

# Algorithmus von Rabin und Karp

a	b	r	a	c	a	d	a	b	r	a
---	---	---	---	---	---	---	---	---	---	---

Gegeben  $u$  und  $v$  mit  $|v| = m$ .

Wähle eine geeignete Hashfunktion  $\Sigma^* \rightarrow \mathbf{N}$ .

- 1 Berechne  $h(v)$  und  $h(u[i \dots i + m - 1])$  für  $i = 1, \dots, |u| - m + 1$ .
- 2 Vergleiche  $v$  mit  $u[i \dots i + m - 1]$  falls Hashwerte übereinstimmen.

Bei einer guten Hashfunktion unterscheiden sich die Hashwerte fast immer, wenn die Strings verschieden sind.

Wie lange dauert es alle  $h(u[i \dots i + m - 1])$  zu berechnen?

# Algorithmus von Rabin und Karp

Wähle eine Hashfunktion, so daß alle

$$h(u[i \dots i + m - 1])$$

in  $O(m + |u|)$  Schritten berechnet werden können.

Zum Beispiel:

$$h(a_1 \dots a_n) = \left( \sum_{i=1}^n q^i a_i \right) \bmod p,$$

wobei  $p$  und  $q$  große Primzahlen sind.

Es gilt

$$h(a_2 \dots a_{n+1}) = \left( h(a_1 \dots a_n) / q - a_1 + q^n a_{n+1} \right) \bmod p.$$

# Editdistanz

Gegeben seien zwei Zeichenketten  $u$  und  $v$ .

Wir dürfen  $u$  auf drei Arten ändern:

- 1 Ersetzen: Ersetze ein Symbol in  $u$  durch ein anderes
- 2 Löschen: Entferne ein Symbol aus  $u$  (Länge wird kleiner)
- 3 Einfügen: Füge ein Symbol an beliebiger Stelle in  $u$  ein (Länge wird größer)

Frage: Wieviele solche Operationen sind **mindestens** nötig, um  $u$  zu  $v$  zu verwandeln.

Anwendung: Wie **ähnlich** sind die beiden Zeichenketten (z.B. DNA-Strings)?

# Editdistanz

Lösung durch dynamisches Programmieren.

GACGTCAGCTTACGTACGATCATTGACTACG

GACGTCAGCTACAGTAGATCATTGACTACG

# Divide-and-Conquer

- **Teilen** der Eingabe in möglichst gleich grosse Teile
- Rekursives Lösen (**Beherrschen**) der Teilprobleme
- Zusammenfügen der Teillösungen zur Gesamtlösung

# Dynamisches Programmieren

- Optimale Lösungen werden aus **optimalen Teillösungen** zusammengesetzt.
- Eine Tabelle von Teillösungen wird **bottom-up** aufgebaut.
- Sonderfall **Memoization**

# 0/1-Knapsack

Gegeben sei ein Rucksack mit einer unbeschränkten Kapazität und verschiedene Gegenstände. Jeder Gegenstand hat eine gegebene Größe und einen gegebenen Wert. Es gibt einen Zielwert. Frage: Können wir Gegenstände auswählen, die in den Rucksack hineinpassen und deren zusammengezählte Werte den Zielwert überschreiten?

Formaler:

