Jagiellonian University

Projekt: Ukryty skarb
Raport

Maryia Babinskaya
Illia Dovhalenko

June, 2024

Spis treści

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;
```

```
        this.node2 = node2;
        this.isVertical = node1.getX() == node2.getX();
    }

    public void remove() {
        if (isVertical) {
            node1.setDown(null);
            node2.setUp(null);
        } else {
            node1.setRight(null);
            node2.setLeft(null);
        }
    }

    public void restore() {
        if (isVertical) {
            node1.setDown(node2);
            node2.setUp(node1);
        } else {
            node1.setRight(node2);
            node2.setLeft(node1);
        }
    }
}
```

3.3 Implementacja Graph w języku Java

```
import java.util.ArrayList;
import java.util.Random;
import java.util.Stack;

public class Graph {
    int n, m;
    Node startPoint;
    ArrayList<ArrayList<Node>> map;

    public Graph(int n_, int m_){
        n = n_;
        m = m_;
        map = new ArrayList<>(n);

        for(int y = 0; y < m; y++){
            ArrayList<Node> oneRow = new ArrayList<>(n);
            for (int x = 0; x < n; x++) {
                Node newNode = new Node(x, y);
                if(x != 0) {
                    newNode.setLeft(oneRow.get(x - 1));
                    oneRow.get(x - 1).setRight(newNode);
                }
                if(y != 0){
                    newNode.setUp(map.get(y - 1).get(x));
                    map.get(y - 1).get(x).setDown(newNode);
                }
                oneRow.add(newNode);
            }
            map.add(oneRow);
        }

        startPoint = map.get(0).get(0);
        Random random = new Random();
        int randX = (int) (random.nextGaussian()*200+500);
        int randY =  (int) (random.nextGaussian()*200+500);
        if(randX>=m) randX=m-1;
        if(randX<0) randX=0;
        if(randY>=n) randY=n-1;
```

```
        if(randY<0) randY=0;
        map.get(randY).get(randX).setFinalDestination();
    }

    public void print(){
        for (ArrayList<Node> row : map) {
            for (Node node : row) {
                System.out.print(node + " ");
            }
            System.out.println();
        }
    }
}
```

3.4 Implementacja GraphIsland w języku Java

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Random;
import java.util.Set;
import java.util.Stack;
import java.util.LinkedList;
import java.util.Queue;


public class GraphIsland {
    int rows, cols;
    Node startPoint;
    ArrayList<ArrayList<Node>> grid;
    Random random = new Random();

    public GraphIsland(int rows, int cols) {
        this.rows = rows;
        this.cols = cols;
        grid = new ArrayList<>(rows);

        for (int y = 0; y < rows; y++) {
            ArrayList<Node> row = new ArrayList<>(cols);

            for (int x = 0; x < cols; x++) {
                Node newNode = new Node(x, y);

                if (x > 0) {
                    newNode.setLeft(row.get(x - 1));
                    row.get(x - 1).setRight(newNode);
                }

                if (y > 0) {
                    newNode.setUp(grid.get(y - 1).get(x));
                    grid.get(y - 1).get(x).setDown(newNode);
                }

                row.add(newNode);
            }

            grid.add(row);
        }

        startPoint = grid.get(0).get(0);
        setRandomFinalDestination();

        int max = ((rows - 1) * cols + (cols - 1) * rows) / 2;
        int numberBrokenEdges = random.nextInt(++max);
        removeEdges(numberBrokenEdges);
```

```
    }

    private void setRandomFinalDestination() {
        Random random = new Random();
        int randX = (int) (random.nextGaussian()*200+500);
        int randY =  (int) (random.nextGaussian()*200+500);
        if(randX>=cols) {
            randX=cols-1;
        }
        if(randX<0) {
            randX=0;
        }
        if(randY>=rows) {
            randY=rows-1;
        }
        if(randY<0){
            randY=0;
        }
        grid.get(randY).get(randX).setFinalDestination();
    }

    private void removeEdges(int numberBrokenEdges) {
        ArrayList<Edge> edges = new ArrayList<>();

        // Collect all edges
        for (int y = 0; y < rows; y++) {
            for (int x = 0; x < cols; x++) {
                Node node = grid.get(y).get(x);
                if (node.getRight() != null) edges.add(new Edge(node, node.getRight()));
                if (node.getDown() != null) edges.add(new Edge(node, node.getDown()));
            }
        }

        int removedEdges = 0;
        while (removedEdges < numberBrokenEdges && !edges.isEmpty()) {
            int index = random.nextInt(edges.size());
            Edge edge = edges.get(index);
            edges.remove(index);

            // Temporarily remove edge
            edge.remove();

            // Check if graph is still connected
            if (isConnected()) {
                removedEdges++;
            } else {
                // Restore edge if graph is not connected
                edge.restore();
            }
        }
    }



    private boolean isConnected() {
        Set<Node> visited = new HashSet<>();
        Stack<Node> stack = new Stack<>();
        stack.push(startPoint);

        while (!stack.isEmpty()) {
            Node node = stack.pop();
            if (!visited.contains(node)) {
                visited.add(node);
                if (node.getRight() != null && !visited.contains(node.getRight()))
                    stack.push(node.getRight());
                if (node.getDown() != null && !visited.contains(node.getDown()))
                    stack.push(node.getDown());
```

