Eidgenössische Technische Hochschule Zürich

Datenstrukturen und Algorithmen

Summer Semester 2026

Lecture Notes

Contents

Datenstrukturen und Algorithmen

Fynn Krebser–fkrebser@student.ethz.ch

Sommersemester 2026

Lec 1

1 Introduction

Ziel des Kurses ist es über Effizienz von Algorithmen zu sprechen. Weiter wird ein vertiefter Einblick in C++ sowie ein wenig über paralleles Programmieren gegeben.

Dazu sprechen wir zunächst über Komplexität und danach über verschiedene Standardalgorithmen. Danach werden wir uns mit Graphen beschäftigen. Wenn wir über Algorithmen sprechen müssen wir auch über Datenstrukturen sprechen, da Algorithmen auf Datenstrukturen arbeiten. Ausserdem werden wir über generische und funktionale Programmierung sprechen.

Wir beginnen mit der Komplexität. Dazu eine Wiederholung des Algorithmus

Definition 1.1:

Ein Algorithmus ist eine wohldefinierte Berechnungsvorschrift welche aus Input einen Output berechnet.

Wir schauen uns das Problem der Sortierung an, da diese recht einfach ist. Der Input ist eine Folge von n Zahlen und der Output ist eine Permutation $(a'_1, \dots, a'_n)$ der Inputfolge, so dass $a'_1 \leq a'_2 \leq \cdots \leq a'_n$ gilt.

Schwierig an diesem Problem ist, wenn wir sehr viele Zahlen haben, also n sehr gross ist. Auch spielt die ursprüngliche Reihenfolge der Zahlen eine Rolle, z.B. wenn die Zahlen bereits sortiert sind, dann ist es einfacher als wenn sie in umgekehrter Reihenfolge vorliegen. Hierbei kommt es auf den Algorithmus an, was einfach und was schwierig ist.

Jedes Beispiel eines Inputs nennen wir eine Instanz des Problems. Es gibt unendlich viele Instanzen zum Sortierproblem, da es unendlich viele Zahlen gibt.

Grundsätzlich gibt es gute und schlechte Instanzen für einen Algorithmus. Daher bewerten wir Algorithmen meistens im Worst-Case. Teilweise kann man aber auch den Durchschnitt oder den Best-Case betrachten.

Example 1.2:

Ein Beispielalgorithmus ist wie folgt:

- Vergleiche a_0 mit jedem weiteren Element a_i für $i = 1, \dots, n-1$. Wenn $a_0 > a_i$ dann vertausche a_0 und a_i.
- Vergleiche a_1 mit jedem weiteren Element a_i für $i = 2, \dots, n-1$. Wenn $a_1 > a_i$ dann vertausche a_1 und a_i.
- ...

In einer Instanz mit 4 Zahlen würde der Algrithmus 6 Schritte benötigen. Im Fall von $n = 6$ Zahlen wären es 15 Schritte.

Der Code für diesen Algorithmus könnte wie folgt aussehen:

```
1 void sort(std::vector<int>& a){
2   int n = a.size();
3   for (int i = 0; i<n-1; ++i){
4     for (int j = i+1; j<n; ++j){
5       if (a[j] < a[i]){
6         std::swap(a[i],a[j])
7 } } } }
```

Wir möchten nun wissen, wie oft welche Zeile ausgeführt wird. Zeile 2 wird genau einmal ausgeführt, Zeile 3 wird abhängig von n ausgeführt, sagen wir $f_1(n) = n - 1$ mal ab. Für die Zeile 4 ist nun für jedes mal wenn Zeile 3 ausgeführt wird, Zeile 4 abhängig von i und n eine gewisse Anzahl von Malen ausgeführt

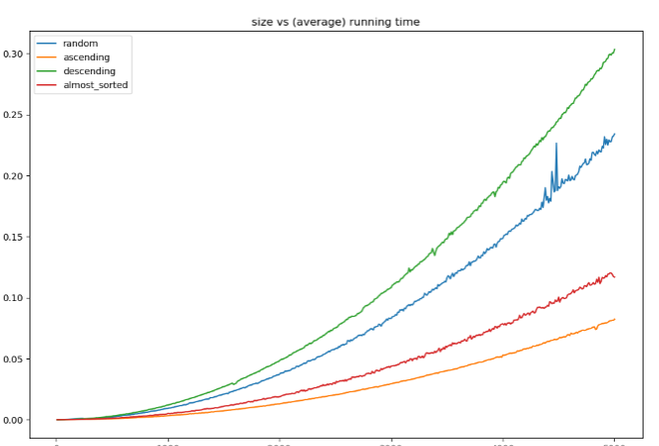
$$f_2(n, i) = \sum_{j=i+1}^{n-1} 1 = n - i - 1.$$

Die 5. Zeile wird dann

$$f_3(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n - i - 1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

Für sehr grosse n ist $f_3(n)$ ungefähr $\frac{n^2}{2}$, da die $-n$ im Zähler nicht mehr so relevant ist.

Der Swap ist nun interessant. Es könnte sein, dass die Zahlen bereits sortiert sind, dann wird Zeile 5 nie ausgeführt. Es könnte aber auch sein, dass die Zahlen in umgekehrter Reihenfolge vorliegen, dann wird Zeile 5 jedes Mal ausgeführt. Also haben wir im besten Fall $f_4(n) = 0$ und im schlechtesten Fall $f_4(n) = \frac{n(n-1)}{2}$.



Grundsätzlich werden Datenstrukturen verwendet um Daten zu organisieren, so dass diese effizient verarbeitet werden können.

2 Effizienz

Ziel ist es das Laufzeitverhalten von Algorithmen Maschinenunabhängig zu beschreiben. Weiter wollen wir die Effizienz von Algorithmen vergleichen abhängig von ihrer Eingabegrösse. Dazu benötigen wir ein Modell. Dazu abstrahieren wir die Technologie. Z.B. sprechen wir nicht mehr über ein Programm sondern über einen Algorithmus. Weiter werden wir eher in Pseudocode sprechen als in einer konkreten Programmiersprache.

Dazu gibt es zwei häufig verwendete Modelle:

Definition 2.1: Random Access Machine (RAM Modell)

Instruktionen werden der Reihe nach auf einem Prozessorkern ausgeführt.

Das Speichermodell wird durch konstante Zugriffszeit auf alle Adressen charakterisiert.

Elementare Operationen wie Addition, Multiplikation, Vergleich, etc. werden in konstanter Zeit ausgeführt.

Letzten Endes haben wir Fundamentaltypen wie grössenbeschränkte Ints oder Floats.

Definition 2.2: Pointer Machine Modell

Objekte beschränkter Grösse können dynamisch erzeugt werden in konstanter Zeit 1.

Auf Attribute der Objekte kann in konstanter Zeit 1 zugegriffen werden.

Die genaue Laufzeit eines Algorithmus hängt von der Implementierung, der Hardware, der Eingabe, etc. ab. Daher wollen wir die Laufzeit von Algorithmen asymptotisch beschreiben. Dabei ignorieren wir konstante Faktoren. Also ist lineares Wachstum mit Steigung 1 genauso gut wie lineares Wachstum mit Steigung 1000, da beide asymptotisch linear wachsen.

Wir schreiben $\Theta(n^2)$ und meinen dass der Algroithmus sich für grosse n wie n^2 verhält. Das bedeutet: Verdoppelt sich die Problemgrösse, so vervierfacht sich die Laufzeit.

Definition 2.3: Asymptotisch obere Schranke

Gegeben sei $g : \mathbb{N} \to \mathbb{R}$. Dann ist

$$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R} \mid \exists c > 0, n_0 \in \mathbb{N} \\ : 0 \leq f(n) \leq c \cdot g(n) \forall n \geq n_0\}.$$

Graphisch ist dies in Abbildung 1 dargestellt. Alle Funktionen f welche für grosse n unterhalb von $c \cdot g(n)$ liegen, gehören zu $\mathcal{O}(g)$. Die obige Definition kann man umdrehen für eine asymptotisch untere Schranke $\Omega(g)$.

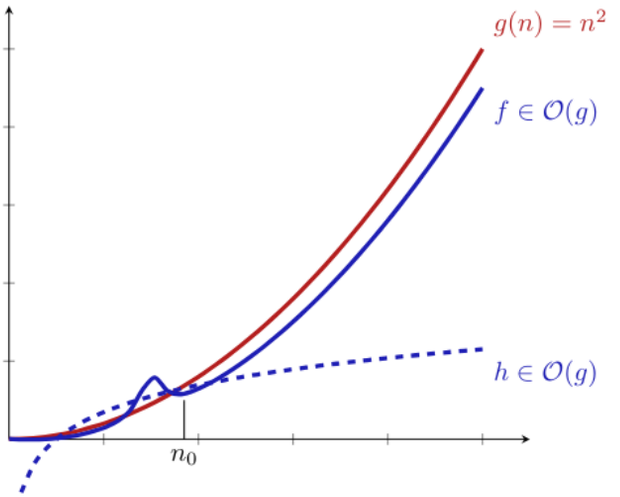


Figure 1: Graphische Darstellung von $\mathcal{O}(g)$

Definition 2.4: Asymptotisch untere Schranke

Gegeben sei $g : \mathbb{N} \to \mathbb{R}$. Dann ist

$$\Omega(g) = \{f : \mathbb{N} \to \mathbb{R} \mid \exists c > 0, n_0 \in \mathbb{N} : 0 \leq c \cdot g(n) \leq f(n) \forall n \geq n_0\}.$$

Weiter gibt es dann die asymptotisch scharfe Schranke $\Theta(g)$.

$$\Theta(g) = \mathcal{O}(g) \cap \Omega(g).$$

Man kann $f = \mathcal{O}(g)$ schreiben, jedoch sollte dieser Ausdruck nicht als Gleichung verstanden werden, sondern als Zugehörigkeit von f zu der Menge $\mathcal{O}(g)$. Daher wird vorzugsweise $f \in \mathcal{O}(g)$ geschrieben.

Lec 2

Definition 2.5: Komplexität

Die Komplexität sind die minimalen Kosten über alle Algorithmen welche das Problem lösen.

3 Algorithmen: Beispiele

3.1 Ägyptische Multiplikation

Eine Beispiel eines Algorithmus ist die Ägyptische Multiplikation. Sie funktioniert wie folgt:

- Gegeben seien zwei Zahlen a und b. Wir wollen $a \cdot b$ berechnen.
- Solange $a \geq 1$ gilt, wiederhole die folgenden Schritte:
  - Wenn a ungerade ist, addiere b zum Ergebnis hinzu.
  - Halbiere a (abrunden) und verdopple b.

Example 3.1:

Berechnen wir $11 \cdot 26$. Dann folgt

a	b	Ergebnis
11	26	0
5	52	26
2	104	78
1	208	78
0	416	286

Das Ergebnis ist 286.

Dieser Algorithmus ist effizient da der Computer sehr schnell mit 2 multiplizieren und zu dividieren ist. Wir wollen nun zeigen, dass die Ägyptische Multiplikation korrekt ist.

Proof. Wir schreiben zunächst unser Problem etwas mathematischer. Im allgemeinen ist

$$a \cdot b = \begin{cases} \frac{a}{2} \cdot 2b & \text{wenn } a \text{ gerade ist} \\ \frac{a-1}{2} \cdot 2b + b & \text{wenn } a \text{ ungerade ist} \\ b & \text{wenn } a = 1 \end{cases}.$$

Als Rekursionsgleichung wäre dies

$$f(a,b) = \begin{cases} f(\frac{a}{2}, 2b) & \text{wenn } a \text{ gerade ist} \\ f(\frac{a-1}{2}, 2b) + b & \text{wenn } a \text{ ungerade ist} \\ b & \text{wenn } a = 1 \end{cases}.$$

Wir wollen nun zeigen, dass $f(a,b) = a \cdot b$ für alle $a, b \in \mathbb{N}$ gilt. Wir verwenden vollständige Induktion über a. $H(1)$: $f(1,b) = b = 1 \cdot b$. $H(a)$: Wir nehmen an, dass $f(a',b) = a' \cdot b$ für alle $a' < a$ gilt. $H(a+1)$: Es gibt zwei Fälle:

- $a+1$ ist gerade: Dann ist $f(a+1,b) = f(\frac{a+1}{2}, 2b)$. Da $\frac{a+1}{2} < a+1$ gilt, können wir die Induktionsannahme anwenden und erhalten $f(\frac{a+1}{2}, 2b) = \frac{a+1}{2} \cdot 2b = (a+1) \cdot b$.
- $a+1$ ist ungerade: Dann ist $f(a+1,b) = f(\frac{a}{2}, 2b) + b$. Da $\frac{a}{2} < a+1$ gilt, können wir die Induktionsannahme anwenden und erhalten $f(\frac{a}{2}, 2b) = \frac{a}{2} \cdot 2b = a \cdot b$. Also ist $f(a+1,b) = a \cdot b + b = (a+1) \cdot b$.

□

Proof. Alternativ könne wir mit invarianten Argumentieren. Dazu wandeln wir unser Programm in ein iteratives um. Wir haben eine Variable res welche der dritten Zeile aus Beispiel 3.1 entspricht. Dann ist die Invariante $res + a \cdot b = a_0 \cdot b_0$, wobei a_0 und b_0 die ursprünglichen Werte von a und b sind.

Am Ende des Algorithmus ist $a = 0$, also $res = a_0 \cdot b_0$. Am Anfang des Algorithmus ist $res = 0$, also $res + a \cdot b = a_0 \cdot b_0$. □

3.2 Schnelle Multiplikation von Zahlen

In der Primarschule lernen wir die Multiplikation von Zahlen mit der sogenannten *long multiplication*. Diese sieht wie folgt aus:

62	37
434	
1860	
	2294

Im allgemeinen braucht dieser Algorithmus n^2 einzahlige Multiplikationen.

Wir bemerken dass wir schreiben können:

$$\begin{aligned} ab \cdot cd &= (10a + b)(10c + d) \\ &= 100ac + 10ac + 10bd + bd + 10(a - b)(d - c). \end{aligned}$$

	a	b		c	d	
	6	2	$\cdot$	3	7	
				1	4	$b \cdot d$
			1	4		$b \cdot d \cdot 10$
			1	6		$(a - b) \cdot (d - c) \cdot 10$
			1	8		$a \cdot c \cdot 10$
		1	8			$a \cdot c \cdot 100$
$=$		2	2	9	4	

Figure 2: Karatsuba Multiplikation

Wenn wir nun grössere Zahlen haben, so können wir diese in Blöcke von $n/2$ Ziffern aufteilen. Dann können wir die Multiplikation mit nur 3 Multiplikationen von Zahlen der Länge $n/2$ durchführen, anstatt 4 Multiplikationen zu benötigen.

$$6237 \cdot 5898 = \underbrace{62}_{\text{a}}\underbrace{37}_{\text{b}} \cdot \underbrace{58}_{\text{c}}\underbrace{98}_{\text{d}}.$$

Folglich benötigen wir für eine vierstellige Multiplikation nur drei zweistellige Multiplikationen also 9 einstellige Multiplikationen, anstatt 16 einstellige Multiplikationen zu benötigen.

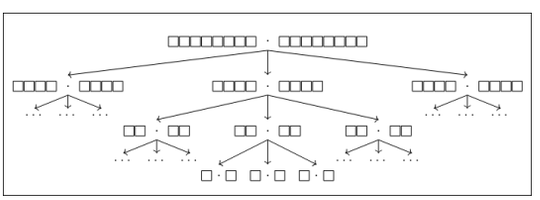


Figure 3: Karatsuba for 8 digit numbers

Nehmen wir nun an dass wir zwei n stellige Zahlen haben mit $n = 2^k$. Dann können wir schreiben

$$\begin{aligned} &(10^{n/2}a + b)(10^{n/2}c + d) \\ &= 10^n ac + 10^{n/2}ac + 10^{n/2}bd + bd + 10^{n/2}(a - b)(d - c). \end{aligned}$$

Dieser Algorithmus ist der Algorithmus von Karatsuba und Ofman.

Die Grundidee hier war Divide and Conquer

Theorem 3.2: Divide and Conquer

Zerlege das Problem in kleinere Teilprobleme, löse diese Teilprobleme und kombiniere die Lösungen der Teilprobleme zu einer Lösung des ursprünglichen Problems.

Wie viele einstellige Multiplikationen benötigen wir nun für diesen Algorithmus? Die Anzahl einstelliger Multiplikationen ist

$$M(2^k) = \begin{cases} 1 & \text{wenn } k = 0 \\ 3M(2^{k-1}) & \text{wenn } k > 0 \end{cases}.$$

Eine Möglichkeit diese Rekursionsgleichung zu lösen ist Teleskopieren. Dabei ersetzen wir $M(2^{k-1})$ durch $3M(2^{k-2})$ und so weiter, bis wir $M(1)$ erreichen. In diesem Fall ist

$$M(2^k) = \underbrace{3 \cdot 3 \cdot \cdots \cdot 3}_{k \text{ mal}} M(1) = 3^k.$$

Eine andere Möglichkeit wäre ein Beweis mit vollständiger Induktion über k zu führen.

Folglich ist $M(n) = M(2^{\log_2 n}) = n^{\log_2 3}$. Da $\log_2 3 < 1.6$ ist, ist dies deutlich besser als die n^2 einstellige Multiplikationen welche die long multiplication benötigt.

Dies ist nicht die schnellste bekannte Methode zur Multiplikation von Zahlen, es gibt sogar Algorithmen welche in $\mathcal{O}(n \log n)$ Zeit multiplizieren können. Dies geht aber über den Inhalt dieses Kurses hinaus.

3.3 Maximum Subarray Problem

Betrachten wir das folgende Problem:

Definition 3.3: Maximum Subarray Problem

Gegeben sei eine Folge von n Zahlen $a_1, \ldots, a_n \in \mathbb{R}$. Finde die zusammenhängende Teilfolge mit der grössten Summe.

Ein erster naiver Algorithmus wäre, alle möglichen zusammenhängenden Teilfolgen zu betrachten und die Summe zu berechnen. Es gibt n Möglichkeiten für den Anfang der Teilfolge und n Möglichkeiten für das Ende der Teilfolge, also gibt es n^2 mögliche Teilfolgen. Das Berechnen der Summe einer Teilfolge dauert im schlimmsten Fall n Schritte, also ist die Laufzeit dieses Algorithmus $\Theta(n^3)$.

Dieser Algorithmus kann verbessert werden durch Präfixsummen. Wenn wir wissen, was die Summe der ersten i Elemente ist, dann können wir die Summe einer Teilfolge von i bis j berechnen, indem wir die Summe der ersten j Elemente minus die Summe der ersten $i - 1$ Elemente nehmen.

$$S_{i,j} = \sum_{k=i}^{j} a_k = \underbrace{\sum_{k=1}^{j} a_k}_{S_j} - \underbrace{\sum_{k=1}^{i-1} a_k}_{S_{i-1}} = S_j - S_{i-1}.$$

Wir berechnen also zuerst die Präfixsummen S_i für $i = 1, \ldots, n$ in $\mathcal{O}(n)$ Zeit. Danach können wir die Summe jeder Teilfolge in konstanter Zeit berechnen, da wir nur zwei Präfixsummen subtrahieren müssen. Da es n^2 mögliche Teilfolgen gibt, ist die Laufzeit dieses Algorithmus $\Theta(n^2)$.

Wir können nun divide and conquer verwenden um den Algorithmus weiter zu verbessern. Wir teilen die Folge in zwei Hälften auf und berechnen das Maximum Subarray in der linken Hälfte, das Maximum Subarray in der

rechten Hälfte und das Maximum Subarray welches die Mitte überschreitet. Das Maximum Subarray, welches die Mitte überschreitet, kann in linearer Zeit berechnet werden, indem wir von der Mitte aus nach links und nach rechts gehen und das Maximum Links und Rechts der Suffix und Präfixsumme betrachten. Die Laufzeit dieses Algorithmus ist dann $\Theta(n \log n)$, da wir die Folge in jeder Rekursion halbieren und die Berechnung des Maximum Subarray, welches die Mitte überschreitet, linear ist.

$$T(n) = \begin{cases} d & \text{wenn } n = 1 \\ 2T(n/2) + c \cdot n & \text{wenn } n > 1 \end{cases}.$$

Die Beobachtung um den Algorithmus zu verbessern ist, dass wenn wir wissen wo der Endpunkt ist, dann können wir den Startpunkt in linearer Zeit berechnen. Daher können wir einfach von rechts nach links gehen und die Summe berechnen, bis wir die maximale Summe erreichen. Dazu setzen wir

$$R_i := \max_{1 \le r \le i+1} S_{r,i} = \max_{1 \le r \le i+1} \sum_{k=r}^{i} a_k.$$

Angenommen, wir haben R_{i-1} berechnet, dann können wir R_i berechnen durch

$$R_i = \max\{R_{i-1} + a_i, a_i\}.$$

4 Effizientes Speichermanagement

Lec 3

Eine Grosse Stärke von C++ ist die Möglichkeit, Variablen mit Werten zu darzustellen. Zum Beispiel kann mit `std::vector<int> w=v` ein ganzer Vektor kopiert werden.

Dies bedeutet auch, dass standardmässig alle Variablen BY VALUE an Funktionen übergeben werden. Somit verändert sich ein Vektor nicht, wenn er in einer Funktion verwendet wird. Andererseits, muss der Speicher kopiert werden, was Zeit kostet.

Es können Referenztypen verwendet werden, um Argumente als Alias zu übergeben, so dass kein Speicher kopiert werden muss. Nachteil davon ist, dass so keine Werte übergeben werden können, welche keine Adresse im Speicher haben, da es auch keinen Sinn macht z.B. `swap(3,a+a)` aufzurufen.

Mögliche Nachteile einer pass by reference sind, dass wir keine R-Werte übergeben können und dass wir nicht mehr sicher sein können, dass die übergebenen Werte nicht verändert werden, da sie als Alias übergeben werden.

Eine Möglichkeit ist eine Const Referenz zu verwenden. Dies löst beide unsere Probleme.

4.1 Wertekategorien

Audrücke in C++ haben einen Typ, repräsentieren einen Wert und repräsentieren möglicherweise ein Objekt mit Adresse.

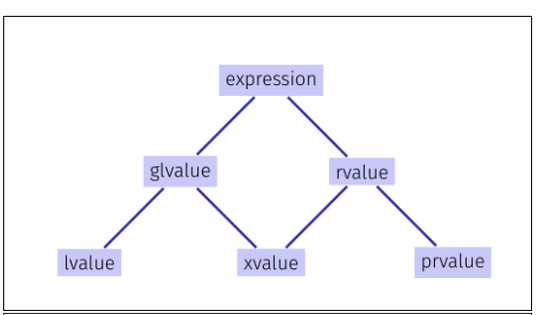
Wenn wir einen Zuweisung wie L=R ausführen möchten, müssen die Typen übereinstimmen aber auch die Werte Kategorien von L und R müssen stimmen.

In Informatik haben wir Werte welche Links von einer Zuweisung stehen L-Werte genannt und Werte welche Rechts von einer Zuweisung stehen R-Werte genannt.

Somit kann ein L-Wert verwendet werden, wo ein R-Wert erwartet wird, aber nicht umgekehrt.

Heutzutage hat sich C++ weiterentwickelt und es gibt weitere Wertekategorien.

z.B. die generalized l value, die pure r value und die expired value.



Eigenschaft		Hat Identität	Keine Identität
		glvalue	
Kann verschoben werden	rvalue	xvalue	prvalue
Kann nicht verschoben werden		lvalue	

Figure 4: Die Wahrheit

Wenn wir v=w schreiben können, dann muss v weiter leben und somit kann v nicht verschoben werden. Wenn

wir z.B. output(abs(v)) schreiben, dann könnte abs(v) verschoben werden, da es nicht weiter lebt. Eine prvalue sind Literale. Sie haben keinen Namen und können nicht verschoben werden, da sie keinen Speicher haben.

Table 1: Aufschlüsselung der Wertekategorien

Kartegorie	Beispiel
L-Wert	Variablen, Zeiger
xvalue	Resultat eines Funktionaufrufs
prvalue	Literale, Resultat von Operatoren

Definition 4.1: R-Wert-Referenz

Eine R-Wert-Referenz ist eine Referenz auf einen R-Wert. Sie wird mit `&&` deklariert.

Wenn das Quellobjekt einer Zuweisung direkt nach der Zuweisung nicht weiter existiert, dann kann der Compiler den Move-Zuweisungsoperator anstelle des Zuweisungsoperators einsetzen. Damit wird eine potentiell teure Kopie vermieden.

Dabei wird der Move-Konstruktor oder der Move-Zuweisungsoperator aufgerufen, wenn das Quellobjekt ein R-Wert ist. Wenn das Quellobjekt ein L-Wert ist, wird der Kopierkonstruktor oder der Kopierzuweisungsoperator aufgerufen.

Definiert werden die Move-Konstruktor und der Move-Zuweisungsoperator wie folgt:

```
class MyClass {
public:
    // Move-Konstruktor
    MyClass(MyClass&& other) {
        // Ressourcen von 'other' übernehmen
    }
    MyClass& operator=(MyClass&& other) {
        if (this != &other) {
            // Ressourcen von 'other' übernehmen
        }
        return *this;
    }
};
```

Ein Beispiel dafür wäre der Code

```
Vec operator + (const Vec& a, const Vec& b) {
  Vec tmp = a;
  // Add b to tmp
  return tmp;
}
int main(){
  Vec f;
  f = f + f + f + f;
}
```

Hierbei werden ganze 4 Kopien erstellt. Wenn wir den Move operator hinzufügen, werden nur noch 3 Kopien erstellt. Hierbei könnte man den Compiler unterstützen und den Vektor a by value übergeben.

```
Vec operator + (Vec a, const Vec& b) {
    // Add b to a
    return a;
}
```

Nun wird nur noch eine Kopie erstellt, bei der ersten Addition.

Move-Semantik kommt zum Einsatz, wenn ein x-wert (expired) zugewiesen wird. R-Wert-Rückgaben von Funktionen sind x-Werte.

5 C++ Advanced: Templates

Betrachte die folgende Funktion:

```
double sum(const std::vector<int>& v){
  double result = 0;
  for (auto x: v)
    result += x;
  return result;
}
```

Wir fragen uns nun, wie wir diese Funktion generischer machen können.

- Anderer Rückgabetyp
- Anderer Datentyp im Vektor
- Anderer Startwert
- Operation Anpassbar
- Datenstruktur anpassbar
- Eine Selektion der Daten

Für den letzten Punkt können wir ein Iterator-Paar übergeben.