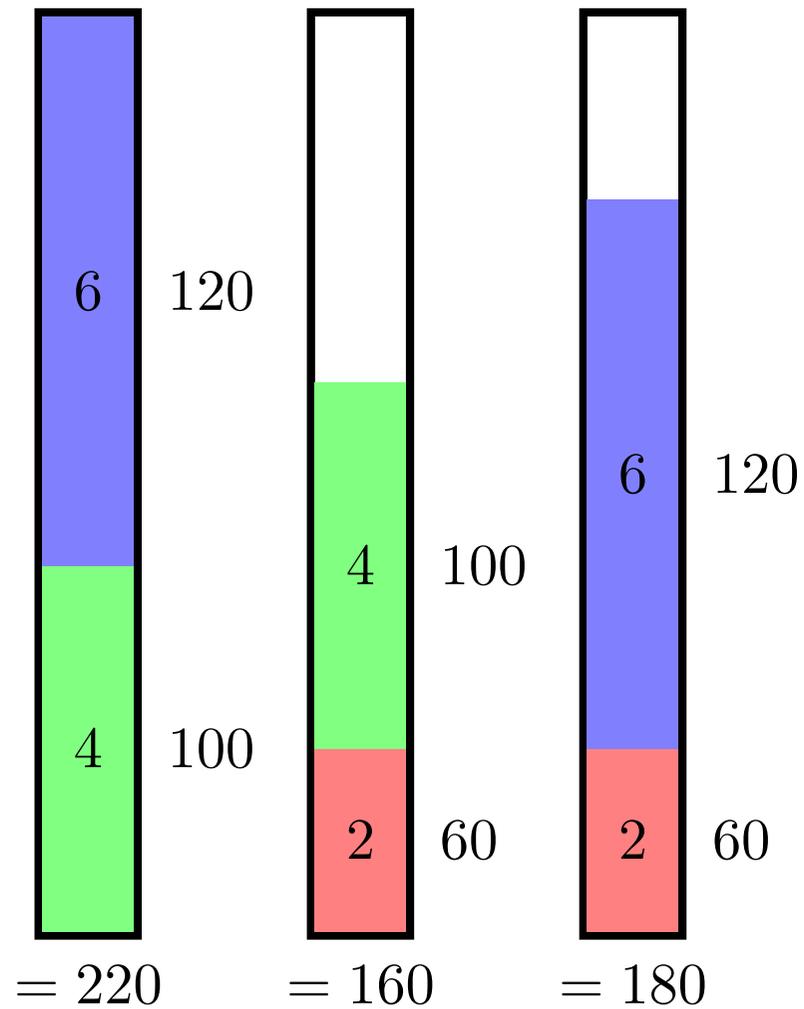
## Definition

Eingabe: Positive Integer  $(c_1, \dots, c_n)$ ,  $(v_1, \dots, v_n)$ , und  $C$

Ausgabe: Der Maximalwert  $v$ , so daß es eine Menge  $I \subseteq \{1, \dots, n\}$  gibt mit

- $\sum_{i \in I} c_i \leq C$
- $\sum_{i \in I} v_i = v$

# Knapsack



# 0/1-Knapsack

0/1-Knapsack kann durch *brute force* gelöst werden, indem alle Untermengen  $I \subseteq \{1, \dots, n\}$  aufgezählt werden und beide Bedingungen überprüft werden.

Es gibt allerdings  $2^n$  solche Untermengen. **Zu langsam!**

Können wir dieses Problem durch dynamisches Programmieren lösen?

Ja, indem alle Unterprobleme mit unterer Kapazität zuerst gelöst werden.

# 0/1-Knapsack

Betrachte den folgenden Algorithmus:

## Algorithmus

```
procedure Knapsack :  
for  $i = 1, \dots, n$  do  
  for  $k = 0, \dots, C$  do  
     $best[0, k] := 0;$   
     $best[i, k] := best[i - 1, k];$   
    if  $k \geq c_i$  and  $best[i, k] < v_i + best[i - 1, k - c_i]$   
    then  $best[i, k] := v_i + best[i - 1, k - c_i]$  fi  
  od  
od
```

$best[i, k]$  ist der höchste Wert, den man mit Gegenständen aus der Menge  $\{1, \dots, i\}$  erhalten kann, die zusammen Kapazität  $k$  haben.

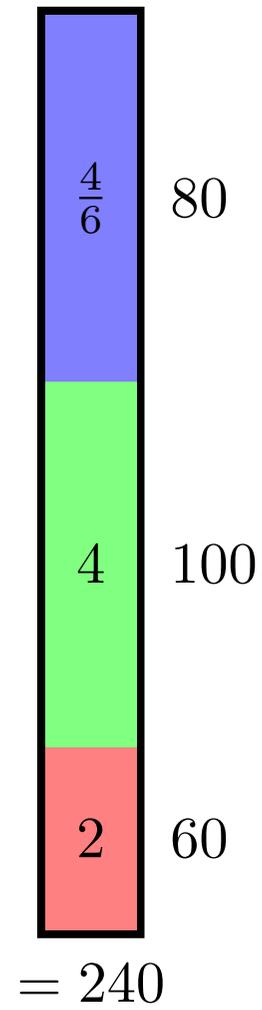
# Laufzeit

Die Laufzeit ist  $O(Cn)$ .

Wir nennen einen solchen Algorithmus **pseudo-polynomiell**.

Die Laufzeit ist **nicht** polynomiell in der Eingabelänge, aber polynomiell in der Eingabelänge **und** der Größe der Zahlen in der Eingabe.

Falls  $C$  klein ist, dann ist dieser Ansatz viel schneller als alle Untermengen durchzuprobieren.



