

# Aufgabe 6: Tiefensuche im Labyrinth

[Aufgabe 6: Tiefensuche im Labyrinth](#)

[Theorie zum Graphen](#)

[Ungerichtete Graphen](#)

[Gewichtete graphen](#)

[Die Matrix der Graphen](#)

[Graphen als Klassen](#)

[Ungewichteter Ungerichteter Graph](#)

[Ungewichteter Gerichteter Graph](#)

[Gewichteter Ungerichteter Graph](#)

[Gewichteter Gerichteter Graph](#)

[Backtracking](#)

[Anwendung](#)

[O\\_graph\\_node](#)

[Backtracking](#)

[Resultate](#)

[Breadth first search \(BFS\) vs. Depth first search \(DFS\)](#)

[Der DFS- und BFS Algorithmus animiert.](#)

[Nutzen des BFS oder A\\* Algorithmus](#)

[Links](#)

[Repository](#)

[Interactive Demo](#)

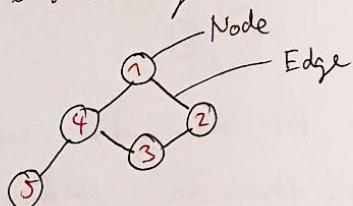
[Videos](#)

## Theorie zum Graphen

Was ist ein Graph?

Ein Graph visualisiert die Beziehungen zwischen den Knoten in einem Netzwerk. Beziehungen zwischen Komponenten/Knoten werden "Edges" genannt.

Die Knoten/Komponenten werden "Nodes" / "Vertices" genannt.

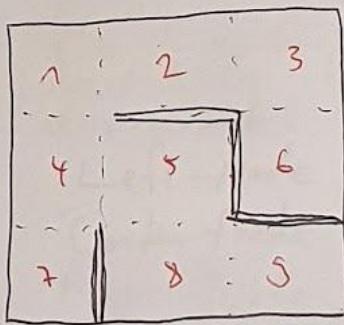
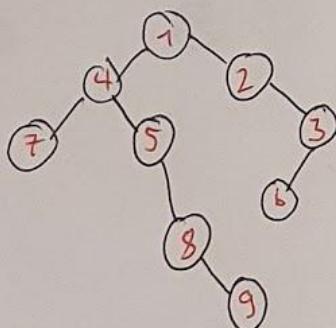


Wir können einen Graphen auch als Labyrinth darstellen

1	2
4	3
5	

Mit der Voraussetzung dass nur nicht diagonal angrenzende Felder beschriftet werden können, ist der oben gezeigte Graph als Labyrinth visualisiert

Ein Graph ist nicht zwingend ein Baum, ist dies jedoch der Fall so gibt es lediglich einen möglichen Pfad vom Start-Node zum Ziel-Node.



Das Labyrinth ist nun ein "Perfect-Maze" / perfektes Labyrinth, da es von jedem beliebigen Start-Node nur genau einem Weg/Pfad zu einem Target-Node gibt.

Was ist Tiefe-Suche (eng. Depth-First-Search, "DFS")

```
graph TD; 1((1)) --- 2((2)); 1 --- 3((3)); 2 --- 4((4)); 2 --- 5((5)); 3 --- 6((6)); 3 --- 7((7)); 6 --- 8((8)); 6 --- 9((9)); 9 --- 10((10)); 9 --- 11((11)); 11 --- 12((12))
```

Wir starten vom "Root-Node" hier ① und falls ① nicht der Target-Node ist, gehen wir zum Next-Node.

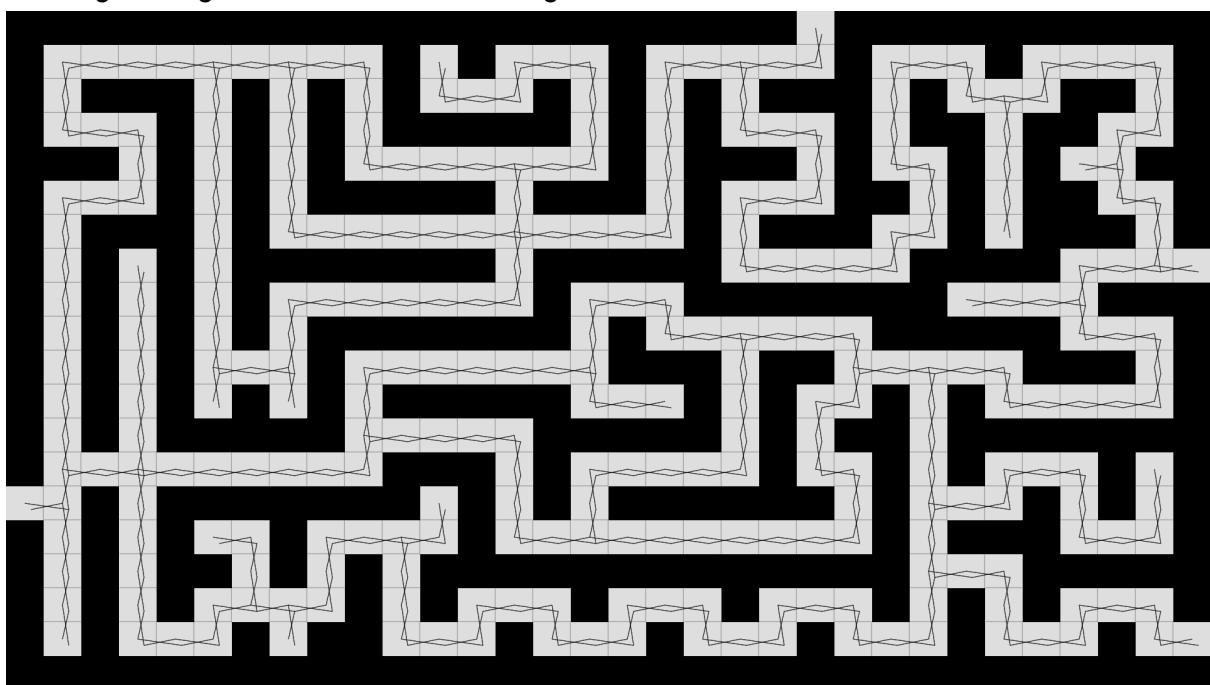
Die Prozedur zum finden des Next-Node ist folgendermassen definiert:

