

Labyrinth – Der Weg zum Ziel finden

Projektarbeit HS 2009

eingereicht an der

Hochschule für Technik Zürich

vorgelegt von

Andreas Brönnimann, Andreas Müller, Linda Hartmann

Klasse I08b

Betreuer

Kruse Lars Peter Dr. sc. techn., dipl.Ing.ETH

04. Februar 2010

Inhalt

Inhalt	1
Einleitung	2
Einschränkungen	3
Such-Algorithmen im Labyrinth	4
Klassendiagramm	10
Generierung des Labyrinths.....	14
GUI Programmierung.....	16
Beschreibung des GUI.....	17
Vergleich Tiefensuche / Breitensuche.....	18
Stolpersteine.....	19
Projektmanagement	20
Fazit	22
Quellen	24
Anhang	25

Einleitung

Als wir für die Projektarbeit ein Thema gesucht haben, war für uns schnell klar, dass wir uns mit Labyrinthen auseinandersetzen wollten.

Es interessiert uns, einem Problem nachzugehen, für das es mehrere Lösungsansätze gibt (Breiten- und Tiefensuche). Und man nicht mit Bestimmtheit voraussagen kann, welche Lösung die bessere sein wird.

Sich mit einem Suchalgorithmus so intensiv zu befassen, bis man ihn wirklich versteht – das war die Herausforderung, die uns gereizt hat.

Weiter wird das Suchen eines Weges heute in der Praxis oft benötigt und zwar im Navigationssystem. Wir erhofften uns durch die Implementierung dieser Suchalgorithmen auch die Berechnungen eines Navigationssystems besser verstehen zu können.

Im Folgenden gehen wir genauer auf unsere Einschränkungen, den Projektverlauf und unsere Umsetzung ein.

Einschränkungen

Nach einer längeren Phase, während der wir möglichst viel über die Labyrinth-Problematik lasen, entschieden wir uns, das Projekt folgendermassen einzuschränken:

- Die Programmiersprache soll Java sein, da uns diese vertraut ist.
- Es soll einen Anfangs- und einen Endpunkt geben, die sich irgendwo im Labyrinth befinden.
- Es soll nur einen möglichen Weg geben.
- Zudem soll die Möglichkeit bestehen, sowohl mit Breiten- wie auch mit Tiefensuche zu einer Lösung zu gelangen.
- Das Programm soll ausbaufähig, also erweiterbar sein. Dies wird durch den modularen Aufbau des Programms gewährleistet.
- Über ein GUI sollte der Benutzer möglichst viele Einstellungen selber vornehmen können, er kann zwischen Breiten- und Tiefensuche wählen, den Start- und Endpunkt selber bestimmen, die Grösse des Labyrinths definieren und die zeitliche Verzögerung der Lösungsdarstellung angeben.