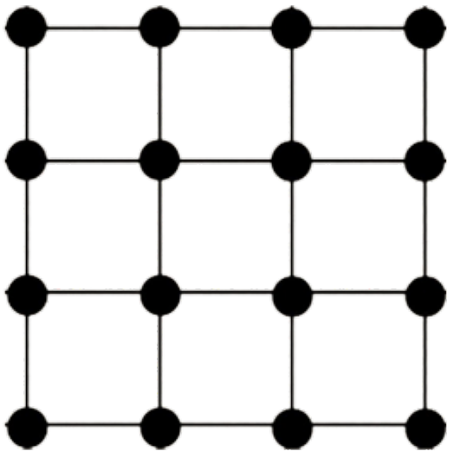
```
117                     if (node.getLeft() != null && !visited.contains(node.getLeft()))
118                         stack.push(node.getLeft());
119                     if (node.getUp() != null && !visited.contains(node.getUp()))
120                         stack.push(node.getUp());
121                 }
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

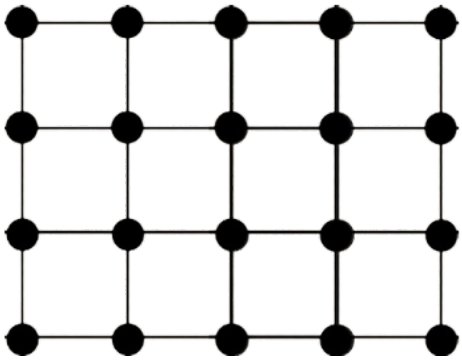
4.1 SUPER idealna wyspa

1) Kwadratowa
2) Mamy wszystkie krawędzi



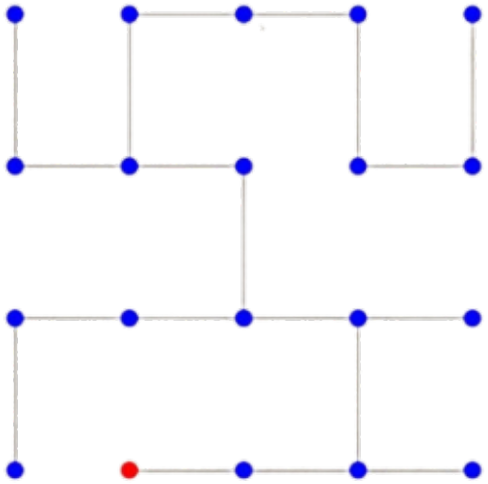
