Algorithmen und Datenstrukturen Überblick Test 2

Styll Patrick

2022S

Im Folgenden biete ich einen Überblick über eventuell wichtige Probleme und Algorithmen - keine Garantie auf Korrektheit. Ich habe einiges, das dennoch wichtig ist, aber nicht in den Folien vorkommt, durch externe Literatur ergänzt:

- *Algorithmen - Eine Einführung* von *Prof.Dr. Ronald Rivest et al.*
- *An Elementary Approach to Design and Analysis of Algorithms* von *Lekh Raj Vermani* und *Shalini Vermani*
- *Algorithmen* von *Robert Sedgewick*

1 Polynomialzeitreduktion und NP-Spezialfälle

1.1 Reduktion

Was heißt Reduktion so ganz allgemein? Wenn wir ein Problem A auf Problem B reduzieren können, dann gilt $A \leq_P B$. Wir *blasen* Problem A im Prinzip auf die Größe von B auf. Konsequent gilt in diesem Fall dann, dass B mindestens so schwer ist, wie A.
Zum Beispiel: Wenn $A \leq_P B$ und B ist polynomiell lösbar, dann ist auch A polynomiell lösbar. Wenn B in NP liegt, so liegt auch A in NP.
Wichtig für den Beweis: Der ganze Spaß muss polynomiell sein. Außerdem müssen die Antworten gleich sein - das heißt, die Antwort für A lautet dann ja, wenn auch die Antwort für B ja ist.
Reduzieren kann man zB durch einfache Äquivalenz, Reduktion eines Spezialfalls auf den allgemeinen Fall oder auch durch Kodierung mit Gadgets (funktioniert zB gut bei der Reduktion von Vertex Cover auf Ham-Cycle).

1.2 P, NP, NPC, NPH

Zunächst ein paar Definitionen:

- **Zertifikat:** Ein Zertifikat ist ein beliebiger Input, der überprüft werden muss; wichtig dabei ist auch, dass der eine polynomiell-beschränkte Größe haben muss!

- **Zertifizierer:** Der überprüft mithilfe von den Zertifikaten oben die Ja-Instanzen - natürlich polynomiell.

(i) **P**: Das sind Ja/Nein-Probleme, für die polynomielle Algorithmen existieren. Zertifizier oder Zertifikate brauchen wir in dem Falle nicht.

(ii) **NP**: Das sind Ja/Nein-Probleme, für die (noch) keine polynomiellen Algorithmen gefunden worden sind, für die wir aber polynomielle Zertifizierer haben. Es gilt $P \subseteq NP$.

(iii) **NPH**: Ein Ja/Nein-Problem B ist NP-hard, falls für jedes Problem A in NP gilt, dass $A \leq_P B$. Wir können also jedes NP-Problem in Polynomialzeit auf NPH-Probleme reduzieren - NPH ist also mindestens so schwer wie jedes Problem in NP.

(iv) **NPC**: Ein Problem B ist NP-complete, falls es sowohl in NP liegt, als auch NP-hard ist. Das sind also die schwierigsten Probleme in NP.

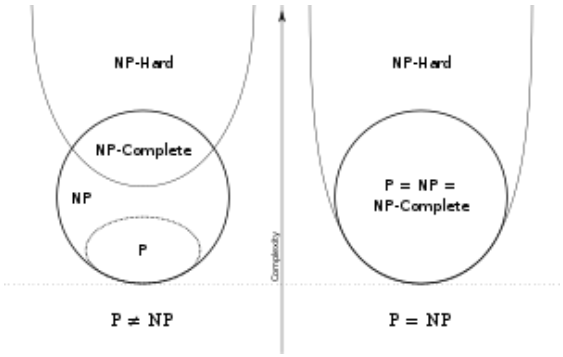
Formal gelten für NPC die Kriterien: Eine Sprache $L \subseteq \{0,1\}^*$ wird als NP-vollständig bezeichnet, wenn

1. $L \in NP$ ist und
2. $L' \leq_P L$ für jedes $L' \in NP$ gilt.

lWenn dann eine Sprache L die Eigenschaft 2 erfült, aber nicht notwendigerweise die Eigenschaft 1, dann gilt $L \in NPH$.

Außerdem ist noch gut zu wissen: Wenn ein NP-vollständiges Problem in polynomieller Zeit lösbar ist, dann gilt $P = NP$. Das heißt auch, wenn irgendein Problem aus NP nicht in polynomieller Zeit lösbar ist, dann ist kein NP-vollständiges Problem in Polynomialzeit lösbar.

Folgende Grafik solltet ihr euch auch noch einverleiben:

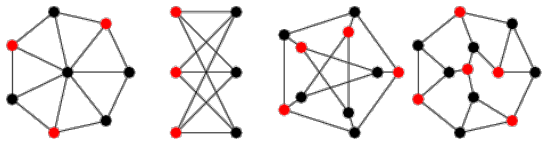


1.3 Problemstellungen und wichtige Reduktionen

1. **Independent Set**
   Ein Independent Set eines Graphen $G = (V, E)$ ist eine Teilmenge $S \subseteq V$, in der es keine zwei adjazenten Knoten gibt. In der Norm gilt dann die Problemstellung: Gegeben sei ein Graph $G = (V, E)$ und eine ganze Zahl k. Gibt es ein Independent Set S von G, sodass $|S| \leq k$? (k ist Teil des Inputs!)

2. **Vertex Cover**
   Ein Vertex Cover eines Graphen $G = (V, E)$ ist eine Menge $S \subseteq V$, sodass jede Kante des Graphen zu mindestens einem Knoten aus S inzident ist. Wir betrachten dann meistens: Gegeben sei ein Graph $G = (V, E)$ und eine ganze Zahl k. Gibt es ein Vertex Cover S von G, sodass $|S| \leq k$?

3. **Vertex Cover + Independent Set**
   Ganz wichtig zu merken: Ein Vertex Cover ist das Komplement zum Independent Set. Das ist dann gemäß dem Konversionslemma: Sei $G = (V, E)$ ein Graph und $S \subseteq V$ und $V = V - S$. S ist ein Independent Set von G genau dann, wenn C ein Vertex Cover von G ist.
   Aus einem Maximum Independent Set erhalten wir dann zum Beispiel ein Minimum Vertex Cover. In der folgenden Abbildung ist das Independent Set rot, das Vertex Cover ist schwarz.



