

7. Advanced Designs

Auswahl algorithmischer Entwurfsmethoden

Divide & Conquer

Löse rekursiv (disjunkte) Teilprobleme

Siehe [Quicksort](#), [Merge Sort](#)

Backtracking

Durchsuche iterativ Lösungsraum

Siehe [Backtracking](#)

Dynamisches Programmieren

Löse rekursiv (überlappende) Teilprobleme durch Wiederverwenden/Speichern

Siehe [Dynamische Programmierung](#)

Greedy

Baue Lösung aus Folge lokal bester Auswahlen zusammen

Siehe [Greedy-Algorithmen](#), [Kruskal](#), [Prim](#), [Dijkstra](#)

Metaheuristiken

Übergeordnete Methoden für Optimierungsprobleme

Siehe [Metaheuristiken](#)

Backtracking

Prinzip

Finde Lösungen $x := (x_1, x_2, \dots, x_n)$ per "Trial-and-Error", indem Teillösung $(x_1, x_2, \dots, x_{i-1})$ durch Kandidaten x_i ergänzt wird, bis Gesamtlösung erreicht ist, oder bis festgestellt, dass keine Gesamtlösung erreichbar ist, dann wird Kandidat x_{i-1} revidiert

Beispiel: Sudoku

```
SUDOKU-BACKTRACKING(B) // B[0...3][0...3] board
    IF isFull(B) THEN
        print "solution: "+B;
    ELSE
        (i,j)=nextFreePos(B);
```

```

FOR v=1 TO 4 DO
    IF isAdmissible(B,i,j,v) THEN // no rules broken?
        B[i,j]=v;
        SUDOKU-BACKTRACKING(B);
B[i,j]=empty;

```

- Letzte Zeile wird nur ausgeführt, wenn die Teillösung nicht zum Ziel geführt hat, dann wird im vorherigen Feld die nächste Zahl versucht
- Backtracking kann man als Tiefensuche auf Rekursionsbaum betrachten, wobei aussichtslose Lösungen evtl. frühzeitig abgeschnitten werden.
- Es ist auch "intelligenter" erschöpfende Suche, die aussichtslose Lösungen vorher aussortiert

Lösungssuche

1. Finde eine Lösung
2. Finde alle Lösungen
3. Finde beste Lösung

Beispiel: Regulärer Ausdruck

- Mustersuche in Strings
- Aufwand kann exponentiell werden

Dynamische Programmierung

Prinzip

- Teile Problem in (überlappende) Teilprobleme
- Löse rekursiv Teilprobleme, verwende dabei Zwischenergebnisse wieder (Memoization)
- Rekonstruiere Gesamtlösung
- Schwierigkeit: Finden geeigneter Rekursionen

Beispiel: Fibonacci

```

Fib-Rek(n) // n>=1
    IF n<=2 THEN
        return 1;
    ELSE
        return Fib-Rek(n-1)+Fib-Rek(n-2);

```

- Vereinfachte Laufzeitabschätzung: $T(n) \in \Theta(2^n)$
- Werte werden mehrfach berechnet
- Lösung: Werte zwischenspeichern (Memoization)

Fibonacci mit Memoization

```
FibDyn(n) // n>=1
    F[]=ALLOC(n); //F_i at F[i-1]
    FOR i=0 TO n-1 DO F[i]=0;
    return FibDynRek(n-1,F);

FibDynRek(i,F) // i>=0
    IF F[i]!=0 THEN return F[i]; // already calculated
    IF i<=1 THEN
        f=1
    ELSE
        f=FibDynRek(i-1,F)+FibDynRek(i-2,F);
    F[i]=f;
    return f;
```

- Wenn Basisfall erreicht ist, nur noch Addieren und Auslesen zu tun
- Laufzeit $\Theta(n)$

Minimum Edit Distance/Levenshtein-Distanz

Ziel:

- Messen der Ähnlichkeit von Texten
- Definiere 3 Buchstaben-Operationen:
 1. `ins(S,i,b)`: fügt an `i`-ter Position Buchstabe `b` in String `S` ein
 2. `del(S,i)`: löscht an `i`-ter Position Buchstaben in `S`
 3. `sub(S,i,b)`: ersetzt an `i`-ter Position in `S` den Buchstaben durch `b`
- Messe Ähnlichkeit anhand der Anzahl der benötigten Operationen zur Überführung zweier Texte ineinander
- Kosten/Operation ist 1, manchmal 2 für Substitution
- Nutze noch `copy(S,i)` für das Kopieren des `i`-ten Buchstabens, Kosten: 0
- Algorithmische Sichtweise:
 - String `X[1..m]` ist von links nach rechts in String `Y[1..n]` zu überführen
 - Zu jedem Zeitpunkt ist `X[1..i]` bereits in `Y[1..j]` transformiert
 - `D[i][j]` sei Distanz, um `X[1..i]` in `Y[1..j]` zu überführen ($i, j \geq 1$)
 - Betrachte nun nächsten Schritt um `x` in `y` zu überführen:
- `copy`: `D[i][j] = D[i-1][j-1]`
Bereits `X[1..i-1]` in `Y[1..j-1]` überführt, jetzt kostenfrei kopieren
- `sub`: `D[i][j] = D[i-1][j-1] + 1`
Bereits `X[1..i-1]` in `Y[1..j-1]` überführt, jetzt ersetzen
- `del`: `D[i][j] = D[i-1][j] + 1`
Bereits `X[1..i-1]` in `Y[1..j]` überführt, jetzt `X[i]` löschen

- **ins:** $D[i][j] = D[i][j-1] + 1$
Bereits $X[1..i]$ in $Y[1..j-1]$ überführt, jetzt $Y[j]$ einfügen
Fasse **copy** und **sub** zusammen:
- **copy/sub:** $D[i][j] = D[i-1][j-1] + (X[i] \neq Y[j])$ (Ausdruck ist 1 wenn wahr, sonst 0)
Suche nach der besten Strategie ist nun:
 $D[i][j] = \min \{D[i-1][j-1] + (X[i] \neq Y[j]), D[i-1][j] + 1, D[i][j-1] + 1\}$
Es gilt noch folgendes für die Ränder:
- $D[0][j] = j$: Füge j Buchstaben $Y[1..j]$ zu leerem String $X[1..0]$ hinzu
- $D[i][0] = i$: Lösche i Buchstaben $X[1..i]$, um leeren String $Y[1..0]$ zu erhalten

Algorithmus

verwendet dynamische Programmierung und Memoization

```
MinEditDist(X,Y,m,n) // X=X[1..m], Y=Y[1..n]
    D[][]=ALLOC(m,n);
    FOR i=0 TO m DO D[i][0]=i;
    FOR j=0 TO n DO D[0][j]=j;
    FOR i=1 TO m DO
        FOR j=1 TO n DO
            IF X[i]=Y[j] THEN s=0 ELSE s=1;
            D[i][j]=min{D[i-1][j-1]+s,D[i-1][j]+1,D[i][j-1]+1};
    return D[m][n];
```

Laufzeit und Speicherbedarf $\Theta(mn)$

Dieser Algorithmus füllt das zweidimensionale Array $D[][]$ mit den Distanzen. Es gilt:

Rückwärtsschrittfolge
 $D[m][n]$ zu $D[0][0]$ entlang
 der ausgewählten Minima
 (bzw. bis Rand erreicht)
 gibt Operationen an:

(↖) : **copy** (± 0)
 ↘ : **sub** (+1)
 → : **del**
 ↓ : **ins**

		X[1..i]							
D	0	1	2	3	4	5	6	7	8
0	0	1	2	3	4	5	6	7	8
1	1	1	2	3	4	5	6	7	8
2	2	2	2	3	4	5	6	7	8
3	3	2	3	3	4	5	5	6	7
4	4	3	2	3	4	5	6	6	7
5	5	4	3	3	3	4	5	6	7
6	6	5	4	4	4	4	5	6	7
7	7	6	5	5	5	5	4	5	6
8	8	7	6	6	6	6	5	4	5
9	9	8	7	7	7	7	6	5	4
10	10	9	8	8	8	8	7	6	5

Y[1..j]

Bei den Diagonalen Schritten gilt:

- Es wird kopiert, wenn die Distanz sich nicht verändert
 - Es wird substituiert, wenn sich die Distanz erhöht
- Im Allgemeinen gibt es mehrere mögliche Sequenzen

Greedy-Algorithmen

Prinzip

Finde Lösung $x = (x_1, x_2, \dots, x_n)$, indem Teillösung (x_1, x_2, x_{i-1}) durch den Kandidaten x_i ergänzt wird, der lokal am günstigsten erscheint

Beispiele

- **Dijkstra**: Wähle immer Knoten, der die kürzeste Distanz hat
 - **Kruskal**: Wähle jeweils leichteste Kante
- Greedy-Algorithmen funktionieren oft, aber nicht immer (z.B. Dijkstra und negative Kantengewichte)

Traveling Salesperson Problem (TSP)

Gegeben vollständiger (un-)gerichteter Graph $G = (V, E)$ mit Kantengewichten $w : E \rightarrow \mathbb{R}$, finde Tour p mit minimalem Kantengewicht $w(p)$. Eine Tour ist ein Weg $p = (v_0, v_1, \dots, v_n)$ entlang der Kanten $(v_i, v_{i+1}) \in E, i = 0, 1, \dots, n-1$, der bis auf Start- und Endknoten $v_0 = v_n$ jeden Knoten genau einmal besucht ($V = \{v_0, v_1, \dots, v_{n-1}\}$)

Graph $G = (V, E)$ ist vollständig, wenn es f.a. $u, v \in V, u \neq v$ eine Kante $(u, v) \in E$ gibt.

Unvollständiger Graph mit Tour lässt sich erweitern, indem man fehlende Kanten (u, v) verboten teuer macht: $w((u, v)) := |V| \cdot \max_{e \in E} \{w(e)\} + 1$ f.a. $(u, v) \notin E$

TSP vs. Dijkstra

- Allgemeiner TSP-Algorithmus:
 - Finde optimale Route, die durch jeden Knoten geht und zum Ausgangspunkt zurückkehrt
- **Dijkstra** löst anderes Problem:
 - Finde optimalen Pfad vom Ausgangspunkt aus
 - Besucht eventuell nicht alle Knoten und betrachtet auch nicht Rückkehr

Ansatz Greedy-Algorithmus für TSP

Starte mit beliebigem oder gegebenem Knoten.

Nehme vom gegenwärtigen Knoten aus die Kante zu noch nicht besuchtem Knoten, die kleinstes Gewicht hat.

Wenn kein Knoten mehr übrig, gehe zu Startpunkt zurück.

```
Greedy-TSP(G,s,w) // |V|=n, s starting node
    FOREACH v in V DO v.color=white;
    tour[]=ALLOC(n); tour[0]=s; tour[0].color=gray;
    FOR i=1 TO n-1 DO
        tour[i]=EXTRACT-MIN(adj(tour[i-1]));
        // get (white) neighbor with minimum edge weight
```

```
        tour[i].color=gray;
    return tour;
```

Ist zu gierig!