- Left-Node, falls verfügbar, sonst
- Center-Node, falls verfügbar, sonst
- Right-Node, falls verfügbar, sonst

Zurück zum letzten Node und Prozedur erneut starten

## Ungerichtete Graphen

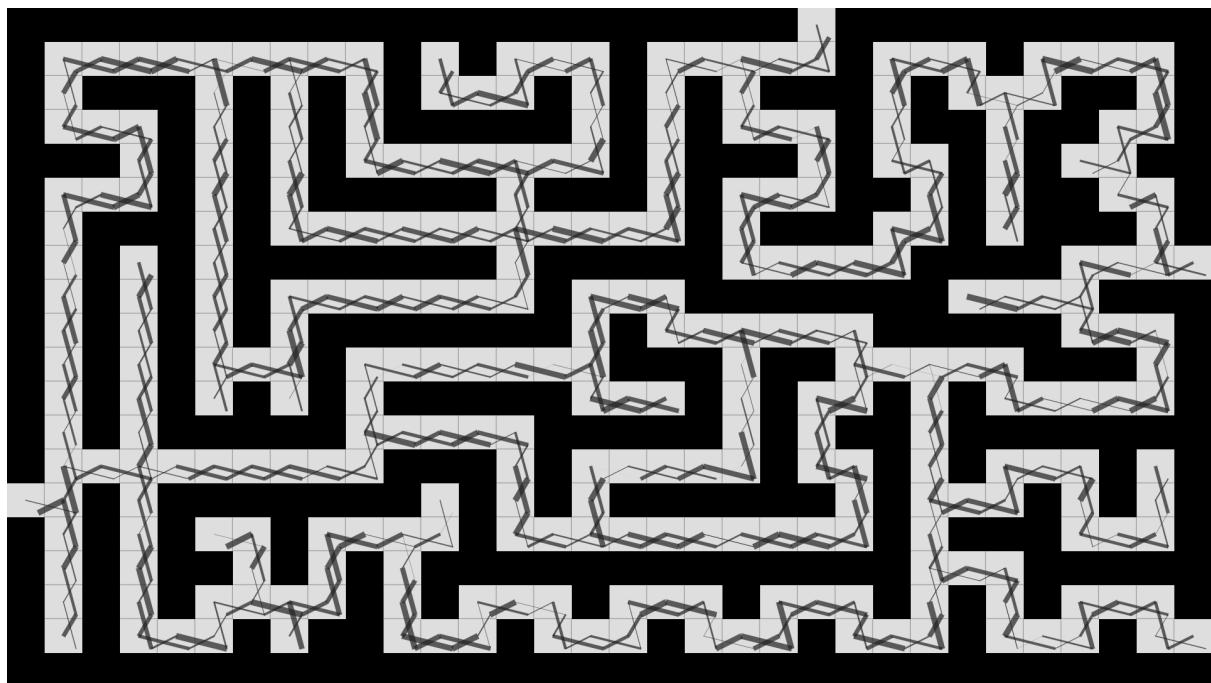
Der Begriff gerichtet/ungerichtet bezieht sich in einem Graphen auf die mögliche Traversierungs-Richtung von einem Node auf den Anderen, welche über einen "Edge" geschieht. Bei einem ungerichteten Graphen gibt es pro "edge" jeweils mehr als nur eine mögliche Richtung. Ein Labyrinth entspricht einem ungerichteten Graphen, weil das Labyrinth beliebig durchlaufen werden kann. Um zu visualisieren dass es sich im labyrinth um einen ungerichteten Graphen handelt, habe ich jede Edge von einem Node zum anderen im Zentrum des Nodes begonnen jedoch nicht im Zentrum des \_\_connected node's beendet sondern um ein paar pixel verschoben auf beiden achsen (x und y). Dies ergibt ein gekreuztes muster aller edges.