4. **Vertex Cover (aber klein diesmal!)**
   Noch nicht die Hoffnung verlieren! Brute-Force ist zwar mit $O(kn^{k+1})$ nicht so toll, aber durch einen Algorithmus, welcher $O(2^k kn)$ bietet, wird das Problem für kleine k so halbwegs handhabbar:

```
Vertex-Cover(G,k):
if G enthält keine Kanten
    return true
if G enthält > k(n − 1) Kanten
    return false
Sei (u, v) eine beliebige Kante von G
a ← Vertex-Cover(G − {u}, k − 1)
b ← Vertex-Cover(G − {v}, k − 1)
return a OR b
```

   Und die dazugehörigen Rekursionsgleichungen, können ja nicht schaden:

$$T(n,k) \leq \begin{cases} c & \text{falls } k = 0 \\ cn & \text{falls } k = 1 \\ 2T(n-1,k-1) + ckn & \text{falls } k > 1 \end{cases} \Rightarrow \quad T(n,k) \leq 2^k ckn$$

5. **Independent Set auf Bäumen**
   Auch dafür gibt es einen eigenen Greedy Algorithmus, welcher ein maximales Independent Set in einem Wald $T = (V, E)$ findet - dabei ist jede Zusammenhangskomponente von T ein Baum. Das Ganze kann in Laufzeit $O(n)$ implementiert werden, wenn man es in Postorder (links, rechts, Wurzel) durchmustert:

```
Independent-Set-In-A-Forest(T):
S ← ∅
while T hat zumindest eine Kante
    Sei e = (u, v) eine Kante, sodass v ein Blatt ist
    Füge v zu S hinzu
    Lösche aus V die Knoten u und v (und alle
    zu diesen Knoten inzidenten Kanten)
S ← S ∪ V
return S
```

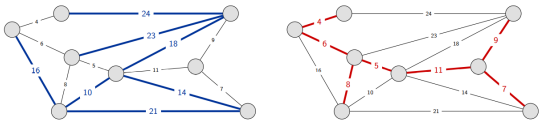
   Das Ganze in gewichteter Form gibt es dann sogar noch durch dynamische Programmierung, ebenfalls in $O(n)$:

```
Weighted-Independent-Set-In-A-Tree(T):
Wähle eine Wurzel r aus
foreach Knoten u von T in Postorder
    if u ist ein Blatt
        M_in[u] ← w_u
        M_out[u] ← 0
    else
        M_in[u] ← w_u + Σ_{v∈Nachfolger(u)} M_out[v]
        M_out[u] ← Σ_{v∈Nachfolger(u)} max{M_in[v], M_out[v]}
return max{M_in[r], M_out[r]}
```

6. **Nicht-Blockierer**
   Gegeben ist ein Graph $G = (V, E)$ mit reellwertigen Kantengewichten $c_e = c_{uv} = c_{vu}$ für $e = (u, v) \in E$. Ein Nicht-Blockierer ist eine Teilmenge der Kanten $N \subseteq E$, sodass es für alle Knotenpaare $u, v \in V$ einen $u-v$-Pfad in G gibt, der keine Kante aus N enthält. Die Kosten des Nicht-Blockierers ist gegeben durch $\sum_{n \in N} c_e$. Ein maximaler Nicht-Blockierer ist dann ein Nicht-Blockierer mit größten Kosten.
   Wir betrachten das Problem: Besitzt G einen Nicht-Blockierer mit Kosten $\geq$ einer gegebenen Zahl k?

7. **Spannbaum**
   Dazu muss ich wohl nicht mehr sagen - wir betrachten wieder mal minimale Spannbäume, also Spannbäume mit kleinsten Kosten. Wir betrachten das Problem: Gegeben ist ein gewichteter Graph G und eine Zahl k. Besitzt G einen Spannbaum mit Kosten $\leq k$?

8. **Spannbäume + Nicht-Blockierer**
   Wie vorhin, das Komplement zu einem MNB ist der MST. Gemäß Konversionslemma: Sei $G = (V, E)$ ein gewichteter Graph, $N \subseteq E$ und $T = E - N$. Dann ist N ein maximaler Nicht-Blockierer genau dann, wenn T ein minimaler Spannbaum ist. Die Kosten von N sind genau $K := \sum_{e \in E} c_e$ minus der Kosten von T.
   Und da wir ja MST in P lösen können (zB Prim oder Kruskal), dann können wir auch MNB in P lösen. In der folgenden Abbildung ist MNB blau und MST rot.



N, Kosten = 126 T, Kosten = 50

9. **Set Cover (Mengenüberdeckungsproblem)**
   Gegeben sei eine Menge U von Elementen und eine Menge $S = \{S_1, S_2, \ldots, S_m\}$ von Teilmengen von U. Ein Set Cover ist eine Teilmenge $C \subseteq S$, also eine Menge von Mengen, deren Vereinigung U entspricht. C ist ein Set Cover von S.
   Wir betrachten das Problem: Existiert für eine gegebene ganze Zahl k eine Teilmenge $C \subseteq S$ mit $|C| \leq k$, sodass die Vereinigung von C gleich U ist?
   Beispiel: $\{a, b\} \cup \{b, c\}$ wäre eine Lösung für $U = \{a, b, c\}$.
   Man kann in speziellen Fällen Vertex Cover auf Set Cover reduzieren und somit zeigen, dass das Problem in NPC liegt.

10. **Exact Cover (Problem der exakten Überdeckung)**
    Ähnlich wie Set Cover, nur halt eine exakte Überdeckung. Beispiel: $\{a, b\} \cup \{c\}$ wäre eine Lösung für $U = \{a, b, c\}$, aber $\{a, b\} \cup \{b, c\}$ ist diesmal nicht erlaubt.
    Liegt ebenso in NPC.

11. **Das Cliquenproblem**
    Eine Clique in einem ungerichteten Graphen $G = (V, E)$ ist eine Teilmenge $V' \subseteq V$ von Knoten, von denen je zwei Knoten durch eine Kante E verbunden sind. Mit anderen Worten ist jede Clique ein vollständiger Teilgraph von G. Die Größe ist die Anzahl der in ihr enthaltenen Knoten. Das Cliquenproblem ist das Optimierungsproblem, eine Clique maximaler Größe in einem Graphen zu bestimmen. Als Entscheidungsproblem formuliert fragen wir einfach, ob eine Clique gegebener Größe k im Graphen existiert. Die formale Definition lautet:

    $$\text{CLIQUE} = \{\langle G, k\rangle : \text{G ist ein Graph mit einer Clique der Größe } k\}.$$

    Dieses Problem ist ebenso NPC.

12. **SAT**
    Satisfiability (SAT) ist im Prinzip: Gegeben ist eine KNF-Formel ϕ. Gibt es eine Wahrheitsbelegung, die ϕ erfüllt?

    Davon gibt es einige Formen, wie etwa:

- *3-SAT:* SAT, bei dem jede Klauseln genau 3 Literale enthält.
- *3-CNF-SAT:* Aussagenlogische Formel in konjunktiver Normalform, wobei eine Klausel genau 3 Literale enthält.
- *CIRCUIT-SAT:* Das Erfüllbarkeitsproblem, nur eben für Schaltkreise.
- *MAX-SAT*
- *MAX-3SAT*

. . . sind alle klarerweise NPC.

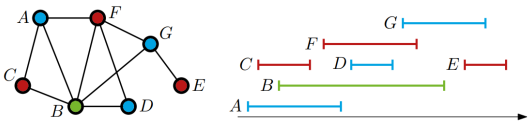
13. **HAM-CYCLE**
Gegeben sei ein ungerichteter Graph G. Existiert ein Kreis C in G, der alle Knoten von G genau einmal enthält? (Hamiltonkreisproblem mit gegebenem k) Ebenso NPC.

