12 Generische Optimierungsansätze

- Black-Box-Löser
- Greedy
- Dynamische Programmierung
- Systematische Suche
- Lokale Suche
- Evolutionäre Algorithmen

Maximierungsproblem $(\mathscr{L}, f)$

- $\mathscr{L} \subseteq \mathscr{U}$: zulässige Lösungen
- $f : \mathscr{L} \to \mathbb{R}$ Zielfunktion
- $\mathbf{x}^* \in \mathscr{L}$ ist optimale Lösung falls $f(\mathbf{x}^*) \geq f(\mathbf{x})$ für alle $\mathbf{x} \in \mathscr{L}$

Minimierungsprobleme: analog

Problem: variantenreich, meist NP-hart

12.1 Black-Box-Löser

- (Ganzzahlige) Lineare Programmierung — siehe unten
- SAT-Löser — Problem $\longrightarrow$ Aussagenlogik $\longrightarrow$ CNF
- Constraint-Programming

Constraint-Programming

Variablen: $x_1, \ldots, x_n \in \mathbb{N}$

Bedingungen (Constraints): $C_1(x_1, \ldots, x_n), \ldots, C_k(x_1, \ldots, x_n) \subseteq \mathbb{N}^n$

Ziel: Erfüllende Belegung (auf Knopfdruck)

Beispiele: n-Damenproblem, scheduling,...

n-Damenproblem

- Positioniere n Damen auf einem $n \times n$ Schachbrett
- Variablen $x_1 \ldots x_n$:

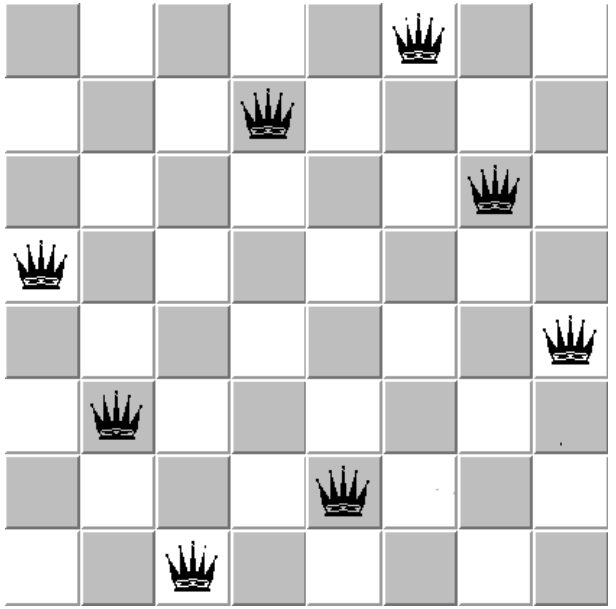
  Dame i ist in Zeile i und Spalte x_i
- Bedingungen:

  $x_i \in \{1, \ldots, n\}$

  AllDifferent$(x_1, \ldots, x_n)$

  AllDifferent$(x_1 + 1, \ldots, x_n + n)$

  AllDifferent$(x_1 - 1, \ldots, x_n - n)$



Bemerkung: Es gibt auch eine Linearzeitalgorithmus!

Lineare Programmierung

Ein lineares Programm mit n Variablen und m Constraints wird durch das folgende Minimierungsproblem definiert

- Kostenfunktion $f(\mathbf{x}) = \mathbf{c} \cdot \mathbf{x}$

  $\mathbf{c}$ ist der Kostenvektor

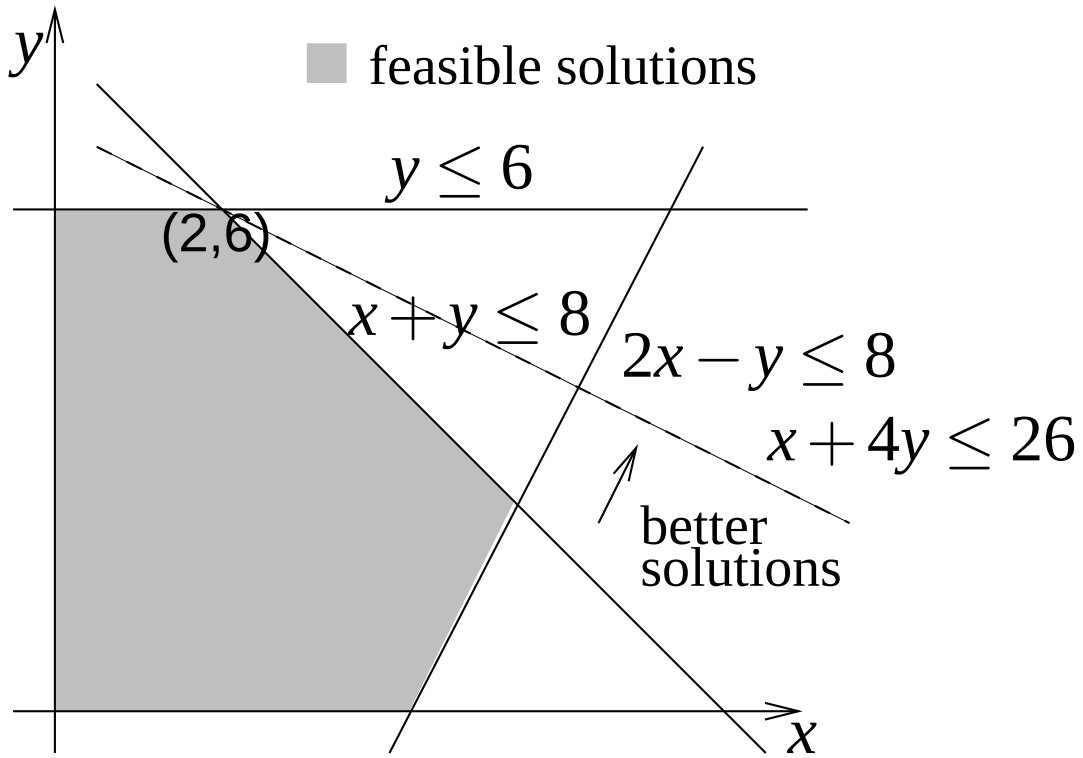
- m constraints der Form $\mathbf{a}_i \cdot \mathbf{x} \bowtie_i b_i$ mit $\bowtie_i \in \{\leq, \geq, =\}$, $\mathbf{a}_i \in \mathbb{R}^n$

  Wir erhalten

$$\mathscr{L} = \{\mathbf{x} \in \mathbb{R}^n : \forall 1 \leq i \leq m : x_i \geq 0 \wedge \mathbf{a}_i \cdot \mathbf{x} \bowtie_i b_i\} \quad .$$

Sei a_{ij} die j-te Komponente von Vektor $\mathbf{a}_i$.

