UNIVERSITÄT
LEIPZIG

Algorithmen und Datenstrukturen II

Vorlesung *und*1*c*1

Leipzig, 11.06.2024

Peter F. Stadler & Thomas Gatter & Ronny Lorenz

RANDOMISIERTE ALGORITHMEN

Randomisierte Algorithmen

Grundidee:

- In vielen Fällen führen zufällige Entscheidungen zu sehr guten Ergebnissen.
- **“Las Vegas Algorithmus”**: Randomisierte Algorithmen, die nie ein unoptimales oder falsches Ergebnis liefern.
  - z.B. Quicksort mit zufälligem Pivot, Gomory Hu …
- **“Monte Carlo Algorithmus”**: Randomisierte Algorithmen, durch die keine perfekte oder optimale Lösung garantiert wird.
  - Wenn randomisierte Algorithmen mit einer gewissen Wahrscheinlichkeit zum Ziel führen, kann man durch hinreichend oftmalige Wiederholung *fast sicher* die gewünschte Lösung erhalten.
  - z.B. Karger’s Algorithmus

Karger's Algorithmus: Minimale Schnitte I

Beispielanwendung

In einem ungerichteten Graphen G, finde einen Schnitt mit minimalem Kantengewichten. Hier wollen wir Quelle und Senke nicht fixieren, sondern alle möglichen Paare von Quellen und Senken verbinden.

- **Startpunkt**: Kantenkontraktionen (von Kanten, die nicht im betrachteten Schnitt liegen) erhalten das Gewicht von Schnitten.
  siehe Konstruktion von Gomory-Hu Bäumen.

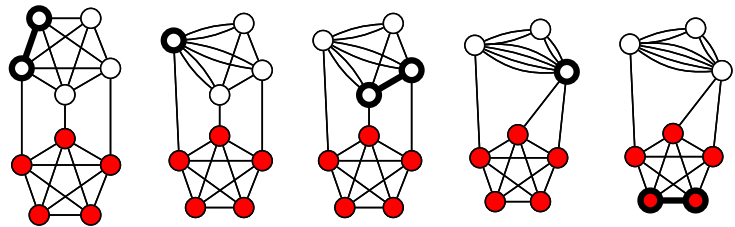
Gegeben Quelle *s* und Senke *t* in *G*, kontrahiere **zufällig** Kanten, solange bis nur mehr zwei Knoten *x* und *y* und eine Kante dazwischen übrig ist.

Karger's Algorithmus: Minimale Schnitte II

- **Die zur Kontraktion gewählte Kanten und ihre Reihenfolge entsprechen einem bestimmten Schnitt $V_1 | V_2$**
  - $\rightarrow$ Alle Knoten in V_1 wurden zu x kontrahiert, und alle Knoten in V_2 zu y.
  - $\rightarrow$ Die verbleibende Kante entspricht dem Schnitt zwischen V_1 und V_2.
- Mit einer gewissen (kleinen) Wahrscheinlichkeit hat man die Kontraktions-Reihenfolge *zufällig so gewählt*, dass der erzeugte Schnitt derjenige mit *minimalem Gewicht* ist.

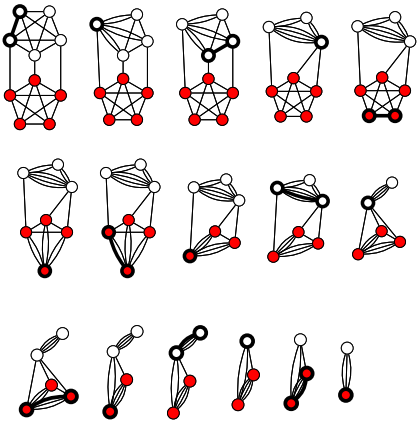
Karger's Algorithmus: Minimale Schnitte III

Der Einfachheit halber betrachten wir ein ungewichtetes Beispiel
Es gilt $w(e) = 1$ für alle Kanten.



- In jedem Schritt wird eine zufällige Kante (dick markiert) kontrahiert.
- Die Gewichte von dadurch entstehenden parallelen Kanten werden addiert.
  Im Beispiel ist das Gewicht einfach durch die Zahl der parallelen Kanten dargestellt.

Karger's Algorithmus: Minimale Schnitte IV



Analyse von Karger's Algorithmus I

Mit einer gewissen Wahrscheinlichkeit ist der erzeugte Schnitt derjenige mit minimalem Gewicht:

Der Kontraktionsalgorithmus liefert am Schluss den minimalen Schnitt, genau dann wenn die zufällig gewählte Kante in keinem Schritt im minimalen Schnitt liegt.

