Algorithmen und Programmierung

10. Kapitel
Textsuche

Prof. Matthias Werner
Professur Betriebssysteme

10.1 Einführung

- Betrachten Algorithmen zum Finden das erste Vorkommen einer bestimmten Zeichenkette (Muster) in einem längeren Text
- **Beispiel:** Gesucht wird „*example*“

Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip example ea commodo consequat. Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero et accumsan et iusto odio dignissim qui blandit praesent luptatum zzril delenit augue duis dolore te feugait nulla facilisi. Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt ut laoreet dolore magna aliquam erat volutpat. Ut wisi enim ad minim veniam, quis nostrud exerci tation ullamcorper suscipit lobortis nisl ut aliquip ex ea commodo consequat.

10.2 Dateien

- Große Datenmengen (wie z.B. längere Texte) werden häufig nicht interaktiv eingegeben, sondern in **Dateien** gespeichert
- Beschäftigen uns daher zunächst mit Dateien

Definition 10.1 (Datei)

*Eine **Datei** (file) ist eine Zusammenstellung von **logisch zusammengehörigen** Daten, die als eine Einheit behandelt werden.*

*Auf sie wird meist unter einem dem Betriebssystem bekannten **Dateinamen** (file name) zugegriffen. Dateien werden häufig auf **persistenen** Dateinträgern (z.B. Festplatten) gespeichert.*

Dateien in C

- Dateien können unterschiedlich organisiert sein
- Im klassischen UNIX ist eine Datei stets ein einzelner, beliebig langer Datensatz von Bytes
- Dieser Datensatz wird als **Bytestrom** aufgefasst
  - Dadurch braucht nicht unterschieden werden, ob eine Eingabe aus einer Datei oder von einem Eingabegerät kommt
- Die Ein-/Ausgabe der C-Standardbibliothek ist diesem Konzept angepasst

Dateien in C (Forts.)

- Die Standardbibliothek von C kennt zwei Klassen von Fileoperationen
  - **Low-Level-Fileoperationen**
    - ➡ Dateien/Datenströme werden über ein **Handle** identifiziert
    - ➡ spezifisch für jeweiliges Betriebssystem
  - **High-Level-Fileoperationen**
    - ➡ Dateien/Datenströme werden über einen **File-Pointer** identifiziert
    - ➡ unabhängig vom Betriebssystem
- Betrachten hier High-Level-Operationen

Dateien in C (Forts.)

- Dateien müssen **geöffnet** werden
  - Dabei werden die Verwaltungsinformationen angelegt
  - Funktion der Standardbibliothek: `FILE* fopen(char* name,char* mode)`
  - Gibt einen Zeiger auf Dateiverwaltungsstruktur und im Fehlerfall NULL zurück
  - **Beispiel**:

```
#include<stdio.h>
/* ... */
FILE *fp;
fp = fopen("myfile.dat","r");
```

- Wenn man die Datei nicht mehr braucht, sollte diese **geschlossen** werden
  - Freigabe der Verwaltungsressourcen
  - Bibliotheksfunktion `int fclose(FILE* stream)`
  - Gibt `0` bei fehlerfreier Ausführung zurück

Modi

Mode	Bedeutung
`r`	Öffnen einer Datei ausschließlich zum **Lesen**
`w`	Datei zum **Schreiben** erzeugen; existiert die Datei bereits, so wird sie **überschrieben**
`a`	Öffnen einer Datei zum **Anfügen**; existiert die Datei bereits, so werden neue Daten an das Dateiende angefügt, ansonsten wird die Datei neu erzeugt
`r+`	Öffnen einer Datei zum **Schreiben und Lesen**; die Datei **muss bereits existieren**
`w+`	Datei zum **Schreiben und Lesen** erzeugen; existiert die Datei bereits, so wird sie überschrieben
`a+`	Öffnen einer Datei zum **Lesen und Schreiben**; existiert die Datei bereits, so werden neue Daten an das Ende angehängt, ansonsten wird die Datei neu erzeugt
`b`	*zusätzlich:* Öffnet Datei im **Binär-Modus** (sonst: Text-Modus)

Dateiein-/ausgabe

- Die Funktionen zur Ein- und Ausgabe von/in Dateien kann nach Interpretation der Daten unterschieden werden:
  - formatiert
  - einzelne Zeichen
  - Zeichenketten (Strings)
  - Binärdaten
- Wesentliche Funktionen sind:

	Eingabe	Ausgabe
formatiert	`int fscanf(FILE*, char*, ...)`	`int fprintf(FILE*, char*, ...)`
Zeichen	`int fgetc(FILE*)`	`int fputc(int, FILE*)`
Strings	`char* fgets(char*, int, FILE*)`	`int fputs(char*, FILE*)`
binär	`size_t fread(void*, size_t, size_t, FILE*)`	`size_t fwrite(void*, size_t, size_t, FILE*)`

Formatierte Dateiein-/ausgabe

- `fscanf()` und `fprintf()` funktionieren genauso wie `scanf()` und `printf()`, nur muss zusätzlich als erster Parameter ein Filepointer angegeben werden
  - Für `fscanf()` muss die Datei (mindestens auch) im Lesemodus geöffnet sein ➡ „r“, „r+“, „w+“, „a+“
  - Für `fprintf()` muss die Datei (mindestens auch) im Schreibmodus geöffnet sein ➡ „w“, „a“, „r+“, „w+“, „a+“

```c
/* fprintf.c -- example for formated file output */
#include<stdio.h>

int main()
{
  FILE* file;
  int n=42;
  file = fopen("out.txt","w");
  fprintf(file,"Hello world! The answer is %d\n",n);
  fclose(file);
  return 0;
}
```


Zeichenweise Dateiein-/ausgabe

- Die Funktion

  `int fgetc(FILE* stream)`

  gibt das nächste Zeichen einer Datei `stream` **als Integer**
- Wenn kein Zeichen (mehr) vorhanden ist, wird die Konstante `EOF` (definiert in `stdio.h`) zurückgegeben
- Gleiches gilt im Fehlerfall

