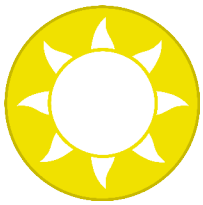
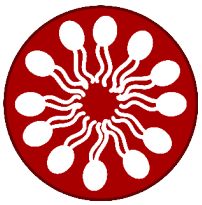
Deutsche JuniorAkademien

JuniorAkademie Adelsheim

14. Science Academy Baden-Württemberg 2016

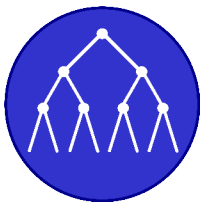


Astronomie



Chemie

Germanistik



Informatik

Physik

TheoPrax

H.W. & J. Hector Stiftung

Regierungspräsidium Karlsruhe
Abteilung 7 – Schule und Bildung

Stiftung Bildung und Jugend

Dokumentation der JuniorAkademie Adelsheim 2016

14. Science Academy Baden-Württemberg

Träger und Veranstalter der JuniorAkademie Adelsheim 2016:
Regierungspräsidium Karlsruhe
Abteilung 7 –Schule und Bildung–
Hebelstr. 2
76133 Karlsruhe
Tel.: (0721) 926 4454
Fax.: (0721) 933 40270
www.scienceacademy.de
E-Mail: joerg.richter@scienceacademy.de
petra.zachmann@scienceacademy.de

Die in dieser Dokumentation enthaltenen Texte wurden von den Kurs- und Akademieleitern sowie den Teilnehmern der 14. JuniorAkademie Adelsheim 2016 erstellt. Anschließend wurde das Dokument mit Hilfe von LaTeX gesetzt.

Gesamtredaktion und Layout: Jörg Richter

Vorwort

Zum 14. Mal bereits fand in diesem Jahr die Junior Akademie Adelsheim statt, traditionell am Landesschulzentrum für Umwelterziehung am Eckenberg in Adelsheim. Schon im Juni starteten wir, Leiter, Mentoren und 71 Teilnehmer, mit dem Eröffnungswochenende und dem damit verbundenen ersten Kennenlernen in die diesjährige Akademie. Mit dem Schreiben der Dokumentation im Herbst wurden die Ergebnisse und Erlebnisse der Akademie festgehalten und die Akademie damit zu einem schönen Abschluss gebracht.

Durch das Arbeiten in den Kursen erhalten die Jugendlichen Einblicke in das wissenschaftliche Arbeiten und erlernen den eigenständigen Umgang mit schwierigen Fragestellungen, indem sie sich intensiv mit einem Thema auseinandersetzen. Neben dem Zuwachs an fachlichem und methodischem Wissen können die Teilnehmer auf der persönlichen Ebene von der Akademie profitieren: Die gemeinsam verbrachte Zeit schweißt all diejenigen, die an der Akademie teilnehmen, zu einer großen Gemeinschaft zusammen und führt zu einer besonderen Akademieatmosphäre.

Getragen werden diese vielfältigen Erfahrungen jedes Jahr durch ein Motto. Es begleitet uns durch die Akademie und regt immer wieder zum Nachdenken und Reflektieren, aber auch zum Hervorheben von besonders witzigen und bemerkenswerten Momenten an. In diesem Jahr stand die Akademie unter dem Motto „Brücken“.

Hier in Adelsheim bauten wir zahlreiche Brücken: Zum einen bauten die Kurse Brücken zu neuem Wissen, und manchmal wurden Eselsbrücken gefunden, um sich neu Gelerntes besser zu merken. Besonders wichtig waren aber auch die neu entstandenen Brücken zwischen den Teilnehmern, die Freundschaften, die oft weit über die Akademiezeit hinaus Bestand haben.

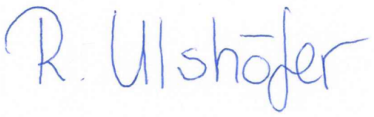
Wir haben jedoch nicht nur im symbolischen Sinne Brücken gebaut: Was genau die Akademie sein würde, das konntet ihr als Teilnehmer vor ihrem Beginn nicht wissen. Verdeutlicht wurde das durch eine große Plakatwand mit einem Abgrund, der zwischen zwei Ufern – dem Eröffnungs- und dem Doku-Wochenende – lag. Durch eure mitgebrachten Stärken, neu entdeckten Talente und gemeinsamen Akademie-Erlebnisse, die ihr fleißig auf Zetteln notiert habt, standen uns solide Bausteine für eine Akademiebrücke zur Verfügung.

Diese hat uns bis zum Doku-Wochenende geführt, bei dem ihr eure wissenschaftlichen Erkenntnisse und all das, was die Akademie sonst noch ausgemacht hat, in Form dieser Dokumentation auf Papier festgehalten habt.

Aber jetzt wünschen wir euch viel Spaß beim Lesen, Schmökern und Erinnern!

Eure/Ihre Akademieleitung

Anna Kandziora (Assistenz)



Rebecca Ulshöfer (Assistenz)

Jörg Richter

Dr. Petra Zachmann

Inhaltsverzeichnis

VORWORT **3**

KURS 1 – ASTRONOMIE **7**

KURS 2 – CHEMIE **29**

KURS 3 – GERMANISTIK **45**

KURS 4 – INFORMATIK **67**

KURS 5 – PHYSIK **83**

KURS 6 – THEOPRAX **101**

KÜAS – KURSÜBERGREIFENDE ANGEBOTE **119**

DANKSAGUNG **137**

Kurs 4: Die Suche nach dem optimalen Weg

Vorwort

Michael Mattes, Bernhard Petzold, Melina Soysal

Die Suche nach dem optimalen Weg ist nicht einfach. Wir wussten zu Beginn nicht, ob wir das hoch gesteckte Ziel erreichen würden und mussten sehr darauf achten, es nicht aus den Augen zu verlieren.

Nicht immer war allen klar, auf welches Ziel wir hinaus wollten. Andere wiederum hatten Zweifel, ob unser eingeschlagene Weg der Richtige ist. Auch mussten wir einige Brücken erst errichten, bevor wir sie überqueren konnten. Uns als Kursleitungsteam war jedoch eines bewusst: Mit einem so motivierten Team steht es außer Frage: Wir erreichen das Ziel!

Wir präsentieren daher mit Stolz die Ergebnisse unseres Kurses Informatik – die Suche nach dem optimalen Weg.

Personen

Alicia Ehrgeizig, motiviert und lustig sind Adjektive, die Alicia sehr gut beschreiben. Wenn es im Kurs eine Aufgabe zu lösen gibt, ist Alicia immer gleich mit Motivation dabei und hört nicht auf, bis sie diese bewältigt hat. Aber auch außerhalb des Kurses ist sie ehrgeizig, wie beispielsweise beim Lösen der Rätsel während der Exkursion. Doch das ist nicht die einzige Seite von Alicia. Sie ist auch immer für einen Spaß zu haben und gut gelaunt bei der Sache. Man kann sich außerdem wunderbar mit ihr unterhalten.

Amelie Rätseln und Rätselmachen, das sind seit dem Wandertag zwei von Amelies Begeisterungen während der Akademie, wobei sie genauso ehrgeizig ist wie im Kurs. Auch wenn mal die Motivation sinkt, macht sie immer noch mit ihrer Aufgabe weiter, was besonders hilfreich und nötig war, als

der Präsentationstag anstand. Reden kann man mit Amelie über alles und Spaß hat man dabei auch, da sie bei Quatsch und Späßen gerne mitmacht.

Dennis ist unser größter Kursteilnehmer und für jeden Spaß zu haben. Er arbeitet auch sehr motiviert mit. Von seinen Zauberwürfel-Künsten können wir uns alle noch etwas abgucken, aber er hat auch noch viele andere Talente, beispielsweise Volleyball und Tischtennis spielen. Er hat immer gute Laune und ist eine unkomplizierte Person.

Henriette ist jedem sofort sympathisch. Sie stürzt sich mit Freude in jede Unterhaltung. Auch wenn es keine gibt, stört sie das nicht sehr. So weiß man zu jeder Zeit ihre Meinung, was beim Programmieren und unseren anderen Aufgaben sehr von Vorteil war. In der Freizeit ist sie immer auf dem Volleyballfeld zu finden.

Julia Dass Julia ein toller Mensch ist, merkt man sofort. Nicht nur aufgeschlossen, motiviert, interessiert und immer gut drauf ist sie, sondern auch ein sehr kompetentes Mädchen in vielerlei Hinsicht. Mit ihr kann man viel Spaß haben, und ihre gute Laune ist stets ein wichtiger Bestandteil der Kursatmosphäre.

Lene ist bei Gruppenarbeiten nicht mehr wegzudenken, denn sie arbeitet immer fleißig mit und entdeckt jedes fehlende Semikolon sofort. Zwischendurch fand sie immer noch Zeit, um sich beim Golf auf dem Laufenden zu halten. Sie brachte uns ihre Interessen näher, sodass wir am Ende auch ein wenig vom Golfen verstanden.

Lennart ist immer bereit, anderen Kursteilnehmern zu helfen, wenn diese mal nicht weiterkommen. Wenn er trotzdem nichts zu tun hat, unterhält er seine Nebensitzer mit amüsanten Youtube-Videos. Außerhalb des Kurses spielt er gerne (und ziemlich gut) Tischtennis und nahm auch an der Musik-KüA teil, wo er mit seiner Querflöte das Ensemble bereicherte.