4.2 Idealna wyspa

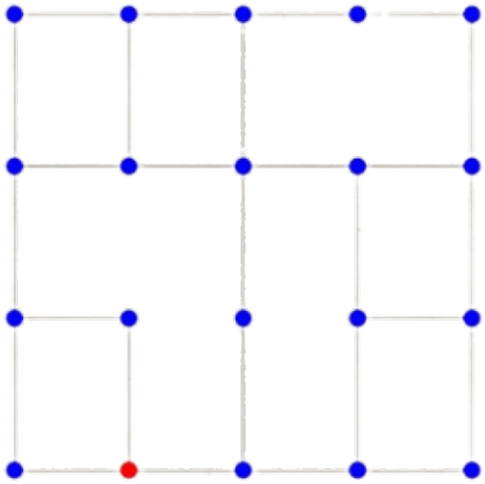
1) Prostokąt
2) Mamy wszystkie krawędzi



4.3 REAL wyspa

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



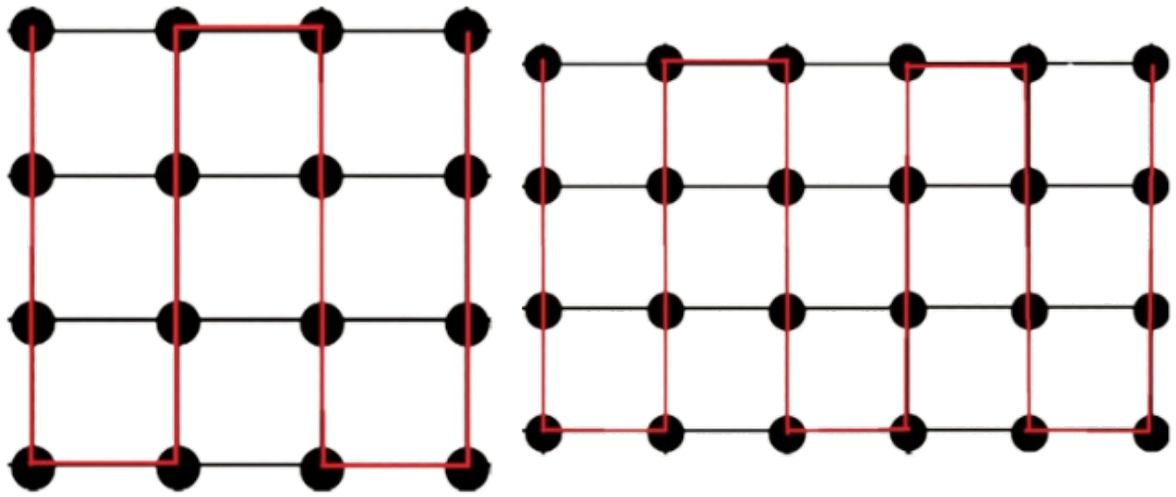


5 Obejścia grafu

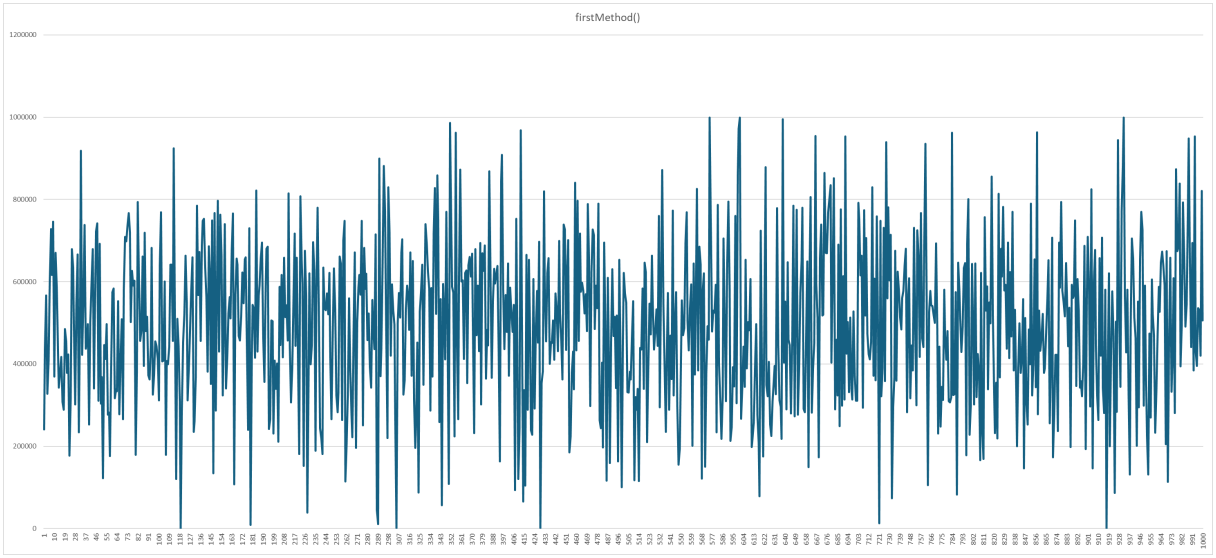
01. Wężykiem
02. Spiralą od krawędzi do środka
03. Spiralą od środka do krawędzi
04. Random
05. BFS/DFS

5.1 Wężykiem

Startujemy z punku (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.

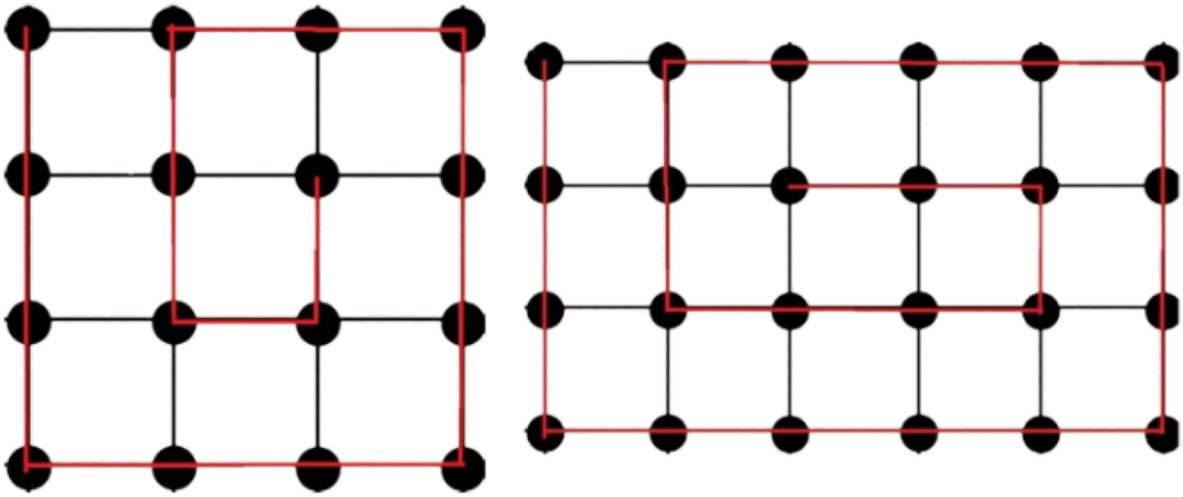
Poniżej przedstawiamy fragment kodu tej metody:

```
    public int firstMethod(){
        int numberOfSteps = 0;
        boolean down = true;
        Node currentNode = startPoint;
        if(currentNode.isFinalDestination()) return numberOfSteps;
        while (currentNode != null){
            while((down ? currentNode.getDown() : currentNode.getUp()) != null){
                if(currentNode.isFinalDestination()) return numberOfSteps;
                currentNode = (down ? currentNode.getDown() : currentNode.getUp());
                numberOfSteps++;
            }
            if(currentNode.isFinalDestination()) return numberOfSteps;
            currentNode = currentNode.getRight();
            down = !down;
```

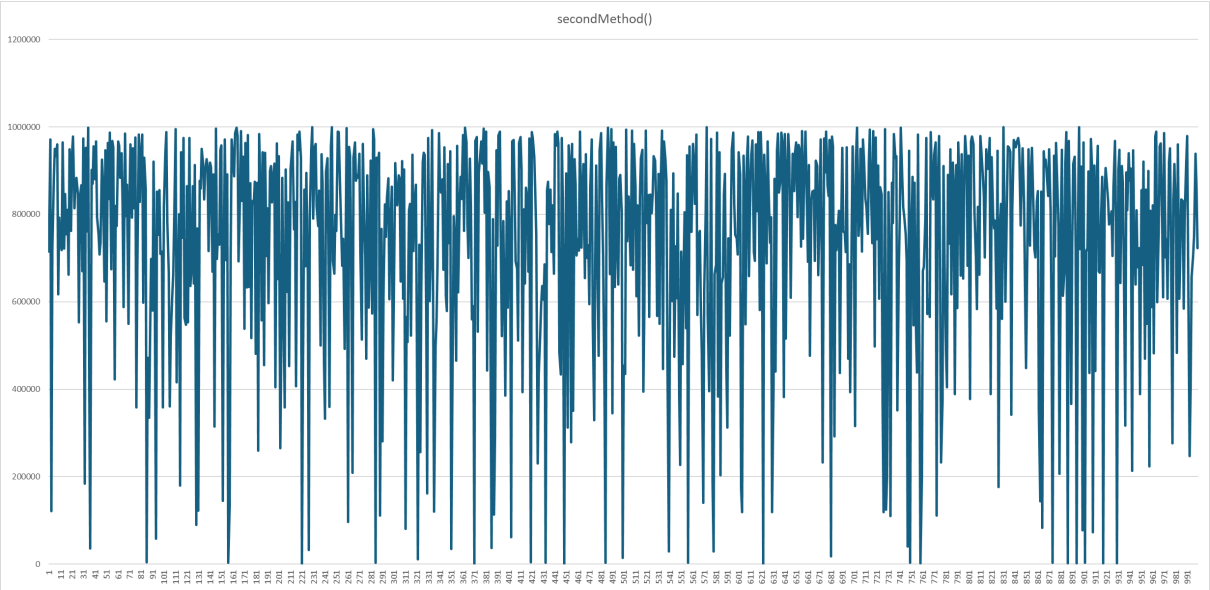
```
15              numberOfSteps++;
16          }
17          return numberOfSteps;
18      }
```

5.2 Spiralą od krawędzi do środka

Startujemy z punku (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:



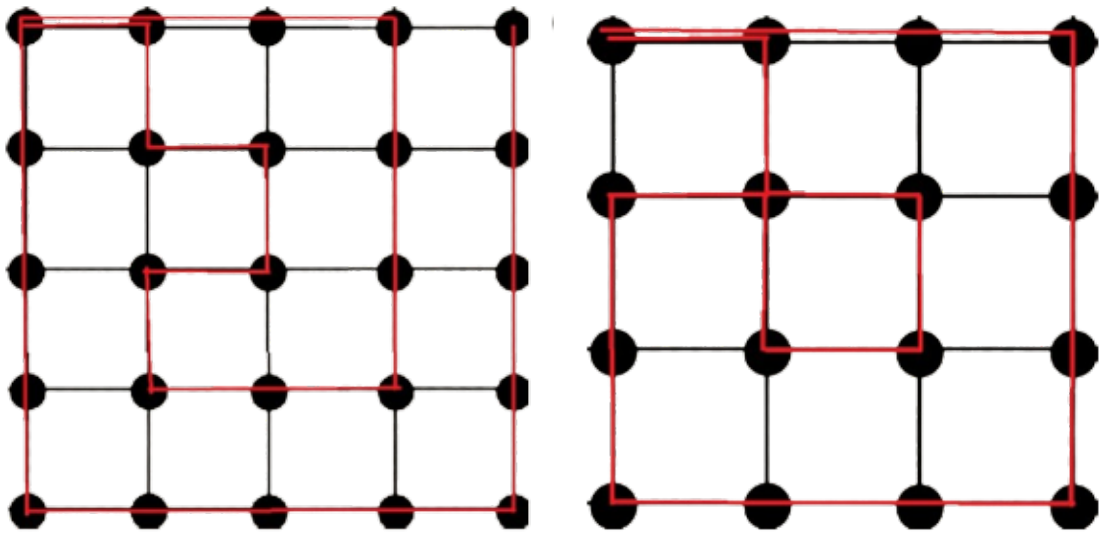
Tak jak nasz skarb z większym prawdopodobieństwem będzie na środku, to średnia liczba potrzebnych kroków do znajdowania skarbu będzie dążyć do 1 mln.. W naszym przykładzie ta liczba wynosi 733887. Poniżej przedstawiamy fragment kodu tej metody:

```
    public int secondMethod(){
        int numberOfSteps = 0;
        Node currentNode = startPoint;
        int direction = 0;
        if(currentNode.isFinalDestination()) return numberOfSteps;
        while(!currentNode.isFinalDestination()){
            switch (direction % 4){
                case 0 -> {
                    while(currentNode.getDown() != null && !currentNode.getDown().getVisited()){
                        currentNode.setVisited();
                        currentNode = currentNode.getDown();
                        numberOfSteps++;
                        if(currentNode.isFinalDestination()) return numberOfSteps;
                    }
                    direction++;
                }
                case 1->{
                    while(currentNode.getRight() != null && !currentNode.getRight().getVisited()){
                        currentNode.setVisited();
                        currentNode = currentNode.getRight();
                        numberOfSteps++;
                        if(currentNode.isFinalDestination()) return numberOfSteps;
                    }
                    direction++;
                }
                case 2 ->{
                    while(currentNode.getUp() != null && !currentNode.getUp().getVisited()){
                        currentNode.setVisited();
                        currentNode = currentNode.getUp();
                        numberOfSteps++;
                        if(currentNode.isFinalDestination()) return numberOfSteps;
                    }
                    direction++;
                }
                case 3->{
                    while(currentNode.getLeft() != null && !currentNode.getLeft().getVisited()){
                        currentNode.setVisited();
                        currentNode = currentNode.getLeft();
                        numberOfSteps++;
                        if(currentNode.isFinalDestination()) return numberOfSteps;
                    }
                    direction++;
                }
            }
        }
        return numberOfSteps;
    }
```

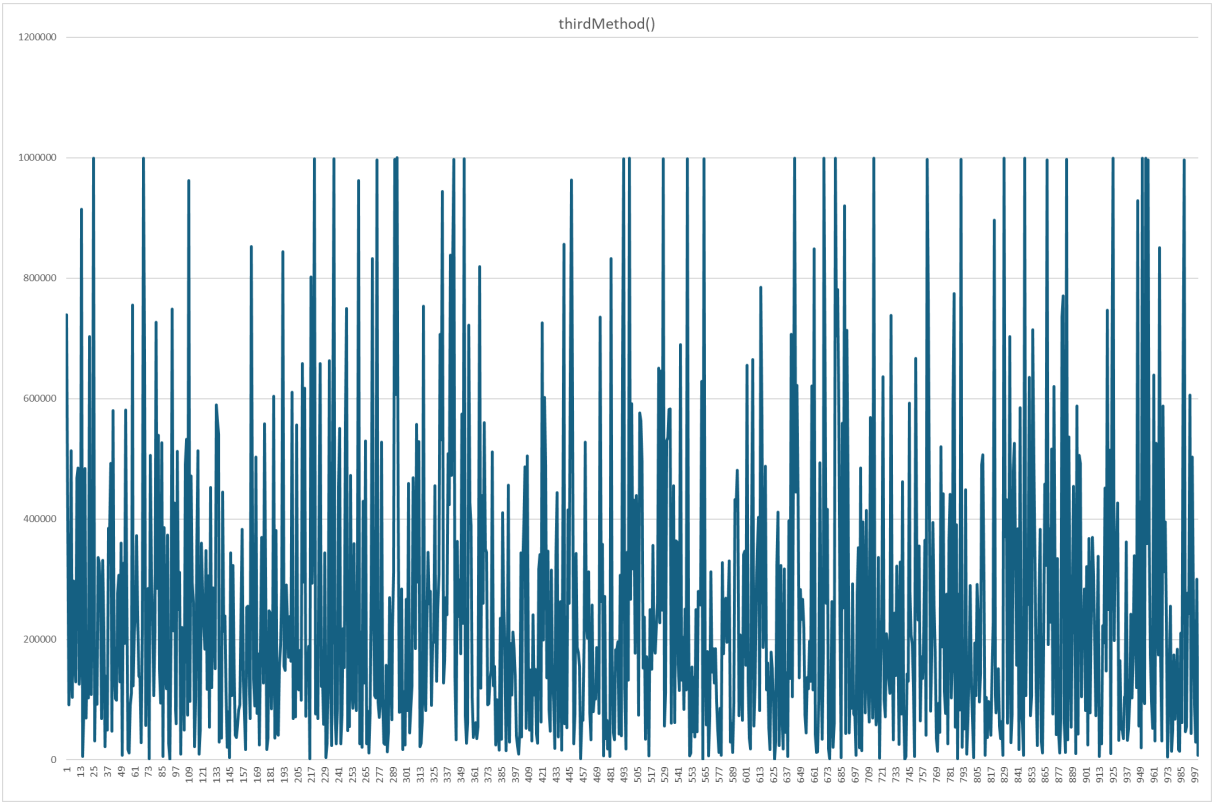
5.3 Spiralą od środka do krawędzi

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

Startujemy z punku (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;
```

```
        Node centerNode;
        boolean odd = false;
        if( n % 2 == 0) centerNode = map.get(n / 2 - 1).get(n / 2 - 1);
        else{
            centerNode = map.get(n / 2).get(n / 2);
            odd = true;
        }
        currentNode.setVisited();
        if(currentNode.isFinalDestination()) return numberOfSteps;
        while(currentNode != centerNode){
            currentNode = currentNode.getRight();
            currentNode.setVisited();
            numberOfSteps++;
            if(currentNode.isFinalDestination()) return numberOfSteps;
            currentNode = currentNode.getDown();
            currentNode.setVisited();
            numberOfSteps++;
            if(currentNode.isFinalDestination()) return numberOfSteps;
        }
        if(odd) {
            for (int i = 1; i < n; i++) {
                if (i == n - 1) {
                    for (int j = 1; j <= i; j++) {
                        currentNode = currentNode.getRight();
                        if(!currentNode.getVisited()){
                            currentNode.setVisited();
                            numberOfSteps++;
                        }
                        if(currentNode.isFinalDestination()) return numberOfSteps;
                    }
                    for (int j = 1; j <= i; j++) {
                        currentNode = currentNode.getUp();
                        if(!currentNode.getVisited()){
                            currentNode.setVisited();
                            numberOfSteps++;
                        }
                        if(currentNode.isFinalDestination()) return numberOfSteps;
                    }
                    for (int j = 1; j <= i; j++) {
                        currentNode = currentNode.getLeft();
                        if(!currentNode.getVisited()){
                            currentNode.setVisited();
                            numberOfSteps++;
                        }
                        if(currentNode.isFinalDestination()) return numberOfSteps;
                    }
                } else if (i % 2 != 0) {
                    for (int j = 1; j <= i; j++) {
                        currentNode = currentNode.getLeft();
                        if(!currentNode.getVisited()){
                            currentNode.setVisited();
                            numberOfSteps++;
                        }
                        if(currentNode.isFinalDestination()) return numberOfSteps;
                    }
                    for (int j = 1; j <= i; j++) {
                        currentNode = currentNode.getDown();
                        if(!currentNode.getVisited()){
                            currentNode.setVisited();
                            numberOfSteps++;
                        }
                        if(currentNode.isFinalDestination()) return numberOfSteps;
                    }
                } else {
                    for (int j = 1; j <= i; j++) {
                        currentNode = currentNode.getRight();
                        if(!currentNode.getVisited()){
```

```
                        currentNode.setVisited();
                        numberOfSteps++;
                    }
                    if(currentNode.isFinalDestination()) return numberOfSteps;
                }
                for (int j = 1; j <= i; j++) {
                    currentNode = currentNode.getUp();
                    if(!currentNode.getVisited()){
                        currentNode.setVisited();
                        numberOfSteps++;
                    }
                    if(currentNode.isFinalDestination()) return numberOfSteps;
                }
            }
        }
    }
    else{
        for (int i = 1; i < n; i++) {
            if (i == n - 1) {
                for (int j = 1; j <= i; j++) {
                    currentNode = currentNode.getRight();
                    if(!currentNode.getVisited()){
                        currentNode.setVisited();
                        numberOfSteps++;
                    }
                    if(currentNode.isFinalDestination()) return numberOfSteps;
                }
                for (int j = 1; j <= i; j++) {
                    currentNode = currentNode.getUp();
                    if(!currentNode.getVisited()){
                        currentNode.setVisited();
                        numberOfSteps++;
                    }
                    if(currentNode.isFinalDestination()) return numberOfSteps;
                }
                for (int j = 1; j <= i; j++) {
                    currentNode = currentNode.getLeft();
                    if(!currentNode.getVisited()){
                        currentNode.setVisited();
                        numberOfSteps++;
                    }
                    if(currentNode.isFinalDestination()) return numberOfSteps;
                }
            }
            else if(i == 1){
                currentNode = currentNode.getRight();
                if(!currentNode.getVisited()){
                    currentNode.setVisited();
                    numberOfSteps++;
                }
                if(currentNode.isFinalDestination()) return numberOfSteps;
                currentNode = currentNode.getDown();
                if(!currentNode.getVisited()){
                    currentNode.setVisited();
                    numberOfSteps++;
                }
                if(currentNode.isFinalDestination()) return numberOfSteps;
                currentNode = currentNode.getLeft();
                if(!currentNode.getVisited()){
                    currentNode.setVisited();
                    numberOfSteps++;
                }
                if(currentNode.isFinalDestination()) return numberOfSteps;
                currentNode = currentNode.getUp();
                if(!currentNode.getVisited()){
                    currentNode.setVisited();
                    numberOfSteps++;
```