Man erhält einen Schnitt, indem man alle Kanten, die mit einem Knoten inzident sind, durchtrennt. Also ist die Zahl der Kanten in einem minimalen Schnitt $c(S)$ nicht größer als der minimale Knotengrad, und damit auch nicht größer als der mittlere Knoten $\bar{d} = 2|E|/|V|$.

Betrachtet also einen Kontraktionsschritt:

Die zufällig gewählte Kante liegt mit einer Wahrscheinlichkeit von höchstens $p \leq c(S)/|E| \leq 2/|V|$ quer über den minimalen Schnitt.

Analyse von Karger's Algorithmus II

Wahrscheinlichkeit, dass der finale Schnitt der minimale (S) ist:
$\rightarrow \quad P(1.\text{ Kante} \notin S) \times P(2.\text{ Kante} \notin S) \times ...$

$$
\begin{aligned}
&\geq \left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right)\left(1-\frac{2}{n-2}\right)\cdots\left(1-\frac{2}{4}\right)\left(1-\frac{2}{3}\right) \\
&= \frac{n-2}{n}\frac{n-3}{n-1}\frac{n-4}{n-2}\cdots\frac{2}{4}\frac{1}{3} \\
&= \frac{2}{n(n-1)}
\end{aligned}
$$

In jedem Durchlauf findet man die optimale Lösung mit Wahrscheinlichkeit $p \geq 2/n^2$.

Analyse von Karger's Algorithmus III

Die Wahrscheinlichkeit ε, dass man optimale Lösung nach M Durchläufen immer noch nicht gefunden hat is daher höchstens

$$\varepsilon \leq (1 - 2/n^2)^M \approx e^{-2M/n^2}$$

(wegen $e = \lim_{x\to\infty}(1 + 1/x)^x$)

Die Zahl M der Durchläufe, die ausreichen um den minimale Schnitt mit einer fix vorgegebenen Restfehlerwahrscheinlichkeit $\varepsilon \ll 1$ zu finden ist also

$$M \leq (-\ln \varepsilon/2)n^2$$

D.h $\Omega(n^2)$ Durchläufe reichen aus.

STOCHASTISCHE OPTIMIERUNGSALGORITHMEN

Optimierungsalgorithmen

Problemstellung: Optimierung

Lösungsmenge X, Bewertungsfunktion $f : X \to \mathbb{R}$.
Finde globales Minimum von f

Die Natur von X bestimmt welche Methoden wir nutzen können:

- diskret vs. stetig
- endlich vs. unendlich dimensional
- statisch vs. dynamisch
- Constraints (Positivität, Energieerhaltung, Wegpunkte, Produktionsziele, Kolissionsvermeidung, ...)
- …

Bewertungsfunktion

Auch die Bewertungsfunktion beeinflusst unsere Möglichkeiten:

- linear, polynomisch, exponentiell, ...
- differenzierbar?
- convex?
- ...

Es gibt zahlreiche Familien von Methoden die (fast) alle Kombinationen abdecken: Linear programming, Quadratic programming, Fractional programming, Constraint satisfaction, Stochastiche Optimierung, ...

Wir wollen uns hier Stochastische Optimierungsalgorithmen etwas genauer anschauen.

Stochastische Optimierungsalgorithmen

Stochastische Optimierungsalgorithmen

Optimierungsalgorithmen, die zufällige Entscheidungen zur Bestimmung einer möglichst guten Lösung verwenden.

Auch hier gibt es eine große Auswahl etablierter, komplexer Methoden:

- Stochastic Gradient Descent (z.B. für Neuronale Netze)
- (Stochastic) Expectation-Maximization
- Schwarmalgorithmen
- Evolutionäre Algorithmen
- ...

Abgrenzung zu Machine Learning

Oft ist es schwierig richtiges Verhalten von Algorithmen direkt zu definieren...

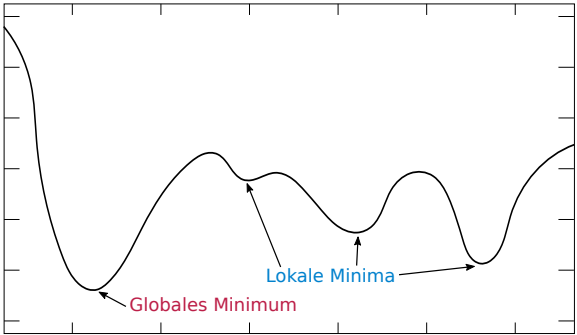
Algorithmen lernen das richtige Verhalten selbst aus gegebenen Daten.