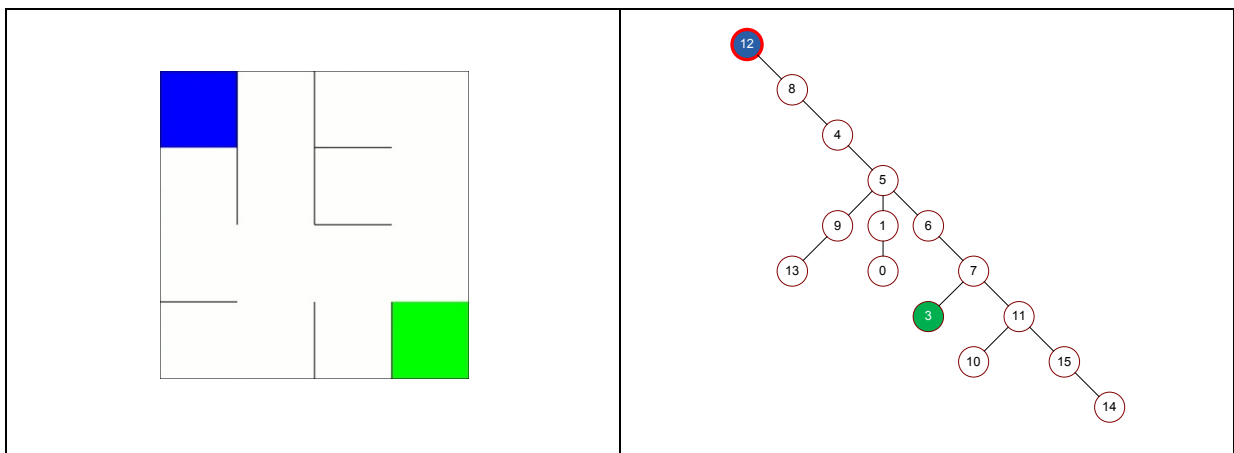
Such-Algorithmen im Labyrinth

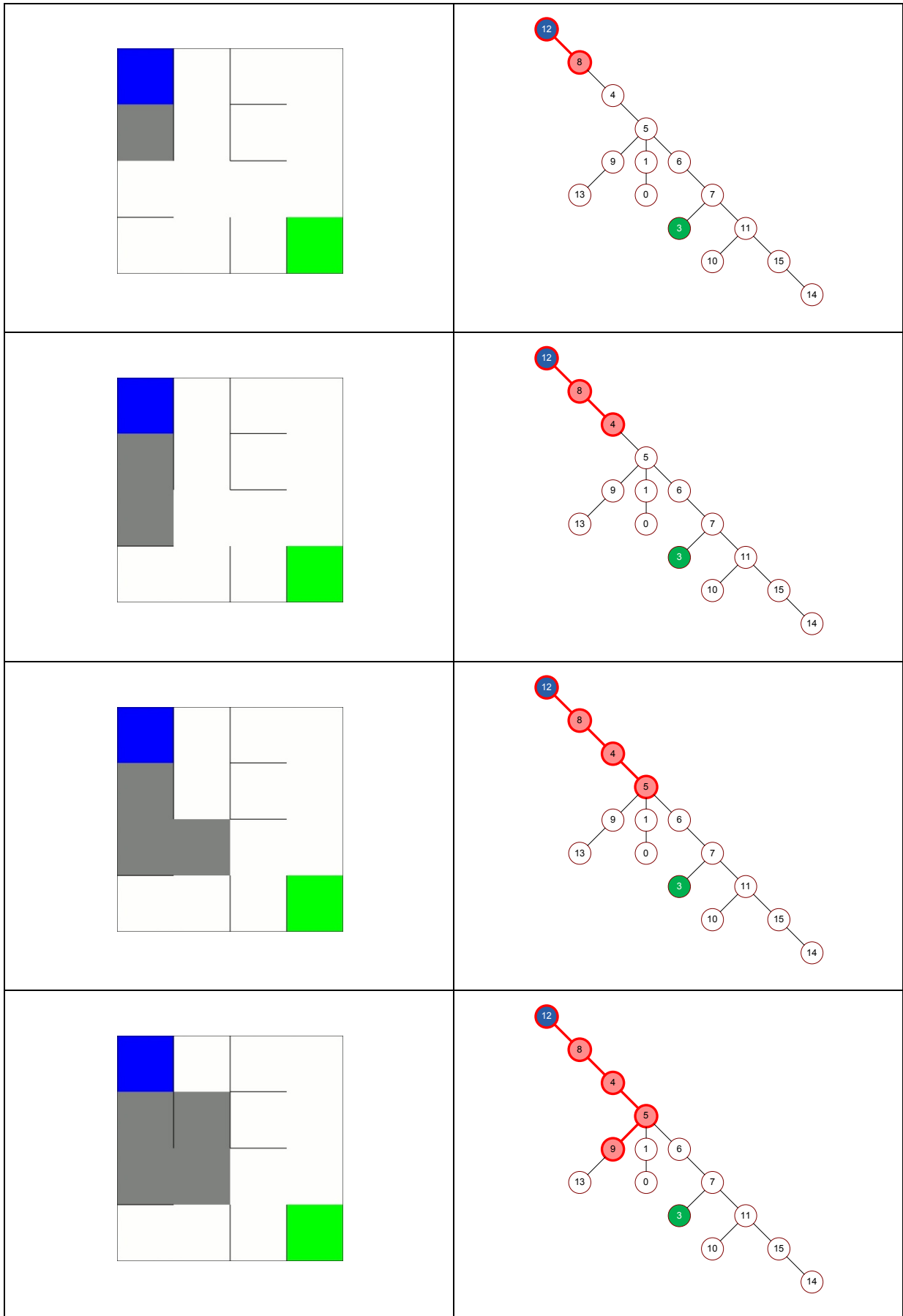
Tiefensuche

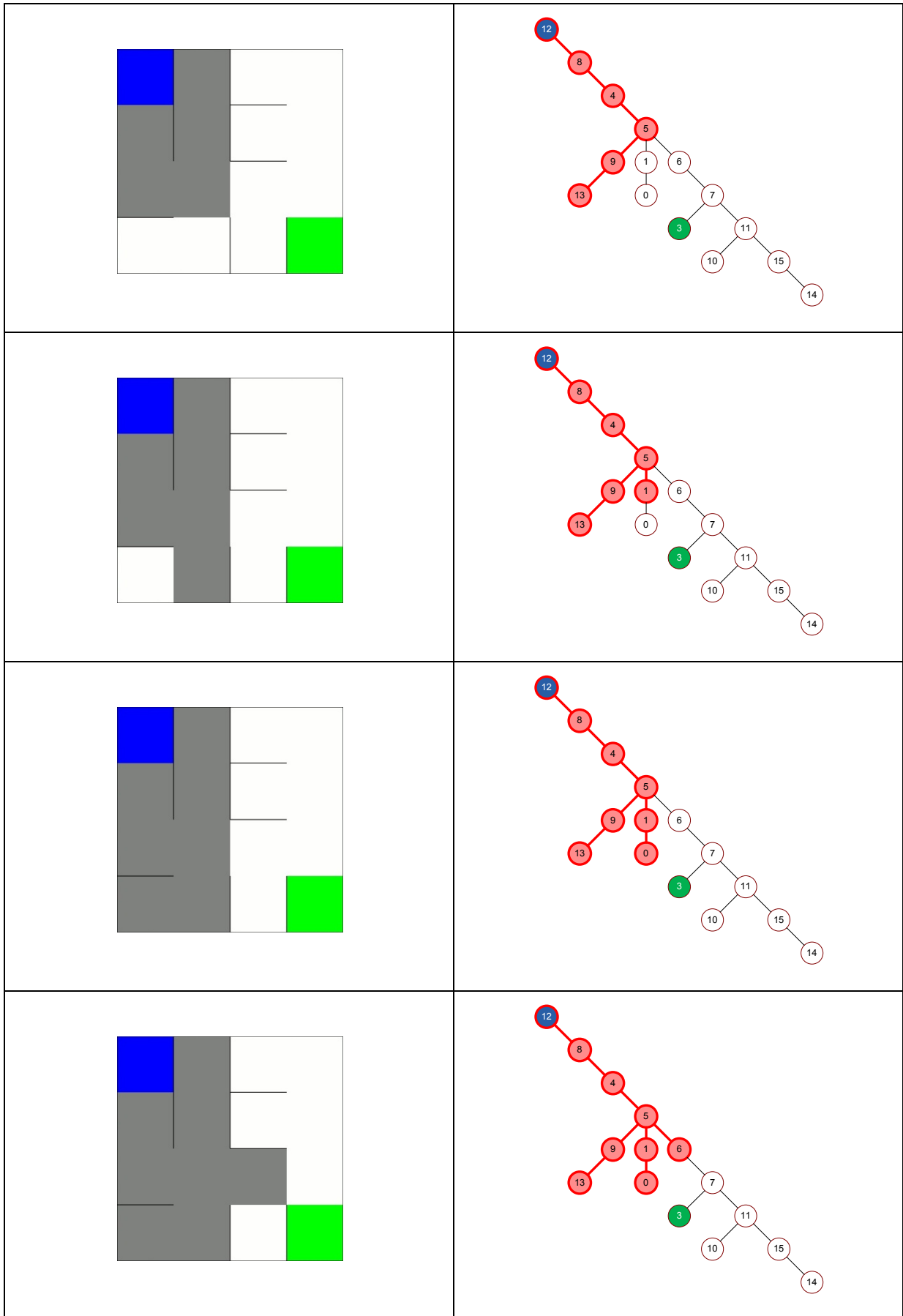
Die Tiefensuche bezeichnet ein bestimmtes Verfahren zum Auffinden eines Knotens in einem Graphen. Wir haben den Startknoten des gesuchten Weges als Anfangsknoten definiert. Jeder Knoten hat bis zu vier Nachbarn. Per Zufallsprinzip wählen wir einen dieser Nachbarn aus. Dieser Nachbar hat wieder bis zu vier Nachbarn. Wir schreiten zu einem von ihnen und fragen dann wiederum dessen Nachbar-Knoten ab. So laufen wir immer weiter, von einem Nachbar zum nächsten. Eine Sackgasse ist erreicht, wenn ein erreichter Knoten von drei Wänden umgeben ist.

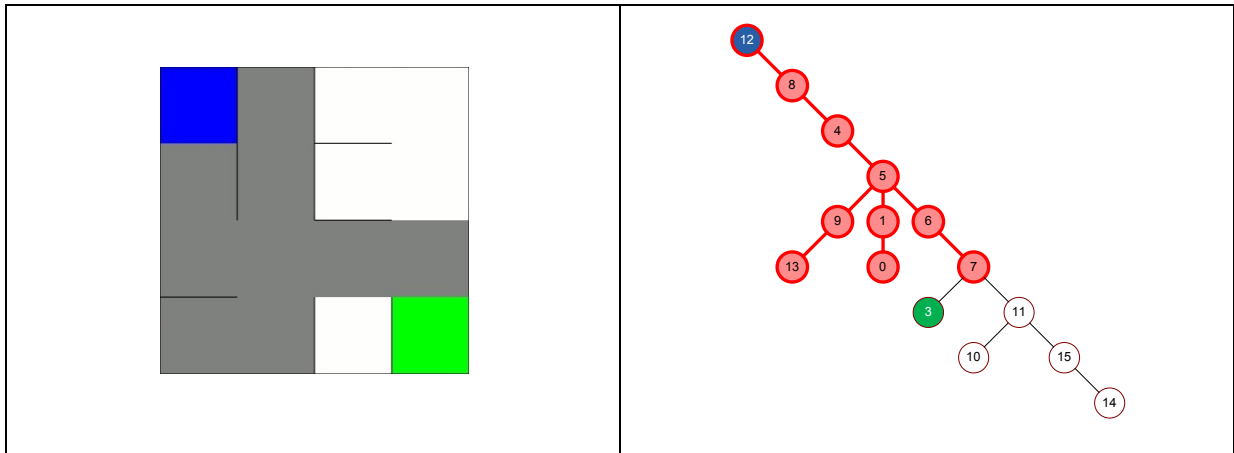
Ist eine Sackgasse erreicht, so geht er zum letzten Knoten, welcher noch nicht besuchte Nachbarn besitzt, zurück und sucht dort weiter. Die nicht besuchten Nodes erkennen wir, weil wir das Attribut „visited“ der beschrifteten Nodes auf „true“ setzen. Wenn der erreichte Knoten gleich dem Ziel ist, so stoppt die Tiefensuche und die Lösung, also der Weg vom Start- zum Endpunkt ist gefunden.

