2. Test Zusammenfassung

Vorwort

Diese Zusammenfassungen wurden basierend auf den Vorlesungsfolien erstellt. Sie dienen als stichwortartige Gedächtnisstütze und orientieren sich stark an den Inhalten der Folien.

Meiner Meinung nach sind die Slides bereits sehr gut aufbereitet, um den Stoff zu lernen, und diese Zusammenfassung dient primär der schnellen Wiederholung und dem Überblick.

> ⚠ **Wichtige Info noch vorab zum Teststoff**
>
> Teststoff ist prinzipiell der komplette Inhalt der LVA, allerdings liegt der Fokus auf den Stoff der in der 2. Hälfte der LVA abgehalten wurde.
>
> Dieses Dokument ist lediglich eine Zusammenfassung vom 2. Teil der LVA

> ⓘ **Inhalt**

9. Polynomialzeitreduktion

Effizient Lösbare Probleme und NP-Klassen

Einleitung: Effizienz und Cobham-Edmonds-Annahme

Ein Problem ist effizient lösbar (oder handhabbar), wenn seine Laufzeit durch ein Polynom der Eingabegröße $O(n^c)$ beschränkt ist. Die Cobham-Edmonds-Annahme postuliert, dass Handhabbarkeit gleichbedeutend mit Polynomialzeit-Lösbarkeit ist. Diese Annahme hat die Informatik geprägt, da polynomielle Algorithmen oft praktikabel sind und auf strukturelle Einsichten in Problemen hinweisen. Ausnahmen sind polynomielle Algorithmen mit hohen Exponenten oder exponentielle Algorithmen, die in der Praxis für kleine Instanzen nützlich sein können.

Probleme klassifizieren: P oder nicht P?

Ziel ist die Unterscheidung zwischen polynomiell lösbaren Problemen (P) und solchen, die es nicht sind. Beispiele für Probleme, die nachweislich mehr als polynomielle Zeit erfordern, sind das Halteproblem mit Schranke und verallgemeinertes Schachgewinn. Für viele fundamentale Probleme wie Maximum Independent Set, 3-Färbbarkeit und SAT ist die Klassifizierung (P oder NP?) noch unbekannt, was ein zentrales ungelöstes Problem der theoretischen Informatik darstellt.

Ja/Nein-Probleme

Zur Vereinfachung konzentriert man sich auf Ja/Nein-Probleme (decision problems), deren Antwort entweder Ja oder Nein ist. Dies grenzt sie von funktionalen Problemen (die eine Lösung liefern) und Optimierungsproblemen (die eine optimale Lösung suchen) ab.

Beispiel:

- Ja/Nein: Gibt es einen Spannbaum mit Kosten $\leq k$?
- Funktional: Finde einen Spannbaum mit Kosten $\leq k$.
- Optimierung: Finde einen Spannbaum mit minimalen Kosten.

Polynomialzeitreduktionen ($X \leq_P Y$)

Eine Polynomialzeitreduktion von Problem X auf Problem Y bedeutet, dass man X effizient lösen kann, wenn man einen effizienten Algorithmus für Y hat. Dabei wird eine Instanz von X in Polynomialzeit in eine äquivalente Instanz von Y umgewandelt.

Definition: Ein Algorithmus R ist eine Polynomialzeitreduktion von X auf Y ($X \leq_P Y$), wenn:

1. x ist eine Ja-Instanz von X genau dann, wenn $y = R(x)$ eine Ja-Instanz von Y ist.
2. Die Laufzeit von R ist polynomiell in der Größe von x.

Wenn $X \leq_P Y$ und Y in Polynomialzeit lösbar ist, dann ist auch X in Polynomialzeit lösbar. Umgekehrt, wenn $X \leq_P Y$ und X *nicht* in Polynomialzeit lösbar ist, dann ist auch Y *nicht* in Polynomialzeit lösbar.

Polynomialzeit-Äquivalenz ($X \equiv_P Y$): Gilt $X \leq_P Y$ und $Y \leq_P X$, sind X und Y bezüglich ihrer Schwierigkeit äquivalent.

Transitivität: Ist $X \leq_P Y$ und $Y \leq_P Z$, dann folgt $X \leq_P Z$.

Nachweis einer PZR: Erfordert den Nachweis von Korrektheit (Ja-Instanzen werden auf Ja, Nein auf Nein abgebildet) und Polynomialität des Reduktionsalgorithmus.

Independent Set und Vertex Cover

Independent Set

- Definition: Eine Teilmenge $S \subseteq V$ von Knoten eines Graphen $G = (V, E)$, in der es keine adjazenten Knoten gibt.
- Problem INDEPENDENT SET: Gegeben G, k. Gibt es ein Independent Set S mit $|S| \geq k$?

Vertex Cover

- Definition: Eine Menge $S \subseteq V$ von Knoten eines Graphen $G = (V, E)$, sodass jede Kante des Graphen zu mindestens einem Knoten in S inzidiert ist.
- Problem VERTEX COVER: Gegeben G, k. Gibt es ein Vertex Cover S mit $|S| \leq k$?

Konversionslemma und Äquivalenz

Konversionslemma: Für einen ungerichteten Graphen $G = (V, E)$ ist $S \subseteq V$ ein Independent Set genau dann, wenn $C = V - S$ ein Vertex Cover von G ist.

Äquivalenz: $VERTEX\ COVER \equiv_P INDEPENDENT\ SET$.
Beweis: Eine Instanz (G, k) von VERTEX COVER kann in $(G, n - k)$ von INDEPENDENT SET umgewandelt werden und umgekehrt, jeweils in Polynomialzeit, basierend auf dem Konversionslemma.

Beispiel: Spannbäume und Nicht-Blockierer

Nicht-Blockierer

- Definition: Eine Kantenmenge $N \subseteq E$ eines gewichteten Graphen $G = (V, E)$, sodass für alle Knotenpaare $u, v \in V$ ein $u - v$-Pfad in G existiert, der keine Kante aus N enthält.
- Problem MNB (Maximaler Nicht-Blockierer): Gegeben G, k. Besitzt G einen Nicht-Blockierer mit Kosten $\geq k$?

Spannbaum

- Definition: Eine Kantenmenge $T \subseteq E$, sodass $G_T = (V, T)$ ein Baum ist (zusammenhängend und zyklenfrei).

- **Problem MST* (Minimaler Spannbaum mit Kosten):** Gegeben G, k. Besitzt G einen Spannbaum mit Kosten $\leq k$? (*MST ist polynomiell lösbar*).

Konversionslemma und Äquivalenz

Konversionslemma: Ein Nicht-Blockierer N ist maximal genau dann, wenn $T = E - N$ ein minimaler Spannbaum ist. Die Kostenbeziehung ist $\text{Kosten}(N) = \sum_{e \in E} c_e - \text{Kosten}(T)$.

Äquivalenz: $MNB \equiv_P MST^*$.
Dies zeigt, dass MNB in Polynomialzeit lösbar ist, da es auf das polynomiell lösbare MST* reduziert werden kann.

Set Cover (Mengenüberdeckungsproblem)

- **Definition:** Gegeben ein Universum U von Elementen und eine Menge $S = \{S_1, \ldots, S_m\}$ von Teilmengen von U. Ein **Set Cover** $C \subseteq S$ ist eine Auswahl von Mengen, deren Vereinigung das gesamte Universum U abdeckt ($\bigcup_{X \in C} X = U$).
- **Problem SET COVER:** Gegeben U, S, k. Existiert eine Teilmenge $C \subseteq S$ mit $|C| \leq k$, die ein Set Cover bildet?