Mario Mit seinem nahezu unbegrenzten Informatikwissen ist Mario eine große Hilfe im Kurs. Aber auch wenn man einfach nur lachen oder rätseln will, ist Mario die richtige Person, da er viele gute Rätsel und Witze kennt. Er ist immer gut und locker drauf. Mit dieser guten Laune kann er auch andere anstecken. Außerdem beeindruckte er alle mit seinen Poi-, Diabolo- und Jongliérkünsten, die er in der Zirkus-KüA anderen nahegebracht und am Abschlussabend präsentiert hat.

Natalie Mit ihrer lockeren und aufgeschlossenen Art ist Natalie nicht nur für jedes Gespräch zu haben, sondern bietet auch jederzeit ihre Hilfe an. Selbst bei schwierigeren Aufgaben ist Natalie nicht zu bremsen und zeigt großes Interesse und Durchhaltevermögen. Sie verpasst anderen gerne kunstvolle Flechtfrisuren, die immer sehr schön aussehen.

Patrick Seine sehr guten Kenntnisse der Möglichkeiten des Power-Point-Programmes konnte er bei unseren beiden Präsentationen anwenden, und er hat auch bei allen anderen Aufgaben immer den Durchblick. Außerhalb des Kurses ist er ein begeisteter Tischtennisspieler, mit dem man gerne eine Partie spielt.

Stefani Mit ihrer unglaublichen Begeisterung für Mathematik bereichert sie den Kurs sehr. Außerdem ist Schrödingers Katze eines ihrer Lieblingsthemen. In jeder Präsentation versteckt sie irgendwo eine Katze. Auch zwischenmenschlich ist Stefani eine sehr offene und liebe Person. Braucht man einmal eine Aufmunterung oder eine kleine Umarmung, muss man gar nicht erst fragen, sondern wird von Stefani sofort fest gedrückt.

Wendelin Da er talentiert und ehrgeizig ist, fallen ihm viele Teile des Kurses sehr leicht. Mit seinen Programmierkenntnissen beeindruckt er uns alle. Wenn er mit seinen Aufgaben fertig ist, hilft er gerne anderen, für die es nicht so einfach ist. Durch seine zahlreichen und vor allem richtig guten Witze hat man oft etwas zu lachen, wodurch die Kursatmosphäre deutlich angehoben wird und er jedem gute Laune bringt. Beim Bergfest und beim Abschlussfest sorgte er als DJ mit coolen Liedern für Stimmung.

Bernhard hatte es schwer dieses Jahr: Er wurde wegen seiner Knieverletzung von uns selektiv geärgert :). Er ließ es sich trotz alledem nicht nehmen, uns nach Heidelberg zu begleiten, und auch seinen Humor hat er nicht verloren. Er versuchte, bei allen Spielen mitzuspielen, was sich beim „Kotzenden Känguru“ und bei „Amöbe“ wegen seiner Krücken überhaupt nicht einfach gestaltete. Zum Glück war er beim Dokumentationswochenende wieder genesen und konnte so ein entspanntes und ärgerfreies Wochenende erleben.

Michael hat immer ein paar lustige Geschichten und Anekdoten parat, mit denen er uns unsere Süßipausen noch amüsanter gestaltete, als sie es durch die lustigen Spiele ohnehin schon waren. Auf der Fahrt nach Heidelberg beschäftigte er uns mit interessanten zusätzlichen Aufgaben und Knobeleien, die sich natürlich nicht nur auf die Algorithmik begrenzten, sondern uns sogar einen kurzen Einblick in die Digitaltechnik boten, welche das Thema des letztjährigen Informatikkurses war. Außerdem buk er uns sehr leckere Laugenweckle.

Melina Unsere richtig coole und fachlich sehr kompetente Schülermentorin Melina unterstützte uns sehr beim Erlernen unserer Kursinhalte. Außerdem steuerte sie einige Spiele zu unseren Pausen bei. Sie versteht alle unsere Probleme und hat uns stets von Irrwegen abgebracht.

Einleitung

Julia Möhrle

Unter unserem Kursthema „Algorithmik – Die Suche nach dem optimalen Weg“ konnten sich anfangs wahrscheinlich nur sehr wenige Genaueres vorstellen.

Doch bald wurde klar: Das Ziel des Kurses Informatik ist es, innerhalb von zwei Wochen die Software eines Navigationsgerätes zu verstehen und selbst programmieren zu können.

Um zu begreifen, wie so ein komplexes System funktioniert, mussten wir uns mit speziellen Algorithmen beschäftigen.

Bild: Pixabay, CC0

Zuerst schauten wir uns verschiedene Sortier- und Suchalgorithmen an, da diese die Grundlagen eines Navigationsalgorithmus darstellen. Parallel dazu lernten wir die Grundlagen der Programmiersprache C++ kennen, womit wir die einzelnen Algorithmen eigenständig programmieren und testen konnten.

Die ersten komplizierteren Algorithmen, mit denen wir uns befassten, konnten schon zwischen verschiedenen Knoten (Städten) den kürzesten Weg finden. Jedoch waren sie noch nicht sehr benutzerfreundlich und hatten eine lange Laufzeit. Schließlich arbeiteten wir uns zum A*-Algorithmus vor, mit dem ein modernes Navigationsgerät arbeitet. Durch das Ausschließen von irrelevanten Strecken entdeckt er den optimalen Weg deutlich schneller als andere Navigationsalgorithmen. Unsere letzte Herausforderung war es, den A*-Algorithmus auch selbst zu programmieren.

Wir haben uns sehr gefreut, als unser selbstgeschriebenes Programm funktionierte und man zwei beliebige Städte eingeben konnte und der Computer die optimale Route zwischen ihnen anzeigte.

Die Ausgabe des Programms sieht beispielsweise so aus:

Welche Stadt ist der Startpunkt? Stuttgart
Welche Stadt ist der Zielpunkt? Hamburg
Betrachte Stuttgart
Besserer Weg von Stuttgart nach Adelsheim (575 statt 8417 km)

Besserer Weg von Stuttgart nach Konstanz (656 statt 8590 km)
Besserer Weg von Stuttgart nach Freiburg (687 statt 8571 km)
Besserer Weg von Stuttgart nach Ulm (577 statt 8521 km)
Besserer Weg von Stuttgart nach Karlsruhe (557 statt 8472 km)
Besserer Weg von Stuttgart nach Regensburg (778 statt 8480 km)
Besserer Weg von Stuttgart nach Heidelberg (600 statt 8428 km)
Besserer Weg von Stuttgart nach Nuernberg (683 statt 8423 km)

Nun hat das Programm alle Wege aus Stuttgart raus betrachtet. Nicht wundern: Die Entfernungsangaben beziehen sich auf die bisher zurückgelegte Strecke seit Stuttgart *plus der Luftlinie zum Ziel Hamburg*. Alle Werte über 8000 km stellen den Wert „noch nicht berechnet“, also Unendlich dar.

Die Warteschlange ist nach geschätzter Entfernung zum Ziel sortiert:
Platz 1: Karlsruhe mit 557 km
Platz 2: Adelsheim mit 575 km
Platz 3: Ulm mit 577 km
Platz 4: Heidelberg ···

Das Programm fährt mit der nächstgelegenen Stadt fort. Das ist mit 557 km Karlsruhe.

Betrachte Karlsruhe
Besserer Weg von Karlsruhe nach Freiburg (683 statt 776 km)
Weg von Karlsruhe nach Stuttgart ist zu teuer (622 statt 482 km)
Weg von Karlsruhe nach Heidelberg ist zu teuer (600 statt 546 km)

Die Wegschätzung für Freiburg hat sich ein wenig verbessert, sodass Freiburg in der Warteschlange weiter aufrückt. Das Programm fährt natürlich trotzdem mit der nächstgelegenen Stadt fort. Das ist mit 575 km Adelsheim.

Betrachte Adelsheim
Weg von Adelsheim nach Stuttgart ist zu teuer (603 statt 482 km)
Weg von Adelsheim nach Heidelberg ist zu teuer (587 statt 546 km)
Weg von Adelsheim nach Nuernberg ist zu teuer (685 statt 624 km)
Besserer Weg von Adelsheim nach Wuerzburg (581 statt 8386 km)
...
Betrachte Berlin
Weg von Berlin nach Dresden ist zu teuer (1105 statt 971 km)
Weg von Berlin nach Magdeburg ist zu teuer (1068 statt 694 km)
Weg von Berlin nach Lueneburg ist zu teuer (1201 statt 751 km)
Besserer Weg von Berlin nach Rostock (1146 statt 8151 km)
Betrachte Lueneburg
Weg von Lueneburg nach Magdeburg ist zu teuer (945 statt 694 km)
Weg von Lueneburg nach Berlin ist zu teuer (1040 statt 912 km)
Weg von Lueneburg nach Hannover ist zu teuer (877 statt 758 km)
Weg von Lueneburg nach Bremen ist zu teuer (897 statt 833 km)
Besserer Weg von Lueneburg nach Hamburg (813 statt 8000 km)
Besserer Weg von Lueneburg nach Rostock (950 statt 1055 km)

Der kürzeste Weg ist:
Stuttgart
Adelsheim (93 km)
Wuerzburg (164 km)
Erfurt (357 km)
Magdeburg (514 km)
Lueneburg (708 km)
Hamburg
Gesamt: 770 km

Unnötige Wege wie beispielsweise die Strecke Ulm–München wurden durch den cleveren Algorithmus von vornherein ausgeschlossen. Somit gibt uns das Programm den optimalen Weg aus.

