Matchings

Graphische Datenverarbeitung I
WiSe 2024/25

1 Definition

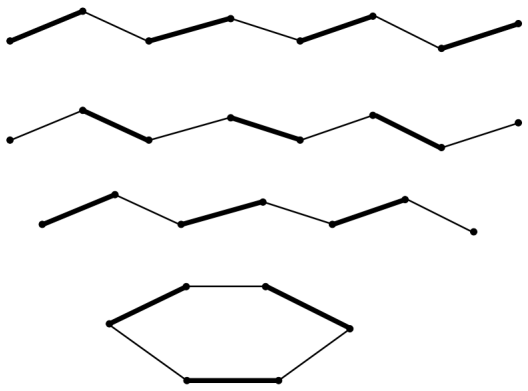
Ein Matching-Problem ist ein Maximierungsproblem in dem das Ziel es ist eine Maximale Anzahl an Matchings in einem ungerichteten Graphen zu finden.

Gegeben sei ein ungerichteter Graph $G = (V, E)$. Eine zulässige Lösung ist eine Menge $M \subseteq E$, sodass keine Kante in M einen Knoten gemeinsam haben. Wenn dies gilt, dann ist M ein Matching von G. F_G ist die Menge aller Matchings in G. Die Zielfunktion zählt dann die Anzahl der Kanten im Matching.

2 Beweis: Matching hat eine exakte Nachbarschaft

2.1 Alternierende Pfade

Um die Nachbarschaft verstehen zu können, müssen wir zunächst Alternierende Pfade verstehen.
Ein Pfad p in G ist ein **elementarer Pfad** wenn jeder Knoten im Pfad nur einmal vorkommt.
Ein elementarer Pfad p in G ist **alternierend** wenn genau jede zweite Kante von p zu M gehört.
Alternierende Pfade können folgende Struktur haben. In diem Schaubild sind dann die dicken schwarzen Linien Kanten vom Matching M.



2.2 Nachbarschaftsdefinition im Matching

Zwei Matchings M_1 und M_2 in einem ungerichteten Graph $G = (V, E)$ sind **benachbart**, wenn die **symmetrische Differenz** ein einzelner alternierender Pfad aus M_1 und M_2 erzeugt.

Die symmetrische Differenz ist eine Menge von Kanten die folgendermaßen definiert ist:

- umfassen alle Kanten, die **entweder** in M_1 **oder** in M_2 enthalten sind, aber **nicht in beiden gleichzeitig**.
- $M_1 \triangle M_2 = (M_1 \backslash M_2) \cup (M_2 \backslash M_1)$

2.3 Lemma: Diese Nachbarschaftsdefinition ist exakt

Wir müssen nun zeigen, dass bei zwei gegebenen Matchings M_1 und M_2 mit $|M_2| > |M_1|$ folgendes gilt:

- es existiert ein alternierender Pfad p, sodass gilt: $|M_1 \triangle p| > |M_1|$

2.3.1 Symmetrische Differenz und Knotengrad

Jedes Matching kann maximal eine Kante pro Knoten enthalten. Somit hat jeder Knoten einen Grad von höchstens 2 in der symmetrischen Differenz. Dies bedeutet, dass die symmetrische Differenz sich in eine Menge von Pfaden und Zyklen zerlegt, die alternierend Kanten aus M_1 und M_2 enthalten.

2.3.2 Alternierende Pfade

Weil die Kanten im alternierenden Pfad abwechselnd aus M_1 und M_2 stammen, und $|M_2| > |M_1|$ gilt, muss es **mind. einen Pfad** geben, der mehr Kanten aus M_2 enthält als aus M_1.
Ein solcher Pfad kann verwendet werden, um das kleinere Matching M_1 zu vergrößern, indem man Kanten aus M_2 hinzufügt und dabei die alternierende Struktur beibehält.

2.3.3 Existenz des gewünschten Pfads

Da mindestens ein Pfad mehr Kanten aus M_2 enthält, führt die symmetrische Differenz mit diesem Pfad zu einer Vergrößerung von M_1 führt. Das bedeutet, dass $|M_1 \triangle p| > |M_1|$ gilt, und das führt zu einem größeren Matching.

3 Zusatz: Lokale Suche für Matching, Theorem von Berge und augmentierte Pfade

Sei M^* das aktuelle Matching, und M eine Nachbarlösung von M^*. Das bedeutet, dass $M^* \triangle M$ ein alternierenden Pfad erzeugt. Dieser alternierende Pfad ist **augmentierend**, wenn beide Endknoten exposed sind bezüglich M^*.

Das Theorem von Berge besagt:

> Ein Matching Mist genau dann maximal, wenn es keinen M-augmentierenden Pfad gibt.

Das beduetet, dass ein Matching nur dann vergrößert werden kann, wenn es einen augmentierenden Pfad gibt, der mehr ungematchte Kanten enthält als gematchte. Andernfalls ist das Matching maximal und ein globale Optimum ist erreicht.

Min-Cost-Flows

Graphische Datenverarbeitung I
WiSe 2024/25

1 Definition

Ein Minimum Cost Flow ist ein Minimierungsproblem, indem das Ziel es ist einen Fluss zu finden, der alle Kapazitätsconstraints einhält und einen maximalen Output erzeugt.
Folgende Dinge müssen gegeben sein:

- ein gerichteter Graph $D = (V, A)$
- untere und obere Kapazitätswerte (Schranken) $0 \leq l[a] \leq u[a] \in \mathbb{R}$
- ein Kostenfaktor $c[a] \in \mathbb{R}$ für jede Kante $a \in A$
- ein Balancewert $b[v] \in \mathbb{R}$ für jeden Knoten $v \in V$

Der gewünschte Output der Algorithmen wäre ein **Flowwert** $f[a] \in \mathbb{R}$ für jede Kante $a \in A$, sodass gilt:

- **Kapazitätsconstraints einhalten**: $l[a] \leq f[a] \leq u[a]$ für jede Kante $a \in A$
- **Flow Balance Constraints einhalten**: für jeden Knoten $v \in V$:

$$\sum_{\substack{w \in V \\ (v,w) \in A}} f[(v,w)] - \sum_{\substack{w \in V \\ (v,w) \in A}} f[(w,v)] = b[v]$$

Das Ziel ist es diese Summe zu minimieren: $\sum_{a \in A} c[a] * f[a]$.
Wenn $b \equiv 0$, dann sind die Flows **zirkuliert**

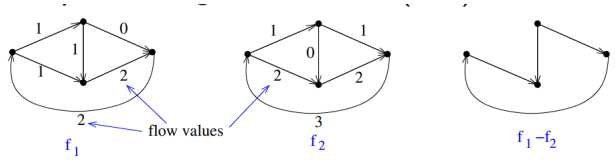
1.1 Weak Paths und Cycles

- Ein Weak Path ist eine Folge von Kanten, bei denen Vorwärts- und Rückwärtsbewegungen entlang der Kanten erlaubt sind. Das bedeutet, dass ein Pfad entlang der Richtung der Kante oder gegen die Richtung der Kante verlaufen kann.
- Ein Weak Cycle ist ein geschlossener Zyklus, der ebenfalls vorwärts und rückwärts entlang der Kanten verlaufen kann. Es gibt also Richtungswechsel.

2 Nachbarschaftsdefinition von Min-Cost-Flows

Sei $(D, l, u, b, c), D = (V, A)$ eine beliebige, aber fixierte Instanz.

Seien f_1 und f_2 zwei zulässige Flüsse für (D, l, u, b, c). Dann sind f_1 und f_2 benachbart, wenn sie sich in nur einem elementaren Weak Cycle unterscheiden.



Das bedeutet, wenn wir die Flussdifferenz der beiden Flüsse nehmen, entsteht ein Weak Cycle, wenn diese Flüsse benachbart sind. Wenn abgezweigte Teilpfade entstehen, ist dies kein benachbarter Fluss.

3 Lemma: Diese Nachbarschaftsdefinition ist exakt

3.1 Prerequisites

3.1.1 Augmentierende Pfade

Sei p ein elementärer Pfad oder Zyklus in G. Und Seien $A_{\text{forw}}(p)$ und $A_{\text{backw}}(p)$ die vorwärts und rückwärts Achsen von p.
Dann wid p als **augmenting** bezeichnet, wenn für die Flusswerte f, unteren Schranken l und oberen Schranken u gelten:

- $f[a] < u[a]$ für alle $a \in A_{\text{forw}}(p)$; Vorwärts Fluss ist kleiner als das Maximum
- $f[a] > l[a]$ für alle $a \in A_{\text{backw}}(p)$; Rückwärts Fluss ist größer als das Minimum

Somit ist ein Augmenting Pfad ein Pfad, auf dem man den Fluss erhöhen oder verringern kann, ohne die Grenzen zu überschreiten/unterschreiten.

Zwei schwache Pfade (oder Zyklen) p_1, p_2 sind **konsistent**, wenn gilt:

- $A_{\text{forw}}(p_1) \cap A_{\text{backw}}(p_2) = \emptyset$
- $A_{\text{backw}}(p_1) \cap A_{\text{forw}}(p_2) = \emptyset$

$\Rightarrow$ konsistent, wenn keine Vorwärtskanten von p_1 die Rückwärtskanten von p_2 sind und vice versa.

3.1.2 Negative Zyklen

Aufgrunddessen, dass Flüsse wie Vektoren auf den Kanten betrachtet werden, kann man zwei Flüsse komponentenweise kombinieren.