- Die Funktion

  `int fputc(int c, FILE* stream)`

  schreibt das (ganzzahlcodierte) Zeichen `c` in die Datei `stream`
- Es wird die Anzahl der geschriebenen Zeichen (1) zurückgegeben
- Im Fehlerfall ist der Rückgabewert `EOF` (in `stdio.h` definiert)

Zeichenkettenein-/ausgabe aus/in Dateien

- Die Funktion

  `char* fgets(char restrict * str, int n, FILE* restrict stream)`

  liest maximal `n`-1 Zeichen aus der Datei `stream` in ein Zeichenarray, auf das `str` zeigt
- Das Lesen wird bei einem Zeilen- oder Dateiende oder bei einem Fehler beendet
- Wenn kein Fehler auftrat, wird „`\0`“ an `str` angehangen
- Der Rückgabewert zeigt im Erfolgsfall auf `str`, sonst wird `NULL` zurück gegeben

Achtung

Es liegt in der Verantwortung des Programmierers, dass `str` auf ein Zeichenarray zeigt, dass (mindestens) `n` Zeichen groß ist!

Zeichenkettenein-/ausgabe aus/in Dateien (Forts.)

- Die Funktion

  `int fputs(char* str, FILE* stream)`

  gibt den (nullterminierten) String `str` in die Datei `stream` aus
- Funktion gibt im Erfolgsfall einen nichtnegativen Wert zurück, im Fehlerfall `EOF`
- **Achtung:** Alte Versionen gaben im Erfolgsfall immer 0 zurück

Binäre Dateiein-/ausgabe

- Die Funktion

  ```
  size_t fread(void* ptr, size_t size, size_t nitems, FILE* stream)
  ```

  liest `nitems` Einträge der jeweiligen Größe `size` aus der Datei `stream` und speichert sie unter der von `ptr` angegebenen Adresse
- Es wird die Anzahl der erfolgreich gelesenen Einträge (nicht der Bytes!) zurückgegeben

- Die Funktion

  ```
  size_t fwrite(void* ptr, size_t size, size_t nitems, FILE* stream)
  ```

  schreibt `nitems` ab der Adresse `ptr` angegebenen Einträge der jeweiligen Größe `size` in die Datei `stream`
- Es wird die Anzahl der erfolgreich geschriebenen Einträge (nicht der Bytes!) zurückgegeben

Standarddatenströme

- Beim Start eines C-Programmes werden automatisch drei Datenströme geöffnet
- Programmierer hat darauf Zugriff über in `stdio.h` Variablen von Typ `FILE*`

Variable	Funktion	Voreingestelltes Ziel
`stdin`	Standardeingabe	Tastatur
`stdout`	Standardausgabe	Bildschirm
`stderr`	Standardfehlerausgabe	Bildschirm

- `stdin`, `stdout` und `stderr` brauchen (und können) nicht geöffnet werden
- Die Unterscheidung von `stdout` und `stderr` kann zur Trennung von Nutz- und Statusdaten genutzt werden ➡ **Umlenkung** auf Kommandozeilenebene

Standarddatenströme (Forts.)

```c
/* fstd.c -- standard out vs. standard error */
#include<stdio.h>

int main()
{
  fprintf(stdout,"Dies sind Nutzdaten.\n");
  fprintf(stderr,"Dies ist eine Statusmeldung.\n");
  return 0;
}
```

```
> ./fstd
Dies sind Nutzdaten.
Dies ist eine Statusmeldung.
> ./fstd 2> /dev/null
Dies sind Nutzdaten.
> ./fstd > /dev/null
Dies ist eine Statusmeldung.
```


Manipulation des Positionszeigers

- Allgemein werden Dateien als Datenströme aufgefasst, auf die **sequentiell** zugegriffen wird
- Bei „echten“ Dateien kann von der sequentiellen Zugriffsweise abgewichen werden
- Dazu stehen folgende Funktionen zur Verfügung:
  - `void rewind(FILE* stream)`
    Setzt die Lese- oder Schreibposition der Datei `stream` wieder auf den Dateianfang
  - `int fseek(FILE* stream, long offset, int whence)`
    Setzt die Lese- oder Schreibposition der Datei `stream` eine Position, die `offset` Bytes von `whence` entfernt ist
  - `whence` ist eine von drei in `stdio.h` definierten Konstanten:
    `SEEK_SET`: `offset` ist relativ zum Dateianfang
    `SEEK_CUR`: `offset` ist relativ zur aktuellen Position
    `SEEK_END`: `offset` ist relativ zum Dateiende
- Mit Hilfe von `long ftell(FILE* stream)` kann die aktuelle Position (relativ zum Dateianfang) abgefragt werden

Beispiel

```c
/* fwrite.c --  write an array to a binary file */
#include <stdio.h>

int main () {
  float array[]= {23.0f, 3.1415f, 42.0f, 47.11f};
  const char* const filename="example.bin";
  FILE* file;
  if ((file = fopen(filename,"w")) == NULL) {
    fprintf(stderr,"error: can't open file\n");
    return 1;
  }
  if (fwrite(array, sizeof(array),1,file) == 1){
    fprintf(stderr,"success.\n");
    return 0;
  } else {
    fprintf(stderr,"write error.\n");
    return 1;
  }
}
```


Beispiel (Forts.)

```c
/* fread -- read an array from a binary file */
#include <stdio.h>

int main () {
  float array[4];
  const char* const filename="example.bin";
  FILE* file;

  if ((file = fopen(filename,"r")) == NULL) {
    fprintf(stderr,"error: can't open file\n");
    return 1;
  }
  if (fread(array, sizeof(array),1,file) == 1){
    fprintf(stderr,"success\n");
    for(int i=0; i<4; ++i) {
      fprintf(stdout,"%f%s",array[i], (i==3)? "\n": ", ");
    };
    return 0;
  } else {
    fprintf(stderr,"read error.\n");
    return 1;
  }
}
```


Beispiel (Forts.)

```
> ./fwrite
success
> cat example.bin
?AVI@(B?p<B
>
> ./fread
success
23.000000, 3.141500, 42.000000, 47.110001
```


