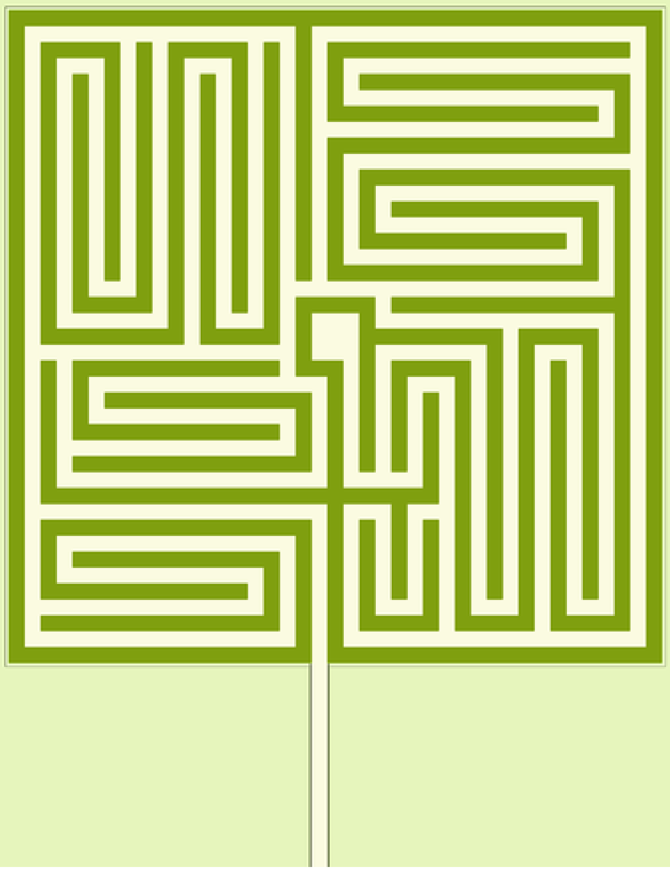
Hier der Arbeitsauftrag für die nächsten beiden Wochen!
Ich hab die Präsentation mit roten Anmerkungen versehen, so sollte es eigentlich auch im Selbststudium verständlich sein.
Begleitend könnt ihr im Buch Kapitel III-4 lesen (S. 108 bis 112)
Dijkstra solltet ihr euch zumindest gründlich durchlesen, den besprechen wir dann noch gemeinsam.

4 Tiefensuche



Finde den Weg durch die folgenden Labyrinthe!

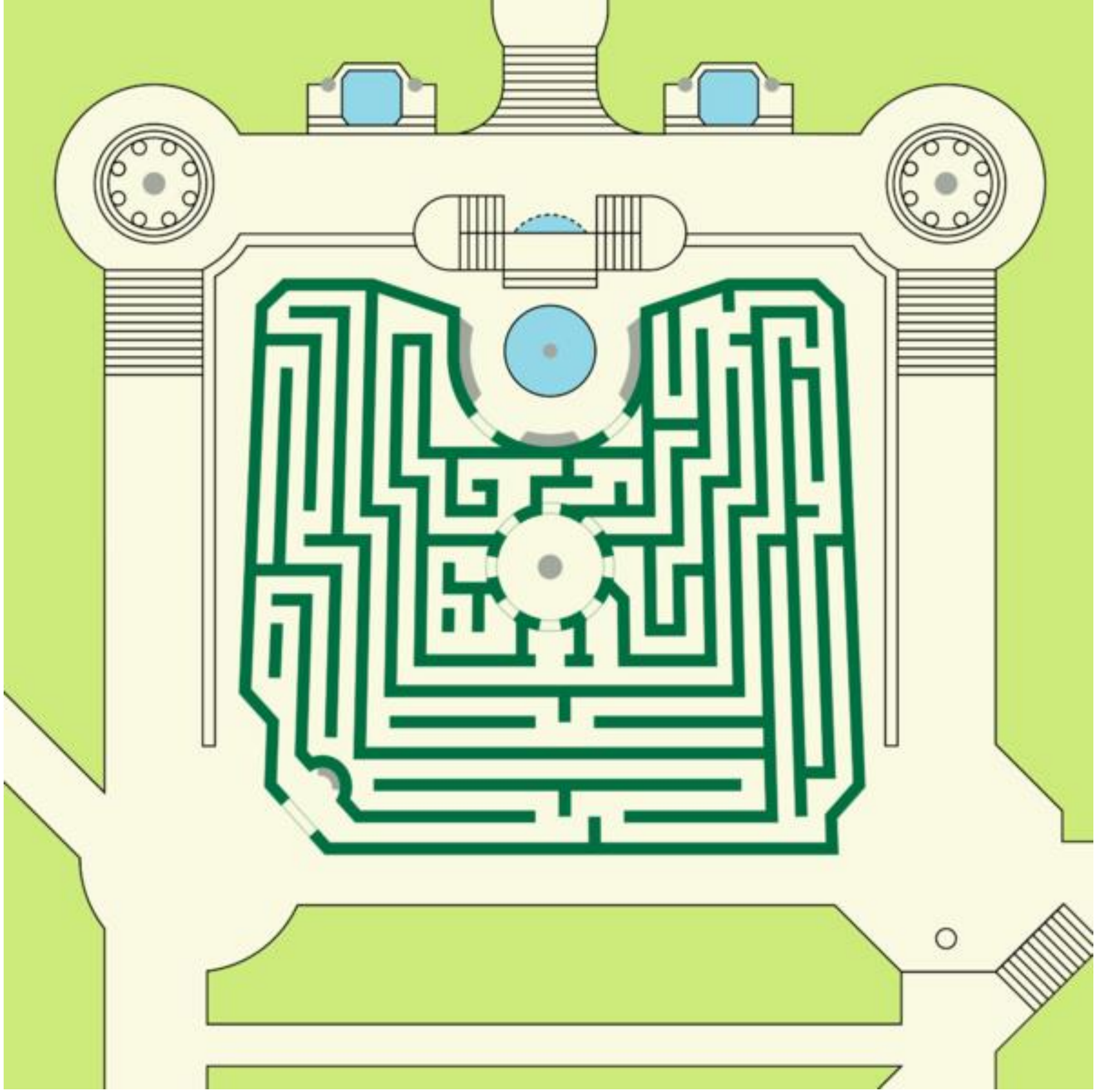
Hier:
Grüningen (1576)

Das eben geschieht den Menschen, die in einem Irrgarten hastig werden: Eben die Eile führt immer tiefer in die Irre. (Seneca)

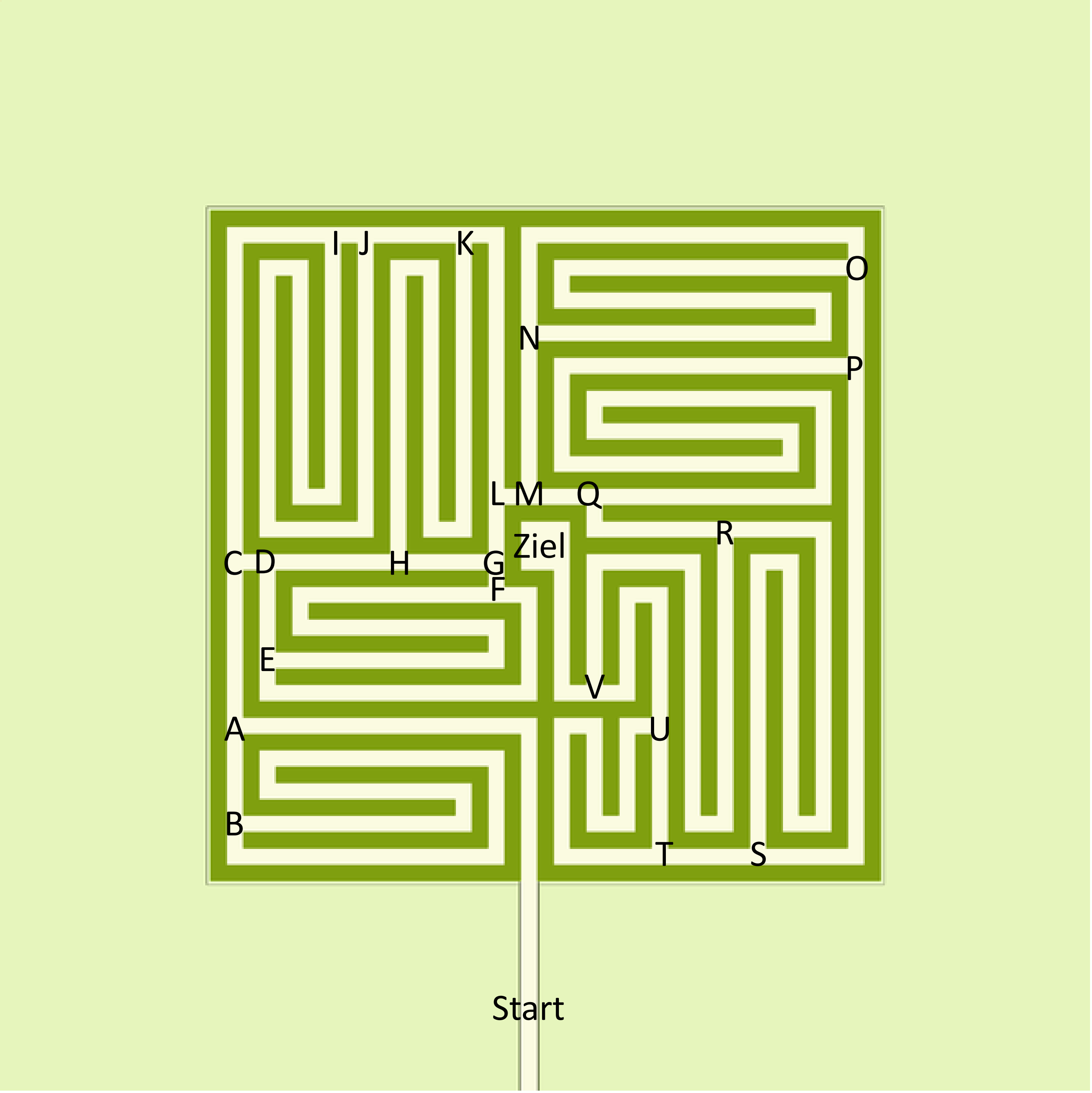
Wirbellabyrinth: d'Argenville
Gleich in der Mitte, aber schwierig, wieder herauszufinden.

Alles auf Erden lässt sich finden, wenn man nur zu suchen sich nicht verdrießen lässt. (Philemon von Syrakus)

Frederiksoord

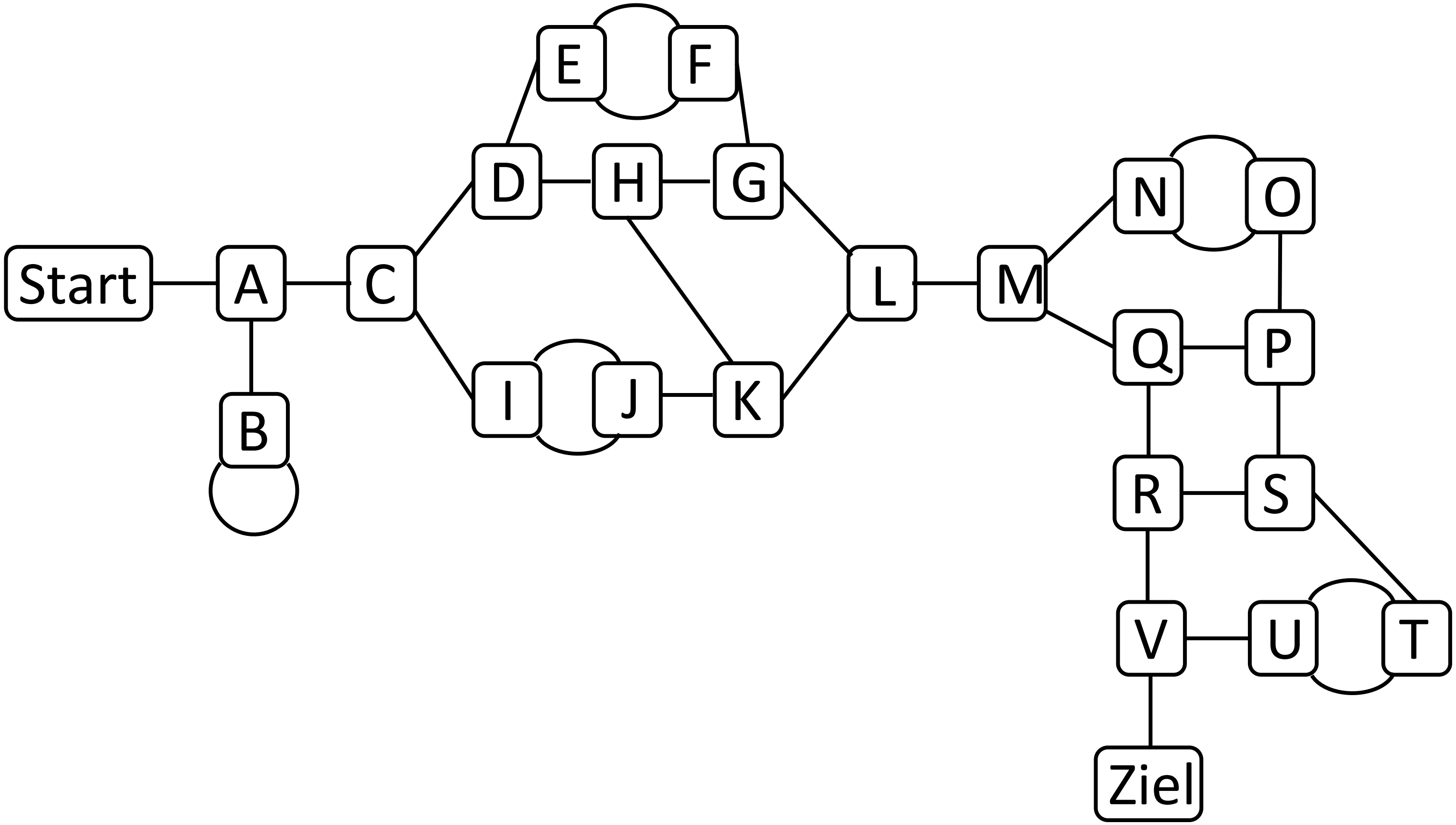


Park des Labyrinths von Horta (Barcelona)
Eingang: links unten
Mitte: Eros-Skulptur
Ausgang: Mitte oben links



Benennt man alle Kreuzungen, so kann man das Labyrinth auch als Graph darstellen.

Stelle dieses Labyrinth vereinfacht als Graph dar! Lösung nächste Folie!


