8. Algorithmen über Graphen und Datenbanken

Klassische graphenbasierte Verfahren, wie die **Zyklensuche** oder die **Suche nach kürzesten Wegen** wurden in der Vergangenheit algorithmisch immer wieder für neue Anwendungen spezifiziert und besitzen bis heute eine zentrale Bedeutung in der Angewandten Informatik.

Beispielsweise werden die **IP-Pakete** im Internet mittels des **Algorithmus von Dijkstra** über kostengünstigste Verbindungen gesendet, wobei eine permanente Lastanalyse der einzelnen Übertragungskanäle stattfindet. Aber auch bei anderen Anwendungen sind derartige Algorithmen hoch aktuell. Das betrifft u.a. die Fahrplanauskunft bei der Bahn, das Suchen nach optimalen Beförderungswegen für Personen oder Güter in Verkehrsnetzen, die Suche von kürzesten alternativen Wegen in Verkehrsleitsystemen oder die Wegesuche seitens GPS - gestützter Verfahren. Als weiteres Beispiele können graphenbasierte Verfahren zum Entwurf von integrierten Schaltungen mit **minimalen Leitungswegen** genannt werden.

Generell lassen sich **alle rekursiven Programme** auch **iterativ** implementieren, was bereits im Rahmen der **Theorie der berechenbaren Funktionen** vor 1970 gezeigt werden konnte.

Die umgekehrte Aussage gilt jedoch nicht. Rekursive Programme sind leichter zu algorithmieren und sind für Außenstehende leichter verständlich, da sie häufig die Problemdefinition umsetzen. Rekursive Algorithmen sind insbesondere dann die geeignete Problemlösung, wenn das zugrunde liegende Problem oder die zu behandelnden Daten rekursiv definiert sind. Das ist vor allem bei mathematischen Definitionen der Fall. Beispielsweise können natürlicher Zahlen wie folgt rekursiv definiert werden:

- 0 sei eine natürliche Zahl
- Der Nachfolger einer natürlichen Zahl ist wieder eine natürliche Zahl

Rekursive Programme werden durch eine **Abbruchbedingung**, **rekursive Aufrufe** des Programmes selbst und durch ein **Backtracking** charakterisiert. Bekannte Sprachen, wie **LISP** oder **PROLOG** bauen auf rekursiven Funktionen und rekursiven Ausdrücken auf.

Neben der **Zyklensuche** eignet sich die **rekursive Tiefensuche** zur Lösung weiterer Fragestellungen über Datenbanken mit rekursiven Beziehungen bzw. Graphen:

- Enthält ein System/Fahrzeug ein bestimmtes Aggregat/Teil ?

- Existiert ein Weg vom Knoten A zum Knoten Z ?

- Besitzen Systeme ab einer gewissen Partitionierung gemeinsame Teile ?

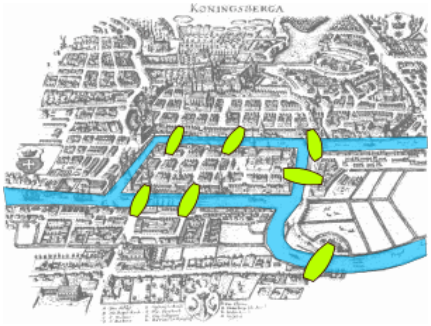
- Ist ein gegebener Graph zusammenhängend ?

- Welche Knoten bilden jeweils einen zusammenhängenden Teilgraphen ?

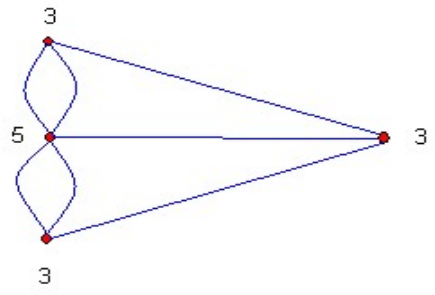
Graphen

Mit dem "Königsberger Brückenproblem" beginnt die dokumentierte Geschichte der Graphentheorie im Jahre 1736. Die beiden Arme des Flusses Pregel umfließen eine Insel, den Kneiphof. Das Problem bestand darin, zu klären, ob es einen Weg gibt, bei dem man alle sieben Brücken über den Pregel genau einmal überquert, und wenn ja, ob auch ein Rundweg möglich ist, bei dem man wieder zum Ausgangspunkt gelangt.

Skizzierter Stadtplan



zugehöriger Graph



Folgende Definitionen sind auf Euler zurückzuführen:

Def.: Sei G ein gerichteter oder ungerichteter Graph. Ein Weg P in G heißt "Eulersch", wenn P jedem Pfeil (gerichtete Kante) aus G genau einmal durchläuft.
Def.: Ist P zusätzlich ein Kreis, so nennt man P "Eulerscher Kreis".
Def.: Graph G heißt "Eulersch", wenn er einen "Eulerschen Kreis" enthält

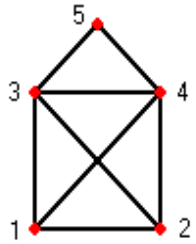
Die Lösung des Problems ergibt sich dem **Satz von Euler** für ungerichtete Graphen:

Ein endlicher, ungerichteter und zusammenhängender Graph ist genau dann "Eulersch", wenn alle Ecken **geraden Grad** haben. Er hat genau dann einen Eulerschen Weg, der kein Kreis ist, wenn **genau 2 Knoten ungeraden Grad** haben (diese bilden den Anfangs- und Endpunkt jedes Eulerschen Weges).

Wie Leonhard Euler 1736 bewies, war ein solcher Weg bzw. "Eulerscher Weg" in Königsberg nicht möglich, da zu allen vier Ufergebieten bzw. Inseln eine ungerade Zahl von Brücken führte.

Es dürfte maximal zwei Ufer (Knoten) mit einer ungeraden Zahl von angeschlossenen Brücken (Kanten) geben. Diese zwei Ufer könnten Ausgangs- bzw. Endpunkt sein. Die restlichen Ufer müssten eine gerade Anzahl von Brücken haben, um sie auch wieder verlassen zu können.

Das beliebte Kinderrätsel "Das-ist-das-Haus-vom-Ni-ko-laus" hingegen enthält einen Eulerweg, aber keinen Eulerkreis, da sein Graph zwei Knoten vom Grad 3 enthält:



Ein Eulerweg ist z.B. 1-2-4-3-1-4-5-3-2. Knoten 1 und 2 haben jeweils 3 Nachbarn, ihr Grad ist ungerade. Um das Haus in einem Zug zeichnen zu können muss man daher an einem dieser beiden Punkte beginnen.

Ein Quadrat mit Diagonalen enthält keinen Eulerweg, da alle seine Knoten den Grad 3 haben.

Speicherung von Graphen

Gerichteter Graph

Ein gerichteter Graph ist ein Quadrupel **G = (V, R,** α**,** ω**)** mit folgenden Eigenschaften:
(i) **V** ist eine nichtleere Menge, die Eck- bzw. Knotenmenge des Graphen
(ii) **R** ist eine Menge, die Pfeil- bzw. gerichtete Kantenmenge des Graphen
(iii) Es gilt $\mathbf{V} \cap \mathbf{R} = \emptyset$
(iv) α**:** $\mathbf{R} \rightarrow \mathbf{V}$ und ω**:** $\mathbf{R} \rightarrow \mathbf{V}$ sind Abbildungen (α(r) ist Anfangsknoten und ω(r) ist Endknoten der gerichteten Kante r))

Ungerichteter Graph

Ein ungerichteter Graph ist ein Tripel **G = (V, E,** γ**)** aus einer nichtleeren Menge **V**, einer Menge **E** mit $\mathbf{V} \cap \mathbf{E} = \emptyset$ und einer Abbildung γ**:** $\mathbf{E} \rightarrow \{ \mathbf{X}: \mathbf{X} \subseteq \mathbf{V}$ **mit** $\mathbf{1} \leq |\mathbf{X}| \leq \mathbf{2}\}$, die jeder Kante ihre Endknoten (Endecken) γ**(e)** $\in$ **V** zuordnet. Die Elemente von **V** heißen wieder Ecken (Knoten) von **G**, die Elemente von **E** heißen Kanten. Für eine Kante **e** heißen die Elemente von γ**(e)** die Endpunkte von **e**.

Speicherung von Graphen

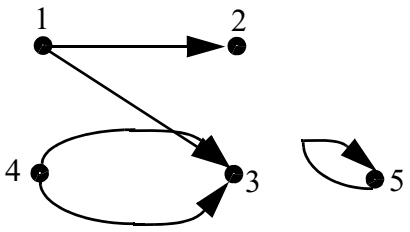
G = (V, R, α**,** ω**)** sei stets ein gerichteter Graph, **G = (V, E,** γ**)** sei stets ein ungerichteter Graph. In $\mathbf{V} = \{ \mathbf{v_1}, ..., \mathbf{v_n} \}$ und $\mathbf{R} = \{\mathbf{r_1}, ... , \mathbf{r_m} \}$ bzw. $\mathbf{E} = \{ \mathbf{e_1}, ..., \mathbf{e_m} \}$ seien die Elemente in einer beliebigen Reihenfolge durchnummeriert.

Adjazenzmatrix

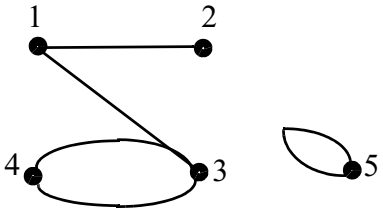
Die Adjazenzmatrix eines **gerichteten** Graphen **G = (V, R,** α**,** ω**)** ist die n x n-Matrix **A(G)** mit $\mathbf{a_{ij}} = |\{\mathbf{r} \in \mathbf{R}: \alpha(\mathbf{r}) = \mathbf{v_i} \text{ und } \omega(\mathbf{r}) = \mathbf{v_j}\}|$.

Die Adjazenzmatrix eines **ungerichteten** Graphen **H = (V, E,** γ**)** ist die n x n-Matrix **A(H)** mit $\mathbf{a_{ij}} = |\{\mathbf{e} \in \mathbf{E}: \gamma(\mathbf{e}) = \{\mathbf{v_i}, \mathbf{v_j}\} \}|$.