Weitere Funktionen

- Die Standardbibliothek noch andere Funktionen zur Verfügung
- Interessant sind u.a.:
  - `int feof(FILE* stream)`
    gibt bei Dateiende Wert ungleich 0 zurück
  - `int ferror(FILE* stream)`
    gibt einen Wert ungleich 0 zurück, wenn vorher ein Dateifehler aufgetreten war
  - `int flush(FILE* stream)`
    Erzwingt das physische Schreiben (Cache wird geleert)
  - `int remove(char* name)`
    Löscht die Datei mit dem Namen `name`

Dateien in Python

- Eine Datei ist ein **Datentyp** in Python
- Eine Dateivariable wird mit

  `f=open(filename[, mode[, bufsize]])`

  angelegt
  - Modi sind eine Obermenge der C-Modi
  - Mit `bufsize` die Cache-Größe für die Datei gesetzt werden
- Verschiedene Dateioperationen stehen zur Verfügung, z.B.:

Operation	Bedeutung
`S=file.read()`	Liest gesamte Datei in einen einzigen String
`S=file.read(N)`	Liest `N` Bytes
`S=file.readline()`	Liest nächste Zeile (bis Zeilenendmarker)
`L=file.readlines()`	Liest gesamte Datei als Liste von Zeilen-Stings
`file.write(S)`	Schreibt String `S` in Datei
`file.writelines(L)`	Schreibt alle Strings einer Liste `L` in Datei
`file.close()`	Schliesst Datei

Dateien in Python (Forts.)

- Durch den Iterationsansatz (vgl. Kapitel 4) kann einfach eine komplette Datei zeilenweise bearbeitet werden

```
f= open("foo.txt","r")
for line in f:
  print(line, end=' ')
```

- Die Python-Bibliothek stellt verschiedene Module für Filemanipulation auf verschiedenen Abstraktionsebenen zur Verfügung, z.B.:
  - Low-Level-Dateien ➡ Modul `os`
  - High-Level-Speicherung komplexer Objekte ➡ Module `shelve` und `pickle`
  - Datenbank-Schnittstellen ➡ Module `dbm` und `anydbm`

10.3 Einfache Suche

- Mit den Kenntnissen über Dateien können wir nun das Textsuchprogramm beginnen
- Das Programm soll folgende Aufrufparameter haben:

```
./search <Suchtext> <Dateiname>
```

- `Suchtext` ist der Text, nach dem gesucht wird
- `Dateiname` ist die Datei, in der gesucht wird
- Wenn der Suchtext gefunden wird, soll die „Umgebung“ ausgegeben werden, in der er vorkommt
- Dabei soll der Suchtext durch eckige Klammern hervorgehoben werden

```
./search "example" lorem.txt
ullamcorper suscipit lobortis nisl ut aliquip [example] ea commodo
```


Größe

- **Problem**: Es ist nicht bekannt, wie lang maximal eine Zeile oder die gesamte Datei ist
- **Lösung 1**: Einen Zeilenpuffer definieren, der „hinreichend“ groß für alle Fälle ist ➡ nicht sicher
- **Lösung 2**: Nicht immer ganze Zeilen einlesen ➡ erschwert Suche, wenn der Suchtext zwischen zwei Einleseblöcken liegt
- **Lösung 3:** Dateigröße ermitteln, dynamisch Platz reservieren und die gesamte Datei einlesen ➡ braucht evtl. sehr viel Speicher
- Wählen Lösung 3
  - ➡ Bietet Geschwindigkeitsvorteile

Größe (Forts.)

- Wie Filegröße ermitteln?
- Unter UNIX gibt es die Funktion

  `int stat(char* name, struct stat* buf)`

  die verschiedene Informationen über Datei `name` nach `buf` schreibt
  - Größe = `buf.st_size` (in Bytes)
  - Nicht kompatibel
- **Alternative**: Trickreiche Nutzung von Standardfunktionen

```
size_t filesize(FILE* file)
{
  size_t ret;
  fseek(file,0L,SEEK_END);
  ret=ftell(file);
  rewind(file);
  return ret;
}
```


main-Funktion

```
6  int main(int argc, char* argv[]){
7    FILE* file;
8    char* text;
9
10   if (argc!=3) {
11     return -2; // wrong number of parameters
12   }
13   if ((file=fopen(argv[2],"r"))==NULL) {
14     return -3; // can't open file
15   }
16   size_t size = filesize(file);
17   if ((text= malloc(size+1))==NULL) {
18     return -4; // out of memory
19   }
20   if (fread(text,size,1,file)!=1){
21     return -5; // can't read file
22   }
23   text[size]='\0';
24   int found=search(argv[1],text,size); // yet to implement
25   if (found != -1)
26     presentresult(found,text,argv[1]); // dito
27   free(text);
28   return found;
29 }
```


Ausgabe

- Die `main()`-Funktion besteht im Wesentlichen aus Fehlerprüfungen
- Außer der eigentlichen Suche brauchen wir noch eine Ausgabe
- Geben 20 Zeichen vor und nach Suchmuster aus

```
void presentresult(int pos, const char* str, const char* pattern)
{
  int start, end, patlen, prelen;
  start=pos>20? pos-20 : 0;
  prelen=pos>20? 20 : pos;
  patlen=length(pattern);
  end=pos+patlen;
  printf("%.*s[%s]%.20s\n",prelen,&str[start],pattern,&str[end]);
}
```


Algorithmus

- Jetzt kann der eigentliche Algorithmus betrachtet werden
- **Idee**: Teste für jede Position des Textes `str`, ob der Suchstring `p` dort beginnt
  - Wenn ja, teste nächstes Zeichen etc.
- **Beispiel**: `str="Sein oder nicht sein"`, `p="er"`

```
sein oder nicht sein
er                  ✗
 er                 ✔✗
  er                ✗
   er               ✗
    er              ✗
     er             ✗
      er            ✗
       er           ✔✔ ➡ gefunden
```

Algorithmus (Forts.)