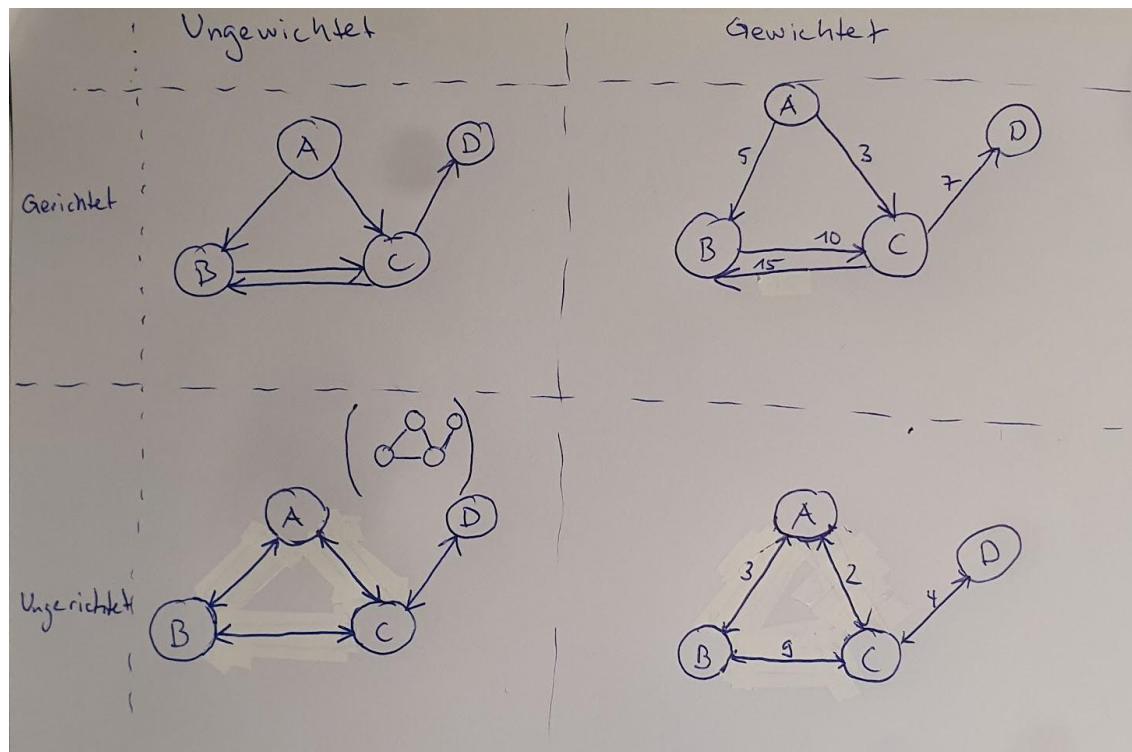
## Gewichtete graphen

Ein gewichteter Graph kann ein gerichteter oder auch ein ungerichteter Graph sein. Bei dem gewichteten Graphen hat jede Verbindung von einem Node zu einem anderen ein gewisses Gewicht ( "weight" ). Das "weight" ist eine Zahl. Die Zahl kann angeben, welche "Kosten" anfallen, falls die Edge traversiert wird, also falls die Verbindung zum Node eingegangen wird.

Um die Gewichtung zu visualisieren, habe ich die edges von jedem Node mit einer entsprechenden Liniendicke dargestellt.



## Die Matrix der Graphen



## Graphen als Klassen

Folgend beschreibe ich, wie die Graphen im Source Code als Klassen erfasst werden können. Natürlich gibt es auch andere Architekturen, wie man Nodes und deren Relationen im Source Code erfassen kann, ich zeige hier nur eine vieler Arten auf, wie es funktioniert. Dabei nutze ich den Vorteil von Javascript Objekten und deren Verhalten als Referenzen bei Neuzuweisung eines bestehenden Objektes zu einer Variable oder in ein Array.

## Ungewichteter Ungerichteter Graph

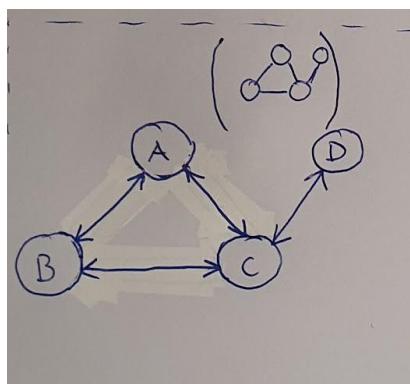
```

class O_graph_node_undirected_unweighted{
    constructor(
        a_o_graph_node_undirected_unweighted
    ){
        this.a_o_graph_node_undirected_unweighted
        |   = a_o_graph_node_undirected_unweighted // array with object references
    }
}

var o_graph_node_undirected_unweighted_A = O_graph_node_undirected_unweighted([]);
var o_graph_node_undirected_unweighted_B = O_graph_node_undirected_unweighted([]);
var o_graph_node_undirected_unweighted_C = O_graph_node_undirected_unweighted([]);
var o_graph_node_undirected_unweighted_D = O_graph_node_undirected_unweighted([]);

o_graph_node_undirected_unweighted_A.a_o_graph_node_undirected_unweighted.push(
    o_graph_node_undirected_unweighted_B,
    o_graph_node_undirected_unweighted_C,
);
o_graph_node_undirected_unweighted_B.a_o_graph_node_undirected_unweighted.push(
    o_graph_node_undirected_unweighted_A,
    o_graph_node_undirected_unweighted_C,
);
o_graph_node_undirected_unweighted_C.a_o_graph_node_undirected_unweighted.push(
    o_graph_node_undirected_unweighted_A,
    o_graph_node_undirected_unweighted_B,
    o_graph_node_undirected_unweighted_D,
);
o_graph_node_undirected_unweighted_D.a_o_graph_node_undirected_unweighted.push(
    o_graph_node_undirected_unweighted_C,
);

```



Der wohl einfachste Typus von graph node. Er besitzt lediglich ein Array mit Referenzen zu weiteren graph nodes. Natürlich kann ein Node selbst auch noch Informationen besitzen, zum Beispiel einen Namen, wie hier auf dem Bild, "A", "B", "C", "D". Diese Namen habe ich jedoch auf dem Node Objekt weggelassen und direkt in den Variablennamen gespeichert.