Anwendungen: Auswahl von Softwarekomponenten, die alle benötigten Eigenschaften abdecken; Minimierung von Pizzen, um alle gewünschten Zutaten zu erhalten.

Vertex Cover auf Set Cover reduzieren

Man kann eine Instanz von VERTEX COVER in eine Instanz von SET COVER überführen:

- Universum U = Kantenmenge E des Graphen.
- Für jeden Knoten $v \in V$ wird eine Menge S_v gebildet, die alle Kanten enthält, die zu v inzidieren.
- Das gesuchte k bleibt gleich.
  Eine Vertex Cover-Lösung im Originalgraphen entspricht einem Set Cover der Kanten. Dies ist eine Reduktion eines Spezialfalls (Vertex Cover) auf den allgemeinen Fall (Set Cover).

Reduktion mit Gadgets

Gadgets sind kleine, wiederkehrende Bausteine, die bei Reduktionen verwendet werden, um eine Instanz eines Problems in eine Instanz eines anderen Problems zu "implementieren".

Erfüllbarkeitsproblem (SAT)

- **Literal:** Variable (x_i) oder ihre Negation ($\neg x_i$).
- **Klausel:** Disjunktion von Literalen (z.B. $x_1 \vee \neg x_2 \vee x_3$).
- **Konjunktive Normalform (KNF):** Konjunktion von Klauseln (z.B. $(x_1 \vee x_2) \wedge (\neg x_1 \vee x_3)$).
- **Wahrheitsbelegung:** Zuordnung von `true` / `false` zu Variablen.
- **SAT Problem:** Gibt es eine Wahrheitsbelegung, die eine gegebene KNF-Formel Φ erfüllt?
- **3-SAT:** SAT, bei dem jede Klausel genau 3 Literale enthält.

Bedeutung von SAT: Wichtig für leistungsstarke heuristische Solver und als Problem, das oft für den Nachweis der Nicht-Handhabbarkeit anderer Probleme dient (NP-Vollständigkeit).

3-SAT auf Independent Set reduzieren

Eine Instanz von 3-SAT kann in eine Instanz von INDEPENDENT SET reduziert werden. Für jede Klausel der 3-SAT-Formel wird ein Dreieck im Graphen erstellt, dessen Knoten die Literale der Klausel repräsentieren. Zusätzlich werden Kanten zwischen einem Literal und seiner Negation in verschiedenen Dreiecken gezogen. Das gesuchte k für das Independent Set entspricht der Anzahl der Klauseln. Ein Independent Set der Größe k im konstruierten Graphen existiert genau dann, wenn die 3-SAT-Formel erfüllbar ist.

Optimierungsprobleme

Optimierungsprobleme (z.B. Finde ein *kleinstes* Vertex Cover) können oft durch mehrfaches Aufrufen des entsprechenden Ja/Nein-Problems (z.B. Existiert ein Vertex Cover der Größe $\leq k$?) gelöst werden.

Algorithmus OptVC(G) zum Finden eines kleinsten Vertex Covers:
Dieser Algorithmus nutzt die Fähigkeit, das Ja/Nein-Problem $VC(G, k)$ zu lösen. Durch iterative Reduzierung der Problemgröße und Abfragen des Ja/Nein-Problems kann das optimale Vertex Cover gefunden werden. Die Gesamtkomplexität ist $O(n^2)$ mal die Komplexität des Ja/Nein-Problems. Dies rechtfertigt den Fokus auf Ja/Nein-Probleme in der Komplexitätstheorie.

Definition von NP

NP-Probleme

- Ein Ja/Nein-Problem ist ein **NP-Problem**, wenn Ja-Instanzen mit Hilfe eines **Zertifikats** effizient (in Polynomialzeit) überprüft werden können.
- **Zertifikat (t):** Beliebiger Input, dessen Größe m polynomiell in der Größe n der Instanz x sein muss ($m \leq p(n)$).
- **Zertifizierer ($C(x, t)$):** Ein Polynomialzeitalgorithmus, der überprüft, ob t ein gültiger "Beweis" dafür ist, dass x eine Ja-Instanz ist.

Wichtige Eigenschaften des Zertifizierers:

- Für jede Ja-Instanz x gibt es ein Zertifikat t, das $C(x, t)$ akzeptiert.
- Für keine Nein-Instanz x gibt es ein Zertifikat t, das $C(x, t)$ akzeptiert.

Anmerkung: "NP" steht für "nicht-deterministisch polynomielle" Zeit, was äquivalent zur hier gegebenen Zertifikatsdefinition ist. Das Finden des Zertifikats kann schwer sein, aber das Prüfen muss effizient sein.

10. NP-Vollständigkeit Spezialfälle

Spezialfälle von NP-vollständigen Problemen

Finden von kleinen Vertex Covers

Das Vertex Cover Problem (VC) fragt, ob es in einem Graphen $G = (V, E)$ eine Knotenmenge $S \subseteq V$ der Größe $|S| \leq k$ gibt, die jede Kante überdeckt (d.h., für jede Kante $(u, v) \in E$ ist $u \in S$ oder $v \in S$). Das Ziel ist es, ein möglichst kleines VC zu finden (wenn $k \ll n$).

Brute-Force Ansatz

Ein naiver Brute-Force-Ansatz prüft alle $\binom{n}{k}$ möglichen Teilmengen der Größe k. Das sind $O(n^k)$ Mengen, und jede Prüfung dauert $O(k|E|)$ oder $O(kn)$ Zeit. Die Gesamtlaufzeit ist $O(kn^{k+1})$.
Für $n = 1000, k = 10$ ist dies 10^{34} Operationen, was undurchführbar ist.

Rekursiver Algorithmus für kleine k

Das Ziel ist, die exponentielle Abhängigkeit von n zu eliminieren und nur eine Abhängigkeit von k beizubehalten, also eine Laufzeit der Form $O(c^k \cdot \text{poly}(n))$.

Behauptung: Wenn G ein Vertex Cover der Größe $\leq k$ hat, und (u, v) eine Kante in G ist, dann hat entweder $G - \{u\}$ oder $G - \{v\}$ ein Vertex Cover der Größe $\leq k - 1$.
Dies führt zu einem rekursiven Algorithmus:

1. Wenn $k = 0$: Prüfe, ob der Graph kantenfrei ist. Wenn ja, ist das VC leer, sonst nicht möglich.
2. Wenn $|E| = 0$: Das VC ist die leere Menge.
3. Wenn G keine Kanten hat ($|E| = 0$), ist ein VC der Größe 0.
4. Wähle eine beliebige Kante $(u, v) \in E$.
5. Rekursiver Aufruf für $G - \{u\}$ mit $k - 1$ (füge u zum VC hinzu) und für $G - \{v\}$ mit $k - 1$ (füge v zum VC hinzu).
   Die Anzahl der Knoten in G ist n.

Dieser Algorithmus hat eine Laufzeit von $O(2^k(n + m))$, da der Rekursionsbaum eine Tiefe von k und an jeder Ebene eine Verzweigung von 2 hat. Jeder Knoten im Rekursionsbaum erfordert $O(n + m)$ Operationen.
Fazit: Wenn k eine Konstante ist, ist der Algorithmus polynomiell. Wenn k eine kleine Konstante ist, ist er auch praktikabel.

Lösen NP-vollständiger Probleme auf Bäumen

Independent Set auf Bäumen

Ein Independent Set (unabhängige Menge) in einem ungerichteten Graphen $G = (V, E)$ ist eine Knotenmenge $I \subseteq V$, sodass keine zwei Knoten in I durch eine Kante verbunden sind. Für allgemeine Graphen ist das Problem, ein maximales Independent Set zu finden, NP-vollständig. Für Bäume ist es jedoch effizient lösbar.