Start
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
Ziel

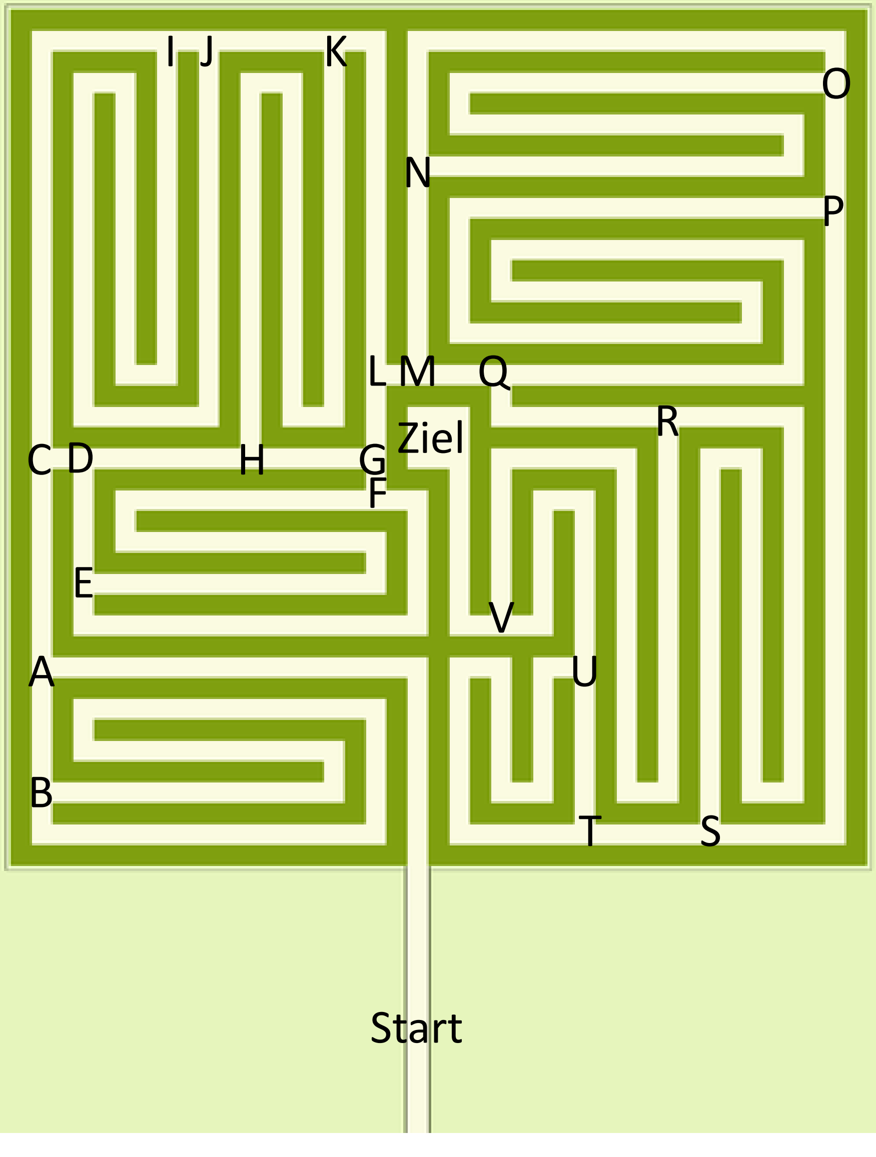
Listen lassen sich einfach der Reihe nach durchlaufen. Bei Bäumen ist es schon schwieriger. Hier hat man bei Binärbäumen die drei Traversierungsstragien Preorder, Inorder und Postorder. Damit erreicht man jedes Element.
So etwas Ähnliches werden wir nun im Hinblick auf Graphen versuchen. Dies ist schwieriger, weil es keine eindeutig vorgegebene Richtung wie bei Listen (eins weiter) oder Bäumen (eins tiefer) gibt.
Aber natürlich gibt es auch hier Tricks und die lernt ihr jetzt….
Der Algorithmus, der quasi in jedem Abitur dran kommt, heißt Tiefensuche!
Lest ihn euch durch und versucht dann die folgenden Labyrinthe per Tiefensuche zu durchschreiten.
Man kann z.B. die Strategie "zuerst rechts" benutzen. Ihr könnt dabei auch schon mal darüber nachdenken, wie es bei der Implementierung dann umgesetzt werden könnte. Gibt es bei der Adjazenzmatrix rechts und links???
Ihr könnt auch mal nachschauen, was es mit dem Faden der Ariadne auf sich hat und ob der überhaupt nötig gewesen wäre…

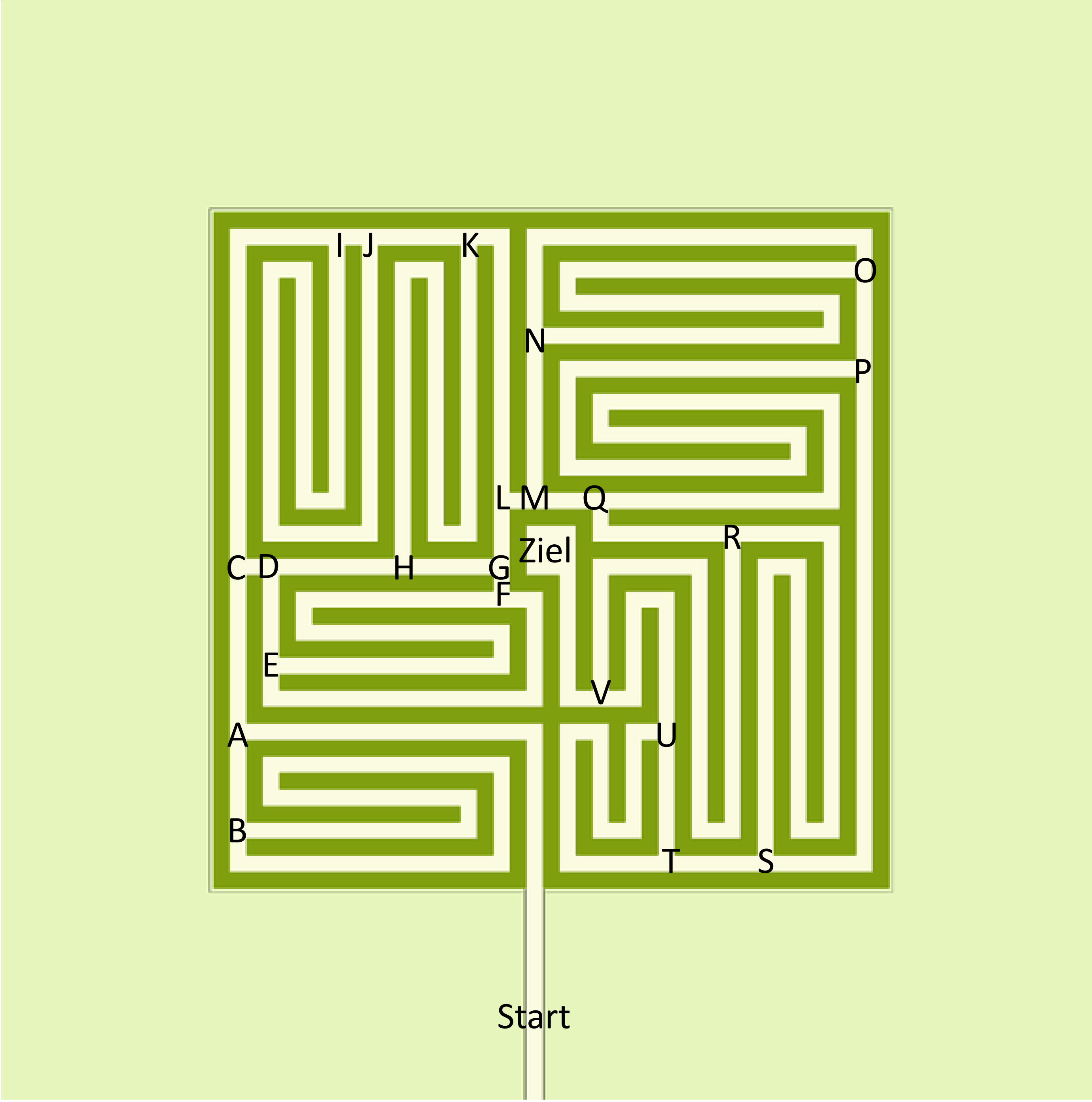
Tiefensuche:

DFS (Depth First Search)

Ausgehend von einem Startknoten sollen alle erreichbaren Knoten des Graphen mindestens einmal besucht werden. (Auch für die Suche eines bestimmten Knotens verwendbar.)

- Gehe vom Start zur ersten Weggabelung, wähle dort einen nicht besuchten Weg (z.B. rechts zuerst) und laufe diesen weiter.
- Wiederhole dies, bis eine Sackgasse erreicht ist oder du an eine Stelle kommst, an der du schon warst.
- Kehre zur letzten Gabelung zurück und wähle einen bisher unbesuchten Weg, usw.

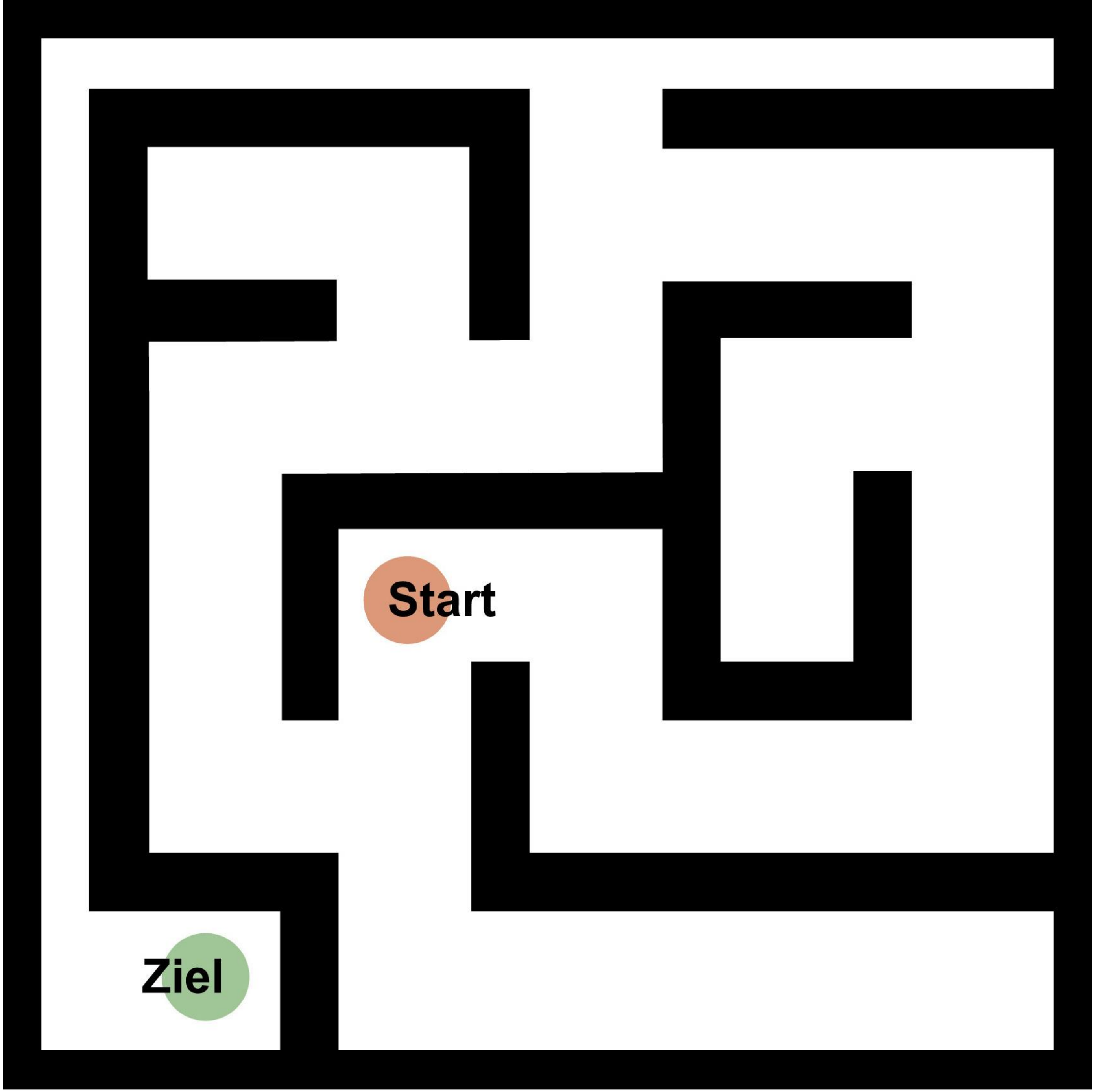

I
J
K
O
N
P
L
M
Q
R
Ziel
C
D
H
G
F
E
V
A
U
B
T
S
Start



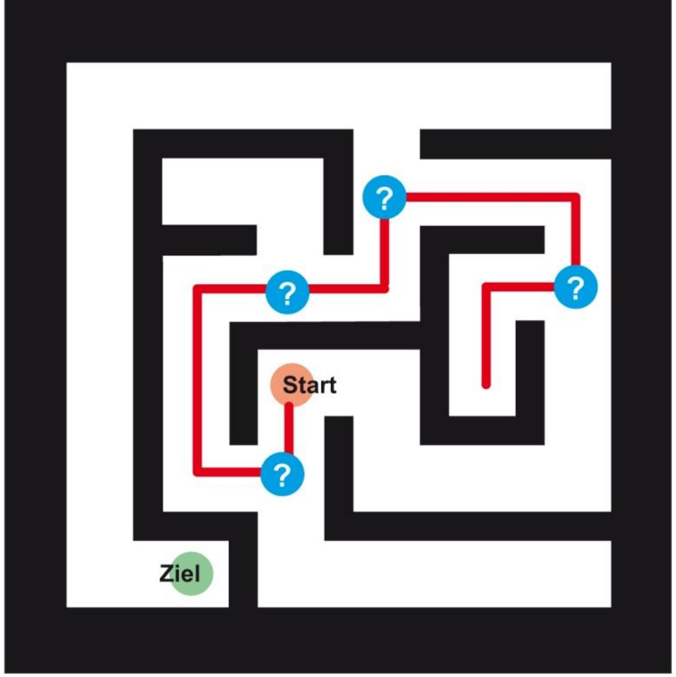
Reihenfolge:
Start
A, C, D, E, F, G, L,
M, Q, R, V, Ziel
(hier nicht fertig,
wir wollen ja alle
Knoten ablaufen!)
U, T, S, P, O, N
zurück, zurück,…
unbesuchter Weg
erst bei L
K, J, I
zurück, zurück,…
H,
zurück, zurück,…
B

Warum heißt
dieses Vorgehen
wohl Tiefensuche?

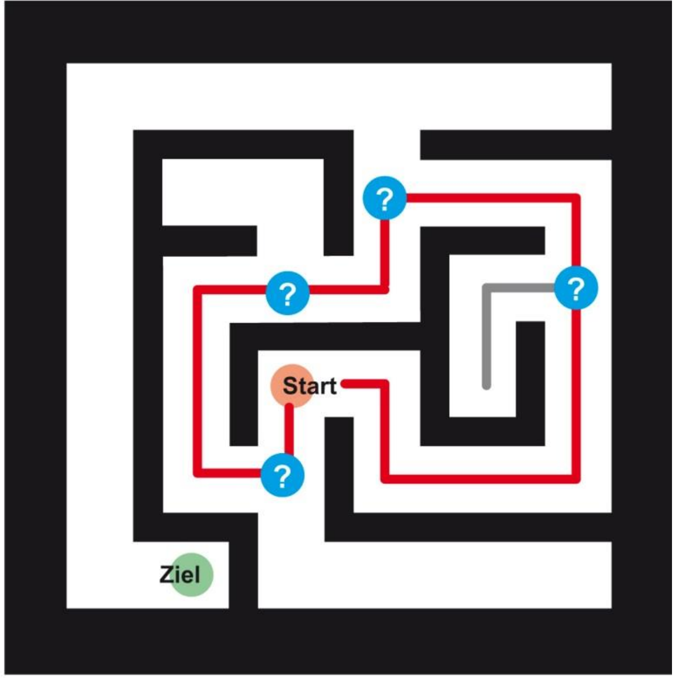
Es heißt Tiefensuche, weil man sehr schnell in die Tiefe gelangt.
Es gibt auch Breitensuche (Siehe Buch S. 115 ff Exkursion. Diese ist nicht Stoff für die Schulaufgabe oder Abi, aber durchaus interessant zu lesen. Würde ich aber erst machen, wenn ich mit der Tiefensuche fertig bin!)
Da man mit der Tiefensuche alle Knoten erwischt, kann sie auch dazu benutzt werden, **einen bestimmten Knoten (Ziel)** zu finden.
Vergebt im nächsten Labyrinth Buchstaben für die Knoten und sucht das Ziel mittels Tiefensuche. Stellt auch dieses Labyrinth als Graphen dar!