C++

Wendelin Verstappen

Die Programmiersprache C++ ist neben Java eine der meistverwendeten objektorientierten Programmiersprachen. Ein Programm ist grundsätzlich folgendermaßen aufgebaut: Zu Beginn teilen wir dem Computer mit, welche Bibliotheken von Befehlen wir benutzen wollen. In unserem Fall ist das zunächst die Bibliothek

<iostream> für die Arbeit mit der Konsole. Das Programm startet mit der *main*-Methode. Hier steht der Code des Hauptprogramms. Das sieht dann so aus:

```
#include <iostream>
using namespace std;

int main(...){
  // Code von der Main
  "//" bedeutet Kommentar
}
```

Unser erstes Programm bestand daraus, einen Text auf dem Bildschirm anzeigen zu lassen. Um dies zu erreichen, verwendeten wir den Befehl `cout` (console out). Die Zeichen nach den zwei spitzen Klammern (`<<`) sollen von der Konsole ausgegeben werden. Um einen Zeilenumbruch zu bewirken, können wir noch *endl* (end line) anhängen, also:

```
cout << "Ausgabetext" << endl;
```

Das Semikolon nach jedem Befehl darf nicht fehlen. Jetzt schreibt der Computer etwas in die Konsole.

Um die Eingabe, also die Informationen, die der Computer von dem Nutzer bekommt, aus der Konsole zu lesen, müssen wir zuerst eine *String*-Variable (Zeichenkette) erstellen:

```
string eingabe;
```

Am Anfang des Codes müssen wir dem Computer noch sagen, dass wir überhaupt Strings verwenden wollen. Das geht so:

```
#include <string>
```

Dann können wir mit `cin` (console in), spitzen Klammern (`>>`) und dem Namen der Stringvariable den eingegebenen Wert in die Zeichenkette „eingabe“ einlesen:

```
cin >> eingabe;
```

So sagen wir, dass etwas vom Nutzer über die Konsole in den Computer und dort in den Speicherplatz gelangt. Das, was wir in die Konsole eingeben, wird also in der Textvariable „eingabe“ gespeichert.

Man kann aber auch andere Sachen in Variablen speichern, zum Beispiel Zahlen: Für ganze Zahlen schreiben wir folgendes. `int` steht für integer = ganze Zahl.

```
int zahl = 5;
```

Mit diesen Variablen kann man auch rechnen:

```
zahl = zahl + 3;
```

bedeutet zum Beispiel, dass zu dem aktuellen Wert der Variable zahl drei addiert werden, also hat sie jetzt den Wert acht. Das Gleiche geht auch mit Kommazahlen. *float* steht für floating point number = Gleitkommazahl:

```
float z = 3.8539;
```

Hierbei ist zu beachten, dass das Komma als Punkt geschrieben wird (englische Schreibweise).

Um komplexere Programme zu schreiben, braucht man Bedingungen. Dazu schreiben wir hinter ein `if` (= falls) die Bedingung in runden Klammern. Das könnte beispielsweise

```
if (zahl == 7)
```

sein, dann schreibt man den Code für eine wahre Bedingung in geschweifte Klammern dahinter. Der Code für eine unwahre Bedingung kommt noch hinter ein `else` ebenfalls in geschweiften Klammern. So sähe dann ein passendes Programm aus:

```
int zahl;
int z;
cin >> zahl;
cin >> z;
if (zahl == z){
     cout << "gleich" << endl;
} else {
     cout << "ungleich" << endl;
}
```

Zu beachten ist der Unterschied zwischen = und ==. Ersteres Symbol bedeutet, dass man einer Variable einen neuen Wert zuschreibt und letzteres vergleicht zwei Werte.

Um Codesegmente zu wiederholen, gibt es das Konstrukt der Schleifen. Die while-Schleife wird solange ausgeführt, bis die Bedingung, die dahinter in Klammern steht, nicht mehr wahr ist. Also erzeugt

```
while (0 == 0)
```

eine Endlosschleife, da die Bedingung „0 ist 0“ immer wahr ist. Wenn eine Schleife eine festgelegte Anzahl an Wiederholungen durchlaufen werden soll, verwendet man am besten eine for-Schleife:

```
for (int zaehl = 1; zaehl <= 10; zaehl++){
    cout << zaehl << endl;
}
```

Hier bestimmen wir in den runden Klammern als erstes, mit welcher Variable wir arbeiten wollen, in diesem Fall erstellen wir dazu eine Variable „zaehl“, die als Zählvariable für die Anzahl der Schleifendurchläufe dient. Dann nennen wir die Bedingung, wie lange die Schleife ausgeführt werden soll und schlussendlich noch, was mit der Variable nach jedem Durchlauf passieren soll. Eine Kurzschreibweise für

```
zaehl = zaehl + 1
```

ist

```
zaehl++
```

Also gibt dieses Programm die Zahlen von 1 bis 10 aus.

BubbleSort und SelectionSort

Mario Feifel

Bereits beim Einführungswochenende sammelten wir einige Ideen, wie wir eine ungeordnete Liste von Zahlen mittels eines Algorithmus sortieren könnten. Dabei wurden zwei Verfahren vorgeschlagen, die wir am Anfang der Akademie tatsächlich bearbeiteten. Diese sind der *SelectionSort* und der *BubbleSort*. Da sie zwei relativ einfach zu programmierende Algorithmen sind, dienten sie uns als Einstieg in die Welt des Programmierens.

Der *SelectionSort* geht sehr simpel vor. In einer ungeordneten Liste von Zahlen geht er zuerst alle Zahlen durch und sucht dabei die mit dem kleinsten Wert. Die gefundene Zahl wird dann an die erste Stelle getauscht. Folglich steht die kleinste Zahl am Anfang und damit an der korrekten Stelle. Deshalb wird sie im weiteren Verlauf nicht weiter beachtet.

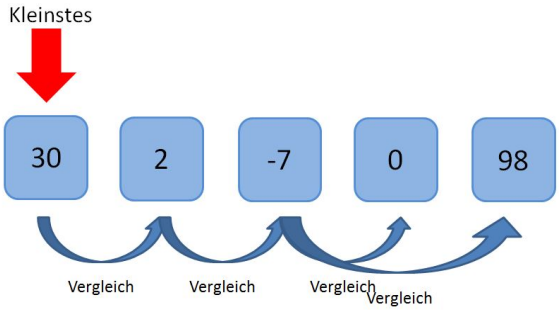
Stattdessen fährt der Algorithmus damit fort, die nun verbliebenen Zahlen durchzusehen, um wiederum die kleinste enthaltene Zahl zu finden. Diese wird jetzt an die zweite Stelle der Liste getauscht, hinter die kleinste Zahl aus dem vorhergehenden Durchlauf. Damit gilt auch die zweite Zahl als richtig einsortiert und wird nicht weiter beachtet. Der Algorithmus fährt letztendlich solange mit diesem Verfahren fort, bis jede Zahl an die richtige Stelle gesetzt wurde.

So sieht der Pseudocode für den SelectionSort aus (entnommen aus der deutschsprachigen Wikipedia):

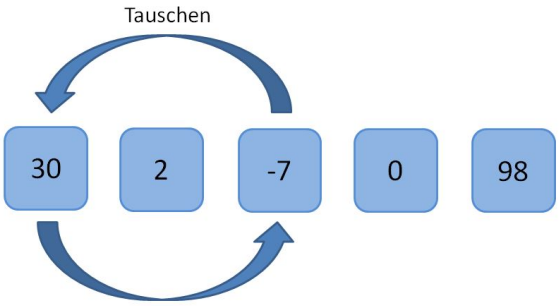
```
prozedur SelectionSort

( A : Liste sortierbarer Elemente )
  n = Laenge(A) - 1
  links = 0
  wiederhole
    min = links
    fuer jedes i von links+1 bis n wiederhole
      falls A[i] < A[min] dann
          min = i
      ende falls
    ende fuer
    Vertausche A[min] und A[links]
    links = links + 1
  solange links < n
prozedur ende
```

Im Gegensatz zum SelectionSort sucht der *BubbleSort* nicht eine bestimmte Zahl (die größte bzw. kleinste), sondern betrachtet zunächst lediglich die ersten beiden Werte der Liste.

Von diesen beiden tauscht er den größeren Wert an die hintere Stelle oder lässt ihn an seiner



SelectionSort: Suchen der kleinsten Zahl



SelectionSort: Tauschen mit der ersten Zahl

derzeitigen Position stehen, wenn die beiden Werte bereits richtig geordnet sind.