Ein einfaches Beispiel



Beispiel: Kürzeste Wege

maximize $\sum_{v \in V} d_v$

subject to $d_s = 0$

$d_w \leq d_v + c(v,w)$ for all $(v,w) \in E$

Eine Anwendung – Tierfutter

- n Futtersorten.

  Sorte i kostet c_i Euro/kg.

- m Anforderungen an gesunde Ernährung.

  (Kalorien, Proteine, Vitamin C,. . .)

  Sorte i enthält a_{ji} Prozent des täglichen Bedarfs

  pro kg bzgl. Anforderung j

- Definiere x_i als

  zu beschaffende Menge von Sorte i

- LP-Lösung gibt eine kostenoptimale “gesunde” Mischung.

Verfeinerungen

- Obere Schranken (Radioaktivität, Cadmium, Kuhhirn, . . .)
- Beschränkte Reserven (z.B. eigenes Heu)
- bestimmte abschnittweise lineare Kostenfunktionen (z.B. mit Abstand wachsende Transportkosten)

Grenzen

- Minimale Abnahmemengen
- die meisten nichtlinearen Kostenfunktionen
- Ganzzahlige Mengen (für wenige Tiere)
- Garbage in Garbage out

Algorithmen und Implementierungen

- LPs lassen sich in polynomieller Zeit lösen [Khachiyan 1979]
- Worst case $O\left(\max(m,n)^{\frac{7}{2}}\right)$
- In der Praxis geht das viel schneller
- Robuste, effiziente Implementierungen sind sehr aufwändig

$\rightsquigarrow$ Fertige freie und kommerzielle Pakete

Ganzzahlige Lineare Programmierung

ILP: Integer Linear Program, lineares Programm mit der zusätzlichen Bedingung $x_i \in \mathbb{N}$.

oft: 0/1 ILP mit $x_i \in \{0,1\}$

MILP: Mixed Integer Linear Program, lineares Programm bei dem einige Variablen ganzzahlig sein müssen.

Lineare Relaxation: Entferne die Ganzzahligkeitsbedingungen eines (M)ILP

Beispiel: Rucksackproblem

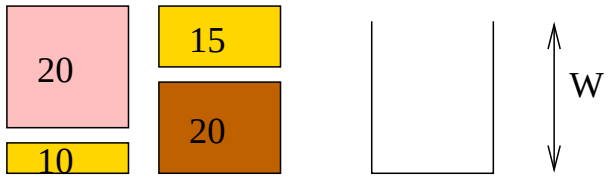
$$\text{maximize} \mathbf{p} \cdot \mathbf{x}$$

subject to

$$\mathbf{w} \cdot x \leq M, x_i \in \{0,1\} \text{ for } 1 \leq i \leq n \ .$$

$x_i = 1$ gdw Gegenstand i in den Rucksack kommt.

0/1 Variablen sind typisch für ILPs



Umgang mit (M)ILPs

- − NP-hard
- \+ Ausdrucksstarke Modellierungssprache
- \+ Es gibt generische Lösungsansätze, die manchmal gut funktionieren
- \+ Viele Möglichkeiten für Näherungslösungen
- \+ Die Lösung der linearen Relaxierung hilft oft, z.B. einfach runden.
- \+ Ausgefeilte Softwarepakete

Beispiel: Beim Rucksackproblem gibt es nur eine fraktionale Variable in der linearen Relaxierung – abrunden ergibt zulässige Lösung.

Annähernd optimal falls Gewichte und Profite $\ll$ Kapazität

12.2 Nie Zurückschauen – Greedy-Algorithmen

Idee: treffe jeweils eine lokal optimale Entscheidung

Optimale Greedy-Algorithmen

- Dijkstra's Algorithmus für kürzeste Wege
- Minimale Spannbäume
  - Jarník-Prim
  - Kruskal
- Selection-Sort (wenn man so will)

Greedy-Approximationsalgorithmen

- Handlungreisendenproblem — Faktor 2
- Rucksackproblem — Faktor 2
- Machine Scheduling — Faktor $2, 4/3, 7/6$
- Knotenüberdeckung — Faktor 2
- Mengenüberdeckung — Faktor $\log U$, “bestmöglich”
- Gewichtetes Matching — Faktor $1/2, 2/3 - \varepsilon$

  [Drake-Hougardy 2003, 2004, Pettie-Sanders 2004]

Mehr: Vorlesung Approximations- und Onlinealgorithmen

Gewichtetes Matching

Gegeben: Ungerichteter Graph $G = (V, E)$ mit Kantengewichten $w(e)$.

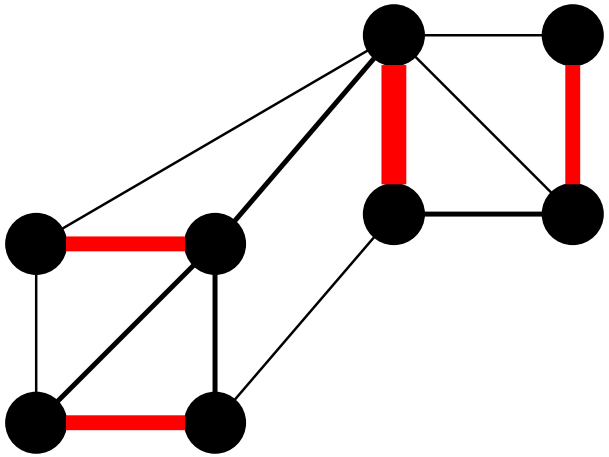
Gesucht: Kantenmenge $M \subseteq E$ mit

(V, M) hat max. Grad 1. *Matching*

$w(M) := \sum_{e \in M} w(e)$ ist maximal

$\exists$ Algorithmus mit Laufzeit $O(n(m + n \log n))$

Trotzdem sind schnelle Approximationsalgorithmen interessant



Greedy Gewichtetes Matching [Avis 1983]

sort E by decreasing weight

$M := \emptyset$

repeat

  invariant M is a matching

  pick first (heaviest) $e = \{u, v\} \in E$

  $M := M \cup \{e\}$

  $E := E \setminus \{\{x, u\} \in E\}$

  $E := E \setminus \{\{x, v\} \in E\}$

until $E = \emptyset$

Laufzeit: $O(m + \text{sort}(m))$

Greedy Gewichtetes Matching: Lösungsqualität

Satz: $w(M) \geq w(M^*)/2$ M^* sei Optimallösung

Beweis: Betrachte beliebige Kante $e = \{u, v\} \in M$.