- **Überwachtes (Supervised) Lernen**: Lernen von Daten mit Labeln zu positivem (und negativen) Verhalten
  z.B. Klassifikation, Regression
- **Verstärkendes (Reinforced) Lernen**: Progressives Lernen basierend auf Daten aus der Interaktion mit der Umwelt
  z.B. Softwareagenten, Sprachproduktion und -erkennung
- **Unüberwachtes Lernen**: Suchen von “interessanten Mustern” in Daten ohne Label
  z.B. Density estimation, Clustering

Lernen in diesem Kontext wird (fast) immer als Optimierung beschrieben. Der Fokus ist hier jedoch auf die Anwendung selbst verschoben. Echte Funktionen die X unterliegen sind meist unbekannt. Das Design von Bewertungsfunktion um trotzdem das Richtige zu lernen ist ein wesentlicher Teil der Forschung.

Globale vs. Lokale Minima

Häufig können wir das globale Minimum nicht exakt/garantiert bestimmen, z.B. weil keine Polynomial-Zeit Algorithmen dafür bekannt sind (TSP, 0/1 Rucksack, ...) oder X aus anderen Gründen nicht enumeriert werden kann.



Methoden müssen also so designed werden, dass sie **lokale Minima** vermeiden.

Grundprinzip

Initialisiere endliche Menge A von erlaubten Lösungen

wiederhole

> Erzeuge eine erweiterte Lösungsmenge B, sodass $|B| > |A|$
> Selektiere die besten Lösungen $A' \subset B$, $|A'| = |A|$
> $A \leftarrow A'$

bis *Abbruchbedingung erreicht*;

Die Wahl von B und A' bestimmen maßgeblich das Optimierungsverhalten.

Im folgenden betrachten wir **diskrete, endliche, statische** Lösungsmengen X.

Zufällige Optimierung

Random Generate and Test

wähle $x \in X$ als eine einzelne **zufällige** Lösung

wiederhole

  wähle $y \in X$ als eine **zufällige** Lösung
  if $f(y) < f(x)$ **then**
    $x = y$
  else
    x bleibt unverändert

bis *z.B. x bleibt über hinreichend Iterationen stabil*;

***Keine** Garantie, dass ein **globales Minimum** gefunden wird!*

Wie kann man die *Struktur* der Bewertungsfunktion f ausnutzen?

Gute Lösungen werden “irgendwie” ähnlich zu anderen guten Lösungen sein.

Fitness-Landschaften

Im Allgemeinen möchten wir die Wahl für Elemente in B “lokal” begrenzen.

- wir definieren eine (erlaubte) Nachbarschaft $N(x)$ für jedes $x \in X$
- diese impliziert eine Graph-Struktur Γ auf X: $(x, y) \in E$ iff $y \in N(x)$

Wir betrachten das Optimierungsproblem als Funktion über der Knotenmenge des Graphen Γ. Die Frage nach geometrischen Eigenschaften wie

- Zahl von lokalen Minima
- Tiefe und Breite von Tälern
- Sattelpunkte zwischen Tälern
- Rauhheits-Maße

und deren Einfluss auf die Optimierung ist Gegenstand der Forschung.

Fitness-Landschaften: Bedingungen

Wir Schränken B auf die Elemente der aktiven Nachbarschaft ein.
Entsprechend gibt es für die Wahl von $N(\cdot)$ einiges zu beachten...

- Änderungen der Nachbarschaften verändern auch die Landschaft und damit das Optimierungsverhalten.
- Einzelne Optimierungsschritte sollten die Qualität der Lösung (meistens) nicht zu drastisch verändern.
- Γ muss stark zusammenhängend sein sonst kann der Prozess auf einer Zusammenhangskomponente eingesperrt bleiben, die kein globales Minimum enthält.

Wie nutzen wir die Nachbarschaft nun effektiv?