Man kann sich diese Suche gut als Baumstruktur vorstellen, bei welcher man immer bis zum äussersten Blatt oder zum Wurzelknoten geht. Sobald man ganz aussen angekommen ist, geht man zum letzten Blatt zurück, das noch weitere, nicht beschrittene Nachbarknoten hat.





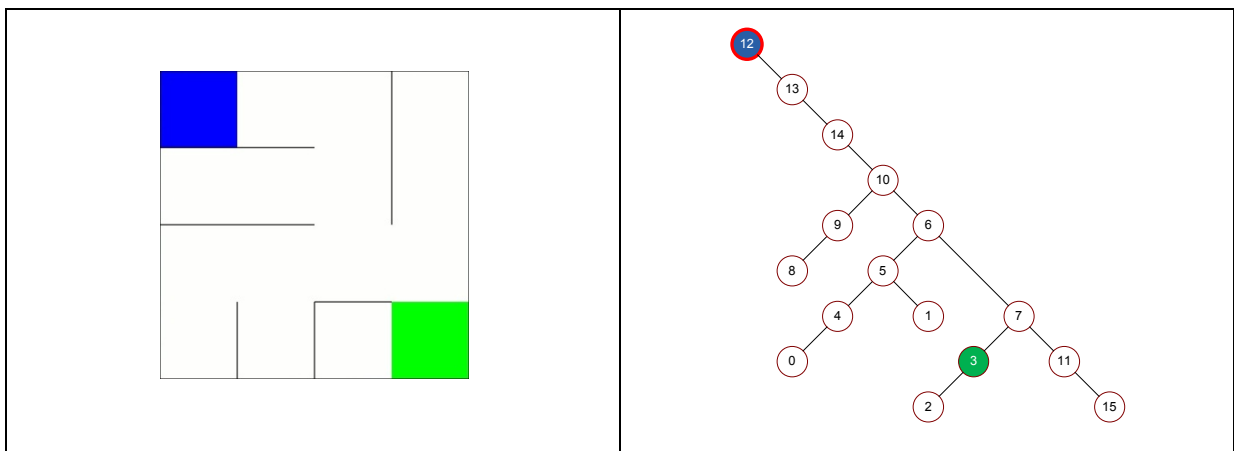


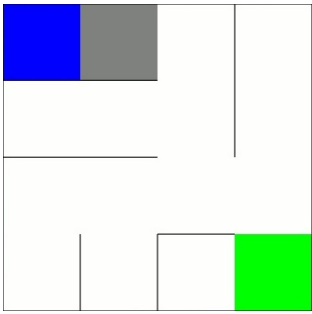
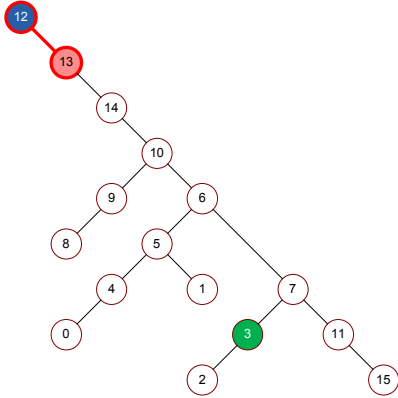
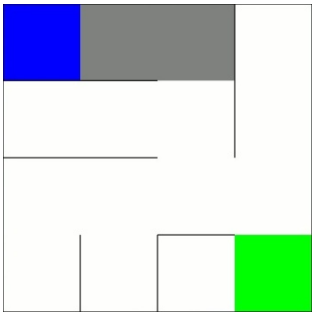
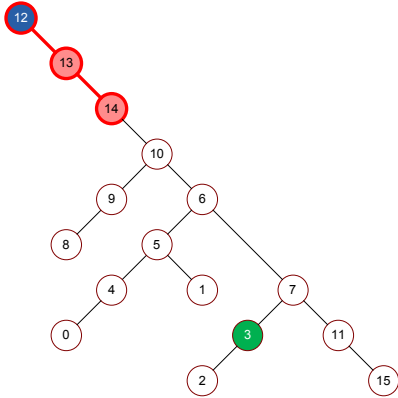
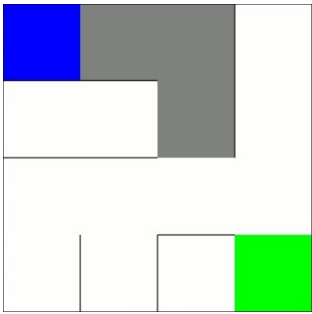
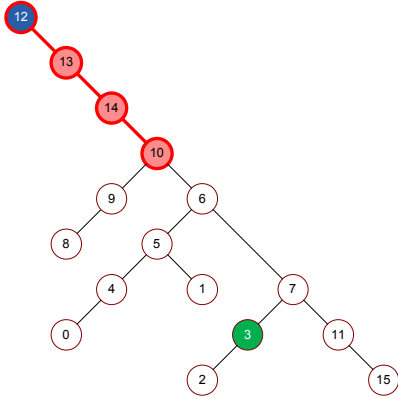
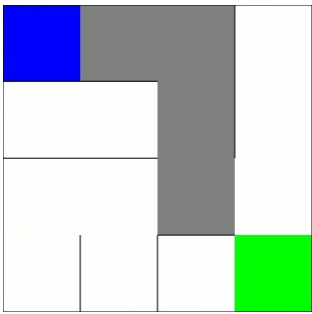
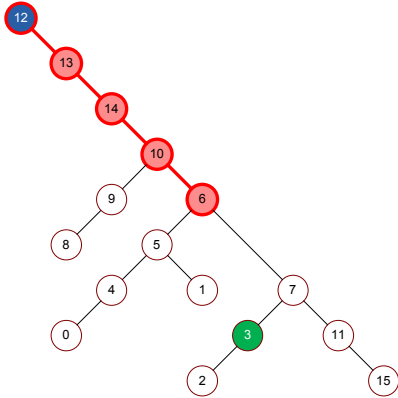