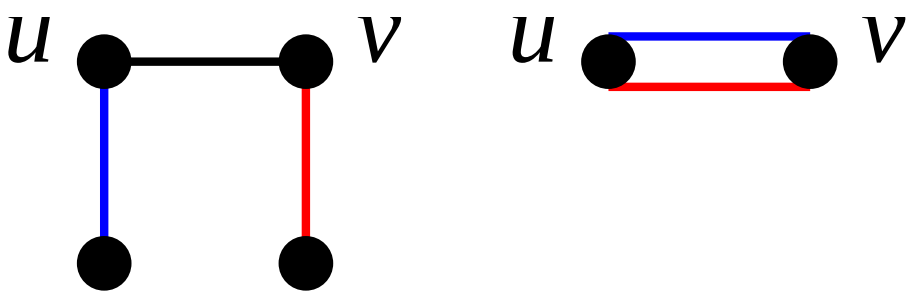
Dadurch werden höchstens zwei Kanten

$\{u,x\}$ und $\{v,y\}$ aus M^* entfernt.

Außerdem: $w(e) \geq w(\{u,x\})$ und $w(e) \geq w(\{v,y\})$.

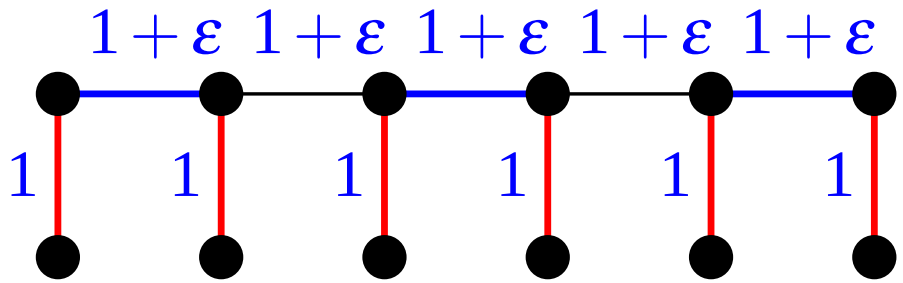
Oder $2w(e) \geq w(\{u,x\}) + w(\{v,y\})$

Also: $\sum_{e \in M} 2w(E) \geq w(M^*)$



Greedy Gewichtetes Matching: Lösungsqualität

Satz: $\exists G : w(M) \approx w(M^*)/2$



Knotenüberdeckungsproblem (vertex cover)

Betrachte Graphen $G = (V, E)$

$S \subseteq V$ ist eine Knotenüberdeckung gdw

$$\forall \{u, v\} \in E : u \in S \vee v \in S$$

minimale Knotenüberdeckung (MIN-VCP):

Finde vertex cover S mit minimalem $|S|$.

Greedy Algorithmus

Function greedyVC(V,E)

$C := \emptyset$

while $E \neq \emptyset$ **do**

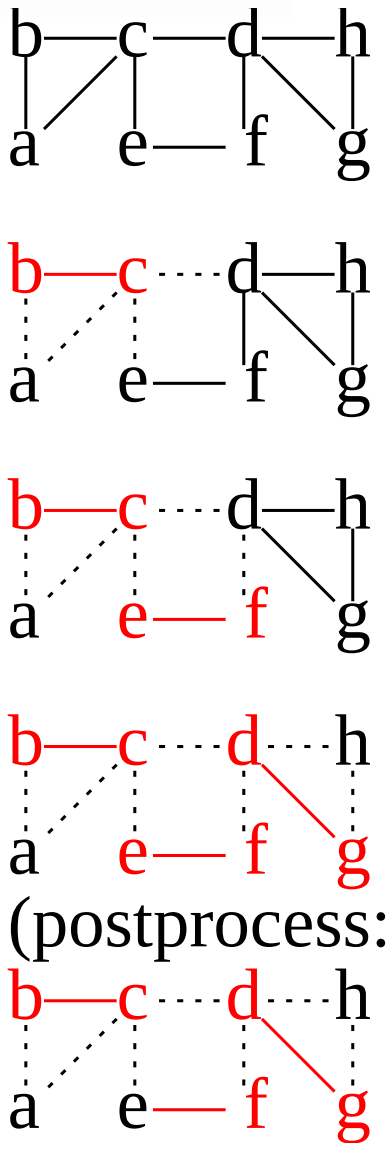
select any $\{u,v\} \in E$

$C := C \cup \{u,v\}$

remove all edges incident to u or v from E

return C

Übung: Implementierung mit Laufzeit $O(|V|+|E|)$



Satz: Algorithm greedyVC berechnet 2-Approximation von MIN-VCP.

Korrektheit: trivial da nur abgedeckte Kanten entfernt werden

Qualität: Sei A die Menge der durch greedyVC ausgewählten Kanten.

Es gilt $|C| = 2|A|$.

A ist ein Matching, d.h., kein Knoten berührt zwei Kanten in A.

Folglich enthält jede Knotenüberdeckung mindestens einen Knoten von jeder Kante in A, d.h., opt$\geq |A|$. □

Das Mengenüberdeckungsproblem (set cover)

Gegeben:

Universum U, Teilmengen $\mathscr{S} = \{S_1, \ldots, S_k\}$, Kostenfunktion $c : \mathscr{S} \to \mathbb{N}$.

Gesucht: billigstes $\mathscr{S}' \subseteq S$ mit $\bigcup_{S \in \mathscr{S}'} S = U$

Greedy-Algorithmus: Wähle jeweils ein $S \in \mathscr{S}$ mit den geringsten Kosten pro neu abgedecktem Element von U

Laufzeit: $O(\sum_{S \in \mathscr{S}} |S| \log |\mathscr{S}|)$

Faktor $\log |\mathscr{S}|$ entfällt bei konstanter Kostenfunktion.

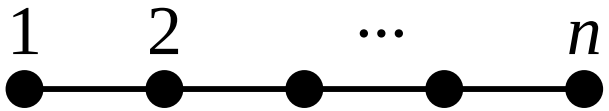
Datenstrukturen dafür: Übung

12.3 Dynamische Programmierung – Aufbau aus Bausteinen

Anwendbar wenn, das **Optimalitätsprinzip** gilt:

- Optimale Lösungen bestehen aus optimalen Löungen für Teilprobleme.
- Mehrere optimale Lösungen $\Rightarrow$ es is egal welches benutzt wird.

Beispiel: Gewichtetes Matching für einen Pfad (Paarung)



Teilprobleme: M_i:= optimales Matching für Teilpfad $1..i$

Rekurrenz: M_0:= $M_1 = \emptyset$,

$M_i = M_{i-1}$ oder $M_{i-2} \cup \{\{i-1, i\}\}$

jenachdem was größeres Gewicht hat.

Teilprobleme: M_i:= optimales Matching für Teilpfad $1..i$

Rekurrenz: M_0:= $M_1 = \emptyset$, $M_i = M_{i-1}$ oder $M_{i-2} \cup \{\{i-1, i\}\}$

Zu zeigen:

M_i ist Matching: OK (Induktion)

M_i hat max. Gewicht: Induktion über i.