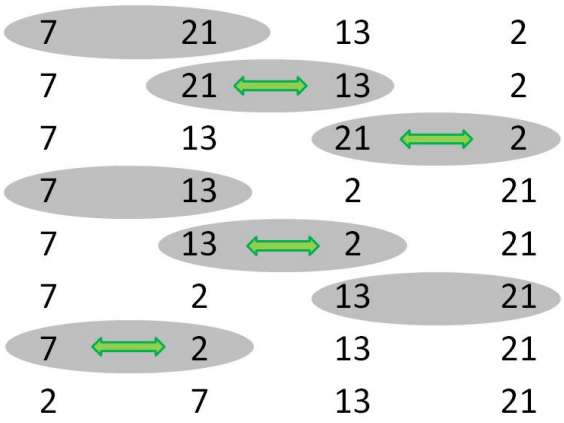
Danach betrachtet der Algorithmus den hinteren (größeren) Wert und dessen Nachfolger in der Liste. Hiervon wird ebenfalls wieder der größere Wert nach hinten getauscht und der Algorithmus rückt wieder eine Stelle weiter.

Nach einem kompletten Durchlauf steht nun die größte Zahl am Ende der Liste, da sie wie eine Blase (Bubble) von ihrer ursprünglichen Stelle in der Liste aufgestiegen ist. Daher kommt auch der Name „BubbleSort“.

Mit dem Rest der Zahlen wird ebenso verfahren, bis nach und nach die größten Blasen (Zahlen) nach hinten „aufgestiegen“ sind und die Liste in der richtigen Reihenfolge steht.

So sieht der Pseudocode für den BubbleSort aus (entnommen aus der englischsprachigen Wikipedia):

```
BubbleSort(Array A)
  n = A.size
  do{
    swapped = false
    for (i=0; i<n-1; ++i){
      if (A[i] > A[i+1]){
        A.swap(i, i+1)
        swapped = true
    } // ende if
  } // ende for
  n = n-1
} while (swapped)
```



Die Funktionsweise des BubbleSort

Das Problem bei beiden Algorithmen ist, dass sie eine relativ lange Laufzeit haben. Da die Algorithmen bei n Zahlen alle vorhandenen Zahlen (n) durchgehen müssen, um eine einzige Stelle zu ordnen, müssen sie dies für jede Zahl in der Liste (n mal) machen. Daraus ergibt sich, dass der Algorithmus ungefähr n^2 Schritte durchführen muss, was bei einer großen Anzahl von Eingabewerten schnell zu einer „Laufzeitexplosion“ führt. Bei sehr großen Zahlenmengen ist eine Programmlaufzeit von einigen Jahrzehnten möglich!

Im besten Fall, also bei einer bereits sortierten Liste, muss der Algorithmus jedes Element der Liste nur ein Mal anschauen (also n mal). Er hat in diesem Spezialfall eine Laufzeit von n.

Die beiden hier beschriebenen Algorithmen kommen praktisch eher selten zum Einsatz, da es bereits wesentlich ausgereiftere Sortierverfahren gibt, wie beispielsweise den Quicksort.

QuickSort

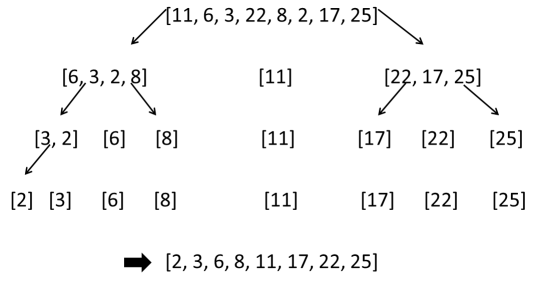
Julia Möhrle

Der QuickSort gilt als einer der schnellsten Sortieralgorithmen der Welt und nutzt dabei das „Divide and Conquer“-Prinzip. Bei diesem Prinzip wird das eigentliche große Problem in kleinere Teilprobleme zerlegt, um diese leichter

lösen zu können. Anschließend werden die einzelnen Teillösungen zu einer Gesamtlösung für das Problem zusammengefasst.

Um zum Beispiel Zahlen in einer ungeordneten Liste zu sortieren, nimmt der QuickSort die an erster Stelle stehende Zahl und setzt diese im ersten Schritt als sogenanntes Pivot-Element in die Mitte. Danach sortiert er alle anderen Zahlen jeweils rechts oder links davon ein, rechts wenn sie größer sind, links wenn sie kleiner sind. Da wir nun drei einzelne Listen haben, wird aus den zwei äußeren jeweils die erste Zahl als neues Pivot-Element bestimmt. Um diese werden nun die anderen Zahlen der Liste, von der sie kommen, sortiert, während das schon benutzte und einsortierte Pivot-Element nicht mehr beachtet wird und an seiner Stelle bleibt. Der Algorithmus führt diesen Durchlauf immer weiter fort, bis nur noch eine Zahl in jeder Liste steht.

Ist dies erfolgreich abgeschlossen, werden die einzelnen Listen – also die Teillösungen – in der Reihenfolge, wie sie nun sortiert vorliegen, zu einer Liste aus sortierten Zahlen zusammengefasst. Der QuickSort halbiert die Zahlenketten in seiner speziellen Weise demnach so lange, bis jede einzeln als Pivot-Element an der richtigen Stelle vorliegt.



Ablauf eines QuickSort-Vorgangs

Der Algorithmus hat bei verschiedenen Fällen verschiedene Laufzeiten.

Wenn der Algorithmus die mittlere Zahl der sortierten Liste als Pivot-Element nimmt, dann ist das der optimale Fall. In diesem Fall hat der Algorithmus eine Laufzeit von $O(n * \log n)$, wobei n die Anzahl der Eingabewerte ist.

Im durchschnittlichen Fall braucht der Algorithmus auch $O(n * \log n)$ (siehe O-Notation). Dieser Fall tritt auf, wenn der Algorithmus teilweise das optimale Pivot-Element wählt, aber teilweise ein ungünstiges Element.

Ungünstig ist es, wenn das Element in der sortierten Liste direkt vor oder nach dem neuen Pivot-Element steht. Es gibt auch einen schlimmsten Fall, in dem der Algorithmus eine Laufzeit von $O(n^2)$ hat. Dies tritt auf, wenn die komplette Liste schon sortiert ist. Denn dann muss der Algorithmus bei jeder Zahl überprüfen, ob es kleiner oder größer ist als das momentane Pivot-Element, obwohl die anderen Elemente, die noch nicht als Pivot-Element festgelegt sind, größer sind.

In C++ ist der QuickSort schon implementiert. Um ihn zu verwenden muss man nur folgende Struktur verwenden:

```
#include <stdio.h>
#include <stdlib.h>
```

Diese zwei Zeilen braucht man um den QuickSort in C++ verwenden zu können.

```
int values[] = { 40, 10, 100, 90, 20, 25 };
```

Damit erzeugt man eine Liste, die man später sortieren soll.

```
int compare (const void * a, const void * b)
{
  return ( *(int*)a - *(int*)b );
}
```

Die Funktion compare braucht man für den QuickSort als Vergleichsfunktion zweier Werte.

```
qsort (values, 6, sizeof(int), compare);
```

Damit sortiert man eine Liste mit dem QuickSort.

```
for (n=0; n<6; n++) {
   cout << values[n];
}
```

Damit gibt man die sortierte Liste aus.

O-Notation

Lennart Ruhrmann

Die O-Notation ist ein mathematisches Hilfsmittel, das uns die Möglichkeit gibt, die Laufzeiten verschiedener Algorithmen zu vergleichen. Daher kommt auch der Name O-Notation, O steht hierbei nämlich für Ordnung.

Die mathematische Definition der O-Notation lautet:

$f \in \mathcal{O}(g) \Leftrightarrow \exists\ C > 0\ \exists\ n_0 > 0\ \forall\ n > n_0 : |f(n)| \leq C \cdot |g(n)|$

Das bedeutet: Wenn $f \in \mathcal{O}(g)$, dann finden wir eine Konstante C sowie eine Untergrenze n_0, ab der $f(n)$ immer kleiner oder gleich $C \cdot |g(n)|$ ist. Anders gesagt kann $f(n)$ für große Werte von n niemals stärker als $g(n)$ wachsen. Die Umkehrung gilt auch. Wenn wir eine entsprechende Konstante und eine Untergrenze finden, haben wir bewiesen, dass $f \in \mathcal{O}(g)$.

In einem Beispiel wollen wir zeigen, dass gilt: $f(n) = 3 \cdot n^3 - 5 \cdot n^2 + 6 \in \mathcal{O}(n^3)$

Dazu wählen wir beispielsweise folgende Werte: $C = 4$ und $n_0 = 1$. Es gilt: $f(2) = 3 \cdot 2^3 - 5 \cdot 2^2 + 6 \leq 4 \cdot 2^3 \Rightarrow 24 - 20 + 6 \leq 32 \Rightarrow 10 \leq 32$

Für $n > 2$ wird der Abstand noch größer, was wir aus Platzgründen in dieser Doku nicht beweisen. Auf jeden Fall ist damit gezeigt, dass $f \in \mathcal{O}(n^3)$

Um das Ganze zu vereinfachen, lässt sich die Operation so beschreiben, dass man in einer Funktion, beispielsweise $f(n) = 3 * n^3 + 7n - 1$ den Summanden mit dem höchsten Exponenten sucht.

Folglich wäre die O-Notation dieser Funktion $O(n^3)$, da 3 der höchste Exponent ist. Das lässt sich damit begründen, dass bei einer Eingabemenge, die gegen Unendlich geht, der Funktionswert ungefähr n^3 ergibt. Die weiteren Faktoren und Summanden $*3 + 7n - 1$ können daher vernachlässigt werden.

Dieses Verfahren haben wir verwendet, um den Rechenaufwand (Laufzeit) unserer Sortieralgorithmen zu vergleichen.

