

**Definition** Unter *Erreichbarkeitsanalyse* versteht man

- Berechnung aller erreichbaren Zustände
  - Resultat kann explizit oder symbolisch repräsentiert werden
  - wird für weitere Algorithmen oder Optimierung der Systeme benötigt
- Überprüfung, ob gewisse Zustände erreichbar sind
  - entspricht Model Checking von Sicherheitseigenschaften (Safety)

**explizit** = jeder Zustand/Übergang wird einzeln repräsentiert

**symbolisch** = Mengen-Repräsentation von Zuständen/Übergängen durch Formeln

	Explizites Modell	Symbolisches Modell
Explizite Analyse	Graphensuche	Explicit Model Checking
Symbolische Analyse	—	Symbolic Model Checking

klassisch: Symbolic MC für HW, explicit MC für SW

(Praxis heute: symbolic und explicit für SW+HW)

- für die Theorie reicht explizite Matrix-Darstellung von  $T$

Übergangsmatrix		0	1	2	3	4	5	6	7
Anfangszustand: 0	0	0	0	1	0	1	0	0	0
	1	0	0	0	1	0	1	0	0
	2	0	0	0	0	1	0	1	0
	3	0	0	0	0	0	1	0	1
	4	1	0	0	0	0	0	1	0
	5	0	1	0	0	0	0	0	1
	6	1	0	0	0	0	0	0	0
	7	0	1	0	1	0	0	0	0

- in der Praxis ist  $T$  in einer Modellierungs- oder Programmiersprache beschrieben

```
initial_state = 0;
```

```
next_state = (current_state + 2 * (bit + 1)) % 8;
```

- symbolische Repräsentation kann exponentiell kompakter sein

- Symbolische Travesierung des Zustandraumes
  - von einer Menge von Zuständen werden in einem Schritt die Nachfolger bestimmt
  - Nachfolger werden symbolisch berechnet und repräsentiert
  - legt Breitensuche nahe
  - im allgemeinen schon nicht berechenbar
- Resultat ist eine symbolische Repräsentation aller erreichbaren Zustände
  - z.B. `0 <= state && state < 8 && even (state)`

- symbolische Repräsentation hat oft einen “Paralleloperator”
  - z.B. Produkt von Automaten, der einzelne Automat wird explizit repräsentiert

$$G = K_1 \times K_2$$

- Komponenten des Systems werden separat programmiert

```
process A begin P1 end || process B begin P2 end;
```

- Additives Eingehen der Größe der symbolischen Komponenten
  - Programmgröße des Systems:  $|K_1| + |K_2|$ , bzw.  $|P1| + |P2|$

- Multiplaktives Eingehen der Anzahl Komponentenzustände in Systemzustandszahl:

$$|S_G| = |S_{K_1}| \cdot |S_{K_2}|$$

- offensichtlich bei sequentieller Hardware:
  - $n$ -Bit Zähler läßt sich mit  $O(n)$  Gattern implementieren, hat aber  $2^n$  Zustände
- Hierarchische Beschreibungen führen zu einem weiteren exponentiellen Faktor
- bei Software noch komplexer:
  - in der Theorie nicht primitiv rekursiv (siehe Ackerman-Funktion)
  - in der Praxis ist der Heap (dynamisch allozierte Objekte) das Problem

- explizite Repräsentation reduziert sich auf Graphensuche
  - suche erreichbare Zustände von den Anfangszuständen
  - linear in der Grösse **aller** Zustände
  
- symbolische Modellierung
  - Berechnung der Nachfolger eines Zustandes ist das Hauptproblem
  - **on-the-fly** Expansion von  $T$  für einen konkreten erreichten Zustand
  - damit Vermeidung der Berechnung von  $T$  für unerreichbare Zustände
  - alternativ symbolische Berechnung/Repräsentation der Nachfolgezustände

- Markiere besuchte Zustände
  - Implementierung abhängig davon, ob Zustände **On-the-fly** generiert werden
- Unmarkierte Nachfolger besuchter Zustände kommen auf den Suchstack
- Nächster zu bearbeitender Zustand wird oben vom Stack genommen
- Falls “Fehlerzustand” erreicht wird hat man einen Weg dorthin
  - der Pfad bzw. Fehlertrace findet sich auf dem Stack
  - der Einfachheit wegen breche die Suche ab

```
recursive_dfs_aux (Stack stack, State current)
{
    if (marked (current))
        return;

    mark (current);
    stack.push (current);

    if (is_target (current))
        stack.dump_and_exit ();    /* target reachable */

    forall successors next of current
        recursive_dfs_aux (stack, next);

    stack.pop ();
}
```

```
recursive_dfs ()  
{  
    Stack stack;  
  
    forall initial states state  
        recursive_dfs_aux (stack, state);  
  
    /* target not reachable */  
}
```

