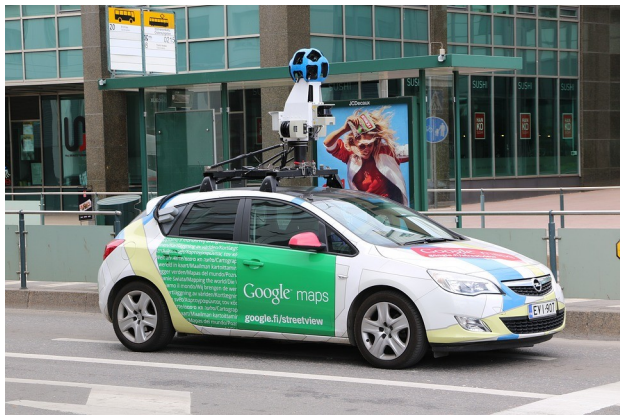
3.4 Tiefensuche

Der Graph hat viele Anwendungen in der Realität. Viele alltägliche Probleme können wir mithilfe von Apps und den Graphen, die sie im Hintergrund verwenden, lösen.

Wir kommen wieder zurück auf unser Beispiel des Straßennetzes als Graph. Betrachten wir einen Graph in dem KNOTEN gewisse Sehenswürdigkeiten oder zentrale Stellen sind und die KANTEN den Weg dorthin darstellen.



Als Google ihr Street-View Feature in Google Maps einbauen wollte, haben sie Kameraautos herumgeschickt, die die Straßenzüge und Kreuzungen abfahren sollten.

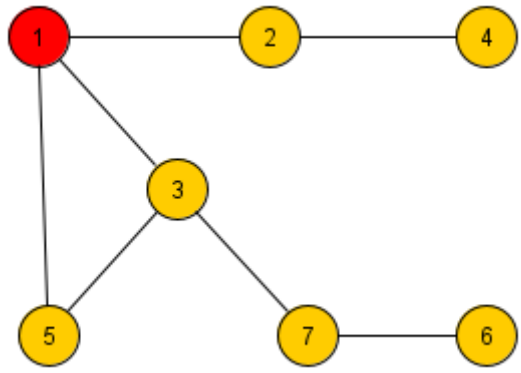
Wir betreuen das Auto, das alle Kreuzungen (KNOTEN) besuchen soll. Damit wir keine auslassen müssen wir überlegen, wie wir vorgehen.

Nehmen wir als Beispiel mal folgenden Straßenzug:

Legende: gelb = noch nicht besucht, rot = aktueller Standort, grün = erledigter Knoten

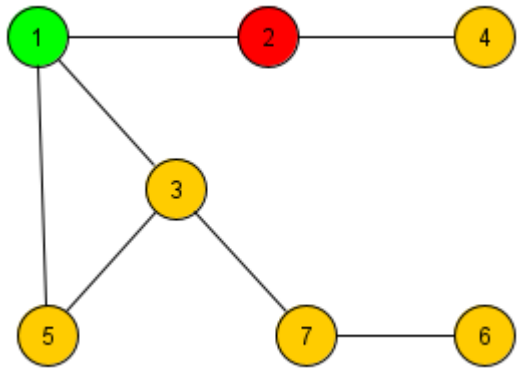
Angenommen das Auto beginnt am Knoten mit der Nummer 1.

Damit stehen zur Auswahl die Knoten 2,3 und 5 als nächste Station, da diese noch nicht besucht wurden.
Theoretisch vollkommen egal wo wir weiter machen, aber es erscheint sinnvoll, einfach der Reihe nach vorzugehen und zu Knoten 2 zu gehen. Knoten 1 haken wir auf unserer Liste als erledigt ab.



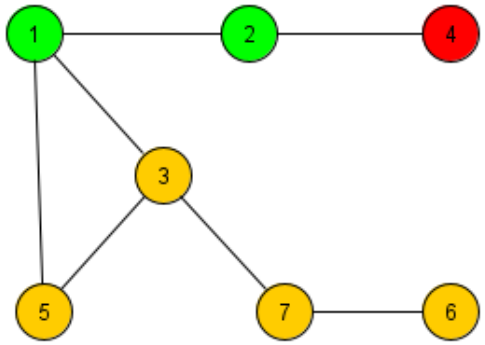
(1)

Wir sehen Knoten 1 ist abgehakt und wir befinden uns an Knoten 2. Hier überprüfen wir, wieder, welche Knoten in der Umgebung noch nicht besucht wurden. Da bleibt uns nur Knoten 4 übrig. Knoten 2 wird abgehakt und zu Knoten 4 gegangen.