14. **Travelling Salesperson Problem**
Man sucht eine Reihenfolge für den Besuch mehrerer Orte, sodass die gesamte Reisestrecke eiens Handlungsreisenden möglichst kurz ist (Minimierungsproblem). Außerdem muss der erste Ort gleich dem letzten Ort sein.

15. **HAM-CYCLE + TSP**
Man kann das Hamiltonkreisproblem auf das TSP reduzieren und somit zeigen, dass dies auch in NPC liegt. Es gilt also $HAM - CYCLE \leq_P TSP$. Eine TSP-Instanz besitzt also eine Tour (einer gewissen Länge) genau dann, wenn ein Graph G einen Hamiltonkreis besitzt.

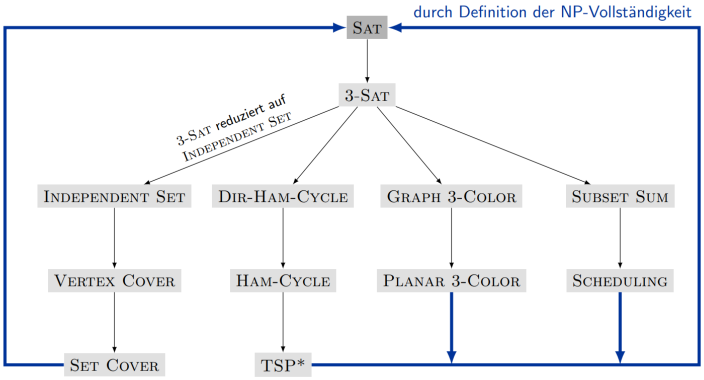
16. **k-COLOR (Knotenfärbeproblem)**
Gegeben sei ein ungerichteter Graph G. Kann man die Knoten des Graphen mit $k \geq 3$ Farben so einfärben, dass benachbarte Knoten nicht die gleiche Farbe haben? Wichtig: k-COLOR ist NPC für alle $k \geq 3$, aber z.B. in P für $k = 2$(Stichwort bipartit)!

    Als Optimierungsproblem betrachten wir dies durch OPT-COLOR: Gegeben sei ein ungerichteter Graph G. Färbe die Knoten des Graphen mit einer minimalen Anzahl Farben so, dass benachbarte Knoten nicht die gleiche Farbe besitzen.
    Und das können wir für spezielle Fälle sogar lösen - das wären dann die Intervallgraphen. Hierbei stellen wir einen Graphen durch Intervalle dar, die sich nur überschneiden, falls die dazugehörigen Knoten indizent sind. Und so ist die halbe Arbeit schon erledigt, das Einfärben geht nun ganz leicht: All die Intervalle, die sich nicht überschneiden, können wir zu einer Ebene zusammenfassen - jede Ebene bekommt dann eine eigene Farbe. Hier ein Beispiel:



Hier passen zB A, D und G zusammen, weshalb sie eine Ebene bilden, ergo eine Farbe zugewiesen bekommen.
Wichtig: Der ganze Algorithmus läuft dann mit n Knoten in $O(n \log n)$.

Und da gibt es noch einige NPC-Probleme mehr ...



2 Branch and Bound

2.1 Kombinatorische Optimierung

Es geht bei der kombinatorischen Optimierung darum, aus einer (großen) Menge von diskreten Elementen eine Teilmenge zu konstruieren, die einerseits gewissen Nebenbedingungen entspricht und andererseits bezüglich einer Kostenfunktion optimal ist.

2.2 Definition

Beschränke eine auf Divide-and-Conquer basierende systematische Durchmusterung aller Lösungen mit Hilfe von Methoden, die untere und obere Schranken liefern, und ermittle eine optimale Lösung. Wichtig ist hierbei immer die Wahl der Heuristiken für obere und untere Schranken, welche Teilinstanz ausgewählt wird und wie das Branching erfolgt. Wird allgemein für kombinatorische Optimierungsprobleme eingesetzt und funktioniert für Maximierungs- und Minimierungsprobleme. Praktisch lassen sich oft hohe Beschleunigungen erreichen, die worst-case Laufzeit bleibt jedoch wie bei der Enumeration.

2.3 Auswahl der Probleme - Branching

1. **Best-first**
   Es wird jeweils ein Teilproblem mit der besten dualen Schranke (also der größten oberen Schranke bei Maximierungsproblemen, der kleinsten unteren Schranke bei Minimierungsproblemen) ausgewählt, wodurch immer die kleinstmögliche Anzahl an Teilproblemen abgearbeitet wird.

2. **Depth-first**
   Es wird jeweils ein zuletzt erzeugtes Teilproblem weiter bearbeitet (vgl. DFS), wodurch man meist am raschesten eine vollständige und gültige Näherungslösung erhält.

In der Praxis wird mit einer Depth-first Strategie begonnen und nach Erhalt einer gültigen Lösung mit Best-first fortgesetzt um die Vorteile zu kombinieren.

2.4 Allgemeine Branch-and-Bound Verfahren

- ad Maximierungsprobleme:

Eingabe: Instanz I

```
Branch-and-Bound-Max(I):
L ← −∞ oder Wert einer initialen heuristischen Lösung
Π ← {I}
while ∃I' ∈ Π
    Entferne I' aus Π
    Berechne für I' lokale obere Schranke U' mit Dualheuristik
    if U' > L
        Berechne für I' gültige heuristische Lösung → untere Schranke L'
        if L' > L
            L ← L'
        if U' > L
            Partitioniere I' in Teilinstanzen I_1,...,I_k
            Π = Π ∪ {I_1,...,I_k}
return beste gefundene Lösung mit Wert L
```

Bounding – Fall $U' \leq L$ nicht weiter interessant.
Branching.

- ad Minimierungsprobleme:

Eingabe: Instanz I

```
Branch-and-Bound-Min(I):
U ← ∞ oder Wert einer initialen heuristischen Lösung

Π ← {I}
while ∃I' ∈ Π
    Entferne I' aus Π
    Berechne für I' lokale untere Schranke L' mit Dualheuristik
    if L' < U
        Berechne für I' gültige heuristische Lösung → obere Schranke U'
        if U' entspricht einer gültigen Lösung für I'
            if U' < U
                U ← U'
        if L' < U
            Partitioniere I' in Teilinstanzen I_1,...,I_k
            Π ← Π ∪ {I_1,...,I_k}
return beste gefundene Lösung mit Wert U
```

Bounding - Fall $L' \geq U$ nicht weiter interessant. Branching.

2.5 Rucksackproblem

Beim Rucksackproblem geht es darum, bei n Gegenständen mit positiven rationalen Gewichten $g_1, \ldots, g_n$ und Werten $w_1, \ldots, w_n$ eine Teilmenge S der Gegenstände mit Gesamtgewicht $\leq$ einer gegebenen Kapazität G und maximalem Gesamtwert zu finden. Wir wollen also den Wert der gewählten Gegenstände maximieren und das Gewicht der gewählten Gegenstände unter einem gewissen Schwellwert halten.