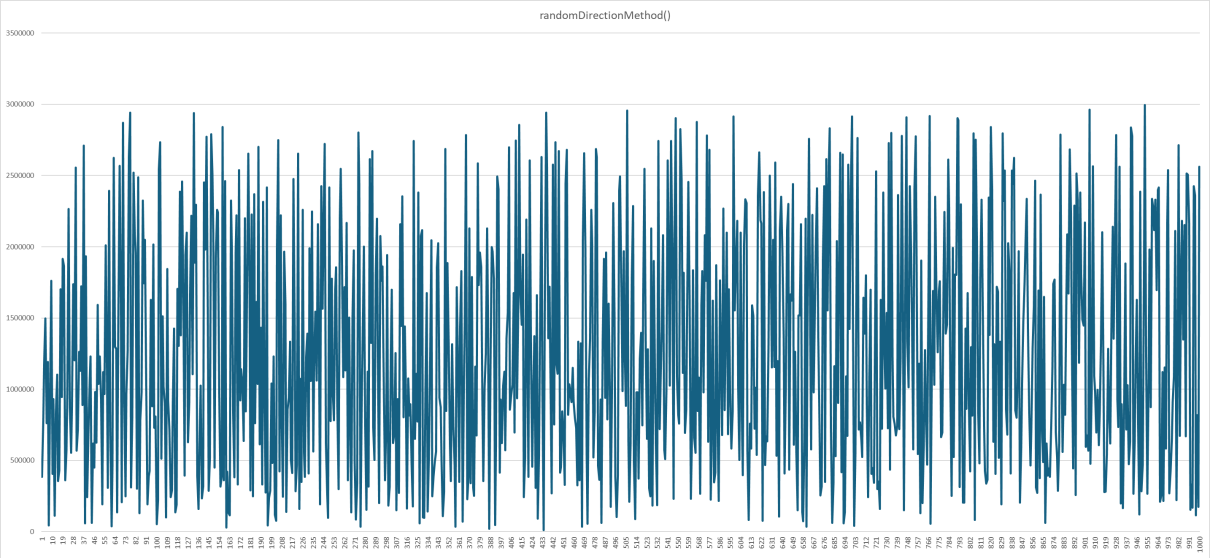
```
                    }
                    if(currentNode.isFinalDestination()) return numberOfSteps;
                    currentNode = currentNode.getLeft();
                    if(!currentNode.getVisited()){
                        currentNode.setVisited();
                        numberOfSteps++;
                    }
                    if(currentNode.isFinalDestination()) return numberOfSteps;
                }
                else if(i == 2){
                    for (int j = 1; j <= i; j++) {
                        currentNode = currentNode.getDown();
                        if(!currentNode.getVisited()){
                            currentNode.setVisited();
                            numberOfSteps++;
                        }
                        if(currentNode.isFinalDestination()) return numberOfSteps;
                    }
                }
                else if (i % 2 == 0) {
                    for (int j = 1; j <= i; j++) {
                        currentNode = currentNode.getLeft();
                        if(!currentNode.getVisited()){
                            currentNode.setVisited();
                            numberOfSteps++;
                        }
                        if(currentNode.isFinalDestination()) return numberOfSteps;
                    }
                    for (int j = 1; j <= i; j++) {
                        currentNode = currentNode.getDown();
                        if(!currentNode.getVisited()){
                            currentNode.setVisited();
                            numberOfSteps++;
                        }
                        if(currentNode.isFinalDestination()) return numberOfSteps;
                    }
                } else {
                    for (int j = 1; j <= i; j++) {
                        currentNode = currentNode.getRight();
                        if(!currentNode.getVisited()){
                            currentNode.setVisited();
                            numberOfSteps++;
                        }
                        if(currentNode.isFinalDestination()) return numberOfSteps;
                    }
                    for (int j = 1; j <= i; j++) {
                        currentNode = currentNode.getUp();
                        if(!currentNode.getVisited()){
                            currentNode.setVisited();
                            numberOfSteps++;
                        }
                        if(currentNode.isFinalDestination()) return numberOfSteps;
                    }
                }
            }
        }
        return numberOfSteps;
    }
```