Start
Ziel

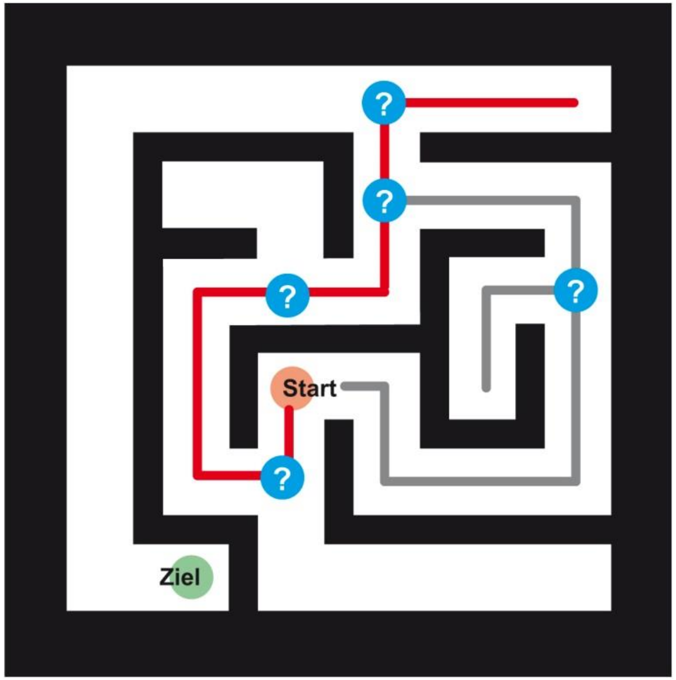
1



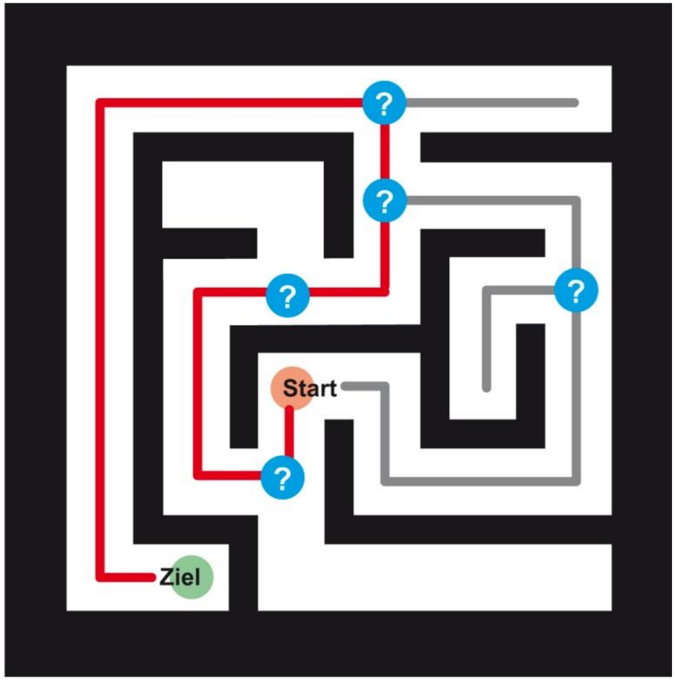
2

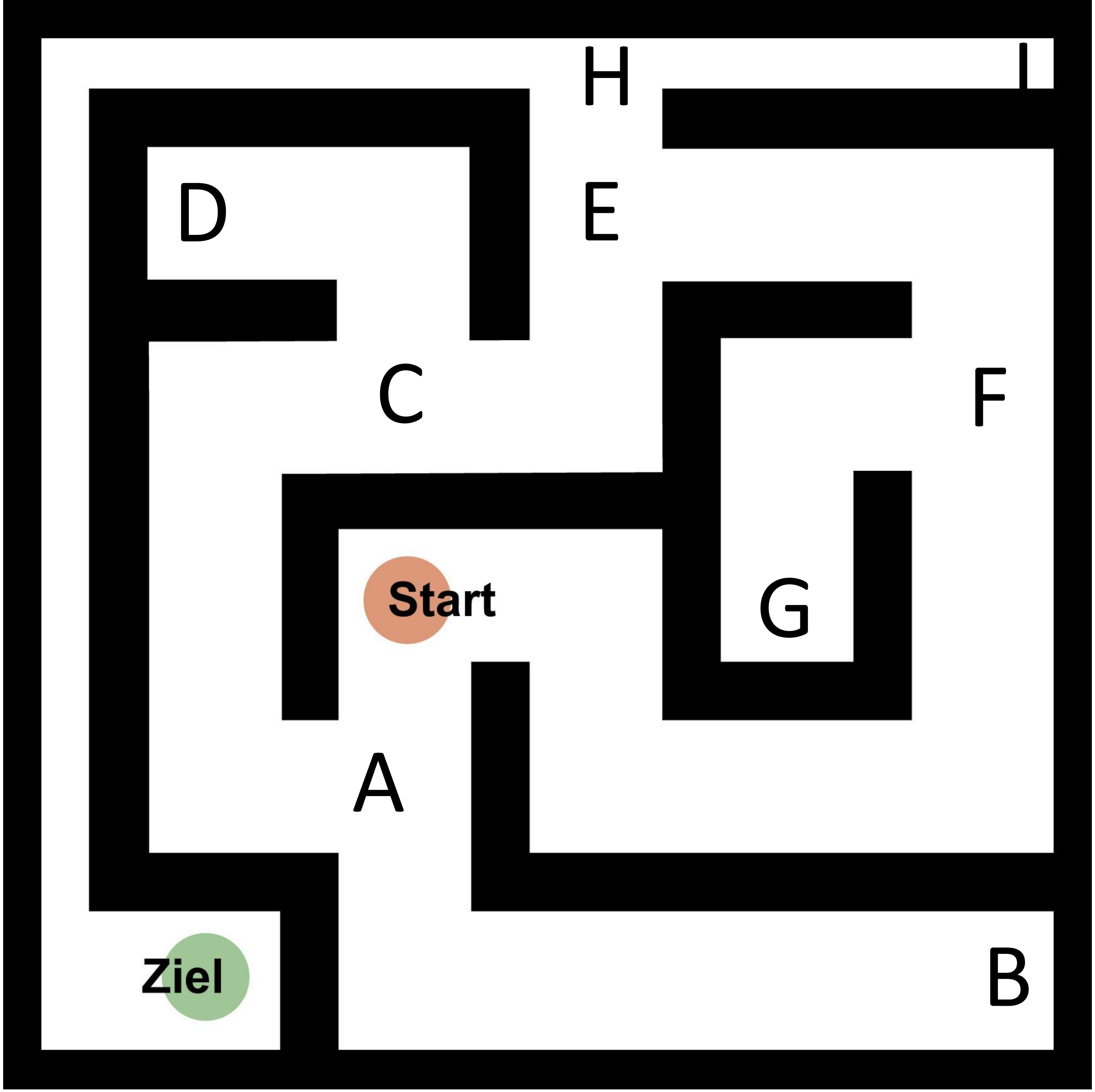


3

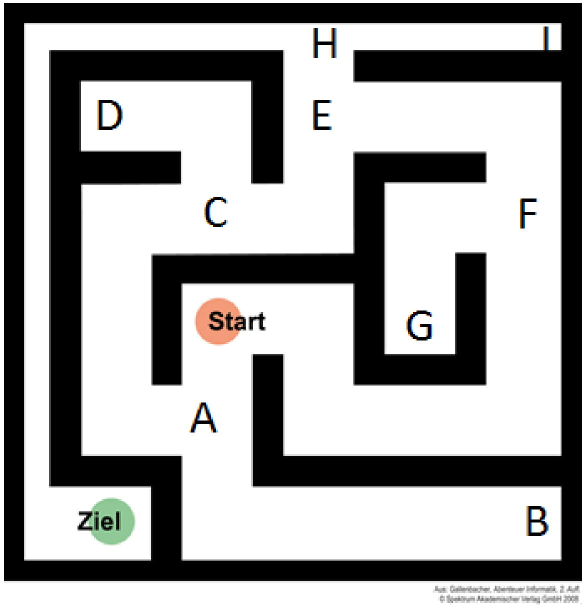


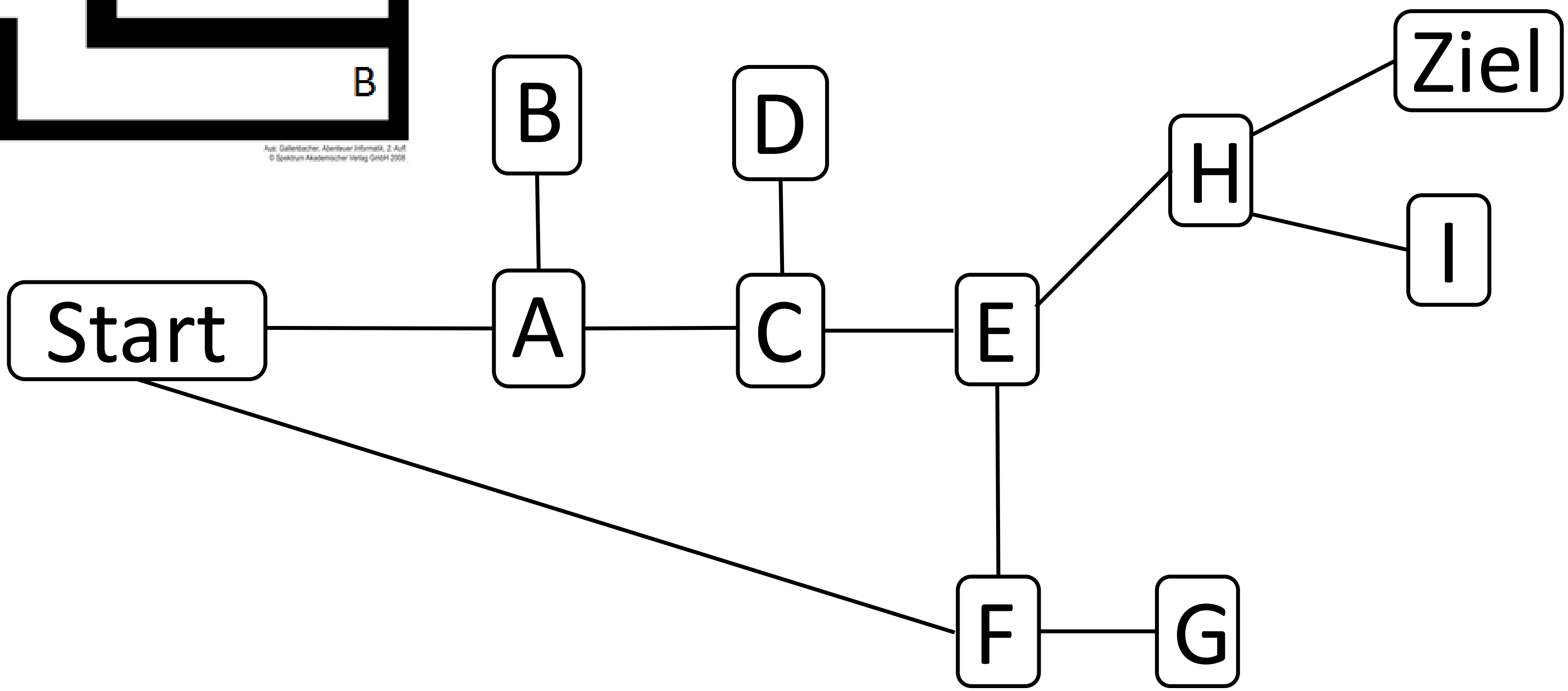
4





Aus: Gallenbacher, *Abenteuer Informatik, 2. Aufl.*





Gebt für diesen Graphen die Adjazenzmatrix an und überlegt, wie wir die Tiefensuche implementieren können! Man muss sich irgendwie durch die Adjazenzmatrix hangeln! Und denkt dran… Rekursion macht das Informatikerleben leichter….;-)

	S	A	B	C	D	E	F	G	H	I	Z
S		x					x				
A	x		x	x							
B		x									
C		x			x	x					
D				x							
E				x			x		x		
F	x					x		x			
G							x				
H						x				x	x
I									x		
Z									x		

Bei der Tiefensuche könnte man sich an der Reihenfolge in der Adjazenzmatrix orientieren, rechts und links gibt es ja hier nicht. Man könnte sich es aber so vorstellen, dass der in der Knotenliste zuerst kommende der Rechteste ist. (Bei der Adjazenzmatrix sind im Übrigen Spalten- und Reihenvertauschungen möglich!) Beginnt man bei S, wäre der nächste besuchte Knoten A, bei geht es hier weiter mit B, von B kommt man nur zu A, also zurück, von A nach C, C nach D, usw.

Die Knoten brauchen bei unserer Implementierung ein weiteres Attribut. Wofür? Was muss hier gespeichert werden?

Die Knoten bekommen ein Attribut boolean besucht, darin wird festgehalten, ob der Knoten schon besucht wurde.
Zu Beginn der Tiefensuche werden alle Knoten auf unbesucht gesetzt (Durchlauf im Knotenfeld).
Dann wird die Tiefensuche mit einem Startknoten gestartet. Die aufgerufene Methode ist dann rekursiv, d.h. um das wiederholte Vorgehen braucht man sich nicht groß zu kümmern, man geht einfach alle unbesuchten Nachbarknoten des Startknotens durch und ruft hier die Tiefensuche wieder auf.
Die Knoten dürfen aber jetzt nicht wieder alle auf unbesucht gesetzt werden, sonst würde man ja nie fertig. Dieses Unbesuchtsetzen und Starten muss daher in eine eigene Methode vorgelagert werden.
Bleibt die Frage, wie man die Nachbarknoten findet…..?

Genau, über die Adjazenzmatrix! Es geht von Knoten X nach Knoten Y, wenn für die dazugehörigen Indizes (z.B. 3 und 7) gilt:
adjmat[3][7] == true oder adjmat[3][7] >0

Formuliere in Worten den Tiefensuche-Algorithmus!

Algorithmus Tiefensuche

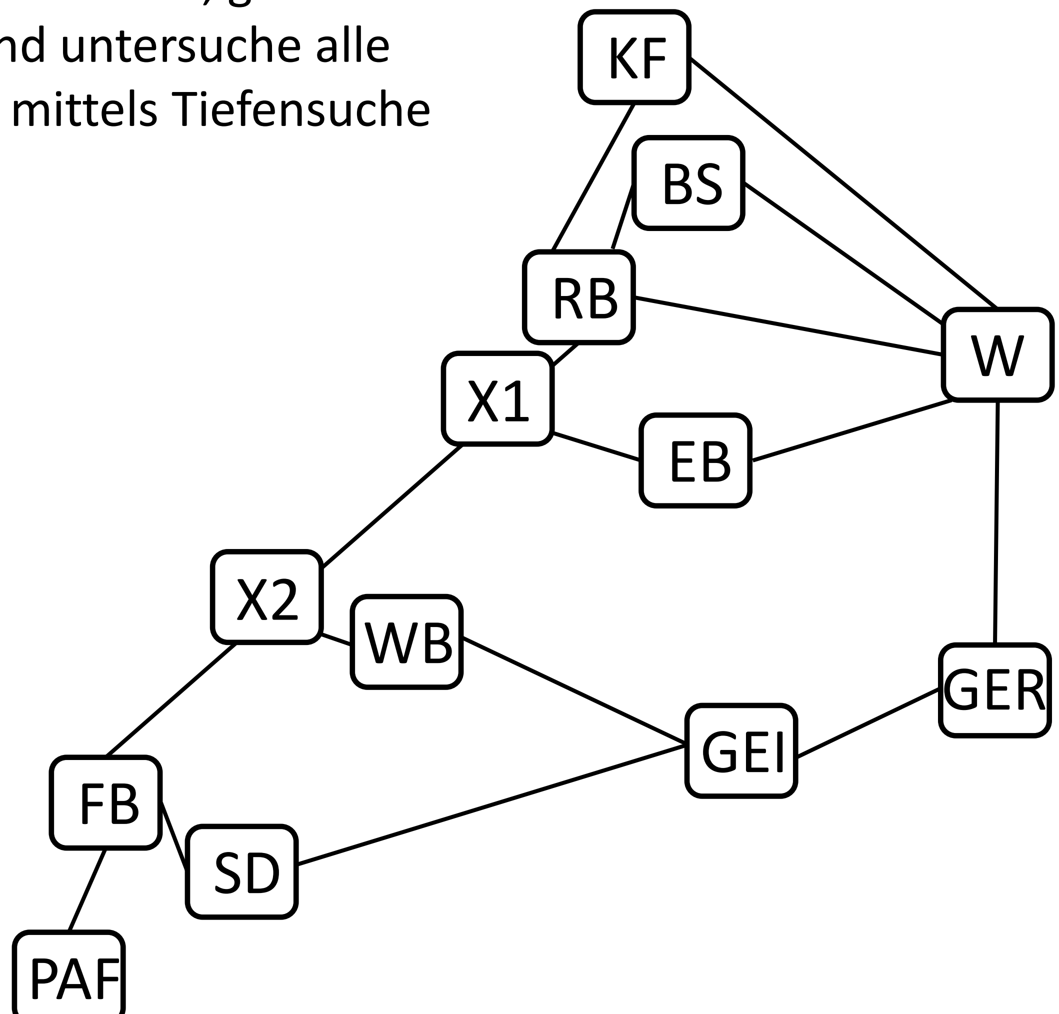
1. Markiere alle Knoten als unbesucht.
2. Wähle beliebigen Startknoten s als aktuellen Knoten und verfahre wie in Punkt 3 beschrieben.
3. Markiere aktuellen Knoten als besucht, gib die entsprechenden Daten aus und untersuche alle unbesuchten Nachbarknoten mittels Tiefensuche (Rekursion).

