Originally published in *Taschenbuch der Algorithmen*, *eXamen.press*, pages 61–73. Springer, 2008.

Tiefensuche (Ariadne und Co.)

Michael Dom, Falk Hüffner, Rolf Niedermeier

Institut für Informatik
Friedrich-Schiller-Universität Jena

> „Das eben geschieht den Menschen, die in einem Irrgarten hastig werden: Eben die Eile führt immer tiefer in die Irre."
>
> — Lucius Annaeus Seneca (4 n. Chr. – 65 n. Chr.)

Ariadne, nach griechischer Sage die Tochter von Minos, dem König von Kreta, verliebte sich in Theseus. Dieser Athener Held war beauftragt, den Minotaurus (ein Ungeheuer halb Mensch, halb Stier) zu töten. Die Herausforderung wurde ungleich größer dadurch, dass der Minotaurus im Labyrinth versteckt war. Die kluge Ariadne stattete ihren Helden mit einer Rolle Faden aus: Indem Theseus ein Fadenende am Eingang des Labyrinths festknotete und den Faden beim Durchforschen des Labyrinths abrollte, konnte er zum einen vermeiden, gleiche Teile des Labyrinths mehrfach zu durchsuchen, und zum anderen sicherstellen, auch wieder den Weg zurück in Ariadnes Arme zu finden.

Nicht nur die alten Griechen mussten sich mit der geschickten Durchmusterung von Suchräumen, wie z.B. Labyrinthen, befassen, sondern auch in der Informatik von heute spielt diese Aufgabe eine zentrale Rolle. Eine Methode hierzu stellt die Tiefensuche (englisch *depth-first search*) dar, die wir nachfolgend genauer betrachten wollen.

Originally published in *Taschenbuch der Algorithmen*, *eXamen.press*, pages 61–73. Springer, 2008.

Algorithmische Idee und Umsetzung

Wie bereits angekündigt, gilt es die Aufgabe zu lösen, ein Labyrinth vollständig zu durchsuchen. Ein Labyrinth ist dabei ein aus Gängen, Sackgassen und Kreuzungen bestehendes Gebilde – die Aufgabe besteht somit darin, jede Kreuzung und jede Sackgasse mindestens einmal zu besuchen. Wünschenswert wäre zudem, wenn kein Gang des Labyrinths mehr als einmal pro Richtung durchlaufen werden würde – schließlich soll Theseus am Ende noch genügend Kraft übrig haben, um weder beim Minotaurus noch bei Ariadne schlappzumachen.

Die wahrscheinlich einfachste Idee zur Lösung dieses Problems ist, vom Startpunkt aus einfach immer weiter ins Labyrinth hineinzulaufen und alle Kreuzungen, auf die man unterwegs stößt, als erledigt abzuhaken. Landet man in einer Sackgasse oder auf einer Kreuzung, die man schon kennt, dreht man einfach um, geht zur letzten Kreuzung zurück und versucht es von dort aus erneut in einer anderen, noch unbekannten Richtung. Gibt es von der letzten Kreuzung aus keine unbekannte Richtung mehr, so geht man von dort aus nochmals eine Kreuzung weiter zurück, usw.

Führt diese Vorgehensweise auch zum Ziel? Sehen wir uns die Suche etwas genauer an: Anstatt eines Fadens benutzen wir, der einfacheren Beschreibung wegen, ein Stück Kreide. Mit diesem markieren wir an jeder Kreuzung 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 erstmal einen Haken an die Wand des Ganges, durch den man gekommen ist, um später ggf. wieder zurück finden 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

Originally published in *Taschenbuch der Algorithmen*, *eXamen.press*, pages 61–73. Springer, 2008.

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.