Beobachtung: Jeder Baum mit mindestens zwei Knoten hat mindestens zwei Blätter (Knoten vom Grad 1). Wenn v ein Blatt ist, existiert immer ein maximales Independent Set, das v enthält.

Gieriger Algorithmus für Independent Set auf Bäumen:

1. Solange der Graph nicht leer ist:
   - Wähle ein Blatt v (beliebiger Knoten vom Grad 1).
   - Füge v zu deinem Independent Set S hinzu.
   - Entferne v und seinen einzigen Nachbarn u (sowie alle inzidenten Kanten) aus dem Graphen.
2. Füge alle verbleibenden isolierten Knoten (ohne Kanten) zu S hinzu.
   Dieser Algorithmus findet ein maximales Independent Set in Bäumen.

Gewichtetes Independent Set auf Bäumen

Wenn Knoten Gewichte $w_v > 0$ haben, kann man das Problem mit **Dynamischer Programmierung (DP)** lösen.
Man wählt einen beliebigen Knoten als Wurzel des Baumes. Die Lösung wird rekursiv (typischerweise in Postorder) von den Blättern zur Wurzel berechnet.

DP-Zustände für einen Knoten u:

- $OPT_{in}(u)$: Maximales Gewicht eines Independent Sets im Unterbaum von u, wenn u *im* Independent Set enthalten ist.
  - $OPT_{in}(u) = w_u + \sum_{v \text{ Kind von } u} OPT_{out}(v)$
- $OPT_{out}(u)$: Maximales Gewicht eines Independent Sets im Unterbaum von u, wenn u *nicht im* Independent Set enthalten ist.
  - $OPT_{out}(u) = \sum_{v \text{ Kind von } u} \max(OPT_{in}(v), OPT_{out}(v))$

Die Laufzeit dieses Algorithmus ist $O(n)$, da jeder Knoten und jede Kante genau einmal betrachtet wird. Dies zeigt, dass Probleme, die für allgemeine Graphen schwer sind, auf Graphen mit spezieller Struktur (wie Bäumen oder Graphen mit beschränkter Baumweite) effizient lösbar sein können.

Knotenfärben in Intervallgraphen

Das **k-Färbungsproblem** fragt, ob die Knoten eines ungerichteten Graphen G mit k Farben gefärbt werden können, sodass benachbarte Knoten unterschiedliche Farben haben. Für $k \geq 3$ ist dieses Problem NP-vollständig. Das **Optimierungsproblem Opt-COLOR** sucht die minimale Anzahl von Farben.

Intervallgraphen

Intervallgraphen sind Graphen, die als Schnittgraph von Intervallen auf der reellen Zahlengeraden dargestellt werden können. Eine Kante (u, v) existiert genau dann, wenn sich die Intervalle I_u und I_v überschneiden.

Beobachtung: Wenn ein Punkt $x \in \mathbb{R}$ in d verschiedenen Intervallen gleichzeitig liegt, dann bilden die entsprechenden d Knoten eine Clique und benötigen mindestens d Farben. Die **Tiefe d** eines Intervallgraphen ist die maximale Anzahl von Intervallen, die sich an einem einzelnen Punkt überschneiden. Ein Intervallgraph benötigt mindestens d Farben.

Theorem: Ein Intervallgraph kann immer mit genau d Farben gefärbt werden.

Algorithmus "Interval-Coloring":

1. Sortiere alle Start- und Endpunkte der Intervalle.
2. Gehe die sortierten Punkte von links nach rechts durch:
   Bei einem Startpunkt eines Intervalls: Weist diesem Intervall die kleinste verfügbare Farbe zu. Wenn keine frei ist, verwende eine neue Farbe.
   Bei einem Endpunkt eines Intervalls: Gib die Farbe dieses Intervalls wieder frei.
   Dieser Algorithmus gewährleistet, dass überlappende Intervalle unterschiedliche Farben erhalten.

Laufzeit: Der Algorithmus hat eine Laufzeit von $O(n \log n)$, hauptsächlich bedingt durch die Sortierung der $2n$ Intervallgrenzen.

11. Branch and Bound

Einleitung: Kombinatorische Optimierung

Kombinatorische Optimierung befasst sich mit der Konstruktion einer optimalen Teilmenge diskreter Elemente, die bestimmte Nebenbedingungen erfüllt und eine gegebene Kostenfunktion optimiert (z.B. minimales Gewicht, kürzeste Strecke).

Viele dieser Probleme sind schwer, d.h., es ist unwahrscheinlich, dass polynomielle Lösungsverfahren für alle Instanzen existieren (NP-vollständig). Für solche Probleme werden Verfahren eingesetzt, die oft eine optimale Lösung finden, aber keine Garantie für polynomielle Laufzeit geben. Dazu gehören:

- Branch-and-Bound
- Dynamische Programmierung
- Approximationsalgorithmen
- Heuristische Verfahren

Branch-and-Bound

Branch-and-Bound ist eine Technik zur systematischen Durchmusterung aller möglichen Lösungen eines Optimierungsproblems, die auf dem Divide-and-Conquer-Prinzip basiert. Sie beschränkt den Suchraum durch den Einsatz von unteren und oberen Schranken, um eine optimale Lösung zu ermitteln. Der Trade-off hierbei ist der Verzicht auf eine Garantie der Polynomialzeit.

Rucksackproblem als Beispiel

Das Rucksackproblem ist ein klassisches kombinatorisches Optimierungsproblem: Man möchte eine Teilmenge von Gegenständen mit maximalem Gesamtwert auswählen, ohne eine gegebene Gewichtsgrenze G zu überschreiten.

Mathematische Formulierung (0/1-Rucksackproblem):

- Entscheidungsvariablen: $x_i \in \{0, 1\}$ für jeden Gegenstand i (1: gewählt, 0: nicht gewählt).
- Maximiere Zielfunktion: $\sum_{i=1}^{n} w_i x_i$ (w_i: Wert von Gegenstand i).
- Nebenbedingung (Gewicht): $\sum_{i=1}^{n} g_i x_i \leq G$ (g_i: Gewicht von Gegenstand i).

Enumeration (Backtracking)

Die vollständige Enumeration (Backtracking) aller 2^n möglichen Teilmengen im Rucksackproblem hat eine Laufzeit von $O(2^n)$. Dabei wird rekursiv jeder Gegenstand entweder gewählt oder nicht gewählt, bis alle Gegenstände betrachtet wurden und die aktuelle Lösung bewertet wird. Die beste bisher gefundene Lösung wird global gespeichert.

Verbesserung durch Branch-and-Bound

Die Kernidee von Branch-and-Bound ist das frühzeitige Abbrechen von rekursiven Aufrufen, wenn offensichtlich ist, dass der aktuelle Zweig des Suchbaums keine bessere Lösung als die bisher beste globale Lösung liefern kann. Dies führt zu einer Beschneidung (Pruning) des Aufrufbaums und kann die praktische Laufzeit erheblich beschleunigen.