Beispiel S.108 ff

http://www.youtube.com/watch?v=S2Fnz1atRBY

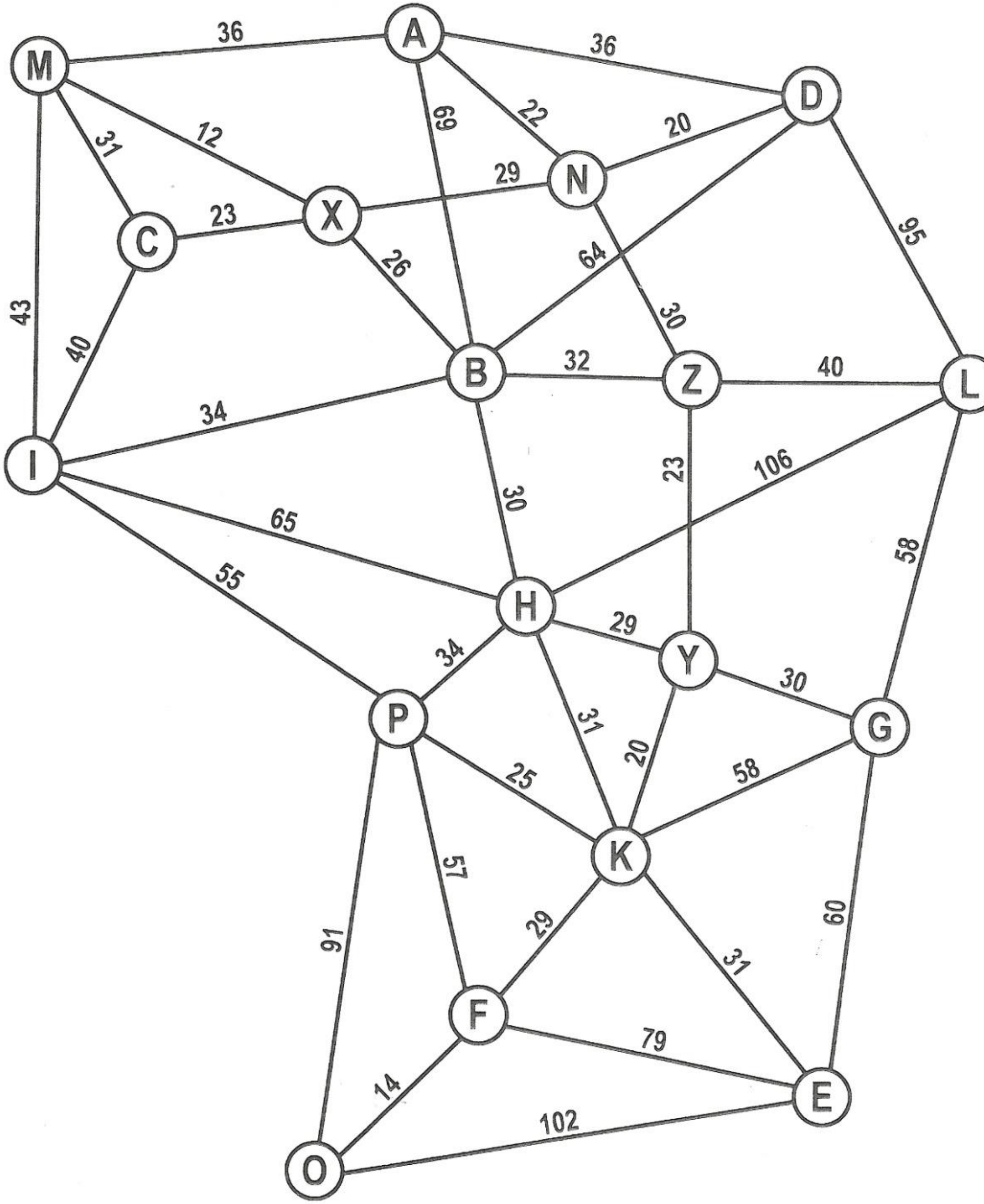
Algorithmus Tiefensuche

1. Markiere alle Knoten als unbesucht.
2. Wähle beliebigen Startknoten s als aktuellen Knoten und verfahre wie in Punkt 3 beschrieben.
3. Markiere aktuellen Knoten als besucht, gib die entsprechenden Daten aus und untersuche alle unbesuchten Nachbarknoten mittels Tiefensuche (Rekursion).



Algorithmus Tiefensuche

1. Markiere alle Knoten als unbesucht.
2. Wähle beliebigen Startknoten s als aktuellen Knoten und verfahre wie in Punkt 3 beschrieben.
3. Markiere aktuellen Knoten als besucht, gib die entsprechenden Daten aus und untersuche alle unbesuchten Nachbarknoten mittels Tiefensuche (Rekursion).



Implementierung des Tiefensuche-Algorithmus

Klassenkarte

Knoten
Datenelement inhalt **boolean markierung**
Knoten(*Datenelement* inh) inhaltGeben() **markierungSetzen(boolean m)** **markierungGeben()**

Erweitere eine der bereits fertigen Graphenumsetzungen um die Tiefensuche!

In der Klasse Graph:

Methode tiefensuche(int startNr)

Methode tiefensucheKnoten(int vIndex)

Ergänze die Klasse Knoten und die entsprechenden Methoden in Graph und Testablauf.

Tiefensuche-Vorlage

```java
public class Knoten{
    private Datenelement inhalt;
    private boolean markierung;

    public Knoten(Datenelement inh){
        inhalt = inh;
        markierung = false;
    }

    public Datenelement inhaltGeben(){
        return inhalt;
    }

    public void markierungSetzen(boolean m){
        markierung = m;
    }

    public boolean markierungGeben(){
        return markierung;
    }
}
```

```java
public void tiefensuche(int startNr){
    //alle Knoten auf unbesucht setzen
    for (int i=0;i<anzahl;i++){
        knoten[i].markierungSetzen(false);
    }
    System.out.println("Reihenfolge bei Tiefensuche:");
    tiefensucheKnoten(startNr);
}
```

```
public void tiefensucheKnoten(int vIndex){
    knoten[vIndex].markierungSetzen(true);
    System.out.println(knoten[vIndex].
                                inhaltGeben().datenGeben());
    for (int i=0;i<anzahl;i++){
        if (adjazenzmatrix[vIndex][i]&&
                                !knoten[i].markierungGeben()){
            tiefensucheKnoten(i);
        }
    }
}

public void orteDurchlaufen(int startNr){
    graph.tiefensuche(startNr);
  }
```

2 Aufgaben im Buch handeln von der Routensuche im ICE-Netz:
Bearbeite S. 107/16a, b, c = S.116/2a, S.116 /2b, 2c, …

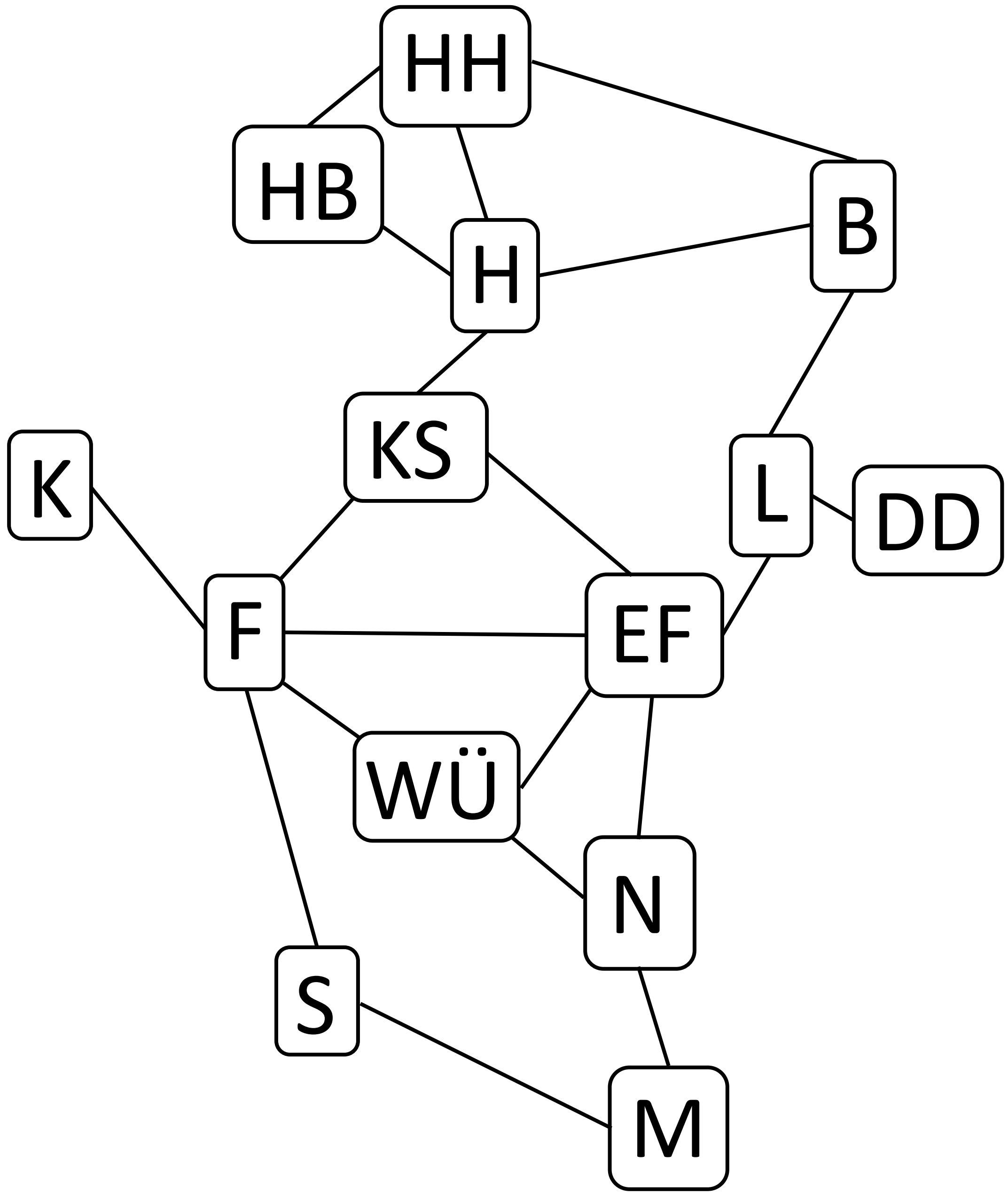
S. 107/16 ICE-Bahnhöfe

a) symmetrische Adjazenzmatrix, d.h. ungerichteter Graph

b) z.B. B – H – HH – B

   oder M – N – WÜ – F – S – M

S. 107 /16c) bzw. 113/2a)



2b)
Tiefensuche:
Rechts zuerst: M-N-EF-L-DD-B
Links zuerst: M-S-F-K-KS-H-HB-HH-B

Problem: Wege müssen evtl. zurückgefahren werden.

2c)
z.B. Wähle als nächsten Bahnhof denjenigen, der dem Ziel am nächsten ist.

2d) Algorithmus für ICE-Pfad in Pseudocode

- Gib Start- und Zielbahnhof ein.
- Deklariere ein Feld *pfad* und trage Startbahnhof als aktuellen Bahnhof in die erste Position ein.
- Wiederhole solange, bis aktueller Bahnhof = Zielbahnhof:
  - aktueller Bahnhof = Nachbarbahnhof, der dem Zielbahnhof am nächsten ist
  - trage aktuellen Bahnhof an der nächsten freien Position in das Feld *pfad* ein.
- Gib das Feld *pfad* zurück

Problem: Köln ist Sackgasse, aber näher an Berlin als Frankfurt. Köln – Berlin, Man kommt von Köln nicht weg, der Algorithmus erlaubt keine Umwege.
Liefert nicht sicher den kürzesten Weg, sondern den bei dem die jeweilige Etappe dem Ziel am nächsten kommt.

Ergänze in der Klasse

- Graph: routeSuchen(...)
  nachbarKnotenWaehlen(...)
- ICE_Routenplaner(...):
  routeSuchen(...)

In Graph:

```
public Knoten[] routeSuchen(int startNr, int zielNr){
    Knoten startKnoten = knotenliste[startNr];
    Knoten zielKnoten = knotenliste[zielNr];
    Knoten[] pfad = new Knoten[20];
    int index = 0;
    pfad[index]=startKnoten;
    index++;
    Knoten aktuellerKnoten = startKnoten;
    while (!aktuellerKnoten.equals(zielKnoten) && index<20){
        aktuellerKnoten = nachbarKnotenWaehlen(aktuellerKnoten,
                                                zielKnoten);
        pfad[index] = aktuellerKnoten;
        index++;
    }
    return pfad;
}
```

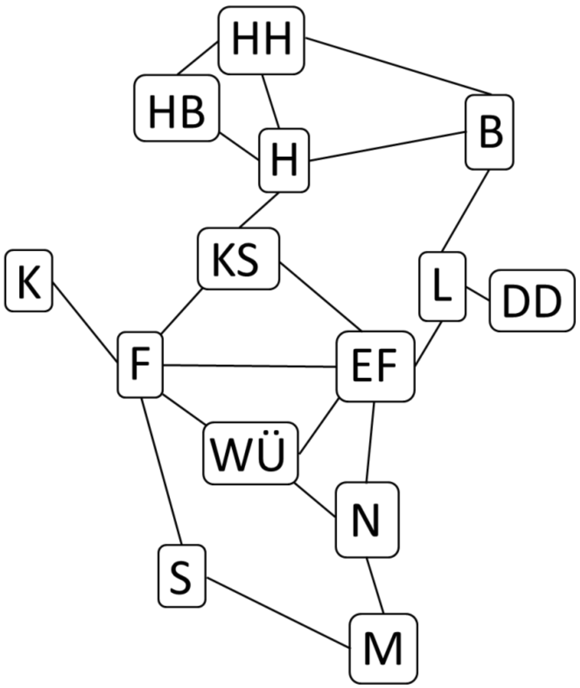
In Graph:

```
private Knoten nachbarKnotenWaehlen(Knoten aktuell, Knoten ziel){
    int aktIndex = knotenindexSuchen(aktuell);
    int zielIndex = knotenindexSuchen(ziel);
    int min =99999;
    Knoten nachfolger = aktuell;
    for (int i = 0; i < anzahl; i++){
        if (adjmatrix[aktIndex][i] && (entfernungsmatrix[zielIndex][i]) <
                                                                        min){
            min = entfernungsmatrix[zielIndex][i];
            nachfolger = knotenliste[i];
        }
    }
    return nachfolger;
}
```

In ICE_Routenplaner:

```
public void routeSuchen(String start, String ziel){
    Knoten[] pfad = graph.routeSuchen
                        (graph.knotenindexSuchen(start),
                         graph.knotenindexSuchen(ziel));
    System.out.println("Berechnete Route: ");
    int i=0;
    while (i<pfad.length && pfad[i]!= null){
        System.out.println(pfad[i].inhaltGeben().nameGeben());
        i++;
    }
}
```

Wh: ICE-Routenplaner



2 Adjazenzmatrizen
(Entfernung, ICE-Verbindung)

Idee:
Wähle als nächsten Bahnhof denjenigen, der dem Ziel am nächsten ist.

Greedy-Algorithmus

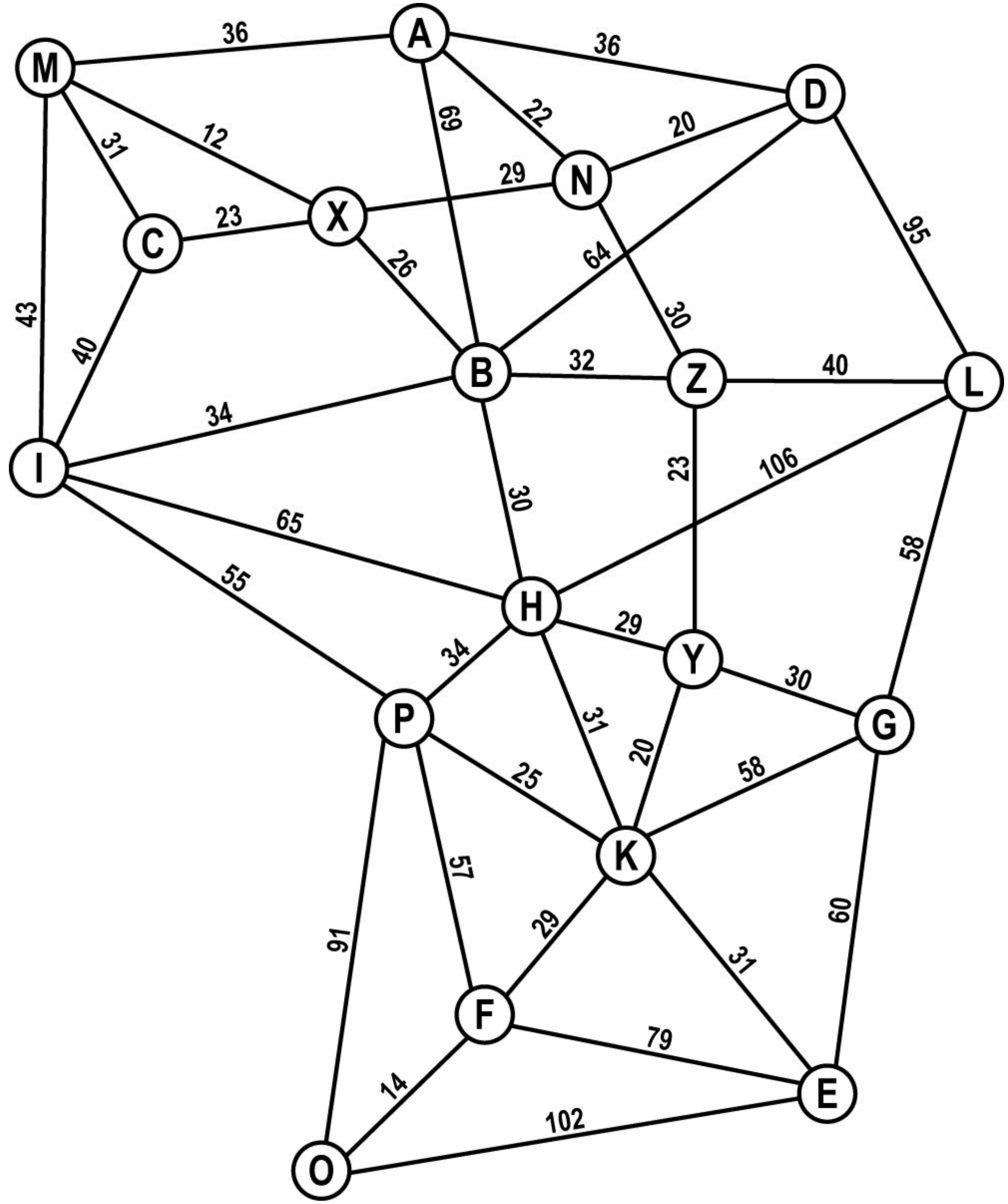
Liefert nicht zwingend den kürzesten Weg.