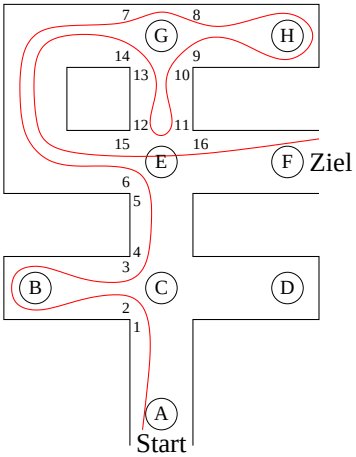


Abb. 1. Beispiel für die Tiefensuche im Labyrinth. Es wird, ausgehend von A, ein Weg nach F gesucht. Die Ziffern kennzeichnen die Stellen, an denen Kreidemarkierungen hinterlassen werden.

Betrachten wir nun folgendes, in Abb. 1 dargestellte Beispiel, in dem ein Weg vom Start A zum Ziel F gesucht wird. (Es soll also wiederum das ganze Labyrinth durchsucht werden, wobei die Suche jedoch abgebrochen werden kann, sobald F gefunden wird.) Wir gehen davon aus, dass eine Sackgasse nur unmittelbar vor ihrem Ende als solche erkannt werden kann.

Man geht los vom Start A Richtung Norden. Die erste Kreuzung ist C. Dort macht man eine Markierung am Südausgang der Kreuzung (1). Natürlich ist hier sonst noch keine Markierung, also nimmt man den ersten unmarkierten Weg von links, d. h. den in Richtung Westen, und markiert ihn (2). Dann erreicht man bei B eine Sackgasse, also dreht man um. Wieder bei C angelangt, hat der Weg nach Westen jetzt schon zwei Markierungen, der nach Süden eine, aber nach Norden ist noch gar nichts markiert, also geht man dort lang. Bei E ist wieder eine unberührte Kreuzung, und man wählt unter den drei Möglichkeiten die nach Westen. Nach zwei Kurven geht man in G geradeaus über die Kreuzung, zwei Markierungen hinterlassend (7 und 8). In H trifft man auf eine Sackgasse, also dreht man wieder um. In G ist nur eine Möglichkeit erlaubt: nach Süden zu E. Hier tritt zum ersten Mal die Regel gegen das Im-Kreis-Laufen in Kraft: Man hat beim Betreten von E einen Haken am Nordausgang von E gemacht (11); außerdem existiert an dieser Kreuzung bereits eine Markierung am Südausgang (5) und eine am Westausgang (6) –

Originally published in *Taschenbuch der Algorithmen*, *eXamen.press*, pages 61–73. Springer, 2008.

demnach muss man also umdrehen. Über die Kreuzung G und zwei Kurven geht es zurück, sodass nun der nördliche Teil komplett abgesucht ist und man wieder bei E landet. Nach Osten gibt's hier noch gar keine Markierung, also geht man dort lang. Schließlich erreicht man in F das Ziel.

Das Prinzip, das wir hier kennengelernt haben, nennt sich *Tiefensuche*, da man, wie schon erwähnt, immer möglichst tief in das Labyrinth hineingeht und nur dann, wenn es nicht mehr weitergeht oder man auf eine schon bekannte Stelle trifft, ein Stück zurückgeht und es von einem früheren Punkt in anderer Richtung erneut versucht.

Die Regeln für die Tiefensuche sind so einfach, dass man sie mit wenigen Zeilen einem Computer beibringen kann. Dabei merkt man 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 im nachfolgend abgebildeten Programmstück). 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 wird gemacht, indem sich die TIEFENSUCHE-Funktion für jede benachbarte Kreuzung Y selbst aufruft (Zeilen 5–7). Dies ist ein beim Programmieren häufig benutzter Trick namens *Rekursion*, der schon in Kapitel 1 beschrieben wurde. Bemerkt die frisch 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

```
1  function TIEFENSUCHE(X):    // Definition 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
```

Manchmal möchte man Rekursionen vermeiden, z.B. weil bei jedem rekursiven Funktionsaufruf zusätzliche Zeit für das Anlegen von Variablen im Speicher etc. benötigt wird. In diesem Fall kann die Tiefensuche mit Hilfe eines *Stapels* auch ohne Rekursion programmiert werden. Unter einem Stapel versteht man dabei eine Datenstruktur, die es erlaubt, Objekte (in unserem Fall Kreuzungen) oben auf den Stapel zu legen oder aber das momentan oberste Objekt vom Stapel zu nehmen. Der Stapel dient bei uns dazu, den „Rückweg" zu speichern, man legt also immer die Kreuzung X, die man gerade verlässt,