Ansatz:

1. Sortiere die Gegenstände nach nicht-steigenden Wert-Gewichts-Quotienten ($\frac{w_i}{g_i}$).
2. Obere Schranke (U'): Für ein Teilproblem (mit bereits fixierten Gegenständen und verbleibender Kapazität $G - g_{curr}$) wird eine obere Schranke für den erreichbaren Wert berechnet. Dies geschieht, indem die verbleibende Kapazität mit dem aktuell betrachteten Gegenstand i und/oder nachfolgenden Gegenständen (mithilfe von Teilmengen/Bruchteilen) aufgefüllt wird. Da die Gegenstände nach Wert-Gewichts-Quotient sortiert sind, ist dies eine optimistische (aber gültige) obere Schranke. $U' \leftarrow w_{curr} + (G - g_{curr}) \cdot \frac{w_i}{g_i}$
3. Abbruchbedingung (Pruning): Wenn die berechnete obere Schranke U' für ein Teilproblem nicht größer ist als der Wert der bisher besten gefundenen globalen Lösung (w_{max}), wird dieser Zweig nicht weiter verfolgt.

Prinzip von Branch-and-Bound

- Branching: Das Problem wird rekursiv in kleinere Teilprobleme zerlegt, wodurch der Lösungsraum partitioniert wird (Divide-and-Conquer).
- Bounding: Für jedes Teilproblem werden eine lokale obere Schranke (U' - Best Case) und eine lokale untere Schranke (L' - Worst Case) berechnet. Der Wert jeder gültigen Lösung ist eine untere Schranke.
- Abbruch (Pruning): Teilprobleme, deren obere Schranke U' kleiner oder gleich einer globalen unteren Schranke L (dem Wert der besten bisher gefundenen Lösung) ist, müssen nicht weiter verfolgt werden.
- Globale Schranken: L ist der Wert der besten bisher gefundenen *vollständigen* zulässigen Lösung.

Rucksackproblem: Verbesserte Schranken

- Verbesserte untere Schranke: Kann durch einen gierigen Ansatz bestimmt werden, indem man die verbleibenden, noch nicht fixierten Gegenstände in sortierter Reihenfolge (nach $\frac{w_i}{g_i}$) in den Rucksack packt, solange Platz ist.
- Verbesserte obere Schranke: Ergänzt den aktuellen Wert der fixierten Gegenstände um den maximal möglichen Wert, der durch "partielles" Einpacken des ersten Gegenstands, der nicht mehr ganz passt, erreicht werden könnte. Diese obere Schranke ist nicht unbedingt eine zulässige Rucksacklösung, aber gültig für das Bounding.

Allgemeines Vorgehen und Algorithmus

Das Problem wird durch Fixieren von Variablen oder Hinzufügen von Randbedingungen in Unterprobleme zerteilt.

- Wenn $U' \leq L$: Pruning (keine bessere Lösung im Teilraum).
- Wenn $U' > L$: Branching (Teilraum weiter aufteilen).
- Fortsetzung, bis alle Teilmengen entweder beschnitten oder vollständig gelöst sind.

Branch-and-Bound ist ein allgemeines Metaverfahren, dessen Effizienz stark von der Wahl der Heuristiken für Schranken, der Branching-Strategie und der Auswahl des nächsten Teilproblems abhängt.

Auswahl des nächsten Teilproblems:

- **Best-first:** Wählt das Teilproblem mit der besten (höchsten bei Maximierung) oberen Schranke. Ziel ist die minimale Anzahl bearbeiteter Teilprobleme bis zur Optimalität.
- **Depth-first:** Bearbeitet das zuletzt erzeugte Teilproblem (analog zur Tiefensuche). Liefert oft schnell eine erste gültige Näherungslösung, die als initiale globale untere Schranke dienen kann. Eine Kombination beider Strategien ist oft sinnvoll.

Branch-and-Bound für Minimales Vertex Cover

Für **Minimierungsprobleme** wie das Minimale Vertex Cover Problem wird der Algorithmus analog angepasst:

- Die Zielfunktion wird minimiert.
- Die Abbruchbedingung ist $U' \geq L$ (d.h., wenn die lokale untere Schranke größer oder gleich der globalen oberen Schranke ist).

Minimales Vertex Cover: Das Ziel ist, ein Vertex Cover mit der kleinstmöglichen Knotenzahl zu finden.

Untere Schranke (L'):

- Eine untere Schranke für die Größe eines Vertex Covers ist die Größe eines **maximalen Matchings** (ein Matching, das nicht durch Hinzufügen weiterer Kanten vergrößert werden kann). Jede Kante in einem Matching muss von mindestens einem Knoten im VC überdeckt werden.

Obere Schranke (U'):

- Eine obere Schranke kann mit einem **gierigen Algorithmus** bestimmt werden: Wähle iterativ den Knoten mit dem höchsten Grad, füge ihn zum VC hinzu und entferne ihn samt seinen Kanten. Wiederhole, bis keine Kanten mehr vorhanden sind. Die Summe der Knoten in C' (bereits fixierte Knoten) und der durch den gierigen Algorithmus gefundenen Knoten ist eine obere Schranke.

Branch-and-Bound: Zusammenfassung

- Branch-and-Bound ist eine vielseitige Technik zur Berechnung exakter Lösungen für kombinatorische Optimierungsprobleme.
- Anwendbar für Maximierungs- und Minimierungsprobleme.
- Praktisch kann es zu erheblichen Beschleunigungen führen, aber die Worst-Case-Laufzeit bleibt oft exponentiell.

Vorgehen beim Entwurf:

- Definition der Teil-Instanzen.
- Entwicklung guter Heuristiken für untere und obere Schranken.
- Festlegung der Branching-Strategie (wie die Instanz zerteilt wird).
- Auswahlregel für die nächste zu bearbeitende Teilinstanz.

12. Dynamische Programmierung

Einleitung

Dynamische Programmierung (DP) ist eine mächtige Technik zur Lösung von Optimierungsproblemen. Sie zerlegt ein Problem in eine Folge von überlappenden Teilproblemen, speichert deren Lösungen (Memoization) und nutzt diese, um sukzessiv Lösungen für größere Probleme zu konstruieren. Das Optimalitätsprinzip von Bellman besagt, dass DP zu einem optimalen Ergebnis führt, wenn die optimale Lösung des Gesamtproblems aus den optimalen Lösungen seiner Subprobleme zusammengesetzt werden kann. DP ist eng verwandt mit Divide and Conquer, unterscheidet sich aber durch die explizite Speicherung und Wiederverwendung von Teillösungen, um redundante Berechnungen zu vermeiden.

Beispiel: Weighted Independent Set auf Bäumen

Ein früheres Beispiel für DP ist das Finden eines Weighted Independent Set auf Bäumen mit einer Laufzeit von $O(n)$. Für jeden Knoten u werden zwei Werte berechnet:

- $M_{in}(u)$: Maximales Gewicht des Independent Set im Unterbaum von u, wenn u *enthalten* ist (Kinder müssen ausgeschlossen werden).
- $M_{out}(u)$: Maximales Gewicht des Independent Set im Unterbaum von u, wenn u *nicht enthalten* ist (Kinder können entweder enthalten oder nicht enthalten sein).

Die Rekursionsbeziehungen sind:

- $M_{in}(u) = w_u + \sum_{v \text{ Kind von } u} M_{out}(v)$
- $M_{out}(u) = \sum_{v \text{ Kind von } u} \max(M_{in}(v), M_{out}(v))$

Blattknoten bilden die Basis: $M_{in}(Blatt) = w_{Blatt}$, $M_{out}(Blatt) = 0$.

Einführendes Beispiel: Fibonacci-Zahlen

Die Fibonacci-Zahlen $F_n = F_{n-1} + F_{n-2}$ mit $F_1 = F_2 = 1$ sind ein klassisches Beispiel für die Vorteile der DP.

Naive Rekursive Laufzeit

Ein einfacher rekursiver Algorithmus `Fibonacci(n)` hat eine exponentielle Zeitkomplexität von $O(\Phi^n)$ (wobei $\Phi \approx 1.618$ der Goldene Schnitt ist), da viele Teilprobleme redundant berechnet werden (z.B. `Fibonacci(3)` mehrfach).