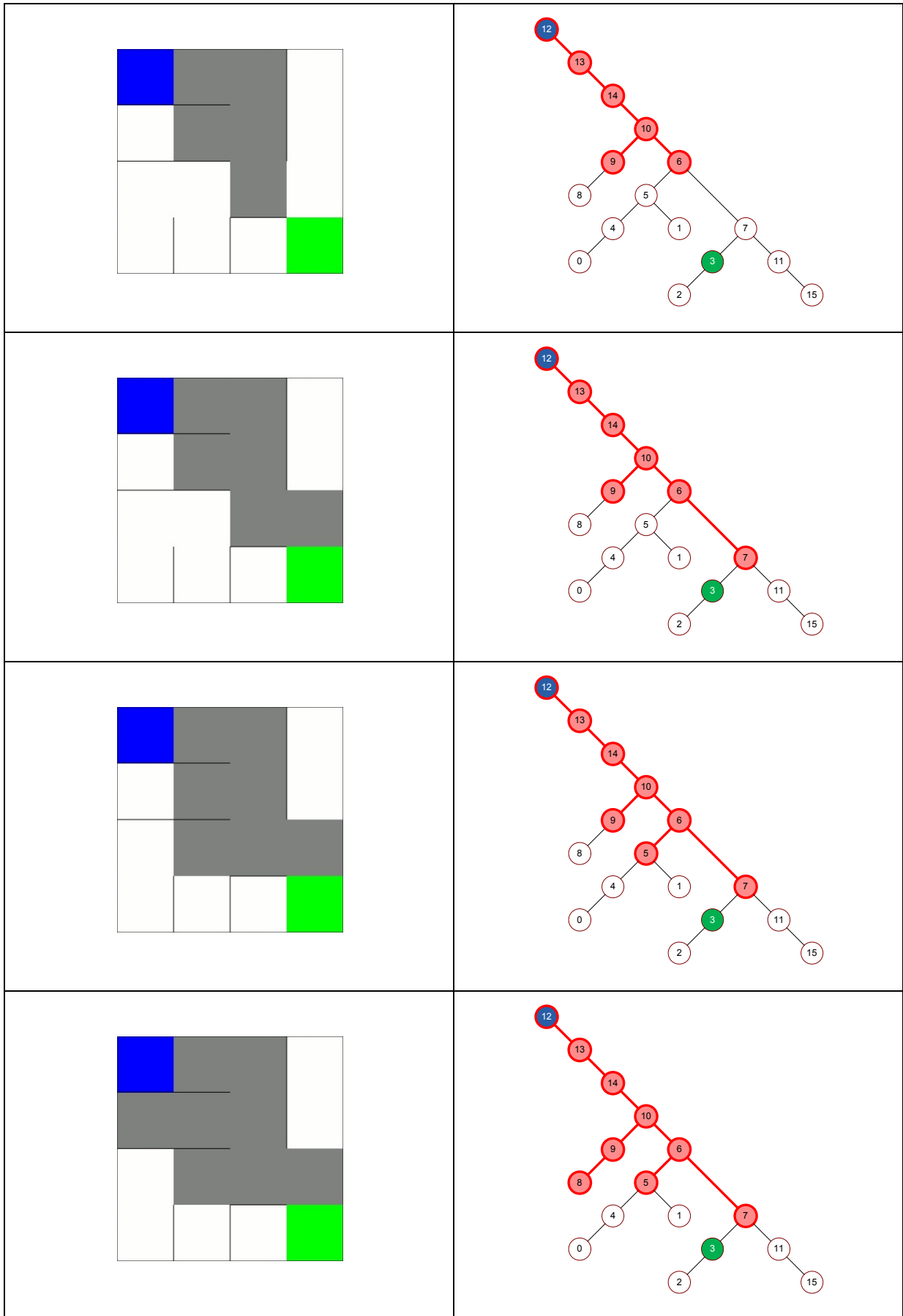
Breitensuche

Auch bei der Breitensuche beginnt die Suche in unserem Fall beim Startknoten. Der Unterschied zur Tiefensuche zeigt sich darin, dass jetzt alle Nachbar-Knoten in eine Warteschlange kommen und dann jeweils nur einen Schritt von jedem Knoten in der Warteschlange aus gemacht wird, anstatt viele Schritte hintereinander in dieselbe Richtung.

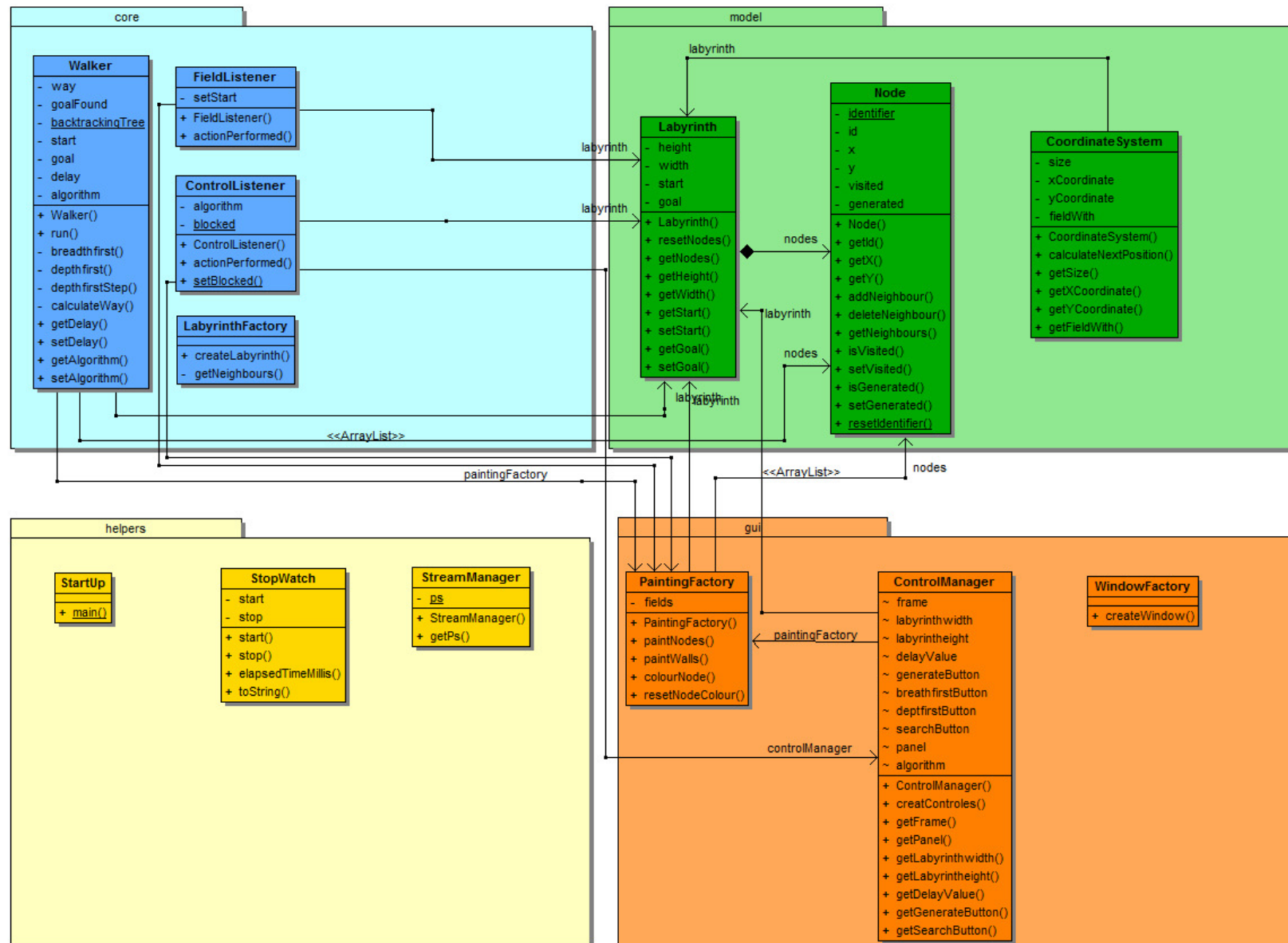
Die Suche geht sichtbar in die Breite, da die Nachbarn der Knoten in der Warteschlange auch wieder in die Warteschlange kommen. So geht es weiter, bis der Ziel-Knoten gefunden wird. Beim Suchprozess wird eine grosse Fläche sichtbar, bei welcher fast zeitgleich in alle Richtungen gesucht wird.