Alternativ:
Mittels Tiefensuche den kürzesten Weg ermitteln.

Die Tiefensuche liefert zwar einen Durchlauf durch den Graphen, sie eignet sich aber nicht den kürzesten Weg von A nach B zu finden, wie er z.B. vom Navi berechnet wird. Hierfür gibt es den sogenannten Dijkstra-Algorithmus.

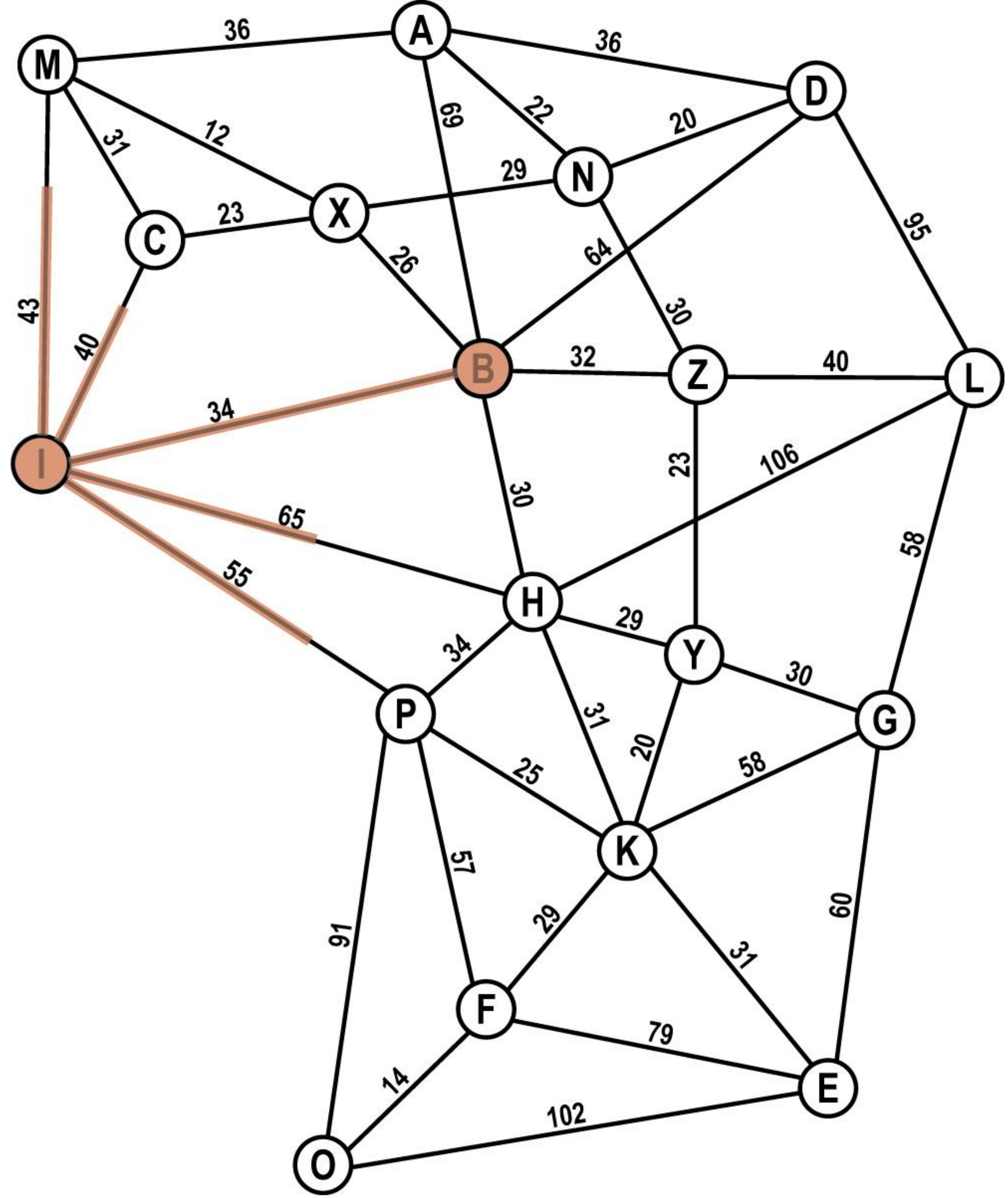
Der Dijkstra-Algorithmus

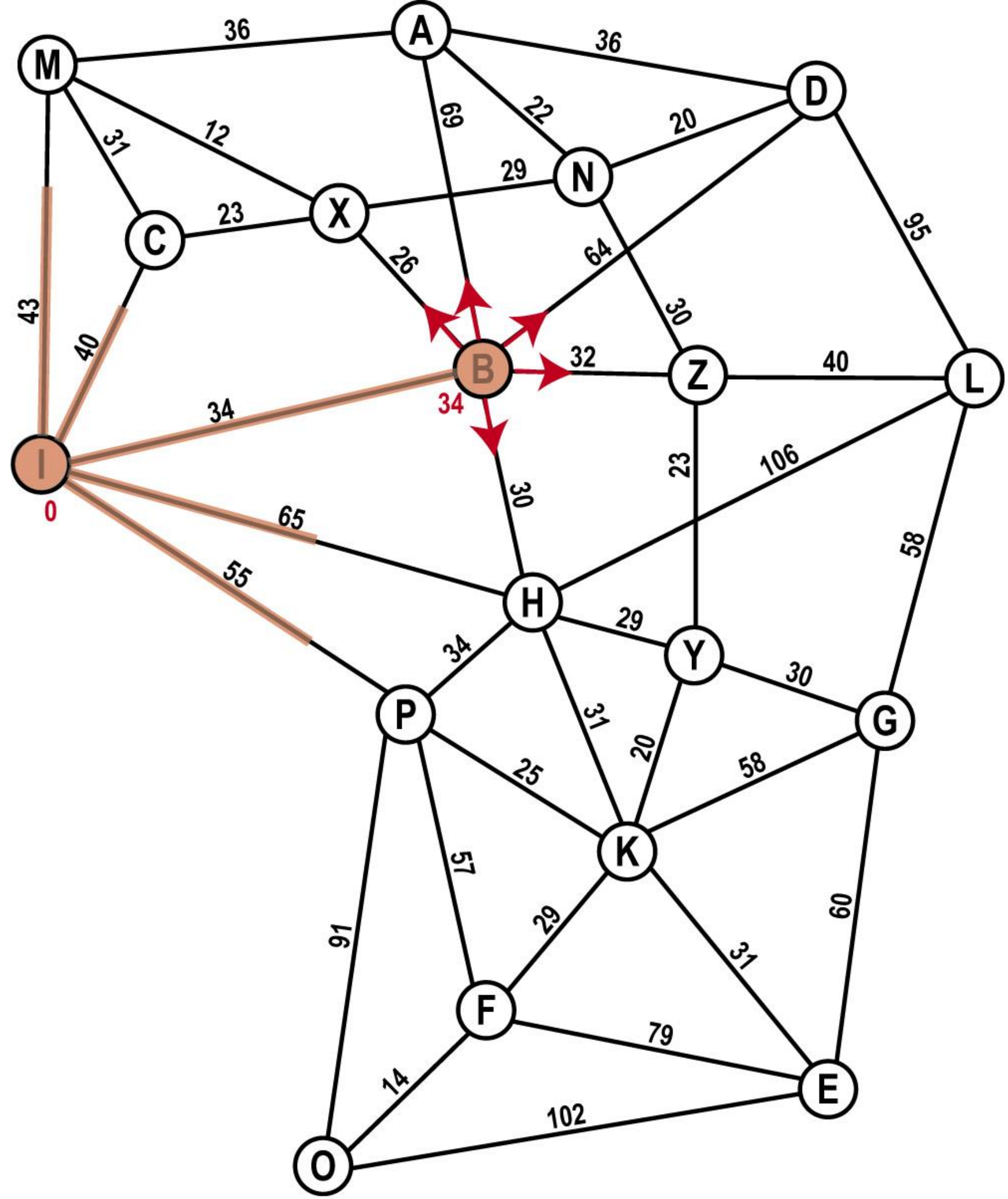
Am besten ihr druckt euch diesen Plan mal aus und macht mit!
Gesucht ist der kürzeste Weg von I nach O.
Man stellt sich vor, gleichzeitig beginnen gleich schnelle Karawanen von Ameisen von I weg in alle Richtungen zu krabbeln!
(Die ist in "Abenteuer Informatik" sehr schön erklärt!)



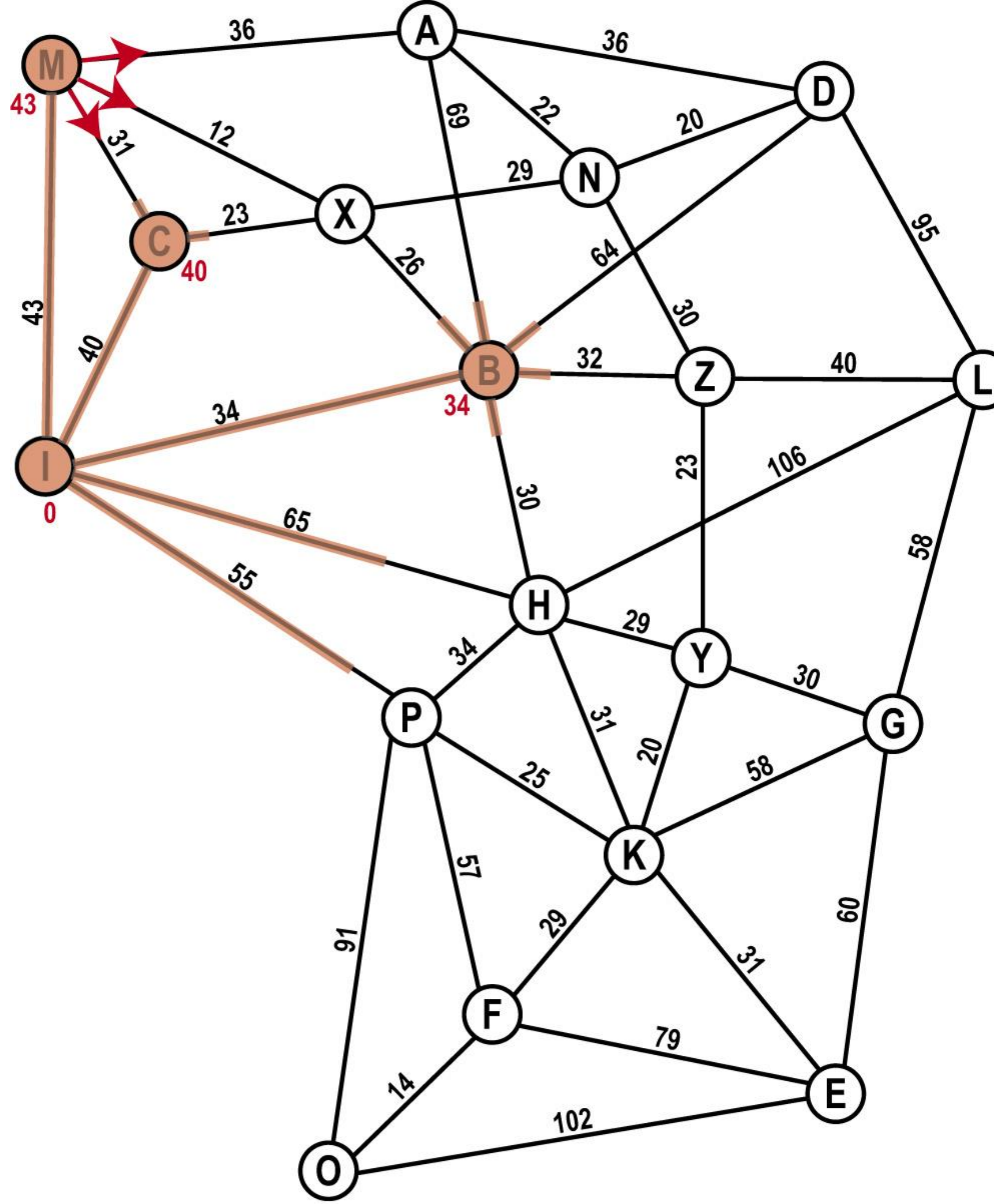
Aus: Gallenbacher, *Abenteuer Informatik*, 2. Aufl.

Von den unpassenden Längen nicht verwirren lassen!
Jedenfalls erreicht die Karawane Richtung B zuerst eine weitere Stadt, nämlich eben B. Weil B am nächsten an I liegt. Damit ist schon mal klar, dass es von I nach B keinen besseren Weg gibt, als den direkten. Man schreibt daher schon mal die 34 bei B an den Weg nach I. Besser geht es nicht!
Von dort aus geht es für die Ameisen wieder gleichzeitig in alle Richtungen weiter nur nicht zurück. Da waren sie ja schon!



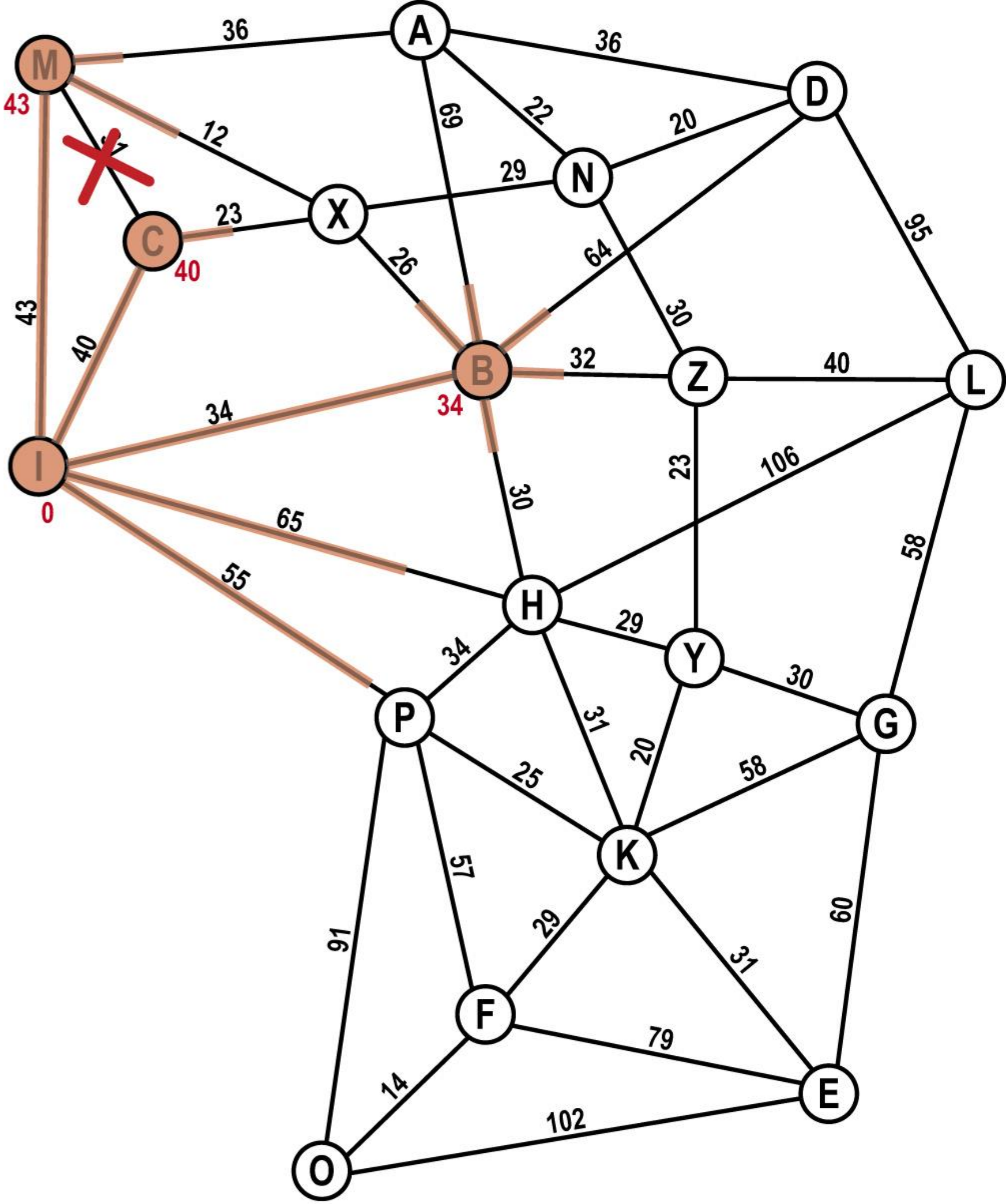

M
A
D
N
X
C
B
Z
L
I
H
Y
G
P
K
F
E
O
36
36
69
22
20
31
12
29
23
26
64
95
43
40
32
40
34
34
0
30
23
106
65
55
58
29
34
30
31
20
58
25
57
91
29
60
31
14
79
102

Die nächste Karawane erreicht dann die Stadt C (40 ist die nächstkleinste Entfernung), also 40 zu C schreiben, Ameisen weiter schicken.
Mit M anschließend das Gleiche.
Nun aber begegnen sich die Ameisen zwischen C und M. Was bedeutet das?