Originally published in *Taschenbuch der Algorithmen*, *eXamen.press*, pages 61–73. Springer, 2008.

oben auf den Stapel, und zwar zusammen mit einer Zahl „Ausgänge“, die angibt, zu wievielen benachbarten Kreuzungen man von X aus schon aufgebrochen ist. Zu jeder Kreuzung existiert eine Liste (genauer: ein Array), in der alle Nachbarkreuzungen verzeichnet sind – somit kann man bei Bedarf gezielt beispielsweise die fünfte Nachbarkreuzung einer Kreuzung X auswählen.

TIEFENSUCHE II

```
 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
```

Anwendungen

Das Verfahren Tiefensuche funktioniert nicht nur für Labyrinthe, sondern findet auch in deutlich anderen Zusammenhängen Anwendung, wie wir in diesem Abschnitt sehen werden.

Suche im Web

Im folgenden Anwendungsbeispiel geht es nochmals ums Suchen, allerdings irrt hier nicht Theseus im Labyrinth umher, sondern ein Schüler namens Sinon sucht eine bestimmte Webseite.

Sinon Schneider war kürzlich auf einer Party seiner Mitschülerin Ariadne und ist dort kurz mit einem netten Mädchen ins Gespräch gekommen. Er würde sie nun gerne wiedersehen, aber dummerweise hat er sie nicht nach

Originally published in *Taschenbuch der Algorithmen*, *eXamen.press*, pages 61–73. Springer, 2008.

ihrem Namen gefragt. Was tun? Sinon könnte natürlich die Gastgeberin Ariadne fragen, aber erstens traut er sich nicht, und zweitens kennt sie selbst auch nur einen Teil ihrer Partygäste. Schließlich kommt Sinon die rettende Idee: Wieso nicht einfach im Internet bei „GaudiVZ.de“ nachsehen? In diesem berühmten Schülerverzeichnis haben fast alle Schüler in Deutschland ein Profil von sich angelegt, meistens sogar mit einem Photo und Links zu den Profilen von Freunden. Also könnte Sinon doch im GaudiVZ Ariadnes Profil besuchen und sich von dort aus durch alle ihre Freunde und alle Freunde von ihren auf der Party anwesenden Freunden und alle Freunde von Freunden von Freunden usw. klicken, bis er entweder seine Angebetete gefunden hat (hoffentlich hat sie ein Bild von sich eingestellt!) oder aber das ganze in Frage kommende Umfeld von Ariadne abgesucht hat. Sinon steht also vor folgender Aufgabe: Durchmusterung aller Profile des GaudiVZ, die von Ariadne aus über Links erreichbar sind und deren Eigentümer er auf der Party gesehen hat. Auch hier liegt die Schwierigkeit darin, einerseits nicht endlos im Kreis zu laufen und andererseits alles gewissenhaft und systematisch zu überprüfen. Dies lässt sich effektiv erledigen mit Hilfe einer Tiefensuche im Geflecht der einzelnen Profile des GaudiVZ.

Nehmen wir also an, Sinon beginnt mit seiner Suche und startet beim Profil von Ariadne. Also klickt er auf den ersten Link in Ariadnes Freundeliste und landet beim Profil von Theseus (siehe Abb. 2). Da es sich dabei nicht um die von ihm gesuchte Person handelt, geht es weiter mit dem ersten Link dieser Seite, wobei Sinon jedoch darauf achtet, keinen Link anzuklicken, der zu einem Profil führt, das er bereits besucht hat – solche Links erkennt er als geübter Websurfer daran, dass sie in seinem Browser in violett anstatt in blau erscheinen (diese hilfreiche Funktion des Browsers entspricht also gewissermaßen dem Malen von Kreidehaken in unserem ersten Beispiel). Sobald er schließlich ein Profil erreicht, dessen Benutzer seiner Meinung nach nicht auf der Party war, oder wenn alle Freunde eines Profils abgearbeitet – d.h. besucht und daher violett dargestellt – sind, so klickt er auf den „Zurück“-Knopf seines Browsers und fährt mit den Freunden des nun angezeigten Profils fort. Wie die Tiefensuche II verwendet die „Zurück“-Funktion des Browsers einen Stapel, auf den beim Anklicken eines Links die Adresse der aktuellen Seite abgelegt und von dem beim Benutzen des „Zurück“-Knopfs die oberste Adresse entnommen wird.