- Abbruch beim Erreichen des Zieles ist schlechter Programmierstil!
- Effizienz verlangt nicht-rekursive Variante
- Markieren sollte durch Hashen ersetzt werden
  - damit Zustände on-the-fly *erzeugt* werden können

```
#include <stdio.h>

void
f (int i)
{
    printf ("%d\n", i);
    f (i + 1);
}

int
main (void)
{
    f (0);
}
```

dieses C-Programm stürzt nach “wenigen” Rekursionsschritten ab

```
non_recursive_dfs_aux (Stack stack)
{
  while (!stack.empty ())
  {
    current = stack.pop ();
    if (is_target (current))
      dump_family_line_and_exit (current);
    forall successors next of current
    {
      if (cached (next)) continue;
      cache (next);
      stack.push (next);
      next.set_parent (current);
    }
  }
}
```

```
non_recursive_dfs ()
{
    Stack stack;

    forall initial states state
        stack.push (state);

    non_recursive_dfs_aux (stack);

    /* target not reachable */
}
```

```
non_recursive_buggy_dfs_aux (Stack stack)
{
  while (!stack.empty ())
  {
    current = stack.pop ();
    if (is_target (current))
      dump_family_line_and_exit (current);
    if (cached (current)) continue;
    cache (current);
    forall successors next of current
    {
      stack.push (next);
      next.set_parent (current);
    }
  }
}
```

```
non_recursive_but_also_buggy_aux (Stack stack)
{
  while (!stack.empty ())
  {
    current = stack.pop ();
    forall successors next of current
    {
      if (cached (next)) continue;
      cache (next);
      stack.push (next);
      next.set_parent (current);
      if (is_target (next))
        dump_family_line_and_exit (next);
    }
  }
}
```

```
bfs_aux (Queue queue)
{
  while (!queue.empty ())
  {
    current = queue.dequeue ();
    if (is_target (current))
      dump_family_line_and_exit (current);
    forall successors next of current
    {
      if (cached (next)) continue;
      cache (next);
      queue.enqueue (next);
      next.set_parent (current);
    }
  }
}
```

```
bfs ()
{
    Queue queue;

    forall initial states state {
        cache (state);
        queue.enqueue (state);
    }

    bfs_aux (queue);

    /* target not reachable */
}
```

- Tiefensuche
  - einfach rekursiv zu implementieren  
(was man aber sowieso nicht sollte)
  - erlaubt Erweiterung auf Zyklenerkennung  $\Rightarrow$  Liveness
- Breitensuche
  - erfordert nicht-rekursive Formulierung
  - kürzeste Gegenbeispiele
    - \* findet immer kürzesten Pfad zum Ziel

- Vorwärtstraversierung bzw. Vorwärtsanalyse
  - Nachfolger sind die als nächstes zu betrachtenden Zustände
  - analysierte Zustände sind alle erreichbar
  
- Rückwärtstraversierung bzw. Rückwärtsanalyse
  - Vorgänger sind die als nächstes zu betrachtenden Zustände
  - manche untersuchten Zustände können unerreichbar sein
  - meist nur in Kombination mit symbolischen Repräsentationen von Zuständen
  - symbolische Darstellungen bei Rückwärtsanalyse meist “zufällig” und komplex
  - **manchmal terminiert Rückwärtsanalyse in wenigen Schritten**

- Globale Analyse
  - anwendbar auf Rückwärts- oder Vorwärtstraversierung
  - arbeitet mit *allen* globalen Zuständen (inklusive nicht erreichbaren)
  - Kombination mit expliziter Analyse skaliert sehr schlecht
- Lokale oder On-The-Fly Analyse
  - arbeitet nur auf den erreichbaren Zuständen
  - generiert und analysiert diese on-the-fly ausgehend von Anfangszuständen
- gemischte Vorgehensweise möglich:
  - bestimme vorwärts erreichbare Zustände, darauf dann globale Analysen

```
process A {  
  int count;  
  run() {  
    while (true)  
      if (b.count != this.count)  
        count = (count + 1) % (1 << 20);  
  }  
}
```

```
process B {  
  int count;  
  run() {  
    while (true)  
      if (a.count == this.count)  
        count = (count + 1) % (1 << 20);  
  }  
}
```