Beispiel:



gerichteter Graph G



ungerichteter Graph H

Adjazenzmatrizen

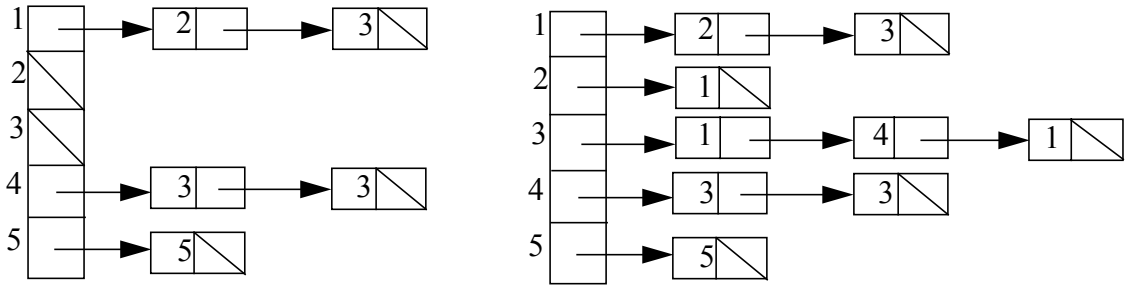
$$\mathbf{A(G)} = \begin{vmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{vmatrix} \qquad \mathbf{A(H)} = \begin{vmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 2 & 0 \\ 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{vmatrix}$$

Adjazenzlisten

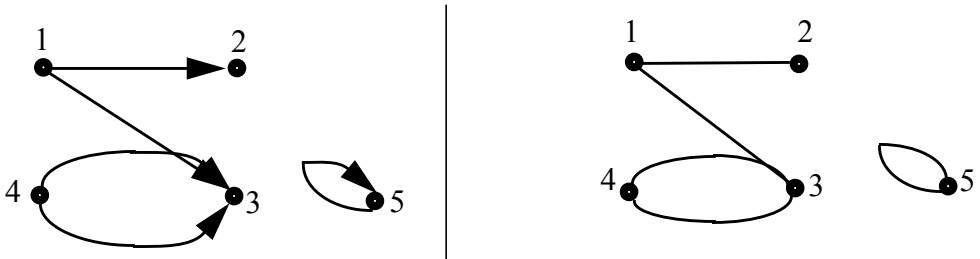
Eine sinnvolle Speicherung eines Graphen ist die **Adjazenzlisten**-Repräsentation.
Diese besteht aus einem **Vektor aus n Adjazenzlisten**, wobei für jede Ecke **v∈V** eine Liste **Adj[v]** vorhanden ist.

gerichteter Graph **ungerichteter Graph**

Adj[] **Adj[]**



Beispiel:



gerichteter Graph G | ungerichteter Graph H

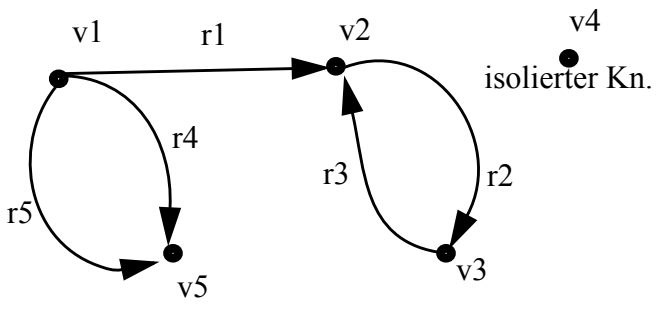
Sei **G** Graph, Eckenmenge V(G), Pfeilmenge R(G)

Definition: Graph G(V,R) endlich, wenn V und R endlich

Anzahl Ecken / Knoten: **n := |V(G)|**
Anzahl Kanten / Pfeile: **m := |R(G)|**

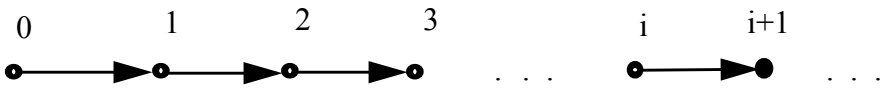
Beispiel: gerichteter Graph, nicht zusammenhängend:

	Vorgänger α	Vorgänger ω	Nachfolger
r1	v1		v2
r2	v2		v3
r3	v3		v2
r4	v1		v5
r5	v1		v5
r6	v5		v5



Gerichtete Graphen

Beispiel: unendlicher gerichteter Graph
G = (V, R, α, ω), V = N = {0, 1, 2, 3, 4, ... } natürliche Zahlen
R = { (i, i+1), i∈N } , α((i, i+1)) = i, ω((i, i+1)) = i+1



Definition: Sei G=(V,R,α,ω) Graph, Pfeil r∈R heißt **Schlinge**, wenn α(r) = ω(r).
Graph G ist **schlingenfrei**, wenn es keine Schlingen gibt.

Definition: Zwei Pfeile r, r'∈R, r ≠r' sind **parallel**, wenn α(r) = α(r') und ω(r) = ω(r').
Pfeile r und r' heißen **invers** oder **antiparallel**, wenn α(r) = ω(r') und ω(r) = α(r').

Definition: Graph ist **einfach**, wenn er keine Schlingen und keine Parallelen enthält.

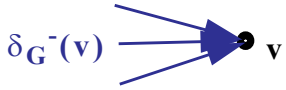
Definition: Sei G=(V,R,α,ω) Graph, **Ecke v** und **Kante r** heißen **inzident**, wenn **v** entweder Anfangs- oder Endecke von **r** ist, d.h. **v∈{α(r), ω(r)}** gilt.

Definition: Sei G=(V,R,α,ω) Graph, zwei Pfeile **r** und **r'** heißen **inzident**, wenn es eine Ecke **v** gibt, die mit **r** und **r'** inzidiert.

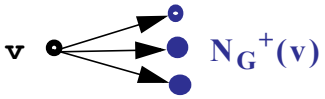
Definition: Sei G=(V,R,α,ω) Graph, zwei Ecken **u** und **v** heißen **adjazent** oder **benachbart**, wenn es einen Pfeil **r∈R** gibt, der zu **u** und **v** inzident ist.

$\delta_G^+(v) := \{r \in R: \alpha(r) = v\}$ heißt das von **v ausgehende** Pfeilbüschel

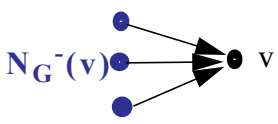
$\delta_G^-(v) := \{r \in R: \omega(r) = v\}$ heißt das in **v mündende** Pfeilbüschel



$N_G^+(v) := \{\omega(r): r \in \delta_G^+(v)\}$ heißt **Nachfolgermenge von v**



$N_G^-(v) := \{\alpha(r): r \in \delta_G^-(v)\}$ heißt **Vorgängermenge von v**



$\mathbf{g_G^+(v) := |\delta_G^+(v)|}$ heißt **Außengrat** von v (Anzahl der von v abgehenden Kanten)

$\mathbf{g_G^-(v) := |\delta_G^-(v)|}$ heißt **Innengrat** von v (Anzahl der in v endende Kanten)

$\mathbf{g_G(v) := g_G^+(v) + g_G^-(v)}$ heißt der **Grad** von v

$\mathbf{\Delta(G) := max\{g_G(v):v \in V\}}$ heißt **Maximalgrad** von G

Sei **G=(V,R,α,ω)** ein Graph mit endlicher Pfeilmenge, $|R|<+\infty$, dann gilt:

$$\sum_{v \in V} g^+(v) = \sum_{v \in V} g^-(v) = |R|$$

d.h.

$$\sum_{v \in V} g(v) = 2 \cdot |R|$$

d.h. sei **G=(V, R, α, ω)** ein endlicher Graph, dann ist die Anzahl der **Ecken in G mit ungeradem Grad gerade**.

Ungerichtete Graphen

Sei **G = (V, E, γ)**, ist **γ(e)={v}** (eindeutig), dann ist **e eine Schlinge**.

δ(v) := {e∈E: v∈γ(e)} heißen die **zu v inzidenten Kanten**

N_G(v) := {u∈V: γ(e)={u,v} für ein e∈E} heißen die zu **v adjazenten Ecken** oder **Nachbarn von v**

g_G(v) := $\sum_{e \in E: v \in \gamma(e)}$ **(3 - |γ(e)|)** ist der **Grad von v**

Δ(G) := max{g_G(v):v∈V} heißt **maximaler Grad von G**

Sei **G = (V, E, γ)** ein endlicher ungerichteter Graph. Die Anzahl der Ecken in G mit ungeradem Grad ist gerade.

Inzidenzmatrix
Die Inzidenzmatrix I(G) eines schlingenfreien gerichteten Graphen G ist eine n x m Matrix mit

$$i_{k,l} := \begin{cases} 1, & \text{falls } \alpha(r_l) = v_k \\ -1, & \text{falls } \omega(r_l) = v_k \\ 0, & \text{sonst} \end{cases}$$

Im Fall eines ungerichteten Graphen H gilt

$$i_{k,l} := \begin{cases} 1, & \text{falls } v_k \in \gamma(e_l) \\ 0, & \text{sonst} \end{cases}$$

Da die beiden obigen Graphen nicht schlingenfrei sind, ist für sie daher die Inzidenzmatrixdarstellung nicht möglich. Das Entfernen der Schlingen liefert für die resultierenden Graphen **G'** und **H'** die folgenden Inzidenzmatrizen:

$$\mathbf{I(G')} = \begin{pmatrix} 1 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & -1 & -1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \qquad \mathbf{I(H')} = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

8.1 Rekursive Tiefensuche

Die Tiefensuche **DFS** (depth first search) dient als effizientes Verfahren zur Gewinnung von strukturellen Informationen über Graphen, um z.B. Zusammenhangskomponenten, Zyklen oder die Existenz von Wegen zu ermitteln.

Der Graph wird durch **DFS** nach der Strategie "zunächst in die Tiefe gehen" durchsucht. Die Pfeile des Graphen werden ausgehend von dem zuletzt gefundenen Knoten **v**, von der noch unerforschte Kanten starten, aus erforscht. Wenn alle von **v** ausgehenden Kanten erforscht sind, dann erfolgt ein "Backtracking" zu dem Knoten, von dem aus **v** entdeckt wurde.

In der folgenden Darstellung des Algorithmus benutzen wir **drei Farbmarkierungen** (**weiß**, **grau**, **schwarz**) für die Knoten.

Am **Anfang** ist jeder Knoten **weiß**, d.h. noch **nicht entdeckt**. Sobald ein **neuer Knoten** entdeckt wird, wird er **grau** gefärbt. Wenn ein **Knoten** komplett abgearbeitet ist, d.h. wenn alle von ihm ausgehenden **Kanten erforscht** worden sind, wird er **schwarz** gefärbt.

DFS versieht die Knoten mit Zeitmarken. Jeder Knoten $\mathbf{v} \in \mathbf{V}$ hat zwei Zeitmarken **d[v] < f[v]**:

1. **d[v]**: Zeitpunkt, zu dem **v** entdeckt wurde

2. **f[v]**: Zeitpunkt, zu dem die Adjazenzliste von **v** komplett erforscht wurde, d.h. zu dem alle von **v** ausgehenden Kanten erforscht wurden.

Mit **I(v) := [d(v), f(v)]** bezeichnet man das durch die Zeitmarken aufgespannte Zeitintervall.

Wenn ein Knoten neu entdeckt wird, während die Adjazenzliste von **u** durchlaufen wird, so wird **u** als Vorgänger π**[v]** von **v** gemerkt.