5.4 Random

1) Wybiera losowo
2) Wybiera nieodwiedzone wierzchołki

Startujemy z punku (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:



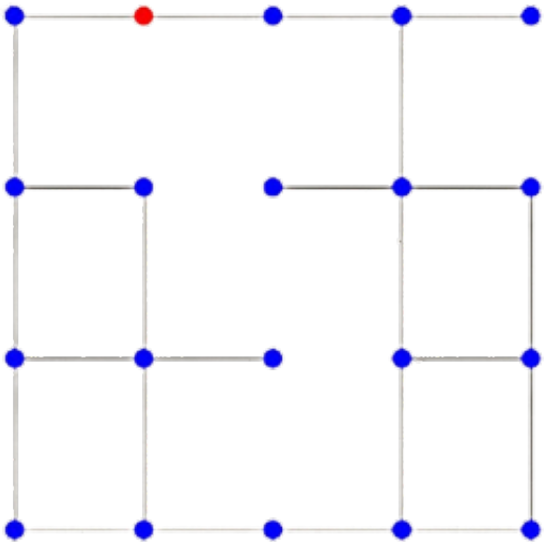
Tak jak nasza metoda jest losowa, ciężko przewidzieć ile wynosi średnia liczba potrzebnych kroków do znajdowania skarbu. W naszym przykładzie ta liczba wynosi 1255954.