## Ungewichteter Gerichteter Graph

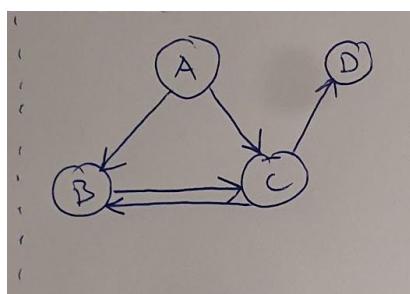
```

class o_graph_node_directed_unweighted{
    constructor(a_o_graph_node_directed_unweighted){
        this.a_o_graph_node_directed_unweighted
            = a_o_graph_node_directed_unweighted // array with object references
    }
}

var o_graph_node_directed_unweighted_A = o_graph_node_directed_unweighted([]);
var o_graph_node_directed_unweighted_B = o_graph_node_directed_unweighted([]);
var o_graph_node_directed_unweighted_C = o_graph_node_directed_unweighted([]);
var o_graph_node_directed_unweighted_D = o_graph_node_directed_unweighted([]);

o_graph_node_directed_unweighted_A.a_o_graph_node_directed_unweighted.push(
    o_graph_node_directed_unweighted_B,
    o_graph_node_directed_unweighted_C,
);
o_graph_node_directed_unweighted_B.a_o_graph_node_directed_unweighted.push(
    o_graph_node_directed_unweighted_C,
);
o_graph_node_directed_unweighted_C.a_o_graph_node_directed_unweighted.push(
    o_graph_node_directed_unweighted_B,
    o_graph_node_directed_unweighted_D,
);

```



Der ungewichtete, gerichtete node unterscheidet sich kaum vom ungewichteten ungerichteten node. Die reine Existenz einer Referenz zu einem anderen Objekt zeigt auf, ob der Node zum nächsten traversierbar ist oder nicht. So zum Beispiel kommt man im oben gezeigten Muster, vom Node A zum Node B, jedoch nicht vom Node B zum Node A, da Node B keine Referenz von dem Objekt `o\_graph\_node\_directed\_unweighted\_A` in seinem Array gespeichert hat.

## Gewichteter Ungerichteter Graph

```

class O_graph_node_undirected_weighted{
    constructor(
        a_o_edge_weighted
    ){
        this.a_o_edge_weighted
            = a_o_edge_weighted
    }
}

class O_edge_weighted{
    constructor(
        a_o_graph_node_undirected_weighted,
        n_weight,
    ){
        this.a_o_graph_node_undirected_weighted = a_o_graph_node_undirected_weighted // array with object references
        this.n_weight = n_weight
    }
}

var o_graph_node_undirected_weighted_A = O_graph_node_undirected_weighted([]);
var o_graph_node_undirected_weighted_B = O_graph_node_undirected_weighted([]);
var o_graph_node_undirected_weighted_C = O_graph_node_undirected_weighted([]);
var o_graph_node_undirected_weighted_D = O_graph_node_undirected_weighted([]);

var o_edge_weighted_A_B = new O_edge_weighted(
    [ o_graph_node_undirected_weighted_A, o_graph_node_undirected_weighted_B ],
    3
)
var o_edge_weighted_B_C = new O_edge_weighted(
    [ o_graph_node_undirected_weighted_B, o_graph_node_undirected_weighted_C ],
    9
)
var o_edge_weighted_A_C = new O_edge_weighted(
    [ o_graph_node_undirected_weighted_A, o_graph_node_undirected_weighted_C ],
    2
)
var o_edge_weighted_C_D = new O_edge_weighted(
    [ o_graph_node_undirected_weighted_C, o_graph_node_undirected_weighted_D ],
    4
)
o_graph_node_undirected_weighted_A.a_o_edge_weighted.push(
    o_edge_weighted_A_B,
    o_edge_weighted_A_C
);
o_graph_node_undirected_weighted_B.a_o_edge_weighted.push(
    o_edge_weighted_A_B,
    o_edge_weighted_B_C
);
o_graph_node_undirected_weighted_C.a_o_edge_weighted.push(
    o_edge_weighted_B_C,
    o_edge_weighted_A_C,
    o_edge_weighted_C_D
);
o_graph_node_undirected_weighted_D.a_o_edge_weighted.push(
    o_edge_weighted_C_D
);

```

Dies ist ein durchaus komplizierter node und beansprucht eine zweite Klasse aus dem Grund , dass eine "edge"/"kante" (die Verbindung zwischen zwei nodes, ein Gewicht enthält.