Algorithmus Depth First Search Hauptprozedur

```
DFS(G) {
  Input: Graph G = (V, R, α, ω) in Adjazenzlistendarstellung (V: Menge der Knoten,
         R: Menge gerichteter Kanten, α: R→V und ω: R→V sind Abbildungen (α(r) ist
         Anfangsknoten und ω(r) ist Endknoten der gerichteten Kante r))
  for each v∈V do {
      farbe[v]:=weiß   // alle Knoten unentdeckt
      π[v]:=nil        //  bisher alle Knoten ohne Vorgänger
  }
  Rπ:=0
  zeit:=0
  for each v∈V do {
      if farbe[v] == weiß then   // Falls es noch einen unerforschten Knoten v gibt, dann
         DFS-VISIT(v)            // erforsche  v
}
```

Algorithmus Depth First Search

```
DFS-VISIT(u) {
  farbe[u] := grau
  d[u] := zeit
  zeit := zeit+1
  for each v∈ADJ[u] do    // Erforsche Kanten von u nach v // alle v mit ADJ[u][v] ≠ 0
      if farbe[v] == weiß then {
         π[v] := u            // u ist Vorgänger von v im DFS-Wald
         Rπ := Rπ ∪ {ruv}, wobei ruv die Kante von u nach v ist, die dem Eintrag von v∈ADJ[u]
                              entspricht
         DFS-VISIT(v)
      }
  farbe[u] := schwarz
  f[u] := zeit
  zeit := zeit + 1
}
```

Als Ergebnis wird der **Vorgängergraph** G_π **:= (V, R,** $\alpha|_{R\pi}$**,** $\omega|_{R\pi}$**)** erhalten, der keinen Kreis enthält.

Die Algorithmen **DFS(G)** und **DFS-VISIT(u)** können auch für **ungerichtete Graphen** verwendet werden. In der Zeile $R_\pi := R_\pi \cup \{r_{uv}\}$ wird dann die ungerichtete Kante zwischen u und v zu R_π hinzugefügt.

Definition Wald und Baum
Ein ungerichteter Graph **G = (V, E,** γ**)** heißt **Wald**, wenn **G** kreisfrei ist, also keinen elementaren Kreis besitzt. Falls G zusätzlich **zusammenhängend** ist, so heißt G **Baum**.

Definition
Zwei Knoten **v**$\in$**V(G)** und **w**$\in$**V(G)** eines gerichteten oder ungerichteten Graphen **G** heißen **zusammenhängend**, wenn **w** von **v** aus über eine Kantenfolge erreichbar ist und umgekehrt. Sind alle Knoten eines Graphen **G** zusammenhängend, so wird **G** zusammenhängend genannt. Alle zusammenhängenden Knoten eines Graphen **G** bilden eine **Zusammenhangskomponente**.

Definition Wurzel, Wurzelbaum
Sei **G = (V, R,** α**,** ω**)** ein gerichteter Graph. Der Knoten **s**$\in$**V** heißt Wurzel von **G** , wenn alle Knoten von **s** aus erreichbar sind.

Ein **Wurzelbaum** mit Wurzel **s** ist ein Baum **G**, der eine Wurzel besitzt. Knoten ohne Nachfolger werden **Blätter** genannt.

Der **Vorgängergraph** G_π **:= (V, R,** $\alpha|_{R\pi}$**,** $\omega|_{R\pi}$**)** ist ein Wald. Jede Zusammenhangskomponente von G_π ist ein Wurzelbaum.

Eine **gerichtete Kante r**$\in$**R**, bei der α**(r)** ein **Nachfahre** von ω**(r)** in G_π ist, wird "**Rückwärtskante**" genannt.

DFS produziert "Rückwärtskanten" genau dann, wenn der Graph Kreise enthält.

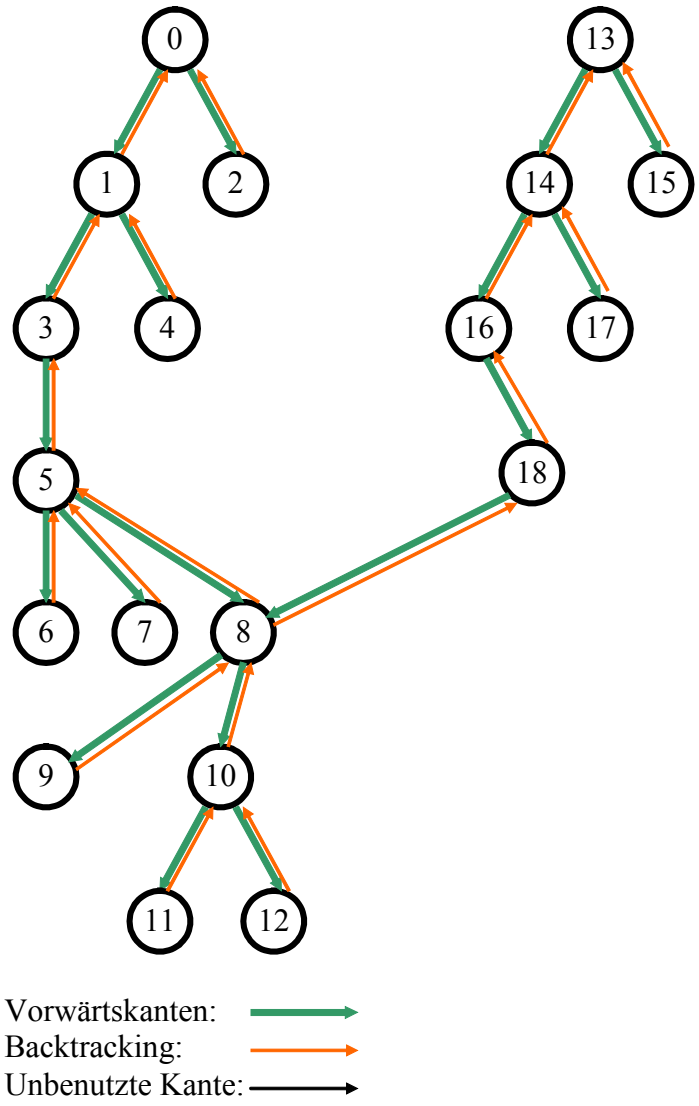
Algorithmus zur Bestimmung aller erreichbaren Knoten E_G(s) in G vom Knoten s
(einfache Form der Breitensuche BFS mit L als Queue)

```
ERREICHBAR(G, s, p) {
 Input:  Graph G = (V, R, α, ω) in Adjazenzlistendarstellung (V: Menge der Knoten,
         R: Menge gerichteter Kanten, α: R→V und ω: R→V sind Abbildungen (α(r) ist
         Anfangsknoten und ω(r) ist Endknoten der gerichteten Kante r)), eine Ecke s∈V(G),
         eine "Markierungszahl" p∈N
 Output: Die Menge E_G(s) der Knoten, die in G von s aus erreichbar sind.

 Setze  marke[s]: = p und marke[v] := nil  für  alle v∈V\{s}
 L := (s)
 while L ≠ 0 do {
        Entferne das erste Element u aus L
        for each v∈ADJ[u] do {                    // alle v mit ADJ[u][v] ≠ 0
            if marke[v]=nil then
              Setze marke[v]:=p  und füge v an das Ende von L an
        }
  }
  return EG(s) := {v ∈ V: marke[v] == p }
}
```

8.1.1 Rekursive Tiefensuche (DFS)

Das Wesen des Algorithmus der [**links]rekursiven Tiefensuche** besteht darin, daß beginnend mit dem ersten Objekt ohne Vorgänger die am weitesten links existierende und noch nicht durchlaufene Kante eines Knotens ausgewählt wird, um zum nächsten untergeordneten Knoten zu gelangen, dort wird rekursiv derselbe Algorithmus wiederholt. Erst wenn in einem Knoten keine oder keine unbenutzte Kante zu einem Nachfolgerknoten existiert, dann erfolgt ein **Backtracking** zum vorherigen Knoten (immer entgegengesetzt des zuletzt durchlaufenen Weges), wo wieder rekursiv nach der bisher nächsten unbenutzten Kante gesucht wird. Da die aus den hierarchischen Darstellungen abgeleiteten Graphen nicht zusammenhängend sein müssen und auch mehrere Knoten ohne Vorgänger existieren können, muß die rekursive Tiefensuche seitens **aller Knoten ohne Vorgänger** durchgeführt werden. Im folgenden Beispiel, welches als gerichteter Graph ohne die Kante **11 -> 18** abgebildet wird, sind das die Knoten **0** und **13**. Aus der Historie des Knotendurchlaufes ist zu erkennen, daß im Falle eines zyklenfreien Graphen im Backtracking immer wieder der Startknoten erreicht wird.

Beispiel: Start in Knoten **0** und **13**



Es wird ein **gerichteter, schleifenfreier Graph G** vorausgesetzt, der aus der Menge der **Knoten V** und der Menge der **Kanten** $E \subseteq \{(v, w): v, w \in V\} = V \times V =: V^2$ besteht.

Mit **V(G)** wird die **Knotenmenge**, mit **E(G)** die **Kantenmenge** des Graphen **G** bezeichnet.

Eine **Kante** $(v, v) \in G$ wird **Schleife** genannt. Ein Graph $G = (V, E)$ heißt **schleifenfrei**, wenn er **keine Schleife** enthält.

Die **Menge der Nachfolger** eines Knotens **v** wird als $N^+(v) = \{w \in V : (v, w) \in E\}$ definiert und ist die Menge aller Knoten **w**, die über eine Kante **(v, w)** mit **v** verbunden sind.