Als laufendes Beispiel verwenden wir

```
class Pair{
  int left; int right;
public:
    Pair(int l, int r): left(l), right(r) {}

    int min() return left < right ? left : right;
};
```

Wir können nun vor der Definition der Klasse eine Template-Deklaration hinzufügen, um die Klasse generisch zu machen.

```
template<typename T>
class Pair{
    T left; T right;
public:
    Pair(T l, T r): left(l), right(r) {}
};
```

Dies nennt man Parametrischer Polymorphismus. Wenn die Klasse zu Ende ist hört dieses Typename automatisch auf. Es kann gelesen werden wie: Für alle Typen T.

Der Compiler generiert dann für jede Instanziierung der Klasse eine neue Klasse. Zum Beispiel wird die Klasse `Pair<int>` generiert, wenn wir `Pair<int> p(1,2)` schreiben.

Aktuell haben wir noch das Problem, dass wir nicht ein Pair von Pairs erstellen können. Der Compiler überprüft nur so wenig wie Nötig was potentiell zu Problemen führen kann.

Für Funktionen können wir ebenfalls Templates verwenden. Zum Beispiel könnte die Funktion

```
template<typename T>
T min(T l, T r){
    return l < r ? l : r;
}
```

Hierbei kann man auch die Funktion min<double> aufrufen, um die Funktion für double aufzurufen.

Bei unserer Implementation müssen nun aber beide Typen identisch sein. Hierfür können wir die Funktion wie folgt anpassen:

```
template<typename T1, typename T2>
auto min(T1 l, T2 r){
    return l < r ? l : r;
}
```

Diese Technik können wir auf unserem Pair Beispiel verwenden, um die Funktion min zu implementieren.

```
template<typename T>
class Pair{
    T left; T right;
public:
    Pair(T l, T r): left(l), right(r) {}
    auto min() return min(left, right);
};
```

Wenn der Typ bei der Instanzierung nicht inferiert werden kann, muss er explizit angegeben werden.

Lec 4

Eine sehr interessante Möglichkeit ist, dass so auch Funktionen als Template-Parameter übergeben werden können. Zum Beispiel könnte eine Funktion wie folgt übergeben werden

```
template <typename Container, typename Function>
void apply(Container& c, Function f){
    for (auto& x: c) f(x);
}
```

Wenn wir nun eine Funktion wie square haben, welche eine Zahl quadriert, dann können wir diese Funktion an apply übergeben. Dabei müssen wir aber den Typ der Funktion angeben, da er nicht inferiert werden kann.

Zusammenfassend können wir unser Eingangsbeispiel wie folgt anpassen:

```
template<typename InputIt, class T, class BinaryOp>
T accumulate(InputIt first, InputIt last, T init,
↪   BinaryOp op){
    for (; first != last; ++first) {
        init = op(init, *first);
    }
    return init;
}
```

Im allgemeinen ist es möglich das sich eine Funktionalität verbessern lässt als Spezialfall einer anderen Funktionalität. Man spricht von Spezialisierung Zum Beispiel könnten bools effizienter gespeichert werden mit

```
template <>
class Pair<bool>{
    short both;
public:
    Pair(bool l, bool r): both((l?1:0) | (r?2:0)) {}
    bool min() return (both & 1) != 0;
};
```

Templates können auch mit Werten parametrisiert werden, wobei der Wert aber zur Kompilierzeit bekannt sein muss. Zum Beispiel unterstützt die Eigen-Bibliothek die Definition von Matrizen mit einer festen Grösse, welche als Template-Parameter übergeben wird.

Um spezifische Anforderungen an die Template-Parameter zu stellen, können wir sogenannte Concepts verwenden. Zum Beispiel könnten wir verlangen, dass der Typ T einen Operator $<$ definiert hat, damit wir die Funktion min verwenden können.

6 Suchen

Das Grundproblem ist das wir eine Menge von Datensätzen haben und jeder Datensatz hat einen Schlüssel. Hierbei sind Schlüssel vergleichbar. Das Ziel ist es nun, einen Datensatz mit einem bestimmten Schlüssel zu finden.

In einem unsortierten Array mit n Elementen $A_1, \ldots, A_n$ mit Schlüssel b Dann können wir einfach alle Elemente durchgehen und vergleichen, ob der Schlüssel von A_i gleich b ist. Im schlimmsten Fall müssen wir alle n Elemente durchgehen, also ist die Laufzeit dieses Algorithmus $\Theta(n)$.

Bei gleichwahrscheinlicher Verteilung der Schlüssel ist der Erwartungswert der Anzahl der Vergleiche

$$\mathbb{E}(X) = \sum_{i=1}^{n} i \cdot \frac{1}{n} = \frac{n+1}{2}.$$

Nehmen wir nun an, dass die Elemente aufsteigend sortiert sind. Dann können wir das Array in der Mitte anschauen und wissen dann, ob wir in der linken oder rechten Hälfte weitersuchen müssen. Dies können wir dann rekursiv machen, bis wir das gesuchte Element gefunden haben oder bis die Suche fehlschlägt. Die Laufzeit dieses Algorithmus ist $\Theta(\log n)$.

Algorithm 6.1: **Binary Search**

Gegeben sei ein sortiertes Array A mit n Elementen und ein Schlüssel b. Wir wollen wissen, ob es ein Element in A gibt, dessen Schlüssel gleich b ist.

Der Algorithmus benötigt im schlechtesten Fall $\Theta(\log n)$ Elementarschritte.

Algorithm 1: Binary Search Algorithmus

Input: Aufsteigend sortiertes Array A mit n Schlüsseln, Schlüssel b
Output: Index $i \in [1, n]$ mit $A[i] = b$, sonst 0
Funktion `BSearch`(A, n, b)**:**

```
if n = 0 then
    return 0 ;                    // erfolglose Suche
end
m ← ⌊(1 + n)/2⌋;
if b = A[m] then
    return m ;                    // gefunden
else
    if b < A[m] then
        return BSearch(A[1 ... m − 1], m − 1, b) ;
         // Element liegt links
    else
        k ← BSearch(A[m + 1 ... n], n − m, b);
         // Element liegt rechts
        if k = 0 then
            return 0
        end
        return m + k
    end
end
```

Es stellt sich die frage ob es einen vergleichsbasierten Suchalgorithmus gibt, welcher schneller als $\Theta(\log n)$ ist.

Wir können für einen Vergleichsbasierten Algorithmus einen Entscheidungsbaum betrachten, welcher die Vergleiche darstellt, welche der Algorithmus durchführt. Der Baum muss jedes Objekt in einem Blatt repräsentieren, folglich muss der Baum mindestens n Knoten haben.

Doch ein binärer Baum mit Höhe h hat höchstens 2^h Blätter, also muss $2^h \geq n$ gelten. Folglich muss $h \geq \log_2 n$ gelten, also ist die Laufzeit jedes vergleichsbasierten Suchalgorithmus mindestens $\Theta(\log n)$.

Theorem 6.2:

Jeder vergleichsbasierte Suchalgorithmus benötigt im schlechtesten Fall $\Omega(\log n)$ Vergleiche für *sortierte* Elemente.

In einem unsortierten Array mit n Elementen ist die Anzahl der Vergleiche im schlechtesten Fall $\Omega(n)$, da wir alle Elemente durchgehen müssen, um sicher zu sein, dass das gesuchte Element nicht vorhanden ist.

Theorem 6.3:

Jeder vergleichsbasierte Suchalgorithmus benötigt im schlechtesten Fall $\Omega(n)$ Vergleiche, wenn die Elemente *unsortiert* sind.

7 Auswählen

Gegeben sei ein unsortiertes Array mit paarweise verschiedenen Elementen. Wir wollen nun das k-t kleinste Element in diesem Array finden. Also das Element, sodass genau $k-1$ Elemente kleiner sind als dieses Element.

Ein erster naiver Algorithmus wäre wiederholt das Minimum zu entfernen. Dann hätten wir den Median in $\Theta(n^2)$ gefunden.

Einfacherweise können wir das Minimum und Maximum in $2n$ Vergleichen finden. Jedoch geht es auch in $\frac{3n}{2}$ Vergleichen, indem wir die Elemente paarweise vergleichen und das kleinere Element mit dem Minimum vergleichen und das grössere Element mit dem Maximum vergleichen.

Somit haben wir noch 3 Vergleiche für nur 2 Elemente, also $\frac{3n}{2}$ Vergleiche für n Elemente.

Dies weist darauf hin, dass wir das k-t kleinste Element effizienter finden können.

Die Idee ist zu PIVOTIEREN. Dazu wählen wir ein Element p aus dem Array aus und partitionieren das Array in drei Teile: die Elemente welche kleiner als p sind, die Elemente welche gleich p sind und die Elemente welche grösser als p sind. Je nachdem, wie viele Elemente kleiner als p sind, können wir entscheiden, ob wir das k-t kleinste Element in der linken Partition, der mittleren Partition oder der rechten Partition suchen müssen. Dann wenden wir Rekursion auf den entsprechenden Teil an. Lec 5

Algorithm 2: Partitionierung um Pivot p (Hoare-Schema)

Input: Array A, das den Pivot p mindestens einmal enthält
Output: Index $k \in [1, n]$, so dass $A_i \leq p$ für $i \in [1, k]$ und $A_i \geq p$ für $i \in (k, n]$; Rückgabe von k

```
l ← 1; r ← n;
while true do
    while A[l] < p do
        l ← l + 1;
    end
    while A[r] > p do
        r ← r − 1;
    end
    if l ≥ r then
        return r;
    end
    tausche A[l] und A[r];
    l ← l + 1;
    r ← r − 1;
end
```

Die Korrektheit des Algorithmus lässt sich durch 3 Invarianten zeigen:

1. Alle Elemente in $A[1 \dots l-1]$ sind kleiner gleich p.
2. Alle Elemente in $A[r+1 \dots n]$ sind grösser gleich p.
3. Die Elemente in $A[l \dots r]$ sind noch nicht untersucht.

Zuletzt terminiert der Algorithmus, da im letzten Schritt l und r um 1 verschoben werden, wodurch die Anzahl der Elemente in $A[l \dots r]$ um 2 reduziert wird.

Um den Pivot an die richtige Stelle zu bringen, können wir den Algorithmus erweitern, indem wir die Stelle des Pivots speichern und am Ende des Algorithmus den Pivot mit dem Element an der Stelle l oder r tauschen.

Der eigentliche Auswahlalgorithmus sieht nun wie folgt aus:

Algorithm 3: Quickselect zur Bestimmung des k-kleinsten Elements

Input: Array A der Länge n, Index $1 \leq k \leq n$
Output: Wert $p \in A$ mit $|\{i : A[i] < p\}| < k$ und $|\{i : A[i] \leq p\}| \geq k$
Funktion `Select`(A, n, k)**:**

```
if n = 1 then
    return A[1];
end
Wähle ein Pivot p aus A;
m ← Partition(A, p);
if k < m then
    return Select(A[1 ... m − 1], m − 1, k);
else
    if k > m then
        return
         Select(A[m + 1 ... n], n − m, k − m);
    else
        return p;
    end
end
```

Die Partition braucht hierbei $\Theta(n)$ Zeit, da jedes Element genau einmal untersucht wird. Wenn wir optimistisch sind, so ist

$$T(n) = T(n/2) + cn = cn + c \cdot \frac{n}{2} + c \cdot \frac{n}{4} + \cdots \leq 2cn = \Theta(n).$$

Im schlechtesten Fall, wenn der Pivot immer das kleinste oder grösste Element ist, haben wir jedoch $\Theta(n^2)$, da wir dann immer nur ein Element aus dem Array entfernen. Wir wollen nun unser Pivot-Glück verbessern. Ein guter Pivot hat linear viele Elemente auf beiden Seiten. Wenn wir die Unterteilung mit einem Faktor q haben so ist

$$\begin{aligned} T(n) &\leq T(qn) + cn \leq T(q^2 n) + cn + cqn \\ &\leq T(q^k n) + cn \sum_{i=0}^{k-1} q^i \\ &\leq T(q^k n) + cn \cdot \frac{1}{1-q} \\ &\leq T(1) + cn \cdot \frac{1}{1-q} = \Theta(n). \end{aligned}$$

Der Zufall hilft uns hier. Im Erwartungswert erhalten wir einen guten Pivot nach 2 mal pivotieren.

Wenn wir nun immer in linearer Zeit einen Pivot finde möchten, können wir den Median der Mediane Algorithmus verwenden. Dabei teilen wir das Array in Gruppen von 5 Elementen auf, berechnen den Median jeder Gruppe und speichern diese Mediane in einem neuen Array. Dann berechnen wir erneut die Mediane dieses neuen Arrays, bis wir nur noch einen Median übrig haben. Dieser Median benutzen wir als Pivot für das Vorherige Array. Somit finden wir einen neuen Median. Dieser neue Median ist ein guter Pivot, da er garantiert, dass mindestens 30% der Elemente kleiner und mindestens 30% der Elemente grösser sind als dieser Pivot. Dieser Pivot wird dann rekursiv berechnet, was zu einer Laufzeit von $\Theta(n)$ führt.

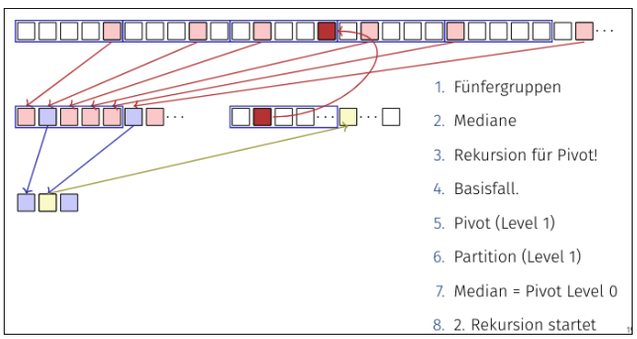


Figure 5: Median der Mediane Algorithmus

Implementiert wird der Median der Mediane Algorithmus wie folgt:

Algorithm 4: Median-of-Medians

Input: Array A der Länge n
Output: Median der Mediane
Funktion `MMSelect`(A, n)**:**

```
if n ≤ 5 then
    return MedianOfFive(A);
end
A' ← MedianOfFiveArray(A);
return MMSelect(A', ⌊(|A'|+1)/2⌋);
```

Hierbei ist `MMSelect` die Quickselect Funktion, welche als Pivot den Median der Mediane verwendet.

Theorem 7.1:
Das k-t kleinste Element in einem Array der Länge n kann in $\Theta(n)$ Zeit gefunden werden.

8 Sortieren

8.1 Iteratives Sortieren

Gegeben ist ein Array der Länge n. Wir wollen eine Permutation A' dieses Arrays finden, so dass A' aufsteigend sortiert ist.

Ein erster naiver Algorithmus wäre sogenanntes Selection Sort. Dabei gehen wir durch das Array und suchen das Minimum, tauschen es mit dem ersten Element und entfernen es dann aus dem Array. Dann suchen wir das Minimum des verbleibenden Arrays, tauschen es mit dem zweiten Element und so weiter.

Die Invariante dieses Algorithmus ist, dass die ersten i Elemente des Arrays die i kleinsten Elemente in aufsteigender Reihenfolge enthalten.

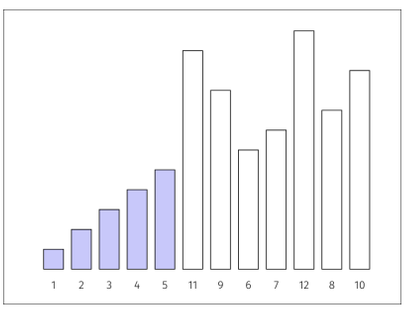


Figure 6: Selection Sort Invariante

Dieser Algorithmus ist in $\Theta(n^2)$, da wir n mal das Minimum suchen müssen, was jeweils $\Theta(n)$ Zeit benötigt. Es finden jedoch nur $n-1$ Vertauschungen statt, was $\Theta(n)$ ist.

Das Problem an diesem Algorithmus ist, dass wir alle verbleibenden Elemente durchgehen müssen, um das Minimum zu finden.

Ein anderer Algorithmus ist Insertion Sort. Dieser erinnert eher an das Sortieren von Jasskarten. Dabei gehen wir von links nach rechts durch das Array und nehmen das aktuelle Element und fügen es an der richtigen Stelle in die bereits sortierte Liste der vorherigen Elemente ein.

Die Invariante dieses Algorithmus ist, dass die ersten i Elemente des Arrays sortiert sind.

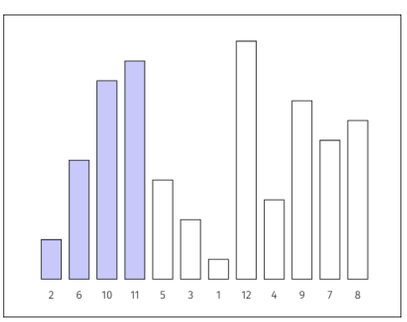


Figure 7: Insertion Sort Invariante

Dieser Algorithmus macht im schlechtesten Fall $\Theta(n^2)$ Vertauschungen/Verschiebungen, da jedes Element im schlimmsten Fall an den Anfang des Arrays verschoben werden muss. Die Anzahl Vergleiche ist ebenfalls $\Theta(n^2)$, da jedes Element mit allen vorherigen Elementen verglichen werden muss. Wir können diesen Algorithmus jedoch verbessern, indem wir die richtige Stelle für das aktuelle Element mit einer binären Suche finden. Dadurch reduzieren wir die Anzahl der Vergleiche auf $\Theta(n \log n)$, während die Anzahl der Vertauschungen/Verschiebungen weiterhin $\Theta(n^2)$ bleibt.

Ein weiterer Algorithmus ist Bubble Sort. Dabei machen wir ein wiederholtes Durchgehen durch das Array und vertauschen benachbarte Elemente, wenn sie in der falschen Reihenfolge sind. Dieser Algorithmus ist ebenfalls $\Theta(n^2)$ im schlechtesten Fall, da jedes Element mit jedem anderen Element verglichen werden muss.

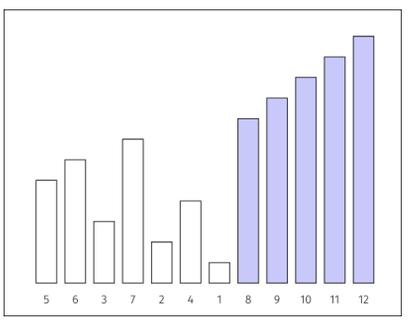


Figure 8: Bubble Sort Invariante

Table 2: Vergleich von Sortieralgorithmen hinsichtlich Vergleichen und Vertauschungen

Algorithmus	Vergleiche	Swaps
Selection Sort	$\Theta(n^2)$	$\Theta(n)$
Insertion Sort	$\Theta(n \log n)$	$\Theta(n^2)$
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$

8.2 Rekursives Sortieren

Lec 6

Wir wollen nun divide and conquer verwenden. Zum Beispiel ist ein Array der Länge 1 trivialerweise sortiert. Wir können also ein Array der Länge n in zwei Hälften teilen, diese Hälften rekursiv sortieren und dann die beiden sortierten Hälften zusammenführen.

Dies ist genau die Idee von Merge Sort. Dabei teilen wir das Array in der Mitte, sortieren die beiden Hälften rekursiv und führen sie dann mit einem Merge-Algorithmus zusammenführt. Der Merge-Algorithmus funktioniert, indem wir zwei Zeiger verwenden, welche die aktuelle Position in den beiden Hälften verfolgen. Wir vergleichen die Elemente an diesen Zeigern und fügen das kleinere Element in das Ergebnisarray ein und bewegen den entsprechenden Zeiger weiter. Sobald wir eines der beiden Arrays vollständig durchlaufen haben, fügen wir die verbleibenden Elemente des anderen Arrays in das Ergebnisarray ein.

Um dies zu zeigen, kann man die Invariante verwenden, dass die ersten i Elemente des Ergebnisarrays die i kleinsten Elemente der beiden Hälften enthalten.

Die Rekursionsgleichung für dieses Problem ist

$$T(n) = \begin{cases} 2 \cdot T(n/2) + cn & \text{wenn } n \geq 2 \\ d & \text{wenn } n = 1 \end{cases}.$$

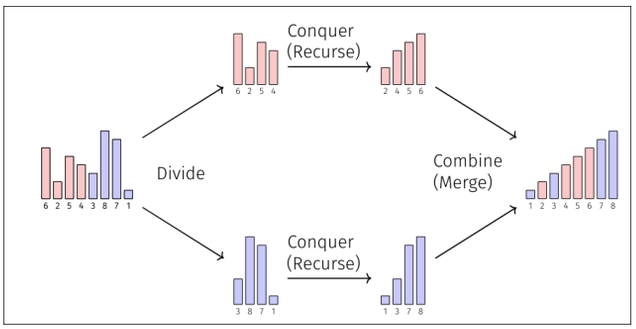


Figure 9: Merge Sort Algorithmus

Algorithm 5: Merge-Algorithmus zum Zusammenführen von zwei sortierten Arrays

Input: Zwei sortierte Arrays A und B der Länge n
Output: Ein sortiertes Array C der Länge $2n$, welches alle Elemente von A und B enthält

```
Z <- new Array(n+l);
x <- 1; y <- 1;
z <- 1;
while x ≤ n und y ≤ n do
    if A[x] < B[y] then
        Z[z] ← A[x]; x ← x + 1;
    end
    else
        Z[z] ← B[y]; y ← y + 1;
    end
    z ← z + 1;
end
while x ≤ n do
    Z[z] ← A[x]; x ← x + 1; z ← z + 1;
end
while y ≤ n do
    Z[z] ← B[y]; y ← y + 1; z ← z + 1;
end
```

Zusätzlich können wir den Merge Sort Algorithmus wie in Algorithmus 6 implementieren.

Wir könnten diese Rekursionsgleichung nun graphisch lösen, indem wir einen Rekursionsbaum zeichnen. Dabei sehen wir, dass die Höhe des Baumes $\log n$ ist und dass auf jeder Ebene cn Arbeit geleistet wird. Somit ist die Gesamtlaufzeit $\Theta(n \log n)$.

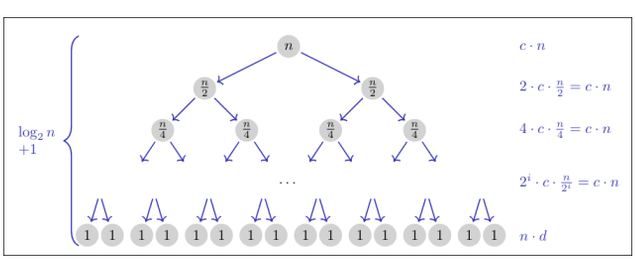


Figure 10: Rekursionsbaum für Merge Sort

Wir möchten solche Rekursionsgleichungen aber auch ein wenig systematischer lösen. Hierfür können wir den folgenden Satz verwenden.

Algorithm 6: Merge Sort Algorithmus

Input: Array A der Länge n
Output: A sortiert
Funktion `MergeSort(`A, n`):`

```
if n = 1 then
    return A;
end
m ← ⌊n/2⌋;
L ← MergeSort(A[1 ... m], m);
R ← MergeSort(A[m + 1 ... n], n − m);
return Merge(L, R);
```

Theorem 8.1: Master Theorem

Gegeben sei die Rekursionsgleichung

$$T(n) = a \cdot T(n/b) + f(n)$$

mit $a \geq 1$ und $b > 1$. Dann gilt:

1. Wenn $f(n) = \mathcal{O}(n^{\log_b a - \epsilon})$ für ein $\epsilon > 0$ gilt, dann ist $T(n) = \Theta(n^{\log_b a})$.
2. Wenn $f(n) = \Theta(n^{\log_b a})$, dann ist $T(n) = \Theta(n^{\log_b a} \log n)$.
3. Wenn $f(n) = \Omega(n^{\log_b a + \epsilon})$ für ein $\epsilon > 0$ gilt und wenn $a \cdot f(n/b) \leq c \cdot f(n)$ für ein $c < 1$ und alle hinreichend grossen n gilt, dann ist $T(n) = \Theta(f(n))$.

Was bei Merge Sort auffällt ist, das wir eine Top-Down Rekursion haben, das heist es wird zuerst komplett der linke Teil sortiert, bevor der rechte Teil sortiert wird. Es gibt jedoch auch die Möglichkeit, eine Bottom-Up Rekursion zu verwenden, bei welcher wir zuerst die kleinsten Teile sortieren und dann immer grössere Teile sortieren, bis wir das gesamte Array sortiert haben. Dies wird als Bottom-Up Merge Sort bezeichnet. Dies wäre ein it-

Algorithm 7: Bottom-Up Merge Sort Algorithmus

Input: Array A der Länge n
Output: A sortiert
Funktion `BottomUpMergeSort(`A, n`):`

```
length ← 1;
while length < n do
    for i ← 1 to n by 2 · length do
        L ← A[i ... i + length − 1];
        R ← A[i + length ... i + 2 · length − 1];
        Merge(L, R);
    end
    length ← 2 · length;
end
```

erativer $\Theta(n \log n)$ Algorithmus.

Eine weitere Effizienzsteigerung könnte durch natürliches Merge Sort erreicht werden. Dabei suchen wir zuerst nach bereits sortierten Teilarrays, welche wir dann mit dem Merge-Algorithmus zusammenführen. Dies kann für teilweise bereits sortierte Arrays schneller sein als der normale Merge Sort, da wir dann weniger Merges durchführen müssen.

In der Tat, führen wir im besten Fall, wenn das Array bereits sortiert ist nur $n-1$ viele Vergleiche durch. Jedoch ist die Laufzeit im Mittel immer noch $\Theta(n \log n)$.

Der Nachteil von Merge Sort ist, dass wir $\Theta(n)$ Speicher benötigen für das Verschmelzen.

Die Lösung kommt durch Quick sort. Dabei ist die Idee, ein beliebiges Element als Pivot zu Wählen. Dann partitionieren wir das Array um diesen Pivot, so dass alle Elemente welche kleiner als der Pivot sind auf der einen Seite und alle Elemente welche grösser als der Pivot sind auf der anderen Seite liegen.

Danach können wir die beiden Seiten rekursiv sortieren.