Algorithm SIMPLE-SEARCH

Require: str, p is text; $length(str) > length(p)$
Ensure: returns index of first appearance of p in str

procedure SIMPLE-SEARCH(str, p)
  $pos \leftarrow 1$ ▷ first element
  while $pos < length(str) - length(p)$ **do**
    $j \leftarrow 1$
    while $(j \leq length(p)) \wedge (str[pos + j - 1] = p[j])$ **do**
      if $j = length(p)$ **then** ▷ all characters matched
        return pos
      end if
      $j \leftarrow j + 1$
    end while
    $pos \leftarrow pos + 1$
  end while
  return „not found“
end procedure

C-Implementierung

- Für den Algorithmus müssen wir die Länge eines Strings ermitteln
- Dafür gibt es eine Funktion in der Standardbibliothek
- Nutzen aber eigene Funktion:

```c
int length(const char* str)
{
  int len=0;
  if (str==NULL) return 0;
  while(str[len]!='\0') ++len;
  return len;
}
```


C-Implementierung (Forts.)

- Der Algorithmus lässt sich relativ ähnlich zum Pseudocode in C implementieren
- Zu beachten:
  - Vorbedingungen sollten überprüft werden
  - Indizes beginnen bei 0

```c
int search(const char* p, const char* str, size_t tlen)
{
  size_t plen=length(p);
  if (plen>tlen) return -1;
  for(size_t pos=0; pos<tlen; ++pos){
    size_t j=0;
    while((j<plen) && (str[pos+j]==p[j])){
      if (j==plen-1) return pos;
      ++j;
    }
  }
  return -1;
}
```


Bewertung des Algorithmus

- Wir haben zwei verschachtelte Schleifen
- Die erste durchläuft die Anzahl n der Elemente der Zeichenkette `str`
- Die zweite durchläuft die Anzahl m der Elemente des Suchmusters `p`
- ➡ Haben demzufolge eine Aufwandsobergrenze von $\mathcal{O}(m \cdot n)$
- Aufwandsuntergrenze liegt bei $\mathcal{O}(m)$
- Gibt es ein effizienteres Verfahren?

Boyer-Moore-Algorithmus

- Algorithmus von Robert S. Boyer und J. Strother Moore 🕮[BM77]
- **Idee**:
  - Algorithmus vergleicht zunächst die **letzte** Stelle des Suchmusters `p` und verschiebt bei fehlender Übereinstimmung `p` soweit nach hinten, bis die letzte Stelle wieder passt
  - Dabei wird in Abhängigkeit des Inhalts von `p` oft mit einem Schritt eine Verschiebung von `p` um mehrere Stellen erreicht
- **Beispiel**

```
sein oder nicht sein
er                    'r'≠'e', aber 'e'=p[1]   ➡ Verschieben um 1
 er                   'r'≠'i', und 'i'∉ p      ➡ Verschieben um 2
   er                 'r'≠' ', und ' '∉ p      ➡ Verschieben um 2
     er               'r'≠'d', und 'd'∉ p      ➡ Verschieben um 2
       er                                       ➡ gefunden
```


Boyer-Moore-Algorithmus (Forts.)

- **Berechnung der Verschiebung**
  - Entscheidend für die Verschiebung ist, wie weit sich das betrachtete Zeichen aus `str` vom **Ende** des Musters `p` befindet
    - Ist es nicht in `p` enthalten, können wir das komplette Suchmuster nach hinten verschieben
    - Ansonsten verschieben wir es um die Anzahl Stellen, die die kürzeste Entfernung des Zeichens zum Ende des Suchmusters ausmacht
  - ➡ Zur Realisierung des Algorithmus brauchen wir also die Information, wie groß für alle in `p` vorkommenden Zeichen ihre **Distanz** zum Ende von `p` ist
  - Für alle anderen (nicht in `p` vorkommenden) Zeichen entspricht diese Distanz der Länge von `p`
  - Letztes Zeichen im Suchmuster (Distanz=0) muss nicht erfasst werden
  - Algorithmus ist besser für lange `p`

Aufbau der Distanztabelle

- Solange wir im ASCII/ANSI-Zeichensatz bleiben, können wir einfach eine Tabelle nutzen
- **Idee**: Tabelle mit **allen** Zeichen, nutzen vorkommendes Zeichen als **Index**

```
enum {MAXCHAR=248};
// covers all ISO 8859-1 umlauts
int disttable[MAXCHAR];

int init_table(const char* pattern) {
  int len=length(pattern);
  for(int i=0; i<MAXCHAR; ++i)
    disttable[i]=len;
  for(int i=0; i<len-1; ++i) {
    if ((((unsigned char)pattern[i])>=MAXCHAR)||(pattern[i]<0))
      return -1;
    if (disttable[(int)pattern[i]]>len-i-1)
      disttable[(int)pattern[i]]=len-i-1;
  }
  return 0;
}
```

- Uneffizient, aber (noch) praktikabel ➡ viel schlechter bei Unicode o.ä.
- Besser: Nutzung eines Hashes ➡ VL „Algorithmen und Datenstrukturen“
- Bei Python kann Dictionary-Typ genutzt werden

Implementation der Suchfunktion

- Die Implementation der Suchfunktion ist ähnlich wie im simplen Algorithmus

```
int search(const char* p, const char* str, size_t tlen)
{
  size_t plen=length(p);
  if (plen>tlen) return -1;
  size_t pos=plen-1;
  while(pos<tlen){
    size_t j=0;
    while((j<plen) && (str[pos-j]==p[plen-j-1])){
      if (j==plen-1)
        return pos-plen+1;
      ++j;
    }
    pos=pos+disttable[(int)str[pos]];
  }
  return -1;
}
```

Bewertung des Algorithmus

- Der Algorithmus arbeitet am effizientesten, wenn die zu durchsuchende Zeichenkette `str` der Länge n nur aus Zeichen besteht, die im Muster `p` der Länge m **nicht** vorkommen (bester Fall)
  - ➡ Dann ist Komplexität $\mathcal{O}(\frac{n}{m})$, also **sublinear**[1]
- Im **ungünstigsten** Fall ist Komplexität jedoch ebenfalls $\mathcal{O}(m \cdot n)$ ➡ zwei verschachtelte Schleifen mit maximal n bzw. m Durchläufen
- Außerdem wird zusätzliche Zeit für Tabellenaufbau benötigt
- In der Praxis (häufig relativ lange Texte) wird der Boyer-Moore-Algorithmus in der Regel jedoch wesentlich **effizienter** ablaufen als der einfache Mustervergleich
- Weitere Verbesserungen (relativ einfach) möglich ➡ $\mathcal{O}(n + m)$