Klassendiagramm



Labyrinth

In der Klasse Labyrinth werden alle Eigenschaften des Labyrinths gespeichert, dazu gehören:

- ArrayList Nodes → enthält alle Knoten des Labyrinths
- Height und width → Höhe und Breite des Labyrinths (Anzahl Knoten)
- Start und goal → Start und Ziel

CoordinateSystem

Das Koordinatensystem berechnet die Koordinaten der Nodes auf einem definierten Label.

Node

Die Klasse Node verwaltet alle Eigenschaften der einzelnen Nodes:

- id → Eindeutige Identifikationsnummer des Nodes
- x → X-Koordinate des Nodes
- y → Y-Koordinate des Nodes
- ArrayList neighbours → Alle zugänglichen Nachbarn des Nodes
- visited → Ob der Knoten bei der Suche bereits einmal betreten wurde
- generated → Ob der Knoten bei der Generierung bereits einmal berücksichtigt wurde

Zusätzlich enthält die Klasse eine statische Variable identifier, welche den Nodes ihre eindeutige ID vergibt.

ControlListener

Alle Ereignisse auf dem ControlPanel werden vom ControlListener verarbeitet:

- Generieren des Labyrinths

- Suchen des Weges
- Ändern des Suchalgorithmus

FieldListener

Alle Ereignisse auf dem Labyrinth, werden vom FieldListener verarbeitet:

- Setzen des Startknotens
- Setzen des Zielknotens

LabyrinthFactory

Erstellt ein zufälliges Labyrinth aufgrund der Eingabeparameter height und width. Wenn das Labyrinth generiert wurde, wird es in Form einer ArrayList zurückgegeben.

Walker

Der Walker sucht den Weg vom Start zum Zielpunkt. Dazu verwendet er den im GUI angegebenen Suchalgorithmus.

StartUp

Initialisiert alle zum Start benötigten Komponenten und erzeugt das Hauptfenster.

StopWatch

Misst die Zeit zwischen dem gesetzten Start- und Stop-Zeitpunkt und gibt diese in Millisekunden zurück.

StreamManager

Leitet alle Konsolenausgaben (system.out) auf eine Textbox im GUI um.

ControlManager

Beinhaltet alle Control-Komponenten, welche auf dem GUI angezeigt werden.

PaintingFactory

Zeichnet das Labyrinth auf einem Panel und stellt verschiedene Funktionen zum einfärben einzelner, bzw. mehrerer Nodes bereit.

WindowFactory

Erstellt ein Fenster aufgrund der Input-Parameter title, height und width.

Generierung des Labyrinths

Zur Generierung des Labyrinths wird wie folgt vorgegangen:

Als erstes wird ein zufälliger Startpunkt innerhalb des Koordinatensystems gesucht. Dies entspricht einer Zufallszahl, welche sich zwischen 0 und der Anzahl Knoten befindet. Dieser Knoten wird dann in eine ArrayList *nodeBucket* eingefügt:

```
randomStartNr = (int) (Math.random()*nodes.size());  
nodeBucket.add(nodes.get(randomStartNr));
```

Anschliessend wird in einer Schleife dieses Array laufend abgearbeitet und mit neuen Knoten wieder gefüllt. Ein beliebiger Knoten wird aus der ArrayList *nodeBucket* genommen (zu Beginn befindet sich nur der Starknoten in der ArrayList, somit wird dieser als erster genommen) und überprüft, welche Nachbarn dieses Knotens noch isoliert (also von vier Wänden umgeben) sind. Sollte der Nachbar isoliert sein, so wird die Wand zu ihm durchbrochen.

```
while (!nodeBucket.isEmpty()) {  
    // take a node out of the bucket  
    randomNodeNr = (int) (Math.random()*nodeBucket.size());  
  
    // get current node and remove it from the list  
    currentNode = nodeBucket.get(randomNodeNr);  
    nodeBucket.remove(randomNodeNr);  
  
    // get all neighbours of the current node (doesn't  
    // matter if they are connected or not)  
    neighbours = getNeighbours(currentNode, nodes, height,  
                                width);  
  
    // loop through all neighbours  
    for(int i = 0; i < neighbours.size(); i++){  
        // get the current neighbour  
        currentNeighbour = neighbours.get(i);  
  
        // get these neighbours neighbours  
        nNeighbours = currentNeighbour.getNeighbours();  
    }  
}
```

```

// if there are no neighbours in the list,
the node is blocked
if(nNeighbours.size() < 1){
    // set connection to the neighbour
    currentNode.addNeighbour(currentNeighbour);

    // set connection backwards
    currentNeighbour.addNeighbour(currentNode);
}

```

Zum Schluss wird geprüft, ob der Nachbar bereits einmal im *nodeBucket* gewesen ist und den Generierungsprozess entsprechend schon einmal durchlaufen hat. Falls nicht, wird der Nachbar dem *nodeBucket* hinzugefügt.

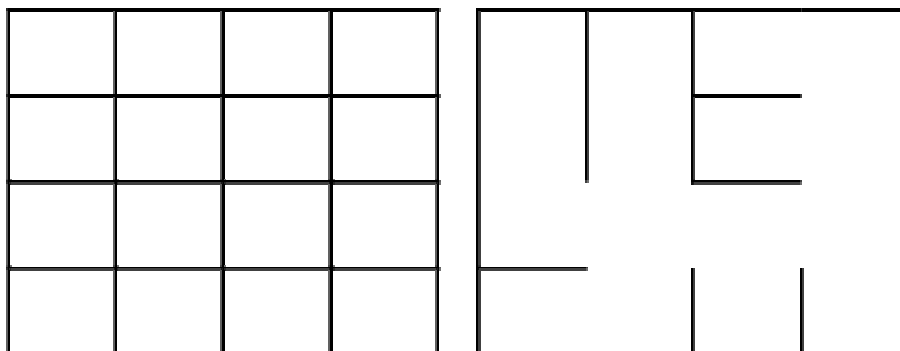
```

// add the neighbour to the bucket
if(!currentNeighbour.isGenerated()){
    nodeBucket.add(currentNeighbour);
    currentNeighbour.setGenerated(true);
}

```

Nun wird dieser Vorgang so lange wiederholt, bis der *nodeBucket* leer ist. Dies ist der Fall, wenn alle Knoten mindestens eine Verbindung besitzen.

Durch dieses Vorgehen ist gewährleistet, dass es immer nur einen möglichen Weg gibt. Wenn ein Knoten bereits über eine Verbindung verfügt, werden keine weiteren Verbindungen zugelassen.