# Greedy-Algorithmen

- Wie bei Dynamic Programming beinhalten optimale Lösungen **optimale Teillösungen**
- Anders als bei Dynamic Programming gilt die **greedy choice property**: eine lokal optimale Lösung ist stets Teil einer global optimalen Lösung
- Korrektheitsbeweise über Theorie der **Matroide**, **Greedoide**, **Matroideinbettungen**, ...

# Flußalgorithmen

Modelliere Optimierungsproblem als Flußproblem.

Varianten:

- Maximaler Fluß
- Fluß mit maximalen Gewinn
- Matchings maximaler Kardinalität
- Matchings maximalem Gewichtes

# Lineares Programmieren

- Viele Probleme können als Maximierung einer linearen Funktion mit linearen Ungleichungen als Nebenbedingungen ausgedrückt werden.
- Mehr dazu in der Vorlesung **Effiziente Algorithmen**

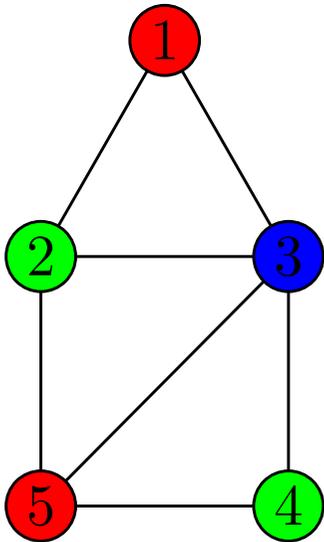
# Randomisierte Algorithmen

- Beispiele in der Vorlesung nutzen Zufall, um den *worst-case* in Form eines *adversary* mit hoher Wahrscheinlichkeit zu vermeiden, sowie um den Algorithmus und/oder die Analyse zu vereinfachen
- Eine Vielzahl von weiteren Techniken zum Entwurf in der Vorlesung **Randomisierte Algorithmen**

# Backtracking

- Tiefensuche auf dem Raum der möglichen Lösungen
- ein natürlicher Weg, schwere Entscheidungsprobleme zu lösen
- Ausdruck “backtrack” von D.H. Lehmer in den 50er Jahren
- Verfeinerung der *brute force*-Suche
- In der Praxis gut bei Problemen wie 3-Färbbarkeit oder 0/1-Knapsack

# Backtracking



## Algorithmus

```
function backtrack( $v_1, \dots, v_i$ ) :  
if ( $v_1, \dots, v_i$ ) is a solution then return ( $v_1, \dots, v_i$ ) fi;  
for each  $v$  do  
    if ( $v_1, \dots, v_i, v$ ) is acceptable vector then  
         $sol :=$  backtrack( $v_1, \dots, v_i, v$ );  
        if  $sol \neq ()$  then return  $sol$  fi  
    fi  
od;  
return ()
```

# Branch-and-Bound

- 1960 vorgeschlagen von A.H. Land und A.G. Doig für Linear Programming, allgemein nützlich für Optimierungsprobleme
- **branching** auf einem Suchbaum wie bei Backtracking
- **bounding** und **pruning** führt zum Auslassen von uninteressanten Zweigen
- Geeignet für Spiele wie Schach oder schwere Optimierungsprobleme

# Beispiel: 0/1-Knapsack

- Verzweige, je nachdem, ob ein Gegenstand mitgenommen wird oder nicht, in einer festen Reihenfolge
- Eine Teillösung  $I \subseteq \{1, \dots, k\}$  ist zulässig, solange 
$$\sum_{i \in I} c_i \leq C$$
- Wenn  $I$  nicht zulässig ist, kann auch keine Lösung, die  $I$  enthält, zulässig sein
- Außerdem ist  $\sum_{i \in I} v_i + \sum_{i \in \{k+1, \dots, n\}} v_i$  eine obere Schranke für den Profit jeder Lösung, die  $I$  erweitert, wenn wir in  $I$  alle Gegenstände bis zum  $k$ . berücksichtigt haben
- Also können wir alle Zweige abschneiden, die entweder nicht zulässig sind oder deren obere Schranke unter dem bis jetzt optimalen Profit liegt.