Der Rechenaufwand des BubbleSorts sowie des SelectionSorts, befindet sich bei n Eingaben in der Größenordnung $O(n^2)$, der Rechenaufwand des QuickSort jedoch in $O(n * \log n)$. Dies hat zufolge, dass bei einer großen Anzahl von Eingabewerten die Laufzeiten von Bubble- und SelctionSort um ein Vielfaches größer sind, als die des QuickSort.

Eingabewerte	Bubblesort	Quicksort
1.000	1 s	1 s
10.000	100 s	14 s
1.000.000	11,5 Tage	33 Minuten
10 Mio.	4 Jahre	6,5 Stunden
100 Mio.	400 Jahre	3 Tage

Vergleich Bubbblesort $O(n^2)$, Quicksort $O(n * \log n)$

Man kann sofort erkennen, dass eine Laufzeit von 4 Jahren, wie sie beim BubbleSort schnell erreicht ist, überhaupt nicht praktikabel ist. Daher wird beim Sortieren von großen Listen hauptsächlich der QuickSort verwendet.

Graphen

Henriette Neuschwander

Um die Algorithmen, mit denen wir uns zum Erreichen unseres Kursziels befassten, besser verstehen zu können, wollten wir diese visualisieren. Dafür benötigten wir Graphen.

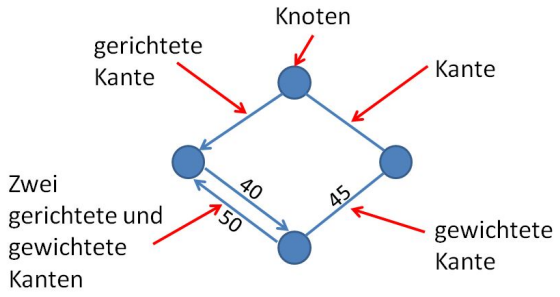
Graphen sind aus Knoten und Kanten bestehende Strukturen. Sie stellen die Beziehung zwischen den einzelnen Objekten (Knoten) mit Hilfe von Verbindungen (Kanten) dazwischen dar. Knoten werden durch Punkte bzw. Kreise dargestellt und können in unseren Visualisierungen beispielsweise für Städte oder Wegkreuzungen stehen. Kanten werden durch Pfeile oder Linien dargestellt und können beispielsweise Entfernungen oder Straßen repräsentieren. Es gibt gewichtete, gerichtete und gewichtet-gerichtete Kanten.

Als *gewichtete* Kanten werden Kanten bezeichnet, die mit einem Wert als Kantengewicht beziehungsweise Kosten versehen werden. Die Kosten stehen normalerweise direkt über oder unter den Pfeilen, die die Verbindung repräsentieren, zu der sie gehören. Kosten können in unserem Fall zum Beispiel für Treibstoffkosten, Mautgebühren, Zeit oder Entfernungen stehen.

Gerichtete Kanten stehen für Verbindungen, die nur in eine Richtung bestehen, wie beispielsweise Einbahnstraßen. Statt der sonst üblichen

Linien werden sie durch Pfeile, die in die entsprechende Richtung zeigen, repräsentiert.

Gerichtete und gewichtete Kanten stehen für Verbindungen, die sowohl nur in eine Richtung bestehen, als auch ein Kantengewicht haben. Sie werden durch mit Werten beschriftete Pfeile dargestellt.



Visualisierung verschiedener Kantentypen

Planare Graphen sind Graphen, deren Knoten und Kanten so angeordnet sind, dass sich die Kanten beim Anordnen innerhalb einer Ebene nicht überlagern.

Einfache Graphen (auch simple Graphen) sind ungerichtet und besitzten weder Mehrfachkanten noch Schleifen.

In einem *Multigraphen* dürfen dagegen zwei Knoten auch durch mehrere Kanten verbunden sein und Schleifen existieren.

Als *Digraphen* (auch orientierte oder gerichtete Graphen) werden Graphen bezeichnet, bei denen die Kanten gerichtet sind.

Mithilfe der Graphen visualisierten wir den Großteil der von uns programmierten Algorithmen, was besonders zum guten Verständnis dieser während des Kurses wie auch in unserer Rotations- und Abschlusspräsentation beitrug.

Als Beispiel das *Traveling Salesman Problem* (TSP):

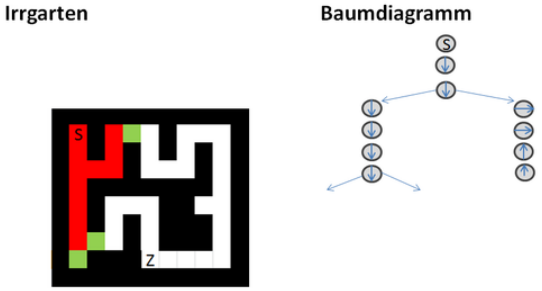
Das Problem besteht darin, den kürzesten Weg zwischen einer bestimmten Anzahl von Städten zu finden, wobei die Städte die Knoten und die Wege gewichtete Kanten sind. Die Schwierigkeit dabei ist, dass jede Stadt nur einmal besucht werden darf; bereits bei zehn Städten sind potenziell 1.814.400 verschiedene solcher Rundtouren möglich. Bei 15 Städten steigt die Zahl der Möglichkeiten bereits auf den ungeheuren Wert 653.837.184.000.

Breiten- und Tiefensuche

Lene Sperling

Die Breitensuche und die Tiefensuche sind zwei Suchalgorithmen, deren Ziel es ist, einen Knoten in einem Graphen zu finden.

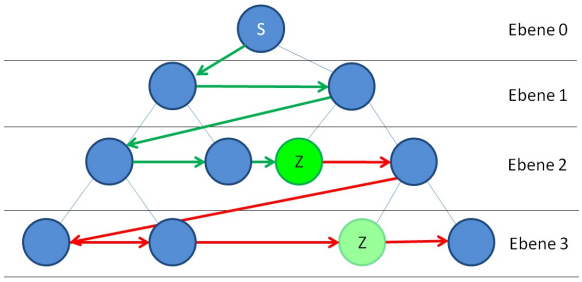
Bei der *Breitensuche* ist der Graph in mehrere Ebenen unterteilt. Es wird jede Ebene nacheinander nach dem Zielknoten durchsucht. Er beginnt in der obersten Ebene und untersucht von links nach rechts alle Knoten, bis der Zielknoten gefunden wird. Falls dieser in der obersten Ebene nicht gefunden wird, wird die nächste Ebene mit dem selben Verfahren betrachtet. Wird der Zielknoten gefunden, so ist diese Lösung mit Sicherheit auch der kürzeste mögliche Weg. Dies wird sichergestellt, da der Algorithmus eine Ebene nach der anderen durchsucht und es somit keinen kürzeren Weg geben kann. Die Breitensuche wird verwendet, wenn es verschiedene Lösungswege gibt und man herausfinden muss, welcher der Schnellste ist. Ein Beispiel dafür ist ein Irrgarten, bei dem es mehrere Lösungswege gibt und der Kürzeste gefunden werden soll.



Irrgarten mit Baumdiagramm

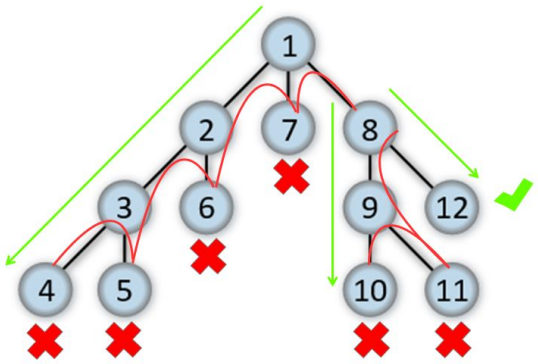
Der Algorithmus probiert alle möglichen Wege gleichzeitig aus, indem er immer einen Schritt in alle möglichen Richtungen geht und überprüft, ob er schon am Ziel angekommen ist. Sobald er an eine Wand stößt oder sich der eingeschlagene Weg als Irrweg entpuppt, wird aufgehört, diesen Pfad weiterhin zu untersuchen. Sobald ein Weg am Ziel angekommen ist, stoppt der Algorithmus, da er den schnellsten Weg gefunden hat.

Die *Tiefensuche* versucht ebenfalls, einen Zielknoten zu finden. Allerdings ist der Graph hierbei nicht in Ebenen, sondern in Spalten



Suchbaum der Breitensuche

unterteilt. Der Algorithmus untersucht zu allererst den Weg ganz links bis zum am tiefsten liegenden Knoten. Sollte der Zielknoten nicht gefunden werden, so wird einen Schritt zurück gegangen und der weiter rechts liegende Pfad überprüft. Diese Schritte werden so lange wiederholt, bis der Zielknoten gefunden wird. Dies kann, wenn man Glück hat und der Zielknoten sich sehr weit links im Baum befindet, sehr schnell gehen. Liegt der Knoten weiter rechts im Baum, kann es aber auch sehr lange dauern. Wenn der Zielknoten gefunden ist, stoppt der Algorithmus. Das Problem hierbei ist, dass die gefundene Lösung nicht mit Sicherheit der schnellste Weg ist. Ein viel schnellerer Weg könnte sich rechts im Baum befinden. Da der Algorithmus aber vorher beendet wird, wird dieser nicht einmal angeschaut.