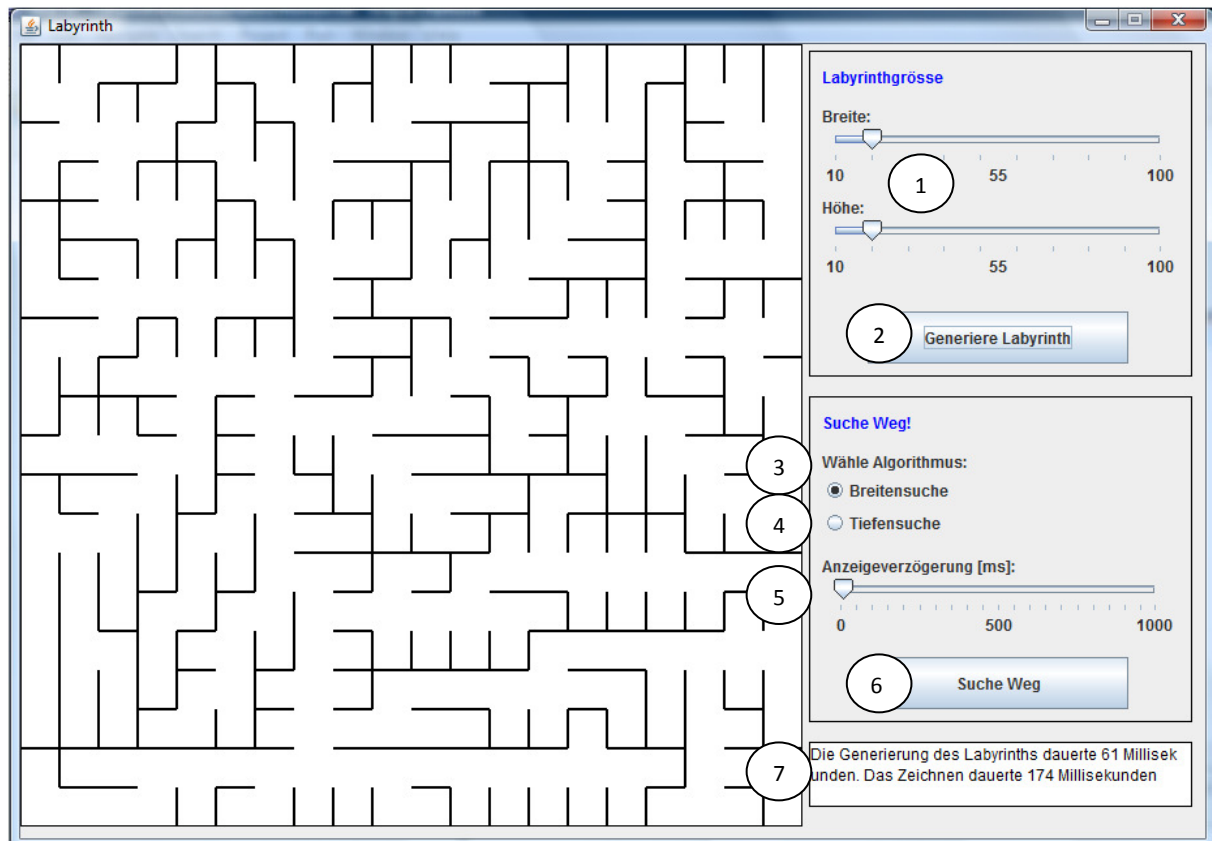
GUI Programmierung

Das GUI, welches wir skizzierten, sollte einfach und intuitiv bedienbar sein. Der Benutzer, welcher dieses Programm öffnet, soll sich nicht lange einarbeiten müssen, sondern direkt loslegen können.

Zu Beginn hatten wir nur ein Panel mit allen Bedienelementen erstellt. Dies war unvorteilhaft, denn aufgrund der Anordnung der Elemente, war ein Layoutmanager nicht brauchbar. Deshalb mussten wir die Elemente manuell platzieren, was wiederum ziemlich aufwändig und unübersichtlich war. Ausserdem konnte so keine klare Abgrenzung der einzelnen Schritte dargestellt werden.

Schlussendlich haben wir uns entschieden, das Controlpanel modular aufzubauen. Das heisst, ein Panel für die Generierung und ein Weiteres für die Suche des Weges.

Beschreibung des GUI



- 1 Hier kann man die Anzahl der Felder für die Höhe und Breite des Labyrinths definieren. Die Skala geht von 10 bis 100.
- 2 Mit Drücken des Buttons „Generiere Labyrinth“ wird der Konstruktor der Klasse Labyrinth mit den Werten aus Punkt 1 aufgerufen. Sobald das Labyrinth generiert wurde, wird es auf dem Panel gezeichnet.
- 3 Festlegen des Algorithmus Breitensuche.
- 4 Festlegen des Algorithmus Tiefensuche.
- 5 Hier kann die Verzögerung der Anzeige (in Millisekunden) gewählt werden. So kann der User je nach Bedarf die Suche Schritt für Schritt verfolgen und nachvollziehen.
- 6 Hiermit wird die Funktion run() aufgerufen, die Suche nach dem Weg beginnt. Dazu werden die Parameter aus den Punkten 3 bis 5 verwendet.
- 7 Im Ausgabefeld erhält der Benutzer Zusatzinformationen: Hier sieht man wie lange die Labyrinth-Generierung dauerte und wie lange die Weg-Suche dauerte. Dies ermöglicht einen Vergleich der beiden Algorithmen.

Vergleich Tiefensuche / Breitensuche

Geschwindigkeit

Bei Punkten, welche nahe beieinander liegen, ist die Tiefensuche meist schneller, da sie strukturiert vom Startpunkt aus einen Weg sucht. Sobald die Punkte jedoch weit auseinander liegen gibt es kein eindeutiges Ergebnis. Da die Tiefensuche ihren Weg bei Verzweigungen zufällig wählt, kann sie sehr schnell zum Ziel kommen, oder auch nicht.

Laufzeit

Die Laufzeit der Breitensuche, wie auch diejenige der Tiefensuche ist linear. Bei beiden Algorithmen ist die Laufzeit abhängig von der Anzahl der Knoten (V) und der Anzahl der Kanten (E): Für beide gilt somit die Laufzeit von $O(V+E)$.

Kürzester Pfad

Da die Breitensuche alle Knoten mit demselben Abstand zum Startknoten direkt nacheinander absucht, findet sie immer den optimalen Weg.

Da es bei den Labyrinthen in unserem Projekt jedoch immer nur einen Weg gibt, konnte diese Eigenschaft vernachlässigt werden.

Stolpersteine

Ein Labyrinth zu erzeugen, welches bei jedem Generieren andere Wege aufweist, in die vorgegebene Grösse das Koordinatensystem passt und nur einen bestimmten Weg von Punkt A nach B zulässt, gestaltete sich schwieriger, als gedacht. Das Labyrinth soll ja einerseits zufällig generiert sein, andererseits den oben genannten Kriterien entsprechen.

Das grösste Problem dabei war, dass es nur einen Weg vom Start zum Ziel geben sollte. Nach viel herumprobieren und einer Besprechung mit Herrn Kruse, welcher uns den Tipp gab, zuerst alle Knoten des Labyrinths als isoliert (also von vier Wänden umgeben) zu betrachten, haben wir die Lösung schliesslich gefunden. Der dazugehörige Algorithmus ist unter dem Punkt „Generierung des Labyrinths“ beschrieben.