Wenn Sinons Herzensdame ein Bild von sich eingestellt hat und wenn sie von Ariadnes Profil aus über eine Folge von verlinkten Profilen erreichbar ist, deren Besitzer alle auf der Party waren (wovon wir ja ausgegangen sind), dann wird er sie mit dieser Methode zwangsläufig finden! Sollte er allerdings irgendwann beim Klicken auf den „Zurück“-Knopf zum wiederholten Male beim Profil von Ariadne landen, und die Links zu all ihren Freunden sind violett gefärbt, d.h. schon besucht, dann würde das für ihn bedeuten, dass er Pech gehabt hat – aber immerhin könnte er sich sicher sein, keines der in Frage kommenden Profile ausgelassen zu haben!

Originally published in *Taschenbuch der Algorithmen*, *eXamen.press*, pages 61–73. Springer, 2008.

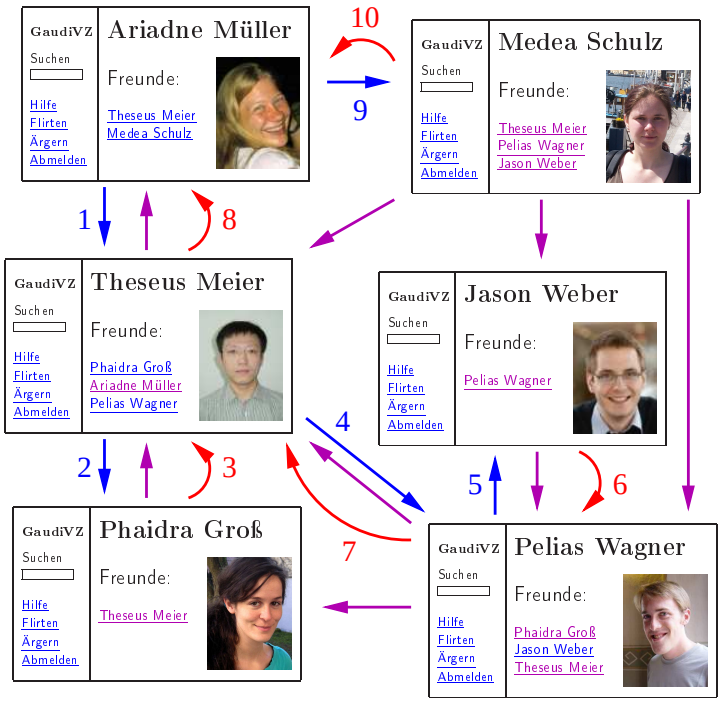
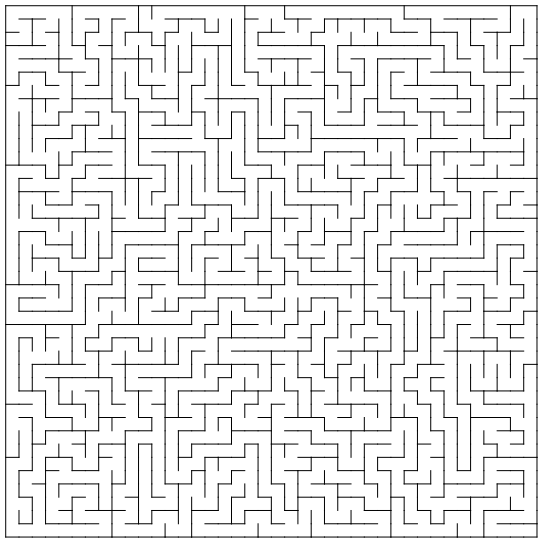


Abb. 2. Tiefensuche im GaudiVZ: Die Zahlen verdeutlichen die Reihenfolge der Sprünge von Seite zu Seite. Ein gerader Pfeil bedeutet, dass eine Seite einen Link auf eine andere Seite enthält. Gebogene Pfeile bedeuten, dass an dieser Stelle der „Zurück"-Knopf benutzt wird. Man beachte, dass nicht alle dargestellten Links angeklickt werden, da manche davon zum Zeitpunkt des Besuchs bereits in violett angezeigt werden.