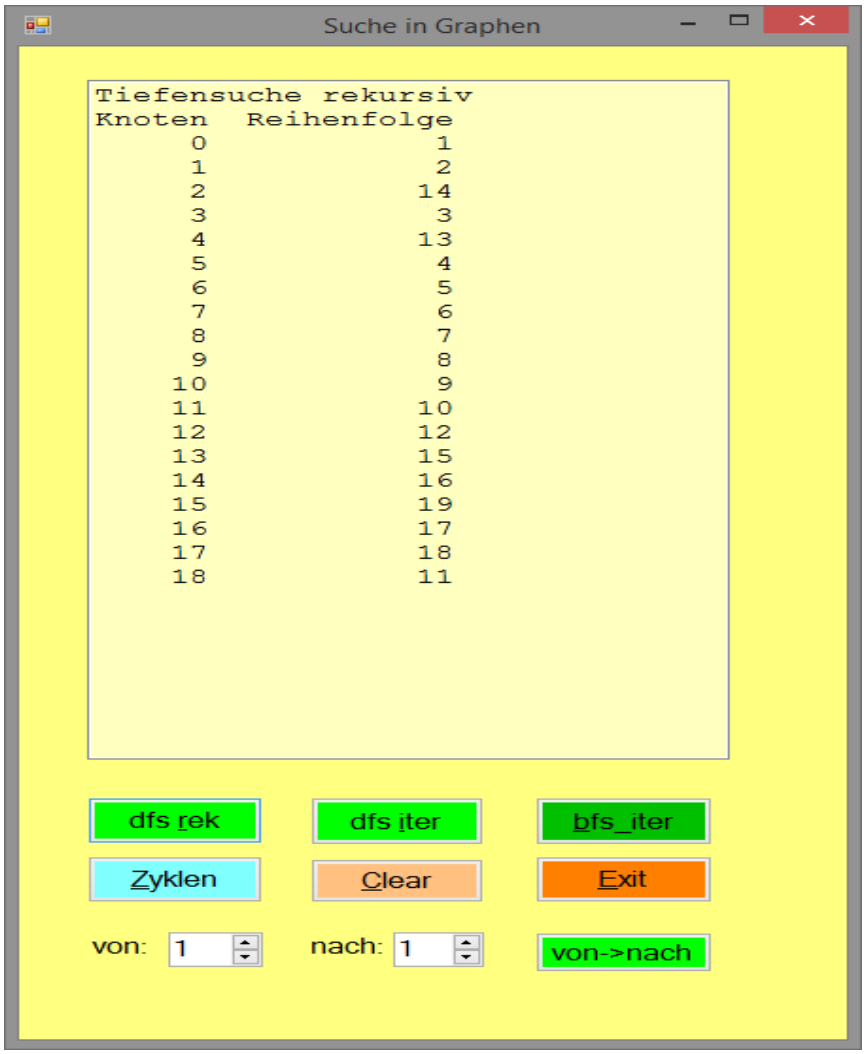
In der folgenden Implementierung der **rekursiven Tiefensuche**, auch **DFS** (depth-first search) genannt, werden erreichte Knoten **v** im Vektor **dfs_num über dfs_num[v] = ++number** mit einer fortlaufenden Nummer **number** versehen. Wird ein Knoten ein zweites Mal besucht, d.h. wenn dessen Nummer größer als 0 ist, wird der bereits durchlaufene Weg soweit zurückgegangen (Backtracking), bis ein Knoten gefunden wird, von dem aus noch mindestens eine unbesuchte Kante abzweigt.

An jedem neu besuchten Knoten **v** wird **number** um **1** hochgezählt und **number** dem Knoten zugewiesen (**dfs_number[v] = ++number;**). Wenn die abgehenden Kanten eines Knoten von links ausgewählt werden, dann nennt man das Verfahren „linksrekursiv“.

Im Rahmen des Projektordners DFS ist in C# der Algorithmus der Tiefensuche wie folgt implementiert worden:

```
private double[,] am;   // Adjazenzmatrix des Graphen
private int[] dfs_num;  // DFS-Numerierung (fortlaufende Numerierung besuchter Knoten
private int n;          // Anzahl Knoten
private int number;     // Variable zum inkrementellen Hochzaehlen der Knoten

public Form1()
{
    n = 19;  // Anzahl Knoten ist 19

    // Instantiierung Matrizen und Vektoren
    am = new double[n, n];
    dfs_num = new int[n];

    // Initialisierung der Adjazenzmatrix
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            am[i, j] = -1; // keine Verbindung zwischen Knonten i und j
        }
    }

    // Graph
    am[0, 1] = 1;
    am[0, 2] = 1;
    am[1, 3] = 1;
    am[1, 4] = 1;
    am[3, 5] = 1;
    am[5, 6] = 1;
    am[5, 7] = 1;
    am[5, 8] = 1;
   am[8, 9] = 1;
    am[8, 10] = 1;
    am[10, 11] = 1;
    am[10, 12] = 1;
    am[13, 14] = 1;
    am[13, 15] = 1;
    am[14, 16] = 1;
   am[14, 17] = 1;
    am[16, 18] = 1;
    am[18, 8] = 1;
```

```
        //  am[11, 18] = 1;   // Zyklus
        }

void dfs(double [,] am, int v, int [] dfs_num, ref int number) // rekursive Tiefensuche
{
 dfs_num[v] = ++number;                // Knoten v besucht, Reihenfolgenummer incr.
 for(int i=0; i<n; i++)                // Alle Knoten i durchlaufen,
   if(am[v,i]>=0 && dfs_num[i]==0)     // die Nachfolger von v (Kante v->i) und nicht besucht
      dfs(am, i, dfs_num, ref number); // Tiefensuche dfs ab neu erreichten Knoten i rekursiv
}

private void button1_Click(object sender, EventArgs e)  // Start Tiefensuche DSF
{
    for (int i = 0; i < n; i++)
        dfs_num[i] = 0;                      // Vektor besuchter Knoten initialisieren

    number = 0;                              // Zaehler initialisieren

    for (int i = 0; i < n; i++)              // Alle Knoten i (Kreuzungen) durchlaufen
    {
        if (dfs_num[i] == 0)                 // Knoten i noch nicht besucht ?
            dfs(am, i, dfs_num, ref number); // Tiefensuche ab Knoten i
    }

    listBox1.Items.Add("Tiefensuche rekursiv");
    listBox1.Items.Add(String.Format("Knoten  Reihenfolge"));
    for (int i = 0; i < n; i++)
    {
         listBox1.Items.Add(String.Format("{0,6}       {1,6}", i, dfs_num[i]));

    }
}
```



8.1.2 Tiefensuche

Algorithmus (informell)

1. Bestimme den Knoten an dem die Suche beginnen soll
2. Expandiere den Knoten und speichere alle Nachfolger in einem Stack
3. Rufe rekursiv für jeden der Knoten in dem Stack DFS (depth first search oder Tiefensuche) auf. Falls der Stack leer sein sollte, tue nichts

Falls das gesuchte Element gefunden worden sein sollte, brich die Suche ab und liefere ein Ergebnis

Algorithmus (formal)

```
DFS(node, goal)    // (node = aktueller Knoten, goal = Zielknoten)
{
  if (node == goal)
    return node;
  else
  {
    stack := expand (node)
    while (stack is not empty)
    {
      node' := pop(stack);
      DFS(node', goal);
    }
  }
}
```

Algorithmusbeispiel: Erzeugen des Tiefensuchwaldes (rekursiv)

Der folgende rekursive Algorithmus erzeugt den Tiefensuchwald eines Graphen G mittels Setzen von Discovery- und Finishing-Times und Färben der Knoten. In Anlehnung an Cormen, Leiserson, Rivest, Stein, Introduction to Algorithms, MIT Press, 2001, werden zunächst alle Knoten weiß gefärbt. Anschließend startet die Tiefensuche per Definition beim alphabetisch kleinsten Knoten und färbt diesen grau. Danach wird, wie oben beschrieben rekursiv dessen weißer Nachbar betrachtet und grau gefärbt. Existiert kein weißer Nachbar mehr, kommt es zum Backtracking, während dessen alle durchwanderten Knoten schwarz gefärbt werden.

```
DFS(G)
1   for each v of G {        // Alle Knoten weiß färben, Vorgänger auf nil setzen
2      col[v] = 'w';
3      pi[v] = nil;
4   }
5   time = 0;
6   for each u of G          // Für alle weißen Knoten: DFS-visit aufrufen
7      if col[u] == 'w'
8         DFS-visit(u);

DFS-visit(u)
1   col[u] = 'g';            // Aktuellen Knoten grau färben
2   time++;                  // Zeitzähler erhöhen
3   d[u] = time;             // Entdeckzeit des aktuellen Knotens setzen
4   for each v of Adj[u] {   // Für alle weißen Nachbarn des aktuellen Knotens
5       if col[v] == 'w' {
6          pi[v] = u;         // Vorgänger auf aktuellen Knoten setzen
7          DFS-visit(v);      // DFS-visit aufrufen
8       }
9    }
10   col[u] = 's';           // Aktuellen Knoten schwarz färben
11   time++;
12   f[u] = time;            // Finishing-Time des aktuellen Knotens setzen
```

8.2 Iterative Tiefensuche

Manchmal möchte man Rekursionen vermeiden, z.B. weil bei jedem rekursiven Funktionsaufruf zusätzliche Zeit für das Anlegen von Variablen im Speicher etc. benötigt wird bzw. der Stack-Speicher überlaufen kann.

In diesem Fall kann die **Tiefensuche** mit Hilfe eines ***Stapels*** auch iterativ und ohne Rekursion programmiert werden. Unter einem **Stapel (Stack)** versteht man dabei eine Datenstruktur, die es erlaubt, Objekte (in unserem Fall Kreuzungen bzw. Knoten) oben auf den Stapel zu legen oder aber das momentan oberste Objekt vom Stapel zu nehmen. Der Stapel dient dazu, den "Rückweg" zu speichern, man legt also immer den Knoten *X*, den man gerade verlässt, oben auf den Stapel, und zwar zusammen mit einer Zahl "Ausgänge" bzw. Kanten, die angibt, zu wie vielen benachbarten Knoten man von *X* aus schon aufgebrochen ist.

Zu jedem Knoten existiert eine Liste (genauer: ein Array), in der alle Nachbarkreuzungen bzw. Nachbarknoten verzeichnet sind – somit kann man bei Bedarf gezielt beispielsweise den fünften Nachbarknoten eines Knotens auswählen.

```
Tiefensuche II (iterativ)
1 X := Startkreuzung; Ausgänge := 0;
2 repeat
3   if Zustand[X] ≠ „entdeckt“ then
4      if X = Ziel then exit „Ziel gefunden!“; endif
5      Zustand[X] := „entdeckt“;
6   else
7      nimm das oberste Paar (X, Ausgänge) vom Stapel;
8   endif
9   if Ausgänge < Anzahl benachbarter Kreuzungen von X then
10     Ausgänge := Ausgänge +1;
11     lege das Paar (X, Ausgänge) oben auf den Stapel;
12     X := (Ausgänge)-te benachbarte Kreuzung von X;
13     Ausgänge := 0;
14   else
15     if Stapel ist leer then
16        exit „Ziel nicht gefunden!“;
17     else
18        nimm das oberste Paar (X, Ausgänge) vom Stapel;
19        gehe zu Zeile 9;
20     endif
21   endif
22 end repeat
```

Iterative Tiefensuche (DFS iterativ)

Alternativ zur rekursiven Tiefensuche kann eine iterative Tiefensuche implementiert werden.

Für jeden Knoten, den wir das erste Mal besuchen, notieren wir uns alle seine Nachbarknoten in einem Notizbuch. Der nächste zu besuchende Knoten ist der zuletzt im Notizbuch eingetragene, den wir zuerst streichen.

Zu beachten ist, daß ein Knoten mehrfach im Notizbuch stehen kann, da er Nachbar mehrerer bereits besuchter Knoten gewesen sein kann. Daher müssen wir für jeden zu besuchenden Knoten erst überprüfen, ob er nicht bereits besucht wurde.