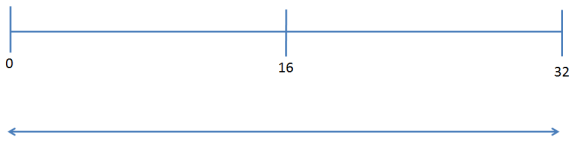
Suchbaum der Tiefensuche

Binäre Suche

Natalie Riegger

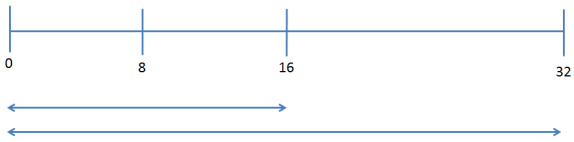
Bei der binären Suche starteten wir mit einem kleinen Zahlenratespiel, dessen Ziel es war, Michaels gedachte Zahl zwischen 1 und 1000 herauszufinden. Anfangs haben wir nur willkürlich Zahlen in den Raum geworfen. Wir merkten jedoch schnell, dass diese Strategie die Falsche ist. Wir erkannten, dass wir fragen müssen, ob seine Zahl größer oder kleiner als unsere ist, wie es auch bei der binären Suche passiert. Die binäre Suche findet ein Element in einer sortierten Liste. Dabei arbeitet sie, wie der Quicksort-Algorithmus, mit dem „Divide & Conquer“-Prinzip.

Wenn man eine Zahl in einer geordneten Liste sucht, überprüft man, ob die gesuchte Zahl größer oder kleiner als deren Mittelwert ist. Nun wird nur noch die Hälfte der Liste betrachtet, in der sich das gesuchte Element befindet. Die binäre Suche wird nun auf die neue, kleinere Liste angewendet. Durch dieses Verfahren wird bei jedem Schritt die Hälfte der Zahlen aussortiert. Das Verfahren wird so lange wiederholt, bis nur noch ein Element übrigbleibt. Ist dieses nicht das richtige, dann ist das gesuchte Element in der Liste nicht vorhanden.



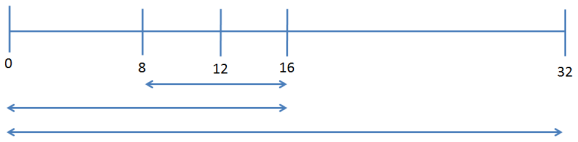
Das Prinzip der Binären Suche: Es befinden sich 32 Elemente in der Liste.

Beispiel: Die geordnete Liste besteht aus 32 nacheinanderfolgenden Zahlen. Die gesuchte Zahl ist 12. Dann würde der Computer zuerst fragen, ob die Zahl größer oder kleiner als 16, also der Hälfte von 32, ist. Da 12 kleiner als 16 ist, wird darauf die Frage folgen, ob die Zahl größer oder kleiner als 8 ist. Da 12 größer als 8 ist, wird als nächstes die Mitte im Intervall von 8 bis 16 genommen, also 12. Danach wird gefragt, ob die gesuchte Zahl größer oder kleiner als 12 ist. Da dies schon die gedachte Zahl ist, stoppt das Programm.



Das Prinzip der Binären Suche: Es befinden sich nur noch 16 Elemente in der Liste

Durch dieses Suchverfahren wird schnell Auskunft über das gesuchte Element und dessen Vorhandensein geliefert, ohne sich jedes einzelne Element anschauen zu müssen. Das Suchverfahren ist dann besonders lohnend, wenn sich eine große Menge an Zahlen in der Liste befindet. Bei einer Liste mit einer Million Zahlen müssen maximal zwanzig Suchvorgänge erfolgen ($1.000.000 < 2^{20}$).



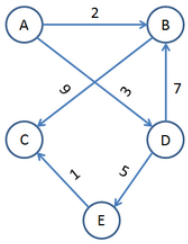
Das Prinzip der Binären Suche: Es befinden sich nur noch 8 Elemente in der Liste

Der Floyd-Warshall-Algorithmus

Stefani Asenova

Mit dem Floyd-Warshall-Algorithmus, nach Robert Floyd und Stephen Warshall, lassen sich die kürzesten Wege zwischen allen Knoten in einem Graphen berechnen.

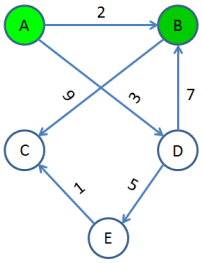
Zu Beginn werden die Knotengewichte, in unserem Fall die Distanz zweier Knoten, aus dem gegebenen Graphen abgelesen und in eine Matrix übertragen. Da die Länge des Weges, beziehungsweise das Gewicht der Kante, von einem Knoten zu sich selbst immer Null beträgt, wird in das entsprechende Feld eine 0 eingetragen.



/	A	B	C	D	E
A	0	2	inf	3	inf
B	inf	0	9	inf	inf
C	inf	inf	0	inf	inf
D	inf	7	inf	0	5
E	inf	inf	1	inf	0

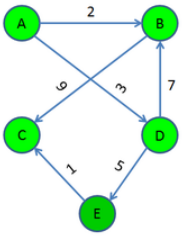
Das Prinzip des Floyd-Warshall-Algorithmus und dessen Visualisierung

Danach werden die Distanzen zwischen den direkten Nachbarknoten hinzugefügt. Sind zwei Knoten keine direkte Nachbarn, so wird in das Feld „Unendlich“, also *inf* für „infinite“ geschrieben. Nachdem diese Schritte für direkte Kanten zwischen allen Knoten gemacht wurden, kann der eigentliche Algorithmus beginnen. Anfangs wird der erste Knoten, im Bild Knoten B, „freigegeben“. Es wird geprüft, ob Knoten B als Brückenknoten zwischen zwei anderen Knoten genutzt werden kann. In diesem Fall zwischen Knoten A und Knoten C.



/	A	B	C	D	E
A	0	2	11	3	inf
B	inf	0	9	inf	inf
C	inf	inf	0	inf	inf
D	inf	7	16	0	5
E	inf	inf	1	inf	0

Floyd-Warshall-Algorithmus nach dem zweitem Schritt



/	A	B	C	D	E
A	0	2	9	3	8
B	inf	0	9	inf	inf
C	inf	inf	0	inf	inf
D	inf	7	6	0	5
E	inf	inf	1	inf	0

Vollständig ausgefürter Floyd-Warshall-Algorithmus

Damit dies der Fall ist, müssen zwei Bedingungen erfüllt sein:

1. Es müssen gerichtete Kanten an Knoten B ankommen und davon abzweigen.

2. Die Distanz von Knoten A nach Knoten C über Knoten B muss kleiner sein, als die direkte Distanz von Knoten A nach Knoten C.

Da es bisher keine direkte Verbindung gibt und somit „Unendlich“ in der Matrix steht, ist der Umweg über B kürzer. Wenn beide Bedingungen erfüllt sind, wird die Entfernung zwischen Knoten A und C überschrieben. Es werden nacheinander alle Knoten des Graphen „freigegeben“, bis es keine mehr gibt.

Den Floyd-Warshall Algorithmus haben wir auch selbst programmiert. Er wird allerdings in modernen Navigationsgeräten nicht genutzt, da er eine sehr lange Laufzeit hat ($O(n^3)$). Bei 10.000 Städten hätten wir eine Rechenzeit von mehreren Tagen.

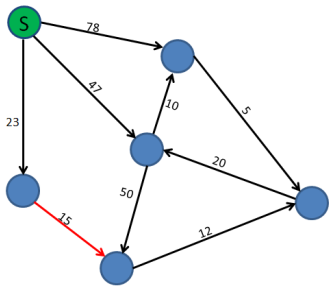
Der Bellman-Ford-Algorithmus

Dennis Rudik

Der Bellman-Ford-Algorithmus ist ein Suchalgorithmus von Richard Bellman und Lester Ford, der die kürzesten Distanzen von einem Startknoten zu allen anderen Knoten berechnet.

Dabei geht der Algorithmus nacheinander alle Kanten ab. Wenn er einmal alle Kanten untersucht hat, beginnt er wieder damit, die Kanten in der gleichen Reihenfolge zu untersuchen.

Beim ersten Durchgang kommt man an die Knoten, die nur eine Kante vom Startknoten entfernt sind. Nach dem zweiten Durchgang kommt man an die Knoten, die maximal zwei Kanten vom Startpunkt entfernt sind.

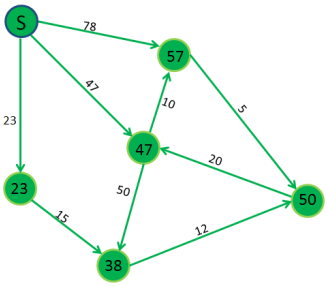


Erste Kante im ersten Durchgang

Am Anfang sind die Werte aller Knoten auf Infinite, also Unendlich, gesetzt. Wenn nämlich ein Knoten nicht zu erreichen ist, sind die Kosten, die man aufbringen muss um ihn zu erreichen unendlich. Beim Überprüfen einer Kante wird untersucht, ob die Summe der Kosten der aktuellen Kante und ihres Ausgangsknotens geringer ist als die momentanen Kosten des Knotens, zu dem die Kante führt. Wenn ja, dann wird der alte Wert überschrieben. Der Algorithmus wird solange wiederholt, bis sich die Kosten keines Knotens mehr ändern.