A a;

B b;

- bei globaler Analyse wird jedem Zustand ein *Markierungsbit* hinzugefügt

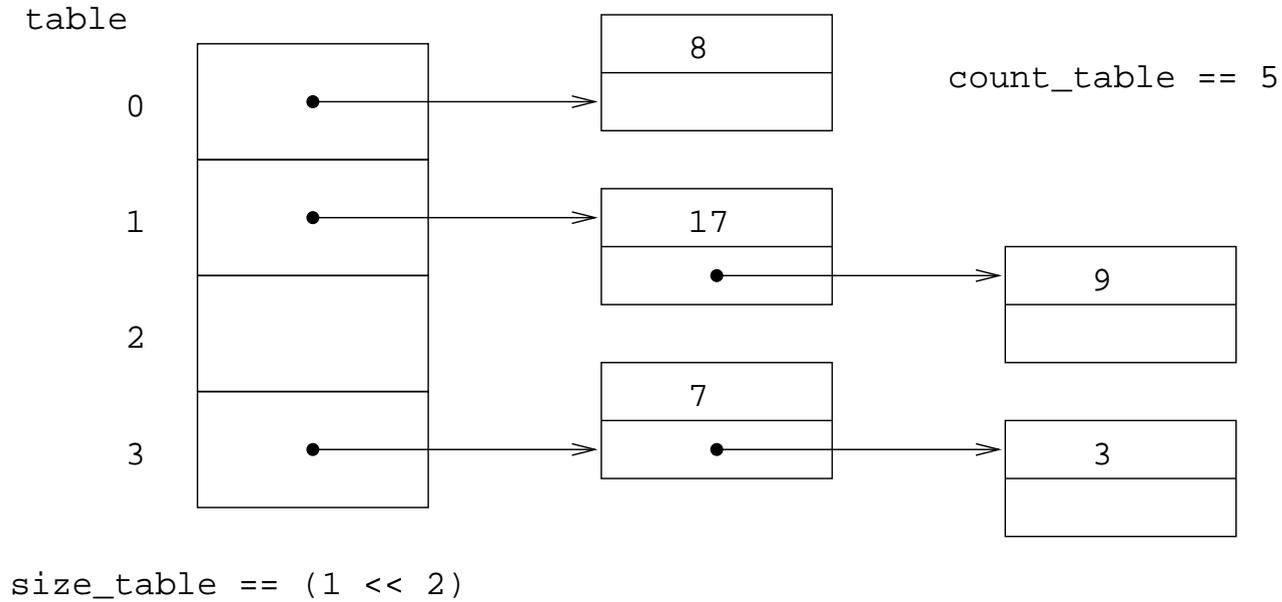
```
bool cached (State state) { return state.mark; }
```

```
void cache (State state) { state.mark = true; }
```

- bei globaler Analyse **ohne** On-the-fly Zustandsgenerierung
  - $2^{64} = 2^{32} \cdot 2^{32}$  **mögliche** Zustände können nicht vorab alloziert werden
- bei lokaler Analyse **mit** On-the-fly Zustandsgenerierung
  - $2 \cdot 2^{20} = 2097152$  **erreichbare** Zustände
  - ergeben 8-fache Menge an Bytes  $\approx 16$  MB
  - (plus Overhead diese zu Speichern,  $<$  Faktor 2)

- Anforderungen an Datenstruktur zur Speicherung von Zuständen:
  - besuchte Zustände müssen markiert/gespeichert werden (cache)
    - ⇒ pro neuem Zustand eine Einfügeoperation
  - neue Nachfolger werden auf “schon besucht” getestet (cached)
    - ⇒ pro Nachfolger eine Suche auf Enthaltensein
- Alternative Implementierungen:
  - Bit-Set: pro *möglichem* Zustand ein Bit (im Prinzip wie bei globaler Analyse)
  - Suchbäume: Suche/Einfügen logarithmisch in Zustandsanzahl
  - **Hashtabelle:** Suche/Einfügen konstant in Zustandsanzahl