Adaptive Walk

wähle $x \in X$ als eine einzelne **zufällige** Lösung

wiederhole

  wähle $y \in N(x)$ gleichverteilt (mit Wkt. $1/|N(x)|$)

  if $f(y) < f(x)$ **then**

    $x = y$

  else

    x bleibt unverändert

bis $\forall y \in N(x) : f(x) \leq f(y)$;

- Folge der Lösungen enspricht einem Pfad in Γ, entlang dessen f strikt abnimmt.
- *Lokales Minimum:* $f(y) \geq f(x)$ für alle $y \in N(x)$.
- Pfade bleiben in lokalen Minima stecken.

Problem

Lösungen können sehr schlechter Qualität sein, d.h. $f(x) \gg f(\min)$.

Häufig gilt: **Je tiefer** das Lokale Minima, **desto steiler** der Anstieg die Werte $f(\cdot)$ in dessen Nachbarschaft in Γ.

Gradient Descent

wähle $x \in X$ als eine einzelne **zufällige** Lösung

wiederhole

  $Z(x) = \{z \mid z \in N(x), f(z)$ minimal über $N(x)\}$

  wähle $y \in Z(x)$ gleichverteilt (mit Wkt. $1/|Z(x)|$)

  if $f(y) < f(x)$ **then**

    $x = y$

  else

    x bleibt unverändert

bis $f(x) \leq f(y)$;

- Folge der Lösungen enspricht einem Pfad in Γ, entlang dessen f so steil wie möglich abnimmt. (Weg des steilsten Abstieg, “Gradient Descent”)
- Endet immer in einem lokalen Minimum.
- Teurer als adaptive walk, weil $N(x)$ aufgelistet und bewertet werden muss.

Um lokalen Minima zu entkommen, lasse auch “aufwärts” Schritte zu.

Metropolis-Walks

wähle $x \in X$ als eine einzelne **zufällige** Lösung

wiederhole

  wähle $y \in N(x)$ gleichverteilt (mit Wkt. $1/|N(x)|$)

  wähle $r \in [0, 1]$ gleichverteilt

  if $f(y) < f(x)$ ***or*** $r \leq \exp(-(f(y) - f(x))/T)$ **then**

    $x = y$

  else

    x bleibt unverändert

bis *z.B. x bleibt über hinreichend Iterationen stabil*;

- *Steil bergauf* = exponentiell selten
- **Eigenschaft:** Wenn man **lange** genug wartet, besucht man zu einem bestimmten Zeitpunkt t die Lösung x mit Wahrscheinlichkeit

$$p(x) = \exp(-f(x)/T)/Z$$

- Für kleine T: Lösungen mit kleinen Funktionswerten werden angereichert.
- Normalisierungskonstante $Z = \sum_{x \in X} exp(f(x)/T)$

Simulated Annealing

Reduziere die “Temperatur” T im Laufe der Simulation

Kirkpatrick, Gelatt and Vecchi (1983) and V. Černý (1985) *(Wikipedia-Link)*

- **Wichtige Rolle:**
  “cooling schedule” T_k, typischerweise eine monoton fallende Funktion der Schrittzahl k
- Der Algorithmus konvergiert falls $T_k \geq d/\log(k)$ für große k.
- Das ist in der Praxis aber zu langsam, man braucht exponentiell viele Schritte k um eine gegebene Genauigkeit $|f(x) - f(\min)|$ zu erreichen.
- Anwendung auf harte Probleme, für die keine effizienten (polynomialen) Algorithmen bekannt sind.

Genetische Algorithmen

Nachahmung natürlicher Evolutionsprozesse zur Optimierung.

- Die Erbinformation von biologischen Organismen (**Individuen**) ist in **Chromosomen** organisiert.
- Jedes Chromosom besteht aus einer linearen Folge von **Genen**.
- **Nachkommen** entstehen aus der **Rekombination** der Gene zweier Eltern
- ... oder als direkte Kopie eines Elters.
- Die Gene von **Nachkommen** weichen von ihren Eltern durch spontane **Mutationen** ab.

Wir übetragen diese Konzepte auf Optimierungprobleme.