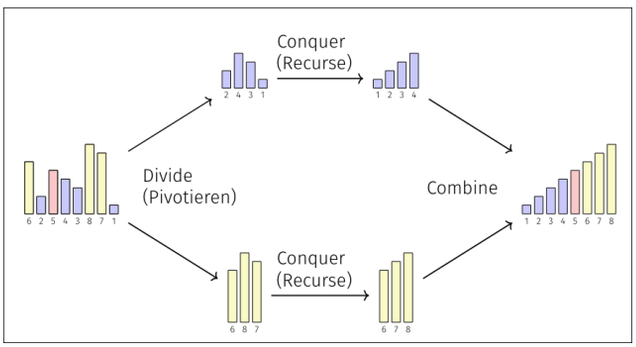


Figure 11: Quick Sort Algorithmus

Dieser Algorithmus kann nun in-place implementiert werden, da die Partitionierung in-place durchgeführt werden kann. Im schlechtesten Fall ist die Laufzeit von Quick Sort

Algorithm 8: Quick Sort Algorithmus

Input: Array A der Länge n
Output: A sortiert
Funktion `QuickSort(`A, n`):`
  if $n > 1$ **then**
    Wähle ein Pivot p aus A;
    $m \leftarrow$ Partition(A, p);
    `QuickSort(`$A[1 \dots m-1], m-1$`)`;
    `QuickSort(`$A[m+1 \dots n], n-m$`)`;
  end

$\Theta(n^2)$, wenn der Pivot immer das kleinste oder grösste Element ist.

Die Maximale Anzahl von Vertauschungen ist die Anzahl Schlüssel im kleineren Bereich plus Pivot (also $\lceil \frac{n}{2} \rceil$). Jedes Element, welches im kleineren Bereich landet, muss für den Tausch zahlen.

Doch ein Element kann höchstens $\log n$ mal im kleineren Bereich landen. Da wir n Elemente haben, haben wir insgesamt $\mathcal{O}(n \log n)$ Vertauschungen.

Theorem 8.2:
Quick Sort benötigt durchschnittlich $\mathcal{O}(n \log n)$ Vergleiche.

Die Rekursionstiefe von Quick Sort ist im schlechtesten Fall $n-1$. Im Schlimmsten Fall könnte dies zu einem Stack Overflow führen, wenn die Rekursionstiefe die maximale Stack-Grösse überschreitet.

Dies können wir verbessern, indem wir immer die kleinere Seite zuerst sortieren. Die grössere Seite wird dann iterativ sortiert, wodurch die maximale Rekursionstiefe auf $\mathcal{O}(\log n)$ reduziert wird.

In der Praxis wird als Pivot oft der Median von drei zufällig gewählten Elementen verwendet, um die Wahrscheinlichkeit eines schlechten Pivots zu reduzieren.

Es existiert auch eine Variante mit konstantem Speicherbedarf, welche für uns aber nicht relevant ist.

Komplizierte Divide and Conquer Algorithmen verwenden oft als Basisfall einen einfachen Sortieralgorithmus wie Insertion Sort, um kleine Teilarrays zu sortieren, da diese Algorithmen für kleine Arrays effizienter sein können als komplexere Algorithmen wie Merge Sort oder Quick Sort. Der Basisfall kann dann zum Beispiel für Arrays der Länge 10 oder kleiner verwendet werden.

Theorem 8.3:
Vergleichsbasierte Sortieralgorithmen benötigen im schlechtesten Fall $\Omega(n \log n)$ Vergleiche.

Die Intuition dahinter ist dass der Algorithmus unter $n!$ Permutationen die richtige ausgeben muss. Unser Baum muss also mindestens $n!$ Blätter haben, da jedes Blatt eine Permutation repräsentieren muss. Ein binärer Baum mit Höhe h hat höchstens 2^h Blätter, also muss $2^h \geq n!$ gelten. Folglich muss $h \geq \log_2 n!$ gelten. Da $\log_2 n! = \Theta(n \log n)$ ist, folgt die Behauptung.

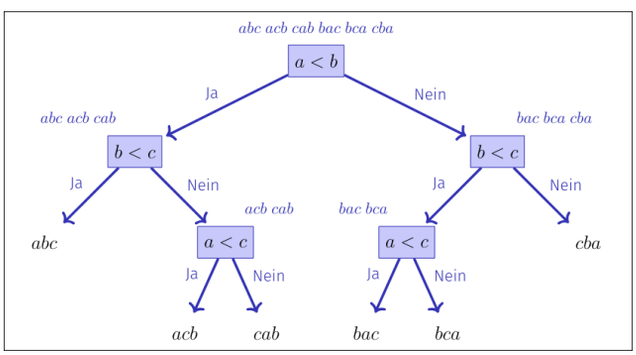


Figure 12: Entscheidungsbaum für Vergleichsbasierte Sortieralgorithmen

Lec 7

Bis jetzt haben wir nur über vergleichsbasierte Sortieralgorithmen gesprochen. Nehmen wir an, dass Schlüssel als Wörter der Länge l aus einem Alphabet der Grösse r dargestellt werden können. Wir nehmen weiter an, dass wir auf die Ziffern in konstanter Zeit zugreifen können.

Wenn wir mit $l = 2, r = 4$ arbeiten, dann könnten wir zwei Zahlen wie 0001 und 0011 haben. Die Grössere Zahl, ist die bei derjenige das erste Zeichen von links verschieden ist, die grössere Zahl. Das linkeste Bit entscheidet.

Dies ist die Idee von Radix Exchange Sort. Dabei sortieren wir die Schlüssel zuerst nach der ersten Ziffer, dann nach der zweiten Ziffer und so weiter. Die Rekursionstiefe ist hierbei l, da wir l Ziffern haben. Pro Level haben wir n Elemente, welche wir sortieren müssen. Somit ist die Gesamtlaufzeit von Radix Sort

$$\mathcal{O}(l \cdot n).$$

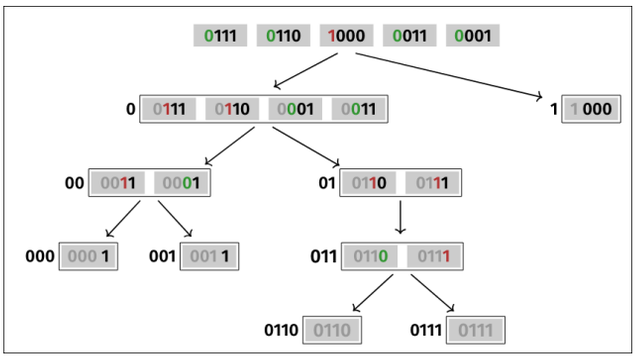


Figure 13: Radix Exchange Sort Algorithmus

Wenn wir anstelle dessen Zahlen von Rechts nach Links betrachten, können wir die Zahlen in sogenannte buckets sortieren, welche durch die Ziffern definiert werden. Anschliessend können wir die Zahlen in dieser Reihenfolge in buckets legen basierend auf der zweitletzten Ziffer und so weiter. Dies ist die Idee von Bucket Sort. Da die Reihenfolge nicht verändert wird, wenn die Ziffern gleich sind, ist die Sortierung stabil.

Implementiert werden kann ein Bucket durch einmal durchgehen der Liste, um die Anzahl Elemente pro Bucket zu zählen, um dann die Elemente in einem zweiten Durchlauf in die entsprechenden Ort in einem Vektor zu kopieren.

Die Laufzeit hier ist $\mathcal{O}(l \cdot (n + r))$.

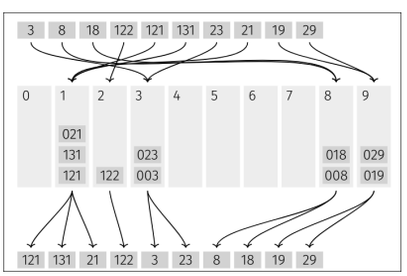


Figure 14: Bucket Sort Algorithmus

9 Datentypen & Datenstrukturen

Eine Warteschlange (queue) ist eine FIFO (First in, First out) Datenstruktur.

Es git Abstrakte Datentypen sowohl als Datenstrukturen.

Der Abstrakte Datentyp definiert das Interface, also welche Art von Problemen gelöst werden können. Sie stellt anforderungen an eine Menge mit gewissen Operationen.

Die Datenstruktur beschreibt die Implementation, wie die Probleme gelöst werden. Die Repräsentation der Daten ist zugeschnitten auf spezielle Operationen.

Die Implementation ist häufig davon abhängig, was man mit der Datenstruktur effizient machen möchte.

Als Beispiel betrachten wir eine Menge.

Table 3: Implementation von Mengen

Operation	**Array**	**Linked List**
create	$\Theta(n)$	$\Theta(n)$
iterate	$\Theta(n)$	$\Theta(n)$
find	$\Theta(n)$	$\Theta(n)$
insert	$\Theta(n)$	$\Theta(1)$
delete	$\Theta(n)$	$\Theta(n)$

Zur Erinnerung: Ein Stack ist ein abstrakter Datentyp mit folgenden Operationen:

- init() - Initialisiert den Stack
- empty() - Gibt zurück, ob der Stack leer ist
- push(x) - Fügt ein Element x auf den Stack hinzu
- pop() - Entfernt das oberste Element vom Stack und gibt es zurück

Implementiert werden kann ein Stack zum Beispiel mit einer Linked List. Somit wären push und pop in $\Theta(1)$, während die anderen Operationen ebenfalls in $\Theta(1)$ sind.

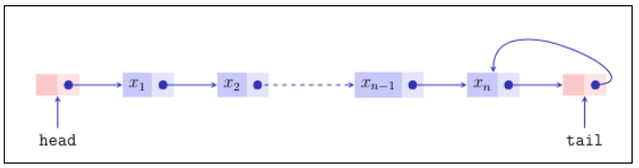


Figure 15: Stack Datenstruktur als Linked List

10 Amortisierte Analyse

Wir wollen verstehen, wieso man bei einem Pushback bei einem Vektor eine amortisierte Laufzeit von $\mathcal{O}(1)$ hat, obwohl die Laufzeit im schlechtesten Fall $\Theta(n)$ ist.

Ein viel einfacheres Beispiel ist der Multistack. Dieser Unterstützt zusätzlich zum Stack noch die Operation multipop. Diese Operation entfernt die oberste n k Elemente vom Stack, oder alle Elemente, wenn der Stack weniger als k Elemente enthält.

Wenn wir nun auf einem Stack mit n Elementen die Operation multipop(k), n-mal aufrufen, ist dann die Laufzeit $\Theta(n^2)$, da wir im schlimmsten Fall jedes Element n-mal entfernen müssen?

Intuitiv ist diese Antwort schlecht, da wir jedes Element nur einmal entfernen können. Wir wollen also eine schärfere Abschätzung für die Laufzeit erhalten. Doch wie können wir dies zeigen?

Diese Frage beantwortet die amortisierte Analyse. Sie schätzt die durchschnittliche Laufzeit jeder betrachteten Operation im schlechtesten Fall über eine Sequenz von Operationen. Achtung: Es geht nicht um den probabilistischen Durchschnitt, sondern um den Durchschnitt über die Sequenz von Operationen im worst case. Es wird also ausgenutzt, dass teure Operationen vielen günstigen Operationen gegenüberstehen, wodurch die durchschnittliche Laufzeit pro Operation günstig ist.

Es gibt 3 Varianten um die amortisierte Analyse durchzuführen.

10.1 Aggregate Analyse

Die Aggregierte Analyse folgt durch direkte Argumentation. Hierbei berechnet man eine Schranke für die Gesamtzahl der Elementarschritte, und teilt durch die Anzahl der Operationen, um die amortisierte Laufzeit pro Operation zu erhalten.

Bei unserem Multistack Beispiel können wir mit n Operationen maximal n Elemente zum Stack hinzufügen und somit maximal n Elemente entfernen. Somit sind die amortisierten Kosten pro Operation $\leq 2 \in \mathcal{O}(1)$.

10.2 Kontomethode

Der Computer basiert auf Münzen. Jede Elementaroperation kostet eine Münze. Für jede Operation wird eine bestimmte Anzahl von Münzen bezahlt. Die Münzen stammen aus einem Konto, welches nach jedem Schritt einen bestimmten Betrag enthält. Mit diesem Konto werden nun Operationen bezahlt. Es muss jedoch immer genügend Geld auf dem Konto sein, um die Operation zu bezahlen. Das Ziel ist es nun, die Kosten pro Operation so zu wählen, dass das Konto nie ins Minus rutscht. Die Einzahlung sind die amortisierten Kosten.

Am (nicht multi-)stack Beispiel kostet push 1 Franken und zusätzlich kommt ein CHF auf das Bankkonto ($a_k = 2$). Ein Aufruf von pop kostet 1 CHF, wird aber durch rückzahlung vom Bankkonto bezahlt ($a_k = 0$). Der Kontostand wird niemals negativ

$$a_k \leq 2 \Rightarrow \text{konstante Amortisationskosten.}$$

10.3 Potentialmethode

Wir definieren ein Potential Φ_i welches zum Zustand der betrachteten Datenstruktur zum Zeitpunkt i gehört. Dieses Potential muss so gewählt werden, dass es bei günstigen Operationen ansteigt und bei teuren Operationen abnimmt.

Wir definieren die amortisierten Kosten als

$$a_i \coloneqq t_i + \Phi_i - \Phi_{i-1}.$$

Am Beispiel des Stacks könnte das Potential die Anzahl Elemente im Stack sein. Somit ist die amortisierte Kosten von push $a_k = 1 + \Phi_i - \Phi_{i-1} = 2$ und die amortisierten Kosten von pop $a_k = 1 + \Phi_i - \Phi_{i-1} = 0$. Somit ist die amortisierte Laufzeit pro Operation konstant. Für diese Konklusion muss $\Phi_0 \leq \Phi_k$ sein.

Bei Multipop wären die realen Kosten k und $\Phi_i - \Phi_{i-1} = -k$, da wir k Elemente entfernen. Somit ist die amortisierte Kosten von multipop $a_k = k + \Phi_i - \Phi_{i-1} = 0$.

Betrachten wir nun einen Binären Zähler. Gegeben ist ein Zähler mit k Bits. Es wird von 0 bis $n-1$ gezählt wobei $n = 2^k$. Wir suchen die durchschnittlichen Zählkosten in Anzahl Bitflips pro Operation. Naiv haben wir pro Operation k Bitflips, da im schlimmsten Fall alle Bits von 1 auf 0 wechseln. Also ist die Laufzeit $\Theta(n \cdot k)$.

Die realen Kosten von Operation i sind $t_i = 1+$ Anzahl der trailing ones in i. Mit der Aggregatmethode bemerken wir, dass Bit 0 jedes mal wechselt, bit 1 wechselt jedes zweite mal, bit 2 wechselt jedes vierte mal und so weiter. Somit haben wir insgesamt

$$\sum_{i=0}^{n-1} \frac{n}{2^i} \leq n \cdot \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n.$$

Somit ist ein Bitwechsel im Durchschnitt $\mathcal{O}(1)$.

Bei der Aggregatsmethode argumentieren wir direkt auf Möglichen Operationsfolgen. Es existieren verhältnismässig wenig formale Beweismittel zur verfügung.

Mit der Kontomethode bemerken wir zunächst, dass die billigen Operationen diejenigen sind, bei welcher nur wenige trailing ones vorhanden sind. Teure Operationen sind diejenigen, bei welcher viele trailing ones vorhanden sind. Wir können nun die amortisierten Kosten von Operation i wie folgt wählen:

1 CHF reale Kosten für das Setzen von $0 \mapsto 1$ Plus 1 CHF für das Konto. Jedes Zurücksetzen von $1 \mapsto 0$ kostet 1 CHF, wird aber durch die Rückzahlung vom Bankkonto bezahlt.

Bei der Kontomethode haben wir auf einer Sequenz von Operationen argumentiert. Es wird nicht mehr global und explizit summiert.

Für die Potentialmethode können wir das Potential als die Anzahl der 1-Bits von x_i definieren. Wir haben

$$0 = \Phi_0 \leq \Phi_i \quad \Phi_i - \Phi_{i-1} = 1 - l_i.$$

Folglich ist

$$a_i = t_i + \Phi_i - \Phi_{i-1} = 1 + l_i + 1 - l_i = 2.$$

Bei der Potentialmethode argumentieren wir nur mit dem aktuellen Zustand der Datenstruktur. Es wird nicht mehr global und explizit summiert. Dies ist ein gutes Mittel für formale Laufzeitaussagen. In der Regel ist es schwierig, das Potential zu finden.

11 Binäre Suchbäume

Lec 8

Als Einführungsbeispiel betrachten wir ein Tischreservierungssystem in einem Restaurant. Es gibt einen Tisch und alle Reservierungen sollen 2 Stunden Abstand haben. Wie finden wir heraus, ob eine neue Reservierung möglich ist?

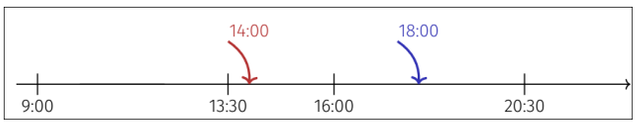


Figure 16: Tischreservierungssystem

Unsere Datenstruktur sollte folgende Operationen unterstützen:

- insert(t): $R < -R \cup \{t\}$
- delete(t): $R < -R \setminus \{t\}$
- check(t): ablehnen falls $\exists r \in R : |t - r| < 2$

Mögliche Datenstrukturen wären ein Array

Table 4: Datenstrukturen für das Reservierungssystem

	Array	**List**	**Sorted Array**	**Hash Table**
insert	$\Theta(1)$ a.	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$ e.
delete	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$ e.
check	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$

Go to slides during exam prep.

Das Problem am sorted array ist, dass wir viele Elemente verschieben müssen. Die Lösung für genau dieses Problem ist ein BALANCED BINARY SEARCH TREE. Damit werden wir all die Operationen in $\Theta(\log n)$ durchführen können.

Die Idee für einen Suchbaum kommt von der binären Suche. Eigentlich gehen wir dabei zunächst in die Mitte des Arrays, um das gesuchte Element zu finden. Danach gehen wir in die Mitte der linken oder rechten Hälfte, je nachdem ob das gesuchte Element kleiner oder grösser als das aktuelle Element ist. Dies ist wie wir effizient durch eine sortierte Liste navigieren können.

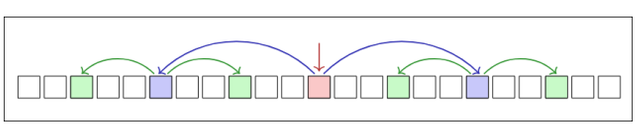


Figure 17: Idee des Binären Suchbaumes

Bäume sind verallgemeinerte Listen. Einzelne Knoten können mehrere Nachfolgern haben. Bäume sind spezielle Graphen, siehe später.

Kurz zur Terminologie: Der Beginn des Baumes heisst WURZEL, ein Konten heisst ELTERN und hat KINDER. Ein Nullpointer wird als BLATT bezeichnet. Die ORDNUNG des Baumes ist die maximale Anzahl Kindknoten. Die HÖHE des Baumes ist die maximale Pfadlänge von der Wurzel zum Blatt.

Um unser Leben zu vereinfachen lassen wir häufig die Pfeile Weg und nehmen an, dass der Baum von oben nach unten geht.

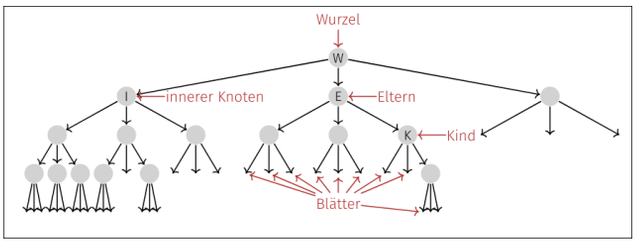


Figure 18: Terminologie von Bäumen

Ein BINÄRER BAUM ist entweder ein leerer Baum, dh ein Blatt, oder ein innerer Knoten mit zwei Bäumen als linken und rechten Nachfolger. In jedem inneren Knoten ist ein Schlüssel und einen Zeiger nach links und rechts gespeichert. Blätter werden durch nullpointer dargestellt.

Wenn wir nun einen Baum für unser Reservierungsproblem verwenden wollen, so müssen wir weitere Struktur hinzufügen. Dazu verwenden wir einen BINÄREN SUCHBAUM. Ein binärer Suchbaum ist ein binärer Baum mit der Suchbaumeigenschaft. Die Suchbaumeigenschaft besagt, dass für jeden inneren Knoten gilt, dass alle Schlüssel im linken Teilbaum kleiner als der Schlüssel des Knotens sind und alle Schlüssel im rechten Teilbaum grösser als der Schlüssel des Knotens sind.

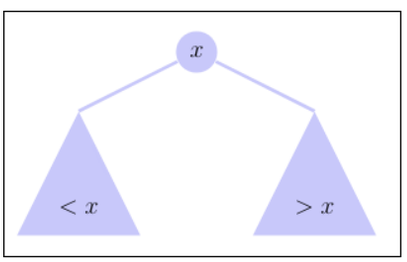


Figure 19: Suchbaumeigenschaft

In einem binären Suchbaum können wir effizient suchen.

Algorithm 9: Suchen in einem binären Suchbaum

Input: Binärer Suchbaum mit Wurzel r, Schlüssel k
Output: Knoten v mit Schlüssel k oder null

```
v ← r;
while v ≠ null do
    if k = v.key then
        return v;
    end
    else if k < v.key then
        v ← v.left;
    end
    else
        v ← v.right;
    end
end
return null;
```

Wir finden unser Element in $\Theta(h)$, wobei h die Höhe des Baumes ist. Leider kann die Höhe eines binären Suchbaums im schlimmsten Fall n sein, wenn der Baum degeneriert, also wie eine Liste aussieht. In diesem Fall wäre die Laufzeit von Suchen $\Theta(n)$. Den Trick um dies zu verhindern, ist es den Baum immer balanciert zu halten, also dafür zu sorgen, dass die Höhe des Baumes immer $\mathcal{O}(\log n)$ ist. Dies besprechen wir in der nächsten Vorlesung.

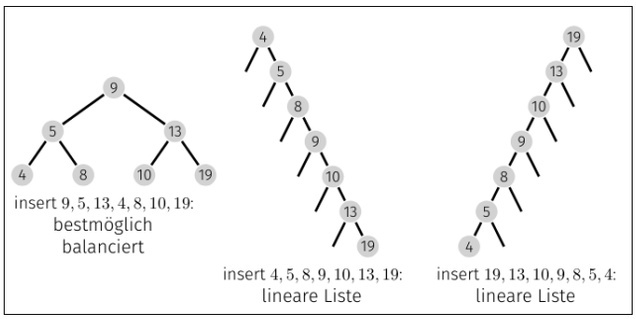


Figure 20: Degenerierter Baum

Für die Korrektheit können wir die Invariante verwenden, dass falls k im Baum ist, dann ist er immer im Teilbaum des aktuellen Knotens v.

Die Höhe $h(r)$ eines binären Baumes mit Wurzel r ist gegeben durch

$$h(r) = \begin{cases} 0 & \text{wenn } r = null \\ 1 + \max(h(r.left), h(r.right)) & \text{sonst} \end{cases}.$$

Wenn wir den Schlüssel k einfügen möchten, so müssen wir zunächst den Knoten v mit Schlüssel k suchen. Wenn v null ist, so können wir k an dieser Stelle einfügen. Ansonsten können wir die Einfügeoperation nicht durchführen, da der Schlüssel bereits im Baum vorhanden ist.

Folglich ist auch hier die Laufzeit $\Theta(h)$, da wir zunächst den Knoten v suchen müssen, um zu entscheiden, ob wir einfügen können oder nicht.

Wenn wir einen Knoten entfernen gibt es drei Fälle zu beachten:

1. Der Knoten v hat keine Kinder: In diesem Fall können wir einfach den Zeiger des Elternknotens auf null setzen. (Achtung: Dynamisches Speichermanagement)

2. Der Knoten v hat ein Kind: In diesem Fall können wir den Zeiger des Elternknotens auf das Kind von v setzen, wodurch v effektiv aus dem Baum entfernt wird.

3. Der Knoten v hat zwei Kinder: In diesem Fall müssen wir einen Ersatzknoten finden, um die Suchbaumeigenschaft zu erhalten. Wir können entweder den SYMMETRISCHEN NACHFOLGER oder den SYMMETRISCHEN VORGÄNGER von v verwenden. Wir finden den Nachfolger, indem wir in den rechten Teilbaum von v gehen und dann so lange nach links gehen, bis wir ein Blatt erreichen.

Nun müssen wir aber aufpassen, dass wir den Nachfolger nicht einfach an die Stelle von v setzen können, da der Nachfolger möglicherweise Kinder hat. Wir müssen also zunächst den Nachfolger entfernen, was entweder Fall 1 oder Fall 2 ist, und dann den Nachfolger an die Stelle von v setzen.

Löschen benötigt somit ebenfalls $\Theta(h)$ Zeit, da wir zunächst den Knoten v suchen müssen, um zu entscheiden, ob wir löschen können oder nicht, und danach möglicherweise den Nachfolger suchen und entfernen müssen, was aber konstante Zeit benötigt.

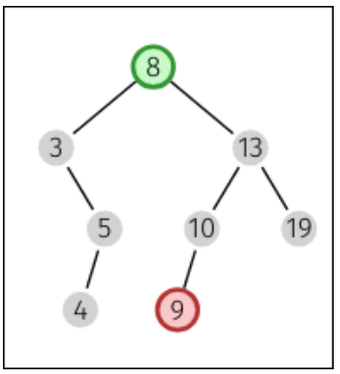


Figure 21: Löschen in einem binären Suchbaum

11.1 Traversierungsarten

Es gibt 3 verschiedene Möglichkeiten, einen Baum zu traversieren. Wenn wir die Knoten Wurzel, L, R haben so sind die Möglichkeiten:

- Preorder: Wurzel, L, R
- Inorder: L, Wurzel, R
- Postorder: L, R, Wurzel

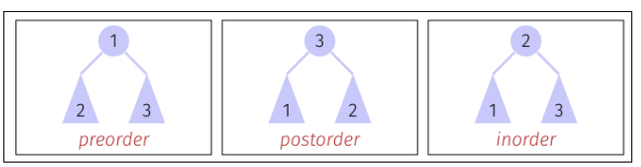


Figure 22: Traversierungsarten

Wenn wir inorder traversieren, so erhalten wir die Schlüssel in aufsteigender Reihenfolge.

Weitere nützliche Operationen sind die Suche nach dem Minimum, dem Maximum, das auflisten aller Elemente sowie das Zusammenführen von zwei Bäumen wenn

$$\max(T_1) < \min(T_2),$$

gilt.

12 Heaps

Wir verwenden im folgenden Max-Heaps[1]. Wir füllen dabei einen Baum von oben nach unten und von links nach rechts. Der Baum ist somit balanciert bei design. Die Heap Bedingung ist, dass alle Schlüssel in den Teilbäumen kleiner als der Schlüssel des Elternknotens sind.