Falls ja, fahren wir mit dem nächsten Knoten vom Ende unseres Notizbuches fort.

Ein solches Notizbuch läßt sich mit Hilfe eines Kellers (stack) realisieren. Ein Keller ist eine Datenstruktur, auf die nach dem Prinzip „last in – first out“ (LIFO) zugegriffen wird.

Folgende Operationen werden u.a. über einem Stack unterstützt:

s.Push(v) – fügt das Element v als oberstes, letztes Element dem Stack s hinzu

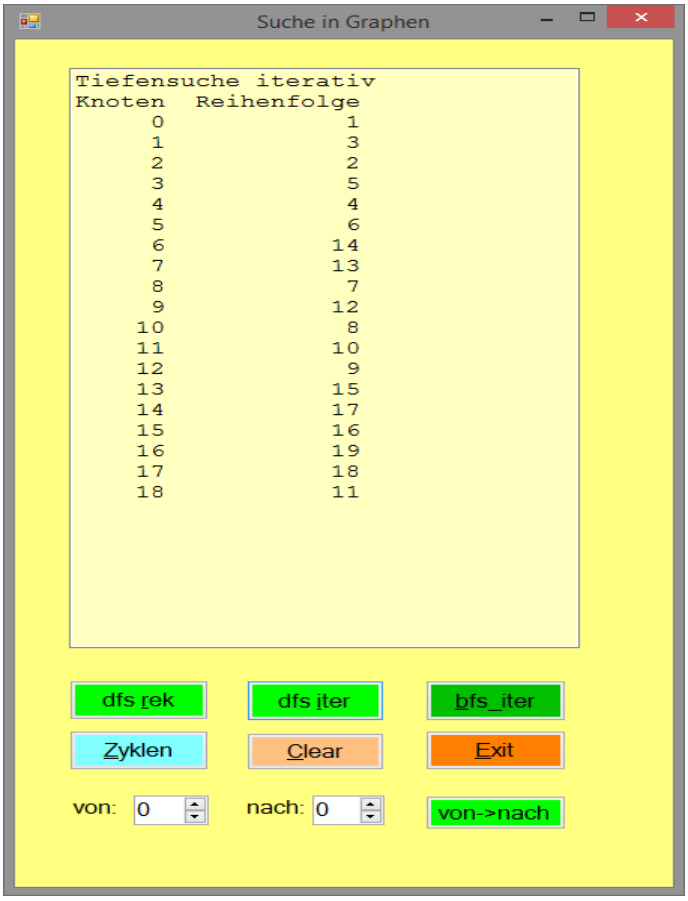
v = s.Pop() – entfernt das zuletzt hinzugefügte Element v von s und liefert es zurück

s.Count – liefert die Anzahl der im Stack s enthaltenen Elemente

Im Rahmen des Projektordners **DFS** ist in C# der Algorithmus der iterativen Tiefensuche wie folgt implementiert worden:

```
void dfs_iter(double[,] am, int v, int[] dfs_num, ref int number) // iterative Tiefensuche
{
            Stack <int> s = new Stack<int>();    // neuen Stack s anlegen
            s.Push(v);                           // v als Element nach s
            while(s.Count > 0)                   // Solange s nicht leer
            {
                v = s.Pop();                     // v als oberstes Element von s entnehmen
                if(dfs_num[v] == 0)              // wenn v nicht besucht
                {
                    dfs_num[v] = ++number;       // Knoten v besucht, Reihenfolgenummer incr.
                    for (int i = 0; i < n; i++)  // Alle Knoten i durchlaufen,
                        if (am[v, i] >= 0)       // die Nachfolger i von v (Kante v->i)
                            s.Push(i);           // i als Element nach s
                }
            }
 }
```

Die folgende Abbildung zeigt das Ergebnis der iterativen Tiefensuche. Gegenüber der vorhergehenden rekursiven Tiefensuche hat sich die Reihenfolge des Durchlaufens der einzelnen Knoten verändert.



8.3 Iterative Breitensuche (BFS iterativ)

Bei der Tiefensuche versucht man immer, erst einmal so tief wie möglich in das Labyrinth vorzustoßen. Im Falle der **Breitensuche** oder **BFS** (breadth-first search) durchläuft man die Knoten in der Reihenfolge des Auffindens.

Trifft man auf einen noch nicht besuchten Knoten, so notiert man sich die von dort erreichbaren Knoten in einem Notizbuch. Der nächste zu besuchende Knoten ist dann der Knoten, der am längsten im Notizbuch steht (FIFO) und noch nicht besucht wurde. Nach dem Besuch eines Knotens streicht man diesen aus dem Notizbuch.

Analog zur **DFS**-Numerierung bei der Tiefensuche wird hier die **BFS-Numerierung** für die Breitensuche verwendet.

Beim Speichern der noch zu besuchenden Knoten wird hier die Strategie **FIFO** (first in – first out) genutzt, die durch eine **Warteschlange (Queue)** realisiert wird.

Folgende Operationen werden u.a. von einer Warteschlange unterstützt:

s.Enqueue(v); Einfügen von v als letztes Element in die Queue s

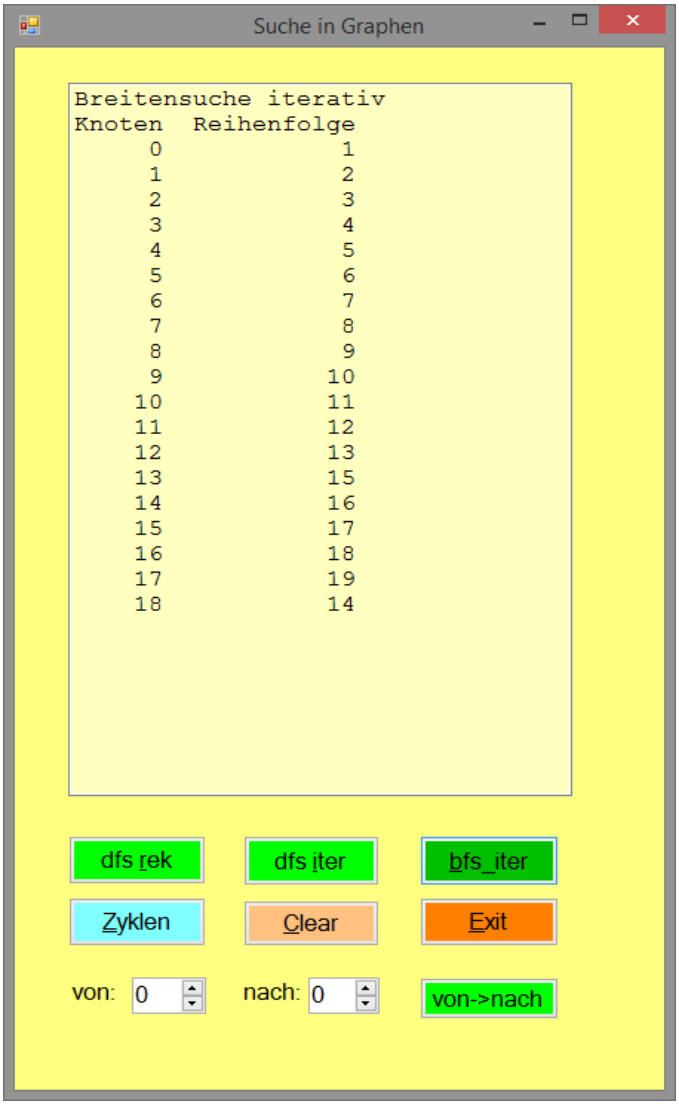
v = s.Dequeue(); Entfernen von v vom Anfang der Queue s

s.Count Anzahl der Elemente in der Queue s

```
void bfs_iter(double[,] am, int v, int[] bfs_num, ref int number) // iterative Breitensuche
{
       Queue<int> s = new Queue<int>();         // neue Queue s anlegen
       s.Enqueue(v);                            // v als Element nach s
       while (s.Count > 0)                      // Solange s nicht leer
       {
           v = s.Dequeue();                     // v als erstes Element von s entnehmen
           if (bfs_num[v] == 0)                 // wenn v nicht besucht
           {
               bfs_num[v] = ++number;           // Knoten v besucht, Reihenfolgenummer incr.
               for (int i = 0; i < n; i++)      // Alle Knoten i durchlaufen,
                   if (am[v, i] >= 0)           // die Nachfolger i von v (Kante v->i)
                       s.Enqueue(i);            // i in Queue s einfuegen
           }
       }
}
```

Zu beachten ist, daß der Vektor **bfs_num** wie der Vektor **dfs_num** vereinbart und behandelt wird.

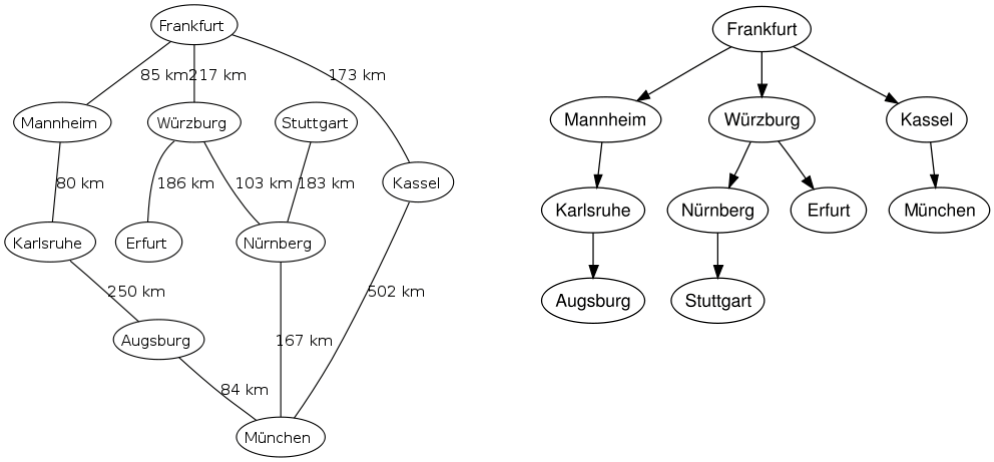
Die Reihenfolge des Durchlaufens der Knoten hat sich gegenüber der rekursiven und auch der iterativen Tiefensuche wieder geändert



8.3.1 Breitensuche allgemein

Die Breitensuche ist eine uninformierte Suche, welche durch Expansion der einzelnen *Level* der Graphen ausgehend vom Startknoten den Graph *in die Breite* nach einem Element durchsucht.

Zuerst wird ein Startknoten u ausgewählt. Von diesem Knoten aus wird nun jede Kante (u,v) betrachtet und getestet, ob der gegenüberliegende Knoten v schon entdeckt wurde bzw. das gesuchte Element ist. Ist dies noch nicht der Fall, so wird der entsprechende Knoten in einer Warteschlange gespeichert und im nächsten Schritt bearbeitet. Hierbei ist zu beachten, dass Breitensuche immer zuerst alle direkt nachfolgenden Knoten bearbeitet, und nicht wie die Tiefensuche einem Pfad in die Tiefe folgt. Nachdem alle Kanten des Ausgangsknotens betrachtet wurden, wird der erste Knoten der Warteschlange entnommen und das Verfahren wiederholt.