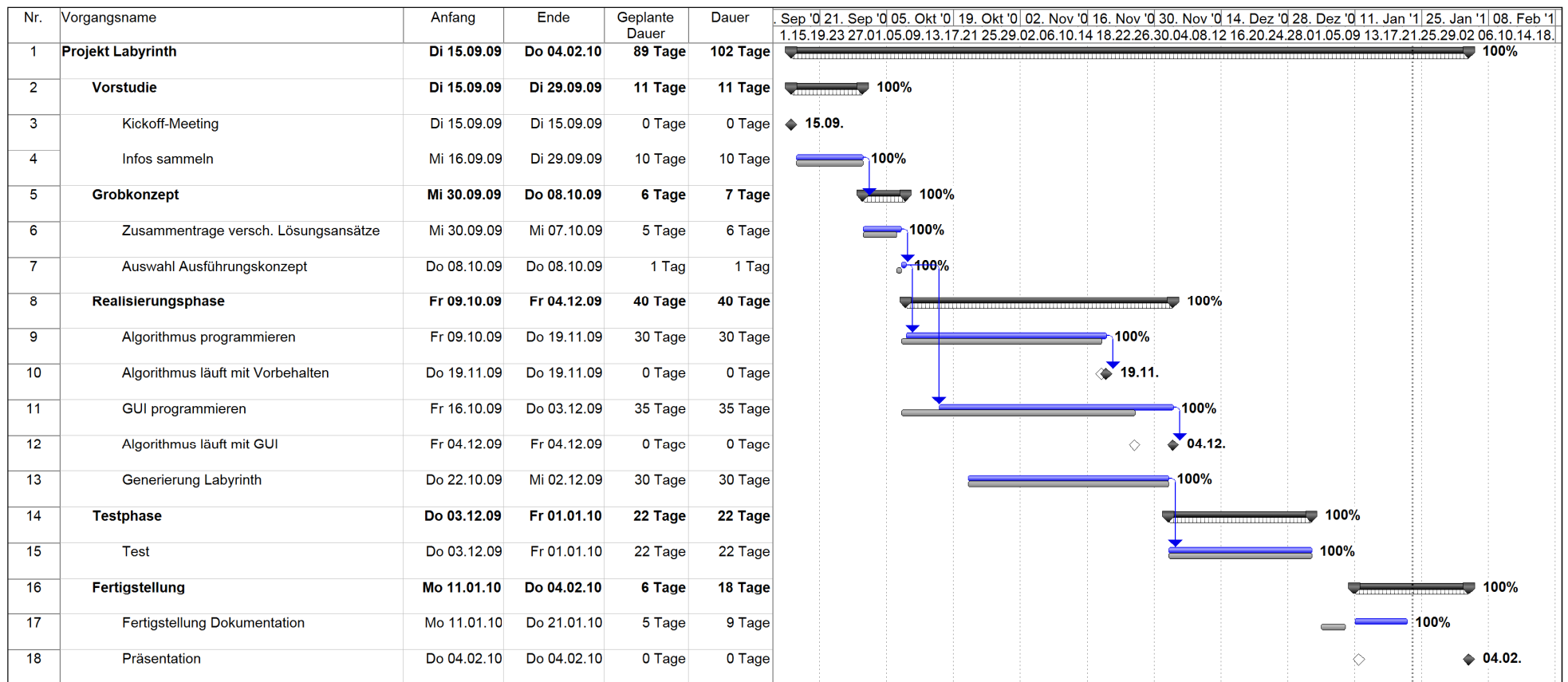
Eine weitere Schwierigkeit war es, das GUI zu programmieren, da wir damit kaum Erfahrung hatten. So mussten wir uns in die entsprechenden Klassen einlesen und die verschiedenen Funktionen ausprobieren, bis es uns gelang alles so darzustellen, wie wir es uns vorgestellt hatten.

Projektmanagement

Der Projektplan konnte bis auf ein paar kleinere Terminverschiebungen eingehalten werden. Durch die Prüfungsflut im Januar mussten wir den Abschluss der Dokumentation etwas hinausschieben und haben deshalb auch den 4. Februar 2010 als Präsentationstermin gewählt.

Der Zeitaufwand belief sich im Rahmen der Vorgaben von 50 Stunden und wurde nicht merklich überschritten.

Bis auf leichte Abweichungen konnten alle Termine eingehalten werden. Die Zeit, welche wir bei einzelnen Phasen gewonnen hatten, verwendeten wir, um die Funktionalität unseres Labyrinths zu erweitern.



Projekt: Labyrinth-Projekt Datum: Sa 23.01.10	Kritisch		Unterbrechung		Geplanter Meilenstein	◇	Projektsammelvorgang	
	Kritische Unterbrechung		Vorgang in Arbeit		Meilenstein	◆	Externe Vorgänge	
	Kritisch in Arbeit		Geplant		Sammelvorgang in Arbeit		Externer Meilenstein	◇
	Vorgang		Geplante Unterbrechung		Sammelvorgang		Stichtag	↓

Fazit

Team

Da wir zu dritt zusammengearbeitet haben, war die die Arbeitsteilung ein entscheidender Faktor. Dank des Quellcodeverwaltungssystems CVS hat die Zusammenarbeit am Projekt sehr gut funktioniert. Sowohl das Programm, wie auch die Dokumentation konnte dadurch von allen drei Teammitgliedern gleichzeitig bearbeitet werden.

Ergebnis

Mit dem erreichten Ergebnis haben wir unsere Erwartungen nicht nur erreicht, wir haben sie sogar übertroffen. Zu Beginn planten wir ein Labyrinth mit festen Start und Zielpunkten, bei welchem der Benutzer lediglich den Weg der Breitensuche visuell verfolgen kann.

Schlussendlich konnten wir das Labyrinth um folgende Funktionalitäten erweitern:

- **Tiefensuche**
Als Algorithmus kann der Anwender zwischen der Breiten- und der Tiefensuche wählen. Die Tiefensuche war nicht geplant, konnte jedoch dank des modularen Aufbaus einfach implementiert werden.
- **Schieberegler für die Grösse des Labyrinths**
Die Grösse des Labyrinths kann über Schieberegler (einer für die Breite und einer für die Höhe) zwischen 10 und 100 eingestellt werden.
- **Eingabemöglichkeit für die Verzögerung**
Die Verzögerung pro Schritt kann in Millisekunden auch mittels Schieberegler definiert werden.
- **Beliebiges Festlegen von Start und Ziel**
Durch klicken auf einen beliebigen Knoten kann der Anwender sowohl den Start- wie auch den Zielpunkt beliebig bestimmen.

Lerneffekt

Unsere Erwartungen sind erfüllt worden: Nach diesem Projekt verstehen wir die beiden Algorithmen Tiefen- und Breitensuche. Dieses Verständnis kam uns auch im Unterricht zu Gute.

Beruflich sind wir alle nicht als ProgrammiererIn tätig. So hat uns das Projekt dabei geholfen, das Gelernte zu vertiefen und anzuwenden. Auch bei der Auseinandersetzung mit der GUI-Programmierung haben wir viel Neues gelernt.

Obwohl unsere Zusammenarbeit als Gruppe gut funktioniert hat, ist es jedes Mal eine gute Erfahrung, sich in einer neuen Gruppe zu organisieren und zurechtzufinden.

Quellen

Internet

- <http://www-i1.informatik.rwth-aachen.de/~algorithmus/algo5.php>, Stand 4.10.2009
- <http://java.sun.com/docs/books/tutorial/uiswing/>, Stand 18.11.2009
- <http://www.codeproject.com/KB/java/BFSDFS.aspx>, Stand 31.10.2009
- <http://www.labyrinth-international.org/cms/index.php?page=411747146&f=1&i=12257&s=411747146>, Stand 29.12.2009

Literatur

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, und Clifford Stein, 2007: Algorithmen – Eine Einführung, Seiten 535-552
- JAVA von Kopf bis Fuss, O'Reilly Verlag Köln 2008, ISBN 978-3-89721-448-4, Seiten 406-417

Anhang

Im Folgenden werden noch die wichtigsten Code-Ausschnitte unseres Programmes aufgezeigt.