- Laufzeit kann als konstant angesehen werden
  - in der Theorie weit komplexere Analyse notwendig
- “gute” Hashfunktion ist das Wesentliche
  - Randomisierung bzw. Verteilung der Eingabebits auf Hashindex
- Anpassung der Grösse der Hashtabelle:
  - entweder durch dynamische (exponentielle) Vergrößerung
  - oder Maximierung und Anpassung an den verfügbaren Speicher



```
unsigned hash (unsigned data) { return data; }
```

```
struct Bucket *  
find (unsigned data)  
{  
    unsigned h = hash (data);  
    h &= (size_table - 1);  
    ...  
}
```

```
unsigned
bad_string_hash (const char * str)
{
    const char * p;
    unsigned res;

    res = 0;

    for (p = str; *p; p++)
        res += *p;

    return res;
}
```

```
unsigned
very_bad_string_hash (const char * str)
{
    const char * p;
    unsigned res;

    res = 0;

    for (p = str; *p; p++)
        res = (res << 4) + *p;

    return res;
}
```

[Dragonbook]

```
unsigned
classic_string_hash (const char *str)
{
    unsigned res, tmp;
    const char *p;

    res = 0;

    for (p = str; *p; p++)
        {
            tmp = res & 0xf0000000;    /* unsigned 32-bit */
            res <<= 4;
            res += *p;
            if (tmp)
                res ^= tmp >> 28;
        }

    return res;
}
```

- empirisch sehr gute Randomisierung bei Bezeichnern von Programmiersprachen
  - relative Anzahl Kollisionen als Maßzahl für die Qualität
- **schnell:** maximal 4 logisch/arithmetische Operationen pro Zeichen
- bei längeren Strings von 8 Zeichen und mehr gute Bit-Verteilung
- Überlagerung der 8-Bit Kodierungen der einzelnen Zeichen  
(man beachte aber die ASCII Kodierung)
- Klustering bei vielen kurzen Strings (z.B. in automatisch generiertem Code)

`n1, ..., n99, n100, ..., n1000`

```
static unsigned primes [] = { 2000000011, 2000000033, ... };  
#define NUM_PRIMES (sizeof (primes) / sizeof (primes[0]))
```

```
unsigned
```

```
primes_string_hash (const char * str)
```

```
{  
    unsigned res, i;  
    const char * p;  
  
    i = 0;  
    res = 0;  
    for (p = str; *p; p++)  
    {  
        res += *p * primes[i++];  
        if (i >= NUM_PRIMES)  
            i = 0;  
    }  
  
    return res;  
}
```

```
static unsigned primes [] = { 2000000011, 2000000033, ... };  
#define NUM_PRIMES (sizeof(primes)/sizeof(primes[0]))
```

```
unsigned
```

```
hash_state (unsigned * state, unsigned words_per_state)
```

```
{
```

```
    unsigned res, i, j;
```

```
    res = 0;
```

```
    i = 0;
```

```
    for (j = 0; j < words_per_state; j++)
```

```
    {
```

```
        res += state[j] * primes [i++];
```

```
        if (i >= NUM_PRIMES)
```

```
            i = 0;
```

```
    }
```