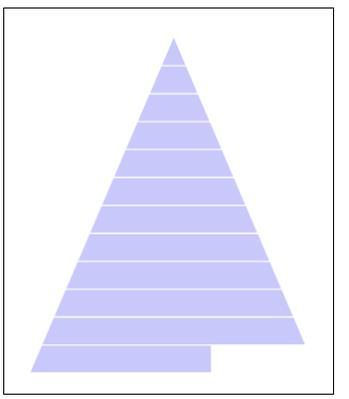


Figure 23: Max-Heap

Die Höhe eines Heaps mit n Elementen ist $\Theta(\log n)$, da der Baum balanciert ist. Da wir keine degenerierten Bäume haben können, können wir den Baum in einem Vektor, Ebene für Ebene, von links nach rechts, speichern. Somit können wir die Kinder eines Knotens i mit den Indizes $2i$ und $2i+1$ erreichen, umgekehrt finden wir den Elter eines Knotens i mit dem Index $\lfloor i/2 \rfloor$.

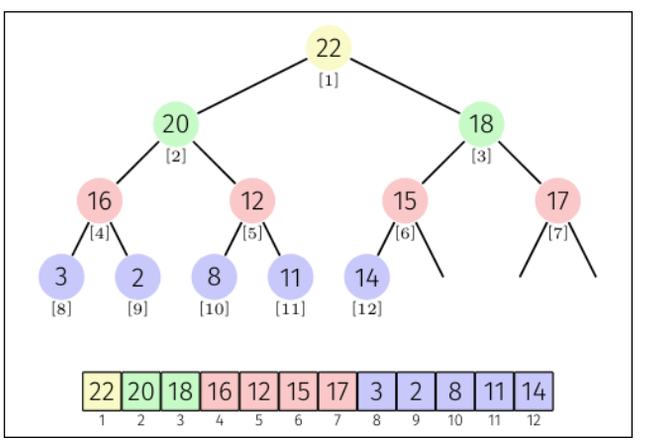


Figure 24: Heap als Array

Einfügen in einen Heap funktioniert, indem wir das neue Element an die letzte Stelle des Heaps einfügen und dann so lange mit seinem Elternknoten tauschen, bis die Heap Bedingung wieder erfüllt ist. Dies hat eine Laufzeit von $\mathcal{O}(\log n)$, da die Höhe des Heaps $\Theta(\log n)$ ist.

Die Invariante ist, dass nur die Kante vom Eingefügten Element zum Elternknoten verletzt sein kann.

In einem Heap soll auf keinen Fall gesucht werden. Dies ist extremst ineffizient. Das Maximum zu finden ist hingegen sehr effizient, da es immer an der Wurzel liegt.

Um das Maximum zu entfernen, können wir das letzte Element des Heaps an die Stelle der Wurzel setzen und dann

[1] Heap ist hier eine Datenstruktur

so lange mit seinen Kindern tauschen, bis die Heap Bedingung wieder erfüllt ist. Dies hat ebenfalls eine Laufzeit von $\Theta(\log n)$.

Die Invariante ist dass nur die zwei Kanten von unserem Element zu seinen Kindern die Heap Bedingung verletzen können.

Lec 9

Beginn mit Einschub zu Amortisierter Analyse

Wir vergleichen folgende Datenstrukturen

Table 5: Vergleich von Datenstrukturen für Bäume

Operation	**Suchbaum**	**Heap**	**Balanced**
in C++		make_heap	std::map
insert	$\Theta(h)$	$\Theta(\log n)$	$\Theta(\log n)$
search	$\Theta(h)$	$\Theta(n)$	$\Theta(\log n)$
delete	$\Theta(h)$	$\Theta(n)$	$\Theta(\log n)$
min/max	$\Theta(h)$	$\Theta(1)/\Theta(n)$	$\Theta(\log n)$

Beim erstellen eines Heaps aus einem Array können wir entweder jedes Element einzeln einfügen, was eine Laufzeit von $\mathcal{O}(n \log n)$ hat.

Wir beobachten: Auf der untersten Ebene ist jeder Knoten für sich schon ein korrekter Heap. Nehmen wir also unseren Vektor, so sind die letzten $\lceil n/2 \rceil$ Elemente bereits Heaps. Auf der zweituntersten Ebene haben wir $\lceil n/4 \rceil$ Knoten, welche jeweils 2 Kinder haben, welche bereits Heaps sind. Somit müssen wir nur noch diese $\lceil n/4 \rceil$ Knoten heapifizieren, was eine Laufzeit von $\mathcal{O}(n)$ hat. Naiv würden wir also sagen, dass wir Laufzeit $\mathcal{O}(n \log n)$ haben, da die Höhe des Heaps $\Theta(\log n)$ ist. Doch tatsächlich haben wir nur $\mathcal{O}(n)$, da die meisten Knoten in der unteren Hälfte des Heaps liegen und somit nur eine geringe Höhe haben.

Wir können Heaps auch verwenden um zu sortieren. Dazu erstellen wir zunächst einen Heap. Das Maximum steht an erster Stelle. Wir tauschen es mit dem letzten Element und entfernen es dann aus dem Heap. Danach müssen wir den Heap wiederherstellen, was eine Laufzeit von $\mathcal{O}(\log n)$ hat. Wir wiederholen diesen Prozess, bis der Heap leer ist. Am Ende haben wir die Elemente in aufsteigender Reihenfolge sortiert. Dies ist die Idee von Heap Sort.

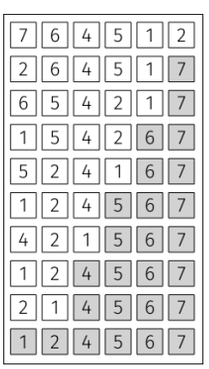


Figure 25: Heap Sort Algorithmus

13 Rot-Schwarz-Bäume

Wenn wir an Suchbäume zurückdenken, so haben wir gesehen, dass die Höhe eines Suchbaums im schlimmsten Fall n sein kann, wenn der Baum degeneriert, also wie eine Liste aussieht. Besser wäre es einen perfekt balancierten Baum zu haben, da die Höhe eines perfekt balancierten Baumes $\Theta(\log n)$ ist.

Der Trick ist, dass wir anstatt 2 Kinder pro Knoten zu haben, zwei Schlüssel $x < y$ zu haben. Links von x liegen alle Schlüssel, welche kleiner als x sind, zwischen x und y liegen alle Schlüssel, welche zwischen x und y liegen und rechts von y liegen alle Schlüssel, welche grösser als y sind.

Dies gibt uns einen sogenannten 2-3-Baum. Ausserdem ist der Baum perfekt balanciert. Die Höhe ist dann irendwo zwischen $\log_2 n$ und $\log_3 n$, da jeder innere Knoten entweder 2 oder 3 Kinder hat also in $\mathcal{O}(\log n)$ liegt.

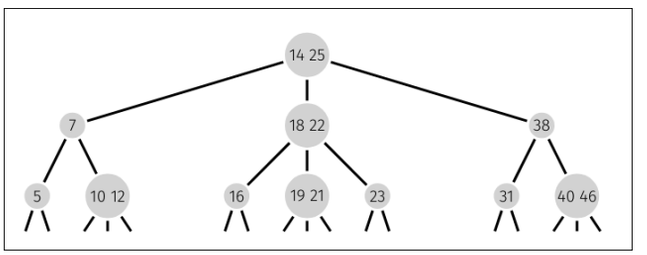


Figure 26: 2-3-Baum

In diesem Baum suchen wir identisch wie in einem binären Suchbaum, nur dass wir nun 2 Schlüssel pro Knoten haben können.

Wenn wir einen Schlüssel einfügen möchten, gibt es mehrere Fälle:

1. Der Knoten hat nur einen Schlüssel: In diesem Fall können wir den neuen Schlüssel einfach in den Knoten einfügen, so dass er sortiert ist. (2 Schlüssel wird zu einem 3 Schlüssel)

2. Wir möchten in einen 3-Knoten einfügen: Als Gedankenexperiment machen wir mal einen 4-Knoten, welcher 3 Schlüssel $x < y < z$ hat. Wir können diesen Knoten in einen 2-Knoten und einen 3-Knoten aufteilen, indem wir den mittleren Schlüssel y nach oben verschieben. Damit gibt es neu einen 3-Knoten mit y welches das vorher Linke und Rechte Kind von y als Kinder hat, und einen 2-Knoten mit x und z als Schlüssel, welches das vorherige Linke Kind von x und das vorherige Rechte Kind von z als Kinder hat.

Wenn wir nun anstatt eines 2-Knoten bereits einen 3-Knoten haben, dann machen wir unseren Prozess einfach rekursiv.

Wenn wir letzten Endes in der Wurzel einen 4-Knoten haben, so können wir diesen in einen 2-Knoten und einen 3-Knoten aufteilen, indem wir den mittleren Schlüssel nach oben verschieben. Dadurch wird die Höhe des Baumes um 1 erhöht, aber der Baum bleibt perfekt balanciert.

Jede dieser Operationen kann konstant gemacht werden. Somit haben wir insgesamt eine Laufzeit von $\mathcal{O}(\log n)$ für das Einfügen.

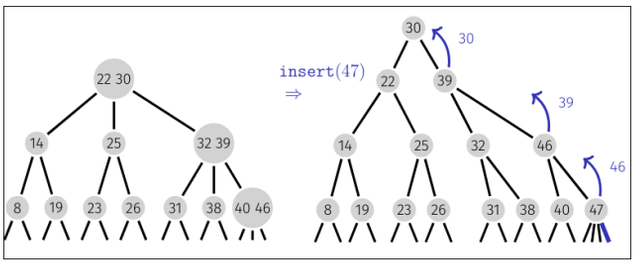


Figure 27: Einfügen in einen 2-3-Baum

Löschen funktioniert auch in $\Theta(\log n)$, dies ist jedoch kompliziert und nicht prüfungsrelevant.

13.1 Rot-Schwarz-Bäume

Ursprünglich waren wir an binären Bäumen interessiert. Prinzipiell können wir uns aus einem 3-Knoten einen binären Knoten machen, indem wir den rechten Schlüssel als neuen Knoten erstellen, welcher das vorherige Rechte Kind von y als Kind hat. Wir können die Verbindung zwischen x und y als roten Pfeil darstellen, um zu zeigen, dass diese beiden Knoten eigentlich zusammengehören. Die anderen Kanten sind schwarze Pfeile, um zu zeigen, dass diese Knoten nicht zusammengehören. Ein solcher Baum wird als Rot-Schwarz-Baum bezeichnet.

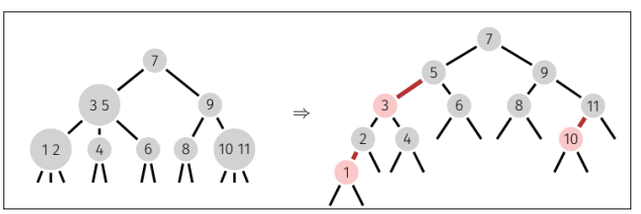


Figure 28: Rot-Schwarz-Baum

Die Operationen auf unserem 2-3-Baum können wir nun auf unseren Rot-Schwarz-Baum übertragen, indem wir die roten Pfeile entsprechend anpassen.

Ein Rot-Schwarz-Baum ist nicht perfekt balanciert, aber er ist so balanciert, dass die Höhe maximal $2 \log n$ ist. Somit haben wir auch hier eine Laufzeit von $\mathcal{O}(\log n)$ für die Operationen.

Ein Red-Black-Tree ist ein binärer Suchbaum mit folgenden Eigenschaften:

- rote Kanten gehen von Knoten zu seinem linken Kind (left leaning)
- kein Knoten hat zwei rote Kanten (keine 4-Knoten)
- Der Baum ist perfekt schwarz balanciert.

Die Suche funktioniert identisch wie in einem binären Suchbaum, da die Suchbaumeigenschaft nach wie vor gilt.

Wir definieren die Rechts-Rotation über x wie folgt: Ist y der Elter von x (ROT) und 3 und x der Elter von 1 und 2, so ist die Rechts-Rotation über x der Baum, welcher x zum Elter von y (ROT) und 1 macht, und y zum Elter von 2 und 3 macht.

Die Inverse Operation ist die Links-Rotation. Wir wechseln also die Rollen von x und y.

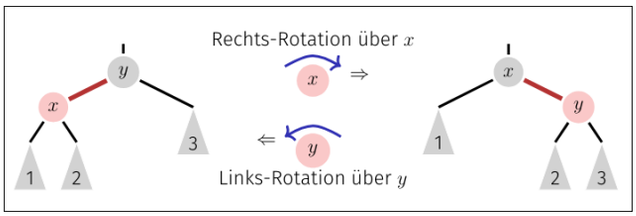


Figure 29: Rechts- und Links-Rotation

Weiter definieren wir den Color-Flip. Wenn wir zwei rote Kanten haben, so können wir diese in schwarze Kanten umwandeln und die schwarze Kante, welche aus dem Elter nach oben geht rot machen. Dies ist push up. Die Rückgängie Operation ist push down.

Wenn wir einen Schlüssel in einen 2-Knoten einfügen, so können wir diesen einfach in den Knoten einfügen, so dass er sortiert ist. (2 Schlüssel wird zu einem 3 Schlüssel)

Beim Rot Schwarz-Baum fügen wir einfach den Schlüssel als roten Knoten ein, da wir ja wissen, dass wir einen 3-Knoten haben werden. Wenn der Baum nun right-leaning wäre, so könnten wir eine Rechts-Rotation durchführen, um ihn left-leaning zu machen.

Spannender ist es in einen 3-Knoten einzufügen. Zunächst fügen wir den Schlüssel einfach mal ein. Nun haben wir jedoch möglicherweise zwei aufeinanderfolgende Rote Kanten. Dann machen wir eine Links-Rotation, um die roten Kanten zu trennen. Nun haben wir zwei rote Kanten, welche beide von y nach unten gehen. Wir können nun einen Color-Flip durchführen, um die roten Kanten in schwarze Kanten umzuwandeln und die schwarze Kante, welche aus dem Elter nach oben geht rot zu machen.

Wenn wir einen Schlüssel in die Wurzel einfügen, so können wir die beiden roten Kanten einfach in schwarze Kanten umwandeln, um die Wurzel schwarz zu machen.

Lec 10

14 Hashing

Beim Hashing geht es darum, wie bei Bäumen darum eine assoziative Datenstruktur zu erstellen. Also eine Verwaltung von Datensätzen mit eindeutigen Schlüsseln. Die Operationen, welche wir unterstützen wollen, sind:

- insert
- hasKey
- find
- delete

Wie bisweilen auch, betrachten wir jedoch hauptsächlich die Schlüssel.

Die Kernidee von Hashing ist es, eine Assoziative Tabellenstruktur zu erstellen. Hierbei wird der Schlüssel mit einer sogenannte HASHINGFUNKTION in einen Index umgewandelt, welcher die Position des Schlüssels in der Tabelle angibt.

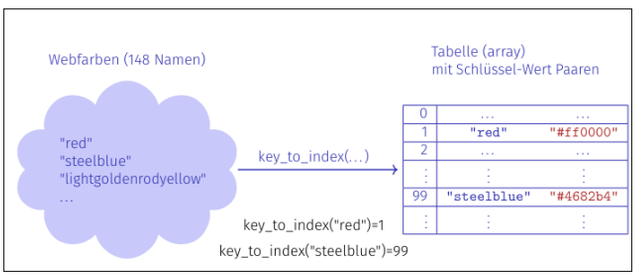


Figure 30: Hashing

Wir wollen uns nun damit befassen, wie man diese Hashingfunktion gestaltet.

Wenn wir ein unendlich grosses Array hätten, so könnten wir die Schlüssel beispielsweise direkt verwenden wenn wir sie in Ihrer Binärschreibweise betrachten. Dies ist bereits eine eineindeutige Zuordnung. Leider haben wir keinen unendlichen Speicher.

Dieses Problem wird PREHASHING durchgeführt. Die Abbildung $ph : \mathcal{K} \to \mathbb{N}$ bildet Schlüssel auf nichtnegative Zahlen ab. Die Abbildung erfolgt auf eine Ganzzahl beschränkter Grösse. Für eigene Datentypen muss dies selbst implementiert werden. Zum Beispiel geschieht dies mit

$$ph(s) = \left(\sum_{i=1}^{l_s} s_i b^i\right) \mod 2^w.$$

Dabei ist b so gewählt, dass ähnliche Strings möglichst verschiedene Indizes haben.

Vorteile davon sind, Objekt auf Index auf alle Elemente anwendbar sind, und dass wir in $\mathcal{O}(1)$ einfügen, nachschlagen und löschen können.

Nachteile sind, das wir extrem Platzineffizient sind und dass es zu Kollisionen kommen kann.

Um die Indexmenge zu verkleinern verwenden wir HASHING. Eine Hashfunktion bildet $h : \mathcal{K} \to \{0, 1, \ldots, m-1\}$ ab. Falls $h(k_1) = h(k_2)$ für $k_1 \neq k_2$ gilt, so sprechen wir von einer Kollision.

Um die Anzahl Kollisionen zu vermeiden, wollen wir die Schlüssel möglichst gleichmässig auf die Indizes verteilen.

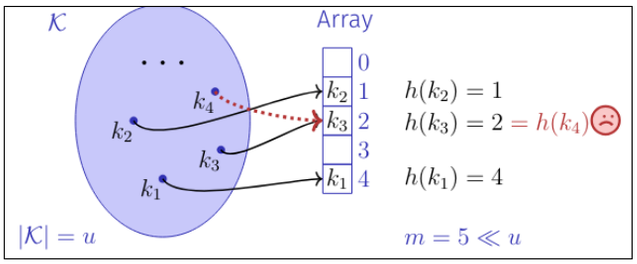


Figure 31: Hashing

Wenn es nun eine Kollision gibt, so können wir an jedem Index eine linked list abspeichern. Wenn wir nun suchen, so gehen wir in den Index und durchsuchen die Liste nach dem Schlüssel.

14.1 Übliche Hashfunktionen

Eine beliebte Methode zum Hashen ist die Divisionsmethode. Bei gegebenem m, wobei m die grösse der Tabelle ist, ist

$$h(k) = ph(k) \mod m.$$

Die Wahl von m ist hier entscheidend. Es sollte eine Primzahl sein, um die Anzahl der Kollisionen zu reduzieren. Dies, da dann $k, 2k, 3k, \ldots$ nicht alle auf den selben Index abgebildet werden für $k < m$. Für die Primzahlen kann man sich im vorhinein eine Primzahlliste erstellen, um die Auswahl zu erleichtern.

Es gibt noch die Multiplikationsmethode,

$$h(k) = \lfloor (a \cdot k \mod 2^w)/2^{w-r} \rfloor \mod m.$$

Als code ist dies sehr schnell geschrieben

```
a*k >> (w-r)
```

Hierbei ist $m = 2^r$ die Grösse der Tabelle und w die Grösse des Zahlensystems.

Idealerweise wäre a eine irrationale Zahl, skaliert und gerundet. z.B.

$$a = \left\lfloor 2^w \cdot \frac{\sqrt{5}-1}{2} \right\rfloor.$$

Was klar ist, es kann immer passieren, dass immer alle Schlüssel auf den selben Index abgebildet werden. Dies ist aber extrem unwahrscheinlich, wenn a gut gewählt ist.

Bei der Kollisionsbehandlung gibt es zwei Kernideen. Offenes hashing, bei welchem pro Tabellenindex ein Gefäss verwendet wird. In Geschlossenem Hashing wird in der Tabelle selbst nach einem freien Index gesucht, um den Schlüssel zu speichern.

Vorteile vom offenen hashing sind dass ein Belegungsfaktor grösser 1 möglich ist, und das Schlüssel sehr einfach gelöscht werden können. Nachteile sind, dass wir eine Laufzeit von $\mathcal{O}(n)$ im schlimmsten Fall haben können, wenn alle Schlüssel auf den selben Index abgebildet werden, und dass wir zusätzlichen Speicher für die Zeiger benötigen. Ausserdem gibt es häufige Speicher (de)allokationen.

Bei geschlossenem Hashing prüfen wir beim einfügen in unser Array zunächst, ob der Index frei ist. Wenn ja, so

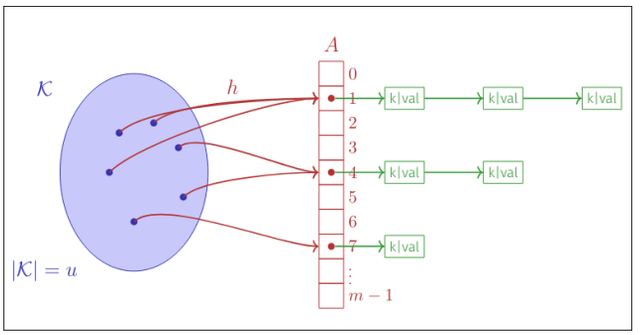


Figure 32: Offenes Hashing

können wir den Schlüssel dort speichern. Falls nein, so suchen wir weiter, bis wir einen freien Index finden. Beim Suchen gehen wir analog vor.

Typischerweise, lässt man die Tabelle immer zur Hälfte leer, um die Anzahl Kollisionen in Grenzen zu halten.

Leider ist dieses immer einen Index weiter gehen, gennant LINEARES SONDIEREN, sehr anfällig für primäre Häufung. Besser wäre quadratisches Sondieren, bei welchem die Anzahl Schritte quadratisch ansteigt, also $1^2, 2^2, 3^2, \ldots$. Eine andere Methode ist das doppelte Hashing, bei welchem wir eine zweite Hashfunktion verwenden, um die Anzahl Schritte zu bestimmen, also $h_2(k), 2 \cdot h_2(k), 3 \cdot h_2(k), \ldots$.

Tip 14.1:

An Prüfungen gern gefragt: Die Double Hashing Funktion ist von der Form

$$h_2(k) = 1 + (ph(k) \mod (n)).$$

Das "+1" ist wichtig, damit wir nicht immer auf den selben Index springen, wenn es eine Kollision gibt.

Ausserdem darf $h'(k)$ m nicht teilen, damit wir alle Indizes erreichen können.

Typischerweise nimmt man $n = m - 2$ wobei m prim ist.

Schwierig ist ausserdem das Löschen von Elementen, da wir nicht einfach den Index als frei markieren können, da dies die Suche nach anderen Elementen, welche auf diesen Index abgebildet werden, stören würde. Was man machen kann, ist den Wert zu nehmen und den Index als belegt zu markieren.

Wie gut ist nun Hashing? Dazu machen wir die stark unrealistische Annahme, dass jeder beliebige Schlüssel mit gleicher Wahrscheinlichkeit unabhängig von den anderen Schlüsseln auf einen Index abgebildet wird. Unter dieser Annahme ist die erwartete Länge einer Kette im offenen Hashing $\alpha = n/m$, wobei α als BELEGUNGSFAKTOR bezeichnet wird.

Lec 11

Wenn wir das noch analysieren, so benötigt die erfolglose Suche α Vergleiche, und das erfolgreiche Suchen benötigt im Schnitt ungefähr $1 + \frac{\alpha}{2}$ Vergleiche.

Zusammenfassend

Theorem 14.2: Offenes Hashing

Sei eine Hashtabelle mit Verkettung gefüllt mit Füllgrad $\alpha = \frac{n}{m}$. Unter einfachem gleichmässigem Hashing, hat die nächste Operation erwartete Laufzeit $\Theta(1 + \alpha)$.

Bei geschlossenem Hashing muss man zusätzlich annehmen, dass die Sondierungsfolge mit gleicher Wahrscheinlichkeit eine der $m!$ vielen Permutationssequenzen von m Indizes ist.

Theorem 14.3: Geschlossenes Hashing

Sei eine Hashtabelle mit geschlossenem Hashing gefüllt mit Füllgrad $\alpha = \frac{n}{m} < 1$. Unter der Annahme von gleichmässigem Hashing hat die nächste Operation erwartete Laufzeit $\leq \frac{1}{1-\alpha}$.

Was wir daraus lernen, ist dass die Grösse der Hash-Tabelle mit der Zeit mitwächst.

Dynamische Tabellen können durch Verdopplung der Tabellengrösse erstellt werden, wenn die Anzahl Elemente n die Grösse der Tabelle m erreicht. Das Einfügen eines Elements hat eine amortisierte Laufzeit von $\Theta(1)$, da das Verdoppeln der Tabelle und das Rehashing aller Elemente nur alle m Einfügungen notwendig ist.

Als Randnotiz: Auch Vektoren funktionieren nach genau diesem Prinzip. Immer wenn sie voll sind, verdoppeln sie ihre Grösse und kopieren alle Elemente in die neue Tabelle.

15 Quadtrees

Quadtrees sind grundsätzlich Bäume in einem zweidimensionalen Raum. Mögliche Fragen sind:

- Enthält die Menge einen Punkt p?
- Welches ist der nächste Punkt zu einem Punkt p?
- Welche Objekte überlappen sich?
- Welche Punkte liegen in einem Rechteck R?

Wenn wir das Problem im eindimensionalen Raum betrachten, so können wir einen rot-schwarz Baum verwenden, um die Punkte zu speichern.

Bei mehrdimensionalen Punkten haben wir keine natürliche Ordnung, jedoch können wir nach wie vor partitionieren. Rechtecke sind die natürliche Fortsetzung von Intervallen im eindimensionalen Raum. Dazu können wir zum Beispiel die Ebene in vier Quadranten aufteilen, welche jeweils ein Rechteck darstellen. Dies ist dann ganz ähnlich zu einem binären Suchbaum, nur dass wir nun vier Kinder pro Knoten haben, anstatt zwei. Dies nennt man einen Punkt-Quadtree.

Nachteil ist, dass ein solcher Baum vergleichsweise viele Knoten hat.

Eine zweite Variante ist, die Ebene in zwei Teile zu teilen, anstatt in vier Teile. Dabei wir jeweils in die andere Richtung geteilt, also erst vertikal, dann horizontal, dann wieder vertikal und so weiter. Dies wird ein K-D-Baum genannt. Dieser hat aber ähnliche Probleme wie der Punkt-Quadtree, da er ebenfalls viele Knoten haben kann und es schwierig ist, ihn balanciert zu halten.

Nehmen wir an, dass die Punkte in einem bekannten Bereich gleichmässig verteilt sind. Die Idee ist einen binären Baum zu erstellen, bei welchem die Blätter die Punkte speichern. Jeder innere Knoten speichert ein Rechteck, welches die Punkte in den Blättern unter ihm umschliesst. Man legt also ein Raster über die Ebene und speichert in jedem inneren Knoten das Rechteck, welches die Punkte in den Blättern unter ihm umschliesst. Dies ist ein Region-Quadtree.