Für einen schwachen Zyklus p, beschreibt ϵ dine zyklische Umlagerung des Flusses. Dabei gilt:

- Vorwärts auf dem Zyklus $\Rightarrow$ Fluss wird um $+\epsilon$ erhöht
- Rückwärts auf dem Zyklus $\Rightarrow$ Fluss wird um $-\epsilon$ verringert
- auf allen anderen Kanten bleibt der Fluss unverändert

Ein Zyklus p wird **negativ** bzgl. einer Kostenfunktion genannt, wenn gilt:

$$\sum_{a \in A_{\text{forw}}(p)} c[a] - \sum_{a \in A_{\text{backw}}(p)} c[a] < 0$$

$\Rightarrow$ ein Zyklus ist negativ, wenn die Kosten für den Vorwärtsfluss kleiner sind als die Kosten für den Rückwärtsfluss.

3.1.3 Flusszerlegung

Für zwei zulässige Flüsse f_1 und f_2, existiert ein nicht-negativer Integer p und ein Weak Cycle $p_1, ..., p_k$, sodass gilt:

- $\exists \epsilon_1, ..., \epsilon_k > 0$ sodass gilt: $f_2 = f_1 + \epsilon_1 * p_1 + ... + \epsilon_k * p_k$.
- Für jeden Zyklus $p_i, i \in \{1, ..., k\}$ gilt:
  - $f_2[a] > f_1[a], \forall a \in A_{\text{forw}}(p)$
  - $f_2[a] < f_1[a], \forall a \in A_{\text{backw}}(p)$

Das bedeutet, dass man von f_1 zu f_2 gelangen kann, indem man Flussänderungen auf Zyklen vornimmt. Man kann den Unterschied zwischen Flüssen in Weak Cycles zerlegen.

3.1.4 Beweis der Exaktheit der Nachbarschaft

Exaktheit bedeutet hier: Wenn ein Fluss nicht optimal ist, dann gibt es immer eine lokale Änderung (auf einem Weak Cycle), die zu einem niedrigeren Kostenfluss führt.

1. Annahme: f_1 ist nicht optimal
2. Ziel: Zeigen, dass es einen negativen Zyklus gibt, der die Kosten verringert
3. Vorgehen
   - es wird angenommen, dass es einen besseren Fluss f_2 gibt, dessen Kosten geringer sind als f_1.

- daher dass f_2 optimal ist, müssen die Kosten von $f2$ kleiner sein.

Laut der Flusszerlegung, kann man die Differenz zwischen f_1 und f_2 in Weak Cycles zerlegen:

$$f_2 = f_1 + \epsilon_1 * p_1 + ... + \epsilon_k * p_k$$

Also kann f_2 durch Änderungen auf diesen Zyklen aus f_1 gebildet werden. Daher dass f_2 günstiger ist als f_1, muss mindestens ein Zyklus p_i negative Kosten haben, weil nur negative Zyklen die Gesamtkosten senken können.
Dieser negative Zyklus p_i ermöglicht es, den Fluss lokal zu ändern und die Kosten zu reduzieren. Dieser Zyklus p_i ist ein augmentierender Pfad, da die Änderung des Flusses auf den Vorwärts-Kanten innerhalb der Obergrenzen bleibt und auf den Rückwärtskanten nicht unter die Untergrenzen fällt.
Dadurch ist $f_1 + \epsilon_i * p_i$ ein weiterhin zulässiger Fluss und die Exaktheit der Nachbarschaft ist bewiesen.

3.2 Zusammenfassung

Wenn ein Fluss nicht optimal ist, dann gibt es immer einen negativen Zyklus, der die Kosten lokal senken kann. Dieser Zyklus kann durch die Flow Decomposition gefunden werden. Damit ist jede Verbesserung lokal erreichbar.

Simulated Annealing

Graphische Datenverarbeitung I
WiSe 2024/25

1 Definition

Simulated Annealing ist ein Suchalgorithmus, welche stark auf der lokalen Suche aufbaut. Der Unterschied hierbei ist, dass eine probabilistische Entscheidung getroffen wird, ob eine Lösung angenommen wird. Auch wenn diese Lösung nicht besser als vorherige Lösung ist, wodurch die Exploration der Suche verbessert wird.
Hierbei wird eine Temperatur gesetzt, die die Probabilistische Entscheidung beeinflusst. Je höher die Temperatur desto wahrscheinlicher, dass eine Lösung angenommen wird. Die Temperatur sinkt mit der Zeit anhand einer Cooling Rate.
Die Terminierung des Algorithmus basiert aufgrund einer Endtemperatur und fixed CPU Time.

1.1 Markov-Kette

Eine Markov-Kette ist ein stoschastisches Modell, dass eine Folgen von Zuständen beschreibt, bei der die Übergangswahrscheinlichkeit zum nächsten Zustand nur vom aktuellen Zustand abhängt. Only the present is important.

Eine Markov Kette heißt **aperiodisch**, wenn es keinen festen Zyklus gibt, nach dem Zustände wiederholt werden.

- Ein Zustand s hat Periode d, wenn d der größte gemeinsame Teiler aller möglichen Schritte k ist, bei denen der Zustand wieder zu sich selbst zurückkehren kann.
- Wenn alle Zustände die Periode 1 haben, ist die Kette aperiodisch.

Eine Markov-Kette ist **irreduzibel**, wenn jeder Zustand von jedem anderen Zustand aus in endlich vielen Schritten erreicht werden kann. Somit sind keine Zustände isoliert und der gesamte Zustandsraum ist abgedeckt.
Eine Markov-Kette ist **ergodisch**, wenn sie irreduzibel und aperiodisch ist.

$\Rightarrow$ eine ergodische Markov-Kette stellt somit sicher, dass von jedem Startzustand eine stationäre Verteilung erreicht werden kann, und die Suche nicht in Zyklen oder Teilbereichen gefangen bleibt.

1.2 Stark zusammenhängende Nachbarschaft

Eine Nachbarschaftsstruktur in einem Graphen ist stark zusammenhängend, wenn jeder Zustand von jedem anderen Zustand aus in endlich vielen Schritten erreicht werden kann.

2 Konvergenz für konstante Temperatur

Simulated Annealing konvergiert und der Vorausserzung, dass die Temperatur positiv konstant ist, weil die Markov-Kette:

- irreduzibel ist, was garantiert, dass der Algorithmus jede mögliche Lösung erreichen kann
- aperiodisch ist, wodurch keine zyklischen Verhalten auftreten und alle Zustände zufällig besucht werden
- wodurch die Markov-Kette ergodisch ist, was bedeutet, dass sie eine eindeutige stationäre Verteilung besitzt.

3 Konvergenz für zulässige Lösung mit T $\to$ 0 und für $k \to \infty$

Für eine zulässige Lösung s gilt:

$$\lim_{T \to 0} \lim_{k \to \infty} P_{k,T}(s) = 0$$

Falls s nicht optimal ist, dann bedeutet es, dass nicht optimale Lösungen bei $T \to 0$ und unendlich vielen Iterationen eine Wahrscheinlichkeit von 0 haben. Somit bleiben nur optimale Lösungen übrig.

3.1 Beweisidee

Betrachtet man zwei zulässige Lösungen s_1 und s_2 mit:

$$obj(s_1) < obj(s_2)$$

Das bedeutet, dass s_1 besser ist als s_2. Die Wahrscheinlichkeit von s_1 gegenüber s_2 wird unendlich größer.
Daraus lässt sich schlussfolgern, dass die Wahrscheinlichkeit eine Wertgrenze von 1 hat, muss die Wahrscheinlichkeit von s_2 gegen 0 gehen:

$$\lim_{T \to 0} \lim_{k \to \infty} P_{k,T}(s_2) = 0$$

Das gilt für alle schlechteren Lösungen im Vergleich zur optimalen Lösung.

3.2 Interpretation

Wenn wir Simulated Annealing mit unendlich vielen Temperaturwerten, die gegen 0 gehen ($T_1 > T_2 > ... \to 0$) und bei jeder Temperatur unendlich viele Iterationen ausführen, dann konvergiert der Algorithmus auf optimale Lösungen und verbleibt dort.
Das gilt unabhängig von der Startlösung, da die stationäre Verteilung nur von den Zielfunktionswerten abhängt. Bessere Lösungen haben immer eine höhere Wahrscheinlichkeit als schlechtere

Lösungen.

Ein logarithmisches Schema kann gut sein, um das Cooling Schedule zu bestimmen.

3.3 Wann ist Simulated Annealing optimal?

- Temperatur muss langsam genug gegen 0 gehen
- bei jeder Temperatur muss es unendlich viele Iterationen geben
- das Nachbarschaftssystem muss stark zusammenhängend sein, d.h. jeder Zustand ist von jedem anderen Zustand erreichbar

4 Fazit

Simulated Annealing findet immer die globalen Optima, wenn die Temperatur langsam genug gegen 0 geht und es genug Iterationen bei jeder Temperaturstufe gibt.
Problematisch ist jedoch, dass gescheite Lösungen sehr lange brauchen um diese zu berechnen und keine Qualität garantiert ist.