[1] ...falls $m \ll n$, sonst irgendwann $\mathcal{O}(m + \frac{n}{m})$ aufgrund des Aufbaus der Tabelle

10.4 Wildcards

- Erweitern Problemstellung: Suchmuster soll **Jokerzeichen** (Wildcards) enthalten können
- Nutzen drei Wildcards:
  - „?“ Passt auf **genau** einen Buchstaben
  - „!“ Passt auf **einen** oder **keinen** Buchstaben
  - „*“ Passt auf eine **beliebige Anzahl** von Buchstaben (auch 0)
- **Beispiele**:
  - „A?C“ wird in „abcABC“ und „aaAcCcc“ gefunden, aber nicht in „aACc“
  - „A!C“ wird in „abcABC“ und „aACc“ gefunden, aber nicht in „aAbbCc“
  - „A*C“ wird in „aAbbCc“ und „aACc“ gefunden, aber nicht in „cCbbAa“

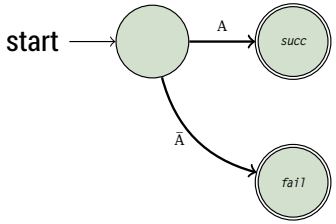
Idee

- Nutzung von Automaten (siehe Kapitel 5)
  - Ausdrücke mit Wildcards sind **regulär**
  - Äquivalenter Automaten existiert zum Test auf Übereinstimmung
    - Der Suchstring definiert eine reguläre Grammatik
    - Automat kommt in einen **Akzeptiere**-Zustand an, wenn ein der Grammatik entsprechender Ausdruck im zu durchsuchenden Text gefunden wird
- Jeder Suchstring ist anders ➡ Automat muss (zur Laufzeit!) **generiert** werden
  - Automat besteht aus generischen Teilautomaten
  - Vom Start kommt man in einen der End-Zustände:
    - (Teil-)**Erfolg** (`s`, „gefunden“) = *accept*
    - **Misserfolg** (`f`, „nicht gefunden“)
  - Ohne Jokerzeichen ➡ Kette ➡ *accept*-Zustand ist *start*-Zustand des nächsten Teilautomat

Automaten

- Automat für Nicht-Joker-Zeichen (z.B. „A“)
  - Ausgangskante für Eingabe von A ➡ `s` = nächster Zustand
  - Ausgangskante für alles andere ➡ `f`

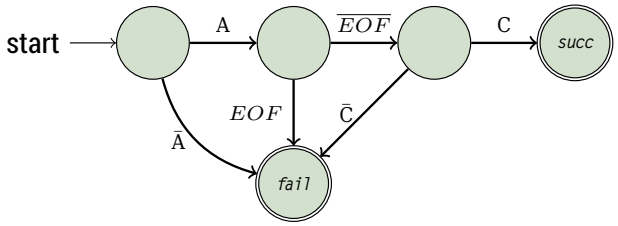
„A“ ➡



Automaten (Forts.)

- Automat für Joker-Zeichen „?“ (z.B. „A?C“)
  - Ausgangskante für End-of-Text (EoT) ➡ `f`
  - Ausgangskante für alles andere ➡ `s` = nächster Zustand

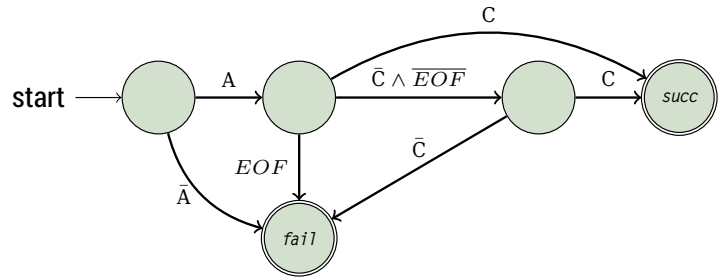
„A?C“ ➡



Automaten (Forts.)

- Automat für Joker-Zeichen „!“ (z.B. „A!C“) ➡ nutzt Zeichen der Ausgangskante des **nächsten** Zustandes
  - Ausgangskante für Eingabe von C ➡ **über**nächster Zustand
  - Ausgangskante für End-of-Text ➡ `f`
  - Ausgangskante für alles andere ➡ nächster Nicht-„!“-Zustand

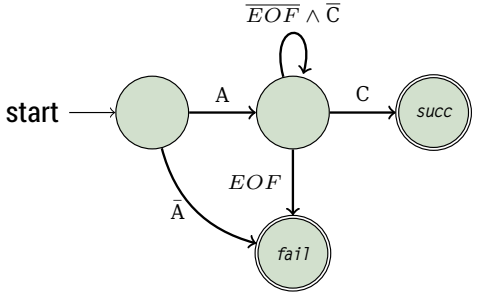
„A!C“ ➡



Automaten (Forts.)

- Automat für Joker-Zeichen „*“ (z.B. „A*C“)
  - Ausgangskante für Eingabe von c ➡ nächster Zustand
  - Ausgangskante für End-of-Text ➡ `f`
  - Für alles andere ➡ Schleife zu sich selbst

„A*C“ ➡



Problem

- **Problem**: Was ist, wenn „!“ oder „*“ **kein** normales Zeichen folgt?
- **Beobachtungen**:
  1. Eine Reihe aus ausschließlich „!“ sind nicht kritisch
  2. Beginnt das Suchmuster mit „*“ oder einem „!“, so kann das Wildcard ignoriert (übergangen) werden
  3. Folgt einem „*“ oder einem „!“ End-of-Text, so kann das Wildcard ignoriert (übergangen) werden
  4. Folgt einem „!“ oder einem „*“ ein „?“, so können beide Zeichen getauscht werden
  5. Folgt einem „!“ ein „*“, so kann das „!“ ignoriert (übergangen) werden
  6. Folgt einem „*“ ein „!“ oder „*“, kann das Folgezeichen ignoriert werden
- **Folgerung**: Durch Transformation kann immer erreicht werden, dass die Generierungsregeln hinreichen sind
- Entsprechend der Beobachtungen 2-6 können Umformregeln in einem Algorithmus formuliert werden