Erstellen von Labyrinthen

Nicht nur für Theseus, sondern auch für den Minotaurus ist Tiefensuche nützlich. Sie kann nämlich auch zum Entwurf besonders verwirrender Labyrinthe benutzt werden. Die Vorgehensweise ist recht einfach: man startet mit einem regelmäßigen rechtwinkligen Gitter. Die Tiefensuche startet in einer beliebigen Zelle. Dann wird die Tiefensuche für alle Nachbarzellen in zufälliger Reihenfolge aufgerufen (dabei kann beispielsweise der Zufallszahlen-Algorithmus aus Kapitel 25 helfen). Wird dabei eine Zelle zum ersten mal besucht, so wird die Wand zur Vorgängerzelle eingerissen. Es ergibt sich ein Muster wie das folgende:

Originally published in *Taschenbuch der Algorithmen*, *eXamen.press*, pages 61–73. Springer, 2008.



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 – nur leicht zu finden ist dieser Weg nicht unbedingt...

Weitere Beispiele: Unterhaltungsshows und Verkehrsplanung

Wie wir gesehen haben, ist Tiefensuche ein sehr einfaches Grundprinzip; in der wissenschaftlichen Literatur lässt sich eine Vielzahl von Anwendungsbeispielen finden. Wir wollen hier nur noch zwei „lebensnahe“ erwähnen – weitere Beispiele finden sich in anderen Kapiteln dieses Buchs (siehe Hinweise am Ende des Kapitels).

Angenommen, die Fernsehshow „Sick Sister“ plant die Produktion von zwei neuen Staffeln. Bei dieser beliebten Show werden die Teilnehmer in einen Container gesteckt und rund um die Uhr mit Fernsehkameras beobachtet. Damit das Ganze nicht zu langweilig wird, sollten sich die Teilnehmer möglichst oft in die Haare geraten, was wiederum bedeutet, dass bei jeder Staffel möglichst keine zwei Teilnehmer im Container sein sollten, die sich sympathisch sind. Nehmen wir nun an, die Teilnehmer für die nächsten zwei Staffeln der Show sind bereits ausgewählt und es geht nun nur noch darum, für jeden der Teilnehmer festzulegen, ob er bei der ersten oder der zweiten Staffel mitmachen soll, wobei natürlich die erwähnte Regel „keine Personen im Container, die sich mögen“ einzuhalten ist.

Zur Lösung dieser Aufgabe könnte man zunächst einen „Sympathiegraph“ erstellen, das heißt, man zeichnet auf ein Blatt Papier für jeden Teilnehmer einen Kringel – *Knoten* genannt – und verbindet zwei Kringel durch eine Linie – *Kante* genannt –, wenn sich die entsprechenden Personen sympathisch

Originally published in *Taschenbuch der Algorithmen*, *eXamen.press*, pages 61–73. Springer, 2008.

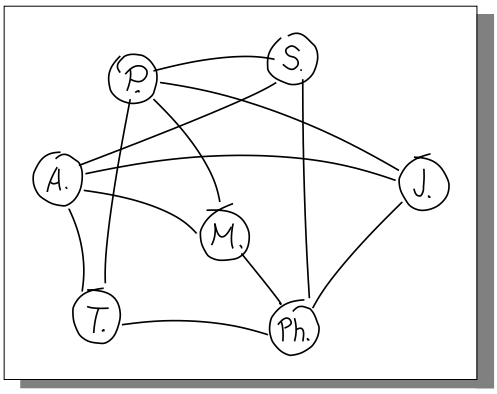


Abb. 3. Ein „Sympathiegraph“: Sind sich zwei Personen sympathisch, so sind die entsprechenden beiden Knoten durch eine Linie miteinander verbunden.