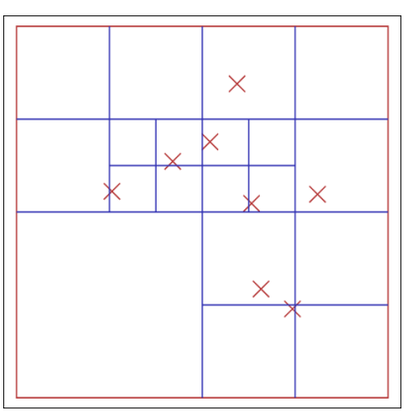


Figure 33: Region-Quadtree

Ein klassisches Problem ist die Kollisionsdetektion. Es gibt verschiedene Möglichkeiten, dies zu tun. Eine Möglichkeit ist, alle Paare von Punkten zu vergleichen, was eine Laufzeit von $\Theta(n^2)$ hat. Eine andere Möglichkeit ist, die Punkte in einem Quadtree zu speichern und dann für jedes Rechteck zu überprüfen, ob es mit anderen Rechtecken kollidiert. Dies hat eine Laufzeit von $\mathcal{O}(n \log n)$, da wir für jedes Rechteck nur die Rechtecke überprüfen müssen, welche sich in der Nähe befinden.

Einfügen können wir indem wir zunächst einen einzigen Knoten haben. Dann werden Objekte zu dem Knote hinzugefügt. Sobald wir zu viele Objekte haben, teilen wir den Knoten in vier Kinder auf.

Algorithm 10: Erstellen eines leeren Quadtree-Knotens

Input: Rechteck $[x, x+w] \times [y, y+h]$
Output: Leerer Quadtree-Knoten, welcher das Rechteck umschliesst

$q \leftarrow$ neuer Quadtree-Knoten;
q.area $\leftarrow$ Rectangle(x,y,w,h);
q.ul $\leftarrow$ q.ur $\leftarrow$ q.ll $\leftarrow$ q.lr $\leftarrow$ null;
q.split $\leftarrow$ false;
q.points $\leftarrow$ leere Liste;
return q;

Algorithm 11: Aufteilen eines Quadtree-Knotens

Input: QuadTree Knoten q ohne Kindsknoten
Output: q hat vier Kindsknoten

$q.split \leftarrow true$;
$q.ul \leftarrow$ createNode$(q.area.x, q.area.y + q.area.h/2, q.area.w/2, q.area.h/2)$;
... **foreach** *Punkt p in q.points* **do**
  insert into child of q which contains p;
end
$q.points \leftarrow$ leere Liste;

Wenn wir nun alle Punkte in einem Rechteck R suchen wollen, so können wir zunächst alle Knoten überprüfen, welche sich mit R überschneiden. Wenn ein Knoten vollständig in R liegt, so können wir alle Punkte in diesem Knoten zurückgeben. Wenn ein Knoten teilweise in R liegt, so müssen wir die Punkte in diesem Knoten überprüfen, um zu entscheiden, welche Punkte in R liegen.

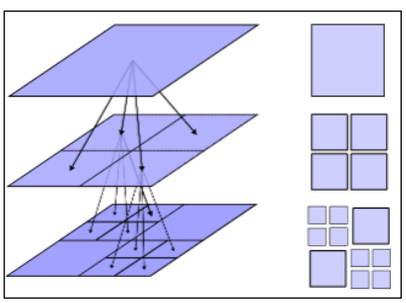


Figure 34: Dreidimensionale Darstellung eines Quadtrees

Lec 12

16 Geometrische Algorithmen I

Gegeben ist eine Menge von Punkten in der Ebene. Wir wollen herausfinden, welche Punkte die äusserste Hülle bilden, also welche Punkte die Ecken des kleinsten konvexen Polygons bilden, welches alle Punkte enthält. Dies wird als Convex Hull bezeichnet.

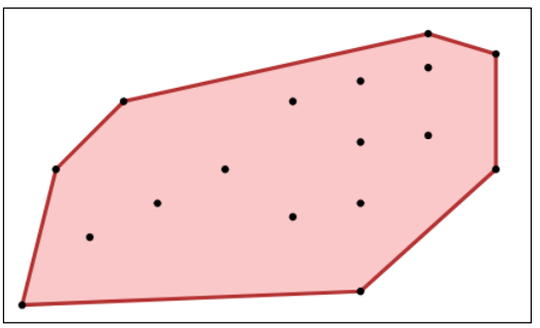


Figure 35: Konvexe Hülle

Unser Input sind n Punkte $P_1, P_2, \ldots, P_n$ in der Ebene. Unser Output sind die Punkte $H_1, H_2, \ldots, H_k$, welche die Ecken des kleinsten konvexen Polygons bilden, welches alle Punkte enthält. Hierbei kann k alle Werte zwischen 3 und n annehmen.

Wir bemerken dass ausserdem Extrempunkte immer auf unserer konvexen Hülle liegen. Also zum Beispiel der linkeste, der rechteste, der oberste und der unterste Punkt.

Wir bemerken ausserdem, dass eine Strecke Teil der konvexen Hülle ist, genau dann wenn alle Punkte auf der selben Seite der Geraden liegen, welche durch die Strecke definiert wird.

Mit einem trivialen Algorithmus könnten wir mit einem $\Theta(n^3)$ Algorithmus alle möglichen Geraden durch zwei Punkte überprüfen und dann für jede Gerade überprüfen, ob alle Punkte auf der selben Seite liegen.

Ein besserer Algorithmus ist der sogenannte Gift Wrapping Algorithmus. Angenommen wir haben bereits eine Strecke $pq \in H$ gefunden. Wie finden wir von p die nächste Ecke r? Es ist der Punkt, welcher den kleinsten Winkel mit der Strecke pq bildet. Der Start könnten wir machen, indem wir mit einer vertikalen Linie durch den linkesten Punkt beginnen, welcher auf der konvexen Hülle liegt.

Die Laufzeit dieses Algorithmus ist $\Theta(nk)$, da wir k mal n Winkel verlgeichen müssen, um die nächste Ecke zu finden.

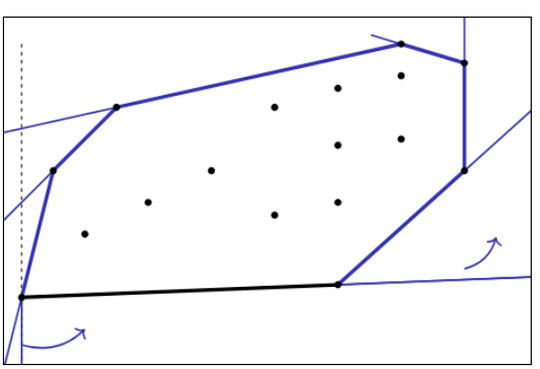


Figure 36: Der Jarvis March / Gift Wrapping Algorithmus

Leider sind Winkelvergleiche numerisch instabil, da sie auf der Berechnung von Arctan basieren, welche numerisch instabil sein kann. Besser ist es mit der Abbiegerichtung zu arbeiten. Wenn wir zwei Punkt r und s als Mögliche Kandidaten für die nächste Ecke haben, so können wir das Kreuzprodukt $(r-q) \times (s-q)$ berechnen. Wenn das Kreuzprodukt positiv ist, so haben wir eine Linkskurve, wenn es negativ ist, so haben wir eine Rechtskurve und wenn es null ist, so liegen die Punkte auf der selben Geraden. Wenn wir gegen den Uhrzeigersinn gehen, so wollen wir die schwächere Linkskurve, also das Kreuzprodukt soll negativ sein.

In der Ebene gilt für das Kreuzprodukt

$$q_1 \times q_2 = \det \begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \end{pmatrix} = x_1 y_2 - x_2 y_1.$$

Algorithm 12: Gift Wrapping Algorithmus zur Berechnung der konvexen Hülle

Input: Punkte $P_1, P_2, \ldots, P_n$ in der Ebene
Output: Punkte $H_1, H_2, \ldots, H_k$, welche die Ecken des kleinsten konvexen Polygons bilden, welches alle Punkte enthält

```
H ← leere Liste;
p ← linkester Punkt;
while p ist nicht der Startpunkt do
    H.append(p);
    q ← nächster Punkt in H;
    r ← erster Punkt;
    foreach Punkt s do
        if (r − q) × (s − q) < 0 then
            r ← s;
        end
    end
    p ← r;
end
return H;
```

Die Laufzeit dieses Algorithmus ist $\Theta(nk)$, und $\mathcal{O}(n^2)$ sowie $\Omega(n)$, da k alle Werte zwischen 3 und n annehmen kann.

Wir wollen dies verbessern. Immer wenn die Eckpunkte entgegen dem Uhrzeigersinn geordnet sind, biegen aufeinanderfolgende Strecken nur nach Links ab.

Die Idee von Graham Scan ist es, zunächst nur die untere konvexe Hülle zu berechnen. Dazu verbinden wir zunächst alle Punkte von Links nach Rechts. Wenn wir dann einen Punkt mit rechtsabbiegung haben, so überspringen wir diesen Punkt und verbinden den vorherigen Punkt mit dem nächsten Punkt. Die Hauptschwierigkeit hier ist, dass entfernen eines Punktes das Entfernen des vorherigen Punktes zur Folge haben kann, da es nun zu einer Rechtsabbiegung kommt. Wir müssen also so lange zurückgehen, bis wir wieder eine Linkskurve haben.

Dies ist aber dennoch in $\Theta(n)$ da jeder Punkt höchstens einmal gelöscht wird.

Für die obere konvexe Hülle machen wir das gleiche, nur dass wir von Rechts nach Links gehen. Am Ende können

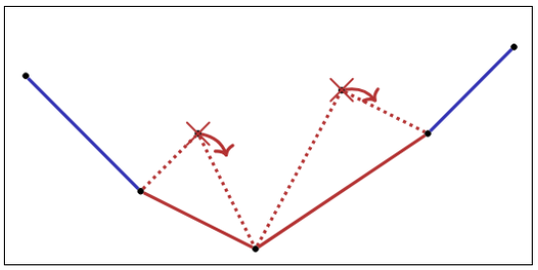


Figure 37: Idee des Graham Scan Algorithmus

wir die beiden konvexen Hüllen zusammenfügen, um die gesamte konvexe Hülle zu erhalten.

Als Datenstruktur um die Hülle zu speichern, müssen wir den hintersten Punkt schnell entfernen und hinzufügen können. Ein Stack ist ideal hierfür geeignet. Dies könnte zum Beispiel mit einer doubly linked list implementiert werden.

Algorithm 13: Graham Scan Algorithmus zur Berechnung der konvexen Hülle

Input: Punkte $P_1, P_2, \ldots, P_n$ in der Ebene
Output: Punkte $S_1, S_2, \ldots, S_k$, welche die Ecken des kleinsten konvexen Polygons bilden, welches alle Punkte enthält

$S \leftarrow$ leere Liste;
push(S,P_1);
push(S,P_2);
for $i \leftarrow 3$ **to** n $i \leftarrow n-1$ **to** 1 **do**
  while *S.size() > 1 und* $(P_i - S.top()) \times (S.top() - S.secondTop()) < 0$ **do**
    pop(S);
  end
  push(S,P_i);
end

Die Laufzeit dieses Algorithmus ist $\Theta(n \log n)$, da wir die Punkte zunächst sortieren müssen, um sie von Links nach Rechts zu verbinden. Das Berechnen der konvexen Hülle selbst hat eine Laufzeit von $\Theta(n)$, da jeder Punkt höchstens einmal gelöscht und einmal hinzugefügt wird.

17 C++ Vertieft: Funktoren und Lambda

Lec 13

Wir wollen uns noch einmal mit generischer Programmierung beschäftigen. Betrachten wir die folgende Filterfunktion

```
template<typename T, typename Function>
void filter(const T& collection, Function f){
    for (const auto& x : collection){
        if (f(x)) std::cout << x << " ";
    }
    std::cout << std::endl;
}
```

Damit dies Funktioniert, muss das erste Argument einen Iterator besitzen, wobei wir die Elemente ausgeben können. Das zweite Argument muss eine Funktion sein, welche ein Element als Argument nimmt.

Nun wollen wir den Code parametrisch auf einen Teil des Algorithmus machen. wie `filter(container, predicate)`.

In unserem Beispiel ist filter eine Funktion, welche eine Funktion als Argument nimmt. Man spricht von Funktionen höherer Ordnung.

Interessant wird es nun, wenn wir als Funktion eine Funktion between nehmen, welche prüft, ob ein Element zwischen zwei Werten liegt. Wir könnten dort zwei Argumente übergeben. Doch das ist nicht möglich, da die Funktion nur ein Argument haben darf.

Eine mögliche Lösung ist es, eine Klasse zu erstellen, welche die Funktionalität von between implementiert. Diese Klasse könnte dann die Werte, zwischen welchen wir prüfen wollen, als Membervariablen speichern. Wir können dann eine Instanz dieser Klasse erstellen und diese Instanz als Funktion übergeben. Dies funktioniert jedoch nicht, da hierbei auch der `this`-Zeiger übergeben wird.

Die Lösung ist es, den Klammeroperator zu überladen.

```
struct Between{
    int a, b;
    bool operator()(int x) const {
        return a < x && x < b;
    }
};
```

Somit wären dann auch Expressionen wie `b(13)` möglich, wobei `b` eine Instanz der Klasse `Between` ist.

Ein Funktor ist eine Klasse, welche den Klammeroperator überlädt. Funktoren sind nützlich, da sie es ermöglichen, Funktionen mit Zustand zu erstellen. Dies funktioniert natürlich auch mit Templates.

Diese Variante ist umständlich, da wir dem Prädikat einen Namen geben müssen, was oft unnötig ist, da wir das Prädikat oft nur einmal verwenden wollen. Ausserdem sind Gute Namen nicht immer möglich, und oft ist der Code an anderer Stelle als seine Anwendung. Ausserdem schreiben wir extrem viel Code, für recht wenig funktionalität.

Eine bessere Lösung bieten Lambda-Ausdrücke. Ein Lambda-Ausdruck wird geschrieben als

```
[](int x)->bool {return x%2 == 0;}
[from, to](int x)->bool {return from < x && x < to;}
```

Innerhalb der eckigen Klammern können wir angeben, welche Variablen aus dem umgebenden Kontext wir verwenden wollen.

Dies ist eigentlich nur syntaktischer Zucker, aus welchem der Compiler einen passenden Funktor erzeugt.

Dies können wir zum Beispiel auch verwenden, wenn wir die sort Funktion nicht nach kleiner gleich sortieren wollen, sondern nach einem anderen Kriterium, wie zum Beispiel der Anzahl der Einsen in der Binärdarstellung der Zahl.

Lambda Expressions evaluieren zu einem temporären Objekt, einer closure.

Eine sehr einfache Lambda expression wäre

```
[](){std::cout << "Hello World!" << std::endl;}
// Aufruf
[](){std::cout << "Hello World!" << std::endl;}();
// Speichern
auto f = [](){std::cout << "Hello World!" <<
↪  std::endl;};
```

Die Minimale Lambda Expression ist `[]()`. Es gibt also keine Parameter und keinen Rückgabewert.

Angenommen wir haben einen Vektor von integers und wir wollen die Summe bilden. Klassischerweise haben wir dies mit einer Schleife gemacht. Wir können dies nun aber auch mit einem Funktor machen, welcher als zweite Funktion das Resultat ausgeben kann. Danach, können wir `std::for_each` verwenden.

```
int main(){
   std::vector<int> v = {1, 2, 3, 4, 5};

   Sum<int> sum;
   std::for_each(v.begin(), v.end(), sum);

   std::cout << sum.get() << std::endl;
}
```

Dies funktioniert jedoch nicht, da `std::for_each` die Funktion als Referenz übergibt, und somit der Zustand des Funktors nicht verändert wird.

Mögliche Lösungen sind

```
sum = std::for_each(v.begin(), v.end(), sum);

std::for_each(v.begin(), v.end(), std::ref(sum));
```

Wir können uns die get funktion auch sparen, wenn wir eine Variable `sum` als Referenz in der Klasse Speichern.

```
template<typename T>
class Sum{
    T& sum;
public:
    Sum(T& sum) : sum(sum) {}
    void operator()(const T& x){
        sum += x;
    }
};

int sumv = 0;

std::for_each(v.begin(), v.end(), Sum<int>(sumv));
std::cout << sumv << std::endl;
```

Dies können wir noch schöner mit einem Lambda Ausdruck machen, da wir hier die Variable `sumv` direkt in der Lambda expression verwenden können.

```
int sumv = 0;
std::for_each(v.begin(), v.end(), [&sumv](const int& x){
    sumv += x;
});
std::cout << sumv << std::endl;
```

Wichtig ist hierbei sumv per Referenz zu übergeben, damit wir den Zustand der Variable verändern können. Wenn wir einen Wert nicht als Referenz übergeben, so wird er als Konst angenommen.

Wenn wir in die eckigen Klammern ein Ampersand geben, so werden alle Variablen als Referenz übergeben. Wenn wir ein Gleichheitszeichen geben, so werden alle Variablen als Wert übergeben. Wichtig hierbei ist, dass Werte bei der Definition der Lambda expression kopiert werden, und somit nicht mehr verändert werden können.

```
int v = 42;
auto func = [=]{std::cout << v << std::endl;};
v = 7;
func(); // Gibt 42 aus, da v bei der Definition von func
↪  kopiert wurde
```

Für Lambdas in Klassen ist wichtig, dass der this pointer seit C++20 explizit übergeben werden muss.

18 Geometrische Algorithmen II

Wir betrachten das Problem, bei welchem wir eine Menge von n Intervallen haben, so dass keine zwei Start oder Endpunkte gleich sind. Wir wollen nun die maximale Anzahl von überlappenden Intervallen finden.

Wir bemerken, wenn wir ein bestimmtes x haben, so ist es extrem einfach zu sehen, wie viele Intervalle sich bei diesem x überlappen. Es sind genau die Intervalle, welche ein Startpunkt haben, welcher kleiner oder gleich x ist, und einen Endpunkt haben, welcher grösser als x ist.

Das Problem wäre gelöst, könnten wir für alle $x \in \mathbb{R}$ die Anzahl bestimmen. Doch kontinuierliche Werte zu überprüfen ist unmöglich. Die Idee von Line Sweep ist es, nur die Werte zu überprüfen, welche Start oder Endpunkte von Intervallen sind. Wir sortieren also alle Start und Endpunkte der Intervalle und gehen dann von Links nach Rechts durch diese Punkte. Wir führen dann einen Zähler mit, welcher die Anzahl der überlappenden Intervalle bei diesem Punkt angibt.

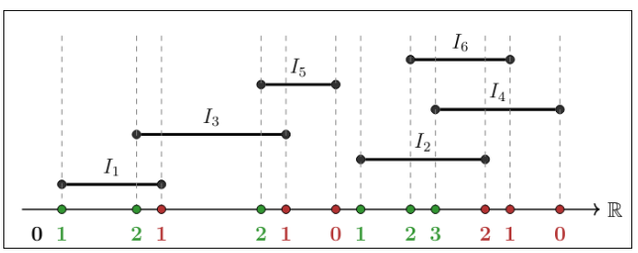


Figure 38: Line Sweep Algorithmus zur Berechnung der maximalen Anzahl von überlappenden Intervallen

Die Laufzeit dieses Algorithmus ist $\Theta(n \log n)$, da wir die Start und Endpunkte sortieren müssen. Das Durchlaufen der Punkte und Aktualisieren des Zählers hat eine Laufzeit von $\Theta(n)$, da jeder Punkt genau einmal besucht wird.

> Tip 18.1:
> Wichtig ist das für Sweep Line sortiert werden muss! Dies ist häufig das teuerste Element eines Line Sweep Algorithmus.

18.1 Oberste Strecken

Wir haben n horizontale Strecken in der Ebene, welche durch ihre Start- und Endpunkte definiertt werden. Wir wollen nun die Menge der Strecken finden, welche für ein bestimmtes x am höchsten liegen. Es könnte mehrere Strecken geben, welche am selben x-Wert liegen, aber wir wollen nur die oberste Strecke zurückgeben.

Erneut bemerken wir, dass es extrem einfach ist, die oberste Strecke für ein bestimmtes x zu bestimmen. Ausserdem reicht es nur die Punkte zu überprüfen, welche Start oder Endpunkte von Strecken sind. Es ändert sich also nichts zwischen Start und Endpunkten.

In diesem Fall ist es besonders sinnvoll, die Strecken in einem *balanced binary search tree* zu speichern, da wir die oberste Strecke schnell finden wollen und wir beliebige Strecken hinzufügen und entfernen wollen.[2]

Lec 14

[2]Das Max-Heap verletzt die zweite Bedingung.

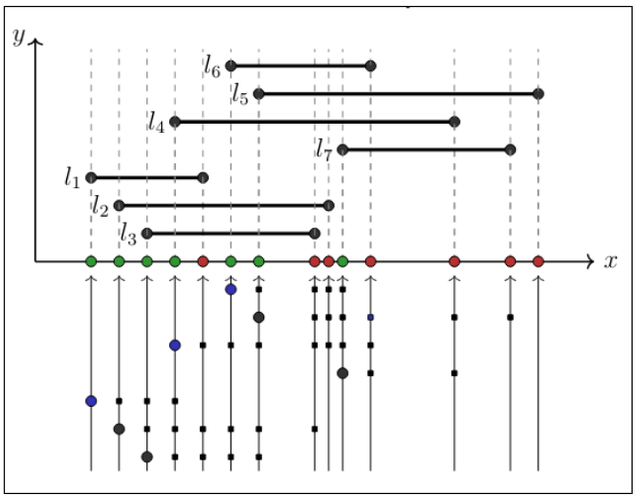


Figure 39: Line Sweep Algorithmus zur Berechnung der obersten Strecken

Im Allgemeinen definieren wir bei Sweep Line Eventpunkte S, und einen Sweep-Line Zustand A. Dann müssen wir die Eventpunkte sortieren, und eine Funktion *handleEvent* implementieren, welche die Sweep-Line bei einem Eventpunkt aktualisiert. Die Datenstrukturen für S und A sollten beschrieben werden.

18.2 Benachbarte Strecken

Gegeben sind n horizontale Strecken in der Ebene, welche durch ihre Start- und Endpunkte definiert werden. Wir wollen nun alle Paare von Strecken finden, welche für ein bestimmtes x benachbart sind. Es könnte mehrere Strecken geben, welche am selben x-Wert liegen, aber wir wollen nur die benachbarten Strecken zurückgeben.

Hier ist es besonders nützlich einen *balanced binary search tree* zu verwenden, da wir die benachbarten Strecken schnell finden wollen und wir beliebige Strecken hinzufügen und entfernen wollen. Immer alle Benachbarungen neu zu überprüfen wäre aber in $\Theta(n^2)$, da es $O(n)$ Benachbarungen gibt, und $\Theta(n)$ Events passieren. Doch wir müssen nicht alle Benachbarungen überprüfen, sondern nur die Benachbarungen der Strecken, welche hinzugefügt oder entfernt werden. Es gibt also nur $O(1)$ Benachbarungen, welche überprüft werden müssen in $\log(n)$ Zeit und somit ist die Laufzeit dieses Algorithmus $\Theta(n \log n)$, da wir die Start und Endpunkte sortieren müssen.

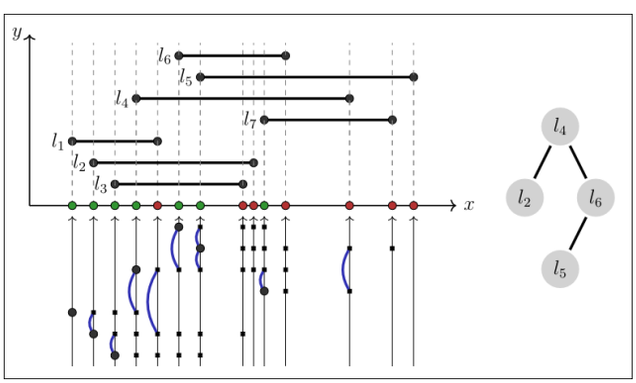


Figure 40: Line Sweep Algorithmus zur Berechnung der benachbarten Strecken

Hierbei sind die Eventpunkte die Start und Endpunkte der Strecken, und der Sweep-Line Zustand ist die Menge der Strecken, welche aktuell über der Sweep-Line liegen sortiert nach y. Bei einem Startpunkt fügen wir die Strecke zum Sweep-Line Zustand hinzu, und überprüfen die Benachbarungen der Strecke. Bei einem Endpunkt entfernen wir die Strecke aus dem Sweep-Line Zustand, und überprüfen die Benachbarungen der entfernten Strecke.

18.3 Schnittpunkte von Strecken

Gegeben sind n nicht-senkrechte Strecken in $\mathbb{R}^2$ so dass alle x und y Koordinaten von Start- und Endpunkten verschieden sind. Ausserdem soll jeder Schnittpunkt von genau zwei Strecken gebildet werden. Wir wollen nun einen Schnittpunkt der Strecken finden.

Naiv alle Strecken zu testen, wäre in $\Theta(n^2)$. Wir wollen dies aber in $\Theta(n \log n)$ haben. Wir bemerken, dass Strecken welche sich schneiden, irgendwann benachbart sein müssen. Wir können also einen Line Sweep Algorithmus verwenden, bei welchem die Eventpunkte die Start- und Endpunkte der Strecken sowie die Schnittpunkte der Strecken sind. Der Sweep-Line Zustand ist die Menge der Strecken, welche aktuell über der Sweep-Line liegen sortiert nach y.

Das Problem ist nun, dass sich die Reihenfolge der Strecken im Sweep-Line Zustand ändern kann. Wir können also nicht bezüglich der Reihenfolge der Startpunkte sortieren. Da wir aber nur einen Schnittpunkt suchen, wird sich die Reihenfolge der Strecken im Sweep-Line Zustand nicht ändern, ausser wenn wir einen Schnittpunkt erreichen, aber dann sind wir fertig.

Wenn wir alle Schnittpunkte suchen wollen, so müssen wir die Schnittpunkte als Eventpunkte hinzufügen. Wenn wir einen Schnittpunkt erreichen, so müssen wir die Reihenfolge der Strecken im Sweep-Line Zustand ändern, und die neuen Benachbarungen überprüfen. Es könnte sein, dass sich dadurch neue Schnittpunkte ergeben, welche wir dann als Eventpunkte hinzufügen müssen.