Fall $i \leq 2$ trivial.

Sei M_i^* das opt. matching für $1..i$, $i > 2$.

Fall $\{i-1, i\} \in M_i^*$: $\{i-2, i-1\} \notin M_i^*$.

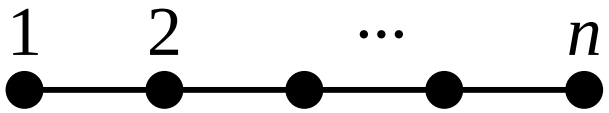
$\Rightarrow M_i^* \setminus \{\{i-1, i\}\}$ ist ein Matching für $1..i-2$

und das muss max. Gewicht haben. Also

$M_i^* = M_{i-2}^* \cup \{\{i-1, i\}\} = M_{i-2} \cup \{\{i-1, i\}\} = M_i$

Fall $\{i-1, i\} \notin M_i^*$: $M_i^* = M_{i-1}^* = M_{i-1}$ □

Beispiel: Gewichtetes Matching für einen Pfad



Teilprobleme: M_i:= optimales Matching für Teilpfad $1..i$

Rekurrenz: $M_0 := M_1 = \emptyset$, $M_i = M_{i-1}$ oder $M_{i-2} \cup \{\{i-1, i\}\}$

Tabellenaufbau: trivial

Lösungsrekonstruktion: $\forall i$: speichern, ob $\{i-1, i\} \in M_i$

Speicherbedarf: n bits + zwei Gewichte

Pfade sind zu speziell? Global Path Algorithmus

sort E by decreasing weight

$M := P := \emptyset$

foreach $e \in E$ **do**

    invariant P is a collection of paths and even cycles

    if $P \cup \{e\}$ fulfills the invariant **then** $P := P \cup \{e\}$

foreach path or cycle C in P **do** $M := M \cup \text{optMatching}(C)$

Übung: optimaler Linearzeit-Algorithmus für Kreise.

Laufzeit: $\text{sort}(m) + O(m)$

Algorithmenentwurf mittels dynamischer Programmierung

1. Was sind die Teilprobleme? Kreativität!
2. Wie setzen sich optimale Lösungen aus Teilproblemlösungen zusammen? Beweisnot
3. Bottom-up Aufbau der Lösungstabelle einfach
4. Rekonstruktion der Lösung einfach
5. Verfeinerungen:
   Platz sparen, Cache-effizient, Parallelisierung Standard-Trickkiste

Anwendungen dynamischer Programmierung

- Bellman-Ford Alg. für kürzeste Wege — Teilpfade
- Edit distance/approx. string matching — Paare von Präfixen
- Verkettete Matrixmultiplikation — Übung
- Rucksackproblem — Gegenstände $1..i$ füllen Teil des Rucksacks
- Geld wechseln — Übung

Gegenbeispiel: Teilproblemeigenschaft

Angenommen, die schnellste Strategie für 20 Runden auf dem Hockenheimring verbraucht den Treibstoff vollständig.

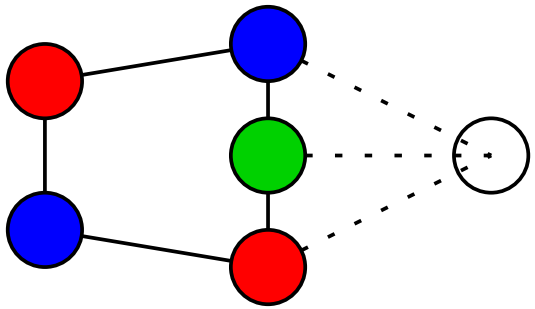
$\Rightarrow$

Keine gute Teilstrategie für 21 Runden.

Frage: Wie kann man constrained shortest path trotzdem als dynamischen Programmier-problem modellieren?

Gegenbeispiel: Austauschbarkeit

Optimale Graphfärbungen sind nicht austauschbar.



12.4 Systematische Suche

Idee: Alle (sinnvollen) Möglichkeiten ausprobieren.

Anwendungen:

- Integer Linear Programming (ILP)
- Constraint Satisfaction
- SAT (Aussagenlogik)
- Theorembeweiser (Prädikatenlogik,. . .)
- konkrete NP-harte Probleme
- Strategiespiele
- Puzzles

Beispiel: Branch-and-Bound für Rucksackproblem

Function bbKnapsack$((p_1,\ldots,p_n),(w_1,\ldots,w_n),M) : \mathscr{L}$

  assert $p_1/w_1 \geq p_2/w_2 \geq \cdots \geq p_n/w_n$

  $\hat{\mathbf{x}}$=heuristicKnapsack$((p_1,\ldots,p_n),(w_1,\ldots,w_n),M)$ $: \mathscr{L}$

  $\mathbf{x} : \mathscr{L}$

  recurse$(1,M,0)$

  return $\hat{\mathbf{x}}$

  //Find solutions assuming $x_1,\ldots,x_{i-1}$ are fixed,

  //$M' = M - \sum_{k<i} x_i w_i$, $P = \sum_{k<i} x_i p_i$.

  Procedure recurse$(i, M', P : \mathbb{N})$

$\mathbf{x}$ // current Solution

$\hat{\mathbf{x}}$ // best solution so far

Procedure recurse$(i, M', P : \mathbb{N})$

u:= P + upperBound$((p_i, \ldots, p_n), (w_i, \ldots, w_n), M')$

if $u > \mathbf{p} \cdot \hat{\mathbf{x}}$ **then**

if $i > n$ **then** $\hat{\mathbf{x}} := \mathbf{x}$

else // Branch on variable x_i

if $w_i \leq M'$ **then** x_i:= 1; recurse$(i+1, M' - w_i, P + p_i)$

if $u > \mathbf{p} \cdot \hat{\mathbf{x}}$ **then** x_i:= 0; recurse$(i+1, M', P)$

Worst case: 2^n rekursive Aufrufe

Averge case: Linearzeit?

Branch-and-Bound – allgemein

Branching (Verzweigen): Systematische Fallunterscheidung,
z.B. rekursiv (Alternative, z.B. Prioritätsliste)

Verweigungsauswahl: Wonach soll die Fallunterscheidung stattfinden?
(z.B. welche Variable bei ILP)

Reihenfolge der Fallunterscheidung: Zuerst vielversprechende Fälle
(lokal oder global)

Bounding: Nicht weitersuchen, wenn optimistische Abschätzung der erreichbaren Lösungen schlechter als beste (woanders) gefundene Lösung.

Duplikatelimination: Einmal suchen reicht (unnötig bei ILP)

Anwendungsspez. Suchraumbeschränkungen: Schnittebenen (ILP),
Lemma-Generierung (Logik),. . .