Wenn man von einem Knoten zu einem anderen kommt und man dabei n Knoten besucht, geht man über $n-1$ Kanten. Falls man einen Knoten nur über alle anderen Knoten erreicht, muss man die Kanten $n-1$ mal durchgehen. Deshalb sind maximal $n-1$ Durchgänge nötig, um alle Knoten zu erreichen. Da es schwer zu erkennen ist, ob ein Knoten nur über alle anderen erreicht werden kann, muss der Algorithmus $n-1$ Wiederholungen machen, damit er ganz sicher alle Möglichkeiten ausprobiert hat.

Der Algorithmus läuft in $O(n*m)$ wobei n die Anzahl der Knoten ist und m die Anzahl der Kanten. Wenn man von jedem Knoten zu jedem anderen Knoten eines Graphen die kürzesten Wege berechnen will, läuft der Algorithmus in $O(n^2*m)$, da er dann n-mal durchgeführt werden muss.



Letzter Schritt im letzten Durchgang

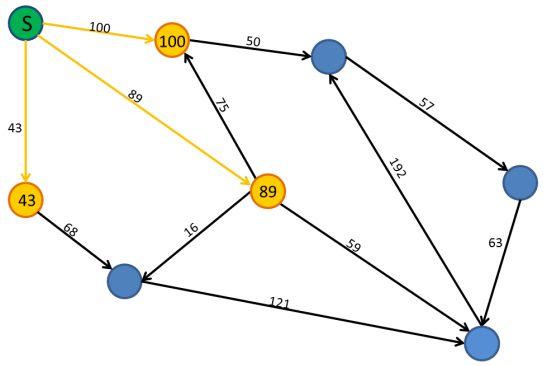
Eine Besonderheit des Bellman-Ford-Algorithmus ist, dass er auch mit negativen Gewichten auf den Kanten umgehen kann. Kreisläufe negativen Gewichtes, die vom Startknoten aus erreichbar sind, führen zu ständig günstiger werdenden Kosten. Daher müssen sie ausgeschlossen werden, denn andernfalls könnten sie beliebig oft durchlaufen werden. Es gäbe folglich überhaupt keinen Weg geringsten Gewichtes. Deshalb muss der Algorithmus Kreisläufe negativen Gewichtes erkennen und ausschliessen.

Der Dijkstra-Algorithmus

Amelie Amann

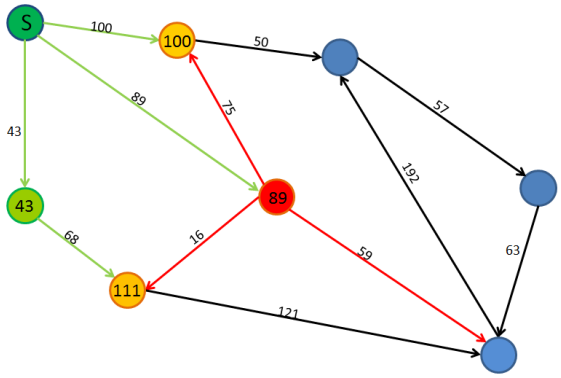
Der Dijkstra-Algorithmus ermittelt, wie der Bellman-Ford, die kürzesten Distanzen von einem ausgewählten Startknoten zu allen anderen Knoten. Dazu beginnt er beim Startknoten, der in eine Warteliste eingefügt wird und dessen Distanz zu sich selbst auf „0“ gesetzt wird. Danach wird der Startknoten wieder aus der Warteliste entnommen, da er nun besucht wurde. Die Gewichte, in unserem Fall die Distanzen, zu allen anderen Knoten werden anfangs

auf „Unendlich“ gesetzt. Nach dem Startknoten betrachtet der Algorithmus alle Knoten, die eine direkte Verbindung zu ihm besitzen. Diese werden daraufhin auch in die Warteliste eingefügt. Er beginnt mit dem Knoten, dessen Distanz zum Startknoten am kürzesten ist. Der vorherige Wert „Unendlich“ wird mit der Distanz vom Startknoten zum aktuellen Knoten überschrieben.



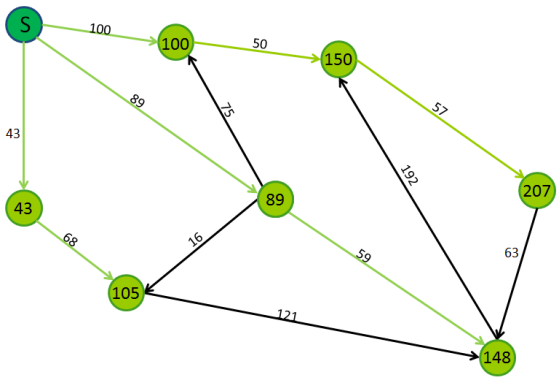
Erster Schritt des Algorithmus

Danach schaut sich der Algorithmus auch beim dem soeben betrachteten Knoten die angrenzenden Knoten an und fügt diese in die Warteliste ein. Es werden alle Kanten, ausgehend vom aktuellen Knoten betrachtet. Die folgenden Knoten erhalten den Wert des Gewichtes der verbindenden Kante addiert und mit dem Gewicht des aktuellen Knotens. Der aktuelle Knoten wird jetzt aus der Warteliste entnommen, da er nun besucht und bearbeitet wurde.



Dritter Schritt des Algorithmus

Daraufhin werden alle ermittelten Streckenlängen verglichen und die insgesamt kürzeste Strecke gewählt, um den beschriebenen Vorgang zu wiederholen. Wenn dabei eine Strecke entdeckt wird, über die der Weg vom Startknoten zu einem anderen Knoten kürzer ist als der bisherige, wird die alte Route und die entsprechende Länge dieser Route durch die kürzere neue Route überschrieben. Der Algorithmus geht so lange alle möglichen Wege durch, bis es keine noch nicht besuchten Knoten mehr gibt, sich also keine Knoten mehr in der Warteliste befinden. Damit wurden die kürzesten Strecken zu allen Knoten gefunden.



Fertig durchlaufener Dijkstra-Algorithmus

Pseudocode des Dijkstra-Algorithmus:

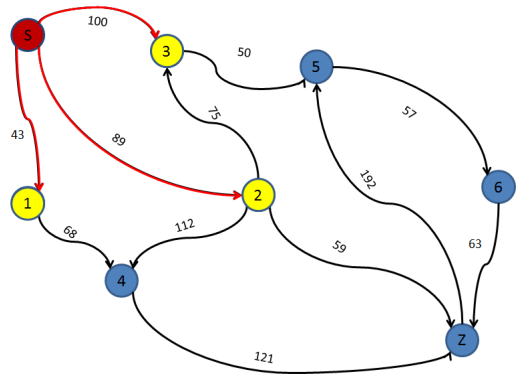
```
Funktion Dijkstra(Graph, Startknoten):
  initialisiere(Graph,Startknoten,abstand[],
                vorgaenger[],Q)

  // Der eigentliche Algorithmus
  solange Q nicht leer:
     u:= Knoten in Q mit kleinstem Wert
        in abstand[]

     // fuer u ist der kuerzeste Weg nun
        bestimmt
     entferne u aus Q

     fuer jeden Nachbarn v von u:
        falls v in Q:

           // pruefe Abstand vom
           // Startknoten zu v'
           distanz_update(u,v,abstand[],
                          vorgaenger[])
  return vorgaenger[]
```

Der A*-Algorithmus

Alicia Appelhagen

Der A*-Algorithmus (gesprochen „A-Stern-Algorithmus“) ist eine Variante des Dijkstra-Algorithmus. Die Funktionsweise des A*-Algorithmus ist der des Dijkstra-Algorithmus ähnlich. Der große Unterschied ist, dass beim Dijkstra-Algorithmus der aktuell kürzeste Weg bis zu dem im Moment betrachteten Knoten angeschaut wird. Beim A* hingegen werden mittels einer Schätzung irrelevante Wege ausgeschlossen. Das sind beispielsweise Wege, die in die falsche Richtung gehen.

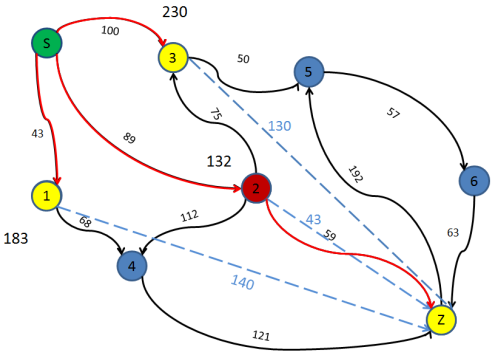
Der A*-Algorithmus verwendet zur Entscheidung, welchen Knoten er als Nächstes besucht – wie der Dijkstra-Algorithmus – die Entfernung zum Startknoten, zusätzlich aber auch eine optimistisch geschätzte Entfernung jedes Knotens zum Ziel. Diese optimistische Schätzung nennt man Heuristik. Mit ihr können viele Knoten ausgeschloßen werden, die zwar nah beim Start liegen, sich jedoch nicht dem Ziel nähern, sondern entfernen. Für die Richtigkeit der Ergebnisse des A*-Algorithmus ist wichtig, dass die geschätzte Strecke zum Ziel die tatsächliche Entfernung nie überschreitet. Wäre dies nämlich der Fall, würden eventuell Knoten fälschlicherweise ausgeschlossen. Als Schätzung bietet sich in der Navigation die Luftlinie eines Knoten zum Ziel an.



