jeżeli mamy sytuację bez wyjścia, czyli nie odwiedziliśmy wszystkich wierzchołków, a sąsiedzi wszystkie odwiedzone, to wracamy do poprzedniego wierzchołka i szukamy naszego nieodwiedzonego sąsiada (oczywiście pamiętając o pierwszeństwie).



Dla grafu 200 * 200 mieliśmy 100 iteracji. Możemy na poniższym wykresie zobaczyć wyniki:



Tak jak za każdym razem mamy różne wyspy. Ciężko przewidzieć jaka jest średnia liczba potrzebnych kroków. W naszym przypadku ta liczba wynosi 6000.

Poniżej przedstawiamy fragment kodu tej metody:

```

1      public int depthFirstSearch() {
2          int numberOfSteps = 0;
3          if(startPoint.isFinalDestination()){
4              return numberOfSteps;
5          }
6          Set<Node> visited = new HashSet<>();
7          Stack<Node> path = new Stack<>();
8          List<Node> result = new ArrayList<>();
9          path.push(startPoint);
10
11         while (!path.isEmpty()) {
12             Node current = path.peek();
13             if(current.isFinalDestination()){
14                 result.add(current);
15                 numberOfSteps++;
16                 return numberOfSteps;
17             }
18             result.add(current);
19             numberOfSteps++;
20
21             if (!visited.contains(current)) {
22                 visited.add(current);
23             }
24
25             Node next = getUnvisitedNeighbor(current, visited);
26             if (next != null) {
27                 path.push(next);
28             } else {
29                 path.pop();
30             }
31         }

```

```

32         return numberOfSteps;
33     }
34
35     private Node getUnvisitedNeighbor(Node node, Set<Node> visited) {
36         if (node.getRight() != null && !visited.contains(node.getRight()))
37             return node.getRight();
38         if (node.getDown() != null && !visited.contains(node.getDown()))
39             return node.getDown();
40         if (node.getLeft() != null && !visited.contains(node.getLeft()))
41             return node.getLeft();
42         if (node.getUp() != null && !visited.contains(node.getUp()))
43             return node.getUp();
44         return null;
45     }

```

6 Wynik

W wyniku analizy wszystkich naszych strategii i biorąc pod uwagę nasze założenia, można stwierdzić, że najbardziej nieoptymalną metodą jest metoda losowa. W tej metodzie istnieje większe prawdopodobieństwo wykonania wielu zbędnych kroków i zmarnowania sił naszego bohatera. Dlatego, biorąc pod uwagę wyspę z przeszkodami, nawet nie braliśmy pod uwagę tej metody.

Najbardziej optymalną strategią będzie spirala od środka do krawędzi. Niestety, w założeniach rozważamy nieznaną wyspę i nie będziemy wiedzieć, gdzie znajduje się jej centrum. Dlatego wybieramy coś pośredniego między tymi strategiami. Wychodzi strategia DFS (przeszukiwanie w głąb). Dla rzeczywistej wyspy jest to najbardziej optymalny i najszybszy sposób dotarcia do celu w najkrótszym czasie. Jeśli wyspa będzie idealna, to strategia DFS będzie działać jak spirala od krawędzi do środka. W takim przypadku, lepiej jest zastosować strategię wężyka.