9.4 Breitensuche und Tiefensuche

LK

Die erste Frage an fast jeden Graphen lautet: *Welche Knoten erreiche ich von hier aus — und in welcher Reihenfolge schaue ich sie an?* Darauf gibt es zwei grundverschiedene Antworten.

9.4.1 Die Breitensuche

Die **Breitensuche** (englisch *breadth-first search*, BFS) arbeitet in Ringen: erst alle direkten Nachbarn, dann deren Nachbarn, dann die nächste Schale. Dafür braucht sie eine **Warteschlange** (Queue): Neu entdeckte Knoten werden hinten angehängt und vorne wieder entnommen — wer zuerst kommt, mahlt zuerst.

```python
def breitensuche(graph, start):
    besucht = [start]
    schlange = [start]
    while schlange:
        knoten = schlange.pop(0)          # vorne entnehmen: Queue
        for nachbar in graph[knoten]:
            if nachbar not in besucht:
                besucht.append(nachbar)
                schlange.append(nachbar)
    return besucht
```

```pascal
procedure Breitensuche(start: Integer);
var
  besucht: array [1..N] of Boolean;
  schlange: array [1..N] of Integer;
  kopf, ende, knoten, n: Integer;
begin
  for n := 1 to N do
    besucht[n] := False;
  kopf := 1;
  ende := 1;
  schlange[1] := start;
  besucht[start] := True;
  while kopf <= ende do
  begin
    knoten := schlange[kopf];
    Inc(kopf);
    Write(ORTE[knoten], ' ');
    for n := 1 to N do
      if (ENTFERNUNG[knoten, n] > 0) and not besucht[n] then
      begin
        besucht[n] := True;
        Inc(ende);
        schlange[ende] := n;
      end;
  end;
  WriteLn;
end;
```

9.4.2 Die Tiefensuche

Die **Tiefensuche** (*depth-first search*, DFS) macht das Gegenteil: Sie folgt einem Weg, so weit es geht, und kehrt erst um, wenn es nicht mehr weitergeht. Dafür braucht sie einen **Stapel** (Stack) — oder, viel bequemer, die Rekursion aus Kapitel 3, die ihren Stapel selbst mitbringt:

```python
def tiefensuche(graph, knoten, besucht=None):
    if besucht is None:
        besucht = []
    besucht.append(knoten)
    for nachbar in graph[knoten]:
        if nachbar not in besucht:
            tiefensuche(graph, nachbar, besucht)
    return besucht
```

```
var
  besucht: array [1..N] of Boolean;

procedure Tiefensuche(knoten: Integer);
var
  n: Integer;
begin
  besucht[knoten] := True;
  Write(ORTE[knoten], ' ');
  for n := 1 to N do
    if (ENTFERNUNG[knoten, n] > 0) and not besucht[n] then
      Tiefensuche(n);
end;
```

Gymnasium Ottweiler

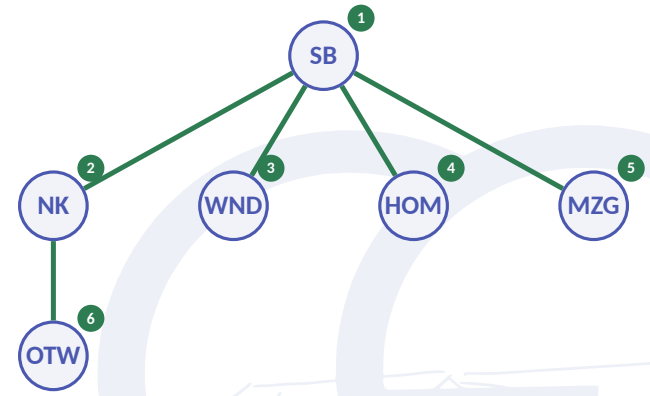
Stolperstein

Der Haken `if nachbar not in besucht` ist nicht Beiwerk, sondern lebenswichtig. Ohne ihn läuft die Suche in jedem Graphen mit einem Kreis — also in fast jedem — endlos im Kreis herum. Bei Bäumen aus Kapitel 3 brauchte man ihn nicht, weil ein Baum per Definition keine Kreise hat. Das ist der eine wesentliche Unterschied zwischen dem Durchlaufen eines Baums und dem eines Graphen.