Klasse Walker

```
public class Walker extends Thread {
    private ArrayList<Node> nodes;
    private ArrayList<Integer> way = new ArrayList<Integer>();
    private boolean goalFound;
    private static int[] backtrackingTree;
    private int start;
    private int goal;
    private PaintingFactory paintingFactory;
    private int delay;
    private int algorithm;
    private Labyrinth labyrinth;

    public Walker(Labyrinth labyrinth, PaintingFactory paintingFactory){
        nodes = labyrinth.getNodes();
        backtrackingTree = new int[nodes.size()];
        start = labyrinth.getStart();
        goal = labyrinth.getGoal();
        this.labyrinth = labyrinth;
        this.paintingFactory = paintingFactory;
        delay = 1;
        algorithm = 0;
    }

    @Override
    public void run() {
        ControlListener.setBlocked(true);
        paintingFactory.resetNodeColour(nodes);

        // create the Stopwatch to measure the time
        Stopwatch stopWatch = new Stopwatch();
        stopWatch.start();

        switch (algorithm) {
            case 0:
                // start walking
                try {
                    if (breadthfirst(start, goal, paintingFactory,
delay)) {
                        calculateWay(start, goal,
paintingFactory, delay);
                    } else {
                        System.out.println("No way has been
found");
                    }
                } catch (InterruptedException e1) {
                    // TODO Auto-generated catch block
                    e1.printStackTrace();
                }
            }
        }
    }
}
```

```

        }
        break;

    default:
        // start walking
        try {
            if (depthfirst(start, goal, paintingFactory,
delay)) {
                calculateWay(start, goal,
paintingFactory, delay);
            } else {
                System.out.println("No way has been
found");
            }
        } catch (InterruptedException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        break;
    }

    stopWatch.stop();
    System.out.println("Der Weg wurde in " +
stopWatch.elapsedTimeMillis() + " Millisekunden gefunden.");

    labyrinth.resetNodes();

    ControlListener.setBlocked(false);
}

private boolean breadthfirst(int start, int goal,
    PaintingFactory paintingFactory, int delay)
    throws InterruptedException {
    Queue<Node> queue = new LinkedList<Node>();
    ArrayList<Node> neighbours;
    Node parentNode;

    // add start note to queue
    for (Node node : nodes) {
        if (node.getId() == start) {
            queue.add(node);
            break;
        }
    }

    while (!queue.isEmpty()) {
        // mark actual node as visited
        queue.peek().setVisited(true);

        // get neighbours
        neighbours = queue.peek().getNeighbours();

        // goal is reached
        if (queue.peek().getId() == goal) {
            // sleep a bit, before painting the way
            Thread.sleep(delay);
            return true;
        }

        // do not colour the start and the goal node

```

```

        if(queue.peek().getId() != labyrinth.getStart() &&
queue.peek().getId() != labyrinth.getGoal()){
            // paint node to show the way
            paintingFactory.colourNode(queue.peek().getId(),
Color.GRAY);
        }

        // sleep to show how the walking works
        Thread.sleep(delay);

        // save last node to know the parent node
        parentNode = queue.poll();

        for (Node neighbour : neighbours) {
            if (!neighbour.isVisited()) {
                queue.add(neighbour);

                backtrackingTree[neighbour.getId()] =
parentNode.getId();
            }
        }

        // no way has been found
        return false;
    }

    private boolean depthfirst(int start, int goal,
        PaintingFactory paintingFactory, int delay)
        throws InterruptedException {
        goalFound = false;

        // do depthfirstStep with start-node
        for (Node node : nodes) {
            if (node.getId() == start) {
                node.setVisited(true);
                // do not colour the start and the goal node
                if (node.getId() != labyrinth.getStart() &&
node.getId() != labyrinth.getGoal()){
                    // paint node to show the way
                    paintingFactory.colourNode(node.getId(),
Color.GRAY);
                }

                Thread.sleep(delay);
                depthfirstStep(node, goal, paintingFactory, delay);
            }
        }

        return goalFound;
    }

    private void depthfirstStep(Node node, int goal,
        PaintingFactory paintingFactory, int delay)
        throws InterruptedException {
        Node parentNode = node;

        for (Node neighbour : node.getNeighbours()) {
            if (!neighbour.isVisited() && !goalFound) {
                neighbour.setVisited(true);

```

```

        backtrackingTree[neighbour.getId()] =
parentNode.getId();

        // goal is reached
        if (neighbour.getId() == goal) {
            goalFound = true;
        }

        // do not colour the start and the goal node
        if(neighbour.getId() != labyrinth.getStart() &&
neighbour.getId() != labyrinth.getGoal()){
            // paint node to show the way
            paintingFactory.colourNode(neighbour.getId(),
Color.GRAY);
        }

        Thread.sleep(delay);
        depthfirstStep(neighbour, goal, paintingFactory,
delay);
    }
}

private void calculateWay(int start, int goal,
    PaintingFactory paintingFactory, int delay)
    throws InterruptedException {
    int position;

    position = goal;
    while (true) {
        // add position to the way
        way.add(position);

        // check if the starting point is reached
        if (position != start) {
            position = backtrackingTree[position];
        } else {
            break;
        }
    }

    // print the way
    for (int i = way.size() - 1; i >= 0; i--) {
        if(way.get(i) != labyrinth.getStart() && way.get(i) !=
labyrinth.getGoal()){
            paintingFactory.colourNode(way.get(i), Color.RED);
        }
        Thread.sleep(delay);
    }
}

public int getDelay() {
    return delay;
}

public void setDelay(int delay) {
    this.delay = delay;
}

```

```

    }

    public int getAlgorithm() {
        return algorithm;
    }

    public void setAlgorithm(int algorithm) {
        this.algorithm = algorithm;
    }
}

```

Klasse Node

```

public class Node {
    private static int identifier;
    private int id;
    private int x;
    private int y;
    private ArrayList<Node> neighbours;
    private boolean visited;
    private boolean generated;

    public Node(int x, int y){
        id = identifier++;
        this.x = x;
        this.y = y;
        this.visited = false;
        this.generated = false;
        neighbours = new ArrayList<Node>();
    }

    public int getId() {
        return id;
    }

    public int getX() {
        return x;
    }

    public int getY() {
        return y;
    }

    public void addNeighbour(Node neighbour){
        // create some more randomness
        int index = (int)(Math.random()*(neighbours.size()+1));
        neighbours.add(index, neighbour);
    }

    public boolean deleteNeighbour(int id){
        for (Node neighbour : neighbours) {
            if(neighbour.getId() == id){
                neighbours.remove(id);
                return true;
            }
        }
    }
}

```

```

        }
    }

    return false;
}

public ArrayList<Node> getNeighbours(){
    return neighbours;
}

public boolean isVisited() {
    return visited;
}

public void setVisited(boolean visited) {
    this.visited = visited;
}

public boolean isGenerated(){
    return generated;
}

public void setGenerated(boolean generated){
    this.generated = generated;
}

public static void resetIdentifier(){
    identifier = 0;
}
}

```

Klasse Labyrinth

```

public class Labyrinth {
    private ArrayList<Node> nodes = new ArrayList<Node>();
    private int height;
    private int width;
    private int start;
    private int goal;

    public Labyrinth(int height, int width) {
        this.height = height;
        this.width = width;
        LabyrinthFactory labyrinthFactory = new LabyrinthFactory();
        nodes = labyrinthFactory.createLabyrinth(height, width);
    }

    public void resetNodes(){
        for (Node node : nodes) {
            node.setVisited(false);
        }
    }

    public ArrayList<Node> getNodes() {
        return nodes;
    }
}

```

```
public int getHeight() {  
    return height;  
}  
  
public int getWidth() {  
    return width;  
}  
  
public int getStart() {  
    return start;  
}  
  
public void setStart(int start) {  
    this.start = start;  
}  
  
public int getGoal() {  
    return goal;  
}  
  
public void setGoal(int goal) {  
    this.goal = goal;  
}  
}
```