## Gewichteter Gerichteter Graph

```

class O_graph_node_directed_weighted{
    constructor(
        a_o_edge_weighted
    ){
        this.a_o_edge_weighted
            = a_o_edge_weighted
    }
}

class O_edge_weighted{
    constructor(
        o_graph_node_directed_weighted__target,
        n_weight,
    ){
        this.o_graph_node_directed_weighted__target = o_graph_node_directed_weighted__target // array with object references
        this.n_weight = n_weight
    }
}

var o_graph_node_directed_weighted_A = O_graph_node_directed_weighted();
var o_graph_node_directed_weighted_B = O_graph_node_directed_weighted();
var o_graph_node_directed_weighted_C = O_graph_node_directed_weighted();
var o_graph_node_directed_weighted_D = O_graph_node_directed_weighted();

```

```

o_graph_node_directed_weighted__A.a_o_edge_weighted.push(
    new O_edge_weighted(
        o_graph_node_directed_weighted__B,
        5
    ),
    new O_edge_weighted(
        o_graph_node_directed_weighted__C,
        3
    )
);

o_graph_node_directed_weighted__B.a_o_edge_weighted.push(
    new O_edge_weighted(
        o_graph_node_directed_weighted__C,
        10
    ),
);
o_graph_node_directed_weighted__C.a_o_edge_weighted.push(
    new O_edge_weighted(
        o_graph_node_directed_weighted__B,
        15
    ),
    new O_edge_weighted(
        o_graph_node_directed_weighted__D,
        7
    ),
);

```

Der wohl komplexeste Typ eines Node. Da jeder node eine einzigartige gewichtete "edge" hat, speichere ich diese direkt als neue Instanzen in dem array auf dem node Objekt.

## Backtracking

Backtracking, auf deutsch Rückverfolgung, ist die Definition für eine Problemlösungsmethode in der Algorithmik. Wie das Wort rückverfolgung schon verrät, gibt es im Backtracking den Vorteil, dass alle durchlaufenden/durchsuchten Informationen gespeichert werden und danach jederzeit wieder aufrufbar sind.

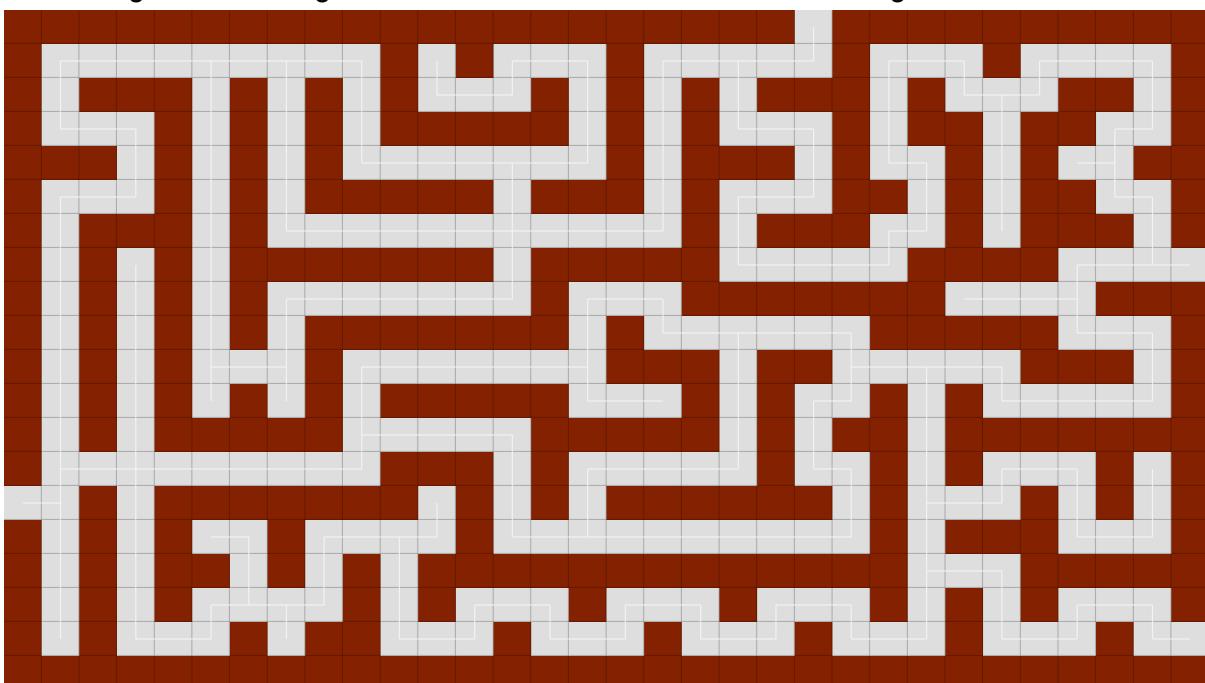
Bei der Tiefensuche kommt Backtracking ins Spiel, nachdem der Algorithmus durchgelaufen ist und ein entsprechendes Objekt mit den Koordinaten von verbundenen Nodes angelegt wurde. Es enthält dann alle wichtigen Informationen, um den Pfad zu rekonstruieren.