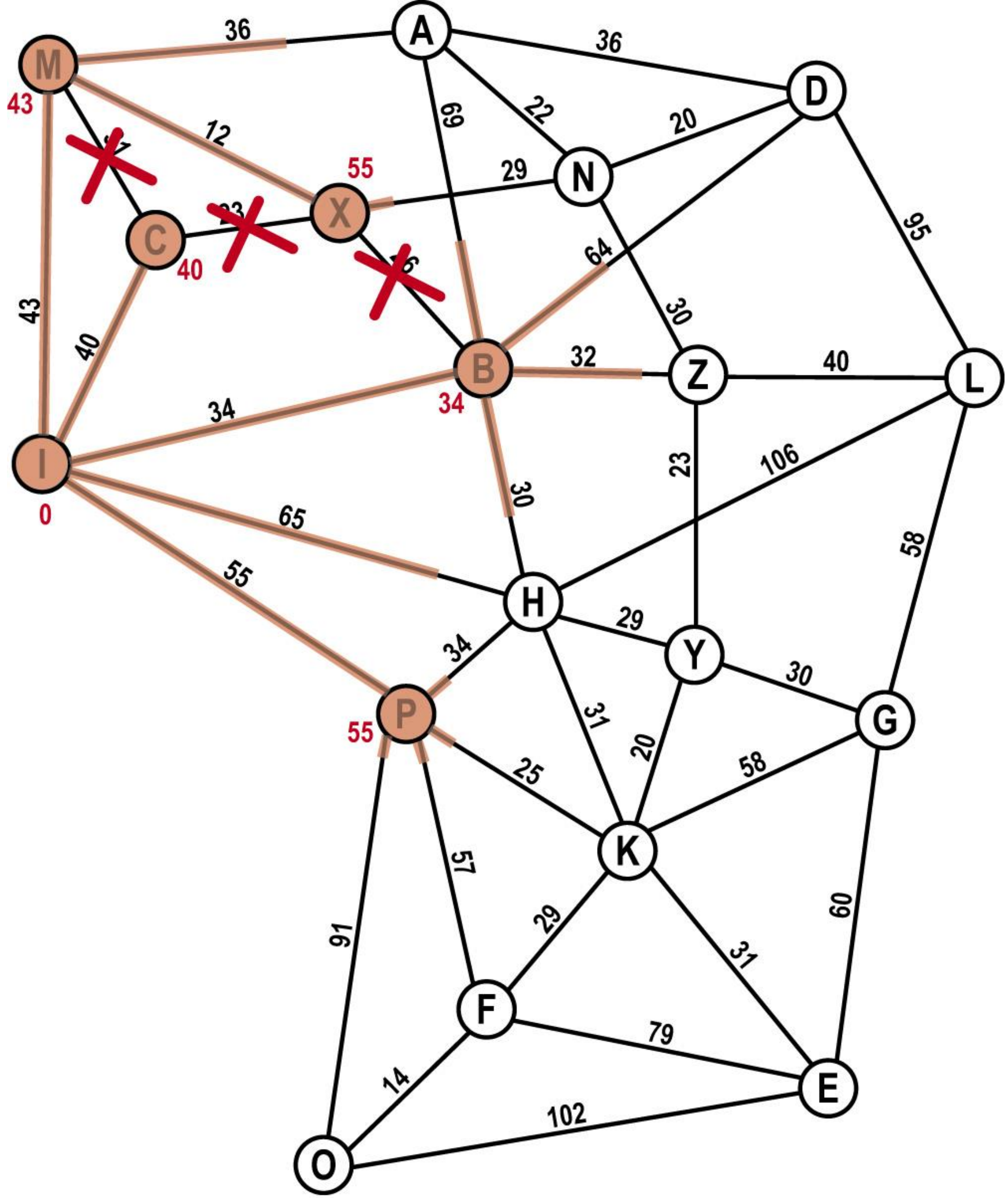
Aus: Gallenbacher, *Abenteuer Informatik*, 2. Au

C und M sind schon "erobert", also drehen die Ameisen um und verfolgen den Weg zwischen C und M nicht weiter. D.h. egal, wo ich von I aus hin will, den Weg zwischen M und C brauch ich nicht, ist beides anders besser erreichbar!



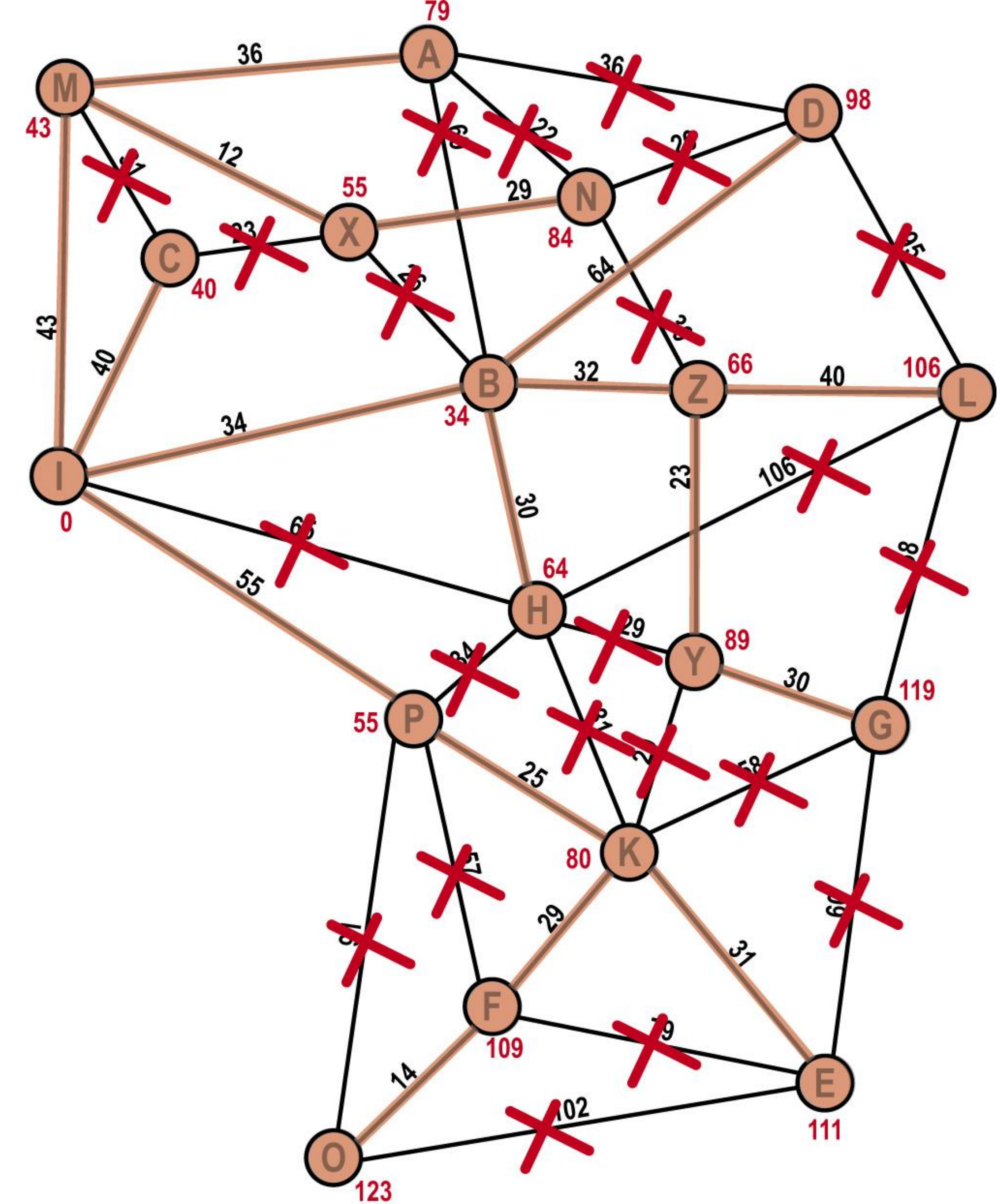
Aus: Gallenbacher, *Abenteuer Informatik*, 2. Aufl.

Das geht jetzt immer so weiter. Man muss nur überlegen mit welcher Strecke man immer weiter machen soll. Einfach immer mit der kürzesten, die noch nicht bearbeitet wurde. Zuerst sind alle von I ausgehenden im Topf, sobald B erreicht ist, kommt H mit 34 +30, Z mit 34+32, … dazu. Das fangen wir gleich noch mal systematisch von vorne an!

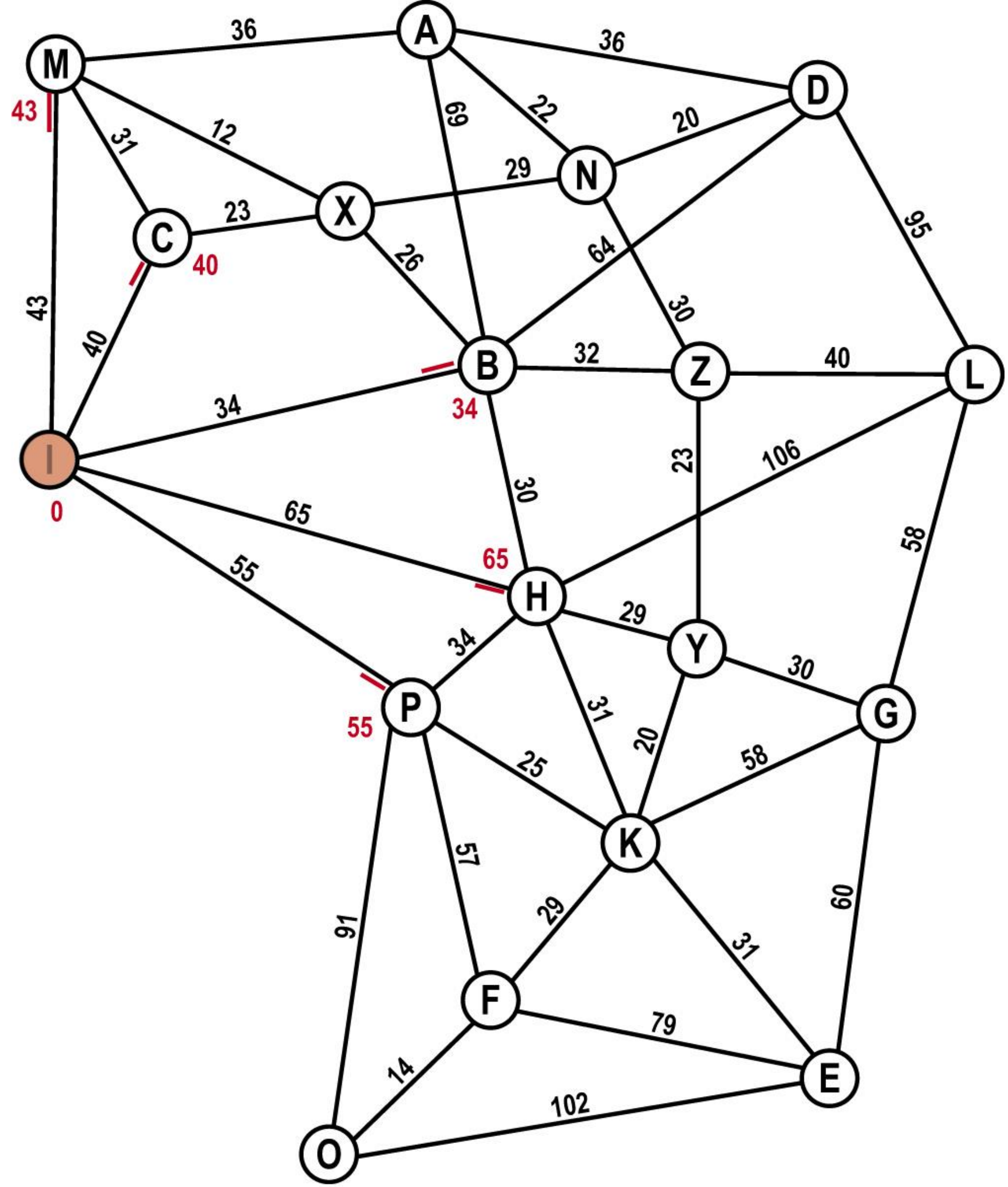


Aus: Gallenbacher, *Abenteuer Informatik*, 2. Aufl.

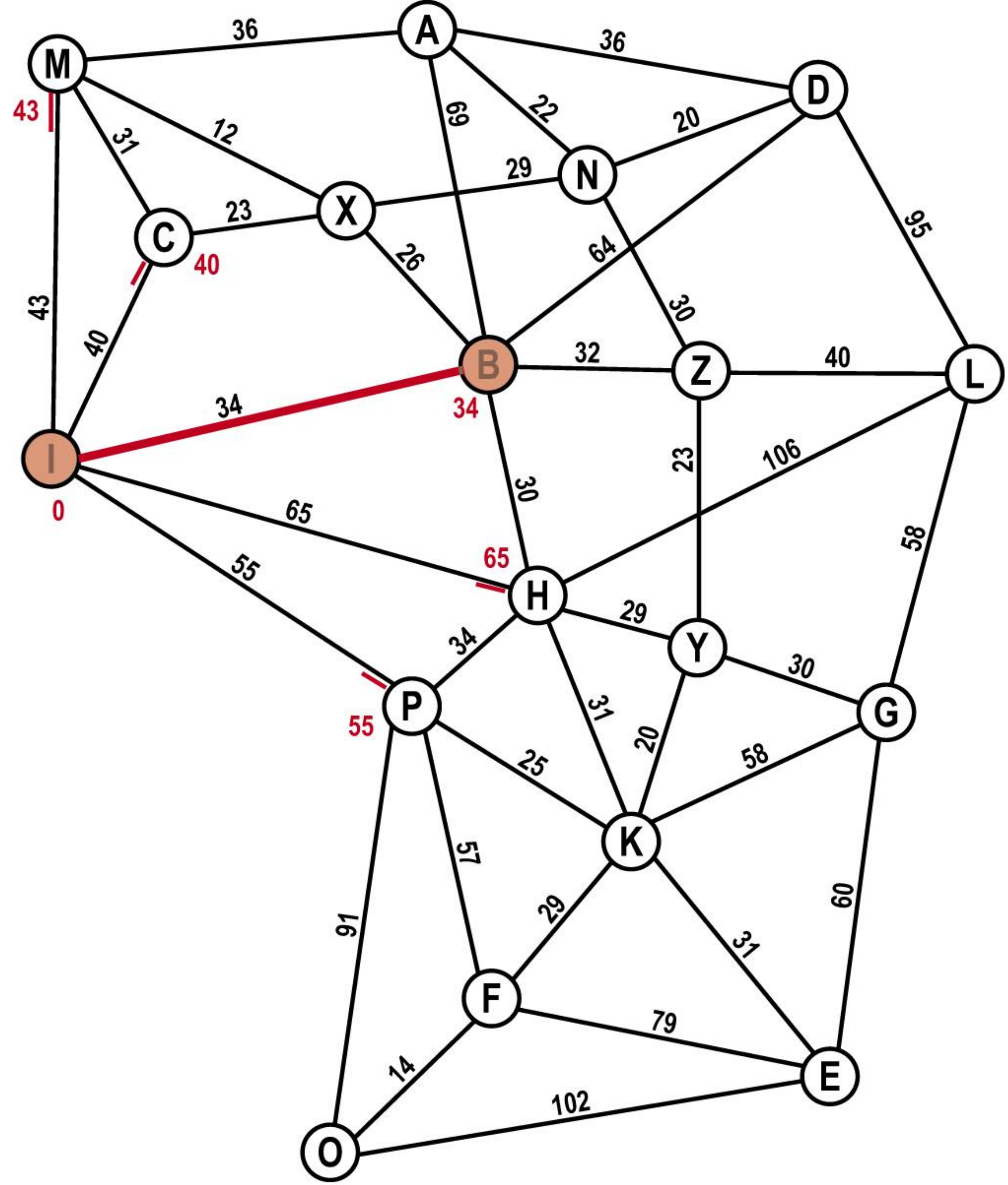
So verlaufen am Ende die Ameisenkarawanen. Sie beschreiben die kürzesten Wege!