Effizienter Algorithmus für TSP

Vermutlich schwierig zu finden

Siehe auch [NP-Vollständigkeit](#)

Metaheuristiken

Heuristik

- Dedizierter Suchalgorithmus für Optimierungsproblem, der gute (eventuell nicht optimale) Lösung für spezielles Problem findet
- Problem-abhängig: Arbeitet mit konkretem Problem

Metaheuristik

- Allgemeine Vorgehensweise, um Suche für beliebige Optimierungsprobleme zu leiten
- Problem-unabhängig: Arbeitet mit abstrakten Problemen

Lokale Suche/Hill-Climbing-Strategie

1. Finde erste Lösung
2. Suche in Nähe bessere Lösungen, bis keine Verbesserung mehr/Zeit um

Hill-Climbing-Algorithmus

```
HillClimbing(P) // maxTime constant
    sol=initialSol(P); // initial solution
    time=0;
    WHILE time<maxTime DO
        new=perturb(P,sol); // slightly alter solution
        IF quality(P,new)>quality(P,sol) THEN
            sol=new; // replace solution with new one iff it's
better
            time=time+1;
    return sol;
```

Beispiel TSP

- `initSol`: Wähle beliebige Tour, z.B. per [Greedy-Algorithmus](#)
- `perturb`: Tausche 2 zufällige Knoten

- `quality`: Gewicht der aktuellen Tour

Lokale/Globale Maxima

Eventuell bleibt Hill-Climbing-Algorithmus in lokalem Maximum hängen, da stets nur leichte Lösungsänderungen in aufsteigender Richtung!

Siehe auch: [6.3. Extremwerte](#)

Iterative, lokale Suche

1. Führe [lokale Suche](#) durch
2. Beginne Suche nochmal von vorne, z.B. mit neuer zufälliger Lösung, eventuell auch mehrmals
3. Akzeptiere beste gefundene Lösung
Problem: Zufällige Lösungen könnten auch schlecht sein

Simulated Annealing

- "Annealing" in Metallverarbeitung:
Härten von Metallen durch Erhitzen auf hohe Temperatur und langsames Abkühlen
 - Entscheide je nach Temperatur, in welche Richtung gesucht wird
1. Temperatur zu Beginn hoch, kühlt langsam ab
 2. Je höher Temperatur, desto wahrscheinlicher Sprung in schlechte Richtung
- Mit Wahrscheinlichkeit in schlechte Richtung

Ansatz

Akzeptiere auch Lösung `new` mit `quality(new) < quality(sol)` mit Wahrscheinlichkeit:

$$\text{rand}(0,1) < e^{\frac{\text{quality}(\text{new}) - \text{quality}(\text{sol})}{\text{temperature}}}$$

temperature nimmt mit Zeit ab:

- Zu Beginn heiße Temperatur: Akzeptiere oft viel schlechtere Lösungen
 - Am Ende kühlere Temperatur: Akzeptiere selbst wenig schlechtere Lösungen fast nie
- Gegen Ende fast [Hill-Climbing-Strategie](#)

```
SimulatedAnnealing(P) // maxTime constant, TempSched[] temperature annealing
    sol=initialSol(P);
    time=0;
    WHILE time<maxTime DO
        new=perturb(P,sol);
        temperature=TempSched[time];
        d=quality(P,new)-quality(P,sol); r=random(0,1); // equally
distributed
        IF d>0 OR r<exp(d/temperature) THEN
```

```
        sol=new;  
        time=time+1;  
    return sol;
```

- Bestimmung eines guten "Annealing schedule" (Starttemperatur und Abnahme) ist nicht Teil der Veranstaltung

Weitere Metaheuristiken

Es gibt noch viele mehr, z.B. Schwarmoptimierung, Ameisenkolonialisierung, ...

Tabu Search

- Suche bessere Lösung in der Nähe ausgehend von aktueller Lösung
- Speichere eine Zeit lang schon besuchte Lösungen, vermeide diese Lösungen
- Wenn keine bessere Lösung in der Nähe, akzeptiere auch schlechtere Lösung

Evolutionäre Algorithmen

- Beginne mit Lösungspopulation
- Wähle beste Lösungen zur Reproduktion aus
- Bilde durch Überkreuzungen und Mutationen der besten Lösungen neue Lösungen
- Ersetze schlechteste Lösungen durch diese neue Lösungen