Entscheidungsbasierte Ansätze

Optimierungsalgorithmen
WiSe 2024/25

1 Entscheidungsbäume

In Feature-basierten Problemen, können die Features als Entscheidungen definiert werden die zuvor getroffen wurden. Diese können in einem Entscheidungsbaum dargestellt werden. Die Knoten stellen dann dar, ob ein Feature gewählt oder abgelehnt wurde, und stellen eine Menge von Lösungen dar, die bisherig akzeptierten Features beinhalten und nicht die die abgelehnt wurden. Ein Blatt ist ein Element des Lösungsraums.
Weil die Bäume exponentiell größer werden, je mehr Entscheidungen getroffen werden können, sind diese sehr ineffektiv zu traversieren. Um dieses Problem zu lösen, ist potenziell anstrengend oder heuristisch begrenzt. Um dies zu lösen kann der Teilbaum gepruned werden.

1.1 Enumeration und Zählen

- Enumerationsproblem: Gebe alle zulässigen Lösungen aus
- Countingprobleme: Gebe die Anzahl aller zulässigen Lösungen aus

1.2 Macht von Preprocessing

Das Ziel beim Preprocessing ist es eine Instanz zu einer kleineren Instanz zu transformieren, bei der wir wissen, wenn wir für die kleine Instanz eine optimale Lösung haben, eine optimale Lösung für die größere Instanz machen kann.

1.2.1 Kardinalitätsmatching

Sei $G_1 = (V, E)$ ein ungerichteter Graph, in der ein maximales Kardinalitätsmatching gesucht wird.
Transformation:
Sei v ein Knoten mit Grad 1 in G_1 und (v, w) eine Incident Edge. Lösche v, w und alle incident Edges von G_1 und erzeuge Graph G_2.
Wenn M_2 ist ein Maximum Kardinalitätsmatching in G_2 ist, dann $M_1 = M_2 \cup \{(v, w)\}$ ist ein Maximum Kardinalitätsmatching in G_1.
Die Transformationsregel soll dann so oft wir möglich angewendet werden.

1.2.2 Hitting Set

Problemdefinition

- Input: eine endliche Grundmenge F und eine Collection S von Untermengen von F.
- Output: eine Untermenge $F' \subseteq F$, sodass $s \cap F' \neq$ für alle $s \in S$
- Ziel: minimiere $|F'|$

Der Input ist ein Hypergraph.
Hypergraph: ein Paar (F, S), wobei F eine endliche Menge ist, und S eine Familien von Untermengen von F.
Die Elemente von F sind Knoten, und Elemente von S sind **hyperedges**.

2 Cutting Strategien

Um einen Teilbaum zu prunen, müssen wir sicherstellen, dass dieser Teilbaum auch wirklich nutzlos ist.
Ob ein Teilbaum definitiv nutzlos ist, kommt auf das Problem an. Wenn wir Machbarkeitsprobleme betrachten, dann kann der Teilbaum verworfen werden, wenn bereits eine zulässige Lösung außerhalb gefunden wurde. Bei Optimierungsproblemen kann ein Teilbaum unmöglich eine Lösung enthalten, wenn der Teilbaum unbeschränkt ist, eine optimale Lösung außerhalb des Teilbaums existiert oder eine bekannte Lösung existiert, die aber besser ist als jede potenzielle Lösung.

2.1 Brand-and-Bound

Nehmen wir an, wir kennen eine obere Schranke U eines Optimalenkostenwertes (jede zulässige Lösung mit einem Kostenwert kann als obere Schranke dienen). Außerdem nehmen wir an, dass für alle Knoten v des Entscheidungsbaums eine niedrige Schranke $l(v)$ berechnen für alle zulässigen Lösungen im Teilbaum mit Wurzel v.
Der Teilbaum ist **nutzlos**, wenn

- $l(v) = +\infty$, weil der Teilbaum unzulässig ist
- $l(v) > U$, weil keine optimale Lösung im Teilbaum ist
- $l(v) = c(s)$ für ein s. Wenn s im Teilbaum von v ist, dann haben wir eine optimale Lösung für den Teilbaum gefunden. Wenn dieser besser ist als zuvoriger Upper Bound, dann haben wir einen neuen Upper Bound. Dann können wir Prunen weil Optimale Lösung.

2.1.1 Wie berechnet man den ersten Upper Bound?

Je kleiner die obere Schranke, desto effektiver kann man Prunen. Man kann einfach schlechte obere Schranken bestimmen, wie zum Beispiel durch heuristische Methoden oder der Gesamtsumme aller Featurekosten in Feature-basierten Problemen.
Wenn die Baumsuche eine bessere Lösung findet, wird die obere Schranke aktualisiert um ineffiziente

Teilbäume frühzeitig zu eliminieren und die Qualität der oberen Schranke im Laufe der Suche zu verbessern.

2.1.2 Relaxierungen

Relaxierungen erlauben schwierige Optimierungsprobleme durch einfachere Probleme zu ersetzen, deren optimale Lösung eine untere Schranke für das ursprüngliche Problem liefert. Das geschieht durch erweitern der Menge der zulässigen Lösungen und Ersetzung der Zielfunktion durch Funktion mit kleineren oder gleichen Werten.

2.1.3 Definition der Relaxierung

Gegeben sei ein Optimierungsproblem: $opt^O = \min\{c(x)|x \in X \subseteq \mathbb{R}^n\}$.
Eine Relaxierung des Problems ist definiert als: $opt^R = \min\{f(x)|x \in T \subseteq \mathbb{R}^n\}$.
Eine Relaxierung ist gültig, wenn:

- die Menge der zulässigen Lösungen im Relaxierten Problem nicht kleiner ist, als die ursprüngliche Menge.
- Zielfunktion nicht größer als die des ursprünglichen Problems ist.

2.1.4 Eigenschaften von Relaxierungen

- Relaxierungen liefern untere Schranke
- Wenn Relaxiertes Problem unlösbar ist, ist auch das ursprüngliche Problem unlösbar.
- Wenn eine optimale Lösung des relaxierten Problems auch eine Lösung des ursprünglichen Problems ist, dann ist diese Lösung auch optimal.

2.1.5 LP-Relaxation

Ein Integer Linear Programming Problem ist schwer zu lösen. Man kann die Bedingung, dass Lösungen ganzzahlig sein müssen, relaxieren. Diese sind einfacher und schneller zu berechnen. Dies liefert eine untere Schranke für das ursprüngliche ILP.

2.1.6 Kombinatorische Relaxation

Eine Relaxation, bei der ein schwieriges kombinatorisches Problem vereinfacht wird.
Wie zum Beispile beim **Symmetrischen Travelling Salesman Problem**. Man lässt einige Einschränken weg, zum Beispiel, dass keine Teilrouten entstehen dürfen. Das erzeugt ein einfaches

2-Faktor Problem, das mit effizienten Algorithmen gelöst werden kann.

1-Tree Relaxation des Symmetrischen Travelling Salesman Problems
Das ist eine spezielle Relaxation. Ein **1-Tree** ist ein Graph, bei dem ein Knoten zwei angrenzende Kanten hat, während die restlichen Knoten ein Spanning Tree sind. Das ist nützlich, weil jede Tour ein 1-Tree ist. Dies liefert eine gültige untere Schranke. Man berechnet hierfür ein Spannbaum für alle Knoten außer einem, und fügt zwei günstige Kanten zum ignorierten Knoten hinzu.

2.1.7 Lagrange Relaxation

Man betrachtet ein Optimierungsproblem, bei dem eine Zielfunktion minimiert werden soll. Es gibt Bedingungen, die eingehalten werden müssen. Bei der Lagrange Relaxation, werden diese Bedingungen nicht mehr hart vorgeschrieben, sondern in die Zielfunktion mit Lagrange-Multiplikationen aufgenommen. Dadurch entsteht eine neue Zielfunktion, die leichter zu lösen ist.
Die Nebenbedingungen werden durch Strafterme ersetzt. Jede Wahl der Lagrange-Multiplikatoren führt zu einer neuen Zielfunktion. Der Wert dieser neuen Funktion ist immer eine untere Schranke für das ursrpüngliche Problem.

Beweisidee

- ursprünglich müssten alle Lösungen die Bedingung erfüllen
- fügt man hierfür ein Strafterm in die Zielfunktion hinzu, wird die Lösung nicht schlechter, sondern bleibt kleiner oder gleich der ursprünglichen Lösung.
- Daher liefert dies eine untere Schranke.

Beim Lagrange Relaxation werden Probleme einfacher gemacht. Dazu sucht man das beste Set an Lagrange-Multiplikatoren um die Lücke zwischen Relaxation und dem Problem zu verkleinern. Hierdurch hat man ein neues Optimierungsproblem: finde das Maximum der Lagrange-Funktion.

2.1.8 Anwendung Lagrange Relaxation auf Symmetrical TSP

Die Nebenbedingungen sind, dass jede Stadt genau zweimal besucht wird. Die Lagrange-Relaxierung ist dass die Bedingung durch Multiplikatoren ersetzt wird. Das Problem wird auf dadurch auf ein 1-Tree Problem reduziert. Anstatt eine optimale Tour zu finden, sucht man einen Minimalem-1-Tree mit geupdaten Kosten.

2.1.9 Relaxationen in Branch and Bound

Sie helfen Optimierungsprobleme zu vereinfachen und schneller Lösungen zu finden. Problematisch ist jedoch, dass nicht jeder Knoten im Baum eine gültige Instanz ist. Wie zB beim TSP. Man

muss für jede Instanz die Relaxation anpassen. Man soll Relaxationen so wählen, dass die neuen Teilprobleme effizient gelöst werden können.

2.1.10 Branch and Bound für Integer Linear Programming

Sind praktisch sehr wichtig, weil viele reale Optimierungsprobleme so formulieren lassen. Wenn die Lösung (Knoten) der LP-Relax ganzzahlig ist, hat man eine optimale Lösung. Falls es nicht ganzzahlig ist, müssen wir branchen.