12.5 Lokale Suche – global denken, lokal handeln

find some feasible solution $\mathbf{x} \in \mathcal{S}$

$\hat{\mathbf{x}} := \mathbf{x}$ // $\hat{\mathbf{x}}$ is best solution found so far

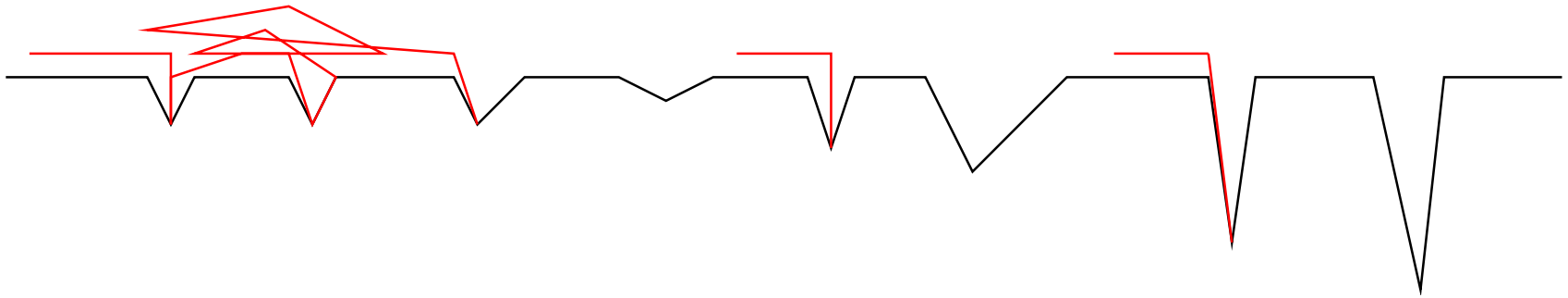
while not satisfied with $\hat{\mathbf{x}}$ **do**

$\mathbf{x}$:= some heuristically chosen element from $\mathcal{N}(\mathbf{x}) \cap \mathcal{S}$

if $f(\mathbf{x}) < f(\hat{\mathbf{x}})$ **then** $\hat{\mathbf{x}} := \mathbf{x}$

Restarts: Ggf. mehrmals wiederholen

(für unterschiedliche Ausgangslösungen)



Hill Climbing

Find some feasible solution $\mathbf{x} \in \mathcal{L}$

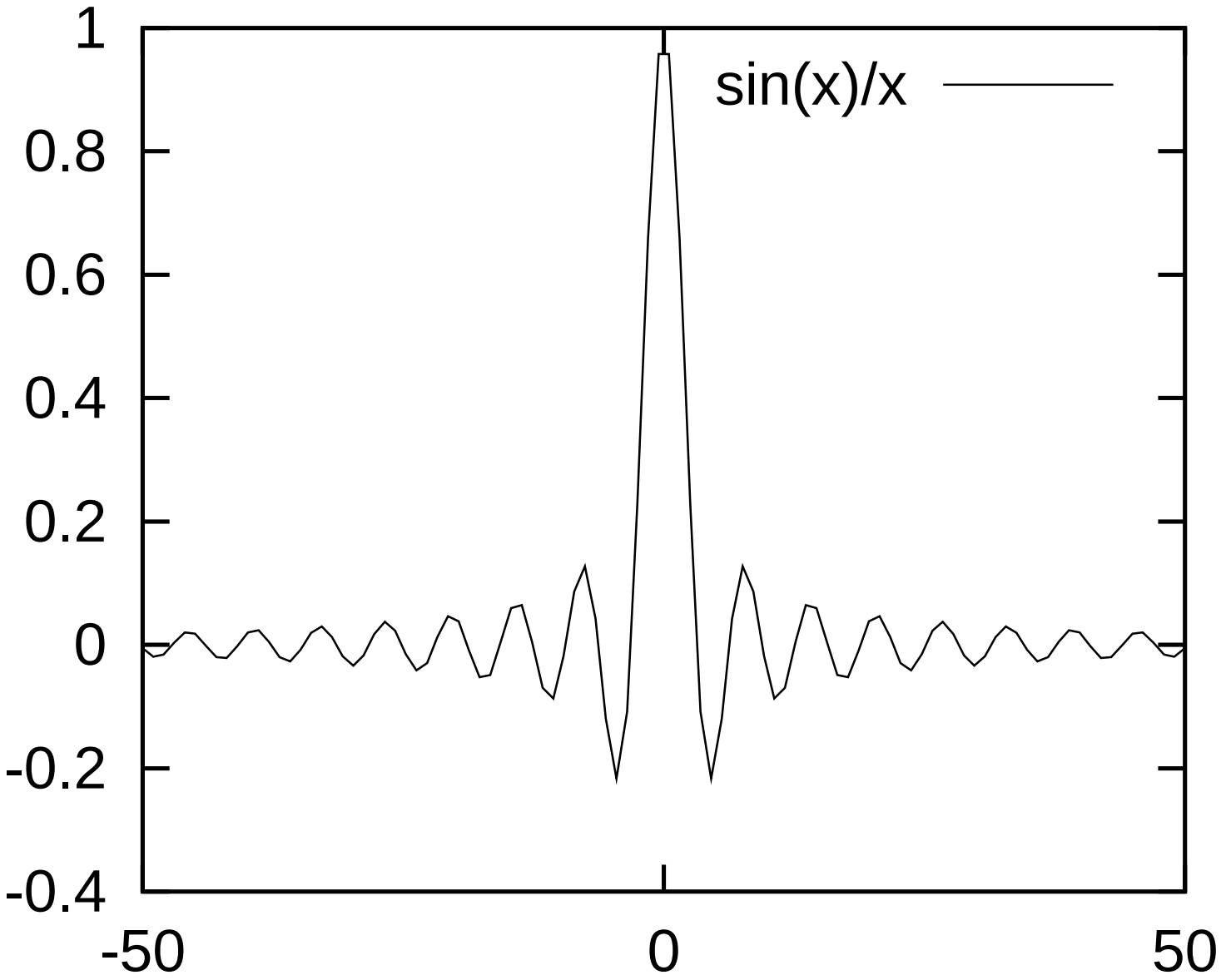
$\hat{\mathbf{x}} := \mathbf{x}$ // best solution found so far