## Anwendung

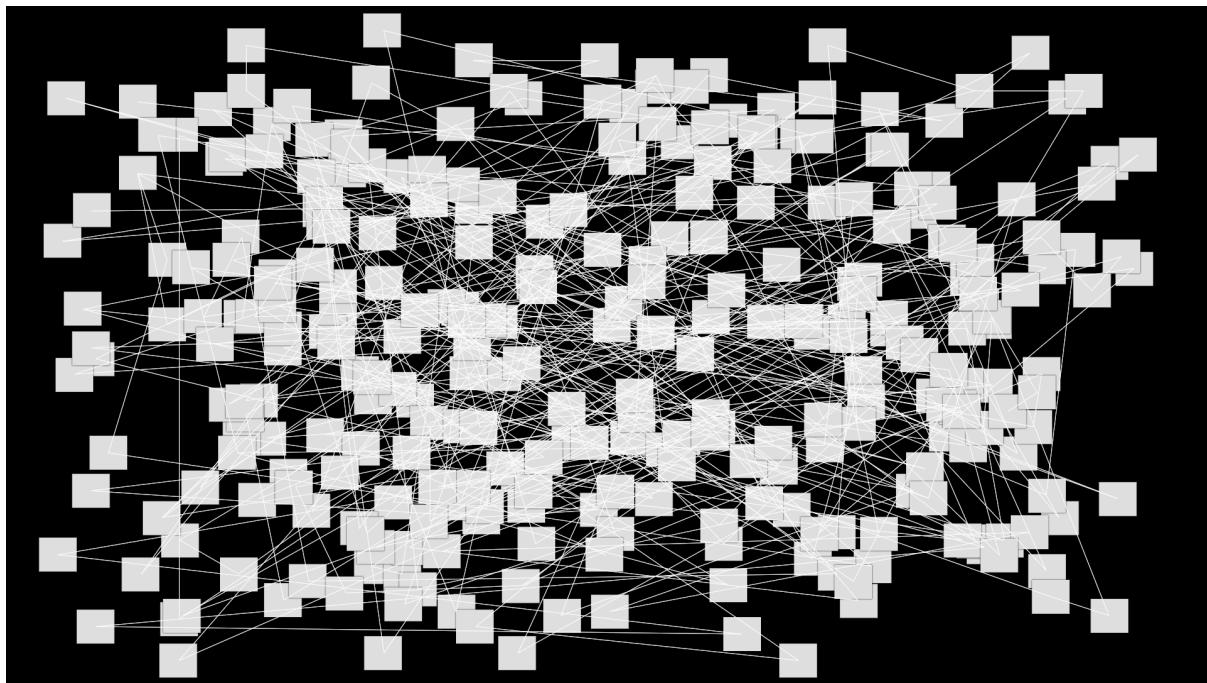
Mit Hilfe von Source-Code habe ich im Labyrinth aus der Aufgabe jede Wand erkannt und alles, was keine Wand ist, als Instanz von `O\_graph\_node` im array `a\_o\_graph\_node` abgespeichert.

Danach habe ich für jeden `o\_graph\_node` im array `a\_o\_graph\_node` durchiteriert und überprüft, ob er jeweils oben/unten/recht/links einen angrenzenden Node besitzt. In jenem Falle habe ich diesen `o\_graph\_node\_\_connected` als Referenz mit entsprechendem Namen `o\_graph\_node\_\_up`, `...\_\_down`, `...\_\_left`, `...\_\_right` auf dem Objekt `o\_graph\_node` abgespeichert.

Diesen Graphen habe ich nun dargestellt. Die Nodes sind weiß, ihre entsprechenden Verbindungen zu den angrenzenden Nodes sind als weiße Linien dargestellt.



Da es mich wundernahm wie es aussehen würde wenn alle Nodes eine zufällige Position besitzen habe ich allen eine zufällige Position gegeben.



DFS - Depth First Search Algorithmus

### O\_graph\_node

Da ein graph node in einem Labyrinth nur maximal 4 Verbindungen zu weiteren Nodes hat, habe ich beschlossen, die folgende Klasse als Repräsentation für einen Graph node zu verwenden.

```
class O_graph_node{
    constructor(
        o_object_2d,
        o_graph_node_left,
        o_graph_node_right,
        o_graph_node_up,
        o_graph_node_down,
    ){
        this.o_object_2d = o_object_2d
        this.o_graph_node_left = o_graph_node_left
        this.o_graph_node_right = o_graph_node_right
        this.o_graph_node_up = o_graph_node_up
        this.o_graph_node_down = o_graph_node_down
    }
}
```

Das Objekt `o\_object\_2d` ist eine Instanz der Klasse `O\_object\_2d` und wird verwendet, um im Grafik-Rendering-Prozess auf eine JavaScript Canvas gezeichnet zu werden.

Wie bereits erwähnt, habe ich im Labyrinth alle Felder als Instanzen von `O\_object\_2d` erfasst. Die felder mit einer helligkeit unter einem schwellwert habe ich mit namen "wall" erfasst und alle anderen mit namen "path".