Für Interessierte: *Wikiversity-Kurs zu genetischen Algorithmen*

Biologie vs. Simulation

- **Kodierung:** explizite (bijektive) Repräsentation der Lösungen
  - jede Repräsentation bestimmt ein eindeutiges Individuum $x \in X$
  - jedes Individuum $x \in X$ hat eine eindeutige Repräsentation
  - Individuen können damit zufällig generiert werden
- **Rekombination:** erzeuge Nachkommen aus zwei Eltern
  - Funktion $R : X \times X \to X$ oder $R : X \times X \to \{X\}$
  - zufällige Kombination der Features/Gene beider Eltern
- **Mutation:** verändere Eigenschaften einzelner Nachkommen
  - Funktion $M : X \to X$
  - zufällige Änderungen von Features/Genen
- **Fitnessfunktion und Selection:**
  - Bewertungsfunktion $S_f : \{X\} \to \{X\}$ von Individuen basierend auf f
  - Auswahl von Individuen für die nächste Generation basierend auf deren Fitness

Jede Rekombination und Mutation muss erneut ein valides Individuum erzeugen.

Grundprinzip - Genetische Algorithmen

Initialisiere endliche Menge A (“Population”) aus zufälligen Individuen

wiederhole

$B = \emptyset$

`// (1) Wähle Eltern und erzeuge mind. 1 Nachkommen pro Paar`
Wähle eine zufällige Menge von Eltern-Paaren $P \subseteq A \times A$
for $(x, y) \in P$ **do**
  $B = B \cup R(x, y)$

`// (2) Erzeuge mutierte Nachkommen einzelner Individuen`
for $x \in Q \subseteq A$ **do**
  $B = B \cup M(x)$

`// (3) Übernehme optional unveränderte Individuen in die nächste Generation`
$B = B \cup S$, mit $S \subseteq A$

`// (4) Selektiere Nachkommen basierend auf deren Fitness`
$A = S_f(B)$

bis *Abbruchbedingung erreicht*;

Kodierung

- **Strings**: z.B. Alignments, Spielzüge (Schach, Go, ...)
- **Permutationen:** z.B. TSP
- **Binäre Vektoren:** z.B. 0/1 Rucksack
- **Reell-wertige Vektoren:** z.B. Funktionsapproximation, parametrische (geometrische) Objekte[1]
- **Bäume:** z.B. Funktionale Programme, Raumplanung für Platinen
- **Partitionen:** z.B. finden von Teilmengen gleicher Wertigkeit
- **Matchings:** z.B. Bilderkennung

Mehrere Kodierungen können gemeinsam für ein Individuum als verschiedene “Chromosome” verwendet werden.

[1] z.B. optimierte Fahrzeuge: **https://rednuht.org/genetic_cars_2/**

Mutation und Rekombination

Methoden zur Rekombination R und Mutation M basieren im allgemeinen auf der Kodierung von X.

Wir schauen uns daher 2 Beispiele aus der Vorlesung an:

0/1-Rucksackproblem

und

TSP

Genetisches 0/1-Rucksackproblem

Zur Erinnerung:

- n Objekte, mit Volumina $t_1, t_2, \ldots, t_n$ und Werten $p_1, p_2, \ldots, p_n$
- Rucksack hat Gesamtvolumen c.
- finde einen $0/1$-Vektor $a_1, ..., a_n \in \{0, 1\}$ mit

$$\sum_{i=1}^{n} a_i t_i \leq c \quad \text{sodass} \quad f(a) = \sum_{i=1}^{n} a_i p_i \to \max$$

Der Vektor a ist bereits eine ideale Kodierung.

Wir bestimmen die Fitness wie folgt:

- Falls $\sum_{i=1}^{n} a_i t_i \leq c$, setze Fitness auf $f(a)$
- Sonst setze Fitness auf 0

Cross-Over

1-Punkt Crossover

```
x1 = 01000010101000101010101101
x2 = 10101011101010100111001101
```

Bestimme einen zufälligen *Bruchpunkt k* und vertausche die Suffixe:

$x'_1 = x_1[1..k]x_2[k+1..n]$ und $x'_2 = x_2[1..k]x_1[k+1..n]$

Für $k = 7$

```
x1' = 0100001.1101010100111001101
x2' = 1010101.0101000101010101101
```

Der Punkt kennzeichnet die Bruchstelle und ist nicht Teil des Strings

Uniformer Crossover

Übernimm jede Position zufällig entweder unverändert oder vertausche die beiden Zeichen zwischen Eltern

```
x1   =  01100011101010101011001001O1
x2   =  10001010101000100110101111O1
tausch --**-****--*******---**---**-*
x1'  =  01000010101000101010101101
x2'  =  10101011101010100111001101O1
```

$x'_1 = x_1$ and $x'_2 = x_2$ für -, $x'_1 = x_2$ and $x'_2 = x_1$ für $*$

Bit-Flip Mutationen

Für binäre Vektoren können wir jede Position in das Gegenteil umkehren.