Dafür gibts einen Enumerationsalgorithmus, wobei eine Enumeration aller zulässigen Lösungen allen Teilmengen unserer n-elementigen Gesamtmenge entspricht. Mit 2^U Teilmengen ist die Laufzeit von $O(2^n)$ auch ziemlich sch . . . suboptimal.
Heil bringt hierfür Branching and Bounding, die das ganze Verfahren ordentlich einschränkt, aber immer noch die optimale Lösung ausspuckt. Hierfür gibt es die globale untere Schranke L (bei Maximierungsproblemen), die globale obere Schranke U (bei Minimierungsproblemen), und die dazugehörigen lokalen Schranken L' und U'. Diese berechnet man durch Greedy-Algorithmen, wobei die Priorität durch $\frac{w_i}{g_i}$ festgelegt wird.

- ad L':
  Man durchläuft alle Gegenstände, deren Variablen noch nicht festgelegt sind, in der sortierten Reihenfolge und packt den jeweils aktuellen Gegenstand ein, falls noch Platz im Rucksack ist.
- ad U': $\quad U' \leftarrow w_{curr} + (G - g_{curr}) \cdot \frac{w_i}{g_i}$

Teilprobleme mit $U' \leq L$ (Maximierungsproblem!) werden nicht länger verfolgt.

2.6 Minimales Vertex Cover

Zunächst die Definition eines *nicht erweiterbaren Matchings:*
Ein nicht erweiterbares Matching (maximales Matching) bedeutet, dass es keine Kante $e \in E \backslash M$ gibt, sodass $\{e\} \cup M$ ein gültiges Matching ist. Das ist nicht notwendigerweise ein größtes Matching. Das ganze kann man durch einen einfachen Greedy-Algorithmus bestimmen: Nimm irgendeine Kante und entferne dann die die dazugehörigen Knoten und deren inzidenten Kanten aus dem Graphen. Mach das, bis es keine Kanten mehr gibt - fertig. Die Anzahl der gewählten Kanten ist dann die Größe des Matchings.

- ad L'
  Hierfür nimmt man einfach die Größe des zuvor besprochenen, nicht erweiterbaren Matchings - Greedy Algorithmus! - addiert mit $|C'|$.

- ad U'
  Wird auch durch einen Greedy Algorithmus bestimmt: Gehe alle Knoten nichtsteigend nach Knotengrad durch, solange der Graph Kanten enthält - füge den Knoten mit höchstem Knotengrad zur Auswahl hinzu und entferne ihn und all seine inzidenten Kanten aus dem Graphen. Wenn du fertig bist, nimm die Anzahl der Auswahl und addiere sie zu $|C'|$.

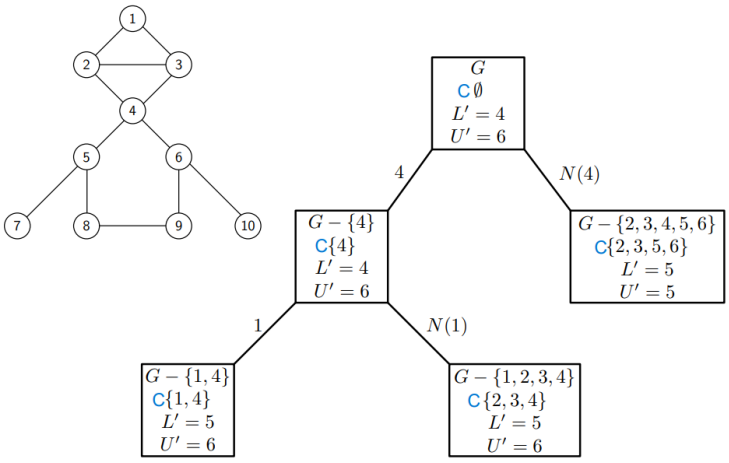
Der Algorithmus lautet dann:

Eingabe: Graph $G = (V, E)$ und Knotenmenge $C = \emptyset$

```
MinVertexCover-BranchAndBound(G,C):
U ← |V| − 1
Π ← {(G, C)}
while ∃I' ∈ Π
    Entferne I' aus Π
    Berechne für I' = (G',C') lokale untere Schranke L' mit Matchingheuristik
    if L' < U
        Berechne für I' gültige heuristische Lösung mit Greedyheuristik
            → obere Schranke U'
        if U' < U
            U ← U'
        if L' < U
            u_max ← Knoten mit maximalem Grad in G'
            Erzeuge Teilinstanzen I_1 = (G' − {u_max}, C ∪ {u_max}) und
                I_2 = (G' − {u_max} − N(u_max), C ∪ N(u_max))
            Π ← Π ∪ {I_1, I_2}
return beste gefundene Lösung mit Wert U
```

$N(u_{\max})$ alle Nachbarknoten von $u_{\max}$

Und da man es an einem Beispiel oft besser sieht:



3 Dynamisches Programmieren

3.1 Definition

Dynamische Programmierung kann dann eingesetzt werden, wenn das Problem aus vielen gleichartigen Teilproblemen besteht und eine optimale Lösung sich aus optimalen Lösungen der Teilprobleme zusammensetzt.

Folglich teilen wir das Problem in eine Folge von überlappenden Teilproblemen auf und erstellen und speichern die Lösungen für immer größere Teilprobleme unter Verwendung der abgespeicherten Lösungen. Ein wesentlicher Aspekt ist somit auch die Speicherung (memoization) von Ergebnissen für Subprobleme zur Wiederverwendung.
Gemäß *Optimalitätsprinzip von Bellman* führt dies zu einem optimalen Ergebnis genau dann, wenn es sich aus den optimalen Ergebnissen der Subprobleme zusammensetzt - die Effizienz hängt hierbei von der Aufteilung und Ermittlung der partiellen Lösungen ab.

3.2 Gewichtetes Interval Scheduling

Sollten alle Gewichte gleich sein, so funktioniert auch ein einfacher Greedy-Algorithmus, welcher sie nach Beendigungszeit in aufsteigender Reihenfolge betrachtet. Für beliebige Gewichte ist dahingegen dynamische Programmierung optimal. Hierbei berechnen wir Teillösungen immer durch:

$$OPT(j) = \begin{cases} 0 & \text{wenn } j = 0 \\ \max\{w_j + OPT(p(j)), OPT(j-1)\} & \text{sonst} \end{cases}$$

Der Algorithmus (iteratives Bottom-up, auch rekursives Top-down möglich) lautet dann wie folgt:

```
Iterative-Compute-Opt():
M[0] ← 0
for j ← 1 bis n
    M[j] ← max(w_j + M[p(j)], M[j − 1])
```

Wichtig: Die Laufzeit wird durch das Abspeichern extrem heruntergeschraubt. Der rekursive Algorithmus (ohne Abspeichern) würde wegen redundanter Subprobleme exponentiell entarten. Dahingegen liegt die Laufzeit hier durch Abspeichern bei $O(n)$.
Die Lösung müssen wir uns dann aber noch erarbeiten, was durch folgenden Algorithmus geschieht:

```
Find-Solution(j):
if j = 0
    Keine Ausgabe
elseif w_j + M[p(j)] > M[j − 1]
    Gib j aus
    Find-Solution(p(j))
else
    Find-Solution(j − 1)
```

3.3 Segmented Least Squares

Problemstellung: Wir haben eine Menge von n Punkten gegeben und wollen diese durch eine Folge von Geraden approximieren, welche eine bestimmte Funktion $f(x)$ minimieren.