sind (siehe Abb 3 für ein Beispiel).[1] Die Aufgabe ist nun, die Knoten mit einer von zwei Farben jeweils so anzumalen, dass jeder Knoten genau eine Farbe erhält und dass keine zwei Knoten, die durch eine Kante verbunden sind, die gleiche Farbe erhalten. Die Farbe eines Knotens gibt am Ende an, bei welcher der beiden Staffeln die entsprechende Person teilnehmen soll. Die gesuchte „Zweifärbung“ des Graphen kann nun mit Hilfe einer Tiefensuche gefunden werden: Man wählt zunächst einen beliebigen Knoten aus und gibt ihm eine beliebige der beiden zur Verfügung stehenden Farben. Anschließend startet man bei diesem Knoten die Tiefensuche. Immer dann, wenn man, ausgehend von einem Knoten X, auf einen noch unentdeckten Knoten Y stößt, gibt man diesem die Farbe, die X nicht hat. Kommt man hingegen von einem Knoten X zu einem bereits entdeckten Knoten Y, so kontrolliert man, ob X und Y unterschiedliche Farben haben. Ist dies nicht der Fall (hat man also zwei verbundene Knoten derselben Farbe gefunden), so kann der Graph nicht wie gewünscht zweigefärbt werden. Andernfalls liefert die Tiefensuche die gewünschte Zweifärbung und sorgt so für einen hohen Unterhaltungswert von „Sick Sister“.[2]

[1] Diese Art von Graphen mit Knoten und Kanten hat übrigens nichts zu tun mit Graphen von Funktionen, wie man sie aus der Analysis kennt. Mit „Knoten-und-Kanten“-Graphen können die unterschiedlichsten Sachverhalte und Objekte modelliert werden, zum Beispiel auch Labyrinthe: Jede Sackgasse und jede Kreuzung wird dann jeweils zu einem Knoten, und jeder Gang zu einer Kante.

[2] In dem Spezialfall, dass der Sympathiegraph aus mehreren, untereinander nicht verbundenen Teilen, den so genannten „Zusammenhangskomponenten“, besteht, muss man die Tiefensuche für jeden Teil einzeln anwenden. Dabei bietet sich unter Umständen sogar die Möglichkeit, die Teilnehmer zahlenmäßig möglichst ausgewogen auf die zwei Staffeln zu verteilen, indem man nämlich im Nachhinein bei manchen Zusammenhangskomponenten die beiden Farben vertauscht.

Originally published in *Taschenbuch der Algorithmen*, *eXamen.press*, pages 61–73. Springer, 2008.

Graphen, die wie beschrieben zweigefärbt werden können, werden übrigens *bipartit* genannt – es sind dies genau diejenigen Graphen, die keinen Kreis ungerader Länge enthalten. Sollen die Teilnehmer allerdings auf drei statt auf zwei Staffeln aufgeteilt werden, so wird die Aufgabe plötzlich sehr viel schwieriger, und niemand weiß, ob man sie mit Hilfe von Tiefensuche effizient lösen kann. (Das Problem ist, dass man beim Besuchen eines bisher noch unentdeckten Knotens zwei Farben zur Auswahl hat und nicht weiß, welche davon man auswählen soll.)

Eine weitere Anwendung der Tiefensuche ist die folgende: Angenommen, Stadtrat Hermes will zwecks Verkehrsberuhigung in der Innenstadt eine Vielzahl von Straßen zu Einbahnstraßen erklären. Dabei muss Hermes sich aber vor dem Zorn der Autofahrer in Acht nehmen und dafür Sorge tragen, dass das Straßennetz nicht so eingeschränkt wird, dass abgeschnittene „Inseln“ entstehen, die man nicht erreichen oder verlassen kann, sprich, dass ein Autofahrer nicht mehr von überall nach überall kommen kann. Graphtheoretisch modelliert bedeutet das, dass der zugrundeliegende Straßennetzgraph weiterhin eine einzige „starke Zusammenhangskomponente“ bleiben muss. Auch dieses Problem kann effizient mit Tiefensuche behandelt werden – in Kapitel 9 wird nochmals genauer darauf eingegangen.