loop

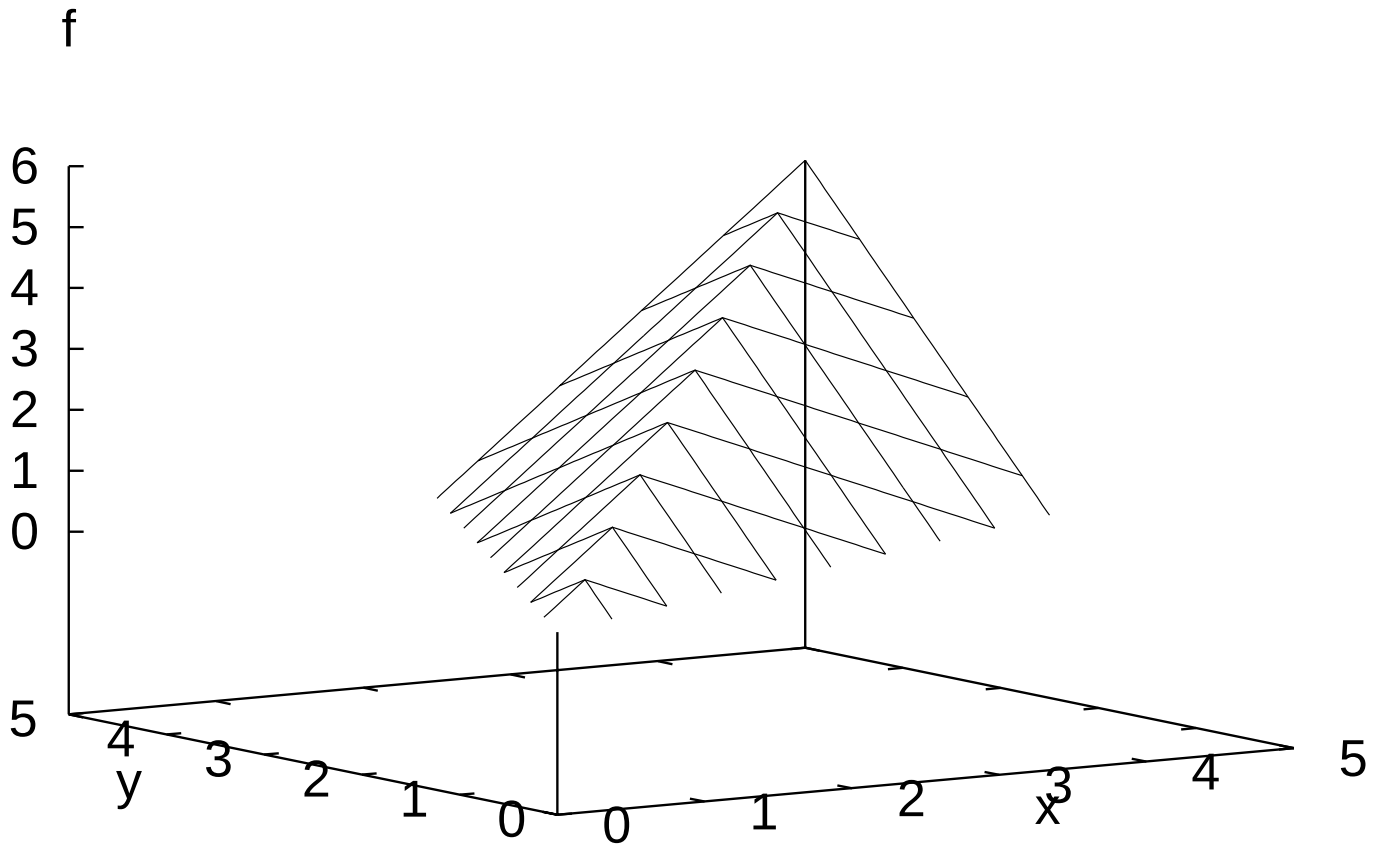
if $\exists \mathbf{x} \in \mathcal{N}(\mathbf{x}) \cap \mathcal{L} : f(\mathbf{x}) < f(\hat{\mathbf{x}})$ **then** $\hat{\mathbf{x}} := \mathbf{x}$

else return $\hat{\mathbf{x}}$ // local optimum found

Problem: Lokale Optima



Warum die Nachbarschaft wichtig ist



Gegenbeispiel für Koordinatensuche

Beispiel: Simplexalgorithmus für LP

Die zulässige Region $\mathscr{L}$ ist ein konvexes Polytop

begrenzt durch Halbräume,

die den Constraints $\mathbf{a}_i \cdot \mathbf{x} \bowtie_i b_i$ entsprechen

($\bowtie_i \in \{\leq, \geq\}$ OBdA keine Gleichheit)

Startlösung: irgendein Knoten

Nachbarschaft: Nachbarknoten

(durch eine Polytopkante erreichbar)

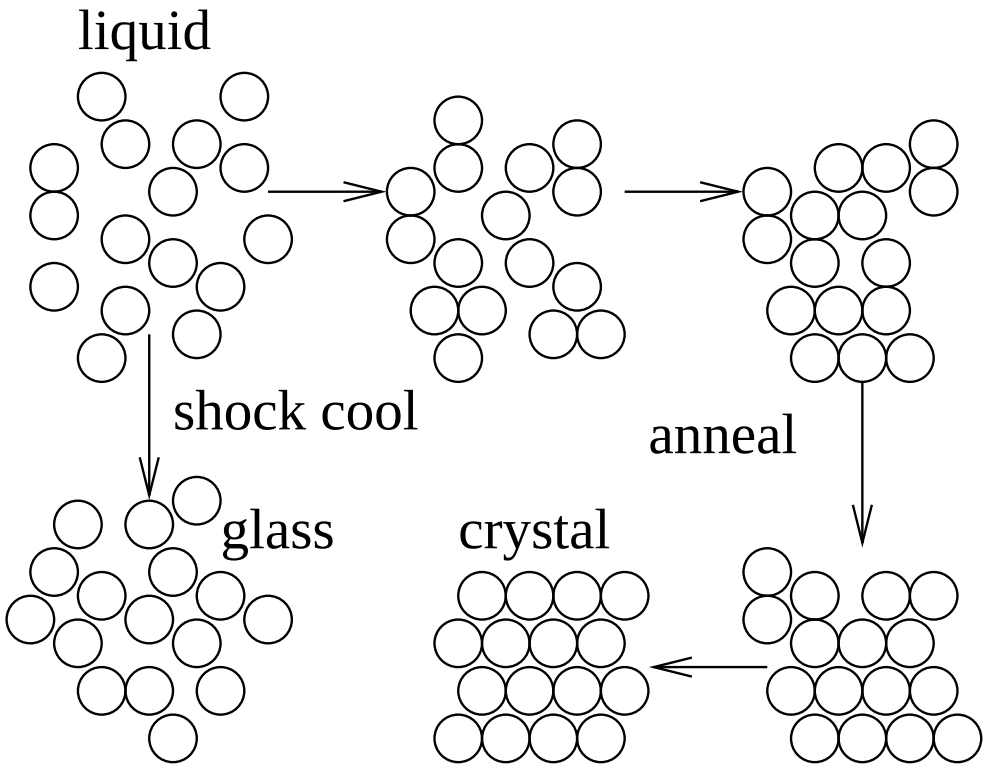
Konvexität $\Rightarrow$ keine lokalen Minima

Offenes Problem: welchen besseren Nachbarn wählen?