Unsere Aufgabe ist also, eine angemessene Wahl für $f(x)$ zu finden, welche Genauigkeit (Höhe des Fehlers) und Sparsamkeit (Anzahl der Geraden) gewährleistet - dieser Tradeoff kann dargestellt werden durch ...

$$E + cL \qquad c > 0$$

... wobei E die Summe der quadrierten Fehler in jedem Segment und L die Anzahl der Geraden sind. Auch hier verschafft uns dynamische Programmierung Erleichterung durch:

$$OPT(j) = \begin{cases} 0 & \text{falls } j = 0 \\ \min\limits_{1 \le i \le j} \{OPT(i-1) + e(i,j) + c\} & \text{sonst} \end{cases}$$

Der Algorithmus läuft dann in $O(n^3)$ *(Hinweis: Kann auch auf* $O(n^2)$ *optimiert werden)* und lautet:

```
Segmented-Least-Squares( P = {p_1, p_2, ..., p_n} )
    M[0] = 0
    for j ← 1 bis n
        for i ← 1 bis j
            berechne Fehler e(i,j) für Punkte p_i, ..., p_j

    for j ← 1 to n
        M[j] = min_{1≤i≤j} (M[i-1] + e(i,j) + c)

    return M[n]
```

3.4 Rucksackproblem

Dynamische Programmierung verschafft auch einen anderen Blickwinkel auf das Rucksackproblem und liefert uns einen Algorithmus in Laufzeit und Speicherkomplexität von $O(nG)$, wobei $nG < 2^n$ ist. **Achtung:** Der Algorithmus ist zwar polynomiell in n, aber er hängt auch von der Rucksackkapazität G ab - und diese ist nun mal exponentiell in der Eingabelänge, weil Zahlen binär kodiert werden. Sowas nennt man dann auch *pseudopolynomiell*. Allgemein kann das Rucksackproblem ohnehin nicht in P gelöst werden, falls $P \neq NP$ gilt.

Das Array befüllt man dann durch die Gleichung:

$$OPT(i,g) = \begin{cases} 0 & \text{wenn } i = 0 \\ OPT(i-1,g) & \text{wenn } g_i > g \\ \max\{OPT(i-1,g), \quad w_i + OPT(i-1, g-g_i)\} & \text{sonst} \end{cases}$$

Der rekursive Top-Down Algorithmus funktioniert dann durch:

Rucksack: Befülle ein $(n+1) \times (G+1)$ Array.

Eingabe: n, G, $g_1, \ldots, g_n$, $w_1, \ldots, w_n$

```
for g ← 0 bis G
    M[0,g] ← 0

for i ← 1 bis n
    for g ← 0 bis G
        if g_i > g
            M[i,g] ← M[i − 1,g]
        else
            M[i,g] ← max{M[i − 1,g], w_i + M[i − 1, g − g_i]}
return M[n,G]
```

Die Lösung findet man durch Backtracking:

```
Find-Solution(M):
i ← n
k ← G
A ← ∅
while i > 0 und k > 0
    if M[i,k] ≠ M[i-1,k]
        A ← A ∪ {i}
        k ← k − g_i
    i ← i − 1
return A
```

Da eigentlich nicht das gesamte Array gespeichert werden muss, kann man hier auch einsparen: einfach nur die letzte Zeile merken, das verbessert die Speicherkomplexität erheblich.

3.5 Kürzeste Pfade

Achtung ist hierbei geboten: negative Kantengewichte sind erlaubt, weshalb der Dijkstra-Algorithmus scheitert. Allgemein gilt, dass das Finden eines kürzesten Pfades in einem gerichteten Graphen mit Kantengewichten in NPC liegt. Verbietet man aber negative Kreise, so ist das Problem wieder in P lösbar! Hierfür sind **Bellman's Gleichungen** von großer Bedeutung:

Sei $G = (V, E)$ ein gerichteter Graph ohne negative Kreise mit Kantengewichten c_{vw} und $t \in V$, dann gilt für die Länge $OPT(v)$ eines kürzesten $v - t$ Pfades P in G:

$$OPT(t) = 0$$

$$OPT(v) = min_{(v,w)\in E}\{c_{vw} + OPT(w)\}, \forall v \in V, v \neq t$$

Für den Bellman-Ford Algorithmus zum Finden kürzester Pfade in Graphen ohne negative Kreisen gilt dann folgende, wichtige Gleichung:

$$OPT(i,v) = \begin{cases} 0 & \text{wenn } i = 0 \text{ und } v = t \\ \infty & \text{wenn } i = 0 \text{ und } v \neq t \\ \min\left\{OPT(i-1,v), \min\limits_{(v,w)\in E}\{c_{vw} + OPT(i-1,w)\}\right\} & \text{ansonsten} \end{cases}$$

Der Algorithmus besitzt dann eine Speicherkomplexität von $O(n^2)$ und eine Laufzeit von $O(mn)$ (langsamer als Dijkstra):

```
Shortest-Path(G,s,t):
foreach node v ∈ V
    M[0,v] ← ∞
M[0,t] ← 0
for i ← 1 bis n − 1
    foreach Knoten v ∈ V
        M[i,v] ← M[i − 1,v]
    foreach Kante (v,w) ∈ E
        M[i,v] ← min (M[i,v],c_vw + M[i − 1,w])
return M[n − 1,s]
```

Wichtig ist noch anzumerken, dass der Algorithmus nach spätestens $n-1$ Iterationen konvergiert.

Aber es kommt noch mehr: man kann nach dem Terminieren des Algorithmus sogar auf negative Kreise überprüfen:

(i) Wenn $OPT(n,v) < OPT(n-1,v)$ für einen Knoten v, dann enthält (ein beliebiger) kürzester $v-t$ Kantenzug K mit maximal n Kanten einen Kreis W. Außerdem hat W negative Kosten.

(ii) Falls G einen negativen Kreis enthält, von dem aus t erreicht werden kann, dann gibt es eine Kante (v,u), sodass $OPT(n-1,v) > c_{vu} + OPT(n-1,u)$ (und somit $OPT(n,v) < OPT(n-1,v)$).

Hieraus folgt das Theorem: G enthält einen negativen Kreis von dem aus t erreicht werden kann genau dann, wenn ein Knoten v mit $OPT(n,v) < OPT(n-1,v)$ existiert. Dies kann mit Laufzeit $O(nm)$ entschieden werden.

Verbesserungen: Verwalte zB nur ein Array $M[v]$ = kürzester $v-t$ Pfad, den wir bisher gefunden haben. Der Algorithmus darf abgebrochen werden, sobald sich nach einer vollen Iteration kein Eintrag mehr in M geändert hat. Dies bewirkt, dass die worst-case Laufzeit in $O(mn)$ (in der Praxis besser) und die Speicherkomplexität in $O(n+m)$ liegt.

