7. Advanced Designs

Auswahl algorithmischer Entwurfmethoden

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 $x := (x_1, x_2, \ldots, x_n)$ per "Trial-and-Error", indem Teillösung $(x_1, x_2, \ldots, x_{i-1})$ 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 "intelligentere" 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

```
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];
```

#TODO add

Greedy-Algorithmen

Prinzip

Finde Lösung $x = (x_1, x_2, \ldots, x_n)$, 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 $w : E \to \mathbb{R}$, finde Tour p mit minimalem Kantengewicht $w(p)$. Eine Tour ist ein Weg $p = (v_0, v_1, \ldots, v_n)$ entlang der Kanten $(v_1, v_{i+1}) \in E, i = 0, 1, \ldots, n-1$, der bis auf Start- und Endknoten $v_0 = v_n$ jeden Knoten genau einmal besucht ($V = \{v_0, v_!, \ldots, v_{n-1}\}$)
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
#TODO passenden Link zu 8 einfügen

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:
$rand(0,1) < e^{\frac{quality(new)-(quality(sol)}{temperature}}$
$temperature$ 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[] temparature 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