Da wir nun auch in S Eventpunkte einfügen müssen, ändern wir die Datenstruktur von S zu einem *balanced binary search tree*, oder einem *min heap*. Die Laufzeit dieses Algorithmus ist $\Theta((n+k)\log n)$, wobei k die Anzahl der Schnittpunkte ist, da wir $O(n+k)$ Eventpunkte haben, und jedes Event in $\Theta(\log n)$ bearbeitet wird.

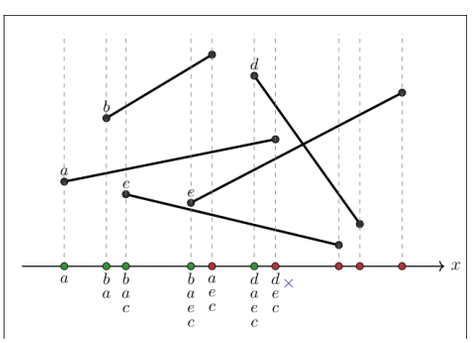


Figure 41: Line Sweep Algorithmus zur Berechnung der Schnittpunkte von Strecken

Noch als technisches Detail, sollte jeder Schnittpunkt nur einmal zu S hinzugefügt werden, um Probleme mit doppelten Schnittpunkten zu vermeiden.

18.4 Dichtestes Punktepaar

Gegeben ist eine Menge P von n Punkten in $\mathbb{R}^2$. Wir wollen nun das dichteste Paar von Punkten in P finden also die zwei Punkte $p, q \in P$ mit minimalem Abstand.

Die Idee ist wie so oft Divide and Conquer. Dazu nehmen wir die vertikale Mittellinie durch einen Punkt M und teilen die Punkte in P in zwei Hälften P_L und P_R auf. Wir rufen dann rekursiv die Funktion auf P_L und P_R auf, um die dichtesten Paare in diesen Teilmengen zu finden. Wir erhalten also zwei Abstände d_L und d_R, welche die Abstände der dichtesten Paare in P_L und P_R sind. Wir nehmen nun den kleineren dieser beiden Abstände, also $d = \min(d_L, d_R)$.

Nun müssen wir noch die Paare von Punkten betrachten, welche sich auf verschiedenen Seiten der Mittellinie befinden. Es könnte nämlich sein, dass das dichteste Paar von Punkten aus einem Punkt in P_L und einem Punkt in P_R besteht. Dazu betrachten wir den vertikalen Streifen um die Mittellinie, welcher die Punkte enthält, welche maximal d von der Mittellinie entfernt sind. Wir nennen diesen Streifen S. Es reicht nun, die Punkte in diesem Streifen zu betrachten, da alle Punkte, welche weiter als d von der Mittellinie entfernt sind, nicht näher beieinander liegen können als d. In degenerierten Fällen kann dies aber auch zu einer Laufzeit von $\Theta(n^2)$ führen, wenn alle Punkte auf einer vertikalen Linie liegen. Wir können dies aber vermeiden in dem wir für jeden Punkt auf der Linken Seite nur die eine 2δ Vertikale Box bauen. In dieser Box können maximal 8 Punkte liegen, und somit müssen wir nur 8 Punkte pro Punkt auf der Linken Seite überprüfen, was zu einer Laufzeit von $\Theta(n \log n)$ pro Rekursionsschritt führt.

Damit ist die Rekursionsgleichung

$$T(n) = 2 \cdot T(\frac{n}{2}) + c \cdot n \cdot \log n \Rightarrow T(n) = O(n \log^2 n).$$

Dies können wir verbessern in dem wir am Anfang sortieren und dann für die Teilschritte die Punkte bereits sortiert übergeben, und somit die Sortierung in den Rekursionsschritten vermeiden. Dann ist die Rekursionsgleichung

$$T(n) = 2 \cdot T(\frac{n}{2}) + c.$$

Dies führt zu einer Laufzeit von $O(n \log n)$.

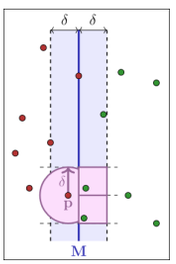


Figure 42: Divide and Conquer Algorithmus zur Berechnung des dichtesten Punktepaares

Lec 15

19 Graphen

Ein GRAPH G ist ein Paar (V, E), wobei V eine Menge von KNOTEN und E eine Menge von KANTEN ist. Eine Kante ist ein Paar von Knoten, also $E \subseteq V \times V$.

$w \in V$ ist ADJAZENT zu $v \in V$, wenn w ein Nachfolger von v ist.

Nachfolger von v: $N^+(v) = \{w \in V | (v, w) \in E\}$.

Vorgänger von v: $N^-(v) = \{w \in V | (w, v) \in E\}$.

Der Eingangsgrad (in-degree) von v ist die Anzahl der Vorgänger von v, also $d^-(v) = |N^-(v)|$. Der Ausgangsgrad (out-degree) von v ist die Anzahl der Nachfolger von v, also $d^+(v) = |N^+(v)|$.

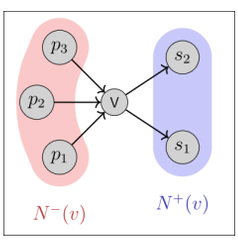


Figure 43: Ein und Ausgangsgrade von Knoten in einem gerichteten Graphen

Ein WEG der Länge k ist eine Sequenz $(v_1, \ldots, v_{k+1})$ von Knoten, so dass $(v_i, v_{i+1}) \in E$ für alle $i = 1, \ldots, k$. Ein EINFACHER WEG hat die zusätzliche Bedingung, dass alle Knoten verschieden sind. Ein KREIS (cycle) ist ein Weg mit gleichem Start- und Endknoten, also $v_1 = v_{k+1}$. Ein EINFACHER KREIS hat die zusätzliche Bedingung, dass alle Knoten verschieden sind, ausser dem Start- und Endknoten.

Wir nennen einen Weg, welcher jede Kante genau einmal benutzt, einen EULER'SCHEN WEG (Eulerian path). Wenn ein Euler'scher Weg ein Kreis ist, so nennen wir ihn einen EULER'SCHEN KREIS (Eulerian cycle).

Einen UNGERICHTETER GRAPH $G = (V, E)$ beinhaltet eine Menge von Knoten V und eine Menge von ungerichteten Kanten E, wobei $E \subseteq \{\{v, w\} | v, w \in V\}$. Beachte, dass die Kante $\{v, w\}$ gleich der Kante $\{w, v\}$ ist.

Hier ist w adjazent zu v, wenn $\{v, w\} \in E$. Die Nachbarn von v sind $N(v) = \{w \in V | \{v, w\}\} \in E\}$. Der Grad von v ist $\deg(v) = |N(v)|$. Als spezialfall zählen Schleifen doppelt, da sie zwei Nachbarn hinzufügen.

Eine wichtige Eigenschaft von einem eulerschen Weg auf ungerichteten Graphen ist, dass er in jeden Knoten gleich oft eintritt wie austritt (ausser in den Start- und Endknoten, welche gleich sein können).

Theorem 19.1:

Ein ungerichteter Graph $G = (V, E)$ hat einen

1. Euler'schen Kreis genau dann, wenn jeder Knoten von G einen geraden Grad hat.
2. Euler'schen Weg genau dann, wenn genau 0 oder 2 Knoten von G einen ungeraden Grad haben.

Um einen Euler'schen Kreis zu finden, können wir bei einem beliebigen Knoten starten und eine beliebige Kante entlang gehen, welche noch nicht benutzt wurde. Dies wiederholen wir, bis alle Kanten benutzt wurden. Danach fügen wir alle Kreis welche wir gefunden haben zusammen, um den Euler'schen Kreis zu erhalten.

Lemma 19.2: Handshaking Lemma

In jedem Graphen $G = (V, E)$ ist

1. $\sum_v \deg^-(v) = \sum_v \deg^+(v) = |E|$.
2. $\sum_v \deg(v) = 2|E|$, falls G ungerichtet ist.

Wir bemerken, beim gerichteten Graphen dass ohne Schleifen,

$$0 \leq |E| \leq |V| \cdot (|V| - 1).$$

und beim gerichteten Graphen mit Schleifen,

$$0 \leq |E| \leq |V|^2.$$

Beim ungerichteten Graphen ohne Schleifen,

$$0 \leq |E| \leq \frac{|V| \cdot (|V| - 1)}{2}.$$

und beim ungerichteten Graphen mit Schleifen,

$$0 \leq |E| \leq \frac{|V| \cdot (|V| + 1)}{2}.$$

Gewisse einfache Fragen beinhalten:

- Existiert eine Kante zwischen v und w?
- Was sind die Nachfolger von v?
- Was ist der Eingangsgrad von v?

Eine Mögliche Variante einen Graphen zu repräsentieren ist die ADJAZENZMATRIX. Hierbei haben wir eine $|V| \times |V|$ Matrix, welche an der Stelle (v, w) eine 1 hat, wenn es eine Kante von v nach w gibt, und 0 sonst. Diese Darstellung ist besonders nützlich, wenn der Graph dicht ist, also viele Kanten hat. Die Matrix ist symmetrisch, wenn der Graph ungerichtet ist. Ausserdem sind Einträge auf der Diagonalen 0, wenn es keine Schleife gibt. Der Nachteil dieser Darstellung ist, dass sie viel Speicherplatz benötigt, da sie $\mathcal{O}(|V|^2)$ Speicherplatz benötigt, auch wenn der Graph nur wenige Kanten hat.

Eine andere Speichermöglichkeit ist die ADJAZENZLISTE. Hierbei haben wir für jeden Knoten eine Liste von Nachbarn. Diese Darstellung ist besonders nützlich, wenn der Graph dünn ist, also wenige Kanten hat. Der Speicherplatzbedarf ist $\mathcal{O}(|V| + |E|)$, da wir für jeden Knoten eine Liste haben, und die Länge der Listen insgesamt $|E|$ ist nach dem Handshaking Lemma.

Seltener verwendet wird auch eine Kantenliste, welche eine Liste aller Kanten enthält. Diese Darstellung ist besonders nützlich, wenn wir nur die Kanten betrachten wollen, und nicht die Nachbarn von Knoten. Der Speicherplatzbedarf ist $\mathcal{O}(|E|)$, da wir nur die Kanten speichern.

19.1 Erste Graphenalgorithmen

Um einen Graphen zu erkunden kann man zum Beispiel die Tiefensuche (Depth-First Search, DFS) verwenden. Hierbei starten wir bei einem beliebigen Knoten und besuchen diesen Knoten. Danach besuchen wir einen Nachbarn dieses Knotens, welcher noch nicht besucht wurde. Dies wiederholen wir, bis es keinen Nachbarn mehr gibt, welcher noch nicht besucht wurde. Danach gehen wir zurück zum letzten Knoten, welcher noch unbesuchte Nachbarn hat, und besuchen einen dieser Nachbarn. Dies wiederholen wir, bis alle Knoten besucht wurden.

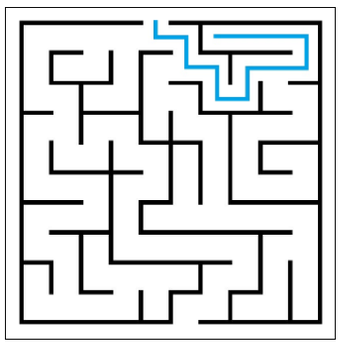


Figure 44: Tiefensuche (Depth-First Search, DFS) auf einem gerichteten Graphen

Um dies zu implementieren, gibt es für uns drei Zustände in denen ein Knoten sich befinden kann: unbesucht, besucht aber nicht vollständig erkundet, und besucht und vollständig erkundet.

Der folgende Algorithmus besucht alle Knoten, welche von einem Startknoten v aus erreichbar sind. Um alle Knoten zu besuchen müssen wir noch einen For-Loop hinzufügen, welcher über alle Knoten iteriert und die Tiefensuche für alle unbesuchten Knoten aufruft. Die Laufzeit ohne

Algorithm 14: Tiefensuche (Depth-First Search, DFS)

Input: Graph $G = (V, E)$, Knoten v
v.color ← gray;
for $w \in N^+(v)$ **do**
  if $w.color = white$ **then**
    DFS(G,w);
  end
end
v.color ← black;

Rekursion ist $\Theta(1 + \deg^+(V))$. Die gesamte Laufzeit ist somit $\Theta(|V| + |E|)$, da wir jeden Knoten und jede Kante genau einmal besuchen.

Dieser Algorithmus ist unter anderem nützlich, um Zyklen in einem Graphen zu erkennen.

Eine andere Möglichkeit einen Graphen zu erkunden ist die Breitensuche (Breadth-First Search, BFS). Hierbei starten wir bei einem beliebigen Knoten und besuchen diesen Knoten. Danach besuchen wir alle Nachbarn dieses Knotens, welche noch nicht besucht wurden. Danach besuchen wir alle Nachbarn der Nachbarn, welche noch nicht besucht wurden. Dies wiederholen wir, bis alle Knoten besucht wurden.

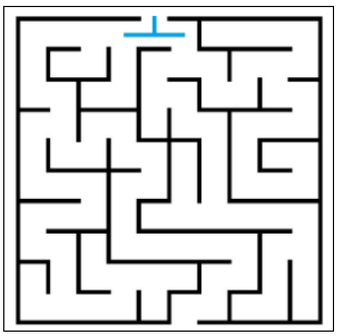


Figure 45: Breitensuche (Breadth-First Search, BFS) auf einem gerichteten Graphen

Praktisch implementieren können wir dies mit einer Queue, in welcher wir die Knoten speichern, welche wir als nächstes besuchen wollen. Die Laufzeit dieses Algorithmus ist ebenfalls $\Theta(|V| + |E|)$, da wir jeden Knoten und jede Kante genau einmal besuchen.

Algorithm 15: Breitensuche (Breadth-First Search, BFS)

Input: Graph $G = (V, E)$, Knoten v
Queue $Q \leftarrow \varnothing$;
enqueue(Q, v);
v.seen ← true;
while $Q \neq \varnothing$ **do**
  u ← dequeue(Q);
  for $w \in N^+(u)$ **do**
    if $w.seen = false$ **then**
      enqueue(Q, w);
      w.seen ← true;
    end
  end
end

Lec 16

Definition 19.3: Topologische Sortierung

Eine topologische Sortierung eines gerichteten Graphen $G = (V, E)$ ist eine Anordnung der Knoten von G in einer linearen Reihenfolge, so dass für jede gerichtete Kante $(u, v) \in E$ der Knoten u vor dem Knoten v in der Reihenfolge erscheint.

Als Beispiel dazu können wir Kleiderstücke betrachten, welche wir anziehen wollen. Die Knoten des Graphen sind die Kleiderstücke, und es gibt eine gerichtete Kante von Knoten u zu Knoten v, wenn wir u anziehen müssen, bevor wir v anziehen können.

Wichtig: Die topologische Sortierung ist nur für gerichtete azyklische Graphen (DAGs) definiert.

Theorem 19.4:

Ein gerichteter Graph $G = (V, E)$ hat genau dann eine topologische Sortierung, wenn G azyklisch ist.

Um eine topologische Sortierung zu finden, können wir jeweils einen Knoten mit Eingangsgrad 0 auswählen, diesen Knoten aus dem Graphen entfernen, und diesen Knoten zur topologischen Sortierung hinzufügen. Dies

wiederholen wir, bis alle Knoten entfernt wurden. Wenn es zu einem Zeitpunkt keinen Knoten mit Eingangsgrad 0 gibt, so hat der Graph einen Zyklus und somit keine topologische Sortierung.

Um die Laufzeit dieses Algorithmus nicht zu teuer zu machen, können wir zu Beginn durch alle Kanten durchgehen, um den Eingangsgrad jedes Knotens zu berechnen. Danach können wir alle Knoten mit Eingangsgrad 0 in eine Queue einfügen. Bei jedem Schritt entfernen wir einen Knoten aus der Queue, und verringern den Eingangsgrad seiner Nachbarn um 1. Wenn der Eingangsgrad eines Nachbarn 0 wird, fügen wir diesen Nachbarn zur Queue hinzu. Die Laufzeit dieses Algorithmus ist $\Theta(|V|+|E|)$, da wir jeden Knoten und jede Kante genau einmal besuchen.

20 Kürzeste Wege I

Gegeben seien Städte A-Z und Distanzen zwischen Städten. Wir wollen nun die kürzeste Route von Stadt A nach Stadt Z finden.

Ein GEWICHTETER GRAPH $G = (V, E, c)$ ist ein Graph, welcher zusätzlich eine Kostenfunktion $c : E \to \mathbb{R}$ hat, welche jeder Kante eine Kosten zuordnet.

Das Gewicht eines Weges ist definiert als die Summe der Kosten der Kanten, welche in diesem Weg enthalten sind. Wir wollen nun den kürzesten Weg von einem Startknoten s zu einem Zielknoten t finden, also einen Pfad von s nach t, welcher das kleinste Gewicht hat.

Wir schreiben $s \overset{p}{\rightsquigarrow} t$ für einen Weg p von s nach t. Der kürzeste Weg ist dann

$$\delta(s,t) = \begin{cases} \infty & \text{kein Weg von } s \text{ nach } t \text{ existiert} \\ \min_{s \rightsquigarrow t} c(p) & \text{sonst} \end{cases}.$$

Wir bemerken, dass der kürzeste Weg nicht immer Eindeutig sein muss. Ausserdem muss nicht zwingend ein kürzester Weg existieren, da es Zyklen mit negativem Gewicht geben könnte, welche den Weg immer weiter verkürzen können.

Ein trivialer Algorithmus zur Berechnung des kürzesten Weges könnte sein, alle Wege auszuprobieren und das Minimum zu nehmen. Dies ist extrem ineffizient, da es exponentiell viele Wege geben kann.

Betrachten wir den einfachsten Fall, bei welchem das Kantengewicht konstant 1 ist. In diesem Fall können wir Breitensuche verwenden, um den kürzesten Weg zu finden, da wir so zuerst alle Wege der Länge 1, dann alle Wege der Länge 2, etc. besuchen, und somit den kürzesten Weg finden, sobald wir das Ziel erreichen. Die Laufzeit dieses Algorithmus ist $\Theta(|V| + |E|)$.

20.1 Dijkstra's Algorithmus

Nehmen wir an, alle gewichte sind positiv. In diesem Fall können wir Dijkstra's Algorithmus verwenden, um den kürzesten Weg zu finden.

Die Idee von Dijkstra's Algorithmus ist es, eine Menge S von Knoten zu haben, für welche wir den kürzesten Weg von s zu diesen Knoten bereits gefunden haben. Zu Beginn ist $S = \{s\}$, da der kürzeste Weg von s zu sich selbst 0 ist. Danach wählen wir einen Knoten v aus $V \setminus S$, welcher am nächsten zu s liegt. Es kann keinen kürzeren Weg von s zu v geben, da alle Wege von s nach v durch S führen müssen, und alle Knoten in S weiter von s entfernt sind als v. Wir fügen diesen Knoten v zu S hinzu. Für alle anderen Knoten welche Nachfolger von s sind, können wir nun die obere Schranke für den kürzesten Weg von s zu diesen Knoten aktualisieren. Dies können wir nun so lange wiederholen, bis wir das Ziel t zu S hinzufügen, oder bis $S = V$ ist.

Wir bemerken, dass wir jeweils nur die Knoten updaten müssen, welche Nachfolger von v sind, da alle anderen Knoten nicht durch v erreichbar sind, und somit nicht näher an s liegen können als vorher.

Die Invariante des Algorithmus ist, dass nach i Schritten, der kürzeste Weg zu i Knoten in S bekannt ist.

Um den Algorithmus zu implementieren, bezeichnen wir mit $d_s : V \to \mathbb{R}$ das Gewicht des kürzesten bisher gefundenen Weges. Zu Beginn ist $d_s = \infty$ für alle Knoten und wir suchen $d_s[v]$. Ausserdem müssen wir den Vorgänger π_s für jeden knoten speichern, damit wir den Pfad rekonstruieren können. Zu Beginn ist $\pi_s[v] = \text{null}$ für alle Knoten.

Algorithm 16: Dijkstra's Algorithmus

Input: Positiv gewichteter Graph $G = (V, E, c)$, Startknoten s

for $v \in V$ **do**
  $d_s[v] \leftarrow \infty$;
  $\pi_s[v] \leftarrow \text{null}$;
end
$d_s[s] \leftarrow 0$;
$U \leftarrow \{s\}$;
while $U \neq \varnothing$ **do**
  $v \leftarrow \arg\min_{u \in U} d_s[u]$;
  $U \leftarrow U \setminus \{v\}$;
  for $w \in N^+(v)$ **do**
    if $d_s[v] + c(v, w) < d_s[w]$ **then**
      $d_s[w] \leftarrow d_s[v] + c(v, w)$;
      $\pi_s[w] \leftarrow v$;
      $U \leftarrow U \cup \{w\}$;
    end
  end
end

Für die Datenstruktur von U brauchen wir ein Insert, ein Extract-Min, und ein Decrease-Key. Wir brauchen also eine Priority Queue. Dies kann man mit einem Min-Heap implementieren, mit Knoten in U als Schlüssel. Das Problem hier ist das man für Decrease-Key die Position des Knotens im Min-Heap kennen muss. Hierfür gibt es zwei Möglichkeiten: Entweder wir speichern die Position am Knoten oder Extern.

Oder, wir verwenden Lazy Deletion. Hierbei fügen wir einfach den Knoten mit dem neuen Schlüssel zum Min-Heap hinzu, und ignorieren den alten Knoten, wenn er später extrahiert wird. Nachteilhaft daran ist, dass der Speicherbedarf vom Heap bis zu $\Theta(|V|+|E|)$ wachsen kann statt $\Theta(|V|)$.

Die Laufzeit von Dijkstra unter Verwendung von Lazy Deletion ist $\Theta(|E| \log |V|)$, unter der Annahme, dass der Graph zusammenhängend ist.

Ohne Lazy Deletion, und mit einem Min-Heap, ist die Laufzeit von Dijkstra ebenfalls $\Theta(|E| \log |V|)$.

Mit Fibonacci Heaps, welche Decrease-Key in amortisierter $\mathcal{O}(1)$ Zeit unterstützen, ist die Laufzeit von Dijkstra $\Theta(|V| \log |V| + |E|)$.

20.2 A* Algorithmus

Der Djikstra Algorithmus sucht die kürzesten Wege zu allen Knoten, in alle Richtungen. Das ist zwar richtig, da die Struktur des Graphen dem Algorithmus nicht bekannt ist, jedoch ist es in aller Regel zwecksmässig in die Richtung des Zielknotens zu suchen, da wir so schneller zum Zielknoten kommen können.

Die Idee ist nun eine Abstandsheuristik $\hat{h}$ zu verwenden, welche die Distanz von einem Knoten v zum Zielknoten t schätzt. Wir können nun die Priorität eines Knotens v in der Priority Queue als $d_s[v] + \hat{h}(v)$ definieren, also die bisher gefundene Distanz von s zu v plus die geschätzte Distanz von v zu t.

Damit dies funktioniert, muss die Distanz unterschätzen werden, also $\hat{h}(v) \leq \delta(v, t)$ für alle Knoten v. Wenn dies der Fall ist, so garantiert der A* Algorithmus, dass der erste Weg von s nach t, welcher gefunden wird, auch der kürzeste Weg ist.

Wenn die Abstandsheuristik nicht monoton ist, kann es Vorkommen, dass ein Knoten mehrfach aus U entnommen und wieder Eingefügt wird, was die Laufzeit von A* verschlechtert.

21 Kürzeste Wege II

Lec 17

21.1 Bellman-Ford Algorithmus

Wie wir gesehen haben, funktioniert Dijkstra's Algorithmus nur für Graphen mit positiven Kantengewichten. Wenn es negative Kantengewichte gibt, so kann Dijkstra's Algorithmus falsche Ergebnisse liefern, da er davon ausgeht, dass der kürzeste Weg zu einem Knoten gefunden wurde, sobald dieser Knoten aus der Priority Queue extrahiert wird.

Als kurze Definition, wir RELAXIEREN eine Kante (u, v), wenn wir überprüfen, ob der kürzeste Weg von s nach v über u kürzer ist als der bisher bekannte kürzeste Weg.

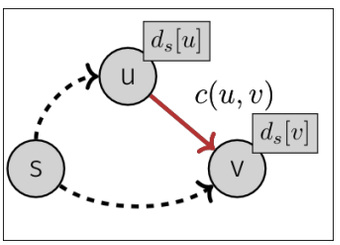


Figure 46: Relaxation einer Kante (u, v)

Ein allgemeiner Algorithmus zur Berechnung der kürzesten Wege initialisiert $d_s[v] = \infty$ für alle Knoten v, und $d_s[s] = 0$. Ausserdem $\pi_s[v] =$ null für alle Knoten v. Nun werden Schritt für Schritt Kanten relaxiert, bis keine Kante mehr relaxiert werden kann.

Die zentrale Frage ist nun, in welcher Reihenfolge die Kanten relaxiert werden sollen, und wie lange dauert es, bis der Algorithmus terminiert.

Die zentrale Beobachtung ist, dass wir eine Induktion über die Anzahl schritte in einem Weg machen können. Nachdem wir i mal alle Kanten relaxiert haben, so ist $d_s[v]$ für alle Knoten v, welche über einen Weg von Länge höchstens i erreichbar sind, korrekt.

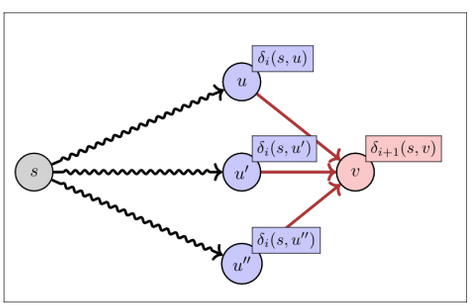


Figure 47: Zentrale Beobachtung zu kürzesten Wegen

Proposition 21.1:
Ein kürzester Weg hat keine Zyklen

Proof. Wenn der kürzeste Weg einen negativen Zyklus enthält, dann existiert kein kürzester Weg, Widerspruch.

Wenn der kürzeste Weg einen positiven Zyklus enthält, dann könnte man den Weg durch weglassen dieses Zyklus verkürzen, Widerspruch.

Wenn der kürzeste Weg einen Zyklus von Gewicht 0 enthält, dann ist per Konvention der kürzeste Weg ohne diesen Zyklus, Widerspruch. □

Somit folgt direkt, dass der kürzeste Weg von s nach t höchstens $|V|-1$ Kanten enthalten kann, da es sonst einen Zyklus geben müsste. Dies ist die Idee, des Algorithmus von Bellman-Ford, welcher alle Kanten $|V| - 1$ mal relaxiert.