Dynamische Programmierung mit Memoization (Top-down)

Durch Memoization (Speicherung der Ergebnisse in einem Array F) wird die Laufzeit auf $O(n)$ reduziert. Bevor eine Fibonacci-Zahl berechnet wird, prüft der Algorithmus, ob sie bereits im Array gespeichert ist. Falls ja, wird der Wert direkt zurückgegeben. Andernfalls wird er berechnet und gespeichert.

```
Initialisiere ein Array F der Größe n+1 mit leeren Werten.
Fibonacci(n):
    if F[n] ist leer:
        if n == 1 oder n == 2:
            F[n] <- 1
```

```
        else:
            F[n] <- Fibonacci(n - 1) + Fibonacci(n - 2)
    return F[n]
```

Dynamische Programmierung (Iterativ / Bottom-up)

Eine iterative Lösung ist oft noch effizienter und übersichtlicher. Hier werden die Fibonacci-Zahlen von den Basisfällen aufwärts berechnet und in einem Array gespeichert. Die Laufzeit ist ebenfalls $O(n)$.

```
Linear-Fibonacci(n):
    F sei ein Array der Größe n+1.
    F[1] <- 1
    F[2] <- 1
    for i <- 3 bis n:
        F[i] <- F[i - 1] + F[i - 2]
    return F[n]
```

Gewichtetes Intervall Scheduling

Problemstellung: Finde eine Teilmenge von paarweise kompatiblen Jobs mit maximalem Gesamtgewicht. Jobs sind durch Startzeit s_j, Endzeit f_j und Gewicht w_j gegeben. Jobs sind kompatibel, wenn sie sich nicht überlappen.

Ein einfacher Greedy-Algorithmus (sortiert nach Endzeit) funktioniert nicht, wenn Gewichte berücksichtigt werden.

DP-Ansatz:

1. Sortiere Jobs aufsteigend nach Endzeit: $f_1 \leq f_2 \leq \cdots \leq f_n$.
2. Definiere $p(j)$ als den größten Index $i < j$, sodass Job i kompatibel zu Job j ist ($f_i \leq s_j$). Falls keiner existiert, $p(j) = 0$.
3. Definition des Teilproblems: $OPT(j)$ = Wert der optimalen Lösung für die Jobs $1, \ldots, j$.
4. Rekursive Definition:

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

   - `w_j + OPT(p(j))`: Job j wird gewählt (dann darf kein Job zwischen $p(j)$ und j gewählt werden).
   - `OPT(j - 1)`: Job j wird nicht gewählt.

Laufzeit

- Vorbereitung: Sortieren der Jobs ($O(n \log n)$) und Berechnen aller $p(j)$ mittels binärer Suche ($O(n \log n)$).
- Berechnung von `Compute-Opt(j)` (Memoization oder Iterativ): Jeder Eintrag in $M[j]$ wird maximal einmal berechnet, und jede Berechnung dauert $O(1)$. Insgesamt $O(n)$ für die DP-Tabelle.
- Gesamtlaufzeit: $O(n \log n)$.

Finden der Lösung

Die Menge der ausgewählten Jobs kann durch Backtracking des gefüllten M-Arrays rekonstruiert werden. Man prüft für $j = n$ abwärts: Wenn $w_j + M[p(j)] > M[j-1]$ war, wurde Job j gewählt, und man

fährt mit $p(j)$ fort; sonst fährt man mit $j-1$ fort. Laufzeit: $O(n)$.

Segmented Least Squares

Problemstellung: Gegeben n Punkte $(x_1, y_1), \ldots, (x_n, y_n)$ mit $x_1 < \ldots < x_n$. Finde eine Folge von Geradensegmenten, die diese Punkte approximiert und die Kostenfunktion $E + cL$ minimiert, wobei E die Summe der quadrierten Fehler und L die Anzahl der Segmente ist ($c > 0$ ist eine Konstante).

- Die beste einzelne Gerade für k Punkte kann in $O(k)$ Zeit gefunden werden.

DP-Ansatz:

1. **Definition des Teilproblems:** $OPT(j)$ = minimale Kosten für die Punkte $p_1, \ldots, p_j$.
2. $e(i, j)$: Minimale Summe des quadrierten Fehlers für die Punkte $p_i, \ldots, p_j$, wenn sie durch eine einzige Gerade approximiert werden. Dies kann in $O(j - i + 1)$ Zeit berechnet werden.
3. **Rekursive Definition:**

$$OPT(j) = \{0 \quad \text{wenn } j = 0 \ \min_{1 \le i \le j} e(i, j) + c + OPT(i-1) \quad \text{sonst}$$

   Man iteriert über alle möglichen Startpunkte i des letzten Segments, das bei Punkt j endet.

Laufzeit

- Die Berechnung von $e(i, j)$ für alle Paare (i, j) benötigt $O(n^3)$ Zeit ($O(n^2)$ Paare, jeweils $O(n)$).
- Die DP-Tabelle $OPT[j]$ wird für $j = 1 \ldots n$ berechnet. Für jeden j-Wert wird über $i = 1 \ldots j$ iteriert, und $e(i, j)$ wird benutzt.
- **Gesamtlaufzeit:** $O(n^3)$. (Kann mit Vorberechnung der $e(i, j)$ in $O(n^2)$ reduziert werden.)

Rucksackproblem (DP-Lösung)

Das Rucksackproblem kann auch mit DP gelöst werden, insbesondere wenn die Kapazität G klein ist.

Falscher Ansatz: $OPT(i)$ als maximaler Profit für Gegenstände $1, \ldots, i$ ohne Berücksichtigung der Kapazität. Scheitert, da die Kapazität eine wichtige Nebenbedingung ist.

Richtiger Ansatz:

Definition des Teilproblems: $OPT(i, g)$ = Maximaler Profit für die Teilmenge der Gegenstände $1, \ldots, i$ mit einer maximalen Gewichtsbeschränkung von g.

Rekursive Definition:

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

- `OPT(i - 1, g)`: Gegenstand i wird nicht gewählt.
- `w_i + OPT(i - 1, g - g_i)`: Gegenstand i wird gewählt.

Bottom-Up Implementierung

Eine $M[(n+1) \times (G+1)]$-Tabelle wird gefüllt. $M[i][g]$ speichert $OPT(i, g)$.

```
Initialisiere M[0][g] = 0 für alle g.
Initialisiere M[i][0] = 0 für alle i.

for i from 1 to n:
    for g from 1 to G:
        if g_i > g:  // Gegenstand i ist zu schwer
            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]
```

Laufzeit

- Die Laufzeit beträgt $O(nG)$, da eine Tabelle der Größe $(n+1) \times (G+1)$ gefüllt wird und jede Zelle in konstanter Zeit berechnet wird.
- Dies ist pseudo-polynomiell, da G exponentiell in der Eingabelänge (binäre Darstellung von G) sein kann. Wenn G klein ist, ist es sehr effizient.

Bestimmen der Lösung

Die ausgewählten Gegenstände können durch Backtracking der M-Tabelle rekonstruiert werden, indem man von $M[n][G]$ zurückverfolgt, welche Entscheidung (Gegenstand i nehmen oder nicht) zum aktuellen Wert geführt hat.

Kürzeste Pfade

Das Finden des kürzesten Pfades in Graphen mit beliebigen (auch negativen) Kantengewichten ist ein komplexes Problem.