Danach habe ich über alle die Objekte mit Namen "path" durchiteriert und eine neue Instanz `O\_graph\_node` erstellt.

```
for(var o_object_2d of a_o_object_2d_wall_and_path){
    if(o_object_2d.s_name == "path"){
        var o_graph_node = new O_graph_node(
            o_object_2d,
            null,
            null,
            null,
            null,
        )
        a_o_graph_node.push(o_graph_node)
    }
}
```

Schlussendlich habe ich noch einmal über alle `O\_graph\_node` Instanzen durchiteriert und existierende angrenzende `O\_graph\_node` Instanzen als Referenzen abgespeichert.

```
for(var o_graph_node of a_o_graph_node){
    for(var s_side of a_s_side){
        if(s_side == "left"){
            var n_x = o_graph_node.o_object_2d.n_x + -1;
            var n_y = o_graph_node.o_object_2d.n_y + 0;
        }
        if(s_side == "right"){
            var n_x = o_graph_node.o_object_2d.n_x + +1;
            var n_y = o_graph_node.o_object_2d.n_y + 0;
        }
        if(s_side == "up"){
            var n_x = o_graph_node.o_object_2d.n_x + 0;
            var n_y = o_graph_node.o_object_2d.n_y + -1;
        }
        if(s_side == "down"){
            var n_x = o_graph_node.o_object_2d.n_x + 0;
            var n_y = o_graph_node.o_object_2d.n_y + +1;
        }
        var o_graph_node_connected = a_o_graph_node.filter(
            o =>
                parseInt(o.o_object_2d.n_x) == parseInt(n_x)
                &&
                parseInt(o.o_object_2d.n_y) == parseInt(n_y)
        )[0]
        if(o_graph_node_connected){
            o_graph_node[`o_graph_node_${s_side}`] = o_graph_node_connected
        }
    }
}
```

Nun war ich bestens darauf vorbereitet den Algorithmus anzuwenden.

Um die Tiefensuche zu starten, muss die Funktion `f\_o\_traversal\_result\_\_traversal\_dfs` aufgerufen werden. Sie benötigt mindestens 2 Argumente. Es sind zwei Instanzen von `O\_graph\_node`, die erste ist der Start-Node, die zweite der Target-Node.

Danach werden in der Funktion 2 Arrays erstellt. Das eine ist der Stack und das andere das Set.

Ausserdem wird das Objekt `o\_backtracking` hier erstellt, denn eine Referenz davon wird vom Algorithmus verwendet, um das Backtracking zu ermöglichen.

Die zweite Funktion ist jene, welche rekursiv aufgerufen wird.

```

39  var f_o_traversal_result__traversal_dfs = function(
40    o_graph_node_start,
41    o_graph_node_target,
42    a_s_side = [
43      "down",
44      "up",
45      "left",
46      "right"
47    ]
48  ){
49    var a_o_graph_node_stack_frontier =[ o_graph_node_start ];
50    var a_o_graph_node_set_explored = [ o_graph_node_start ];
51    var o_backtracking = {}
52    f_traversal_dfs_recursive(
53      a_o_graph_node_stack_frontier,
54      a_o_graph_node_set_explored,
55      o_graph_node_target,
56      o_backtracking,
57      a_s_side
58    );
59
60    return new O_traversal_result(
61      o_graph_node_start,
62      o_graph_node_target,
63      a_o_graph_node_stack_frontier,
64      a_o_graph_node_set_explored,
65      o_backtracking,
66    );
67  }
68  class O_traversal_result{
69    constructor(
70      o_graph_node_start,
71      o_graph_node_target,
72      a_o_graph_node_stack_frontier,
73      a_o_graph_node_set_explored,
74      o_backtracking
75    ){
76      this.o_graph_node_start = o_graph_node_start
77      this.o_graph_node_target = o_graph_node_target
78      this.a_o_graph_node_stack_frontier = a_o_graph_node_stack_frontier
79      this.a_o_graph_node_set_explored = a_o_graph_node_set_explored
80      this.o_backtracking = o_backtracking
81    }
82  }
83

```

Parameter `a\_o\_graph\_node\_stack\_frontier` ist das Stack array, es muss beim ersten Funktionsaufruf mit dem Target-Node gefüllt sein.

Parameter `a\_o\_graph\_node\_set\_explored` ist das set array, es muss beim ersten funktionsaufruf mit dem Target-Node gefüllt sein.

Parameter `o\_backtracking` ist eine Referenz zu einem Objekt, das für das Backtracking verwendet wird.

Parameter `o\_graph\_node\_\_target` ist eine Referenz zur Instanz von `O\_graph\_node` welche den Target-Node enthält.

Parameter `a\_s\_side` ist ein array mit strings, welche jeweils eine Seite repräsentieren , also "up", "down", "left", "right". Die reihenfolge dieser strings, in Kombination mit der beschaffenheit des labyrinth und der position von start und Target-Node, kann einen einfluss auf die effizienz des algorithmus haben.

Zuerst wird geprüft, ob das Stack Array leer ist (Zeile 8) , ist dies der Fall, gibt es nur 2 Möglichkeiten. Entweder wurde der Target-Node nicht gefunden, es gibt also keinen Pfad zum Target-Node, oder beim ersten Aufruf der Funktion wurde ein leeres Array übergeben. Ist das stack array also leer, wird ein `console.log` gemacht (Zeile 9) und mit `return` aus der rekursion ausgebrochen(Zeile 10).