# Das Problem des Handlungsreisenden

Wir untersuchen das folgende Problem:

## Definition (TSP)

Eingabe: Ein Graph  $G = (V, E)$  mit Kantengewichten

$length : E \rightarrow \mathbf{Q}$

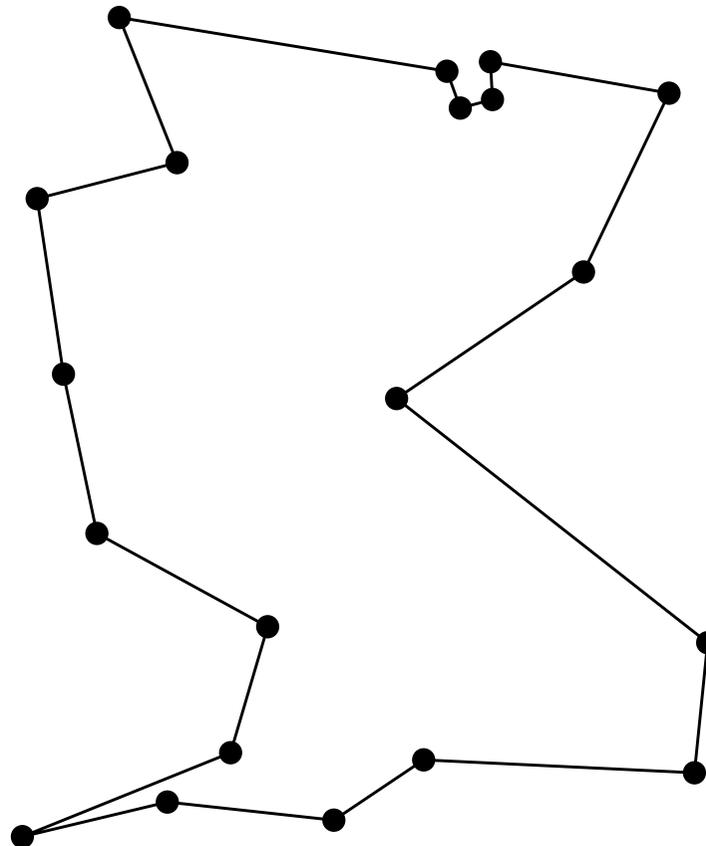
Ausgabe: Eine geschlossene Rundreise über alle Knoten mit minimaler Gesamtlänge.

Wir denken zum Beispiel an  $n$  Städte, die alle besucht werden müssen.

Was ist die minimale Länge einer Tour, die alle Städte besucht und im gleichen Ort beginnt und endet?

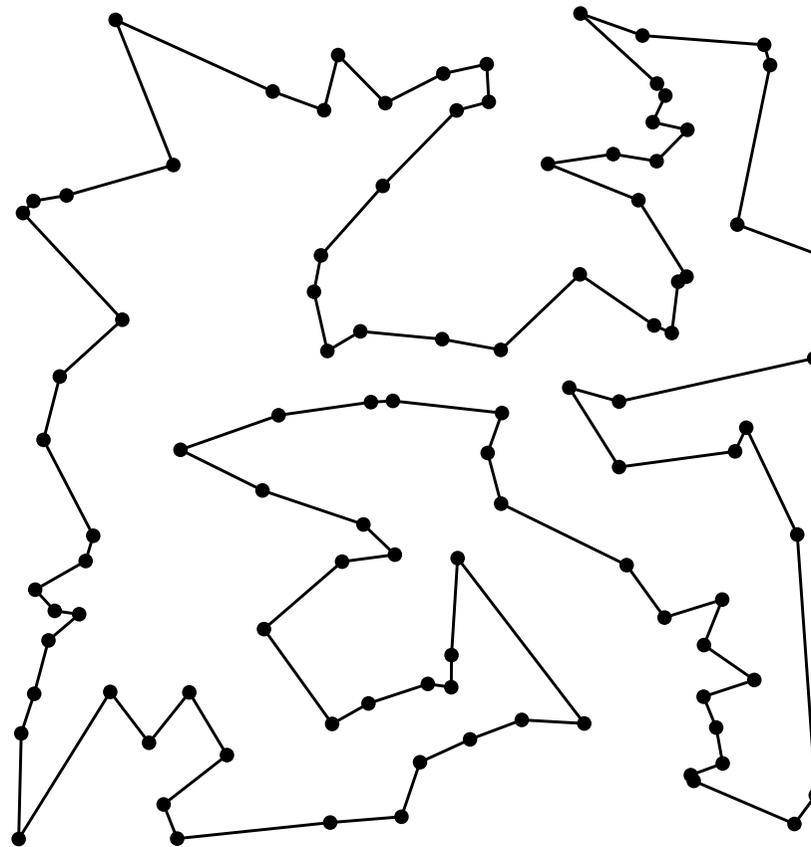
# Das Problem des Handlungsreisenden

Ein kleines Beispiel:



# Das Problem des Handlungsreisenden

Ein mittelgroßes Beispiel:



# Der naive Ansatz

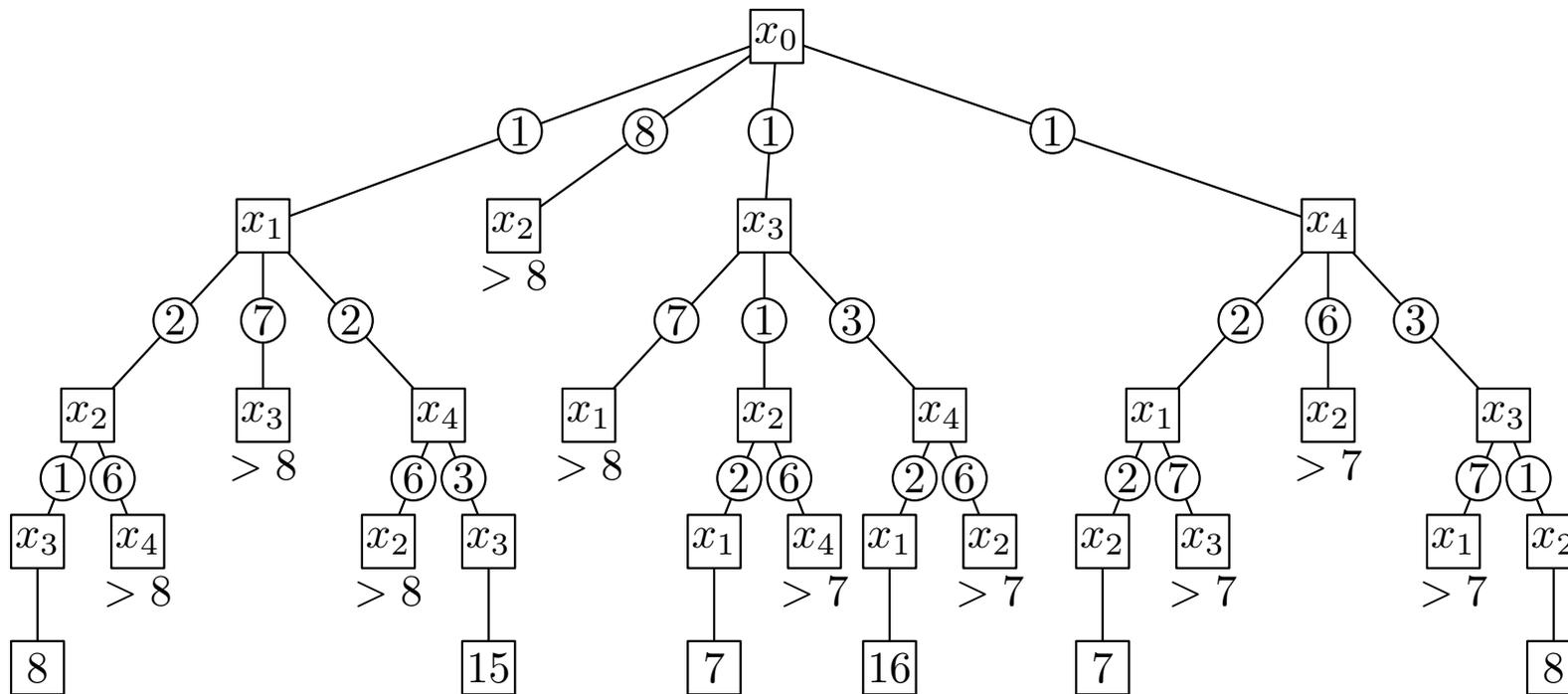
Der einfachste Ansatz dieses Problem zu lösen besteht darin, **alle Möglichkeiten durchzuprobieren**.

Wieviele Möglichkeiten gibt es?

Es gibt  $(n - 1)!$  mögliche Rundreisen, da es  $n!$  Permutationen gibt. Wir könnten alle Rundreisen durchprobieren, ihre Kosten berechnen und die billigste auswählen.