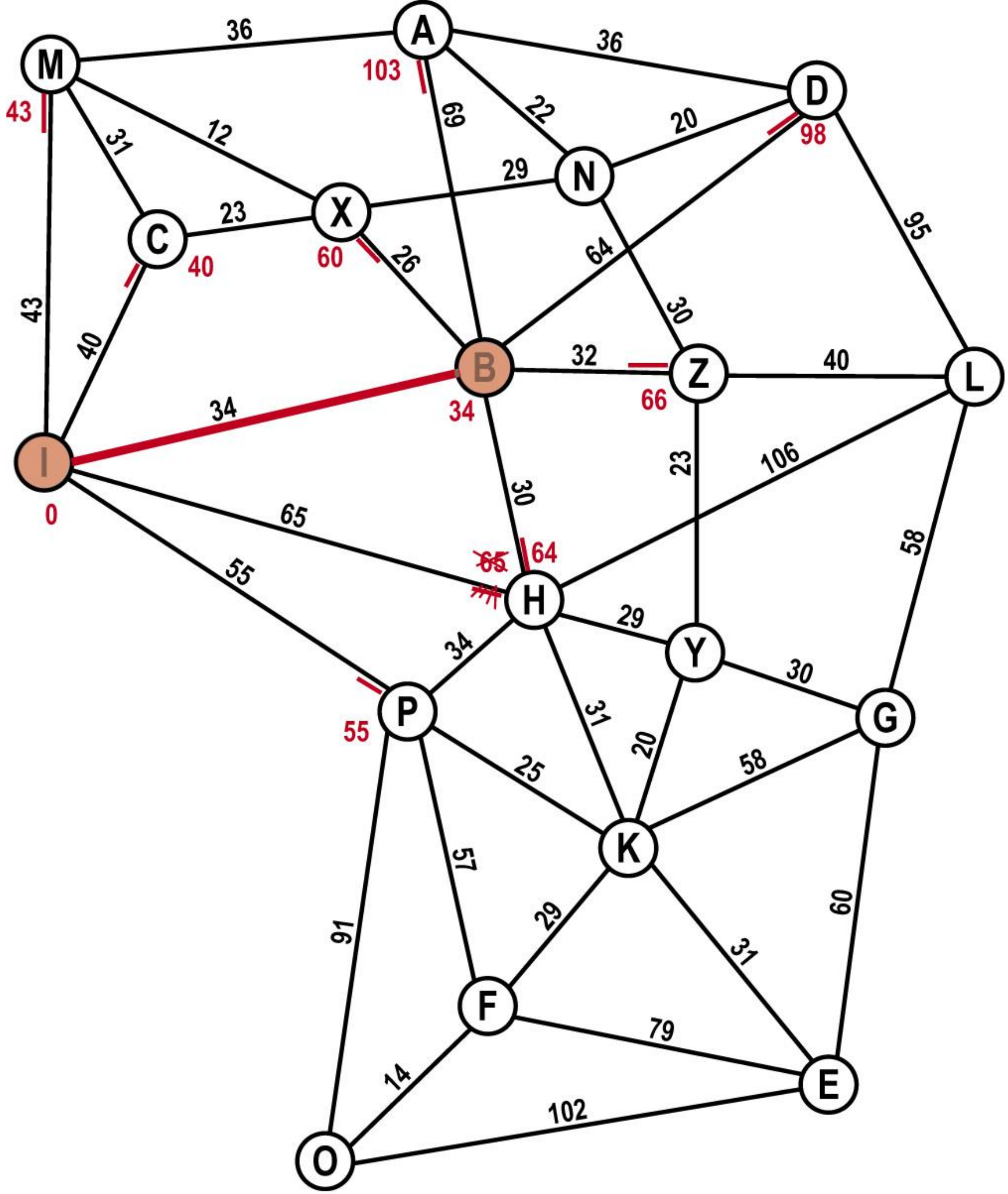
Nochmal von vorn etwas informatischer:
Markiere den Startknoten, schreibe alle Entfernungen in rot zu den Nachbarknoten und markiere den Eingang mit einem Strich.



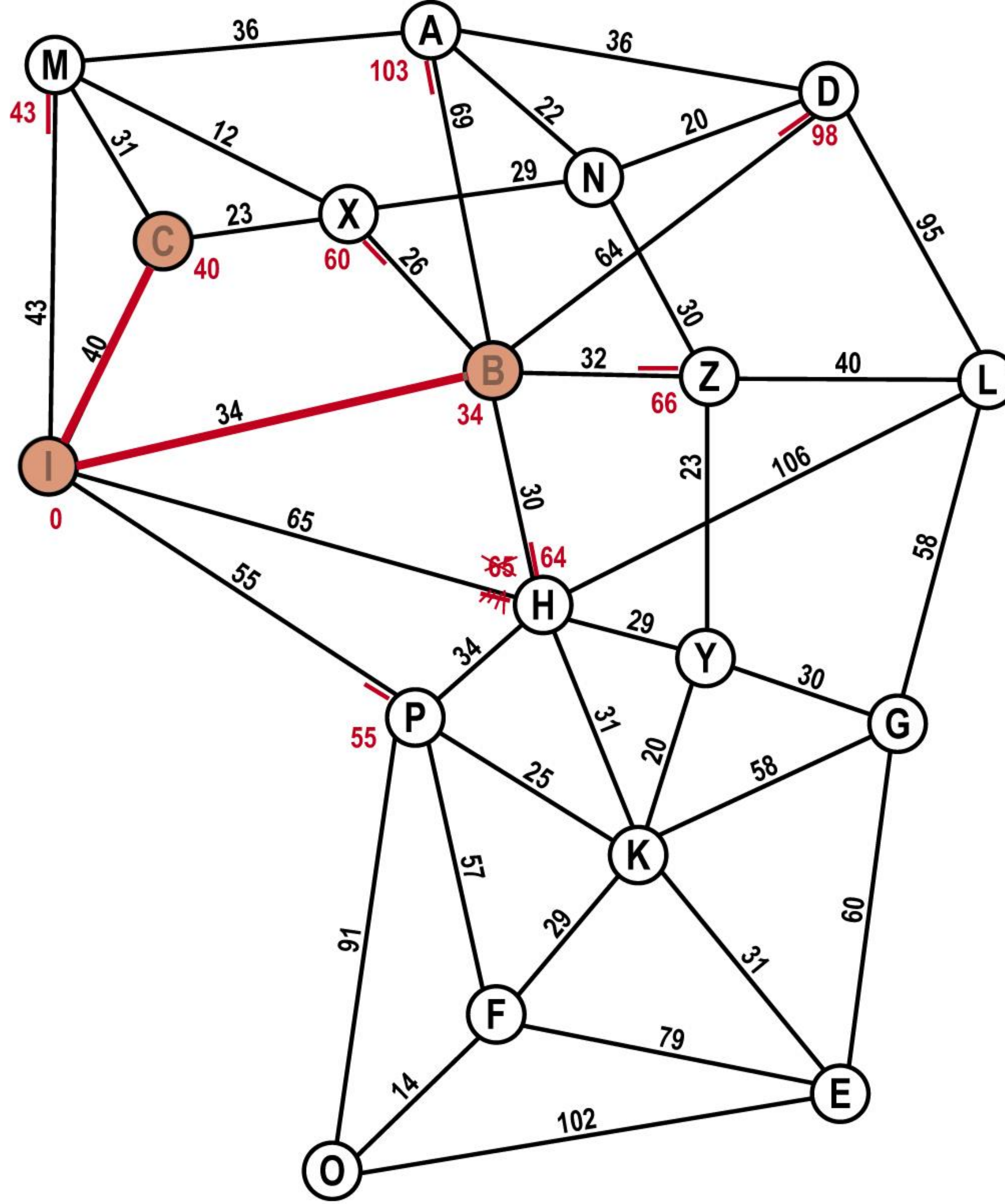
Markiere die Strecke mit dem kleinsten Wert, besser kommt man auf keinen Fall wohin. Markiere den Ort, zu dem diese Strecke führt.



Ergänze die Eingänge von den nun über B errecihbaren Orte und schreib die Entfernung dazu: Von I über B nach X ergibt 60, usw. Du stellst fest dass du über B günstiger nach H kommst (64<65), als auf direktem Weg von I nach H, also weg mit der Strecke I-H.



Sonst gibt es bei B nichts zu tun, also nächstkleineres Kantengewicht suchen und da weiter machen: C
usw.



0. Markiere die **Startstadt** rot, weise ihr die **Kennzahl** 0 zu. Bezeichne diese als **aktuelle Stadt**.

1. Gehe aus von der **aktuellen Stadt** zu allen direkt erreichbaren **Nachbarstädten**

... und führe das Folgende für jede **Nachbarstadt** durch:

Errechne die **Summe** aus der Kennzahl an der **aktuellen Stadt** und der Länge der Strecke dorthin.

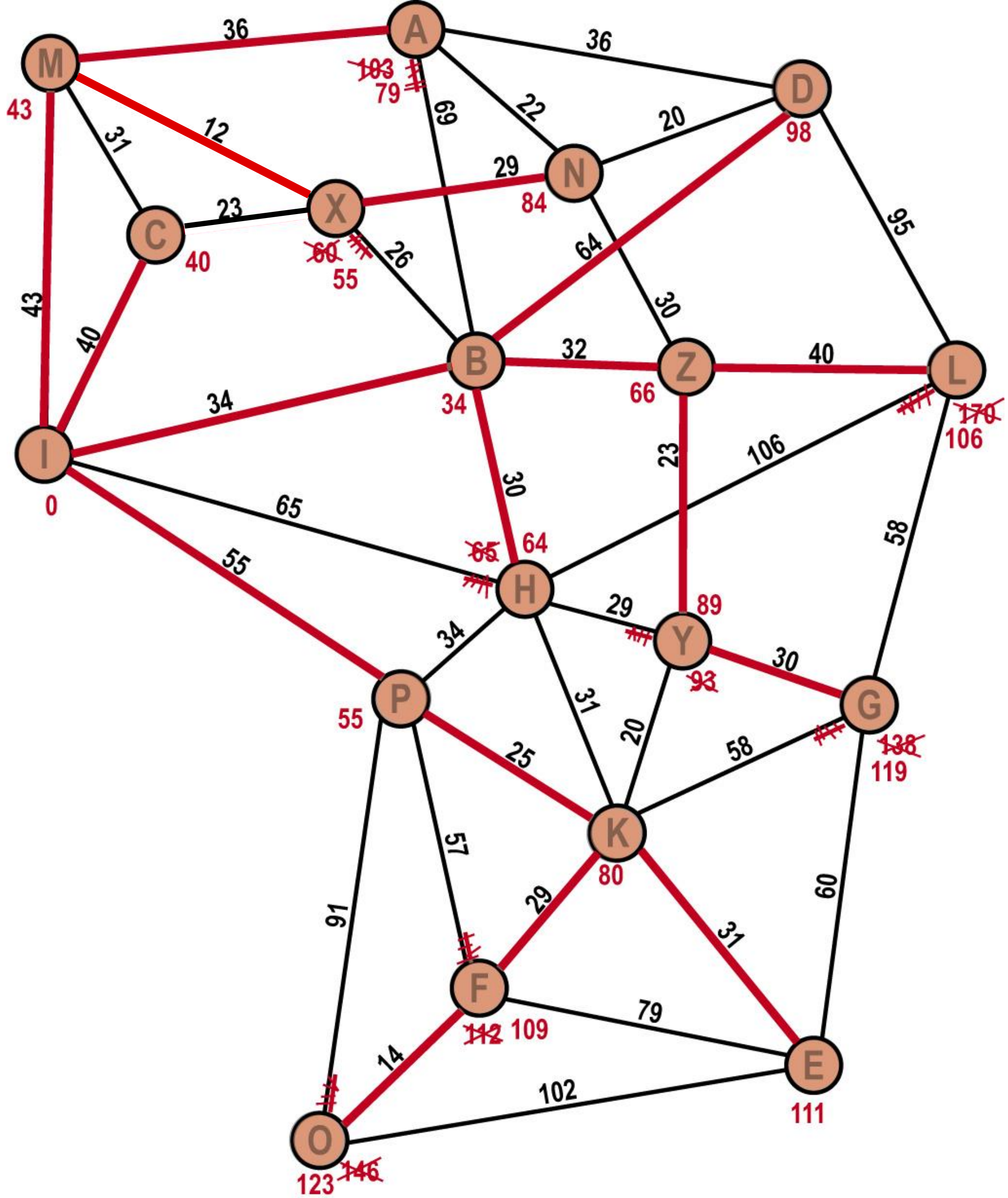
- Ist die **Nachbarstadt** bereits rot markiert, mache nichts.
- Hat die **Nachbarstadt** keine **Kennzahl**, weise ihr die **Summe** als **Kennzahl** zu. markiere die Strecke zur **aktuellen Stadt**.
- Hat die **Nachbarstadt** eine **Kennzahl** kleiner der **Summe**, mache nichts.
- Hat die **Nachbarstadt** eine **Kennzahl** größer der **Summe**, streiche die dortige **Kennzahl** sowie die Markierung. Weise ihr danach die **Summe** als neue **Kennzahl** zu. Markiere die Strecke zur **aktuellen Stadt**.

2. Betrachte alle Städte, die zwar eine **Kennzahl** haben, aber noch nicht rot markiert sind. Suche die Stadt mit der kleinsten **Kennzahl**.

3. Bezeichne diese als **aktuelle Stadt**. Weisen mehrere Städte die kleinste **Kennzahl** auf, wähle eine beliebige davon als **aktuelle Stadt**.