4 Approximation

4.1 Definition

Erzeuge in polynomieller Zeit eine Näherungslösung, die eine Gütegarantie besitzt. Die Güte eines Algorithmus sagt etwas über die Fähigkeit aus, optimale Lösungen gut oder schlecht anzunähern.

ad Gütegarantie: Sei A ein Algorithmus, der für jede Instanz x eines Problems X eine gültige Lösung mit Lösungswert $c_A(x) > 0$ liefert. Sei $c_{opt}(x) > 0$ der Wert einer optimalen Lösung.

Für Minimierungsprobleme mit $\varepsilon \geq 1$:

$$\frac{c_A(x)}{c_{opt}(x)} \leq \varepsilon$$

Für Maximierungsprobleme mit $\varepsilon \in [0, 1]$:

$$\frac{c_A(x)}{c_{opt}(x)} \geq \varepsilon$$

Für diese ε nennt man einen Approximationsalgorithmus dann ε-Approximationsalgorithmus, unser ε ist dann die Gütegarantie.
Beispiel: Haben wir einen 2-Approximationsalgorithmus, so ist dieser offensichtlich für ein Minimierungsproblem - es wird außerdem gesagt, dass im worst-case das Doppelte der Optimallösung ausgespuckt wird.
Wichtig: Bei $\varepsilon = 1$ haben wir einen exakten Algorithmus vor uns.

4.2 Approximationsalgorithmen für Minimum Vertex Cover

Ein 2-Approximationsalgorithmus, welcher durch Adjazenzlisten mit einer Laufzeit von $O(n + m)$ implementiert werden kann:

```
Approx-Vertex-Cover(G):
C ← ∅
while E ≠ ∅
    Wähle eine beliebige Kante (u, v) ∈ E
    C ← C ∪ {u, v}
    Entferne aus E alle Kanten, die inzident
          zu u oder v sind
return C
```

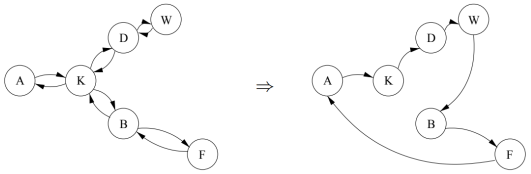
Alternativ mit logarithmischer Gütegarantie:

```
Approx-Vertex-Cover2(G):
C ← ∅
while E ≠ ∅
    Wähle einen Knoten u mit maximalem Grad im aktuellen Graphen
    C ← C ∪ {u}
    Entferne aus E alle Kanten, die inzident zu u sind
return C
```

4.3 Spanning-Tree-Heuristik für das symmetrische TSP

Definition des symmetrischen TSP: Gegeben ist ein ungerichteter vollständiger Graph $G = (V, E)$ mit Distanzmatrix c mit $c_{ii} = +\infty$ und $c_{ij} \geq 0$. Für alle Knotenpaare (i, j) sind die Distanzen in beide Richtungen identisch, d.h. es gilt $c_{ij} = c_{ji}$. Jede Tour hat dieselbe Länge in beide Richtungen.

Für die Spanning-Tree-Heuristik gehen wir nun so, vor, dass wir eine Tour aus einem MST für den Graphen G ableiten: So bestimmen wir zunächst einen MST, verdoppeln darin alle Kanten und bestimmen eine Eulertour, die hierbei alle Kanten enthält. Anschließend wählen wir von einem Startknoten aus eine Tour, welche wir so durchlaufen, dass jeder Knoten nur ein einziges Mal durchlaufen wird. Das schaut graphisch so aus:



Das Ganze funktioniert dann schließlich, vor allem auch aufgrund des Bestimmens eines MST, mit Laufzeit $O(n^2)$.

Gut zu wissen: Eine Eulertour existiert in einem ungerichteten Graphen genau dann, wenn er zusammenhängend ist und jeder Knoten einen geraden Knotengrad hat. Durch die Verdopplung der Kanten hat jeder Knoten einen geraden Grad, wodurch eine Eulertour immer existiert.

4.4 Algorithmus von Hierholzer

Dieser ausgeklügelte Algorithmus findet eine Eulertour, falls eine existiert, in linearer Zeit $O(n + m)$. Die grundlegende Idee basiert darauf, auf Zyklen iterativ Zyklen zu bauen. Der Algorithmus funktioniert nun folgendermaßen:

Wähle irgendeinen beliebigen Knoten des Graphen und konstruiere ausgehend hiervon einen geschlossenen Pfad, der keine Kante im Graphen zweimal durchläuft - das ist ein Zyklus. Wenn der eh schon alle Knoten durchläuft und somit eine Eulertour ist, können wir schon aufhören. Andernfalls löschen wir alle Kanten, die wir im Zyklus durchlaufen, aus dem Graphen heraus. An einem Knoten des Zyklus, dessen Grad >0 ist, lässt man einen neuen Zyklus entstehen, der keine Kante im Graphen zweimal enthält. Und die

beiden Zyklen verschmelzen wir jetzt - der Startpunkt von Zyklus 2 wird beim ersten Auftreten von Zyklus 1 durch alle Knoten von Zyklus 2 in der durchlaufenen Reihenfolge ersetzt. Das ganze machen wir so lange, bis wir eine Eulertour gefunden haben.

4.5 2-Approximierbarkeit des TSP

Fakt ist, dass, sollte $P \neq NP$ gelten, es keinen polynomiellen 2-Approximationsalgorithmus für das symmetrische TSP geben kann. Es gilt sogar, dass für ein allgemeines $\varepsilon > 1$ gezeigt werden kann, dass es keinen polynomiellen ε-Approximationsalgorithmus für das symmetrische TSP gibt. **Das symmetrische TSP ist nicht approximierbar.**

Ausnahme - Metrisches TSP: Ein TSP heißt metrsich, wenn für die Distanzmatrix C die Dreiecksungleichung gilt ... in summa summarum heißt das, dass sich zwischen Knoten keine Umwege auszahlen, sondern immer der direkte Weg optimal ist.
Auch das *Euklidische TSP* ist metrisch, da hierbei die Knoten Punkten in der euklidischen Ebene entsprechen und die Distanzen euklidische Distanzen sind.

Schließlich besitzt das metrische TSP eine Gütegarantie von 2.

4.6 Lastverteilung

Gegeben haben wir hierbei m identische Maschinen, n Jobs j und dazugehörige Bearbeitungszeiten t_j. Es gilt, dass die Bearbeitung von Jobs nicht unterbrochen werden darf, und eine Maschine nur einen Job auf einmal ausführen kann.
Die Last einer Maschine ist dann die Summe aller zugewiesenen Jobs, die Bearbeitungsdauer (*makespan*) im Gesamten ist dann die maximale Last auf irgendeiner Maschine.
Unsere Aufgabe ist nun, jeden Job einer Maschine so zuzuteilen, dass der makespan minimal gehalten wird - wir habens ja stressig. Der Algorithmus funktioniert dann folgendermaßen:

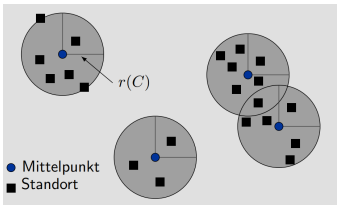
```
List-Scheduling(m,n,t_1,t_2,...,t_n):
for i ← 1 bis m
    L_i ← 0
    J_i ← ∅
for j ← 1 bis n
    i = argmin_{k=1,...,m} L_k
    J_i ← J_i ∪ {j}
    L_i ← L_i + t_j
return J_1,...,J_m
```

□ *Maschine i hat geringste Last*

Durch Verwenden einer Priority-Queue läuft das ganze dann in $O(n \log m)$. Des Weiteren besitzt der Algorithmus eine Gütegarantie von 2.