Poniżej przedstawiamy fragment kodu tej metody:

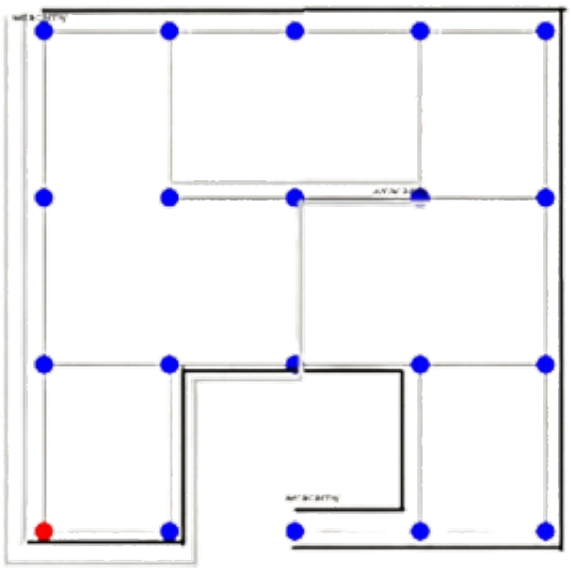
```
1     public int randomDirectionMethod(){
2         int numberOfSteps = 0;
3         Node currentNode = startPoint;
4         Stack<Node> stack = new Stack<>();
5         if(currentNode.isFinalDestination()) return numberOfSteps;
6         stack.push(currentNode);
7         while (!stack.isEmpty() && !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

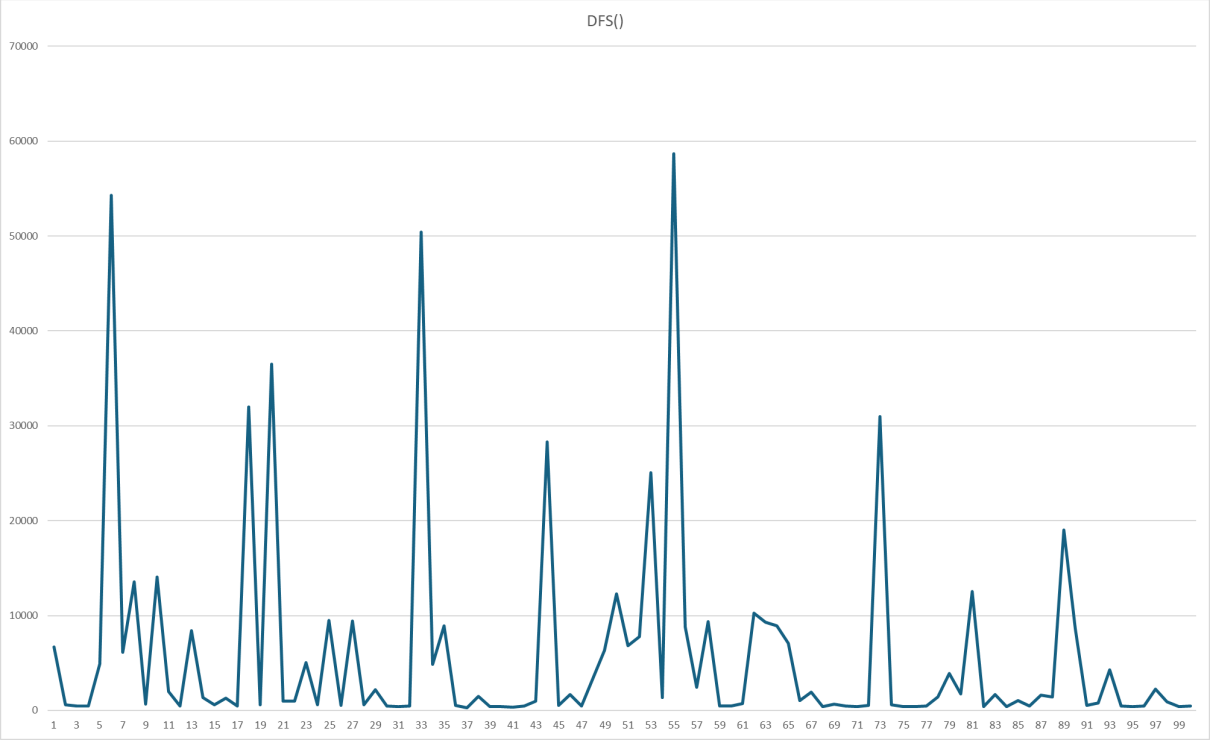
1) BFS nie będzie działać na naszym grafie
(bo nie możemy przeskakiwać po przekątnej)
Używając BFS będziemy mieć taką drogą: $(0,0) \rightarrow (0,1)$ i $(1,0)$



2) DFS dla prawdopodobnej wyspy
Pierwszeństwo: prawo, dół, lewo, góra
Startujemy z punku (0,0) i idziemy, dopóki możemy w prawo, zatem w dół, w lewo i do góry. Jeżeli mamy sytuację bez wyjścia, czyli nie odwiedziliśmy wszystkie wierzchołki, 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:

```
    public int depthFirstSearch() {
        int numberOfSteps = 0;
        if(startPoint.isFinalDestination()){
            return numberOfSteps;
        }
        Set<Node> visited = new HashSet<>();
        Stack<Node> path = new Stack<>();
        List<Node> result = new ArrayList<>();
        path.push(startPoint);

        while (!path.isEmpty()) {
            Node current = path.peek();
            if(current.isFinalDestination()){
                result.add(current);
                numberOfSteps++;
                return numberOfSteps;
            }
            result.add(current);
            numberOfSteps++;

            if (!visited.contains(current)) {
                visited.add(current);
            }

            Node next = getUnvisitedNeighbor(current, visited);
            if (next != null) {
                path.push(next);
            } else {
                path.pop();
            }
        }
```

```
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.