Algorithm 17: Bellman-Ford Algorithmus

Input: Graph $G = (V, E, c)$, Startknoten s
for $v \in V$ **do**
  $d_s[v] \leftarrow \infty$;
  $\pi_s[v] \leftarrow$ null;
end
$d_s[s] \leftarrow 0$;
for $i = 1$ *to* $|V|$ **do**
  $f \leftarrow$ false;
  for $e \in E$ **do**
    $f \leftarrow f \vee$ Relax(e);
  end
  if $f =$ *false* **then**
    return true;
  end
end
return false;

Wenn es keinen kürzesten Zyklus gibt, dann wird sich der kürzeste Weg bei der $|V|$-ten Iteration sich noch ändern, da es einen kürzeren Weg gibt, welcher einen Zyklus enthält. Wenn es keinen kürzesten Zyklus gibt, so wird sich der kürzeste Weg bei der $|V|$-ten Iteration nicht mehr ändern, da alle kürzesten Wege höchstens $|V| - 1$ Kanten enthalten können.

Die Laufzeit dieses Algorithmus ist $\Theta(|V| \cdot |E|)$, da wir $|V|$ mal alle $|E|$ Kanten relaxieren.

Auf DAG's können wir den Algorithmus noch verbessern, indem wir im Stil einer Breitensuche die Knoten in Topologischer Reihenfolge relaxieren. Die Laufzeit dieses Algorithmus ist $\Theta(|V| + |E|)$, da wir zuerst eine topologische Sortierung in $\Theta(|V| + |E|)$ Zeit berechnen können, und danach jede Kante genau einmal relaxieren.

Zusammenfassend: Für $c \equiv 1$ können wir BFS verwenden, für $c > 0$ können wir Dijkstra's Algorithmus verwenden, bei einem DAG können wir die topologische Sortierung verwenden, und für allgemeine Graphen (ohne negative Zyklen) können wir Bellman-Ford verwenden.

21.2 Floyd-Warshall Algorithmus

Wir wollen nun die kürzesten Wege zwischen allen Paaren von Knoten in einem Graphen berechnen. Ein trivialer Algorithmus könnte sein, für jedes Paar von Knoten den kürzesten Weg mit Dijkstra's Algorithmus zu berechnen. Dies würde aber $\Theta(|V|^2 \log |V| + |V| \cdot |E|)$ Zeit benötigen, was für dichte Graphen $\Theta(|V|^3)$ ist.

Im allgemeinen suchen wir eine Matrix d welche an der Stelle (v, w) die Länge des kürzesten Weges von v nach w enthält. Dafür wollen wir erneut Induktion verwen-

den, diesmal aber über die Anzahl Zwischenknoten. Zum Beispiel ist

$$d_0[v,w] = \begin{cases} 0 & v = w \\ c(v,w) & (v,w) \in E \\ \infty & \text{sonst} \end{cases}.$$

Der k-te Schritt der Induktion ist nun

$$d_k[v,w] = \min(d_{k-1}[v,w], d_{k-1}[v,k] + d_{k-1}[k,w]).$$

Algorithm 18: Floyd-Warshall Algorithmus

```
Input: Graph G = (V, E, c)
for v, w ∈ V do
    if v = w then
        d[v, w] ← 0;
    end
    else if (v, w) ∈ E then
        d[v, w] ← c(v, w);
    end
    else
        d[v, w] ← ∞;
    end
end
for k = 1 to |V| do
    for v, w ∈ V do
        d_k[v, w] ←
          min(d_{k−1}[v, w], d_{k−1}[v, k] + d_{k−1}[k, w]);
    end
end
```

Die Laufzeit dieses Algorithmus ist $\Theta(|V|^3)$, da wir drei verschachtelte For-Loops haben, welche jeweils über alle Knoten iterieren.

21.3 Anzahl Wege

Gegeben sei eine Adjazenzmatrix A eines gerichteten Graphen, wollen wir die Anzahl der Wege von einem Knoten v zu einem Knoten w berechnen. Es ist leicht zu sehen, dass die Anzahl der Wege von v nach w der Eintrag an der Stelle (v,w) in der Matrix A^k ist, da die Anzahl der Wege von v nach w über genau k Kanten die Summe über alle Knoten u von der Anzahl der Wege von v nach u über $k-1$ Kanten mal der Anzahl der Wege von u nach w über 1 Kante ist, also

Theorem 21.2:
Sei $G = (V,E)$ ein Graph und $k \in \mathbb{N}$. Dann gibt das Element $a_{i,j}^{(k)}$ der Matrix A^k die Anzahl der Wege von Knoten i zu Knoten j mit Länge k an.

21.4 Reflexive Transitive Hülle

Die REFLEXIVE TRANSITIVE HÜLLE eines gerichteten Graphen $G = (V,E)$ ist ein gerichteter Graph $G^* = (V, E^*)$, wobei E^* alle Kanten beinhaltet, welche durch einen Pfad in G repräsentiert werden, also

$$E^* = \{(v,w) | \exists \text{ Weg von } v \text{ nach } w \text{ in } G\}.$$

Im Prinzip haben wir dieses Problem bereits gelöst, da die reflexive transitive Hülle eines Graphen G die Adjazenzmatrix A^* hat, welche definiert ist als

$$A^* = \begin{cases} 1 & \text{wenn } a_{i,j}^{(k)} > 0 \text{ für irgendein } k \\ 0 & \text{sonsts} \end{cases}.$$

Wir starten für die Berechnung von A^* mit $A^* = A$ und setzen $a_{i,i}^* = 1$ für alle i. Danach iterieren wir über alle Knoten k, und verwenden unser angepasstes Matrixprodukt, um A^* zu aktualisieren. Dies ist dann in $\Theta(|V|^4)$ Zeit möglich, da wir vier verschachtelte For-Loops haben, welche jeweils über alle Knoten iterieren. Dies lässt sich zu $\Theta(|V|^3 \log |V|)$ verbessern, wenn wir bemerken, dass die Matrizenmultiplikation assoziativ ist, und somit die Anzahl der Multiplikationen durch exponentielles Hochzählen der Matrix A reduzieren können.

Lec 18

22 Minimale Spannbäume

Als Motiviation haben wir Häuser, welche durch ein Stromnetz verbunden werden sollen. Wir wissen für jedes Paar von Häusern, wie viel es kosten würde, diese beiden Häuser direkt zu verbinden. Wir wollen nun die Häuser so verbinden, dass alle Häuser miteinander verbunden sind, und die Kosten minimal sind. Ausserdem verlangen wir, dass es nur ein zusammenhängendes Stromnetz gibt. Ausserdem soll es keine Zyklen geben.

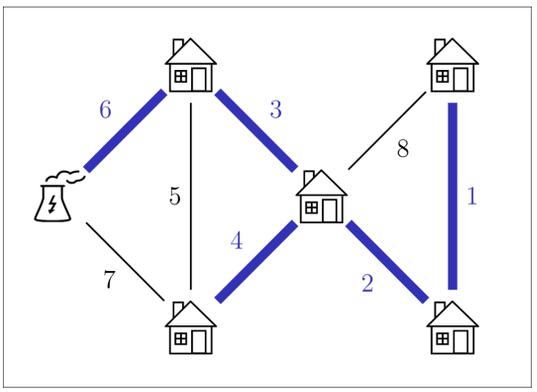


Figure 48: Ein Beispiel für ein Stromnetz, welches die Häuser verbindet.

Definition 22.1: Spannbaum

Gegeben ein ungerichteter, zusammenhängender Graph $G = (V, E)$, ist ein SPANNBAUM von G ein Teilgraph $T = (V, E')$ von G, sodass $V' = V$ und $E' \subseteq E$ ist, und T ein Baum ist, also zusammenhängend und Zyklenfrei ist.

Wir bemerken, dass ein Spannbaum von G genau $|V| - 1$ Kanten hat, da er zusammenhängend sein muss, und keine Zyklen enthalten darf. Ausserdem, gibt es für jedes Paar von Knoten genau einen Pfad.

Bisher waren Bäume gerichtete Bäume, welche eine Wurzel hatten, diese waren schwach zusammenhängend und hatten gerichtete Kanten.

Als Beobachtung, ein äquivalentes Problem dazu ist den Maximalen Spannbaum zu finden. Der Minimale Spannbaum ist nicht zwingen eindeutig[3], da es mehrere Spannbäume mit dem gleichen Gewicht geben kann.

Die Idee für den Algorithmus stammen von Jarnik, Prim und Djikstra. Wir starten mit einem $v \in V$ und lassen von dort einen minimalen Spannbaum wachsen. Dafür wählen wir jeweils die Kante mit dem kleinsten Gewicht, welche von einem Knoten im Spannbaum zu einem Knoten ausserhalb des Spannbaums führt, und fügen diese Kante zum Spannbaum hinzu. Dies wiederholen wir, bis alle Knoten im Spannbaum enthalten sind. Am ende verwerfen wir alle anderen Kanten, welche nicht im Spannbaum enthalten sind.

Um das Minimum zu finden, verwenden wir ein Min-Heap, in welchem die Schlüssel die Kosten der Kanten sind. Die

[3]Wenn alle Kanten ein unterschiedliches Gewicht haben, dann ist der minimale Spannbaum eindeutig.

Algorithm 19: Jarnik, Prim, Dijkstra Algorithmus

Input: Graph $G = (V, E, c)$, Startknoten v
$S \leftarrow \{v\}$;
$M \leftarrow \varnothing$;
for $i = 1$ *to* $|V| - 1$ **do**
  Wähle eine Kante (u, w) mit $u \in S$, $w \in V \setminus S$, und minimalem Gewicht;
  $S \leftarrow S \cup \{w\}$;
  $M \leftarrow M \cup \{(u, w)\}$;
end
return M;

Invariante des Algorithmus ist, dass nach i Schritten, der Spannbaum auf i Knoten bekannt ist.

Die Implementation ist somit wie bei Dijkstra's Algorithmus, mit der Ausnahme, dass wir die Kosten der Kanten als Schlüssel verwenden, anstatt der bisher gefundenen Distanz von s zu einem Knoten.

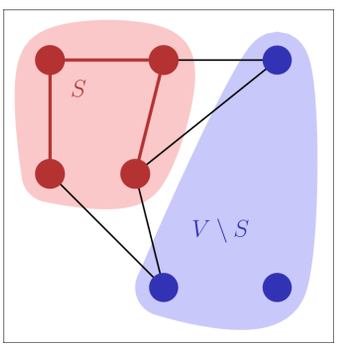


Figure 49: Die Beiden Mengen S und $V \setminus S$ während der Ausführung von Prim's Algorithmus.

22.1 Kruskal's Algorithmus

Eine alternative Variante zum Finden eines MST ist Kruskal's Algorithmus. Hierbei sortieren wir zuerst alle Kanten nach Gewicht, und fügen danach die Kanten in aufsteigender Reihenfolge zum Spannbaum hinzu, solange sie keinen Zyklus bilden. Sobald wir $|V| - 1$ Kanten hinzugefügt haben, können wir aufhören, da der Spannbaum dann vollständig ist.

Die grosse Frage ist nun, wie wir überprüfen können, ob eine Kante einen Zyklus bildet. Die Idee hier ist es sich Komponenten zu merken, welche durch die bereits hinzugefügten Kanten verbunden sind. Zu Beginn ist jede Komponente ein einzelner Knoten. Wenn wir eine Kante (u, v) hinzufügen, so müssen wir überprüfen, ob u und v in der gleichen Komponente sind. Wenn dies der Fall ist, so würde die Kante einen Zyklus bilden, und wir dürfen sie nicht hinzufügen. Wenn dies nicht der Fall ist, so können wir die Kante hinzufügen, und die Komponenten von u und v zusammenführen.

Wir möchten eine Datenstruktur, welche die folgenden Operationen unterstützt:

- `MakeSet(`v`)`: Erstelle eine neue Komponente, welche nur den Knoten v enthält.

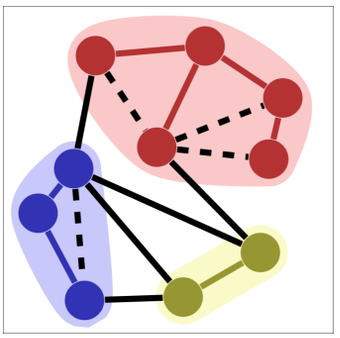


Figure 50: Die Komponenten während der Ausführung von Kruskal's Algorithmus.

- `Find`(v): Finde diejenige Komponente, welche den Knoten v enthält.
- `Union`(C_1, C_2): Vereine die Komponenten C_1 und C_2 zu einer neuen Komponente.

22.2 Union-Find Datenstruktur

Das allgemeine Problem, welches wir haben, ist das wir eine Partition haben. Der abstrakte Datentyp, heisst Union-Find. Die Idee ist es einen Baum für jede Teilmenge in der Partition zu haben, und die Wurzel eines Baumes als Repräsentant der Teilmenge zu verwenden. Dabei speichern wir für jeden Knoten einen Zeiger auf seinen Elternknoten, und die Wurzel eines Baumes zeigt auf sich selbst. Dies kann auch recht einfach mit einem Array repräsentiert werden, in welchem der Index des Arrays der Knoten ist, und der Wert des Arrays der Elternknoten ist. Wenn wir nun die Zugehörigkeit suchen wollen, so müssen wir einfach den Pfad von diesem Knoten zur Wurzel des Baumes verfolgen. Die Laufzeit von `Find` ist somit proportional zur Höhe des Baumes.

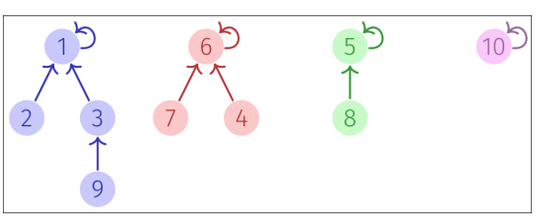


Figure 51: Die Union-Find Datenstruktur.

Implementieren können wir das wie folgt:

MakeSet: $p[i] \leftarrow i$

Find: while$(p[i] \neq i)$ do $i \leftarrow p[i]$; return i

Union(i, j): $p[j] \leftarrow i$. Hierbei muss i und j die Wurzeln der Bäume sein, welche wir vereinigen wollen.

Nun wollen wir noch die Höhe des Baumes beschränken, damit die Laufzeit von `Find` nicht zu hoch wird. Dafür hängen wir einfach immer den kleineren Baum an den grösseren Baum an. Die Höhe eines Baumes ist dadurch logarithmisch!

Eine alternative Verbesserung ist es wenn wir bei `Find` den Pfad von einem Knoten zur Wurzel verfolgen, so können wir alle Knoten auf diesem Pfad direkt an die Wurzel hängen, damit die Höhe des Baumes noch weiter reduziert wird. Es lässt sich zeigen, dass die amortisierte Laufzeit von `Find` mit dieser Verbesserung $\mathcal{O}(\alpha(n))$ ist, wobei $\alpha(n)$ die inverse Ackermann-Funktion ist, welche extrem langsam wächst.

Lec 19

23 Flüsse in Netzwerken

Ein Flussnetzwerk ist ein gerichteter, gewichteter Graph $G = (V, E, c)$, mit Kapazitäten $c : E \to \mathbb{R}^+$. Der Einfachheit verlangen wir zusätzlich, dass es keine antiparallelen Kanten gibt. Ausserdem gibt es zwei spezielle Knoten, die Quelle s und die Senke t.

Ein Fluss ist eine Funktion $f : E \to \mathbb{R}^{\geq 0}$, so dass wir eine Kapazitätsbeschränkung haben, also $f(e) \leq c(e)$ für alle $e \in E$, und wir brauchen eine Flusserhaltung, also für alle Knoten $v \in V \setminus \{s, t\}$ muss die Summe der Flüsse auf den eingehenden Kanten gleich der Summe der Flüsse auf den ausgehenden Kanten sein.

$$\sum e \in E^-(v) f(e) = \sum_{e \in E^+(v)} f(e).$$

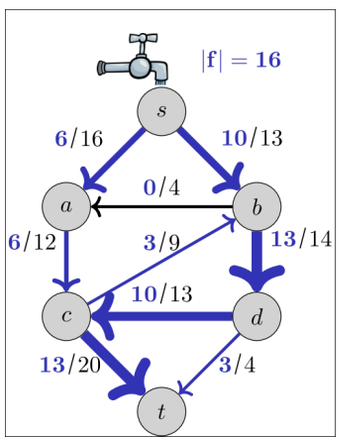


Figure 52: Ein Beispiel für ein Flussnetzwerk.

Das Problem das wir nun lösen wollen ist gegeben ein Flussnetzwerk, einen maximalen Fluss von der Quelle s zur Senke t zu finden, also eine Funktion f, welche die Kapazitätsbeschränkung und die Flusserhaltung erfüllt, und für welche die Summe der Flüsse auf den ausgehenden Kanten von s maximal ist.

Eine Intuition ist es sich alle Wege von s nach t vorzustellen, und dann über jeden Weg ein gewissen Fluss zu schicken, bis wir die Kapazitätsbeschränkung einer Kante erreichen und diesen Weg nicht mehr verwenden können. Das wäre dann ein Greedy Algorithmus.

Genauer definieren wir die Restkapazität einer Kante e als $r(e) = c(e) - f(e)$. Die Restkapazität eines Weges p ist dann $r(p) = \min_{e \in p} r(e)$. Startend mit $f(e) = 0$, für alle $e \in E$, solange es einen Weg mit Restkapazität grösser 0 gibt, schicken wir einen Fluss von $r(p)$ über diesen Weg.

Dieser Algorithmus ist im allgemeinen jedoch falsch! Wir möchten nun die Idee machen, dass wir den Fluss auch verringern können. Der Fluss kann jeweils um höchstens $r(e)$ erhöht werden und um höchstens $f(e)$ verringert werden.

Die Ford-Fulkerson Methode ist nun, dass wir jeweils auch die Rückwärtskanten betrachten, welche die Möglichkeit bieten, den Fluss zu verringern. Wir definieren das Restnetzwerk G_f als den gerichteten Graphen, welcher die gleichen Knoten wie G hat, und für jede Kante $e = (u, v)$ in G eine Vorwärtskante (u, v) mit Restkapazität $r(e)$ und eine Rückwärtskante (v, u) mit Restkapazität $f(e)$ hat.

Solange es einen Pfad von s nach t im Restnetzwerk gibt, können wir den Fluss erhöhen oder verringern, um einen grösseren Fluss von s nach t zu erhalten.

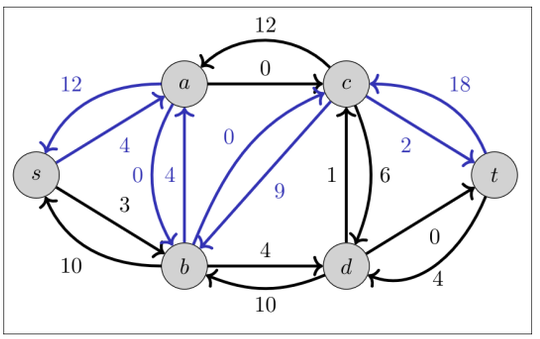


Figure 53: Ein Beispiel für ein Restnetzwerk.

Algorithm 20: Ford-Fulkerson Methode

Input: Flussnetzwerk $G = (V, E, c)$, Quelle s, Senke t
$f(e) \leftarrow 0$ für alle $e \in E$;
while *es gibt einen positiven Pfad p von s nach t im Restnetzwerk G_f* **do**
  $d \leftarrow \min_{e \in p} c_f(e)$;
  for $e \in p$ **do**
    if *e ist eine Vorwärtskante* **then**
      $f(e) \leftarrow f(e) + d$;
    end
    else
      $f(e) \leftarrow f(e) - d$;
    end
  end
end
return f;

Pro Iteraton müssen wir einen Erweiterungsweg in $\Theta(E + V)$ finden. Die Anzahl Iterationen ist unter Annahme, dass die Kapazitäten ganzzahlig sind, höchstens $|f_{\max}|$, da wir in jeder Iteration den Fluss um mindestens 1 erhöhen. Somit ist die Laufzeit dieses Algorithmus $\mathcal{O}(|f_{\max}| \cdot (|E|)$.

Eine Verbesserung liefert der Edmonds-Karp Algorithmus, wobei wir jeweils immer den kürzesten Erweiterungsweg verwenden. Es lässt sich zeigen, dass die Anzahl Iterationen in diesem Fall höchstens $\mathcal{O}(|V| \cdot |E|)$ ist.

Obere Schranken für die Flussgrösse sind was z.B. was aus s fliessen kann, oder was in t hineinfliessen kann. Im allgemeinen können wir einen beliebigen Schnitt betrachten, welcher s und t trennt, und die Summe der Kapazitäten der Kanten in diesem Schnitt als obere Schranke für die Flussgrösse verwenden.

Formal ist ein Schnitt eine Partition (S, T) von V, sodass $s \in S$ und $t \in T$ ist. Die Grösse des Schnittes ist definiert als die Summe der Kapazitäten der Kanten von S nach T:

$$c(S, T) = \sum_{e=(u,v) \in E, u \in S, v \in T} c(e).$$

Der Fluss durch einen Schnitt (S, T) ist definiert als

$$f(S, T) = \sum_{e = \in E(S,T)} f(e) - \sum_{e = \in E(T,S)} f(e).$$

Wir beobacthen, dass $|f| \leq f(S,T) \leq c(S,T)$ für alle Schnitte (S,T) gilt, da die Summe der Flüsse von S nach T maximal der Summe der Kapazitäten von S nach T entsprechen kann, und die Summe der Flüsse von T nach S minimal 0 ist. Somit gilt

$$|f_{\max}| \leq c_{\min}.$$

Nachdem wir den Edmund-Karp Algorithmus ausgeführt haben, können wir die Knoten in S als die Knoten definieren, welche von s aus im Restnetzwerk erreichbar sind, und die Knoten in T als die Knoten, welche von s aus im Restnetzwerk nicht erreichbar sind. Es lässt sich zeigen, dass (S,T) ein Schnitt ist, und dass $|f_{\max}| = f(S,T) = c(S,T)$ gilt. Somit haben wir gezeigt, dass der maximale Fluss gleich der minimalen Schnittkapazität ist.

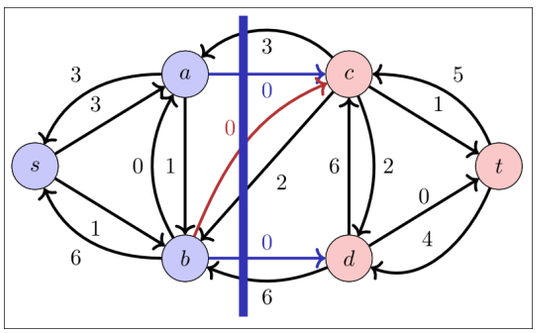


Figure 54: Ein Beispiel für einen Schnitt.

Theorem 23.1: Max-Flow Min-Cut Theorem

In einem Flussnetzwerk $G = (V, E, c)$ mit Quelle s und Senke t sind folgende Aussagen äquivalent:

- f ist ein maximaler Fluss von s nach t
- Das Restnetzwerk G_f enthält keinen Pfad von s nach t
- Es gibt einen Schnitt (S,T) von s und t, für welchen $|f| = c(S,T)$ gilt.

Eine interessante Anwendung ist das maximale Matching in bipartiten Graphen. Ein Bipartiter Graph ist ein Graph, dessen Knoten in zwei Mengen A und B aufgeteilt werden können, so dass alle Kanten von einem Knoten in A zu einem Knoten in B führen. Ein MATCHING ist eine Teilmenge von Kanten, so dass kein Knoten in mehr als einer Kante enthalten ist. Ein MAXIMALES MATCHING ist ein Matching mit maximaler Anzahl von Kanten.

Dafür kann man unseren Graphen erweitern indem wir einen Startknoten hinzufügen, welcher mit allen Knoten in A verbunden ist, und einen Zielknoten hinzufügen, welcher mit allen Knoten in B verbunden ist. Alle Kanten haben Kapazität 1.

24 Dynamische Programmierung I

Lec 20

Dynamische Programmierung ist im Grunde Rekursion mit Wiederverwendung.

Ein erstes Beispiel ist ein Frosch welcher über Seerosen springt, aber jedes Blatt hat eine Gefahr $C_i \geq 0$. Er kann vorwärts zum nächsten Feld gelangen was ihn nichts kostet, oder er kann über zwei Felder springen, was ihn 3 kostet. Gesucht ist ein Weg mit den geringsten Kosten. Das Problem ist, dass es exponentiell viele Wege gibt, und wir wollen nicht alle ausprobieren.

Die wichtigste Erkentniss ist: *Angenommen ich kann* die minimalen Kosten zum vorletzten und viertletzten Feld berechnen, *dann kann ich auch* die minimalen Kosten zum letzten Feld berechnen.

$$M(n) = C_n + \min(M(n-1), M(n-3)+3).$$

Genau diese Überlegung können wir jetzt auf jedes Feld anwenden.

$$M(i) = C_i + \begin{cases} \min(M(i-1), M(i-3)+3) & i \geq 3 \\ M(i-1) & 0 < i \leq 2 \\ 0 & i = 0 \end{cases}.$$

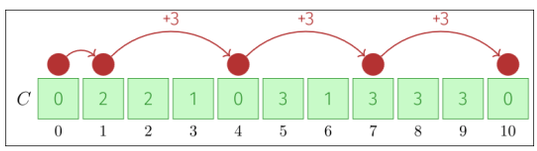


Figure 55: Die Kosten $M(i)$ für die ersten 10 Felder.

Dieser Algorithmus hat eine Laufzeit zwischen $\mathcal{O}(2^n)$ und $\Omega(\sqrt[3]{2}^n)$, da es exponentiell viele Wege gibt. Das Problem ist das wir die selben Werte mehrfach berechnen. Die Laufzeit lässt sich jedoch verbessern mit DYNAMISCHER PROGRAMMIERUNG. Dabei speichern wir uns die bereits berechneten Werte in einem Array, und wenn wir einen Wert benötigen, welcher bereits berechnet wurde, so können wir diesen Wert einfach zurückgeben, anstatt ihn neu zu berechnen. Dies wird MEMOIZATION genannt. Die Laufzeit dieses Algorithmus ist nun $\Theta(n)$. Ausserdem brauchen wir $\Theta(n)$ Speicherplatz. Jedoch brauchen wir ohnehin $\Theta(n)$ Speicherplatz, für die Verwaltung der Rekursion.