4.7 Center Selection

Gegeben haben wir eine Menge n von Standorten $s_1, \ldots, s_n$ und eine ganze Zahl $k > 0$. Unsere Aufgabe ist es nun, k Mittelpunkte C zu wählen, sodass die maximale Distanz von einem Standort zu einem nächsten Mittelpunkt minimal gehalten wird - hierfür gibt es im Prinzip unendlich viele potentielle Lösungen. Graphisch sieht das Ganze so aus:



Der Algorithmus funktioniert folgendermaßen:

```
Greedy-Center-Selection(k,n,s_1,s_2,...,s_n)
C = {s_1}
wiederhole k − 1 mal
    Wähle einen Standort s_i mit maximaler dist(s_i,C)
    Füge s_i zu C hinzu
return C
```

Standort am weitesten von jedem Mittelpunkt.

Gut zu wissen: Der Algorithmus besitzt eine Gütegarantie von 2 - und besser als das geht es gar nicht, falls $P \neq NP$ gilt!

5 Heuristische Verfahren

5.1 Definition

Erzeuge in polynomieller Zeit eine Näherungslösung. In der Praxis kann eine solche Lösung häufig sehr gut oder sogar optimal sein, es gibt aber *keine Garantie* dafür. Hierfür gibt es:

- Konstruktionsverfahren
- Verbesserungsheuristiken - Lokale Suchverfahren
- Metaheuristiken
  - Simulated Annealing
  - Tabu Suche
  - Evolutionäre Algorithmen

5.2 Konstruktionsverfahren

Diese sind meist sehr problemspezifisch, intuitiv gestaltet, und meistens Greedy - solange wir keine vollständige Lösung haben, fügen wir immer wieder die besten Teillösungen hinzu. Beispiel hierfür ist etwa auch Prim oder Kruskal für MST. Auch gibt es eine Insertion-Heuristik für das TSP, das in der Praxis meist 10 bis 20 Prozent über dem Optimum liegt, aber *keine* konstante Gütegarantie hat.
Fazit: Intuitiv und schnell, aber keine guten Lösungen.

5.3 Verbesserungsheuristiken - Lokale Suche

Wir versuchen das Problem des Konstruktionsverfahren zu minimieren, indem wir eine Ausgangslösung durch kleine Änderungen iterativ verbessern. Dafür haben wir aber wiederum *keine Gütegarantien* - aber akzeptable Lösungen und Laufzeiten.
Lokale Suche funktioniert im Prinzip so, dass Lösungen in einer gewissen Nachbarschaft $N(x)$ gesucht werden - sollten davon welche besser sein, werden wir diese annehmen.

Wichtig sind dabei Lösungsrepräsentation, die Wahl der Nachbarschaftsstruktur, die Schrittfunktion und das Terminierungskriterium:

(i) **Nachbarschaftsstruktur**
Ist eine Funktion $N : S \to 2^S$, die jeder gültigen Lösung $x \in S$ eine Menge von Nachbarn $N(x) \subseteq S$ zuweist.

- Meist implizit durch mögliche Veränderungen (Züge, Moves) definiert.
- Als Nachbarschaftsgraph darstellbar: Knoten entsprechen Lösungen, die indizenten Kanten sind die Nachbarn.
- Wichtig ist es, Größe der Nachbarschaft und den benötigten Suchaufwand abzuwägen.

(ii) **Schrittfunktionen**
Die Wahl kann starken Einfluss auf Performance haben - die Wahl hängt stark vom Problem ab, und keines ist im Allgemeinen besser.

- **Best Improvement:** Durchsuche $N(x)$ vollständig und nimm eine *beste* Nachbarlösung. (längere Laufzeit)
- **Next Improvement:** Durchsuche $N(x)$ in einer bestimmten Reihenfolge, nimm erste Lösung, die besser als x ist.
- **Random Neighbor:** Wähle eine zufällige Lösung aus $N(x)$. (sehr schnell)

(iii) **Lokale und globale Optima**
Ein lokales Maximum in Bezug auf eine Nachbarschaftsstruktur N ist eine Lösung x, für die $f(x) \geq f(x')$ für alle $x' \in N(x)$ gilt. Durch Nachbarschaftsstrukturen können wir bestimmen, welche Lösungen *lokal optimal* sind. Da wir nach globalen Optima suchen, ist das eher suboptimal. Verbessern können wir es etwa durch größere Nachbarschaftsstrukturen, iterierte Lokale Suche oder Kombination diverser lokaler Suchmethoden.

(iv) **Terminierungskriterium**
In der Norm wird die lokale Suche beendet, wenn man nichts mehr verbessern kann - dadurch sehen wir, dass wir ein *lokales Optimum* erreicht haben.
Achtung: Bei zB Random Neighbor kann man das nicht immer gleich erkennen - alternativ terminiert man auch nach einer bestimmten, festgelegten Iterationsanzahl oder Zeit, oder auch wenn man schon eine Lösung hat, die gut genug ist.

5.3.1 Lokale Suche für das Vertex Cover

Definition der Nachbarschaftsstruktur: Wenn wir aus dem derzeitigen Vertex Cover durch Löschen eines einzelnen Knotens wieder ein Vertex Cover erhalten. Dies können wir iterativ wiederholen, wobei der Algorithmus nach $O(|V|)$ Schritten terminiert.
Verbesserung können wir durch *alternative Nachbarschaft* erhalten: Das gibt uns zB die Möglichkeit, zwei Knoten zu löschen und einen hinzuzufügen - das wird aber um einiges aufwändiger, und wir erhalten eine Laufzeit von $O(|V|^3)$.
Wichtig: In beiden Fällen erhalten wir nur lokale Optima, nicht immer globale!

5.3.2 Lokale Suche für das symmetrische TSP

Basiert auf **2-opt**, also dem Austausch zweier Kanten - die Größe der Nachbarschaft liegt in $O(n^2)$ und die Zeit für eine vollständige unabhängige Berechnung des Zielfunktionswerts liegt in $O(n)$, benötigt durch das inkrementelle Verbessern aber nur *konstante* Zeit.

Laufzeitkomplexität: Da eben $|N(x)| = O(n^2)$ und jede Nachbarlösung in konstanter Zeit inkrementell evaluiert werden kann, benötigt eine Iteration $O(n^2)$ Zeit. Im worstcase brauchen wir dann für ein lokales Optimum aber $O(n!)$ Zeit, ist also nicht polynomiell. In der Praxis aber bei großen Instanzen sehr schnell und benötigt nur wenige Iterationen.