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; im GaudiVZ etwa kann man davon ausgehen, dass sich das gesuchte Profil in einer Distanz von z. B. höchstens drei zur Gastgeberin befindet. In diesem Fall 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“, 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 folgende Programmstück zeigt die Breitensuche im Detail.

Originally published in *Taschenbuch der Algorithmen*, *eXamen.press*, pages 61–73. Springer, 2008.

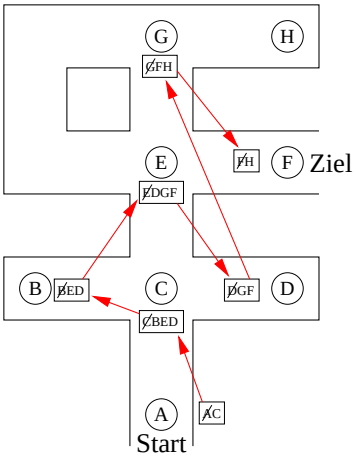


Abb. 4. Beispiel für die Breitensuche im Labyrinth.

BREITENSUCHE

```
1  // 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
```

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).

Als Beispiel sehen wir uns den gleichen Graphen an, der bereits bei der Tiefensuche zum Einsatz kam: das Labyrinth (Abb. 4). Am Anfang hat man eine Warteschlange, die nur den Knoten A enthält. Dieser wird herausgenommen, und alle Nachbarn von A werden eingefügt: das ist nur C. Die sich ergebende Warteschlange ist neben A abgebildet. Weiter geht's beim vordersten Element der Warteschlange, also C. Streng genommen würden jetzt alle vier Nachbarn von C an die Warteschlange angehängt; als kleine Optimierung

Originally published in *Taschenbuch der Algorithmen*, *eXamen.press*, pages 61–73. Springer, 2008.

ignorieren wir jetzt jedoch A, da es schon den „entdeckt“-Status hat und beim Wiederherausnehmen aus der Warteschlange sowieso nichts bewirken würde. Es kommen also B, E und D hinzu. Bei B kommen gar keine neuen Knoten hinzu, es geht direkt weiter bei E. Dort werden dann G und F angehängt, diesmal wird C ignoriert. Bei D passiert wieder nichts, es geht direkt weiter bei G. Dort wird noch H angehängt, aber bei F finden wir dann bereits das Ziel.

Man sieht sofort, dass die Reihenfolge, in der die Knoten besucht werden, ganz anders ist als bei der Tiefensuche (Abb. 1). Bei der Breitensuche werden nämlich die Knoten in der Reihenfolge ihrer Entfernung vom Startknoten besucht: Zuerst Knoten C (Abstand 1), dann B, E und D (Abstand 2), dann G und F (Abstand 3). Daraus ergibt sich auch, dass Breitensuche immer einen kürzestmöglichen Weg zum Ziel findet, und außerdem zwischendurch keine längeren Pfade untersucht.

Übrigens: Verwendet man hier anstatt einer Warteschlange einen Stapel, so führt der Algorithmus statt einer Breitensuche eine Tiefensuche durch! Im Gegensatz zum Algorithmus Tiefensuche II wird dabei jedoch der Stapel nicht dazu verwendet, um den „Rückweg“ zu speichern, sondern um sich diejenigen Kreuzungen zu merken, zu denen man bereits die Gänge entdeckt hat, die man jedoch nicht gleich besucht hat. Dadurch kann der Stapel deutlich größer werden kann als bei Tiefensuche II.