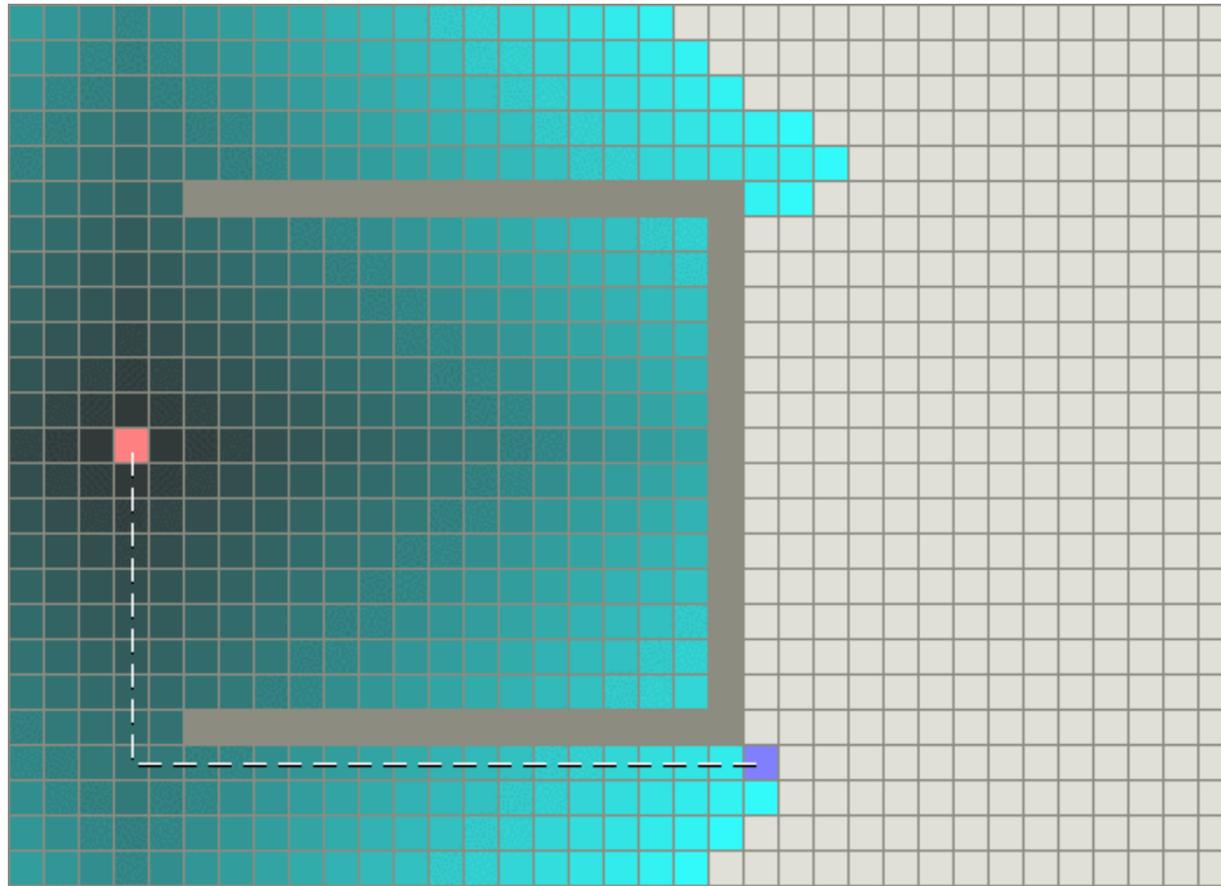
Es eine schnellere Lösung mit Hilfe von **dynamischer Programmierung**. Wir versuchen es mit Branch-and-Bound.