Algorithmus

Algorithm SANITIZE
Require: str, possibly with wildcards
Ensure: returns sanitized str

repeat
  while $str[first] =$'*' $\vee$ $str[first] =$'!' **do**
    remove first character from str ▷ #2
  end while
  while $str[last] =$'*' $\vee$ $str[last] =$'!' **do**
    remove last character from str ▷ #3
  end while
  $changed \leftarrow$true
  for $\forall\, sub \in str, length(sub) = 2$ **do**
    if $sub =$'!?' **then**
      replace in str sub with '?!' ▷ #4
    else if $sub =$'*?' **then**
      replace in str sub with '?*' ▷ #4
    else if $(sub =$'!*'$) \vee (sub =$'*!'$) \vee (sub =$'**'$)$ **then**
      replace in str sub with '*' ▷ #5,#6
    else
      $changed \leftarrow$false
    end if
  end for
until ($changed =$false)

Implementation der Transformationsregeln

- Da in Python Stringmanipulationen einfacher sind, betrachten wir zunächst eine Implementation in Python

```
def sanitize(str):
  changed=True
  while (changed==True):                    # repeat as often as needed
    changed=False
     while (str[0]=='*') or (str[0]=='!'):   # delete leading * or !
      str=str[1:]
    while (str[-1]=='*') or (str[-1]=='!'): # delete tailing * or !
      str=str[:-1]
      for i in range(0,len(str)-2):
        if (str[i:i+2]=='*?') or (str[i:i+2]=='!?'):
          changed=True
          str=str[0:i]+'?'+str[i]+str[i+2:] # *? => ?* and !? => ?!
        if ((str[i:i+2]=='!*') or (str[i:i+2]=='*!') or
            (str[i:i+2]=='**')):
          changed=True
          str=str[0:i]+'*'+str[i+2:]        # !* or *! or ** => *
  return str
```


C-Implementierung

- Die C-Implementierung ist etwas länger, macht aber genau das Gleiche

```
char* sanitize(char* str) {
  char* nstr=str;
  bool changed;
  int len=length(nstr);
  if ((nstr==NULL) || (len==0)) return NULL;
  do {
    changed=false;
    while((nstr[len-1]=='*') || (nstr[len-1]=='!')){
      nstr[--len]='\0'; /* shorten tail by 1 */
    }
    while((nstr[0]=='*') || (nstr[0]=='!')) {
      nstr=&nstr[1];    /* move start 1 to the right  */
      --len;
    }
    for(int i=0; i<len; ) {
      if (((nstr[i]=='*') || (nstr[i]=='!')) && (nstr[i+1]=='?')) {
        nstr[i+1]=nstr[i];  /* !? => ?! resp. *? => ?* */
        nstr[i]='?';
        changed=true;
      } else ++i;
    }
                                    ⋮
```


C-Implementierung (Forts.)

```
                                    ⋮
    for(int i=1; i<len-1; ){
      if (((nstr[i]=='*') && (nstr[i+1]=='!')) ||
          ((nstr[i]=='!') && (nstr[i+1]=='*')) ||
          ((nstr[i]=='*') && (nstr[i+1]=='*'))) {
        --len;
        nstr[i]='*';
        int j=i+1;
        do{   /* move remains one to the left */
          nstr[j]=nstr[j+1];
          ++j;
        } while(nstr[j]!='\0');
        changed=true;
      }
      else ++i;
    }
  } while(changed);
  return nstr;
}
```

Datenstrukturen

- Der Automat muss zur Laufzeit generiert werden ➡ Darstellung in Daten, nicht im Code
- Was ist eine geeignete Datenstruktur?
  - Automat ist ein Graph (vergleiche Exkurs im Kapitel 7), d.h., er besteht aus Knoten und Kanten
  - Knoten sind die Zustände im Automaten
  - Kanten sind die Übergänge
- Da sowohl Knoten als auch Kanten weitere Informationen „tragen“, ist eine Adjazenzmatrix ungeeignet ➡ wir nehmen `struct`s mit Verweisen, vergleiche Kapitel 7, Folie 13

Datenstrukturen (Forts.)

- Was ist nötig zur Beschreibung eines **Übergangs**?
  1. Ausgangs- und Endzustand
  2. Bedingung
- Wir werden die Ausgangskanten den entsprechenden Zuständen zuordnen, deshalb brauchen wir nur das Ziel
- Es kommen im Automaten nur fünf verschiedene Bedingungen vor:
  1. Das Zeichen wird gefunden
  2. Das Zeichen wird nicht gefunden
  3. Der Text ist zu Ende
  4. Der Text ist nicht zu Ende
  5. Es wurde weder das Zeichen noch das Textende gefunden
- Entsprechend können wir Konstanten und eine Datenstruktur vereinbaren:

```
17 typedef enum { MatchChar,
         MatchNotChar, MatchEOT,
         MatchNotEOT,
         MatchNotCharNotEOT,
         NoEdge} match_t;

19 typedef struct{
20   match_t condition;
21   int          next;
22 } edge_t;
```

- Mit der Konstante `NoEdge` sollen Kanten mariert werden, die im Automaten nicht existieren

Datenstrukturen (Forts.)

- Wir Beschreibung einen Zustand mit:
  1. Markierung ob Endzustand (`s` oder `f`) oder anderer
  2. Das Zeichen, auf den sich der Zustand bezieht
  3. Die Übergänge von diesem Zustand
- Da es immer nur maximal drei Übergänge zu einem anderen Zustand gibt, erlauben wir uns, ggf. etwas Speicherplatz zu verschwenden und ein Array für genau drei Kanten anzulegen

```
25 typedef enum { StateDefault,
         StateSuccess,
         StateFail} statetype_t;

27 typedef struct{
28   statetype_t type;
29   char        ch;
30   edge_t      edge[3];
31 } state_t;
```


Datenstrukturen (Forts.)