Was soll man nun nehmen bei einem konkreten Problem, Tiefen- oder Breitensuche? Tiefensuche ist normalerweise etwas einfacher zu programmieren, da man sich bei Benutzung von Rekursion nicht explizit um eine Datenstruktur wie die Warteschlange bei Breitensuche kümmern muss. Außerdem benötigt Breitensuche im allgemeinen mehr Speicher; bei schwierigen Problemen kann es sogar sein, dass der vorhandene Speicher schlicht nicht ausreicht. Dafür findet Breitensuche immer einen kürzestmöglichen Pfad (gemessen an der Anzahl Kanten), und ist insbesondere dann schnell, wenn man einen sehr großen Graphen hat, sich das Ziel aber in der Nähe des Starts befindet; hier kann sich die Tiefensuche leicht in weit entfernten Regionen „verfranzen“. Es kommt also auf die konkrete Situation an, welcher Algorithmus der bessere ist.

Danksagung

Wir danken Martin Dietzfelbinger (Ilmenau) für seine vielen konstruktiven Verbesserungsvorschläge.

Zum Weiterlesen

1. Darstellungen zur Tiefensuche finden sich in den meisten Lehrbüchern über Algorithmen.

Originally published in *Taschenbuch der Algorithmen*, *eXamen.press*, pages 61–73. Springer, 2008.

2. Kapitel 9 (Zyklensuche in Graphen)
   In diesem Kapitel wird eine weitere Anwendung für Tiefen- und Breitensuche vorgestellt.
3. Kapitel 8 (Der Pledge-Algorithmus)
   In unserem Labyrinth-Beispiel sind wir davon ausgegangen, dass man an einer Kreuzung stehend, alle Ausgänge sehen kann. Aber was ist, wenn die Fackel ausgeht und man im Dunkeln steht? Auch dann kann man noch zum Ziel finden; wie das geht, ist in Kapitel 8 beschrieben.
4. Kapitel 34 (Kürzeste Wege)
   Die Breitensuche findet einen kürzesten Weg, wenn man die Anzahl durchlaufener Kanten als Maßstab nimmt. Oft sind aber die Abstände von Knoten unterschiedlich groß, und man möchte einen Weg, bei dem die Summe der Kantenlängen möglichst klein ist. Dieses Problem wird in Kapitel 34 behandelt.

> „Alles auf Erden lässt sich finden, wenn man nur zu suchen sich nicht verdrießen lässt.“
>
> — Philemon von Syrakus (um 360 v. Chr. – 264 v. Chr.)

Originally published in *Taschenbuch der Algorithmen*, *eXamen.press*, pages 61–73. Springer, 2008.

Nachtrag

Der nachfolgend dargestellte Algorithmus Tiefensuche III führt, wie der Algorithmus Tiefensuche II, eine Tiefensuche ohne Rekursion unter Verwendung eines Stapels durch. Er ist ist länger, aber intuitiver als Tiefensuche II. Die Variable „Modus“ gibt an, ob man gerade einem noch unerkundeten Gang folgt oder aber von einer bereits entdeckten Kreuzung zurückkommt.

Tiefensuche III

```
 1  X := Startkreuzung;   Modus := „vorwärts“;
 2  repeat
 3     if Modus = „vorwärts“ then     // wir folgen einem neuen Gang
 4        if Zustand[X] = „entdeckt“ then
 5           Modus := „rückwärts“;
 6           nimm das oberste Paar (X, Ausgänge) vom Stapel;
 7        else    // unentdecke Kreuzung
 8           if X = Ziel then exit „Ziel gefunden!“; endif
 9           Zustand[X] := „entdeckt“;
10           if X hat keine Ausgänge then exit „Ziel nicht gefunden!“; endif
11           lege das Paar (X, 1) oben auf den Stapel;
12           X := erste benachbarte Kreuzung von X;
13        endif
14     else    // wir kommen zurück
15        if Ausgänge < Anzahl benachbarter Kreuzungen von X then
16           Ausgänge := Ausgänge + 1;
17           lege das Paar (X, Ausgänge) oben auf den Stapel;
18           Modus := „vorwärts“;
19           X := (Ausgänge)-te benachbarte Kreuzung von X;
20        else    // es gibt hier keine unerkundeten Gänge mehr
21           if Stapel ist leer then
22              exit „Ziel nicht gefunden!“;
23           else    // es geht noch weiter zurück
24              nimm das oberste Paar (X, Ausgänge) vom Stapel;
25           endif
26        endif
27     endif
28  end repeat
```