$$\text{Bit-Flip : bitflip}(x) = \begin{cases} 0 & \text{if } x == 1 \\ 1 & \text{if } x == 0 \end{cases}$$

Anwendung:

- Definiere eine (globale) Mutationswahrscheinlichkeit m mit der ein Gen/eine Position mutiert.
- Wähle ein zu mutierendes Individuum.
- Wende bitflip(x) mit Wahrscheinlichkeit m auf jede Position an.

```
x1   =   01100011101010101011001001O1
Mutation ----m-------m------------m--
x1'  =   01101011101000101011001000O1
```

Genetisches Traveling Salesman Problem (TSP)

Zur Erinnerung:

- n Städte mit paarweise Distanzen: d_{ij} Entfernung von Stadt i nach Stadt j.
- finde Rundreise mit minimaler Länge durch alle Städte,
  also Permutation $\pi : (1, ..., n) \rightarrow (1, ..., n)$, für die $c(\pi)$ minimal ist.

$$c(\pi) = \left[\sum_{i=1}^{n-1} d_{\pi(i)\pi(i+1)} \right] + d_{\pi(n)\pi(1)}$$

Die Permutation π ist eine mögliche Kodierung. Maximiere $-c(\pi)$ als Fitness.

Crossover für Permutationen

Wir müssen jede Stadt genau ein mal besuchen. Die bisherigen Crossover-Methoden garantieren das nicht.

Geordneter Crossover

Wähle ein zufäliges Interval eines Elters und übernimm dieses unverändert. Fülle die freien Positionen in der Reihenfolge des 2. Elters auf. Ist eine Position des 2. Elters Teil des zufälligen Intervals, so wird diese ausgelassen und die nächste Postion übertragen, für die dies nicht gilt.

```
x1  =  123456789
x2  =  928165743
x'  =  921456783
```

Mutation mittels Positionstausch

Auch für die Mutation müssen die Permutationseigenschaften und alle Städte erhalten bleiben. Wir können also nicht einfach nur 1 Position ändern.

Stattdessen:

- Wähle ein zu mutierendes Individuum.
- Wähle 2 zufällige Positionen.
- Tausche die Werte von beiden Positionen.
- Individuen können ggf mehrfach so mutiert werden.

```
x  =  123456789
x' =  128456739
```

Was gibt es sonst noch?

Crossover für Vektoren

Mittelwertbildung:

$$z = \frac{1}{2}(x + y)$$

Konvexe Kombination:

$$z = px + (1 - p)y \qquad p \in [0, 1]$$

Die Zufallszahl p wird typischerweise aus einer Gleichverteilung gezogen

Spezielle Verfahren für andere Typen von Objekten, wie z.B., Permutationen, Bäume …

Selection

Es gibt viele Varianten:

- fitness-proportional:
  Ziehe aus B mit Wahrscheinlichkeiten proportional zu $-f(z)$ oder $\exp(-f(z))$ bis genau n Lösungen gezogen sind
- ziehe zwei Lösungen z und z'. Falls $z > z'$, akzeptiere z. Wiederhole bis n Lösungen akzeptiert sind.
- Wähle die besten n Lösungen.
- Wähle die besten $n - x$ Lösungen und x zufällige Lösungen.
- ...

Abbruchbedingungen

– *Normalerweise ist nicht bekannt, ob das Ziel bereits erreicht ist, d.h., ob ein globales Minimum bereits gefunden ist.*

→ Fixe Anzahl von Iterationen

→ Keine Verbesserung seit K Iterationen

→ Verbesserung in den letzten K Iterationen weniger als ϵ
(im Fall von reell-wertigen Problemstellungen)

Zusammenfassung

- randomisierte Algorithmen werden gerne für Probleme ohne *bekannte polynomielle* Lösung benutzt
- bei Problemen mit vielen lokalen Optima kann das Finden einer global optimalen Lösung nicht garantiert werden (in endlicher Zeit)
- In vielen Fällen kann man *statistische Garantien* geben, z.B., Abschätzungen der Wahrscheinlichkeit, dass der Algorithmus versagt. (siehe Beispiel Karger)
- Oft ist es nicht möglich solche Schranken zu finden, z.B., im Fall von genetischen Algorithmen. Oder sie helfen wenig, wie im Fall von Simulated Annealing (wo man exponentiell viele Schritte für eine garantierte Lösung benötigt).