2.1.11 Branching Strategien für ILP

- Maximum Infeasibility Rule: Variable wählen, die am weitesten von einer Integer Lösung ist (Große Änderung früh kann schneller zu Lösung führen)
- Minimum Infeasibility Rule: Variable, die am nächstgelegenen zu einer Integer Lösung ist (Kann schneller zu einer ersten gültigen Lösung führen, langsamer)
- Strong Branching: Testet vorab verschiedene Branches um beste Richtung zu finden. (Gut für große Probleme, aber rechenintensiv)

2.2 Constraint Satisfaction Problems

Besteht aus Variablen, eine Menge an möglichen Werten pro Variable und eine Menge an Constraints. Eine Lösung ist eine Variablebelegung, die die Constraints erfüllen.

2.2.1 Backtracking

Lösungstechnik für CSP. Assigne Variablenwerte bis Constraint verletzt. Dann Backtrack. (Tiefensuche).

Probleme

- Es werden keine Problemquellen ermittelt. (Kann durch Backjumping gelöst werden)
- Inkonsistente Werte sind nicht gespeichert, heißt man checkt gleiche Werte mehrmals. (Kann durch Backchecking gelöst werden)
- Backtracking erkennt Konflikte wenn sie passieren, nicht davor. (Kann durch Forward Check gelöst werden)

Backjumping springt zum frühsten Knoten, wo der Konflikt resolved wird.

2.2.2 Konsistenzen

- Knoten-Consistency: Variablenwert erfüllt alle unären Constraints
- Arc-Consistency: Variable ist arc-konsistent, bzgl. einer anderen wenn für jeden Wert dieser Variable ein kompatibler Wert in anderen existiert.

2.2.3 Wie achieved man Arc Consistency?

Durch Algorithmen wie AC-3. Entfernt inkonsistente Werte aus Domäne.

2.2.4 Reicht Arc-Consistency aus?

Nein, weil selbst wenn alle Variablen paarweise konsistent sind, es keine Garantie gibt, dass eine Lösung für das gesamte Problem existiert. Man kann k-Konsistenz machen, aber je höher k desto mehr Rechenpower. Und Backtracking kann nie vermieden werden.

2.2.5 Look Ahead Strategies (Constraint Propagation)

Reduziert Suchraum, indem Constraintkonflikte frühzeitig erkannt werden, um Domäne zu verringern. Bei Full-Look Ahead, wird nach jeder Variablenzuweisung Arc Konsistenz für alle Variablen gecheckt wird.

2.2.6 Welche Variable nimmt man?

Es gibt verschiedene Heuristiken.

- Minimum Remaining Values: Welche Variable hat wenigsten Werte?
- Degree: Welche Variable hat die meisten Constraints?

2.3 Dynammic Programming

DP ist eine Methode zur exakten optimierung, bei der Ein Problem durch Kombination der Lösungen von Teilproblemen gelöst wird. Man speichert hierbei die Zwischenlösungen in einer Tabelle.
Divide and Conquer berechnet Teilprobleme mehrfach.

2.3.1 Wann kann man Dynamisches Programmieren verwenden?

Ein Problem muss zwei Eigenschaften haben.

- Eine optimale Lösung enthält optimale Lösungen von Teilproblemen
- gleichen Teilprobleme werden mehrfach benötigt.

Beispiel: Fibonacci-Zahlen, kürzester Pfad in Graphen oder Matrix Chain Multiplikcation.

2.3.2 Matrix-Kettenmultiplikation Problem

Zu bestimmen ist die Produktreihenfolge mehrerer Matrizen, um die Anzahl der Skalarmultiplikationen zu minimieren. Die Multiplikationsreihenfolge beeinflusst die Berechnungszeit sehr.

2.3.3 Finden der optimalen Lösung mit dynamischen Programmieren

Das Problem muss aufgeteilt werden, sodass die optimale Reihenfolge gefunden wird.

1. Struktur der optimalen Lösung
   Eine optimale Klammerung der Matrizen teilt das Problem an einer bestimmten Stelle k in zwei Teile. Diese Teilprobleme muss man auch klammern
2. rekursive Definition
   Definiere $m[i, j]$ als minimale ANzahl von Multiplikationen für die Matrizen A_i bis A_j.
   Falls nur eine Matrix vorhanden ist, dann sind keine Multiplikationen nötig, also 0.
   Falls mehrere Matrizen vorhanden sind, dann wähle eine Teilung k und berechne die Kosten für die Teilbereiche.
3. Bottom-Up berechnung
   Statt Rekursion immer wieder zu berechnen, nutzen wir eine Tabelle. Basisfälle werden auf 0 gesetzt. Für alle möglichen Intervalle i, j wird $m[i, j]$ berechnet. Die beste Trennung wird für jede Kombination gespeichert.
4. Rekonstruktion
   Mit der gespeicherten Tabelle können wir die beste Klammerung schrittweise konstruieren.
   Wenn die Matrizen A_i bis A_j berechnet werden sollen, teile Sie bei $s[i, j]$ in zwei Gruppen.
   Wende das gleiche Verfahren rekursiv auf beide Gruppen an.
   Dies ergibt die beste Klammerung.

2.3.4 Sequence Alignment

Gegeben sein zwei Zeichenketten. Diese müssen verglichen werden, wobei bestimmte Mutationen erlaubt sind. Das Ziel ist es die beste Übereinstimmung zwischen zwei Sequenzen zu finden, indem

Einfügungen, Löschungen und Ersetzungen erlaubt sind. (Anwendung in der Bio-Informatik).

2.3.5 Definition Alignment

Ein Alignment besteht aus zwei Zeichenketten der gleichen Länge, die aus dem ursprünglichen Alphabet und einem Lücken-Symbol bestehen. Beide Sequenzen sind gleich lang, und keine Position darf auf zwei Lücken treffen. Durch Entfernen der Lücken soll die ursprüngliche Sequenz rekonstruiert werden können.

2.3.6 Terminologie

- Match: gleiche Zeichen
- Substitution: Ein Zeichen wurde durch ein anderes ersetzt
- Insertion: ein Zeichen wurde in einer der Sequenzen hinzugefügt
- Deletion: ein Zeichen wurde entfernt

Jedes Alignment hat eine Kostenfunktion, die bestimmt wie teuer eine Übereinstimmung oder Änderung ist. Match = 0, Insertion=Deletion=2, Substitution = 3. Die Alignment-Distanz ist die minimale Summe der Kosten für ein gültiges Alignment.

2.3.7 Globales Sequenzen Alignment

Zwei Seuqnzen sollen optimal ausgerichtet werden, um die minimale Alignment-Distanz zu bestimmen. Hierfür verwenden wir dynamische Programmierung. Erser Algorithmus: $\textbf{Needleman}_{W}unsch-Alg$

2.3.8 Dynamische Programmierung für Sequence Alignment

1. **Ersttelle Tabelle D der Größe $(n+1) \times (m+1)$, wobe n, m Länge der beiden Sequenzen sind.**
2. **Rekursion:**

$$D(i,j) = \min \begin{cases} D(i-1,j) + \text{Kosten für Einfügen} \\ D(i-1,j-1) + \text{Kosten für Ersetzen} \\ D(i,j-1) + \text{Kosten für Löschen} \end{cases}$$

3. **Tablle von links oben nach rechts unten führen**
4. **optimale Distanz ist $D(n,m)$**

2.3.9 Edit Graph und kürzester Pfad

Die Rekursion kann als ein kürzester Pfad in einem gerichteten azyklischen Graphen gesehen werden. Jeder Punkt in der Tabelle (i, j) ist ein Knoten.
Folgende Übergänge sind möglich:

- **Von Oben (Insertion)**
- **Von Links (Deletion)**
- **Diagonal (Substitution/Match)**

Der kürzeste Pfad von $(0, 0)$ nach (n, m) gibt das optimale Alignment.

2.3.10 Integer Knapsack

Warum ist das Integer Knapsack Problem NP-hart?
Weil es keine effiziente Methode gibt, um optimal alle möglichen Kombinationen von Gegenständen in polynomialer Zeit zu überprüfen. Viele Kombinationen müssen ausprobiert werden, was exponentiellen Aufwand verursachen kann. Es kann auf das Subset Sum Problem reduziert werden, was ebenfalls NP-hart ist.

2.3.11 Integer Knapsack mit DP lösen - Algo 1

Eine rekursive Lösung kann wie folgt definiert werden:

- **Sei $g_r(\lambda)$ die maximale Wertfunktion, die für ein bestimmtes Budget λ und die ersten r Gegenstände möglich ist.**
- **Rekursive Formel:**
  $$g_r(\lambda) = \max\{g_{r-1}(\lambda), c_r + g_r(\lambda - a_r\}$$
  Entweder nehmen wir den Gegenstand r nicht. Oder wir nehmen diesen, und der Wert der Gesamtkosten erhöhrt sich, aber das verbleibende Budget reduziert sich

Dies führt zu einem Algorithmus mit Laufzeit $O(n * B)$ was pseudopoynomial ist.

2.3.12 Integer Knapsack als längster Pfad

Das Problem kann als längster Pfad in einem gerichteten azyklischen Graphen formuliert werden. Jeder Knoten repräsentiert ein Budget λ, und die Kanten repräsentieren

den Übergang, wenn ein Gegenstand hinzugefügt wird. Die Lösung entspricht dem längsten Pfad von 0 nach B, wobei die Kantengewichte den Gegenstandwerten entsprechen. Dies führt auch zu $O(n * B)$ was pseudopolynomial ist.
Weil B in binärer Darstellung $O(\log B)$ Speicher braucht, ist es nur pseudopolynomial.