Als zweiten Prozess wird die letzte Instanz von `o\_graph\_node` aus dem stack array entfernt (Zeile 12). Danach wird geprüft (Zeile 13) ob diese instanz dieselbe ist wie die gesuchte `o\_graph\_node\_\_target` instanz, ist dies der fall, wurde erfolgreich ein pfad gefunden, das set array enthält nun alle wichtigen nodes welche teil vom pfad sind. Es wird ein `console.log` gemacht (Zeile 14) und mit `return` aus der rekursion ausgebrochen(Zeile 15).

Falls die funktion weiterläuft, wird durch das array mit den seiten "up", "down", "right" oder "left" durch iteriert (Zeile 18-30) und immer wieder derselbe prozess durchgeführt. Der prozess verhält sich folgendermassen:

Es wird geprüft, ob der Node überhaupt eine gültige Kante/Edge zu der gewissen Seite hat (Zeile 20). Falls nein geht die iteration direkt weiter.

Nun wird geprüft ob der verbundene node noch nicht im set array enthalten ist (Zeile 21), ist dies der fall, wird der über eine kante verbundene node in beiden arrays - stack array und set array - via `push` funktion am ende hinzugefügt (Zeile 22 und 23).

Nun werden die x und y Koordinaten des aktuell verbundenen Node's genommen und als property name für das Objekt `o\_backtracking` verwendet (Zeile 24-26). So ist es uns später nach Durchführung des Algorithmus möglich, den Pfad vom Target- zum Start-Node zu finden.

Schlussendlich ruft sich die Funktion selbst auf (Zeile 30), was zur Rekursion führt. Dies geschieht nun so lange, bis einer der anfangs erwähnten Fälle (Zeile 8 und 13) eingetroffen sind.

## Backtracking

```
1 var f_a_o_graph_node_path = function(
2     o_graph_node_target,
3     o_backtracking,
4 ){
5     debugger
6     var o_graph_node = o_graph_node_target
7     var a_o_graph_node_path = []
8     while(o_graph_node != undefined){
9         a_o_graph_node_path.push(o_graph_node)
10        o_graph_node = o_backtracking[
11            `${o_graph_node.o_object_2d.n_x}|${o_graph_node.o_object_2d.n_y}`
12        ]
13    }
14    return a_o_graph_node_path;
15 }
```

Während des Algorithmus wird auf dem Backtracking der aktuelle Node gespeichert und zwar mit dem property name, der die Koordinaten des verbundenen Nodes enthält. Dies ermöglicht uns schlussendlich, einen Pfad vom Target-Node zum Start-Node zu rekonstruieren. Die Funktion `f\_a\_o\_graph\_node\_path` macht genau das. Sie gibt ein Array zurück mit allen Nodes, die im Pfad enthalten sind. Achtung, dieses Array ist in der umgekehrten Reihenfolge, also vom Target-Node zum Start-Node, falls das Array umgedreht werden soll, könnte es ganz einfach mit der Javascript Funktion `Array.reverse` umgedreht werden.

## Resultate

Ich habe einige Tests durchgeführt und dabei sowohl die DFS als auch die BFS Funktion verwendet.



## Breadth first search (BFS) vs. Depth first search (DFS)

DFS sucht zuerst immer nur den Pfad ab, welcher nur in eine Richtung geht, wessen Nodes also immer nur mit einem weiteren Node verbunden sind. Je nach Konfiguration der Anordnung der abgesuchten Richtungen "left", "up", "right", "down", dauert dies länger oder kürzer.

BFS hingegen sucht alle möglichen Pfade immer parallel ab. Dadurch dauert die Suche immer relativ lange, jedoch kann nach der Suche garantiert werden, dass der kürzeste Pfad gefunden wurde.

## Der DFS- und BFS Algorithmus animiert.

Ich habe den algorithmus mit folgender reihenfolge der seiten [ "down", "up", "left", "right" ] laufen lassen und animiert. Das animierte Resultat als Video darf gerne betrachtet werden, der link befindet sich zuunterst in diesem Dokument.

Schlussendlich habe ich eine interaktive Version gemacht, bei der via Tastatur-Eingaben mit den Tasten 'a', 'o', 's', 'r' die Parameter verändert werden können. Die interaktive Version darf gerne hier ausprobiert werden.

<http://labyrinth.11235.ch/client.html>

## Nutzen des BFS oder A\* Algorithmus

Der BFS wird immer den schnellsten Weg vom Start-Node zum Target-Node finden, jedoch ist er sehr zeitaufwändig, da er immer alle möglichen Pfade abtestet. Der A\* Algorithmus wird auch jedesmal den kürzesten Pfad zwischen Start-Node und Target-Node finden, jedoch ist er effizienter als der BFS, da er anhand von Kostenberechnungen schlauer entscheiden kann, welche Nodes er entdecken soll.

## Links

### Repository

[https://github.com/jonasfrey/denojs\\_code/tree/main/labyrinth](https://github.com/jonasfrey/denojs_code/tree/main/labyrinth)

### Interactive Demo

<https://labyrinth.11235.ch/client.html>

### Videos

#### Depth First Search (DFS) - Algorithm

<https://www.youtube.com/watch?v=XAWtoNXtCBk>

#### Breadth First Search (BFS) - Algorithm

<https://www.youtube.com/watch?v=IZGsWBvpuCM>