4. Markiere die **aktuelle Stadt** rot, zeichne die dort markierte Strecke in rot ein.

5. Falls es noch Städte gibt, die nicht rot markiert sind, weiter bei (1.)



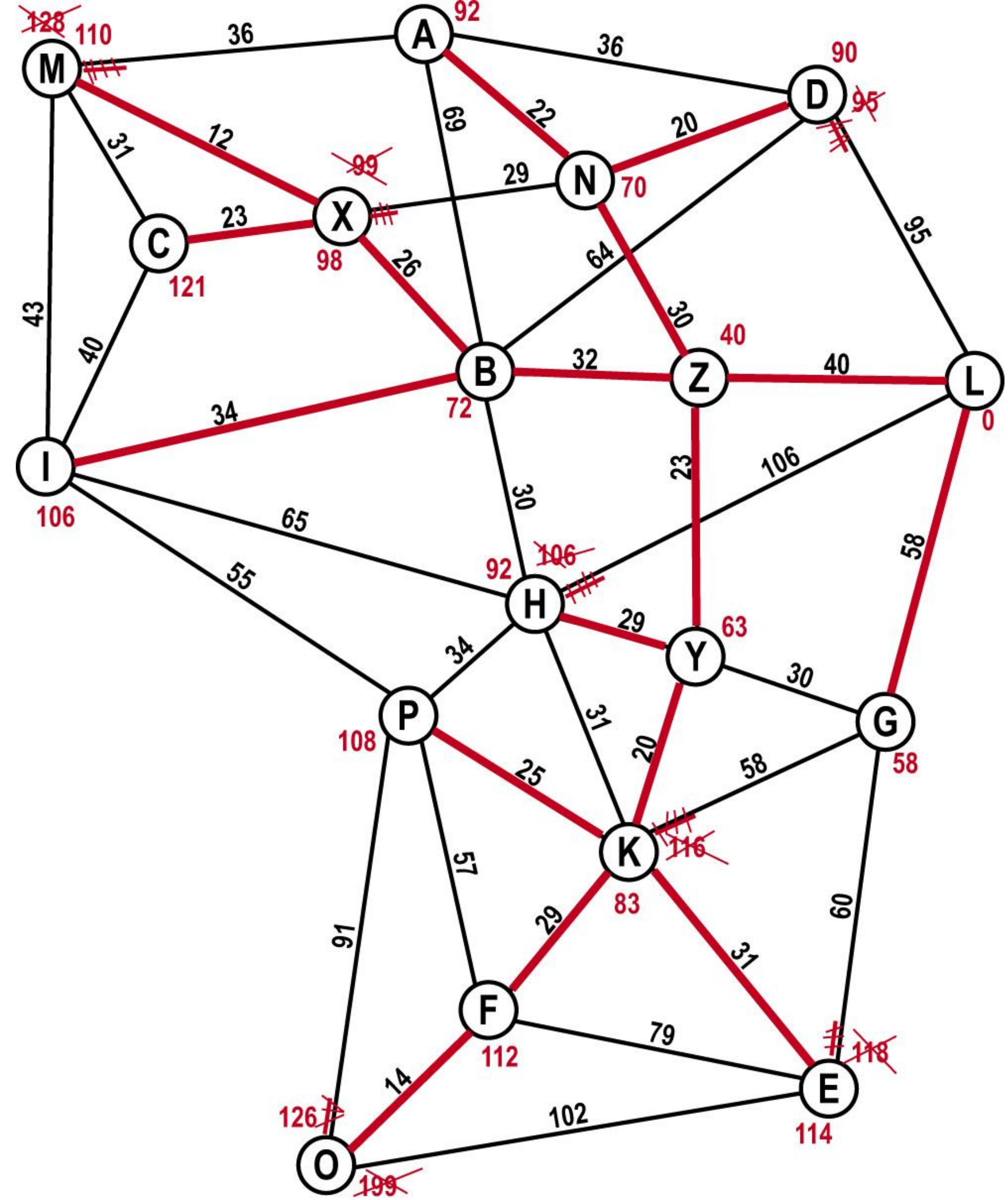
M
A
D
N
X
C
B
Z
L
I
H
Y
P
G
K
F
E
O
36
36
22
20
69
12
31
29
23
26
64
95
30
43
40
32
40
34
106
23
30
65
58
55
29
34
30
31
20
58
25
57
91
29
31
60
79
14
102
43
~~103~~
79
98
84
40
~~60~~
55
34
66
~~170~~
106
0
~~65~~
64
89
~~93~~
55
~~138~~
119
80
~~112~~
109
111
123
~~146~~

Der Dijkstra-Algorithmus liefert nicht nur den kürzesten Weg vom Start zum Ziel, sondern auch alle anderen kürzesten Wege, die von Start ausgehen.

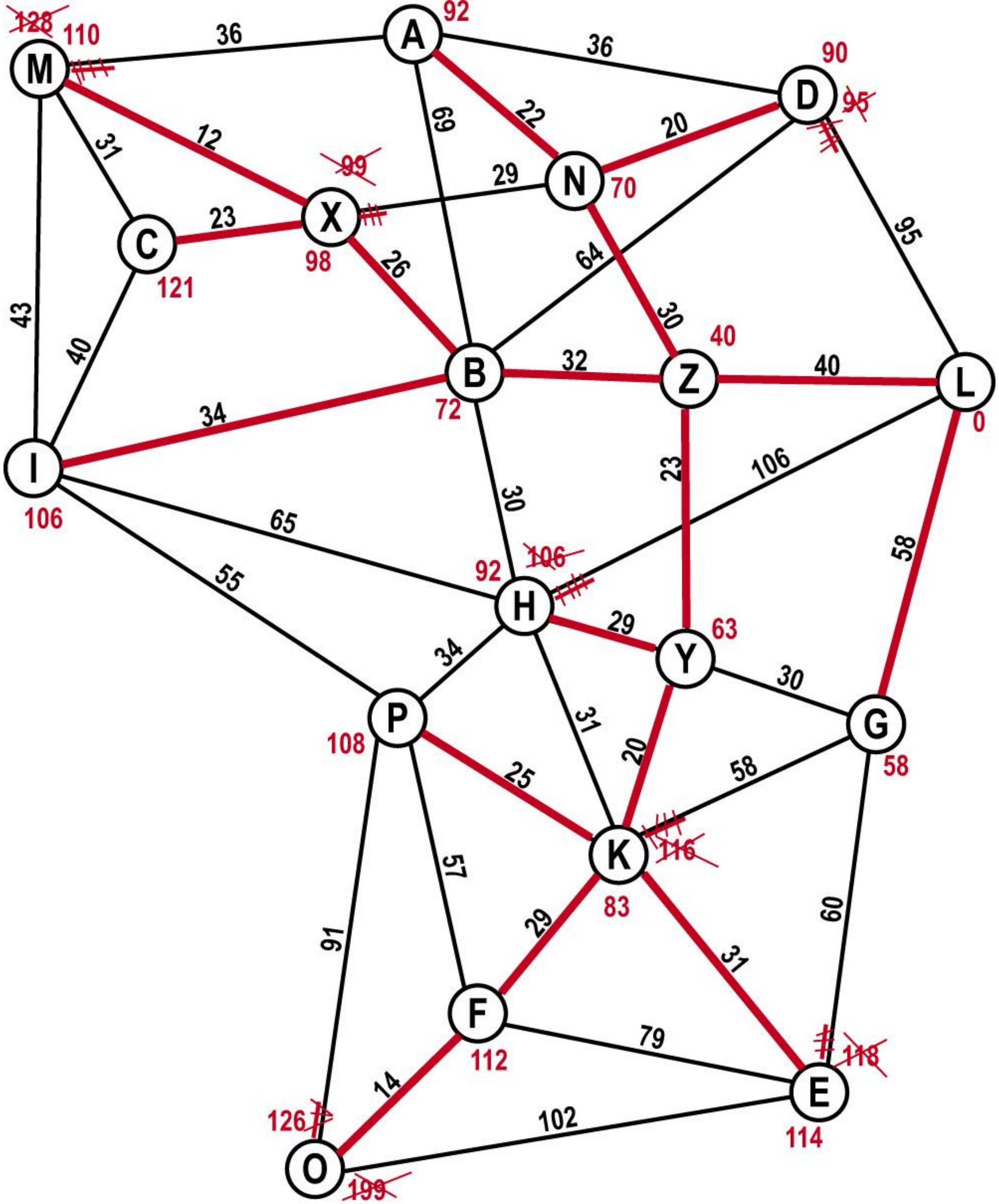
Klassenkarte

Knoten
Datenelement inhalt **Knoten vorgaenger** **int pfadgewicht**
...

Ermittle mittels Dijkstra alle kürzesten Wege ausgehend von Lupera!



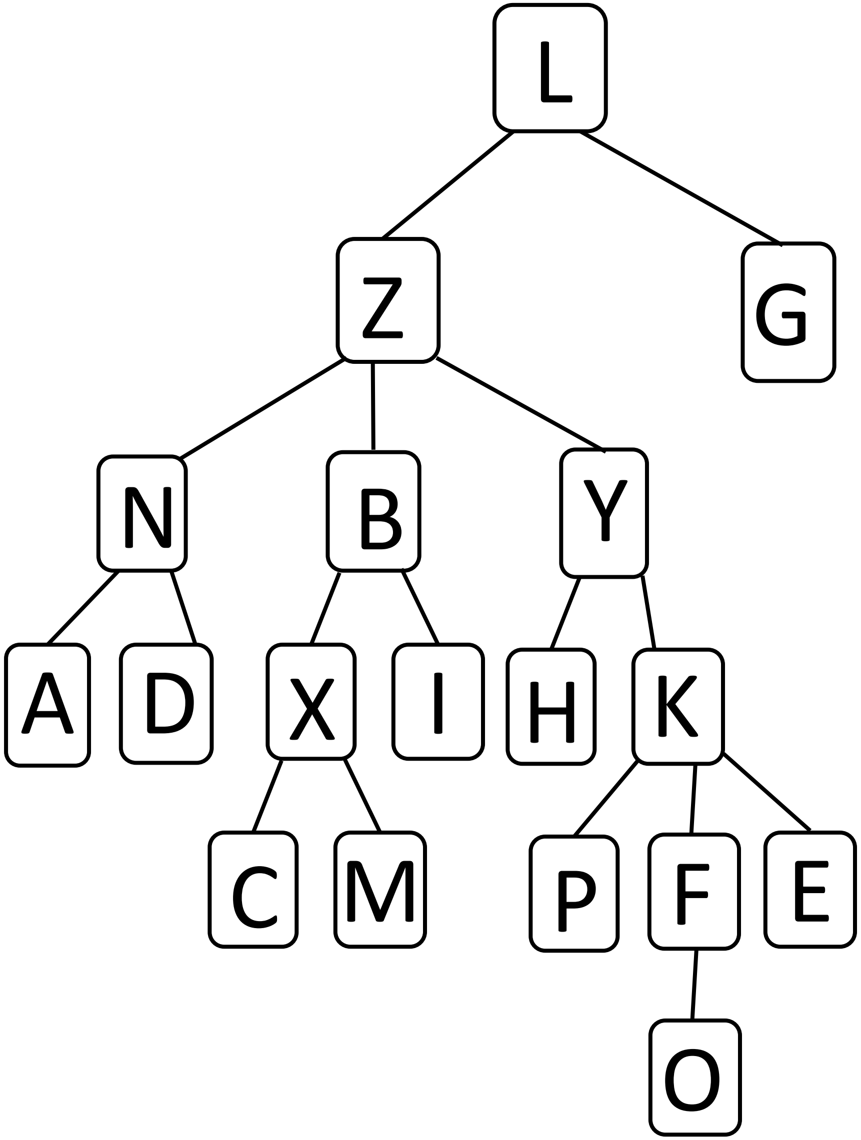
Aus: Gallenbacher, *Abenteuer Informatik*, 2. Aufl.



Aus: Gallenbacher, *Abenteuer Informatik*, 2. Aufl.

Welche Besonderheit weist der rote Graph auf?

Zeichne den entstanden Baum übersichtlicher auf!
Handelt es sich um einen Binärbaum?
Ermittle die Tiefe des Baums!



Welche Besonderheit weist der rote Graph auf?

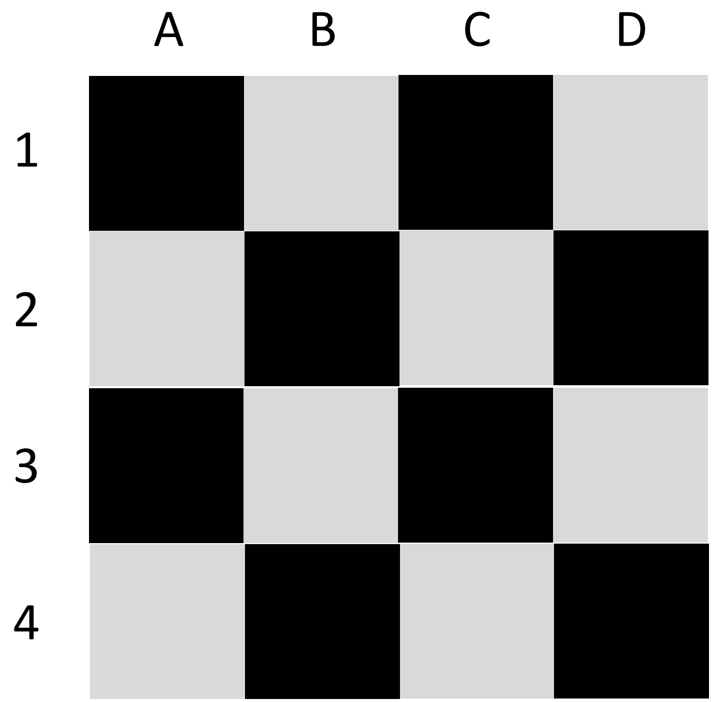
Zeichne den entstanden Baum übersichtlicher auf!
Handelt es sich um einen Binärbaum?
Ermittle die Tiefe des Baums!

Springerproblem: (Hamiltonkreis)
https://www.brainbashers.com/knight.asp
http://www.dr-mikes-math-games-for-kids.com/knights-tour.html

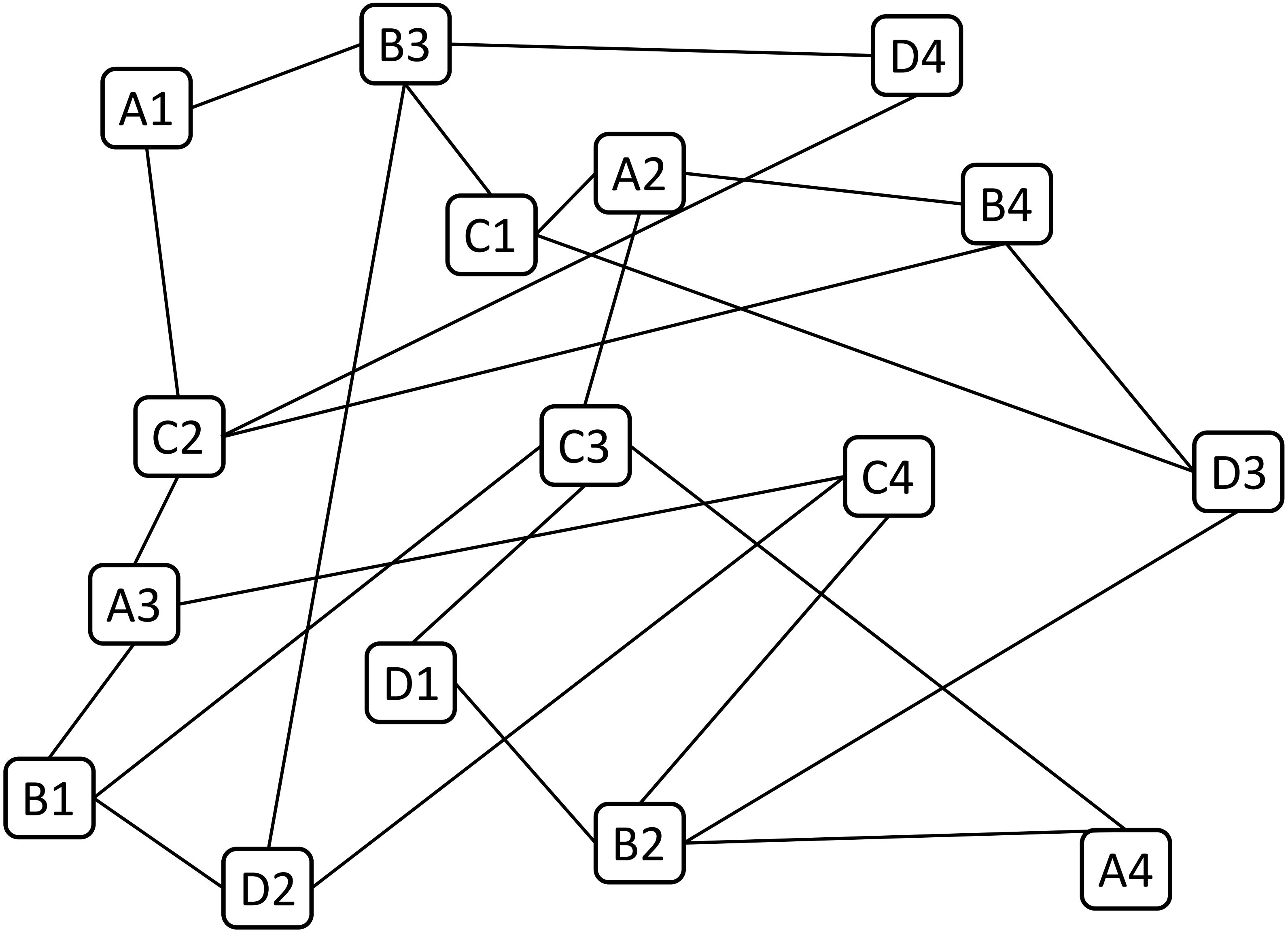
Ein Hamiltonkreis ist ein geschlossener Pfad (Zyklus) in einem Graphen, der jeden Knoten genau einmal enthält.

Ein Eulerkreis ist ein geschlossener Pfad (Zyklus) in einem Graphen, der alle Kanten genau einmal enthält.

Ein Eulerweg ist ein Pfad in einem Graphen, der alle Kanten genau einmal enthält. (Haus vom Nikolaus)



Erstelle den zum Schachbrett gehörenden Graphen bzgl. des Springerproblems!


B3
D4
A1
A2
B4
C1
C2
C3
C4
D3
A3
D1
B1
B2
D2
A4