Crossover

Graphische Datenverarbeitung I
WiSe 2024/25

1 Definition

Crossover ist die Rekombination von Genen um neue Gene zu erzeugen.

1.1 Arten von Crossovers

- One-point Crossover: Eine Position wird durch eine beliebige Auswahlstrategie ausgesucht, und an der Position wird das Gen getauscht mit einem anderen.

$$ABCD||EFGH$$

$$ABGD||EDCH$$

- Two-point Crossover: Zwei Punkte werden ausgesucht, und in diesem Intervall werden die Gene getauscht

$$ABCDE||FGHIJ$$

$$AGHIE||FBCDJ$$

- Uniform crossover: Es wird mit probabilistischen Entscheidungen entschieden, wann ein Genteil getauscht wird.

1.2 Problematik von Crossovers

Die Lösung ist zwar eine abstrakte Repräsentation, aber möglich von einer unzulässigen Lösung. Dies kann man verhindern, indem man alle Lösungen zulässig macht indem Constraints verfallen werden, oder man Bestrafungen einführt. Man kann alternativ auch die genetische Repräsentation von Lösungen so definieren, dass fast zulässige Lösungen von Crossovern generiert werden. Also führt man das Crossover durch und repariert die Lösung daraufhin.

1.3 Crossover und Repair: Beispiel PMX für TSP

PMS ist partially mapped crossover. Beim PMX wird ein Two-Point Crossover durchgeführt. Dadurch entsteht eine abstrakte Repräsentation von einer TSP Lösung, in der Städte doppelt vorkommen können.

Man permutiert zunächst das gewählte Intervall. Dann füllt man den Rest auf. Für O_1 Position 1, versuchen wir das Chromosom von Position 1 in P_1 einzutragen. Wenn dies jedoch durch das

Crossover nun verdoppelt ist, schauen wir an der Stelle der doppelung i im zweiten Offspring nach also nehmen wir $O_2[i]$ und versuchen dies einzusetzen. Wenn dies auch doppelt ist, machen wir weiter, bis das Offspring gelöst ist.

2 Beweis der Konvergenz des Repair Loops

2.1 Graphmodell und Zyklen

Man betrachtet einen gerichteten Graphen $G = (V, A)$ wobei:

- V die Menge der Städte ist
- $(i, j) \in A$ existiert, wenn Stadt i aus P_2 durch Stadt j aus P_1 ersetzt wurde.

Zyklusentstehung: Wenn eine Stadt mehrfach ersetzt werden muss, entsteht ein Pfad im Graphen. Aber in dieser Problemdefinition, würde es bedeuten, dass eine Stadt verdoppelt war, bevor das Crossover durchgeführt wurde.

2.2 Eigenschaften des Graphen

- jeder Knoten hat maximal eine eingehende Kante
- Pfade entstehen durch rekursive Mapping-Schritte
- Zyklen entstehen nur, wenn eine Stadt sich selbst ersetzt

2.3 Warum terminiert der Loop?

Jede Iteration des Repair Loops folgt einem Pfad im Graphen. Da kein Knoten mehrfach betreten wird, sind Zyklen ausgeschlossen. Der Repair Loop terminiert, wenn alle Doppelungen aufgelöst sind. Denn jeder Knoten in einem Zyklus kann nicht von anderen Knoten erreicht werden, die außerhalb des Zyklus sind. Da kein Zyklus betreten wird, terminiert der Loop immer

Approximationen

Optimierungsalgorithmen
WiSe 2024/25

1 Definitionen

1.1 Was bedeutet Approximativ?

"Approximativ" ist die Eigenschaft eines Algorithmus. Ein Algorithmus heißt 'approximativ', wenn es eine Lösung findet, bei der bewiesen werden kann, dass diese nicht weit von einer zulässigen Lösung ist. Bei Optimierungsalgorithmen, sind es dann Lösungen die nicht so weit weg von einer optimalen Lösung sind. Die Distanz zwischen diesen Lösungen wird durch ein vernünftiges Maß determiniert, welche je nach Problemstellung anders ist.

1.2 Absolute Approximationen

Ein polynomialzeit Algorithmus A für ein Optimierungsproblem P ist ein *Algorithmus zur absoluten Approximation*, wenn eine Konstante k existiert, sodass für alle Instanzen I von P gilt:

$$|\mathrm{APP}(I) - \mathrm{OPT}(I)| \leq k$$

Algorithmen zur absoluten Approximation sind nur für wenige klassische $NP-$harten Optimierungsprobleme bekannt.

1.3 k-Faktor Approximation

Sei P ein Optimierungsproblem und $k \geq 1$ eine Konstante.
Ein k-faktor Approximationsalgorithmus für P ist ein polynomialzeit Algorithmus A für P, sodass für alle Instanzen I von P gilt:

$$\frac{1}{k} * \mathrm{Opt}(I) \leq \mathrm{APP}(I) \leq k * \mathrm{OPT}(I)$$

ein k-Faktor Approximationsalgorithmus liefert also in polynomialzeit eine Lösung zu einem Optimierungsproblem, welche höchstens um den Faktor k von der optimalen Lösung abweicht.

2 Beweis: Greedy Algorithmus liefert Faktor 2 Approximation für das Maximum Weighted Matching Problem

2.1 Problemdefinition: Maximum weighted Matching

Gegeben sei ein gewichteter, ungerichteter Graph $G = (V, E)$ mit Gewichtsfunktion $w : E \to \mathbb{R}^+$, die jedem Kantenpaar ein nicht-negatives Gewicht zuweist.
Ein Matching $M \subseteq E$ ist eine Menge von Kanten, sodass keine zwei Kanten einen gemeinsamen Knoten haben. Das Ziel des Problems ist es, ein Matching M^* mit **maximalem Gesamtgewicht** zu finden.

$$w(M^*) = \sum_{e \in M^*} w(e)$$

2.2 Beweis

Sei M_{opt} ein maximales gewichtetes Matching und M_{app} das vom Greedy-Algorithmus erzeugte Matching. Wir betrachten für den Beweis die symmetrische Differenz $M_{\text{app}} \triangle M_{\text{opt}}$. In dieser Differenz existieren keine isolierten Kanten.
Eine isolierte Kante ist eine Kante im Graphen, die nicht mit einer anderen Kante verbunden ist.
Jede Kante $e \in M_{\text{opt}}$ ist benachbart zu höchstens zwei Kanten e' und e" aus M_{app} innerhalb der Differenz. Falls $w(e) > w(e')$ und $w(e) > w(e$") gelten würde, hätte der Greedy Algorithmus stattdessen die Kante e gewählt. Daher muss eines der beiden Ungleichungen zutreffen:

$$w(e') \geq w(e) \text{ oder } w(e") \geq w(e)$$

Das heißt, dass das Gewicht von e einer angrenzenden Kante aus M_{app} zugeordnet werden kann, dessen Gewicht **mindestens gleich** dem von e ist.
Da jede Kante in der Approximation höchstens zweimal belastet wird, folgt:

$$w(M_{\text{opt}}) \leq 2 * w(M_{\text{app}})$$

$\Rightarrow$ somit ist der Greedy Algorithmus eine 2-Approximation für das Maximum Weighted Matching Problem.

3 Beweis: 2-Approximation für Vertex Cover

3.1 Problemdefinition: Vertex Cover

Gegeben sei ein ungerichteter Graph $G = (V, E)$. Das Ziel ist es eine minimale Menge von Knoten $C \subseteq V$ zu finden, sodass jede Kante $(u, v) \in E$ mindestens einen Endpunkt in C hat. Das heißt jeder Kante im Graphen muss mindestens ein Knoten aus C zugeordnet werden.
Dieses Problem ist NP-hart.

3.2 Betrachteter Algorithmus

Um das Vertex Cover Problem zu lösen wird folgender Algorithmus betrachtet.

1. Berechnung eines maximalen Matchings M
   - Es wird ein maximales Matching berechnet, bei dem keine weiteren Kanten mehr hinzugefügt werden können, ohne einen bestehenden Knoten doppelt zu verbinden.
2. Vertex-Cover Lösung erstellen
   - wähle alle Endpunkte der Kanten aus M als Vertex Cover.
   - Für jede Kante $(u, v) \in M$ wird u und v in die Lösung aufgenommen.

3.3 Beweis: Matching2VertexCover ist eine 2-Approximation

Der Algorithmus liefert eine Lösung, die höchstens doppelt so groß ist wie das optimale Vertex Cover.