- Wir wissen erst zur Laufzeit, wie groß unser Automat wird, d.h., wie viele Zustände er hat
- Wir legen anonymes Array an ➡ brauchen Zeiger
- Zusätzlich Index des Startzustands

```
35 typedef struct{
36   state_t* state;
37   int      initial;
38 } automata_t;
```

Konstruktion des Automaten

- Nun kann der Automat zur Laufzeit aus dem Suchmuster erzeugt werden

```
48 automata_t init_automata(const char*);
```

- Wir brauchen...
  - einen `Success`-Zustand
  - einen `Fail`-Zustand
  - einen Zustand pro Nicht-`*`-Zeichen im Suchmuster
- Dafür wird Speicher reserviert

```
188 automata_t init_automata(const char* p) {
189   int snr, len;
190   len=snr=length(p);
191   for(int i=len-1; i>=0; --i)
192     if(p[i]=='*') --snr;  // decrease for each "*"
193   automata_t a;
194   a.state = calloc(snr+2, sizeof(state_t));
```


Konstruktion des Automaten (Forts.)

- In das anonyme Array werden die Elemente (Automatenzustände) eingetragen
- Der `Success`- und der `Fail`-Zustand sind immer gleich, wir vereinbaren dafür Indexkonstanten:

```
33 enum { S_FAIL=0, S_SUCC=1 };
195   /* Fail */
196   a.state[S_FAIL].type=StateFail;
197   a.state[S_FAIL].edge[0].condition=NoEdge;
198   a.state[S_FAIL].edge[1].condition=NoEdge;
199   a.state[S_FAIL].edge[2].condition=NoEdge;
200   /* Success */
201   a.state[S_SUCC].type=StateSuccess;
202   a.state[S_SUCC].edge[0].condition=NoEdge;
203   a.state[S_SUCC].edge[1].condition=NoEdge;
204   a.state[S_SUCC].edge[2].condition=NoEdge;
```

- Theoretisch könnten sie ausgelassen werden
- Jedoch vereinfacht ihre Existenz die allgemeine Behandlung

Konstruktion des Automaten (Forts.)

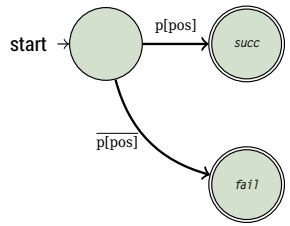
- Die anderen Zustände werden durch Fallunterscheidung aus dem Suchmusterstring generiert
- Fangen vom Ende an ➡ haben Folgezustände stets zur Verfügung

```
206   snr=1;
207   for(int pos=len-1; pos>=0; --pos){
208     switch (p[pos]) {
                                    ⋮
245     }
246   }
```


Konstruktion des Automaten (Forts.)

- Zunächst der Fall für normale Zeichen:
  - Zwei Ausgangskanten
  - Eine hat das Zeichen aus dem Suchmuster zur Bedingung, die andere die Negation
  - Bedingungen mit der Markierung `NoEdge` werden später nicht ausgewertet

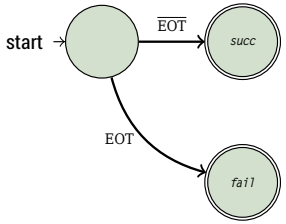
```
208     switch (p[pos]) {
209     default:
210       ++snr;
211       a.state[snr].ch=p[pos];
212       a.state[snr].type=StateDefault;
213       a.state[snr].edge[0].condition=MatchChar;
214       a.state[snr].edge[0].next=snr-1;
215       a.state[snr].edge[1].condition=MatchNotChar;
216       a.state[snr].edge[1].next=S_FAIL;
217       a.state[snr].edge[2].condition=NoEdge;
218       break;
```



Konstruktion des Automaten (Forts.)

- Der Fall für das Wildcard-Zeichen „?“ ist ähnlich wie der für ein normales Zeichen, nur dass die Bedingung $\overline{EOT}$ ist:

```
219     case '?':
220       ++snr;
221       a.state[snr].type=StateDefault;
222       a.state[snr].edge[0].condition=MatchNotEOT;
223       a.state[snr].edge[0].next=snr-1;
224       a.state[snr].edge[1].condition=MatchEOT;
225       a.state[snr].edge[1].next=S_FAIL;
226       a.state[snr].edge[2].condition=NoEdge;
227       break;
```



Konstruktion des Automaten (Forts.)

- Der Zustand für das Wildcard-Zeichen „!“ braucht drei Ausgangskanten
  - Eine Kante führt zum **Folgezustand des Folgezustands** mit der Bedingung, dass das **Zeichen des Folgezustandes** gefunden wird
  - Eine Kante führt zum Folgezustand, wenn weder das Zeichen des Folgezustandes noch EOT (d.h. $\neq$ '\0') gefunden wird
  - Bei EOT landet der Automat in `Fail`

```
228     case '!':
229       ++snr;
230       a.state[snr].ch=a.state[snr-1].ch; // get character from next state
231       a.state[snr].type=StateDefault;
232       a.state[snr].edge[0].condition=MatchChar;
233       a.state[snr].edge[0].next=a.state[snr-1].edge[0].next;
234       a.state[snr].edge[1].condition=MatchEOT;
235       a.state[snr].edge[1].next=S_FAIL;
236       a.state[snr].edge[2].condition=MatchNotCharNotEOT;
237       a.state[snr].edge[2].next=snr-1;
238       break;
```


Konstruktion des Automaten (Forts.)

- Das Wildcard-Zeichen „*“ generiert **keinen** neuen Zustand
- Vielmehr wird der Zustand des Folgezeichens **modifiziert**
  - Die Bedingung der Kante zu `Fail` wird auf EOT abgeschwächt
  - Eine Schleife zu sich selbst wird hinzugefügt, wenn weder das ursprüngliche Zeichen noch EOT eintritt

```
239     case '*':
240       // no ++snr !
241       a.state[snr].edge[1].condition=MatchEOT;
242       a.state[snr].edge[2].condition=MatchNotCharNotEOT;
243       a.state[snr].edge[2].next=snr;
244       break;
```


Konstruktion des Automaten (Forts.)