(2)

Wir sehen, Knoten 1 und 2 sind erledigt und wir befinden uns an Knoten 4.
Wir überprüfen wieder, welche Knoten, die eine Kante zu uns haben, noch nicht besucht wurden. Da finden sich diesmal keine. Also müssen wir irgendwie zurückkommen. Die einfachste Möglichkeit ist wieder zurück zum vorher besuchten Knoten zu springen.
Wir haken also 4 ab und gehen zurück zu 2.

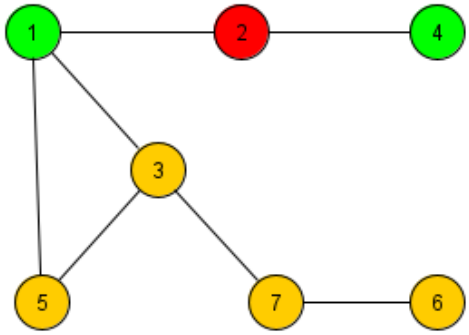


(3)

Hier wieder gleiches Spiel wie bisher:

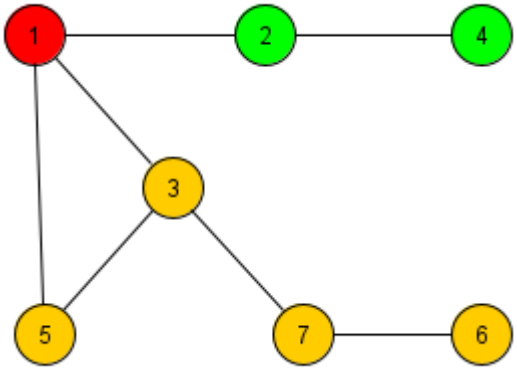
1. Überprüfen, ob in der nähe ein Knoten unbesucht ist.
→ Nein, also zurück zu dem davor

Wir springen also zurück zu Knoten 1.

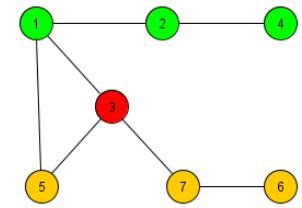


(4)

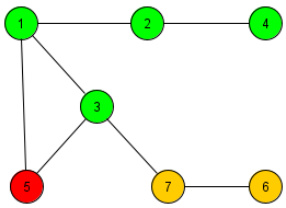
Wir sehen, dass nun die Überprüfung weiter gehen kann. In jedem Knoten müssen wir eigentlich immer das gleiche tun. Schauen wir uns die restlichen im Schnelldurchlauf an…



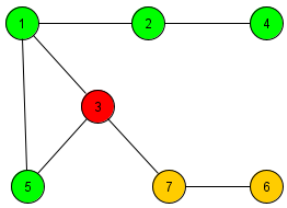
(5)



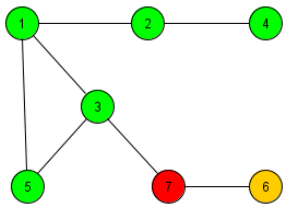
(6)



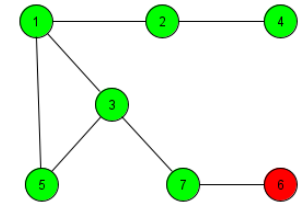
(7)



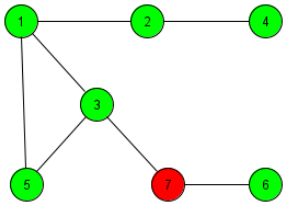
(8)



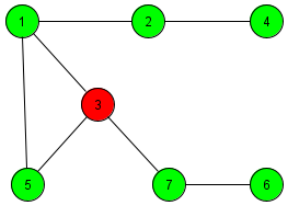
(9)



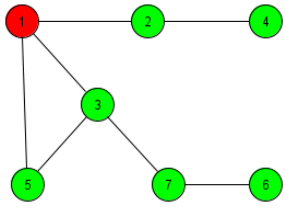
(10)



(11)



(12)



(13)

Wir sehen, in Schritt (13) ist der Algorithmus fertig, da vom Startknoten kein weiterer Weg nicht besucht ist.