1. Die Lösung ist ein gültiges Vertex Cover
   - weil jede Kante in M genau einen Endpunkt in der Lösung hat und M maximal ist, muss jede Kante in G abgedeckt sein.
2. jede optimale Lösung muss mindestens $|M|$ Knoten enthalten:
   - im optimalen Vertex Cover muss mindestens ein Knoten pro Kante von M enthalten sein
   - weil M ein maximales Matching ist, kann keine Vertex-Cover-Lösung weniger als $|M|$ Knoten enthalten.
   - daraus folgt:
     $$|M| \leq OPT(I)$$
     wobei $OPT(I)$ die Größer der Optimalen Lösung ist
3. der Algorithmus wählt $2|M|$ Knoten:
   - Jede Kante in M trägt zwei Knoten zur Lösung bei, also gilt: $APP(I) = 2|M|$
   - Da $|M| \leq OPT(I)$ gilt, folgt:
     $$APP(I) = 2|M| \leq 2 * OPT(I)$$

$\Rightarrow$ der Algorithmus liefert eine 2-Approximation

4 Beweis: Approximationsalgorithmus für das Steiner Tree Problem

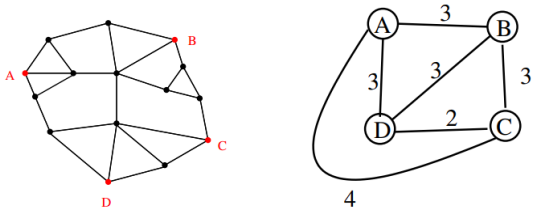
4.1 Problemdefinition: Steiner Tree Problem

Gegeben ist ein eingerichteter Graph $G = (V, E)$ mit Kantenkosten w und einer Menge von Terminalknoten $K \subseteq V$. Das Ziel ist es einen Teilgraphen von G mit minimalen Gesamtkosten zu finden, der alle Terminalknoten aus K verbindet. Dabei dürfen auch zusätzliche Steiner-Knoten (also Knoten, die nicht Terminalknoten sind), verwendet werden, wenn sie helfen, eine kostengünstige Verbindung herzustellen.

4.2 Betrachteter Approximationsalgorithmus

Der Approximationsalgorithmus besteht aus drei grundlegenden Schritten.

1. Distanznerzwerk N_d konstruieren
   - Man berechnet ein Distanznetzwerk, welches nur die Terminalknoten enthält



   - Das Distanznetzwerk ist ein vollständiger Graph auf den Terminalknoten, indem die Kostender Kanten die Distanz zwischen zwei Knoten auf dem Originalgraph darstellen
2. Minimalen Spannbaum auf Distanznetwerk berechnen
3. In Steiner-Baum transformieren
   - Zuerst ersetzt man die Kanten des Minimal Spanning Trees mit den korrespondierenden Pfaden des Originalgraphen
   - Wenn der Pfad ein Zyklus beinhaltet, entfernt man die längste Kante von jedem Zyklus
   - entferne Blätter des Baums, die keine Terminal sind

4.3 Beweis: Algorithmus liefert 2-Approximation

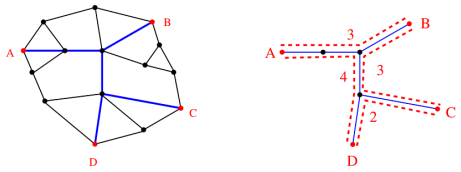
Der Algorithmus garantiert, dass die berechnete Steiner-Lösung höchstens doppelt so lang ist wie die optimale Steiner-Lösung. Die Idee des Beweises basiert auf folgende Überlegungen:

1. Jeder Steiner-Baum enthält einen Spannbaum in N_d

2. MST auf N_d ist eine Approximation für diesen Spannbaum
    - Kanten in N_d sind kürzeste Wege im Originalgraphen, also kann der MST von N_d höchstens doppelt so teuer sein, wie der optimale Steiner Baum
    - Es gilt also: $w(T') \leq 2 * w(OPT)$, wobei T' der MST in N_d ist.
3. der optimale Steiner-Baum ist **höchstens so teuer wie ein MST in** N_d

Also gilt das Lemma. Das liegt daran, dass der optimale Steiner-Baum eine Unterstruktur enthält, die bereits einen Spannbaum bildet. Der MST im Distanznetzwerk nutzt kürzeste Wege und ist daher verdoppelt. Die Transformation in den finalen Steiner-Tree erzeugt keine zusätzliche Kostenverdopplung, wodurch der Faktor 2 bestehen bleibt.

4.3.1 Beispiel von Vorlesung



Spanning tree with edges *A-B*, *B-C* and *C-D* in N_d has this property.

Wir betrachten eine optimale Lösung. Und wir zählen hierbei jede Kante höchstens 2 mal, wodurch eine maximale Verdoppelung von 2 zum Optimum entsteht.
Wir generieren ja einen Minimalen Spannbaum. Dieser ist höchstens besser, aufgrund der Minimierung. Wenn wir diesen dann Transformieren im Algorithmus zu einem Steiner Tree, dann kann sich dieser auch nur verbessern, weil wir dort höchstens Kanten entfernen.

5 Metric TSP - NP-Hard

5.1 Problemdefinition

Gegeben sei ein kompletter Graph $G = (V, E)$ mit den Gewichten $c : E \to R_+$ sodass für alle $x, y, z \in V$ gilt:

$$c(x, y) + c(y, z) \leq c(x, z)$$

Das Ziel ist es ein Hamilton-Kreis zu finden mit minimalen Gewichten. Also eine minimale Rundreise.

5.2 Beweis: das Metric TSP ist NP-hart

Der Beweis erfolgt durch eine Reduktion vom Hamiltonian Circuit Problem, ähnlich wie beim general TSP.

5.2.1 Hamiltonian Circuit Problem

Gegeben sei ein ungerichteter Graph. Gibt es ein Zyklus, der jeden Knoten genau einmal besucht?

5.2.2 Hamiltonian Circuit Problem auf Metric TSP reduzieren

Aus dem ungerichteten Graphen konstruieren wir ein vollständigen Graphen G', indem für alle nicht vorhandenen Kanten eine sehr große Distanz gesetzt wird. Dieser neue Graph erfüllt die Dreiecksungleichung. Ein optimaler TSP-Zyklus mit Gesamtkosten $|V|$ existiert genau dann, wenn ein Hamiltonkreis in G existiert.

5.2.3 Schlussfolgerung

Weil das Hamltonian-Circuit Problem NP-Vollständig ist, folgt, dass Metric TSP NP-hart ist, weil eine Lösung für das Metric TSP eine Lösung für das Hamiltonian Circuit Problem liefern kann. Das Metric TSP bleibt in der Klasse NP-hart statt vollständig, weil nur die Kostenoptimierung das eigentliche schwierige Problem ist, nicht nur die Entscheidung ob ein Hamiltonzyklus existiert.

6 Double-Tree Algorithmus: Metric TSP

6.1 Algorithmusdefinition

Der Double-Tree Algorithmus besteht aus 4 Schritte.

1. finde minimalen Spannenden Baum T in G
2. erstelle Multigraph T' indem zwei Kopien von jeder Kante verwendet werden
3. FInde einen Eulerischen Pfad in T', der jede Kante einmal besucht.
4. Transformiere den Pfad in eine Rundtour, indem Abkürzungen verwendet werden.

6.2 Beweis: Double-Tree Algorithmus ist eine 2-Faktor Approximation für das Metric TSP

Die Länge des Minimalen Spannbaums $c(E(T))$ ist definitiv kleiner als die Länge der Optimalen Lösung, weil das Entfernen einer Kante von jeder beliebiger Tour ein Spannbaum erzeugt.
Weil wir alle Kanten verdoppeln gilt: $c(E(T')) = 2 * c(E(T)) \leq 2 * OPT$.
Im vierten Algorithmus Schritt, wird ein Eulerischer Pfad der Länge $c(E(T'))$ in eine Tour umgewandelt. Diese Tour ist definiert durch die Reihenfolge in der die Knoten auftauchen. Wir betrachten hierbei nur die ersten Vorkommen der Knoten. Die Dreiecksungleichung impliziert, dass diese Tour nicht länger ist als $c(E(T'))$, weil Abkürzungen definitiv nichts erhöhen.

7 Christofides Algorithmus

7.1 Algorithmus beschreibung

1. Finde einen minimalen Spannbaum T in G.
2. Seien W die Knoten in T die ein ungeraden Grad haben. Finde ein Minimum Weight Perfect Matching M im kompletten Graph der aus nur Knoten besteht
3. erstelle ein Multigraph $T' = (V, E(T) \cup M)$ der aus den Kanten von T und M aufgebaut ist
4. finde einen Eulerpfad in T'
5. Transformiere Eulerpfad in eine Tour durch Abkürzungen

7.2 Beweis: Christofides Algorithmus liefert eine $\frac{3}{2}$ Approximation für das Metric TSP

Wir wissen zunächst, dass die Länge der berechneten Tour nicht länger als der Eulerweg T' sein kann. Der MST T fungiert als untere Schranke für die optimale Lösung wie zuvor. Das Ziel ist es zu zeigen, dass die Kosten des Matchings, kleiner gleich sind als der Hälfte des Optimums.
Hierfür konstruiert man zwei perfekte Matchings M_1 und M_2. Die Dreiecksungleichung garantiert, dass $OPT \geq c(M_1) + c(M_2)$.
Das minimale perfekte Matching M kann nicht länger sein als $c(M_1) + c(M_2)$. Das bedeutet aus vorherigem Schritt es kann nicht länger sein als Optimum.

$$OPT \geq 2 * c(M)$$

Wir teilen nun durch zwei, und tauschen so die Ungleichung:

$$c(M) \leq \frac{1}{2} * OPT$$

Mit dieser Information können wir folgende ungleichung darstellen:

$$c(E(T')) = c(E(T)) + c(M) \leq OPT + \frac{1}{2} * OPT = \frac{3}{2} * OPT$$

$\Rightarrow$ Das MST gibt eine untere Schranke für das Optimum, weil jede optimale TSP-Tour eine Teilstruktur enthält, die mindestens so teuer ist wie der MST. Das minimale Perfekte Matching hat Kosten von höchstens der Hälfte des Optimums, weil zwei verschiedene perfekte Matchings aus der optimalen Tour ableiten können. Die Gesamtkosten des Algorithmus setzen sich aus den MST-Kosten und den Matching-Kosten zusammen.