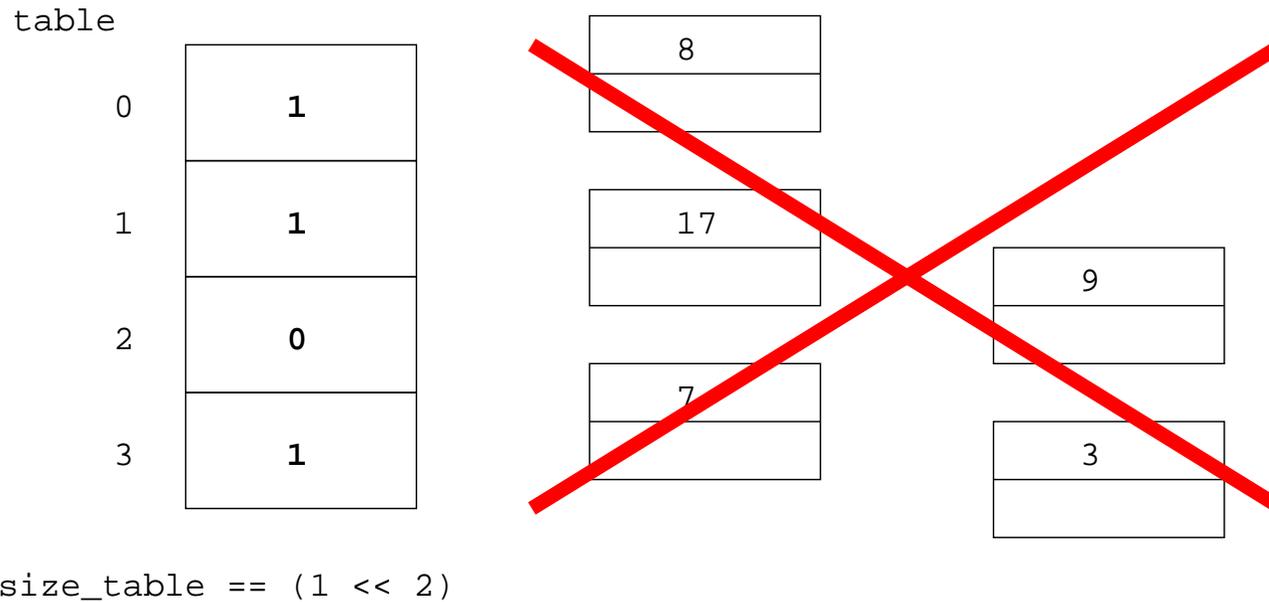
```
    return res;
```

```
}
```

```
#define CRC_POLYNOMIAL 0x82F63B78
static unsigned
crc_hash_bit_by_bit (const char * str)
{
    unsigned const char * p;
    unsigned res = 0;
    int i, bit;
    for (p = str; *p; p++) {
        res ^= *p;
        for (i = 0; i < 8; i++) {
            bit = res & 1;
            res >>= 1;
            if (bit)
                res ^= CRC_POLYNOMIAL;
        }
    }
    return res;
}
```

- lässt sich leicht parametrisieren
  - durch verschiedene Primzahlen oder Anfangspositionen
- heutzutage ist Integer-Multiplikation sehr schnell
  - durch mehr Platz für die Multiplizierer auf der ALU
  - und durch Vorhandensein mehrerer Integer-Functional Units
- was noch fehlt ist Anpassung an Tabellengröße
  - bei Zweierpotenzgröße: einfache Maskierung
  - sonst Primzahlgröße und Anpassung durch Modulo-Bildung

- Probleme mit expliziter **vollständiger** Zustandsraum-Exploration:
  1. wegen Zustandsexplosion oft zu viele Zustände
  2. ein einziger Zustand braucht oft schon viel Platz (dutzende Bytes)
  
- Ideen des Super-Trace Algorithmus:
  1. behandle Zustände mit demselben Hash-Wert als identisch
  2. speichere nicht die Zustände, sondern Hash-Werte erreichter Zustände
  