# Branch and Bound - Beispiel TSP



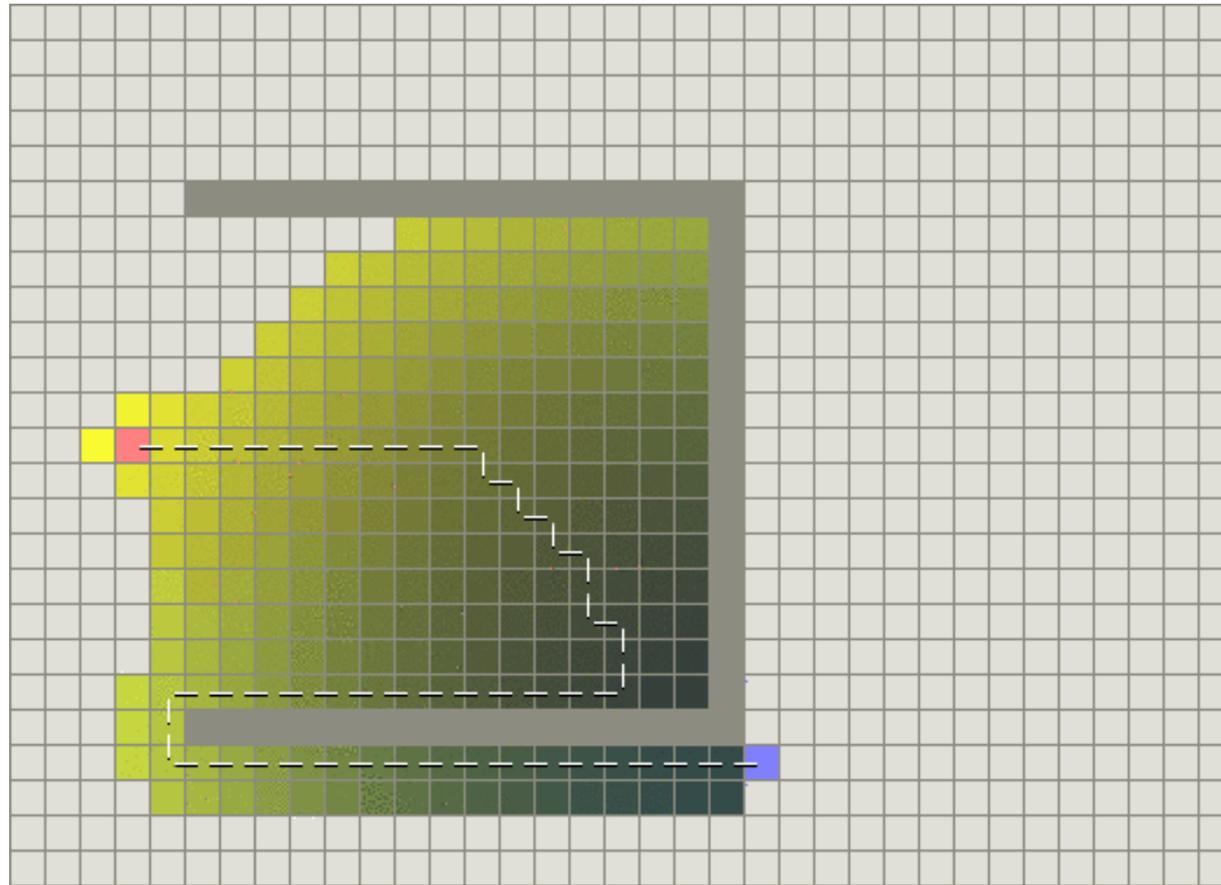
# A\*-Suche

- 1968 von Peter Hart, Nils Nilsson, and Bertram Raphael als A-search (A\*-search mit optimaler Heuristik)
- Graphsuchverfahren mit Anwendungen in KI (Planung) und Suche in großen Suchräumen
- Generalisierung von Dijkstras Algorithmus und best-first search
- Knotenexpansion in Reihenfolge von  $f(x) = g(x) + h(x)$  mit tatsächlichen Kosten  $g$  und **Heuristik**  $h$