Unabhängigkeitssysteme und Matroiden

Optimierungsalgorithmen
WiSe 2024/25

1 Definitionen

Sei E eine endliche Menge, und $F \subseteq 2^E$.
Ein Mengensystem (E, F) ist ein Unabhängigkeitssystem, wenn gilt:

- $\in F$
- wenn $X \subseteq Y$ und $Y \in F$, dann gilt auch $X \in F$

Die Elemente von F sind **unabhängig**. Alle Untermengen von E die **nicht** in F sind, heißen **abhängig**.
Eine Basis ist das Maximum an unabhängigen Mengen.
Die Menge F ist üblicherweise nicht durch ihre Elemente gegeben. Wir nehmen an ein Orakel zu haben, welches entscheidet ob eine Menge F $\subset E$ Element von F ist.

2 Maximierungs- und Minimierungsprobleme für Unabhängigkeitssysteme

2.1 Maximierungsproblem

Gegeben sei ein Unabhängigkeitssystem (E, F) und eine Kostenfunktion $c : E \to \mathbb{R}$.
Das Ziel ist es ein $X \in F$ zu finden, sodass gilt: $c(X) = \sum_{e \in X} c(e)$ ist Maximum

2.2 Minimierungsproblem

Gegeben sei ein Unabhängigkeitssystem (E, F) und eine Kostenfunktion $c : E \to \mathbb{R}$.
Das Ziel ist es eine Basis $B \in F$ zu finden, sodass gilt: $c(B) = \sum_{e \in B} c(e)$ ist Minimum

3 Optimierungsprobleme für Unabhängigkeitssysteme

3.1 Maximum Weight Stable Set Problem

Gegeben sei ein Graph G mit einer Menge an Knoten $V(G)$ und Gewichte $c : V \to \mathbb{R}$. Finde eine stabile Menge X in G mit maximalem Gewicht.
Hierbei sind $E = V(G)$ und $F = \{F \subseteq E : F \text{ ist stabil in } G\}$

Eine stabile Menge ist eine Menge von Knoten des Graphen, wobei gilt, dass keine zwei Knoten der Menge benachbart sind.

3.2 TSP

Gegeben sei ein kompletter, ungerichteter Graph G und Gewichte $c : E(G) \to \mathbb{R}_+$. Finde ein minimal gewichteten Hamiltonzyklus in G.
Hierbei sind $E = E(G)$ und $F = \{F \subseteq E : F$ ist eine Untermenge eines Hamiltonzyklus in $G\}$.

3.3 Shortest Path Problem

Given a directed Graph $D = (V, A), c : A \to \mathbb{R}$ und $s, t \in V$, sodass t von s aus erreichbar ist.
Finde den kürzesten Pfad $S - T$ in D in Bezug auf c.
Hierbei gilt: $E = A$ und $F = \{F \subseteq E : F$ ist eine Untermenge des Pfades $S - T\}$

3.4 0-1-Knapsack-Problem

Gegeben sind n Objekte von jedes ein Gewicht a_i, und einen Wert c_i hat. Außerdem gibt es eine Kapazität B (Fassungsvermögen vom Rucksack).
Gesucht ist eine Teilmenge $S \subseteq \{1, ..., n\}$ von Objekten, sodass:

- die Summe der Gewichte der gewählten Objekte die Kapazitätsgrenze B **nicht** überschreitet
- der Gesamtwert der gewählten Objekte maximal ist.

Bei einem Unabhängigkeitssystem ist die Menge der betrachteten Objekte E. Und die Menge $F \subseteq E$ ist zulässig, wenn die Summe der Gewichte der gewählten Objekte nicht größer ist als B.

3.5 Minimum Spanning Tree

Gegeben sei ein zusammenhängender, ungerichteter Graph G mit einer Gewichtsfunktion $c : E(G) \to \mathbb{R}$. Das Ziel ist es einen Spannbaum zu finden mit minimalen Gesamtgewicht.
Die Grundmenge des Unabhängigkeitssystems ist die Menge aller Kanten des Graphen. Die zulässigen Mengen F sind alle Mengen von Kanten, die keine Zyklen enthalten (Wälder).
Das System ist abwärts geschlossen. Das bedeutet, dass alle Teilmengen eines Waldes auch Wälder sind.

3.6 Maximum Weight Matching Problem

Gegeben sei ein ungerichteter Graph G mit einer Gewichtsfunktion, die jeder Kante ein Gewicht zuweist. Ziel ist es ein Matching mit maximalem Gesamtgewicht zu finden.

Die Grundmenge des Unabhängigkeitssystems ist die Menge aller Kanten des Graphen. Die zulässigen Mengen F, sind Mengen von kanten, in denen kein Knoten mehrfach vorkommt.
Das System ist abwärts geschlossen, aber kein Matroid.

4 Best-In-Greedy Algorithmus

Wir betrachten das Maximierungsproblem für Unabhängigkeitssysteme.
Der Input sei ein Unabhängigkeitssystem (E, F). F ist gegeben durch ein Unabhängigkeitsorakel, welches entscheidet ob es enthalten ist oder nicht, und einer Gewichtsfunktion für die Kanten.
Der Output ist eine Menge $f \in F$.

1. Sortiere die E, sodass das "teuerste" Element am Anfang ist. $c(e_1) \geq c(e_2) \geq ... \geq c(e_n)$
2. Setze F=
3. Für $i = 1$ bis n Do:
   Wenn F$\cup\{e_i\} \in F$ dann setze F=F$\cup\{e_i\}$

$\Rightarrow$ Negative Gewichte müssen nicht betrachtet werden, weil Elemente mit negativen Gewichten nicht in der Lösung auftauchen würden

5 Worst-Out-Greedy Algorithmus

Wir betrachten das Minimierungsproblem für Unabhängigkeitssysteme. Hierbei benötigen wir ein komplizierteres Orakel. Es Entscheidet, ob die geprüfte Menge eine Basis ist. (Basis Superset Orakel)
Der Inpute sei ein Unabhängigkeitssystem (E, F) in Bezug auf das Basis Superset Orakel, und eine Gewichtsfunktion.
Der Output ist eine Basis F von (E, F)

1. Sortiere die Menge E sodass die Elemente Absteigend sortiert sind im Bezug auf ihr Gewicht.
2. Setze F= E
3. Für $i = 1$ bis n Do:
   Wenn F$\backslash\{e_i\}$ enthält eine Basis, dann setze F=F$\backslash\{e_i\}$

6 Matroiden

Ein Unabhängigkeitssystem (E, F) ist ein Matroid, wenn gilt:
Wenn $X, Y \in F$ und $|X| > Y$, dann gibt es ein $x \in X\backslash Y$ mit $Y \cup \{x\} \in F$.

Ein Matroid ist also ein Unabhängigkeitssystem, wobei falls zwei unabhängige Mengen unterschiedliche Größe haben, kann ein Element von der größeren zur kleineren hinzugefügt werden, sodass sie unabhängig bleibt.

6.1 Matric Matroid

Die Grundmenge E ist die Menge der Spalten einer Matrix A über einem Körper. Die zulässige Mengen F sind die zugehörigen Spalten in A die linear unabhängig sind.

6.2 Graphic Matroid

Die Grundmenge E ist die Menge der Kanten eines ungerichteten Graphen $G = (V, E)$. Eine Menge ist zulässig, wenn der Teilgraph (V, F) ein Wald ist (also zyklenfrei).

6.2.1 Austauschbarkeitseigenschaft von Graphischen Matroiden gilt

Seien $X, Y \in F$ zwei unabhängig Mengen. Man nimmt an: $Y \cup \{x\} \notin F$ für alle $x \in X \backslash Y$. Um zu zeigen, dass die Austauschbarkeitseigenschaft erfüllt ist, müssen wir zeigen, dass gilt: $|X| \leq |Y|$

Jede Kante $x = \{v, w\} \in X$ verbindet zwei Knoten v und w im gleichen Zusammenhangskomponenten von (V, Y). Das bedeutet, dass die Zusammenhangskomponenten von (V, X) Unterkomponenten derer von (V, Y) sind.
Die Anzahl p der verbundenen Komponenten der Wälder (V, X) ist größer oder gleich der Anzahl q von Verbundenen Komponenten (V, Y).
Weil $p = |V| - |X|$ und $q = |V| - |Y|$, impliziert: $|X| \leq |Y|$.

6.3 Charakterisierung von Matroiden

Sei (E, F) ist ein Unabhängigkeitssystem. Dann sind folgende Aussagen äquivalent:

1. wenn $X, Y \in F$ und $|X| > |Y|$, dann gibt es ein $x \in X \backslash Y$ mit $Y \subset \{x\} \in F$
2. wenn $X, Y \in F$ und $|X| = |Y| + 1$, dann gibt es ein $x \in X \backslash Y$ mit $Y \subset \{x\} \in F$
3. für alle $X \subseteq E$, alle Basen von X haben die selben Kardinalitäten.