Negative Kreise

- Wenn ein Graph einen negativen Kreis enthält (ein Kreis, dessen Kanten summiert einen negativen Wert ergeben), ist der kürzeste *Kantenzug* nicht mehr wohldefiniert, da man durch beliebiges Durchlaufen des Kreises die Länge immer weiter reduzieren kann.
- Kürzeste *Pfade* (jeder Knoten max. einmal besucht) können in Anwesenheit negativer Kreise jedoch noch sinnvoll definiert sein. Das Finden des kürzesten Pfades ist dann NP-vollständig. Ohne negative Kreise ist es polynomiell lösbar.

Bellman's Gleichungen

Für einen gerichteten Graphen $G = (V, E)$ ohne negative Kreise und ein Ziel $t \in V$ gilt für die Länge $OPT(v)$ eines kürzesten $v-t$-Pfades:

- $OPT(t) = 0$
- $OPT(v) = \min_{(v,w)\in E} c_{vw} + OPT(w)$ für alle $v \in V, v \neq t$.

Diese Gleichungen bilden die Basis für DP-Algorithmen.

Kürzester Pfad mit Dynamischer Programmierung (Bellman-Ford-ähnlich)

Definition: $OPT(i, v)$ = Länge eines kürzesten $v-t$-Pfades, der höchstens i Kanten benutzt.

Rekursive Definition:

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

- $OPT(i-1,v)$: Pfad benutzt weniger als i Kanten.
- $\min_{(v,w)\in E} c_{vw} + OPT(i-1,w)$: Pfad benutzt genau i Kanten, erste Kante (v,w), dann optimaler Pfad mit $i-1$ Kanten.

Die endgültige Länge des kürzesten Pfades ist $OPT(n-1,v)$, da ein Pfad in einem Graphen mit n Knoten maximal $n-1$ Kanten haben kann, ohne einen Kreis zu enthalten.

Implementierung und Laufzeit

Ein iterativer Algorithmus berechnet $OPT[i][v]$ für $i = 1 \ldots n-1$ und alle $v \in V$.

- Tabellengröße: $O(n^2)$.
- Jede Zelle: Iteration über ausgehende Kanten $O(\deg(v))$.
- Gesamtlaufzeit: $O(n \cdot m)$ (n Iterationen, m Kanten bei jedem Schritt).

Überprüfung auf negative Kreise

Nachdem der Algorithmus terminiert ist (nach n Iterationen), kann man negative Kreise erkennen: Wenn für irgendeinen Knoten v gilt $OPT(n,v) < OPT(n-1,v)$, dann existiert ein negativer Kreis, der über v erreichbar ist.

13. Approximation

Einführung in Approximationsalgorithmen und Gütegarantien

Approximationsalgorithmen sind nützlich, um gute, wenn auch nicht unbedingt optimale, Lösungen für NP-schwere Probleme in polynomieller Zeit zu finden. Die Qualität der Lösung wird durch eine Gütegarantie (ε) quantifiziert.

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

- Für Minimierungsprobleme: Wenn ein $\varepsilon \geq 1$ existiert, sodass für alle Instanzen x von X gilt:

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

  dann ist A ein ε-Approximationsalgorithmus.
- Für Maximierungsprobleme: Wenn ein ε mit $0 < \varepsilon \leq 1$ existiert, sodass für alle Instanzen x von X gilt:

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

  dann ist A ein ε-Approximationsalgorithmus.

Ein $\varepsilon = 1$ bedeutet, dass A ein exakter Algorithmus ist.

Approximationsalgorithmus für Vertex Cover

Ein Vertex Cover C eines Graphen $G = (V, E)$ ist eine Knotenmenge, sodass jede Kante des Graphen zu mindestens einem Knoten in C inzident ist. Das Finden eines minimalen Vertex Cover ist ein NP-schweres Problem.

Der 2-Approximationsalgorithmus: Approx-Vertex-Cover(G)

Dieser Algorithmus findet ein Vertex Cover, das höchstens doppelt so groß wie das Optimum ist:

1. Initialisiere $C = \emptyset$ und $E' = E$.
2. Solange E' nicht leer ist:
   - Wähle eine beliebige Kante $(u, v) \in E'$.
   - Füge u und v zu C hinzu.
   - Entferne alle Kanten aus E', die zu u oder v inzident sind.
3. Gib C zurück.

Laufzeit: Die Laufzeit dieses Algorithmus ist polynomiell, $O(n + m)$.

Gütegarantie:
Sei M die Menge der vom Algorithmus ausgewählten Kanten. M ist ein Matching. Jede Kante in M muss von C^* (dem optimalen Vertex Cover) abgedeckt werden, was impliziert, dass $|C^*| \geq |M|$. Da `Approx-Vertex-Cover` ein Vertex Cover der Größe $2|M|$ findet, gilt für alle Instanzen x:

$$c_{AVC}(x) = 2|M| \leq 2 \cdot c_{opt}(x)$$

Die Gütegarantie ist somit $\varepsilon = 2$. Diese Schranke ist scharf, beispielsweise für vollständige bipartite Graphen $K_{n,n}$.

Alternativer Algorithmus: Auswahl nach maximalem Grad

Ein anderer Ansatz, der den Knoten mit dem höchsten Grad wählt, hat eine schlechtere logarithmische Gütegarantie ($\Omega(\log n)$). Dies zeigt, dass eine scheinbar "intelligentere" lokale Greedy-Entscheidung nicht unbedingt zu einer besseren globalen Approximation führt.

Spanning-Tree-Heuristik (ST) für das symmetrische TSP

Das Traveling Salesperson Problem (TSP) ist ein NP-schweres Minimierungsproblem. Beim symmetrischen TSP sind die Distanzen $c_{ij} = c_{ji}$.

Minimum Spanning Tree Heuristik (MST)

Die Heuristik leitet eine Tour aus einem Minimum Spanning Tree (MST) ab:

1. Bestimme einen MST (V, B) des Graphen G.
2. Verdopple alle Kanten in B zu (V, B_2). Dieser Graph ist eulersch (jeder Knoten hat geraden Grad).
3. Bestimme eine Euler-Tour T in (V, B_2).
4. Konvertiere die Euler-Tour in eine Hamilton-Tour T' durch "Abkürzen" (überspringe bereits besuchte Knoten).

Laufzeit: Die Heuristik ist polynomiell, dominiert durch die Berechnung des MST ($O(n^2)$).

Approximierbarkeit des symmetrischen TSP

Theorem: Unter der Annahme $P \neq NP$, gibt es keinen polynomiellen ε-Approximationsalgorithmus für das symmetrische TSP für ein beliebiges $\varepsilon > 1$. Dies wird durch eine Reduktion vom Hamiltonkreis-Problem bewiesen.

Beweisskizze: Man konstruiert eine TSP-Instanz G' aus einem Graphen G: Kanten, die in G existieren, erhalten Kosten 1; Kanten, die nicht in G existieren, erhalten sehr hohe Kosten ($2n + 1$ oder $\varepsilon n + 1$). Wenn G einen Hamiltonkreis hat, existiert eine Tour der Kosten n. Wenn nicht, ist jede Tour deutlich teurer (mindestens $3n$). Ein ε-Approximationsalgorithmus könnte dann das NP-schwere Hamiltonkreis-Problem lösen, was einen Widerspruch zu $P \neq NP$ darstellt.

Fazit: Das allgemeine symmetrische TSP ist nicht approximierbar.

Metrisches TSP

Ein TSP ist metrisch, wenn die Dreiecksungleichung für die Distanzen gilt: $c_{ik} \leq c_{ij} + c_{jk}$. Das Euklidische TSP ist ein Beispiel dafür.