Der Algorithmus läuft also folgendermaßen:

1. Beginne mit Start als aktiven Knoten

2. Markiere den aktiven Knoten als besucht.

3. Überprüfe, ob Knoten einen unbesuchten Nachbarn hat.

    → Falls Ja: Nachbar wird aktiver Knoten (z.B. kleinstem Index) (→ weiter bei 2)

    → Falls Nein: Bin ich der Start?

        → Falls Ja: Fertig

        → Falls Nein: Gehe zurück zum vorherigen Knoten. (→ weiter bei 2)

Diesen Algorithmus nennen wir die **Tiefensuche** (englisch: „depth first search“ daher die Abkürzung DFS), da beim Graph zuerst in die Tiefe gegangen wird.

Umsetzung in Java

Klar ist, dass wir für die Markierungen, welcher Knoten bereits besucht wurde ein Feld mit Datentyp `boolean` brauchen. Da `boolean` ein primitiver Datentyp ist, müssen wir im Konstruktor nur angeben, wie groß das Feld ist, da primitive Datentypen automatisch mit einem Wert belegt werden. Bei `boolean` ist das tatsächlich der Wert `false`, was uns entgegenkommt.

Beim Aufstellen der Überlegungen für das Besuchen springen uns zwei Dinge ins Auge:

1. Wir müssen **immer wieder das gleiche tun**
2. Wir müssen zu einem vorher aufgerufenem Knoten **zurückspringen**

Beides spricht sehr für eine rekursive Umsetzung[1]. Wir erinnern uns an die Umsetzung des Baumdurchlaufs. Im Sequenzdiagramm konnte man gut sehen, dass die Rekursion das Zurückspringen zum richtigen Vorgänger von alleine für uns erledigt hat. Das gleiche Phänomen können wir auch bei der Tiefensuche umsetzen. Praktischerweise sind beim Graph, wenn er mit einer Adjazenzmatrix umgesetzt wurde, alle Informationen in der Klasse GRAPH gespeichert. Wir haben hier alle KANTEN (in der Matrix) und alle KNOTEN (in einem Feld) veorrätig. Damit können wir (im Gegensatz zum Baum oder der Liste) alles in der Klasse GRAPH erledigen.

---

1 Anmerkung: Nur aus 1. und 2. folgt eine Rekursion. Nur aus 1. wäre eine Wiederholung ausreichend

Überlegen wir uns die nötigen Struktogramme der Methoden `Tiefensuche(int start)` und `BesucheDFS(int knotenIndex)`.

void Tiefensuche(int startKnoten)

Ist "startKnoten" ein erlaubter Index?	
Wahr	Falsch
Setze im Feld alle Knoten auf unbesucht (false) BesucheDFS(startKnoten)	Fehlermeldung: startKnoten ist kein gültiger Knoten