Das kann den Unterschied zwischen exponentieller und polynomieller Laufzeit ausmachen.

Simulated Annealing

Eine physikalische Analogie



Qualitative Erklärung:

Lokal optimal Teilstrukturen haben Zeit sich aufzulösen

Die Boltzmann-Verteilung

W-Dichte der Energie im Gleichgewicht

$$\mathbb{P}[E_{\mathbf{x}}] = \frac{\exp -\frac{E_{\mathbf{x}}}{T}}{\sum_{\mathbf{y}\in\mathscr{L}} \exp -\frac{E_{\mathbf{y}}}{T}}$$

$T =$ Temperatur des Systems in Kelvin multipliziert mit der *Boltzmann-Konstante* $k_B \approx 1.4 \cdot 10^{-23} J/K$.

Satz A: Falls die Menge der möglichen Energien endlich ist, gilt

$$\lim_{T\to 0} \mathbb{P}[E_{\mathbf{x}}] = 1$$

gdw $E_{\mathbf{x}}$ ist ein globales Minimum. $\Longrightarrow$ Falls wir unendlich langsam abkühlen, ergibt sich ein perfekter Kristall

Übersetzung in ein Minimierungsproblem

Energie	$E_{\mathbf{x}} \longleftrightarrow f(\mathbf{x})$	Zielfunktion
Systemzustand	$\mathbf{x} \longleftrightarrow \mathbf{x}$	zulässige Lösung
Boltzmann-Verteilung	$\longleftrightarrow$	Boltzmann-Verteilung (!?)
Temperatur	$T \longleftrightarrow T$	ein Parameter der die lokale Suche steuert

Annealing Auswahlregel

wähle $\mathbf{x}'$ aus $\mathcal{N}(\mathbf{x}) \cap \mathcal{L}$ zufällig gleichverteilt

Annealing Akzeptanzregel

with probability $\min(1, \exp(\frac{f(\mathbf{x}) - f(\mathbf{x}')}{T})$ **do** $\mathbf{x} := \mathbf{x}'$

Satz B: Betrachte den Graphen

$G = (\mathcal{L}, E)$ mit $E = \{(\mathbf{x}, \mathbf{y}) : y \in \mathcal{N}(\mathbf{x})\}$.

Falls G regulär, ungerichtet und stark zusammenhängend ist,

dann gilt für eine feste Temperatur T, dass simulated annealing gegen

einen Zustand konvergiert in dem die Energien einer

Boltzmann-Verteilung unterliegen

Theoretische Verfeinerungen

[Aarts, Korst: Simulated Annealing and Boltzmann Machines, Wiley 1989.]

- Der Beweis von Satz B benutzt die Theorie der Markov-Ketten. (siehe Vorlesung randomisierte Algorithmen)
- Regularität and Ungerichtetheit sind unrealistisch. Verzicht darauf zerstört die Boltzmann-Verteilung aber es gilt immer noch

$$\lim_{T \to 0} \mathbb{P}\left[E_{\mathbf{x}}\right] = 1 \text{ gdw } x \text{ ist Optimum}$$

- $2 \times \infty$ (Gleichgewicht, kühlen) lassen sich entfernen. Zum Beispiel, das TSP wird in Zeit $\mathrm{O}\left(n^{n^{2n-1}}\right)$ gelöst. Vergleiche mit $n!$ für den naiven Ansatz.

Simulated Annealing in der Praxis

Nachbarschaft: Wie immer bei lokaler Suche

Auswahlregel $\mathcal{N}$: ggf. nicht gleichverteilt

Anfangstemperatur: So klein wie möglich, dass immer noch viel akzeptiert wird.

Gleichgewichtserkennung: #Schritte $\gg |\mathcal{N}|$,....

Abkühlen: T mit Konstante multiplizieren, typischerweise $0.8 \ldots 0.99$.

Terminierung: $T \approx \min\{|f(\mathbf{x}) - f(\mathbf{x}')| : \mathbf{x}, \mathbf{x}' \in \mathcal{L}\} \Rightarrow T := 0$
danach im wesentlichen hill climbing

Parameterwahl ist “Hexenwerk”.

Dynamische Abkühlpläne: Versuche Entscheidungen adaptiv unter Berücksichtigung des Zustands zu treffen.

Beispiel: Graphfärben mit Fixed-K-Annealing

- $\mathscr{L}$ = alle Färbungen mit k Farben (illegal OK)
- $f(\mathbf{x}) = |\{(u,v) \in E) : \mathbf{x}(u) = \mathbf{x}(v)\}|$
- Nachbarschaft: ändere eine Farbe an einem Knoten

Eine exp. Studie zu Graphfärben

[Johnson,Aragon,McGeoch,Schevon 1991]

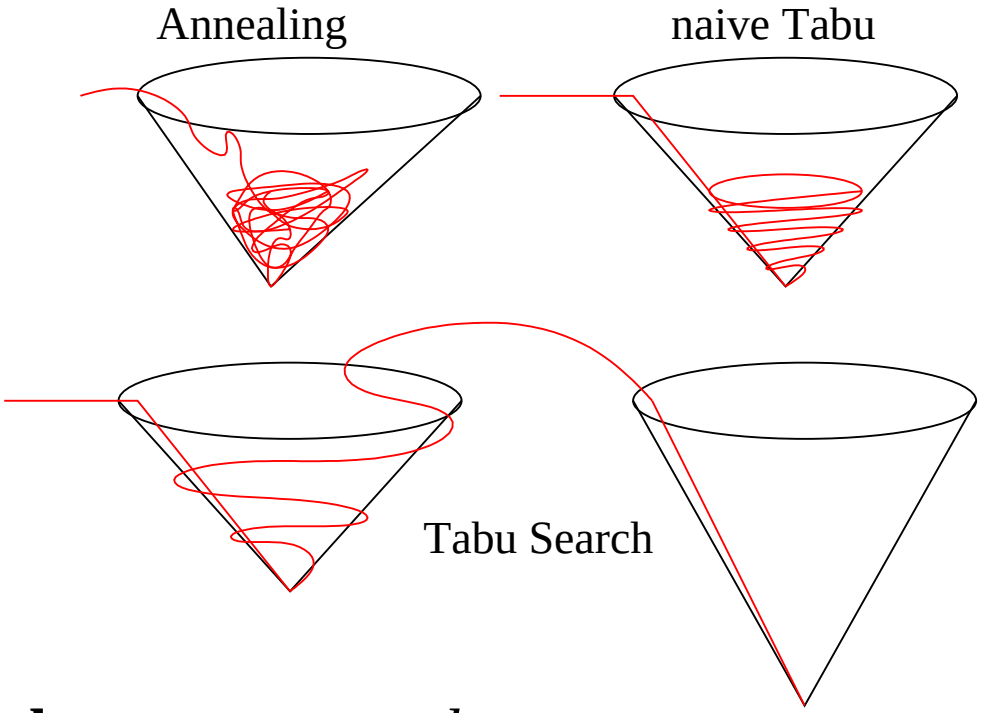
Mehrere Varianten von simulated annealing, und greedy Algorithmen.