Eine Landkarte von Deutschland mit den Verbindungen zwischen einigen Städten und der Baum, welcher entsteht, wenn man BFS auf die Landkarte anwendet und in Frankfurt startet.

Algorithmus (informell)

1. Bestimme den Knoten, an dem die Suche beginnen soll, und speichere ihn in einer Warteschlange ab.
2. Entnimm einen Knoten vom Beginn der Warteschlange und markiere ihn.
   Falls das gesuchte Element gefunden wurde, brich die Suche ab und liefere "gefunden" zurück.
   Anderenfalls hänge alle bisher unmarkierten Nachfolger dieses Knotens, die sich noch nicht in der Warteschlange befinden, ans Ende der Warteschlange an.
3. Wiederhole Schritt 2.

Wenn die Warteschlange leer ist, dann wurde jeder Knoten bereits untersucht.
Beende die Suche und liefere "nicht gefunden" zurück.

Breitensuche (breadth first search - BFS)

Beim Eintreffen an einem Knoten $\mathbf{u} \in \mathbf{V}$ werden zunächst alle von **u** ausgehenden Kanten erforscht, bevor bei einem Nachfolger von **u** weitergesucht wird.

Der **Abstand dist(s,v)** des Knoten $\mathbf{v} \in \mathbf{V}$ von $\mathbf{s} \in \mathbf{V}$ aus wird als die **Länge |P|** (Anzahl der Kanten) eines **kürzesten Weges P** von **s** nach **v** in **G** definiert. Falls $\mathbf{v} \notin \mathbf{E_G(s)}$, dann **dist(s,v):=+∞**

Das Verfahren der Breitensuche (**BFS**) berechnet für einen Knoten $\mathbf{s} \in \mathbf{V}$ die Knoten $\mathbf{V_i} \subseteq \mathbf{V}$ mit Abstand **i** von **s** durch folgende einfache Rekursion:

$\mathbf{V_0} := \{ \mathbf{s} \}$
$\mathbf{V_{i+1}} := \{ \mathbf{v} \in \mathbf{V} \setminus (\mathbf{V_0} \cup \ldots \cup \mathbf{V_i}) : \mathbf{v} \in \mathbf{N^+(V_i)} \}$

Hierbei ist $\mathbf{N^+(V_i)}$ die Menge aller Nachfolger der Menge $\mathbf{V_i}$ (im ungerichteten Fall ist $\mathbf{N^+}$ durch **N** zu ersetzen).

Die Korrektheit der Rekursion ist durch Induktion zu sehen:
Ist $\mathbf{v} \in \mathbf{V_i}$, so existiert ein Weg der Länge **i+1** von **s** nach **v**, der über den Vorgängerknoten von **v** in $\mathbf{V_i}$ führt. Ist umgekehrt **dist(s, w) = i > 0**, so besitzt **w** auf dem kürzesten Weg von **s** nach **w** einen Vorgänger **u**, welcher per Induktion korrekt nach $\mathbf{V_i}$ eingeordnet wird.

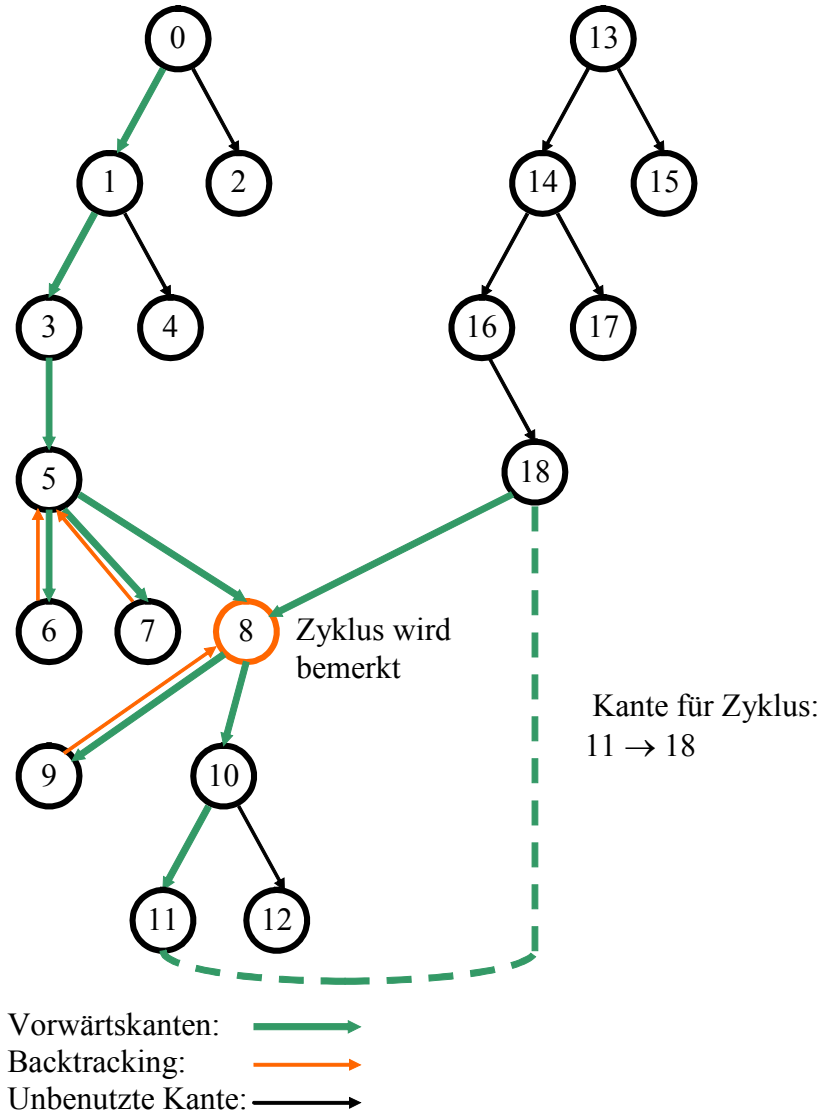
Im folgenden BFS-Algorithmus entspricht die Menge $\mathbf{V_0} \cup \ldots \cup \mathbf{V_i}$ der Menge derjenigen Knoten $\mathbf{v} \in \mathbf{V}$, welche **d[v]** = +∞ haben.

```
BFS (G, s) {
  Input:   Graph G (un)gerichtet in Adjazenzlistendarstellung; ein Knoten s ∈ V(G)
  Output: Für jeden Knoten v∈V der Abstand dist(s,v) von s zu v

  for each v∈V do {
     d[v] := +∞;
  }

  d[s] := 0;
  Q := {s};

  while (Q ≠ ∅)  do {
     Entferne das Element u aus Q mit kleinstem Schlüsselwert d[u]
     for each v∈ADJ[u] do {              // alle v mit ADJ[u][v] ≠ 0
        if (d[v] == +∞) {
            d[v] := d[u] + 1;
            Q := Q ∪ {v};
        }
     }
  }
}
```

8.4 Zyklensuche mit rekursiver Tiefensuche

Im Falle von **Zyklen** erreicht der Algorithmus wiederholt einen bereits durchlaufenen Knoten in Vorwärtsrichtung. Das folgende Beispiel mit der neuen Kante **11 -> 18** verdeutlicht diesen Fall.

Beispiel: Start in Knoten **0** und **13** mit **neuer Kante 11 -> 18**



Die Kante **am[11, 18] = 1** wurde zusätzlich in die Adjazenzmatrix eingefügt.

Die Methode **cycle** besitzt gegenüber **dfs** den zusätzlichen Parameter **out bool cyclus**. Im Falle des Erreichens eines Nachfolgers, der bereits früher besucht wurde, besteht mindestens ein Zyklus im Graphen. Der Parameter **cyclus** wird auf **true** gesetzt und die Suche wird beendet.

Falls in **button3_Click** die Anweisung **if(cyclus == true) break;** entfernt würde, werden alle weiteren Zyklen entdeckt.

```
private void button3_Click(object sender, EventArgs e) // Start Zyklensuche
{
    for (int i = 0; i < n; i++)
                dfs_num[i] = 0;                    // Vektor besuchter Knoten initialisieren

    number = 0;                                    // Zaehler initialisieren
    bool cyclus;

    for (int i = 0; i < n; i++)                    // Alle Knoten i (Kreuzungen) durchlaufen
    {
        if (dfs_num[i] == 0)                       // Knoten i noch nicht besucht ?
        {
            cycle(am, i, dfs_num, ref number, out cyclus); // Zyklensuche ab Knoten i
            if(cyclus) break;                      // Zyklus gefunden, Abbruch beim 1.Zyklus
        }
    }
}

void cycle(double [,] am, int v, int [] dfs_num, ref int number, out bool cyclus)
{
    cyclus = false;
    dfs_num[v] = ++number;                    // Knoten v besucht, Reihenfolgenummer incr.
    for (int i = 0; i < n; i++)               // Alle Knoten i durchlaufen
    {
         if (cyclus == true) break;
         if (am[v, i] >= 0)                    // Ex. Kante v->i ?
             if (dfs_num[i] == 0)              // Nachfolger i noch nicht besucht ?
               cycle(am, i, dfs_num, ref number, out cyclus); // Zyklensuche cycle ab Knoten i
             else
             {
               listBox1.Items.Add(String.Format("Zyklus von {0} nach {1}", v, i));
               cyclus = true;                  // Zyklus, Nachfolger i besucht
             }
     }
}
```

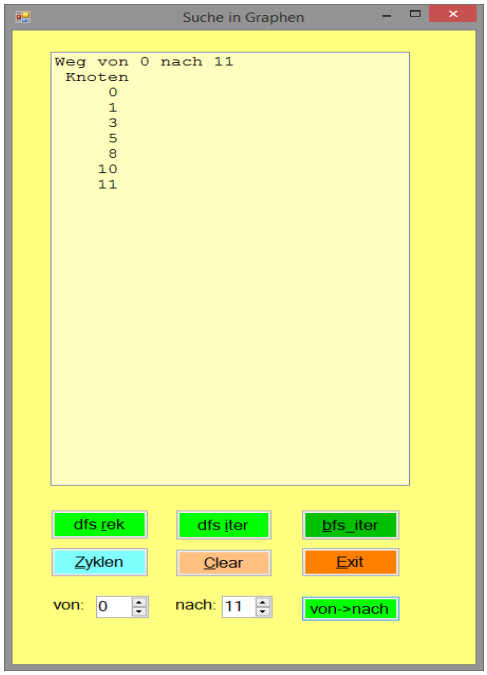
8.5 Wegesuche mit rekursiver Tiefensuche

Die Wegesuche ermittelt, ob wenigstens ein Weg von einem Start- nach einem Zielknoten existiert. Das ist nicht notwendig der kürzeste Weg.