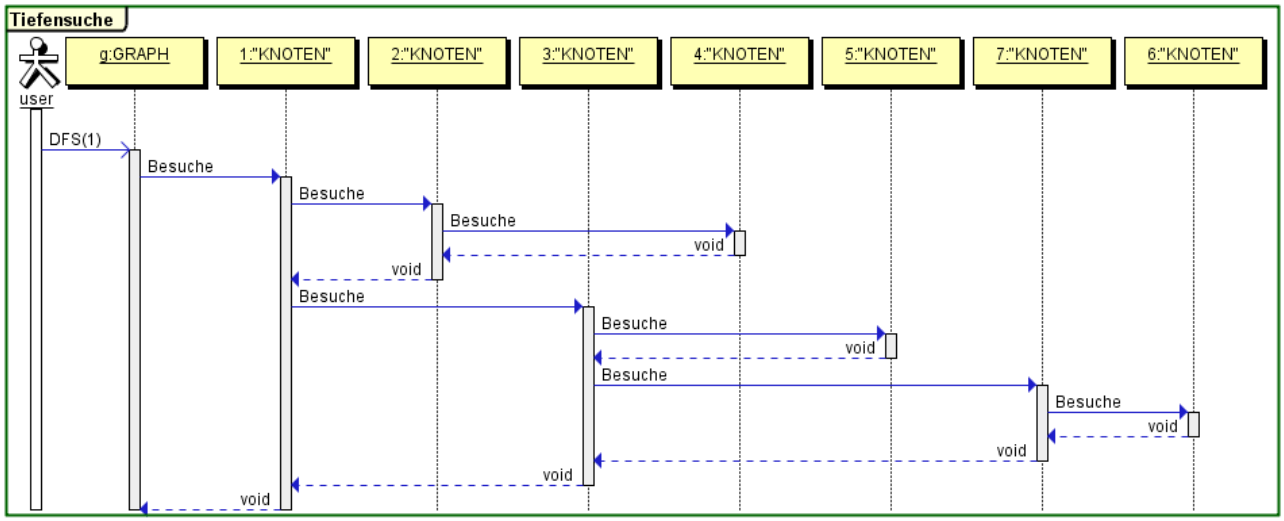
void BesucheDFS(int knotenIndex)

besuchteKnoten[knotenIndex] = true

Wiederhole für i=0 bis AnzahlKnoten-1

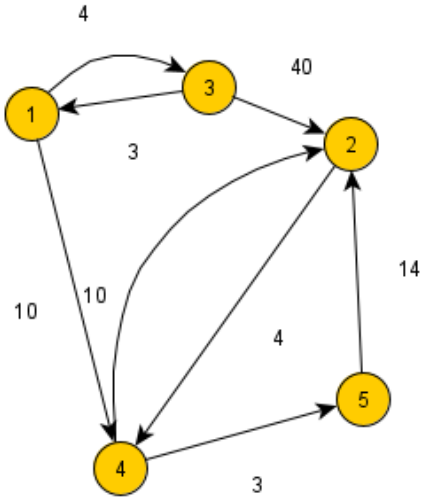
matrix[knotenIndex][i] > 0 && knotenBesucht[i] == false	
Wahr	Falsch
BesucheDFS(i)	//nichts

Im Sequenzdiagramm sieht man sehr gut, warum die Rekursion automatisch dazu führt, dass wir zum passenden Index zurückspringen. Achtung beim Lesen des Struktogrammes muss man im Hinterkopf behalten, dass wir nicht wirklich in die KNOTEN springen (als Objekte) sondern nur in die Methode `BesuchenDFS` mit dem jeweiligen Index, der oben steht:



Aufgaben:

1) Überlegen Sie, in welcher Reihenfolge die Knoten im Graph rechts besucht werden würden, wenn man den Graph ausgehend von Knoten 1 nach dem DFS Algorithmus besuchen würde. (Tipp: bei gerichteten Graphen kann man trotzdem „zurückspringen“ auch wenn es keinen Rückweg durch eine Kante gibt!)



2) Können Sie einen Graph angeben, in dem man mit dem DFS nicht alle Knoten besuchen würde?

3) Kopieren Sie die Vorlage Graph_mit_Adjazenzmatrix_DFS_Vorlage aus dem Klassenordner.

4) Machen Sie sich mit der Methode `Tiefensuche` vertraut. Die Vorlage hat diese Methode und das Feld `boolean[] erledigt` bereits implementiert.

5) Implementieren Sie den Algorithmus DFS in der Vorlage. Verwenden Sie die Methoden `Tiefensuche` und `BesuchenDFS` in der Vorlage.

6) In manchen Anwendungen ist es wichtig, einen „Rahmen-GRAPH“ zu erstellen, der aus dem GRAPH besteht, der nur durch das besuchen nach DFS entsteht. Er enthält alle KNOTEN, aber nur die KANTEN, die beim DFS benutzt wurde. Implementieren Sie eine Methode `DFSGraph` in der Klasse GRAPH, die aus dem existierenden Graph den Rahmen-GRAPH erstellt und diesen zurückgibt.