Welcher Algorithmus am besten ist hängt vom Zeitbudget und den Instanzen ab.

Fixed-K-Annealing ist gut für dünne Zufallsgraphen.

Tabu-Suche

Idee: Annealing kann lange brauche um “Löchern” zu entkommen.



Erster Versuch: Verbiete letze k Zustände

$\rightsquigarrow$ keine Kreise d. Länge $\leq k$

Zweiter Versuch: Verbiete lokale Eigenschaften

Beispiel: Tabu-Listen-Einträge bei Graphfärben könnten Paare $($Knoten, Farbe$)$ sein.

Tabu-Suche als ausgewachsene Meta-Heuristik

- Auswahl der Startlösung
- Auswahlregel: $\mathcal{N}^* := \mathcal{N}(\mathbf{x}) \cap \mathcal{L}$.

  Wähle bestes $\mathbf{x} \in \mathcal{N}^*$, das den Regeln unten genügt.
- Tabu-Bedingungen verbieten Elemente
- Ausnahmebedingunen, z.B. verbesserte Lösung
- Zeitweise Veränderung der Zielfunktion zwecks

  Intensivierung und Diversifizierung
- Stopregeln

Tabu-Suche $\leftrightarrow$ Simulated Annealing

- $\square$ Ähnliche Anwendungen
- $+$ flexibler
- $-$ zu flexibel?
- $+$ Wechselt automatisch zwischen hill-climbing in unerforschten Gebieten und Strategien zum Entkommen aus lokalen Optima
- $-$ Weniger ansprechend?

12.6 Evolutionäre Algorithmen

Ausführliche Behandlung würde den Rahmen sprengen.

Verallgemeinerung von lokaler Suche:

- $\mathbf{x} \longrightarrow$ Population von Lösungskandidaten
- Reproduktion fitter Lösungen
- Mutation ähnlich lokaler Suche
- zusätzlich: geschlechtliche Vermehrung.

  Idee: erben guter Eigenschaften beider Eltern

Zusammenfassung

Wie geht man an ein Optimierungsproblem heran?

Welcher Lösungsansatz?

Der wissenschaftliche Standpunkt

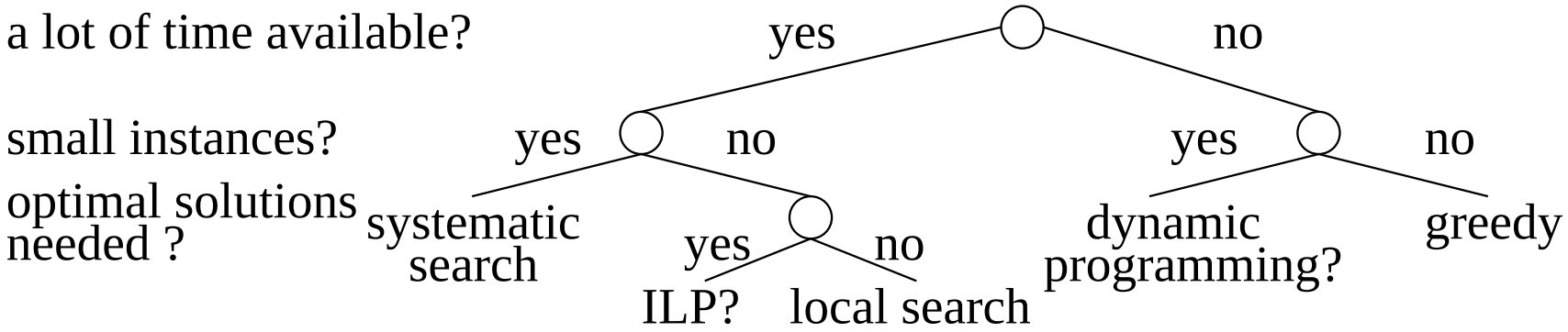
1. Das Problem verstehen
2. Gibt es eine einfache Lösung?
3. Literatur studieren auch für verwandte Probleme
4. Gibt es einen vielversprechenden Spezialalgorithmus?
5. Metaheuristiken ranken
6. Experiments sorgfältig entwerfen um Hypothesen zu testen
7. Implementiere und vergleiche die vielversprechendsten Ansätze
8. vermeide Dogmen
9. Einsicht kommt vor erschöpfenden Experimenten
10. Iterieren

Ökonomische Modifikationen

1. Das Problem wirklich verstehen
2. Existierende Software verwendbar?
3. Implementierungen flexibel halten
4. Prototyp mit einfachen Ansatz, z.B. Greedy. Ggf wiederverwenden
5. Plane brauchbare Lösung die pünktlich fertig wird
6. Zeit für Verbesserungen einplanen
7. Weitere Verbesserungen in späteren Versionen
8. Risikoreiche Ansätze in akademische Kooperationen auslagern.
9. Kleinere Auswahl von Ansätzen als in der reinen Lehre
10. Verfeinerungen einstellen, wenn Kosten- Nutzenverhältnis nicht mehr stimmt

Welchen Ansatz auswählen?

Wenn Speziallösungen und Literaturstudium nicht erbracht haben:



Zusammenfassung Vor- und Nachteile

Greedy: Einfach und schnell. Selten optimal. Manchmal Approximationsgarantien.

Systematische Suche: Einfach mit Werkzeugen z.B. (I)LP, SAT, constraint programming. Selbst gute Implementierungen mögen nur mit kleinen Instanzen funktionieren.

Linear Programming: Einfach und einigermaßen schnell. Optimal falls das Modell passt. Rundungsheuristiken ergeben Näherungslösungen

Dynamische Programmierung: Optimale Lösungen falls Teilprobleme optimal und austauschbar sind. Hoher Platzverbrauch.

Integer Linear Programming: Leistungsfähiges Werkzeug für optimale Lösungen. Gute Formulierungen können Einiges know how erfordern.

Lokale Suche: Flexibel und einfach. Langsam aber oft gute Lösungen für harte Probleme.

Hill climbing: einfach aber leidet an lokalen Optima.

Simulated Annealing und Tabu Search: Leistungsfähig aber langsam. Tuning kann unschön werden.

Evolutionäre Algorithmen: Ähnliche Vor- und Nachteile wie lokale Suche. Durch geschl. Vermehrung potentiell mächtiger aber auch langsamer und schwieriger gut hinzukriegen. Weniger zielgerichtet.