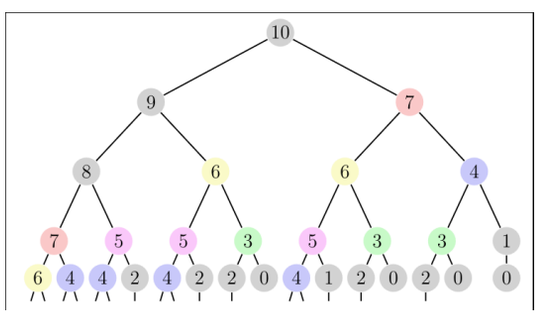


Figure 56: Der Rekursionsbaum für die Berechnung von $M(n)$.

Bei genauerem hinsehen, fällt auf, dass wir eigentlich die Werte von unten nach oben berechnen. Wir können also auch iterativ vorgehen, und die Werte von $M(0)$ bis $M(n)$

Tip 24.1:
In der Prüfung sollte man iterativ vorgehen, da so Segfaults umgangen werden.

Mit der DP Tabellen können wir auch den optimalen Pfad rekonstruieren. Dafür schauen wir ob $M_I = C_i + M_{i-1}$ oder $M_I = C_i + M_{i-3} + 3$ gilt, und je nachdem gehen wir dann zum vorherigen oder zum drittvorherigen Feld, und so weiter, bis wir am Anfang angekommen sind. Dieser Weg ist nicht notwendigerweise eindeutig.

Die Kernideen der dynamischen Programmierung sind also:

1. Top-Down Rekursion
2. Memoization
3. Bottom-Up-Algorithmen
4. Platz sparen (optional)
5. Rekonstruktion des optimalen Pfades (manchmal mit separater Tabelle)
6. Laufzeit: Anzahl Einträge · Operationen pro Eintrag

24.1 Schneiden von Eisenstäben

Gegeben sei ein Eisenstab der Länge n. Stäbe werden in ganzzahlige Stücke geschnitten. Für jede Länge $l \in \mathbb{N}$ ist der Wert $v_l \in \mathbb{R}^+$ eines Stabes der Länge l gegeben. Gesucht ist eine Schnittfolge, welche den Wert des Stabes maximiert. Auf der Suche nach Teilproblemen gehen wir

l	1	2	3	4	5	6
v_l	1	5	8	9	10	17

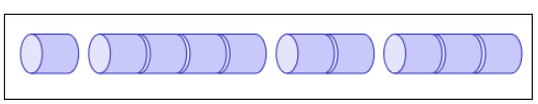


Figure 57: Ein zerschnittener Eisenstab.

systematisch vor. Für $n = 0 : r_0 := 0$. Für $n = 1 : r_1 = v_1 = 1$. Für $n = 2 : r_2 = \max(v_2, r_1 + r_1) = 5$. Für $n = 3 : r_3 = \max(v_3, r_2 + r_1, r_1 + r_2) = 8$. Wir beobachten, dass eigentlich die Entscheidung zwischen $v_1 + v_1$ und v_2 bereits gefallen ist. Im allgemeinen müssen wir für $n \geq 1$ folgende Rekursion lösen:

$$r_n = \max_{1 \leq i \leq n} (v_i + r_{n-i}).$$

Um dann die optimale Schnittfolge zu rekonstruieren, können wir uns zusätzlich eine Tabelle s merken, in welcher s_n die Länge des ersten Stückes der optimalen Schnittfolge für einen Stab der Länge n enthält. Diese Tabelle können wir während der Berechnung von r_n füllen, indem wir jedes Mal wenn wir ein neues Maximum finden, die entsprechende Länge in s_n speichern.

Somit ist die Berechnung von r_n und s_n in $\Theta(n^2)$ Zeit möglich, da wir für jedes n von 1 bis n über alle möglichen ersten Stücke iterieren müssen, um das Maximum zu finden. Die Rekonstruktion der optimalen Schnittfolge ist in $\Theta(n)$ Zeit möglich, da wir höchstens n Stücke schneiden können.

Der grosse Unterschied zwischen dynamic programming und divide and conquer ist, dass bei divide and conquer die Teilprobleme disjunkt sind, während bei dynamic programming die Teilprobleme überlappen können, und wir diese Überlappung ausnutzen, um die Laufzeit zu verbessern.

24.2 Matrixkettenmultiplikation

Wir wollen möglichst effizient das Produkt von n Matrizen $A_1, A_2, \ldots, A_n$ berechnen. Dazu relevant ist, dass Matrizenmultiplikation assoziativ ist. Wir nehmen an, dass die Multiplikation einer $r \times s$ Matrix mit einer $s \times u$ Matrix $\Theta(r \cdot s \cdot u)$ Zeit benötigt.

Bisher haben wir nur Präfix-Suffix Probleme betrachtet. Dies ist hier fundamental anders. Lec 21

Bei Matrixkettenmultiplikation, setzt sich die Lösung zusammen aus zwei Teilproblemen der linken und rechten Seite. Wenn wir nun die bestmögliche Berechnung von $(A_1 \cdot A_2 \cdot \cdots \cdot A_i)$ und $(A_{i+1} \cdot A_{i+2} \cdot \cdots \cdot A_n)$ kennen, so können wir die bestmögliche Berechnung von $(A_1 \cdot A_2 \cdot \cdots \cdot A_n)$ berechnen, indem wir das beste i bestimmen.

$$M[p,q] = \begin{cases} 0, \text{wenn } p = q \\ \min_{p \leq i < q}(M[p,i] + M[i+1,q] + r_p \cdot c_i \cdot c_q) \end{cases}$$

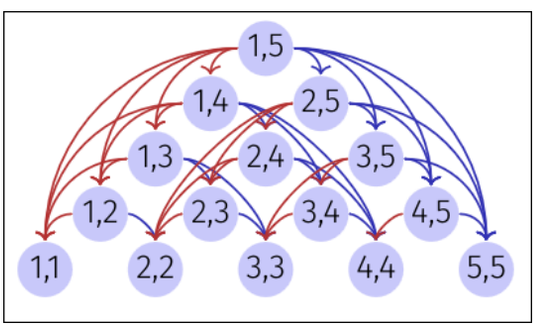


Figure 58: Rekursionsbaum für die Berechnung von $M[p,q]$.

Folglich werden wir die DP-Matrix von der Diagonalen ausgehend nach oben rechts füllen, da die Berechnung von $M[p,q]$ die Werte von $M[p,i]$ und $M[i+1,q]$ benötigt, welche beide unterhalb der Diagonalen liegen.

Ein naiver Algorithmus um Matrizen zu multiplizieren hat eine Laufzeit von $\Theta(n^3)$, da wir für jede der n^2 Einträge des Ergebnisses n Multiplikationen durchführen müssen.

Eine Standardstrategie dies zu verbessern ist divide and conquer. Dazu können wir die Matrix in 4 Blockmatrizen aufteilen, und diese dann multiplizieren gemäss

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}.$$

Unter der Annahme, dass $n = 2^k$ ist, sind die Anzahl elementarer Operationen $M(n) = 8M(\frac{n}{2})$, nach dem Master-Theorem ergibt dies eine Laufzeit von $\Theta(n^3)$, also keine Verbesserung gegenüber dem naiven Algorithmus.

Strassen konnte zeigen, dass es genügt lediglich sieben statt acht Blockmatrizen zu berechnen. Damit ergibt sich mit dem Master-Theorem eine Laufzeit von $\Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$. Der schnellste bekannte Algorithmus hat eine Laufzeit von $\mathcal{O}(n^{2.371})$. Dies ist ein sogenannter GALAKTISCHER ALGORITHMUS

24.3 Längste aufsteigende Teilfolge

Gegeben sei eine Folge von Zahlen $a_1, a_2, \ldots, a_n$. Gesucht ist die Länge der längsten aufsteigenden Teilfolge, also die maximale Länge einer Teilfolge $a_{i_1}, a_{i_2}, \ldots, a_{i_k}$, mit $i_1 < i_2 < \cdots < i_k$ und $a_{i_1} < a_{i_2} < \cdots < a_{i_k}$.

Wenn wir annehmen, für jedes $j \leq k$ ist die mit a_j endende LAT L_j bekannt. Dann können wir L_k berechnen, indem wir über alle $j < k$ iterieren, und das Maximum über alle L_j mit $a_j < a_k$ nehmen, und danach 1 addieren, um a_k zu der Teilfolge hinzuzufügen. Somit hätten wir eine $\Theta(n^2)$ Zeit Lösung, da wir für jedes k von 1 bis n über alle $j < k$ iterieren müssen.

Um dies effizienter zu gestalten, speichern wir nun eine Tabelle, welche jeweils an Index k die kleinste Zahl enthält, welche das Ende einer aufsteigenden Teilfolge der Länge k ist. Diese Tabelle können wir mit binärer Suche in $\Theta(\log n)$ Zeit aktualisieren, da sie sortiert ist und wir das kleinste Element suchen, welches grösser als das neue Element ist. Somit ergibt sich eine Laufzeit von $\Theta(n \log n)$.

Leider haben wir noch Speicherplatz von $\Theta(n^2)$. Wir können jedoch lediglich den Vorgänger jedes Elements in der längsten aufsteigenden Teilfolge speichern, um den Speicherplatz auf $\Theta(n)$ zu reduzieren.

24.4 Editierdistanz

Gegeben sind zwei Zeichenketten A_n, B_n. Wir haben die folgenden Operationen, um A in B zu transformieren:

- Einfügen eines Zeichens
- Löschen eines Zeichens
- Ändern eines Zeichens

Frage: Wie viele Editieroperationen benötigen wir mindestens, um A in B zu transformieren?

Wir wissen, dass $n + m$ eine Obere Schranke für die Editierdistanz ist, da wir jedes Zeichen von A löschen und jedes Zeichen von B einfügen können.

Die Laufzeiten der Operationen sind wie folgt: Einfügen: $\Theta(1)$, Löschen: $\Theta(1)$, Ändern: $\mathbb{1}(c \neq c')$.

Da wir Buchstabenweise Transformationen durchführen, zum beispiel von Ziege nach Tiger, gibt es jeweils 3 Möglichkeiten.

1. $$cost(Zieg \to TiGE) + repl(E, R).$$
2. $$cost(ZIEGE \to TIGE) + ins(R).$$
3. $$cost(ZIEG \to TIGER) + del(E).$$

Von diesen müssen wir jetzt jeweils das Minimum ziehen, um die Editierdistanz zu berechnen. Somit ergibt sich folgende Rekursion:

$$E(i,j) = \min \begin{cases} \text{del}(a_i) + E(i-1, j) \\ \text{ins}(b_j) + E(i, j-1) \\ \text{repl}(a_i, b_j) + E(i-1, j-1) \end{cases}.$$

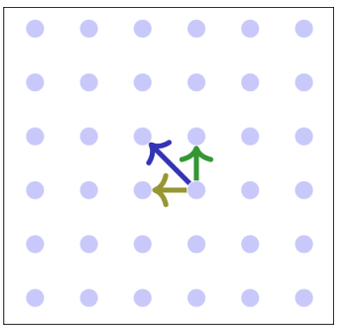


Figure 59: Rekursionsbaum für die Berechnung der Editierdistanz.

Wir sehen, dass wir die Berechnung von links oben nach rechts unten durchführen, Zeilen oder Spaltenweise. Den Editierweg können wir dann auch aus der Matrix rückverfolgen.

Die Laufzeit dieses Algorithmus ist $\Theta(n \cdot m)$, da wir eine $n \times m$ Matrix füllen müssen, und die Berechnung jedes Eintrags in konstanter Zeit möglich ist.

25 Dynamic Programming II

25.1 Subset Sum Problem

Gegeben ist eine Menge von n Gegenständen mit Werten $w_1, w_2, \ldots, w_n$, und ein Zielwert W. Gesucht ist eine Teilmenge von Gegenständen, welche genau den Wert W hat.

Ein naiver Algorithmus wäre es, einen Bitvektor $b = (b_1, \ldots, b_n) \in \{0,1\}^n$ zu verwenden, um die Teilmenge zu repräsentieren, und über alle möglichen Bitvektoren zu iterieren, um diejenige Teilmenge zu finden, welche den Wert W hat. Dieser Algorithmus hat eine Laufzeit von $\mathcal{O}(n \cdot \Theta(2^n))$.

Es geht ein wenig besser, wenn man das Problem in zwei gleich grosse Teilproblem aufteilt und alle Teilsummen beider Hälten berechnet. Nun sucht man mit zwei Zeigern in den beiden Teilsummen nach einem Paar von Teilsummen, welche zusammen W ergeben. Die Laufzeit kann somit zu $\mathcal{O}(n \cdot \sqrt{(2)^n})$ verbessert werden.

Wir können eine DP-Tabelle erstellen der grösse $n \times W$, in welcher $M[i, w]$ angibt, ob es mit ob es mit den ersten i Gegenständen möglich ist, eine Teilmenge mit Wert w zu bilden.

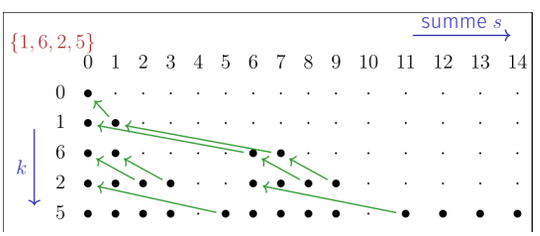


Figure 60: Rekursionsbaum für die Berechnung von $M[i, w]$.

Lec 22

Die Laufzeit dieses Algorithmus ist in der Grösse $n \cdot z$ für eine fixe Zahl z. Der Algorithmus hat jedoch keine polynomielle Laufzeit, da z nicht in der Eingabe kodiert ist, sondern als Parameter übergeben wird. Wir bezeichnen solch eine Laufzeit als PSEUDO-POLYNOMIELLE LAUFZEIT.

26 Dynamic Programming III

Gegeben sind n Schlüssel $k_1 < k_2 < \ldots < k_n$, und n Wahrscheinlichkeiten $p_1, p_2, \ldots, p_n$, wobei p_i die Wahrscheinlichkeit ist, dass der Schlüssel k_i gesucht wird. Wir wollen nun einen binären Suchbaum konstruieren, welcher die Schlüssel enthält, und die erwartete Suchzeit minimal ist.

$$C(T) = \sum_{i=1}^{n} (d(k_i) + 1) \cdot p_i.$$

Da wir einen binären Suchbaum haben, müssen wir die Wurzel finden. Dazu suchen wir die besten Teilbäume in der linken und rechten Halte. Dies ist im Grunde das selbe Problem wie bei der Matrix-Kettenmultiplikation, da wir auch hier die optimale Aufteilung suchen.

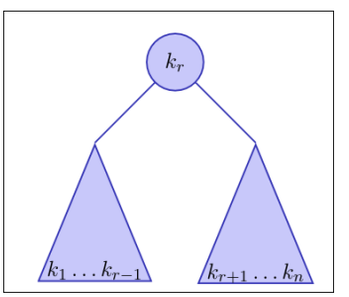


Figure 61: Teilsuchbäume für die Berechnung der Kosten eines Suchbaums.

27 Greedy Algorithmen

Ein rekursive lösbares Problem kann mit Greedy gelöst werden, wenn das Problem eine optimale Substruktur hat, und die Greedy Choice Property erfüllt. Dies bedeutet, die Lösung eines Problems kann durch die Wahl einer lokalen optimalen Lösung konstruiert werden und unabhängig der Teilprobleme ist.

27.1 Huffman-Codierung

Ziel ist es, eine Folge von Zeichen mit einem effizienten binären Zeichencode zu kodieren. Nehmen wir als Beispiel eine Datei aus 100000 Buchstaben von a bis f. Die Häufigkeit der Buchstaben ist wie folgt:

a	b	c	d	e	f
45000	13000	12000	16000	9000	5000

Dazu kann man ganz einfacherweise eine fixe Länge von 3 Bits pro Buchstabe verwenden, da wir 6 Buchstaben haben, und $2^3 = 8$ Möglichkeiten. Die Länge wäre dann 300'000 Bits. Das kann man jedoch verbessern.

Dazu betrachten wir Präfixcodes. Kein Codewort ist Präfix eines anderen Codeworts. Wenn wir uns die Binären Codes vorstellen, so muss jedes Zeichen ein Blatt des Baumes sein. Mit Präfixcodes kann die optimale Datenkompression erreicht werden.

Eigenschaften so eines Codebaumes sind das jeder innere Knoten immer zwei Kinder hat. Die zu minimierende Summe kann man schreiben als

$$\sum_{w \in W} f(w) \cdot d_T(w) = -\sum_{w \in W} f(w) \cdot \log_2 g_T(w).$$

Wobei $g_T(\cdot) := 2^{-d_T(\cdot)}$ die Wahrscheinlichkeit ist, dass ein Blatt w erreicht wird, wenn wir von der Wurzel aus zufällig einen Pfad wählen, und $f(w)$ die Häufigkeit von w ist. Dies ist nun eine Wahrscheinlichkeitsverteilung weshalb wir die Gibbs'sche Ungleichung anwenden können, welche besagt, dass die Summe minimal ist, wenn $g_T(w) = f(w)$ für alle $w \in W$ gilt. Das bedeutet, dass die optimale Kodierung erreicht wird, wenn die Wahrscheinlichkeit eines Blattes gleich der Häufigkeit des entsprechenden Zeichens ist.

Bei Wahrscheinlichkeiten, welche von 2-er Potenzen abweichen, ist es möglich zu approximieren (Shanon-Fano-Codierung), oder die optimalen Blätter zu kombinieren, um die nächsten Blätter zu erhalten, bis wir nur noch einen Baum haben (Huffman-Codierung).

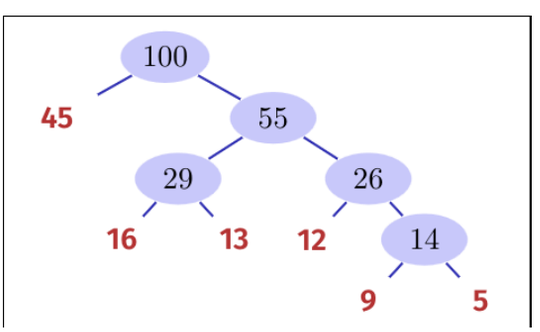


Figure 62: Shanon-Fano-Codierung.

Lec 23

Dabei schreiben wir die Blätter in einem Min-Heap und ersetzen iterativ die beiden knoten mit kleinster Häufigkeit durch einen neuen Knoten, welcher die Summe der Häufigkeiten der beiden Knoten als Häufigkeit hat, und die beiden Knoten als Kinder hat. Dies macht man so lange, bis man nur noch einen einzelnen Knoten hat, welcher die Wurzel des Baumes ist. Die Laufzeit dieses Algorithmus ist $\Theta(n \log n)$, da wir n Blätter in den Min-Heap einfügen müssen, und danach $n-1$ mal die beiden kleinsten Knoten extrahieren und einen neuen Knoten einfügen müssen.

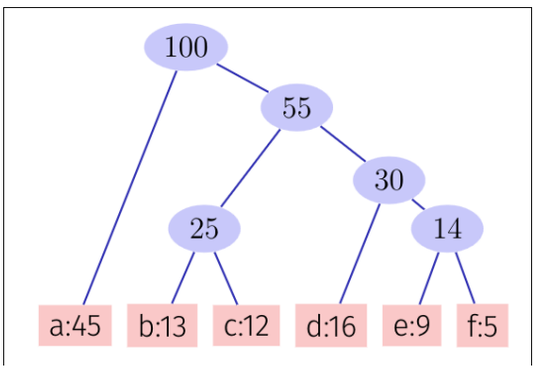


Figure 63: Huffman-Codierung.

28 Parallel Programming

Stellen wir uns vor, wir haben einen Maler, welcher ein Bild malen soll. Sagen wir, dass er das Bild in n Teile aufteilt, und jeden Tag einen Teil davon malt. Das Modell ist eine SEQUENZIELLE AUSFÜHRUNG. Wenn wir nun zum Beispiel 4 Maler haben, so könnten wir das Bild in 4 Teile aufteilen, und jeder Maler malt einen Teil davon. Das Modell ist eine PARALLELE AUSFÜHRUNG.

Es könnte sein, dass wir beschränkte Ressourcen haben, zum Beispiel nur 3 Pinsel. Das zugehörige Modell ist die NEBENLÄUFIGE AUSFÜHRUNG (Concurrency).

Wenn wir uns Mikroprozessoren ansehen, so gilt Moore's Law, welches besagt, dass sich die Anzahl Transistoren auf einem Mikroprozessor alle 2 Jahre verdoppelt. Die Single Thread Performance hat sich jedoch nicht so stark verbessert, da die Taktfrequenz nicht so stark erhöht werden konnte, da sie durch die Wärmeentwicklung begrenzt ist. Aus diesem Grund haben die Hersteller von Mikroprozessoren begonnen, mehrere Kerne auf einem Mikroprozessor zu integrieren, um die Leistung zu verbessern.

In C++ wird ein Thread wie folgt implementiert

```cpp
#include <thread>
#include <iostream>

void Hello(id){
    std::cout << "Hello from thread " <<
    ↪  std::this_thread::get_id() << std::endl;
}

int main(){
    std::thread t1(Hello, 1);
    std::thread t2(Hello, 2);


    std::cout << "Hello from main" << std::endl;

    t1.join();
    t2.join();
    return 0;
}
```

Wichtig zu bemerken ist, dass Threads nicht eins nach dem anderen ausgeführt werden, sondern dem Modell entsprechend parallel. Die Ausführungsreihenfolge der Threads ist NICHT deterministisch.

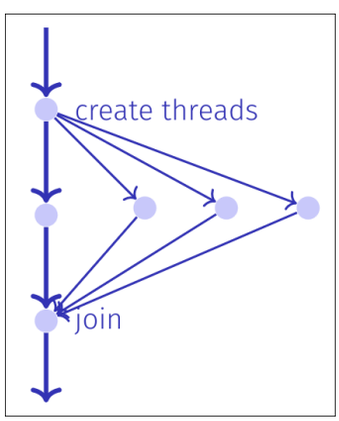


Figure 64: Ausführungsdiagramm für die Threads.

Obige Implementation ist ein Fork-Join Modell. Wenn wir nicht auf den Thread warten wollen, so können wir in als Hintergrundthread laufen lassen, indem wir ihn detachten.

```cpp
#include <thread>
t = std::thread(Hello, 1);
t.detach();
```

Um ein Programm zu parallelisieren, müssen wir es in unabhängige Teilprobleme aufteilen können.

Es kann jeweils nur einen Thread-Vektor geben. Um Daten anzuhängen kann man threads.emplace back verwenden

```cpp
#include <vector>
std::vector<std::thread> threads;
for(int i = 0; i < n; i++){
    threads.emplace_back(
        [=]{
            Hello(i);
        });
}
```

Parallelisierung kann die Laufzeit reduzieren, wenn die parallelisierte Berechnung lange genug dauert und genügend CPUs vorhanden sind.

Ein klassisches Prinzip ist es die Arbeit in Blöcke zu unterteilen, und dann jeden Block einem Thread zuzuweisen. Es ist wichtig, dass die Blöcke groß genug sind, damit die Kosten für die Erstellung der Threads und die Kommunikation zwischen den Threads nicht zu hoch werden.

Es stellt sich nun die Frage nach Skalierbarkeit. Sei T_1 die Sequenzielle Ausführungszeit auf einer CPU, und T_p die parallele Ausführungszeit auf p CPUs. Der Speedup is $S_p = \frac{T_1}{T_p}$. Idealerweise wäre $S_p = p$, da wir die Arbeit perfekt auf p CPUs aufteilen könnten. In der Praxis ist jedoch $S_p < p$. Letztlich ist $S_p > p$ als Hexerei bezeichnet, was zum Beispiel durch Cache-Effekte erreicht werden kann.

In einem durchschnittlichen Programm gibt es einen Parallelen Teil und einen Sequenziellen Teil. Sagen wir 80% des Programms kann parallelisiert werden, und 20% muss sequenziell ausgeführt werden. Wenn $T_1 = 10$ ist, folgt

$$T_8 = 0.2 \cdot 10 + \frac{0.8 \cdot 10}{8} = 3.$$

Die Geschwindigkeitssteigerung ist somit $S_8 = \frac{10}{3} \approx 3.3 < 8$.

Bei AMDAHL'S LAW handelt es sich um eine Formel, welche die maximale Geschwindigkeitssteigerung eines Programms durch Parallelisierung beschreibt. Sie besagt, dass die maximale Geschwindigkeitssteigerung S_p durch die Formel

$$S_p = \frac{T_1}{T_p} \leq \frac{W_s + W_p}{W_s + \frac{W_p}{p}}.$$

Hierbei ist W_s die Zeit, welche für den sequenziellen Teil des Programms benötigt wird, und W_p die Zeit, welche für den parallelen Teil des Programms benötigt wird. Dieses Gesetz ist eigentlich keine gute Nachricht, da alle nicht parallelisierbaren Teile eines Programms die maximale Geschwindigkeitssteigerung begrenzen.

Gustafson's Law ist eine alternative Formel. Gegeben ist die Arbeit die mit einem Prozessor in der Zeit T erledigt werden kann, und die Zeit T die mit p Prozessoren benötigt wird, um die gleiche Arbeit zu erledigen. Dann ist die Geschwindigkeitssteigerung

$$S_p = \frac{T_1}{T_p} = \frac{W_s + p \cdot W_p}{W_s + W_p} = p \cdot (1 - \lambda) + \lambda.$$

Ein wichtiger Unterschied ist, dass bei Amdahl der relative Anteil der sequenziellen Arbeit konstant ist, während bei Gustafson die absolute Zeit der sequenziellen Arbeit konstant ist.

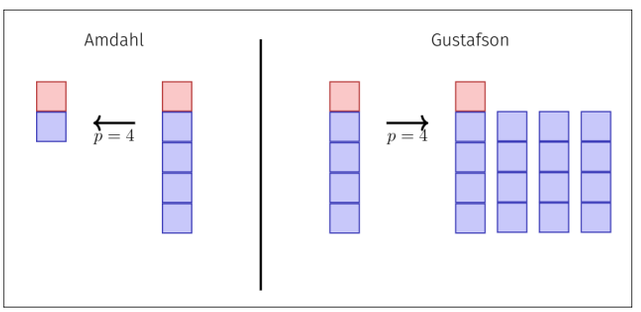


Figure 65: Geschwindigkeitssteigerung nach Amdahl's Law und Gustafson's Law.

Index