9.4.3 Was die beiden unterscheidet

Beide Verfahren besuchen am Ende dieselben Knoten — alle erreichbaren. Die Reihenfolge ist aber eine völlig andere, und die Knoten, über die sie dabei zum ersten Mal entdeckt werden, bilden jeweils einen Baum, den **Suchbaum**:

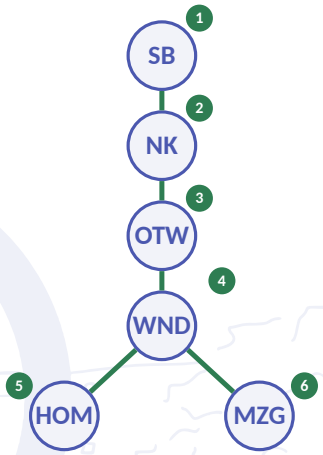
Breitensuche: erst alles Nahe



SB, NK, WND, HOM, MZG, OTW

Tiefe 2, Breite 4 — kürzeste Wege nach Kantenzahl

Tiefensuche: erst so weit wie möglich



SB, NK, OTW, WND, HOM, MZG

Tiefe 4, Breite 2 — ein langer Pfad

Beide Suchen starten in Saarbrücken und besuchen dieselben sechs Orte — in verschiedener Reihenfolge und mit verschieden geformten Suchbäumen.

	Breitensuche	**Tiefensuche**
Hilfsstruktur	Warteschlange (Queue)	Stapel (Stack) oder Rekursion
Suchbaum	breit und flach	schmal und tief
findet zuerst	Knoten mit den wenigsten Kanten Abstand	irgendeinen Knoten, möglichst weit weg
Speicherbedarf	kann riesig werden (eine ganze Schale)	nur der aktuelle Pfad
typische Aufgabe	kürzester Weg nach Kantenzahl, Nachbarschaft	Kreise finden, Labyrinthe, Zusammenhang prüfen

Im Bedienfeld lassen sich beide Suchen — und die beiden Algorithmen der nächsten Abschnitte — auf genau diesem Graphen Schritt für Schritt verfolgen. Grün heißt fertig, Blau heißt gerade in Arbeit.

> In der Webfassung steht hier ein Bedienfeld zum Ausprobieren: **Graph-Werkzeug**.

Merke

Die Breitensuche findet den Weg mit den **wenigsten Kanten** — aber nicht den kürzesten in Kilometern. Sie zählt Stationen, nicht Gewichte.

Im Saarland-Graphen sieht man das an Merzig und Neunkirchen: Die Breitensuche liefert MZG → SB → NK mit zwei Kanten, und das sind $45 + 24 = 69$ km. Der kürzeste Weg ist aber MZG → WND → OTW → NK mit drei Kanten und nur 66 km. Für gewichtete Graphen braucht es deshalb den Algorithmus aus dem nächsten Abschnitt.

Nebenbei beantworten beide Suchen eine Frage aus Abschnitt 9.2.2 ganz von selbst: Besucht die Suche von einem Knoten aus *alle* Knoten, dann ist der Graph zusammenhängend.

Arbeitsauftrag 4

17. Führen Sie Breiten- und Tiefensuche auf dem Saarland-Graphen ab Ottweiler aus. Notieren Sie beide Reihenfolgen und zeichnen Sie beide Suchbäume.
18. Beide Verfahren hängen davon ab, in welcher Reihenfolge die Nachbarn aufgelistet sind. Geben Sie für die Tiefensuche ab Saarbrücken eine Nachbarreihenfolge an, bei der Merzig als zweiter Ort besucht wird.
19. Erklären Sie, warum die Breitensuche eine Warteschlange und die Tiefensuche einen Stapel braucht — und was passiert, wenn man die beiden vertauscht.
20. Schreiben Sie die Tiefensuche ohne Rekursion, mit einer Liste als Stapel.
21. Erweitern Sie die Breitensuche so, dass sie zu jedem Knoten zusätzlich seinen Abstand in Kanten vom Startknoten ausgibt.
22. Schreiben Sie eine Funktion `zusammenhaengend(graph)`, die mit einer einzigen Suche prüft, ob der Graph zusammenhängend ist.
23. In einem Labyrinth sucht ein Roboter den Ausgang. Welches der beiden Verfahren würden Sie wählen, wenn der Speicher knapp ist, und welches, wenn der Ausgang garantiert nah liegt? Begründen Sie.

Gymnasium Ottweiler