Theorem: Das metrische TSP besitzt einen polynomiellen 2-Approximationsalgorithmus durch die Spanning-Tree-Heuristik. Für eine gegebene Instanz x gilt:

$$\frac{c_{ST}(x)}{c_{opt}(x)} \leq 2$$

Beweisskizze:

- $c_{ST}(x) \leq c_{B_2}(x)$: Aufgrund der Dreiecksungleichung kann das "Abkürzen" die Tourkosten nur reduzieren.
- $c_{B_2}(x) = 2c_B(x)$: Die Kanten des MST werden verdoppelt.
- $c_B(x) \leq c_{opt}(x)$: Jede optimale TSP-Tour enthält einen Spannbaum. Der MST ist der "billigste" Spannbaum.

Kombiniert man diese, ergibt sich $c_{ST}(x) \leq 2c_{opt}(x)$.

Lastverteilung (Load Balancing)

Problem: Verteile n Jobs mit Bearbeitungszeiten t_j auf m identische Maschinen, um die maximale Maschinenlast (Makespan) $L = \max L_i$ zu minimieren.

List-Scheduling-Algorithmus

Dieser Greedy-Algorithmus weist jeden Job der Maschine mit der aktuell geringsten Last zu.

Laufzeit: $O(n \log m)$ mit einer Prioritätswarteschlange.

Theorem: List-Scheduling ist ein 2-Approximationsalgorithmus.
Wichtige untere Schranken für die optimale Dauer L^*:

1. $L^* \geq \max_j t_j$
2. $L^* \geq \frac{1}{m} \sum_j t_j$

Beweisskizze: Betrachtet man die Maschine mit der maximalen Last L und den letzten dort zugewiesenen Job j, so muss diese Maschine vor der Zuweisung von j die geringste Last gehabt haben. Daraus lässt sich folgern, dass $L \leq 2L^*$.

Center Selection

Problem: Wähle k Mittelpunkte C aus n Standorten $s_1, \ldots, s_n$, um die maximale Distanz von einem Standort zum nächsten Mittelpunkt zu minimieren ($r(C) = \max_{s_i \in S} \min_{c \in C} dist(s_i, c)$). Die Distanzfunktion erfüllt die Dreiecksungleichung.

Greedy-Algorithmus (korrekter Ansatz)

Ein naiver Greedy-Ansatz ist ungeeignet. Der korrekte Ansatz ist:

1. Wähle den ersten Mittelpunkt beliebig aus den Standorten.
2. Wähle iterativ den nächsten Mittelpunkt als den Standort, der am weitesten von jedem existierenden Mittelpunkt entfernt ist.

Laufzeit: Abhängig von der Implementierung, z.B. $O(nk)$.

Beobachtung: Nach Terminierung sind alle Mittelpunkte in C paarweise mindestens $r(C)$ voneinander entfernt.

Analyse: Gütegarantie des Greedy-Algorithmus

Theorem: Sei C^* eine optimale Menge von Mittelpunkten. Dann gilt $r(C) \leq 2r(C^*)$. Der Greedy-Algorithmus ist ein 2-Approximationsalgorithmus für das Center-Selection-Problem.

Beweisskizze (durch Widerspruch): Angenommen, $r(C) > 2r(C^*)$. Dies würde bedeuten, dass die vom Algorithmus gewählten k Mittelpunkte so weit voneinander entfernt sind, dass selbst die optimalen k Mittelpunkte nicht ausreichen würden, um alle Standorte innerhalb eines Radius von $r(C^*)$ abzudecken, was der Definition von C^* widerspricht. Die Dreiecksungleichung spielt hier eine entscheidende Rolle.

Anmerkung: Die Gütegarantie von 2 gilt auch, wenn optimale Mittelpunkte nicht auf Standorten liegen müssen.

Theorem: Ein ε-Approximationsalgorithmus für das Center-Selection-Problem mit beliebigem $\varepsilon < 2$ existiert nur, wenn $P = NP$ gilt. Dies klassifiziert das Problem als **NP-schwer** für eine bessere als 2-Approximation.

14. Heuristiken und Lokale Suche

Verfahren in der Optimierung

Die Optimierung von komplexen Problemen nutzt verschiedene Verfahren:

- Konstruktionsverfahren: Erstellen eine initiale Lösung.
- Verbesserungsheuristiken (Lokale Suche): Verbessern eine bestehende Lösung iterativ.
- Metaheuristiken: Übergeordnete Strategien, die lokale Suchverfahren erweitern, um globale Optima besser zu finden. Dazu gehören:
  - Simulated Annealing
  - Tabu-Suche
  - Evolutionäre Algorithmen

Konstruktionsverfahren

Eigenschaften

- Meist sehr problemspezifisch.
- Häufig intuitiv gestaltet.
- Oft basierend auf dem Greedy-Prinzip.

Greedy-construction-heuristic

Ein iteratives Verfahren, das mit einer leeren Lösung beginnt und diese schrittweise erweitert, indem in jedem Schritt die aktuell nützlichste Erweiterung gewählt wird.

$$
\begin{aligned}
&\text{Greedy-construction-heuristic}(P): \\
&1: \quad x \leftarrow \emptyset \\
&2: \quad \textbf{while } x \text{ ist nicht vollständig } \textbf{do} \\
&3: \qquad e \leftarrow \text{bestimme die aktuell nützlichste Erweiterung für } x \\
&4: \qquad x \leftarrow x \oplus e \\
&5: \quad \textbf{return } x
\end{aligned}
$$

Beispiele für Konstruktionsverfahren

- Minimaler Spannbaum (MST): Prim- und Kruskal-Algorithmen (exakte Greedy-Algorithmen).
- 0/1-Rucksackproblem: First-Fit-Heuristik (sortiert nach Wert/Gewicht).
- Vertex Cover Problem: Der 2-Approximationsalgorithmus `Approx-Vertex-Cover`.
- Lastverteilung: List Scheduling Algorithmus mit einer Gütegarantie von 2.
- TSP (Traveling Salesperson Problem):
  - Spanning-Tree-Heuristik: Gütegarantie von 2 für das metrische TSP (wenn die Dreiecksungleichung gilt).
  - Nearest-Neighbor-Heuristik: Geht immer zum nächsten unbesuchten Nachbarn.
  - Insertion-Heuristiken: Startet mit kleiner Tour, fügt Knoten schrittweise ein. Strategien wie Nearest, Cheapest, Farthest oder Random Insertion existieren. Praktische Ergebnisse sind oft $10-20\%$ über dem Optimum, jedoch keine konstante Gütegarantie für metrisches TSP.

Lokale Suche (Verbesserungsheuristiken)

Konstruktionsheuristiken liefern oft nur Ausgangslösungen. Die lokale Suche versucht, diese Lösungen durch kleine iterative Änderungen zu verbessern.

- In aller Regel keine Gütegarantien.
- In der Praxis oft deutliche Verbesserung der Lösungen.
- Meist akzeptable Laufzeiten.

Prinzip der Lokalen Suche

$\text{Lokale Suche}(x_0): 1: \quad x \leftarrow x_0 \text{ (initiale Lösung)} 2: \quad \textbf{while } \exists x' \in N(x) \text{ mit } f(x') \text{ besser als } f(x) \textbf{ do} 3: \quad x \leftarrow$

Dabei ist $N(x)$ die Nachbarschaft von x, d.h., die Menge aller Lösungen, die durch eine "kleine" Änderung aus x entstehen.