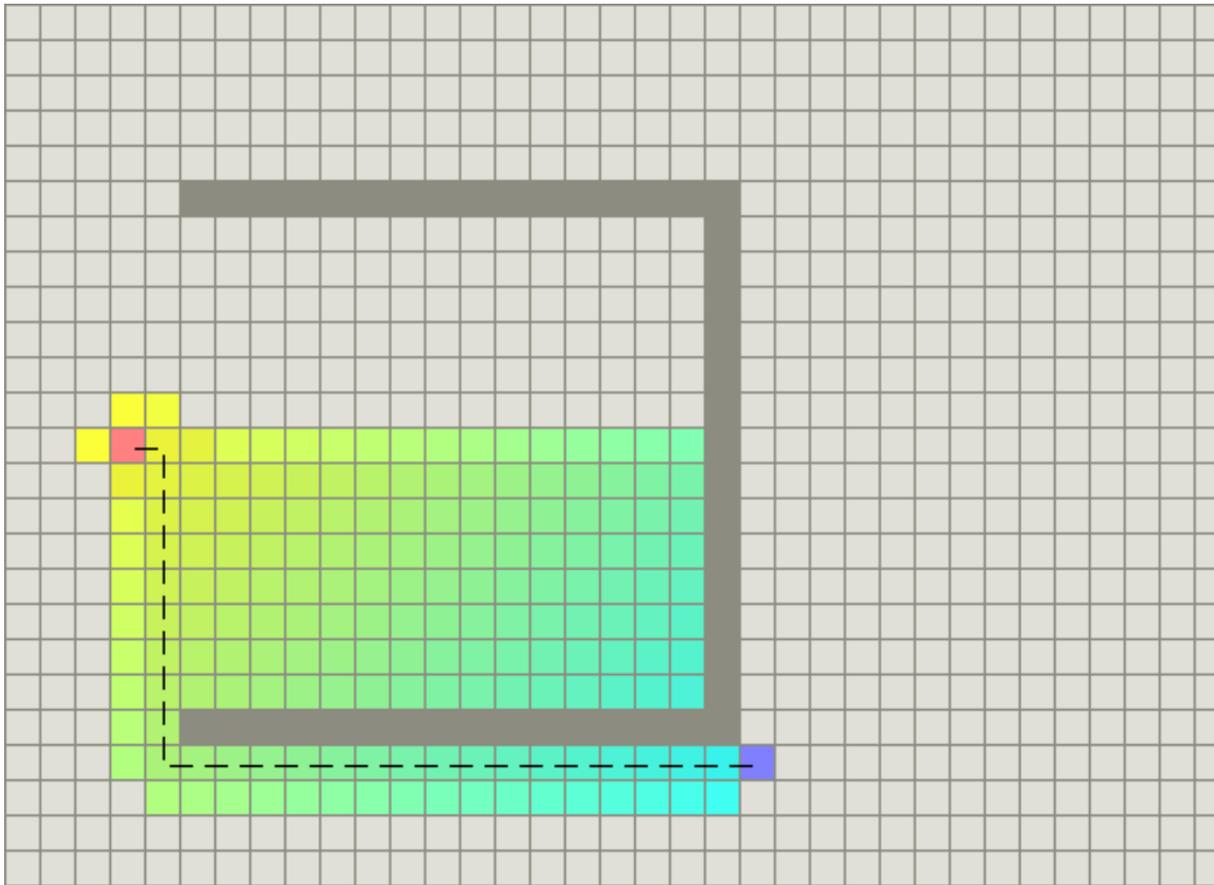
Dijkstra

Bilder von Amit Patel/Stanford



## Best-First-Search

Bilder von Amit Patel/Stanford



A\*

Bilder von Amit Patel/Stanford

## Algorithmus

**procedure** *a* – *star*(*s*) :

*open* := **new** priority queue

*open.enqueue* *s*;

*closed* :=  $\emptyset$

**while** (*true*) **do** *current* := *open.extract\_min*;

**if** *current* = *goal* **then break fi**; *closed.insert* (*current*);

**for** all neighbors of *current* **do**

*cost* = *g*(*current*) + *weight*(*current*, *neighbor*);

**if** *cost* < *g*(*neighbor*) **then**

*open.remove*(*neighbor*); *closed.remove*(*neighbor*) **fi**;

**if!***open.iselement*(*neighbor*) **and!***closed.iselement*(*neighbor*)

**then** *g*(*neighbor*) := *cost*; *neighbor.parent* := *current*;

*open.enqueue*(*neighbor*) with rank *g*(*neighbor*) +

*h*(*neighbor*);

**fi**

**od**

**od**

# Eigenschaften der A\*-Suche

## Theorem

*Der A\*-Algorithmus findet einen optimalen Pfad, wenn  $h$  optimistisch ist.*

## Beweis.

Wenn der Algorithmus terminiert, ist die Lösung billiger als die optimistischen Schätzungen für alle offenen Knoten. □

# Eigenschaften der A\*-Suche

## Theorem

*Für jede Heuristik  $h$  betrachtet die A\*-Suche die optimale Anzahl von Knoten unter allen zulässigen Algorithmen.*

## Beweis.

Nimm an, Algorithmus B mit Heuristik  $h$  betrachtet einen Knoten  $x$  nicht, der in A\* einmal offen war. Dann beträgt  $h(x)$  höchstens Kosten des Lösungspfades von B und kann B nicht ausschließen, daß es einen Pfad über  $x$  gibt, der billiger ist. Also ist B nicht zulässig. □

# A\* - Kosten

- Laufzeit: Im allgemeinen Anzahl der betrachteten Knoten exponentiell in der Länge des optimalen Pfades
- stark abhängig von der Güte der Heuristik: polynomiell wenn  $|h(x) - h^*(x)| \in O(\log h^*(x))$
- Problem: Speicherplatz ebenfalls exponentiell
- Varianten/Verbesserungen: IDA\*, MA\*, SMA\*, AO\*, Lifelong Learning A\*, ...

# Heuristiken

- Hier: Verfahren, die in der Praxis oft funktionieren, aber für die man keine worst-case-Schranken zeigen kann
- Beispiele: Lokale Suche, Simulated Annealing, Genetische Algorithmen

# Lokale Suche

- Heuristik für Optimierungsprobleme
- Definiere Nachbarschaft im Suchraum
- Gehe jeweils zum lokal optimalen Nachbarn
- Problem: lokale Minima

# Lokale Suche für TSP