Im Falle von **found = true** werden beim Backtracking des rekursiven Algorithmus der für jede Aufrufebene gewählte Knoten in einem **Stack** gespeichert. Dadurch kann die Methode **button7_Click** die durchlaufenen Knoten von der Quelle zum Ziel in der durchlaufenen Reihenfolge ausgeben.

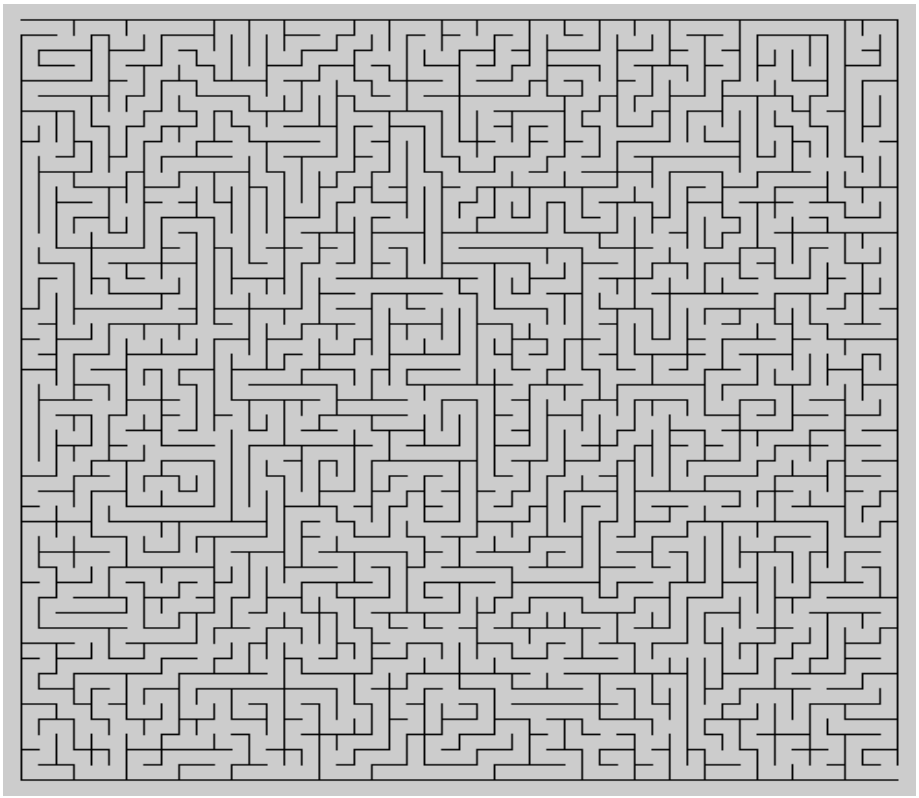
```
private void button7_Click(object sender, EventArgs e) // Start Weguche von -> nach
{
    for (int i = 0; i < n; i++)
        dfs_num[i] = 0;                          // Vektor besuchter Knoten initialisieren

    int von = (int)numericUpDown1.Value;         // Knoten von
    int nach = (int)numericUpDown2.Value;        // Knoten nach
    bool found = false;

    if (von == nach)
    {
        listBox1.Items.Add(String.Format("von und nach sollten unterschiedlich sein"));
                return;
    }

    Stack<int> s = new Stack<int>();       // neuen Stack s anlegen

    dfs_von_nach(am, von, nach, dfs_num, ref found, s);    // Tiefensuche ab Knoten von

    if(found == false)
    {
        listBox1.Items.Add(String.Format("Kein Weg von {0} nach {1}", von, nach));
        return;
    }

    listBox1.Items.Add(String.Format("Weg von {0} nach {1}", von, nach));
    listBox1.Items.Add(String.Format(" Knoten"));

    s.Push(von);

    while(s.Count>0)
    {
        int i = s.Pop();
        listBox1.Items.Add(String.Format("{0,6}", i));
    }
}
```

```
void dfs_von_nach(double[,] am, int von,int nach,int[] dfs_num,ref bool found, Stack<int> s)
{
  dfs_num[von] = 1;                          // Knoten v besucht, Reihenfolgenummer incr.

  for (int i = 0; i < n; i++)                // Alle Knoten i durchlaufen,
    if (am[von, i] >= 0 && dfs_num[i] == 0) // Nachfolger von (Kante v->i) und nicht besucht
    {
        if (i == nach)                        // Ziel nach gefunden
        {
            s.Push(i);
            dfs_num[nach] = 1;
            found = true;                     // Weg gefunden
            break;                            // for-Schleife verlassen
        }

        dfs_von_nach(am, i, nach, dfs_num, ref found,s); // rekursiver Aufruf

        if (found)                            // Weg gefunden ?
        {
            s.Push(i);                        // lokales i nach s
            break;                            // for-Schleife verlassen
        }
    }
 }
```



8.6 Erstellen von Labyrinthen

Man startet mit einem regelmäßigen rechtwinkligen Gitter. Die Tiefensuche startet mit einer beliebigen, frei wählbaren Zelle. Dann wird die Tiefensuche für alle Nachbarzellen in **zufälliger Reihenfolge** gerufen. Wird dabei eine Zelle zum ersten Mal besucht, so wird die Wand zur Vorgängerzelle eingerissen. Damit ergibt sich ein Muster wie das folgende:



Da die Tiefensuche letztlich jede Zelle besucht, muß es von jeder Zelle einen Weg zur Startzelle geben und somit auch von jeder Zelle zu jeder anderen.

8.7 Wegesuche über deduktive SQL-Erweiterungen

Die Wegsuche vom Knoten x zum Knoten y kann auch über Deduktionsregeln gelöst werden. Definiert werden kann eine Relation für eine gerichtete Kante:
kante = ({**kvon**, **knach**,...})

Mit folgenden Regeln können alle Paare von Knoten bestimmt werden, für die es einen Weg vom Knoten x zum Knoten y gibt.

```
Regeln: kante(x,y) -> weg(x,y)
        weg(x,z) ^ weg(z,y) -> weg(x,y)
```

Äquivalent können diese Regeln auch als PROLOG-Klauseln formuliert werden:

```
PROLOG-Klausel: weg(x,y) :- kante(x,y)
                weg(x,y) :- weg(x,z), weg(z,y)
```

Neuere Datenbanken, wie DB2 UDB mit einer erweiterter SQL-Version (SQL3) erlauben die deduktive und rekursive Auswertung von SQL-Ausdrücken. Die beiden genannten Regeln bzw. PROLOG-Ausdrücke lassen sich dort wie folgt formulieren:

SQL3:

```
WITH weg ( kvon, knach ) AS
     (( SELECT kvon, knach FROM tbl_kante)
     UNION
     (SELECT P1.kvon, P2.knach FROM weg P1, weg P2 WHERE
     P1.knach = P2.kvon))
SELECT * FROM weg
```

Beispielsweise lassen sich alle Paare von Knoten, für die es einen Weg vom Knoten 1 zum Knoten 8 gibt, in dieser SQL-Version wie folgt bestimmen:

```
WITH weg ( kvon, knach ) AS
     (( SELECT kvon, knach FROM tbl_kante WHERE kvon = 1 )
      UNION
     (SELECT P1.kvon, P2.knach FROM weg P1, weg P2 WHERE
      P1.knach = P2.kvon))
SELECT * FROM weg WHERE knach = 8
```

8.8 Wege aus dem Labyrinth (rekursive und iterative Tiefensuche)

Ein **Labyrinth** besteht aus Kreuzungen (**Knoten**) und Gängen (**Kanten**) und wird vorteilhaft als **ungerichteter Graph** abgebildet. Ein ungerichteter Graph ist **symmetrisch bzgl. der Hauptdiagonalen** der **Adjazenzmatrix** und kann auch als **spezieller gerichteter Graph** interpretiert werden, wobei für jede Kante vom Knoten **x** nach **y** auch immer die Kante von **y** nach **x** existiert.

Algorithmus rekursiv:
An jeder Kreuzung markieren wir die abgehenden Gänge, und zwar mit einem Haken für bereits einmal durchlaufene Gänge und mit zwei Haken für zweimal durchlaufene. Konkret lauten die Regeln für unsere Suche im Labyrinth folgendermaßen.

• Befindet man sich in einer Sackgasse, so dreht man um und geht zurück zur letzten Kreuzung.

• Hat man dagegen eine Kreuzung erreicht, zeichnet man erst einmal seinen Haken an die Wand des Ganges, durch den man gekommen ist, um später ggf. wieder zurückfinden zu können. Anschließend gibt es mehrere Möglichkeiten:

1. Zunächst kontrolliert man, ob man im **Kreis** gelaufen ist:
Wenn der Gang, durch den man gekommen ist, soeben seinen ersten Haken bekommen hat und wenn außerdem noch weitere Haken an anderen Gängen der Kreuzung sichtbar sind, so ist dies der Fall, und man macht einen zweiten Haken an den Gang, aus dem man kam, und dreht um.

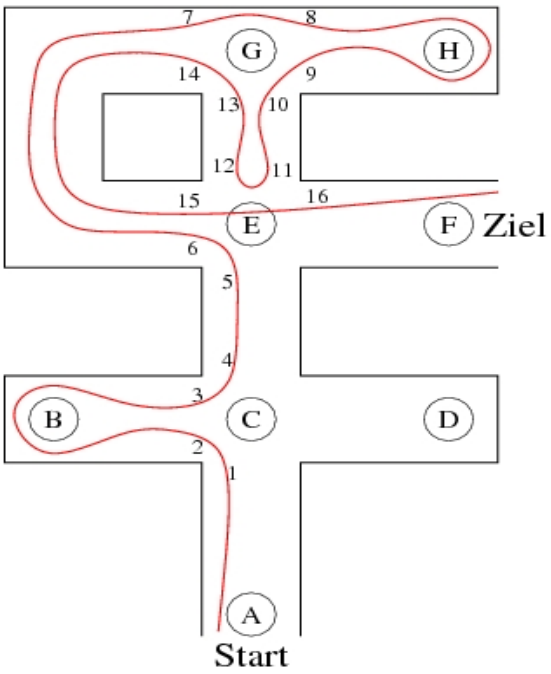
2. Ansonsten prüft man, ob die Kreuzung noch unerkundete Gänge hat:
Falls es noch Gänge ohne Markierungen gibt, so wählt man von diesen einen beliebigen – sagen wir, den ersten von links – aus, zeichnet dort einen Haken an die Wand und verlässt die Kreuzung durch diesen Gang. (Dieser Fall gilt übrigens auch zu Beginn der Suche an der Startkreuzung!)