Wichtig: Dieses Verfahren kann auch auf r-opt erweitert werden! Es werden dann $r \geq 2$ Kanten durch neue ersetzt. Die Größe einer r-opt Nachbarschaft liegt aber dann in $|N(x)| = O(n^r \cdot r!) = O(n^r)$, das ganze dauert dann auch wesentlich länger - 4-opt ist dann meistens nicht mal mehr praktikabel.

5.3.3 Lokale Suche für MAX-CUT

Problemdefinition: Gegeben sei ein ungerichteter Graph $G = (V, E)$ mit positiven ganzzahligen Kantengewichten w_{uv} für alle Kanten $(u, v) \in E$. Finde eine Partition der Knoten (A, B), sodass das Gesamtgewicht von Kanten, die Knoten in den unterschiedlichen Partitionen verbinden, maximiert wird. Also:

$$w(A, B) := \sum_{u \in A, v \in B} w_{uv}$$

Dieses Problem liegt in NPC!

Zur lokalen Suche verwenden wir eine *Flip-Nachbarschaft* - wenn wir eine Partition $(A, B$ haben, verschieben wir einen Knoten von A nach B oder von B nach A. Dies können wir ausgehend von einer gültigen Initiallösung aufbauend unmittelbar durch eine lokale Suche anwenden. Hierbei ist wichtig, dass eine Approximationsgüte von $\frac{1}{2}$ gilt!

5.4 Metaheuristiken

Dies sind bereits problemunabhängig formulierte Algorithmen zur Lösung schwieriger Optimierungsaufgaben - nur Teile müssen an das jeweilige Problem angepasst werden.

5.4.1 Simulated Annealing

Wichtig: Auch schlechtere Nachbarlösungen werden mit einer bestimmten Wahrscheinlichkeit akzeptiert. Im Allgemeinen wird Random Neighbor als Schrittfunktion gewählt. Außerdem werden nur geringfügig schlechtere Lösungen mit höherer Wahrscheinlichkeit akzeptiert, als viel schlechtere - und das auch nur anfänglich. Die Wahrscheinlichkeit, dass schlechtere Lösungen genommen werden, konvergiert relativ rasch.

Der Algorithmus ist dabei meist einfach zu implementieren, das erforderte Parametertuning ist nicht schwierig und spuckt dafür recht passable Ergebnisse aus - durch ausgefeiltere Methoden kann man natürlich auch bessere Resultate erzielen. Des Weiteren sind Kombination mit anderen Methoden, Parallelisierung und dynamische Strategien für das Abkühlen (Wiedererwärmen) möglich. Der Algorithmus lautet:

- Z: (Pseudo-)Zufallszahl $\in [0, 1)$
- T: „Temperatur“

```
Simulated-Annealing():
t ← 0
T ← T_init
x ← Ausgangslösung
while Abbruchkriterium nicht erfüllt
    Wähle x' ∈ N(x) zufällig
    if x' besser als x
        x ← x'
    elseif Z < e^{-|f(x')-f(x)|/T}
        x ← x'
    T ← g(T, t)
    t ← t + 1
```

Metropolis-Kriterium

5.4.2 Tabu-Suche

Diese basiert auf einem Gedächtnis (History) über dem bisherigen Optimierungsverlauf, um so über lokale Optima hinwegzukommen - dabei vermeiden wir Zyklen durch Verbieten des Wiederbesuchens früherer Lösungen. Hier verwenden wir als Schrittfunktion meistens *Best Improvement* - wir nehmen also immer die beste erlaubte Nachbarlösung, auch wenn sie schlechter ist, als die aktuelle Lösung.
Durch das explizite Speichern von Lösungen wird das Ganze leider schnell speicher- und zeitaufwändig, weshalb man eher (verbotene) Tabuattribute (einzelne Aspekte besuchter Lösungen) betrachtet.

Wichtig: Als Parameter muss dann die Tabulistenlänge gewählt werden - was relativ schwierig ist. Sind sie zu kurz, geraten wir schnell in Zyklen; sind sie zu lang, verbieten wir viele mögliche Lösungen und beschränken die Suche zu stark. Die geeignete Länge ist hierbei problemspezifisch und wird meist experimentell bestimmt oder währenddessen adaptiv angepasst (*Reactive Tabu Search*). Dabei ist auch das Problem, dass wir oft sogar sehr gute Lösungen verbieten - hier verschafft das *Aspirationskriterium* Abhilfe: Dieses überschreibt den Tabu-Status einer potentiell guten Lösung, wodurch man sie wieder wählen darf. Das geschieht etwa, wenn wir eine Lösung verbieten, die besser ist, als jede bisher erhaltene Lösung.

Im Allgemeinen ist Tabu-Suche relativ schnell und liefert exzellente Ergebnisse - dafür ist aber oft das Fine-Tuning ziemlich aufwändig.

Der Algorithmus sieht folgendermaßen aus:

```
Tabu-Suche():
x_best ← x ← Ausgangslösung
TL ← {x}
while Abbruchkriterium nicht erfüllt
    X' ← Teilmenge von N(x) unter Berücksichtigung von TL
    x' ← beste Lösung von X'
    Füge x' zu TL hinzu
    Lösche Elemente aus TL, welche älter als t_L Iterationen sind
    x ← x'
    if x besser als x_best
        x_best ← x
```

5.4.3 Evolutionäre Algorithmen

Im Prinzip wird die natürliche Evolution nachgeahmt, nur mit Lösungen:

- **Population:** Es wird eine Menge von aktuellen Kandidatenlösungen bearbeitet.
- **Selektion:** Natürliche Auslese, bessere Lösungen überleben und zeugen neue Lösungen, schlechte Lösungen werden aus dem Genpool genommen.
- **Rekombination:** Neue Lösungen werden durch zufallsgesteuerte Kreuzung und Vererbung von in Eltern vorkommenden Lösungsmerkmalen abgeleitet.
- **Mutation:** Kleine zufällige Änderung bringt nicht in Eltern vorkommende Lösungsmerkmale ein und dadurch ist eine Variation von Elternlösungen möglich. (funktioniert zB durch zufällige Bitflips oder Moves in $N(x)$)

Aufpassen muss man bei der Parameterwahl des Selektionsdrucks - ist er zu niedrig, erhalten wir mehr oder weniger eine Zufallssuche. Ist er zu hoch, erhalten wir eine rasche Verlust an Vielfalt, und das Verfahren konvergiert schnell zu einem lokalen Optimum. Damit wir den Selektionsdruck steuern können, skalieren wir die Bewertungsfunktion, meist über eine lineare Funktion.

Das Grundprinzip ist hierbei meist leicht umsetzbar und Parallelisierung ist gut möglich, die Lösungsgüte und Laufzeit hängen aber stark von den konkreten Operatoren ab.

- Selektierte Eltern Q_s
- Zwischenlösungen Q_r

```
Evolutionär():
P ← Menge von Ausgangslösungen
Bewerte(P)
while Abbruchkriterium nicht erfüllt
    Q_s ← Selektion(P)
    Q_r ← Rekombination(Q_s)
    P ← Mutation(Q_r)
    Bewerte(P)
```