- Super-Trace wird auch **Bit-State-Hashing** genannt

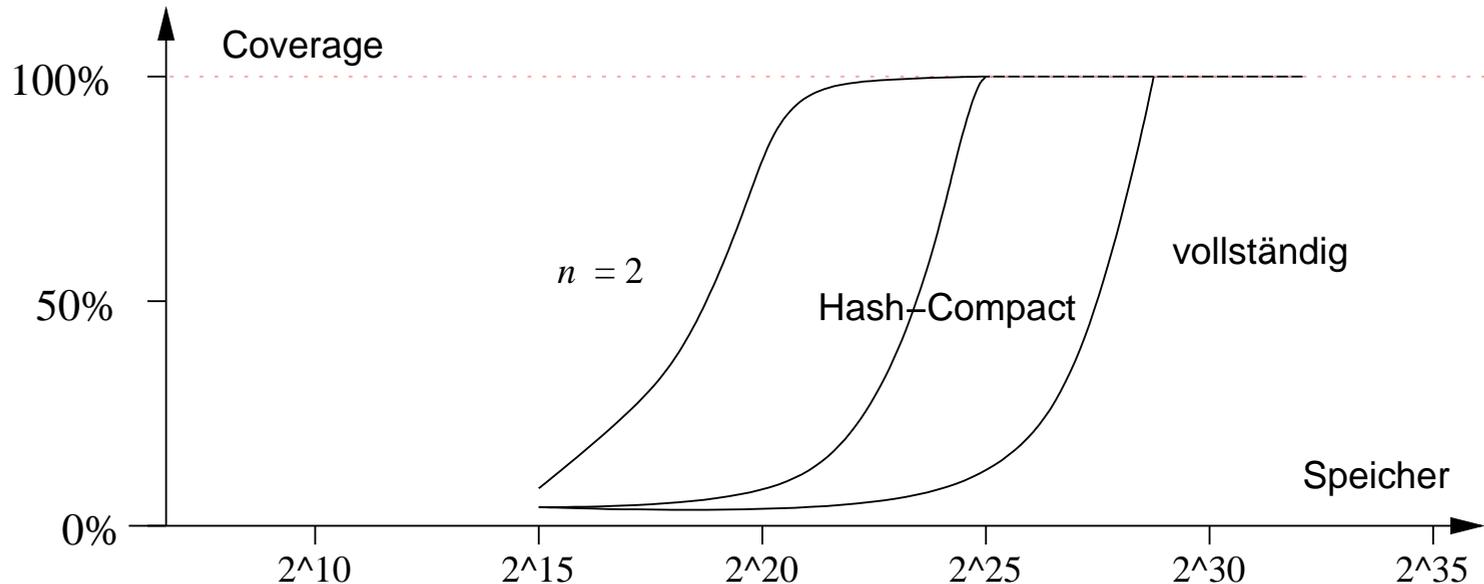


Falls Zustand mit gleichem Hash-Wert früher schon besucht wurde, dann gilt momentaner Zustand als besucht!

- Vorteile:
  - drastische Reduktion des Speicherplatzes auf ein Bit pro Zustand
  - Reduktion um mindestens 8 mal Größe des Zustandes in Bytes
  - Anpassung der Größe der Hash-Tabelle/Bit-Set an vorhandenen Speicher
  - Ausgabe einer unteren Schranke der Anzahl besuchten Zustände
  - Parametrisierung der Hash-Funktion erlaubt unterschiedliche Exploration
- Nachteile:
  - bei nicht-kollisionsfreier Hash-Funktion (der Normalfall) Verlust der Vollständigkeit
  - nur vage Aussage über Abdeckung möglich

- berechne zwei Hashwerte mit unterschiedlichen Hashfunktionen
  - Speichern von Zuständen durch Eintrag jeweils eines Bits in zwei Hashtabellen
  - Zustand wird als schon erreicht angenommen gdw. beide Bits gesetzt
  - z.B.  $h_1, h_2: Keys \rightarrow \{0, \dots, 2^{32} - 1\}$
  - zwei Hashtabellen mit jeweils  $2^{32}$  Bits =  $2^{29}$  Bytes = 512 MB
- lässt sich auch leicht auf  $n$  Hashfunktionen ausbauen
  - $n = 4$  mit 2 GB Speicher für Hashtabellen ist durchaus realistisch
  - vergl. Parameterisierung der Hashfunktion basierend auf Primzahlmultiplikation

- statt  $n$  Bits zu setzen kann man auch einfach den  $n$ -Bit Hash-Wert speichern
  - bei 256 MB =  $2^{28}$  Bytes Platz für die Hashtabelle bis zu  $2^{26}$  Hashwerte/Zustände (32-Bit Hashfunktion  $h: Keys \rightarrow \{0, \dots, 2^{32} - 1\}$ , 4 Bytes Hashwert pro Zustand)
- beliebige Varianten ergeben ungefähr folgendes Bild:



(schematisch nach [Holzmann 1998], 427567 erreichbare Zustände, 1376 Bits)

- Idealisierte Best-Case Annahmen:
  - verwendete Hash-Funktion ist möglichst kollisionsfrei
  - Hashtabelle wird solange wie möglich ohne Kollisionen gefüllt
- $m$  Speicherplatz in Bits,  $s$  Zustandsgrösse in Bits,  $r$  erreichbare Zustände

$$coverage_{n=2}(m) = \frac{m}{r \cdot 2}$$

$$coverage_{\text{hashcompact}}(m) = \frac{m}{r \cdot w} \quad w = \text{Wordgröße in Bits, z.B. 32 Bit, } w \leq \lceil \log_2 r \rceil$$

$$coverage_{\text{complete}}(m) = \frac{m}{r \cdot s}$$

(im vorigen Schaubild ist die x-Achse logarithmisch dargestellt)

- in der Praxis gibt es Kollisionen und die unvollständigen brauchen mehr Speicher