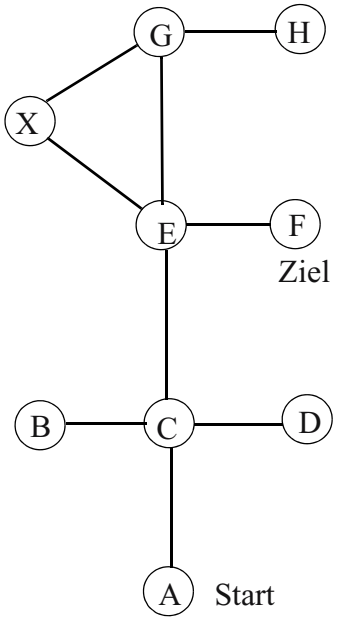
3. Andernfalls gibt es höchstens einen Gang mit nur einem Haken, und alle anderen Gänge haben zwei Haken. Man hat also bereits alle von der aktuellen Kreuzung abgehenden Gänge untersucht und geht durch den Gang zurück, der nur einen Haken hat, wobei man diesem Gang der Ordnung halber einen zweiten Haken verpasst. Gibt es gar keinen solchen Gang,
d. h., haben gar alle Gänge zwei Haken, so steht man wieder am Start und hat das Labyrinth vollständig durchsucht.

Die Regeln für die Tiefensuche lauten wie folgt:
Man merkt sich für jede Kreuzung einen "**Zustand**", wobei am Anfang alle Kreuzungen "**unentdeckt**" sind. Wird die Tiefensuche-Funktion an einer Kreuzung ***X*** gestartet, so wird zunächst getestet, ob man im Kreis gelaufen ist (Zeile 2). Als nächstes wird geschaut, ob man das Ziel gefunden hat (Zeile 3) – falls ja, so wird das Programm mit dem Befehl "**exit**" beendet, und die Suche ist zu Ende. Andernfalls geht es weiter, und die Kreuzung ***X*** wird als „**entdeckt**“ markiert (Zeile 4). Nun müssen alle noch nicht erkundeten benachbarten Kreuzungen besucht werden – dies geschieht, indem sich die Tiefensuche-Funktion für jede benachbarte Kreuzung ***Y*** selbst aufruft (Zeilen 5–7). Dies ist ein rekursiver Aufruf.
Bemerkt die aufgerufene Tiefensuche-Funktion, dass ***Y*** schon besucht wurde und man somit im Kreis gelaufen ist, so kehrt sie sofort (Zeile 2) per "**return**" zur aufrufenden Funktion an der Kreuzung ***X*** zurück. Andernfalls geht es nun an der Kreuzung ***Y*** weiter. . .

```
Tiefensuche I (rekursiv)
1 function Tiefensuche(X):    // Def. der Tiefensuche-Funktion
2     if Zustand[X] = „entdeckt“ then return; endif
3     if X = Ziel then exit „Ziel gefunden!“; endif
4     Zustand[X] := „entdeckt“;
5     for each benachbarte Kreuzung Y von X
6          Tiefensuche(Y);
7     end for
8 end function                // Ende der Tiefensuche-Funktion
9 Tiefensuche(Startkreuzung); // Hauptprogramm
```

Das folgende Labyrinth kann über den nebenstehenden Graphen abgebildet werden. Um mehrfache Kanten zu vermeiden, wurde der Hilfsknoten X eingeführt. Um die im Graphen angegebene Kantenfolge zu simulieren, müßte im Falle einer linksrekursiven Tiefensuche Knoten B der erste Nachfolger von C und X der erste Nachfolger von E sein. Zusätzlich würde noch E nach G getestet, falls G der zweite Nachbarknoten von E wäre.





8.9 Breitensuche

Ein Problem bei der Tiefensuche ist, dass man sich schnell sehr weit vom Start entfernen kann. In vielen Fällen weiß man aber, dass das Ziel nicht allzuweit weg ist. In vielen Fällen bietet sich die **Breitensuche** (auf englisch ***breadth-first search***) an:

Sie sucht den Graphen um den Ausgangspunkt schichtenweise ab, also **erst alle direkten Nachbarn** (Abstand 1), dann alle Knoten im Abstand 2 usw. Dazu benutzt man die Datenstruktur "**Warteschlange**" (**Queue**), in die man einen Knoten hinten einstellen kann und aus der man vorne einen Knoten herausnehmen kann, zu dem man dann springt.

Für die Labyrinthsuche ist Breitensuche also nicht geeignet: Man kann nicht einfach eine Kreuzung auf einer Liste notieren und dann bei Bedarf dort "hinspringen". Für viele andere Anwendungen (wie die Web-Suche) ist das jedoch kein Problem.

Das unten angegebene Programmstück zeigt die Breitensuche im Detail. Dabei werden in der Warteschlange immer die Knoten gespeichert, die noch besucht werden müssen. Am Anfang kommt also der Startknoten in die **Warteschlange** (Zeile 2). Solange die **Warteschlange** nicht leer ist, wird immer der erste Knoten herausgenommen (Zeilen 3 und 4). Dann werden alle Nachbarn dieses Knotens in die **Warteschlange** eingefügt (Zeilen 8 und 9). Damit wir von einem Knoten nicht mehrmals die Nachbarn absuchen, wird ein so behandelter Knoten als "entdeckt" markiert (Zeile 7), und bereits markierte Knoten werden übersprungen (Zeile 5).

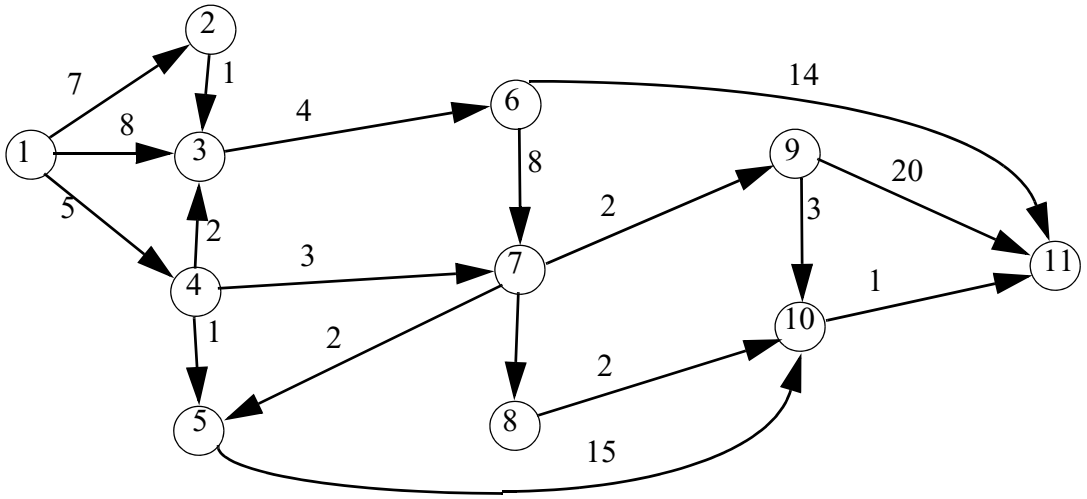
```
Breitensuche (iterativ)
1 begin                              // am Anfang ist die Warteschlange leer
2    stelle den Startknoten hinten in die Warteschlange;
3    while Warteschlange ist nicht leer
4       nimm den vordersten Knoten X aus der Warteschlange;
5       if Zustand[X] ≠ „entdeckt“ then
6          if X = Ziel then exit „Ziel gefunden!“; endif
7             Zustand[X] := „entdeckt“;
8             for each benachbarter Knoten Y von X
9                 stelle Y hinten in die Warteschlange;
10            end for
11         endif
12   end while
13 end
```

8.10 Zyklensuche für gerichtete Graphen

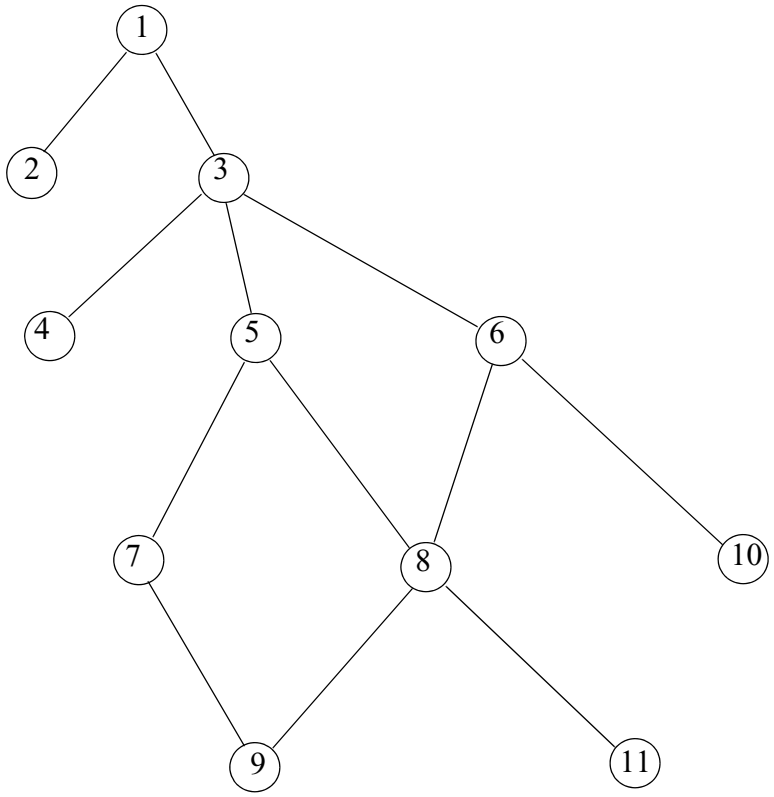
```
Tiefensuche gerichtete Graphen
1  procedure Tiefensuche (Knoten x)
2  begin
3    if Ziel erreicht then stop
4    else if x unmarkiert then
5            markiere x;
6            for all Nachfolgerknoten y von x do
7                Tiefensuche(y)
8            endfor
9    endif
10 end

Zyklensuche gerichtete Graphen
1  procedure Zyklensuche (Knoten x)
2  begin
3    if Markierung(x) = „in Bearbeitung“ then Zyklus gefunden
4    else if Markierung(x) = „noch nicht begonnen“ then
5            Markierung(x) := „in Bearbeitung“;
6            for all Nachfolgerknoten y von x do
7                Zyklensuche(y)
8            endfor;
9            Markierung(x) := „abgeschlossen“
10   endif
11 end

Breitensuche gerichtete Graphen
1  procedure Breitensuche (Knoten x)
2  begin
3    Erreichbar := {}; Front := {x};
4    repeat
5      Front := {y | y ist Nachfolgerknoten irgendeines z aus
                    Front und y ist nicht in Erreichbar }
6      if x aus Front then Zyklus von x nach x existiert, stop;
7      endif ;
8      Erreichbar := Erreichbar vereinigt mit Front;
9    until Front = {}
10 end
```

Beispielgraphen

Graph 1



Graph 2



Graph 3