Komponenten einer lokalen Suche

- Lösungsrepräsentation: Wie ist eine Lösung kodiert?
- Nachbarschaftsstruktur ($N(x)$): Welche Lösungen sind "Nachbarn"? Definiert durch mögliche "Züge" (Moves).
  - Größe von $N(x)$ ist ein Kompromiss zwischen Suchaufwand und Qualität.
- Schrittfunktion: Wie wird der nächste Nachbar aus $N(x)$ ausgewählt (z.B. Best Improvement, First Improvement, Random Neighbor).
- Terminierungskriterium: Wann wird die Suche beendet (z.B. Erreichen eines lokalen Optimums, maximale Iterationen, Zeitlimit).

Lokale Suche: Vertex Cover

Problem: Finde eine minimale Teilmenge $C \subseteq V$ von Knoten, die alle Kanten abdeckt.

Nachbarschaftsstruktur: $N(C)$ enthält C' wenn C' aus C durch Löschen eines einzelnen Knotens erzeugt werden kann *und* C' immer noch ein gültiges Vertex Cover ist. $|N(C)| = O(|V|)$.

Algorithmus: Starte mit $C = V$. Iterativ wird ein Knoten entfernt, wenn das Ergebnis immer noch ein gültiges VC ist.

- Terminiert nach $O(|V|)$ Schritten.
- Liefert nicht immer eine optimale Lösung, kann in lokalen Optima stecken bleiben.

Beispiel für Einschränkung: Bei manchen Graphen (z.B. Dreiecke) kann kein einzelner Knoten entfernt werden, ohne die VC-Eigenschaft zu verlieren, obwohl das aktuelle VC nicht optimal sein muss.

Alternative Nachbarschaft $\mathcal{N}'$: Erlaubt z.B. das Entfernen von zwei Knoten und das Hinzufügen eines Knotens. Erhöht die Komplexität auf $O(|\mathcal{V}|^3)$ pro Iteration.

Lokale Suche: SAT (Optimierungsvariante: MAX-SAT)

Problem: Finde eine Variablenzuweisung, die die Anzahl der erfüllten Klauseln maximiert.

Lösungsrepräsentation: Binärer Vektor $x \in \{0,1\}^n$.

k-flip Nachbarschaft: Nachbarlösungen unterscheiden sich in bis zu k Bits (Hamming-Distanz $\leq k$). Größe $O(n^k)$.

Lokale Suche für das symmetrische TSP

Nachbarschaftsstruktur: 2-exchange (2-opt). Hierbei werden zwei nicht benachbarte Kanten (i_p, i_{p+1}) und (i_q, i_{q+1}) aus der Tour entfernt und durch (i_p, i_q) und (i_{p+1}, i_{q+1}) ersetzt, wenn dies die Tourlänge verkürzt.

- Größe der Nachbarschaft: $O(n^2)$.
- Inkrementelle Evaluierung des Zielfunktionswerts in $O(1)$.
- Eine Iteration benötigt $O(n^2)$ Zeit.
- Worst-Case Iterationen: Bis zu $O(n!)$ (exponentiell). In der Praxis jedoch meist schnell konvergent.

r-opt Nachbarschaft: Verallgemeinerung auf r Kanten. Größe $O(n^r)$.

Praktische Ergebnisse:

- 2-opt: Oft $6 - 8\%$ über dem Optimum.
- 3-opt: Oft $3 - 4\%$ über dem Optimum (deutlich zeitaufwändiger).
- Lin-Kernighan Heuristik: Eine führende Heuristik für große TSPs, oft $1 - 2\%$ über dem Optimum, nutzt eine variable Tiefensuche.

Maximaler Schnitt (MAX-CUT)

Problem: Partitioniere die Knoten eines Graphen $G = (V, E)$ mit Kantengewichten w_{uv} in zwei Mengen (A, B), sodass $|A| = |B|$ (falls möglich) und das Gesamtgewicht der Kanten zwischen A und B maximiert wird ($w(A, B) = \sum_{u \in A, v \in B} w_{uv}$). MAX-CUT ist NP-vollständig.

1-Flip-Nachbarschaft: Verschieben eines einzelnen Knotens von A nach B oder umgekehrt.

Analyse der lokalen Suche: Eine lokal optimale Partition (A, B) (bezüglich 1-Flip-Nachbarschaft) erfüllt eine Approximationsgüte von $1/2$:

$$w(A, B) \geq \frac{1}{2} \sum_{e \in E} w_e \geq \frac{1}{2} w(A^*, B^*)$$

wobei (A^*, B^*) eine optimale Partition ist.

Metaheuristiken

Definition: Problemunabhängig formulierte Algorithmen zur heuristischen Lösung schwieriger Optimierungsaufgaben. Sie müssen in Teilen an das Problem angepasst werden (z.B. Nachbarschaftsstruktur).

Simulated Annealing (SA)

Inspiriert durch physikalische Abkühlungsprozesse.

- Grundidee: Akzeptiert auch schlechtere Nachbarlösungen mit einer bestimmten Wahrscheinlichkeit, um lokale Optima zu verlassen.
- Schrittfunktion: Meist Random Neighbor.
- Metropolis-Kriterium: Die Akzeptanzwahrscheinlichkeit einer schlechteren Lösung x' ist gegeben durch $P(\text{accept } x') = e^{-|f(x') - f(x)|/T}$.
    - Geringfügig schlechtere Lösungen werden mit höherer Wahrscheinlichkeit akzeptiert.

    - Bei hoher Temperatur (T) werden schlechtere Lösungen mit größerer Wahrscheinlichkeit akzeptiert als bei niedrigerer Temperatur.
- Abkühlplan: Definiert, wie T reduziert wird.
    - Geometrisches Abkühlen: $T \leftarrow T \cdot \alpha$ mit $\alpha < 1$ (z.B. 0.999).
    - Adaptives Abkühlen: T wird basierend auf dem Anteil der Verbesserungen angepasst.

Beispiel: Graph-Bipartitionierung

- Ziel: Partitioniere V in A, B mit $|A| = |B|$, minimiere Kanten zwischen A und B.
- Nachbarschaft: Tausch eines Knotens von A mit einem von B.
- SA war eine der ersten Anwendungen für dieses Problem.

Fazit zu SA: Einfach zu implementieren, erfordert Parametertuning, liefert oft gute Ergebnisse.

Tabu-Suche (TS)

- Basiert auf einem Gedächtnis (History) des Suchverlaufs, um lokale Optima zu überwinden.
- Vermeidung von Zyklen: Verbot des Wiederbesuchens früherer Lösungen.
- Schrittfunktion: Im Allgemeinen Best Improvement der erlaubten Nachbarlösung (auch wenn diese schlechter ist).

Das Gedächtnis: Tabuliste

- Speichert Tabuattribute (z.B. Variablenwerte oder umgekehrte Züge), die für eine bestimmte Zeit t_L verboten sind.
- Tabulistenlänge t_L: Kritischer Parameter. Zu kurz $\rightarrow$ Zyklen; zu lang $\rightarrow$ starke Einschränkung des Suchraums. Oft experimentell oder adaptiv bestimmt.
- Aspirationskriterien: Erlauben es, eine eigentlich tabuierte Lösung zu wählen, wenn sie z.B. besser ist als die bisher beste gefundene Lösung.

Beispiel: Graphenfärbeproblem

- Ziel: Minimiere die Anzahl der "verletzten" Kanten (benachbarte Knoten mit gleicher Farbe).
- Nachbarschaft: (Hier fehlt im bereitgestellten Text die konkrete Definition der Nachbarschaft für das Graphenfärbeproblem, aber typisch wäre z.B. das Ändern der Farbe eines einzelnen Knotens.)