- Zum Schluss: Bestimmung des Initialzustands des Automaten
- Da wir das Suchmuster „von hinten“ bearbeitet haben, ist der Initialzustand der letzte generierte Zustand

```
206   snr=1;
207   for(int pos=len-1; pos>=0; --pos){
208     switch (p[pos]) {
                                          ⋮
245     }
246   }
247   a.initial=snr;
248   return a;
249 }
```

- Damit ist der Automat konstruiert und kann eingesetzt werden

Ausführung des Automats

- Nachdem wird den Automat für die Suche initialisiert haben, wird er bei der Suche ausgeführt
- Da durch die Wildcards die Länge des gefundenen Textstücks nicht vorher klar ist, muss die Suche zwei Werte zurückgeben

```
40 typedef struct {
41   int start;
42   int end;
43 } searchresult_t;

45 searchresult_t search(automata_t, const char*, size_t);
```

- Bei der Ausführung muss im Wesentlichen abgeprüft werden, ob eine Bedingung zutrifft, und dann in den entsprechenden Folgezustand gewechselt werden
- Kommt der Automat dabei in `Fail` oder `Success` wird der Automat beendet

Ausführung des Automats (Forts.)

```
88 searchresult_t search(automata_t a, const char* text, size_t tlen) {
89   searchresult_t res;
90   for(size_t pos=0; pos<tlen; ++pos){
91     int snr=a.initial;
92     size_t j=pos;
93     while(a.state[snr].type==StateDefault){
94       for(int i=0; i<3; i++){
95         if (((a.state[snr].edge[i].condition==MatchChar) &&
96              (text[j]==a.state[snr].ch))    ||
97             ((a.state[snr].edge[i].condition==MatchNotChar) &&
98              (text[j]!=a.state[snr].ch))    ||
99             ((a.state[snr].edge[i].condition==MatchEOT) &&
100             (text[j]=='\0'))               ||
101            ((a.state[snr].edge[i].condition==MatchNotEOT) &&
102             (text[j]!='\0'))               ||
103            ((a.state[snr].edge[i].condition==MatchNotCharNotEOT) &&
104             (text[j]!='\0') &&
105             (text[j]!=a.state[snr].ch)))
106           {
```


Ausführung des Automats (Forts.)

```
107             snr = a.state[snr].edge[i].next;
108             ++j;  /* next character */
109             break;
110           }
111       }  /* end for over edges */
112     }
113     if (a.state[snr].type==StateSuccess) {
114       res.start=pos;
115       res.end=j;
116       return res;
117     }
118   } /* end for over text */
119   res.start=-1;
120   res.end=-1;
121   return res;
122 }
```


Ausgabe

- Durch den neuen Typ des Suchergebnisses muss auch die Ausgabe modifiziert werden

```
139 void presentresult(searchresult_t res, const char* str) {
140   int start, prelen;
141   start=res.start>20? res.start-20 : 0; /* start where? */
142   prelen=res.start>20? 20 : res.start;  /* length of prefix */
143   printf("%.*s[%.*s]%.20s\n",prelen, &str[start],
144          res.end-res.start, &str[res.start],&str[res.end]);
145 }
```

`main()`-Funktion

- Die `main()`-Funktion ist schließlich ähnlich der der bisherigen Suchprogramm-Varianten
- Das komplette Listing ist im Anhang **??** des Skripts zu finden

```
55 int main(int argc, char* argv[]) {
56   FILE* file;
57   char* text;
58   char* pattern;
59   automata_t automata;
60   searchresult_t found;
61   size_t size;

 1         /* ... error checks and file/memory
 2               handling as in simple search ... */

76   text[size]='\0';
77   pattern = sanitize(argv[1]);
78   automata= init_automata(pattern);

80   found= search(automata,text,size);
81   if (found.start != -1)
82     presentresult(found,text);
83   free(text);
84   free(automata.state);
85   return found.start;
86 }
```


Diskussion

- Der Algorithmus hat – wie die einfache Suche – eine Komplexität $\mathcal{O}(m \cdot n)$, wobei n die Größe des zu durchsuchenden Texts und m die **maximale** (expandierte) Länge des Suchmusters ist
  - Da im schlechtesten Fall bei Vorkommen von „*“ im Suchmuster die expandierte Länge die des zu durchsuchenden Textes ist, ist die Komplexität $\mathcal{O}(n^2)$
  - Daher wird in der Praxis meist vereinbart, dass Suchen nur zeilenweise durchgeführt werden, ein Zeilenende nicht „gematcht“ wird oder es eine maximale Expansionslänge gibt
- Wildcard-Suchalgorithmen werden häufig nicht über den hier dargestellten Automatenansatz realisiert, sondern mit Hilfe **rekursiver** Funktionen

Aufgaben

Aufgabe 10.1

Erweitern Sie den Wildcard-Algorithmus so, dass der reguläre Text die Zeichen „*“, „!“ und „?“ enthalten darf, d.h. dass Sie **trotz** Wildcards **alle** Zeichen auch regulär nutzen können!

Aufgabe 10.2

Implementieren Sie eine Textsuche mit Wildcards mit Hilfe von Rekursion und Backtracking (vgl. Kapitel 9)!

Aufgaben (Forts.)

Aufgabe 10.3

Gegeben sei eine sequentielle Datei mit maximal 4 Milliarden 32-Bit-Integerzahlen in zufälliger Reihenfolge. Sie sollen einen effizienten Algorithmus entwickeln, der eine Zahl findet, die **nicht** in dieser Datei vorkommt. Sie dürfen zwar davon ausgehen, dass Sie viel RAM-Speicher zur Verfügung haben, aber auf jeden Fall weniger als 3 GByte.

Bedenken Sie, dass ein Zugriff auf eine Datei 3 – 6 Zehnerpotenzen langsamer ist, als ein Zugriff auf den Speicher. Versuchen Sie deshalb, die Anzahl der Filezugriffe zu minimieren.

10.6 Referenzen

[BM77] Robert S. Boyer und J. Strother Moore. „A fast string searching algorithm“. In: *Communications of the ACM* 20.10 (1977), S. 762–772