A*-Algorithmus: Entfernung zum Startknoten

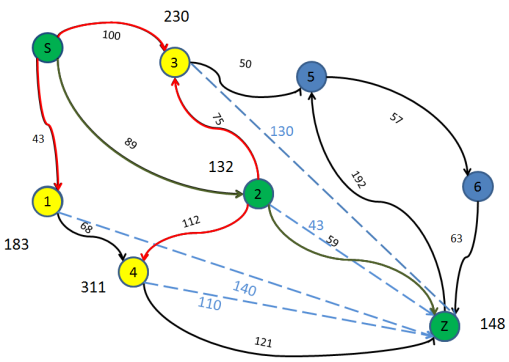
Die Funktionsweise des Algorithmus besteht darin, dass zunächst alle direkt erreichbaren Nachbarknoten des Startknotens auf eine Warteliste gesetzt werden. Dann wird die Luftlinie von jedem dieser Knoten zum Zielknoten ermittelt.

Die Summe der tatsächlichen Entfernung eines Knotens zum Start und seiner Luftlinie zum Ziel ergibt den sogenannten f-Wert, welcher ausschlaggebend ist für das weitere Vorgehen des Algorithmus. Denn in der Warteliste wird der Knoten nach vorn gesetzt und somit als nächstes bearbeitet, der den geringsten f-Wert hat. Bearbeiten bedeutet, dass nun auch die f-Werte der Nachbarn des aktuellen Knoten bestimmt werden. Dann wird wieder der Knoten mit dem kleinsten f-Wert bearbeitet.



A*-Algorithmus: Neue f-Werte

Dieser Vorgang wird wiederholt, bis entweder der Zielknoten auf die Warteliste gesetzt und wieder entnommen wurde oder die Warteschlange leer ist, wobei der Zielknoten nicht erreicht wurde. In letzterem Fall gibt es keinen Weg zum Ziel.



A*-Algorithmus: Zweiter Durchlauf – der Weg zum Ziel ist sichtbar

Exkursion

Patrick Ng

Für unseren Kursausflug trafen wir uns am Montag, den 5.9.2016 um 08:05 Uhr vor dem Eckenberg-Gymnasium. Als wir vollzählig waren, machten wir erstmal ein bisschen Frühsport, da wir mit schnellem Tempo zum Bahnhof laufen mussten, von wo aus wir den Zug nach Heidelberg nahmen. Nachdem wir einmal umgestiegen waren, kamen wir um 10:00 Uhr am Heidelberger Hauptbahnhof an. Anschließend gingen wir zu dem nahe am Bahnhof gelegenen McDonald's, da wir noch etwas Zeit bis zu unserem Termin hatten. Dort aßen wir gemeinsam ein zweites Früstück und nutzten die Zeit, um Michaels Rätsel zu lösen. Nachdem wir alle etwas Warmes gegessen hatten, machten wir uns mit Bus und S-Bahn auf den Weg zum Institut für Software Engineering.

Dort hatten wir um 11:30 Uhr einen Workshop zum Thema „Prinzipien und Techniken für saubereren Code“, geleitet von Herrn Seiler und Frau Prof. Paech. Zu Beginn lernten wir, welche Vorteile sauberer Code mit sich bringt. Man behält dadurch einen bessseren Überblick und der Programmcode wird auch für Außenstehende verständlich. Desweiteren wurde uns gezeigt auf welche häufigen Fehlerquellen wir besonders achten sollen. Oft haben Zeitdruck und Desinteresse einen negativen Einfluss auf die Qualität des Programms.

Anschließend diskutierten wir darüber, was sauberer Code für uns ist und welche Merkmale er auch aus Sicht bekannter Programmierer hat. Aus der Sicht von Bjarne Stroustrup – dem Erfinder von C++ – macht sauberen Code aus, dass er eine geradlinige Logik hat und seine Aufgabe gut erfüllt. Danach erfuhren wir etwas über die allgemeinen Regeln, wie man am besten seinen Code für andere Programmierer lesbar und nachvollziehbar machen kann, was in der Informatik sehr wichtig ist. Dies kann man zum Beispiel mit dem Schreiben von Kommentaren erreichen. Wenn man Code verbessert, gibt es verschiedene Dinge zu beachten, wie etwa die Verbesserung von Variablennamen, Zerlegung von zu großen Operationen und Eliminierung von doppeltem und nicht verwendeten Code.

Um dies alles auch an unseren selbstgeschriebene Programmen anwenden zu können, wurde uns das Programm „Codeblocks IDE“ nähergebracht, das uns dabei hilft, unseren Code besser zu strukturieren und unnötigen Code zu eliminieren. Im Anschluss lernten wir noch ein paar Regeln zum Benennen von Variablen, Schreiben von Kommentaren, Formatierung von Programmen und zur Fehlerbehandlung.

Nachdem wir alles einmal selbst ausprobiert hatten, ging der sehr informative Workshop um 14:30 Uhr zu Ende. Da wir noch Zeit hatten, bis unser Zug kam, verbrachten wir noch alle zusammen etwas Zeit in der Innenstadt, wo wir Eis aßen und noch eine Kleinigkeit vom Bäcker kauften. Danach machten wir uns pünktlich um 15:20 Uhr auf den Rückweg und kamen dann rechtzeitig zum Abendessen gegen 18:30 Uhr wieder am Eckenberg-Gymnasium an.

Unser Dank für den tollen Tag gilt dem ISE, besonders Herrn Seiler und Frau Prof. Paech, die uns den Besuch eines informativen und interessanten Vortrags ermöglicht haben.

Anmerkung der Kursleiter: Vielen Dank an das ISE für die tolle Gelegenheit!
P.S.: Die redaktionelle Verantwortung für die Schleichwerbung (Fastfood-Unternehmen) verbleibt bei unseren Kursteilnehmern.

. . . zum Schluss

- Unser Schlachtruf:
  `cout << "Wir sind laut!";`
  `cin >> "Der Sieg ist drin!";`
- „Ich sag' gar nix mehr!“ (Henriette)
- „Wenn man eine Katze in die Luft wirft, landet sie immer auf dem Boden.“ (Michael)
- „Ich finde das so kompliziert mit den Namen von den ganzen Begriffen.“ (Lennart)
- „Diikstra, Deikstra, Däikstra, Daikstra“
- „Hopp, Hopp!“
- *Bernhard hinkt mit seinen Krücken hinterher*: „Düs, Düs, Düs“
- „Amöbe, Amöbe!“ (Mario)
- „Das heißt nicht Labyrinth, sondern Irrgarten!“ (Lennart)
- *Stefi dreht Stuhl im Kreis*: „Ich bin DJ!“

Danksagung

Wir möchten uns an dieser Stelle bei denjenigen herzlich bedanken, die die 14. JuniorAkademie Adelsheim / Science-Academy Baden-Württemberg überhaupt möglich gemacht haben.

Finanziell wurde die Akademie in erster Linie durch die Stiftung Bildung und Jugend, die H. W. & J. Hector Stiftung, den Förderverein der Science-Academy sowie durch den Fonds der Chemischen Industrie unterstützt. Dafür möchten wir an dieser Stelle allen Unterstützern ganz herzlich danken.

Die Science-Academy Baden-Württemberg ist ein Projekt des Regierungspräsidiums Karlsruhe, das im Auftrag des Ministeriums für Kultus, Jugend und Sport, Baden-Württemberg und mit Unterstützung der Bildung & Begabung gGmbH Bonn für Jugendliche aus dem ganzen Bundesland realisiert wird. Wir danken daher dem Leiter der Abteilung 7 des Regierungspräsidiums Karlsruhe, Herrn Vittorio Lazaridis, der Referatsleiterin Frau Leitende Regierungsschuldirektorin Dagmar Ruder-Aichelin, Herrn Jurke und Herrn Dr. Hölz vom Ministerium für Kultus, Jugend und Sport sowie dem Koordinator der Deutschen Schüler- und JuniorAkademien in Bonn, Herrn Volker Brandt, mit seinem Team.

Wie in jedem Jahr fanden die etwas über einhundert Gäste sowohl während des Eröffnungswochenendes und des Dokumentationswochenendes als auch während der zwei Wochen im Sommer eine liebevolle Rundumversorung am Eckenberg-Gymnasium mit dem Landesschulzentrum für Umwelterziehung (LSZU) in Adelsheim. Stellvertretend für alle Mitarbeiter möchten wir uns für die Mühen, den freundlichen Empfang und den offenen Umgang mit allen bei Herrn Oberstudiendirektor Meinolf Stendebach, dem Schulleiter des Eckenberg-Gymnasiums, besonders bedanken.

Zuletzt sind aber auch die Kurs- und KüA-Leiter gemeinsam mit den Schülermentoren und der Assistenz des Leitungsteams diejenigen, die mit ihrer hingebungsvollen Arbeit das Fundament der Akademie bilden.

Diejenigen aber, die die Akademie in jedem Jahr einzigartig werden lassen und die sie zum Leben erwecken, sind die Teilnehmerinnen und Teilnehmer. Deshalb möchten wir uns bei ihnen und ihren Eltern für ihr Vertrauen ganz herzlich bedanken.