6.3.1 Beweis des Charakterisierungstheorems

Es ist klar, dass 1 $\Rightarrow$ 2 gilt, weil 1 eine allgemeine Formulierung ist und 2 nur den Spezialfall behandelt mit $|X| = |Y| + 1$.
2 $\Rightarrow$ 3 bedeutet, dass alle maximalen unabhängigen Mengen (Basen) in einer Teilmenge $X \subseteq E$ die gleiche Größe haben. Wenn dies nicht der Fall wäre, gäbe es eine Basis X mit größerer Kardinalität als eine andere Basis Y. Dann würde 2 garantieren, dass wir ein Element aus X zu Y hinzufügen können, um eine größere unabhängig Menge zu erhalten. Dies ist ein Widerspruch zur Basisdefinition.
Um nun 3 $\Rightarrow$ 1 zu zeigen, nehmen wir an, dass $X, Y \in F$ und $|X| > |Y|$ gilt. Da Y eine kleinere Unabhängige Menge als X ist, kann Y keine Basis von $X \cup Y$ sein. Nach 3 müssen alle Basen gleichgroß sein. Also existiert ein $x \in X \backslash Y$, sodass $Y \cup \{x\} \in F$. Das ist Aussage. 1

6.4 Matroiden und der Greedy Algorithmus

Sei (E, F) ein Unabhängigkeitssystem.
(E, F) ist ein Matroid wenn der Best-In-Greedy Algorithm eine optimale Lösung für das Maximierungsproblem für (E, F, c) für jede Kostenfunktion $c : E \to \mathbb{R}_+$ findet.
Der Greedy Algorithmus liefert also eine optimale Lösung, wenn das gegebene Maximierungsproblem eine matroidale Struktur hat.

6.4.1 Erster Teil des Beweises: Wenn ein Matroid gegeben ist, dann ist Greedy optimal

Wir nehmen an, dass wir einen Matroiden (E, F) haben. Die Menge $F = \{e_1, ..., e_k\}$ wird durch den Greedy-Algorithmus erstellt.
Angenommen, es gäbe eine andere unabhängige Menge F' mit einem besseren Wert.
Das würde bedeuten, es muss ein Index m geben, sodass gilt: $c(e_m) < c(f_m)$. Durch die Matroid Eigenschaft, sollte es möglich sein, ein Element von F' zu F hinzuzufügen. Dies erzeugt jedoch einen Widerspruch, weil Greedy das bessere Element f_m bereits früher gewählt hätte, wenn es besser gewesen wäre. Das ist jedoch nicht der Fall.
$\Rightarrow$ daraus folgt, dass wenn (E, F) ein Matroid ist, dann findet Greedy eine bessere Lösung.

6.4.2 Zweiter Teil des Beweises: Wenn es kein Matroid ist, dann ist Greedy nicht optimal

Wir nehmen an das gegebene System ist kein Matroid. Wir müssen zeigen, dass es mindestens eine Kostenfunktion gibt, für die der Greedy Algorithmus versagt.
Konstruktion des Gegenbeispiels
Es gibt zwei unabhängige Mengen F_1 und F_2, wobei $|F_1| = p$ und $|F_2| = p + 1$.
F_1 kann nicht durch hinzufügen eines beliebigen Elements aus $F_2 \backslash F_1$ vergrößert werden (Weil kein Matroid).

Kostenfunktion definieren:

- Elemente in F_1 bekomment Kosten $p + 2$
- Elemente in $F_2 \backslash F_1$ bekommen Kosten $p + 1$
- andere Elemente haben Kosten 0

Analyse

- Wir wissen F_1 ist suboptimal, weil $c(F_2) > c(F_1)$
- Greedy wählt zuerst alle ELemente aus F_1, die hohe Kosten haben
- danach kann Greedy den Wert nicht mehr verbessern, weil weitere Elemente entweder ungültig sind oder nutzlos.
- Widerspruch: Greedy hätte F_2 wählen sollen, tat es aber nicht

$\Rightarrow$ daraus folgt, dass wenn (E, F) kein Matroid ist, kann der Greedy-Algorithmus scheitern.

6.5 Matroiden und Minimierungsprobleme

Für Matroiden sind Minimierungsprobleme äquivalent zu Maximierungsprobleme. Das Best-In-Greedy Algorithmus löst das Originalminimierungsproblem optimal. Hierbei wird dann die Elementreihenfolge von F umgedreht, weil es ein Minimierungsproblem ist.

Wir wissen, dass der Minimum Spanning Tree auf einem Graph als Minimierungsproblem formuliert werde kann über einem Unabhängigkeitsystem. Außerdem wurde verifiziert, dass das Unabhängigkeitsystem ein Matroid ist. Dadurch können wir schließen, dass der Best-In-Greedy Algorithmus eine Optimalelösung erzeugt. $\Rightarrow$ Kruskals Algorithmus für Minimale Spannbäume

Genetische Algorithmen

Optimierungsalgorithmen
WiSe 2024/25

1 Motivation

Normale Lokale Suche hat das Problem, dass die Suche wahrscheinlich nur einen kleinen Subraum des Suchraumes betrachtet. Sehr gute zulässige Lösungen könnten wo anders sein.
Multi-Start Lokale Suche ist die simpelste Idee um diesem Problem entgegenzuwirken.

1.1 Multi-Start Lokale Suche

Man generiert eine Menge von zulässigen Lösungen. Dann startet man eine lokale Suche von all den Lösungen und liefert die beste Lösung von all diesen Lösungen.

2 Populationsbasierte Strategien

Multi-start ist die einfachste Populationsbasierte Strategie. Wir betrachten nun Evolutionäre Strategien, Genetische Algorithmen und Ameisenkolonie.

2.1 Evolutionäre Strategien

In dieser Strategie generiert man eine Menge an zulässigen Lösungen und startet eine Suche von allen. Der Unterschied nun ist, dass die Suche Rundenbasiert ist. Die Suchen verlaufen pseudo-parallel und am Ende "Überleben" die Suchen, die die besten Nachkommen erzeugen.
Die Suchen machen einen Schritt im Nachbarschaftsgraphen. Die erzeugten Lösungen werden miteinander verglichen, und die besten werden verwendet um neue Lösungen (Nachbarn) zu erzeugen.

Der Evolutionsschritt ist hierbei die Modifikation von Nachkommen, um den Suchraum weiter zu durchsuchen. Die Nachkommen basieren dann vollkommen auf eine vorherige Lösung. Also ist der Fokus hier **asexuelle** Mutation.

2.2 Genetische Algorithmen

Genetische Algorithmen basieren ebenfalls auf Populationen und sind sehr ähnlich zu evolutionäre Strategien. Der Hauptunterschied ist, dass Nachkommen durch zwei Eltern erzeugt werden durch sexuelle Rekombination. Die Suche verläuft hierbei nicht auf zulässigen Lösungen, sondern auf

abstrakten Repräsentation (Genen).
Für jede Instanz, werden alle abstrakten Repräsentationen als Strings dargestellt, die gleich Lang sind und aus dem Alphabet Σ bestehen.

2.2.1 Feature basierte Problem Definition mit Genen

Betrachte eine fixierte Instanz eines Feature-basierten Problems und sei n eine Anzahl von Features der Instanzen.
Enkodiere jede zulässige Lösung s als ein 0/1-Vektor x_s der Länge n sodass für $i \in \{1, ..., n\}$ gilt:

$$x_s[i] = 1 \Leftrightarrow i \text{ ist gewählt in } s$$

2.2.2 TSP als Gene

Eine Lösung des TSP könnte hier als Permutation von Städten dargestellt werden. Statt einer direkten Permutation kann man auch eine $n \times n$ Matrix mit binären Einträgen verwenden.
$X[i, j] = 1$ würde bedeuten, dass die Stadt i als j-te besucht wird.

2.2.3 Graph Coloring

Sei die Anzahl der Farben begrenzt auf die Anzahl der Knoten n. Eine zulässige Lösung kann als String der Länge n codiert werden, über dem Alphabet $\{1, ..., n\}$.
Oder man kodiert eine Lösung als $n \times n$-Matrix in binärer Weise:

$$X[i, j] = 1 \Leftrightarrow \text{ Knoten } i \text{ hat die Farbe } j$$

2.3 Gen-Rekombination, Crossover

Typischerweise werden zwei Nachfolger gleichzeitig produziert aus einer Elternkombination. Beide werden durch die selbe Prozedur erzeugt.

2.3.1 Qualität einer Crossover Strategie

Die Nachkommen sollten die Eltern gut repräsentieren. Fitte Eltern sollten Fitte Nachkommen erzeugen. Wenn ein Feature zB in beiden Eltern vorhanden ist, dann sollte es auch in Nachkommen sein. Wenn ein Feature in keinem der Eltern ist, dann sollte es auch nicht im Nachkommen sein.
Außerdem sollte ein Nachkomme von beiden Eltern erben, nicht nur ein Elternteil. Es sollte nicht ähnlich sein zu einem Elternteil, und unterschieden vom anderen.

2.3.2 Crossover Strategien

- One-Point: Gen wird an einem Index der Eltern vertauscht
- Two-Point: Intervall wird gewählt und zwischen Eltern getauscht
- Uniform: Bei jeder Genwahl wird durch Probabilistische Entscheidung entschieden ob von Elternteil 1 oder 2.

2.3.3 Zulässigkeit von Crossover

Das Ergebnis der Rekombination könnte eine unzulässige Lösung sein. Dieses Problem kann man Lösen indem man alle Side Constraints dropped (nicht empfohlen), oder die unzulässigkeit bestraft. Alternativ kann man auch die genetische Repräsentation so redesignen, sodass (fast) zulässige Lösungen erzeugt werden. Oder man wendet den Crossover an, und repariert im Nachhinein.

2.3.4 Crossover Resource Constrained Scheduling

Seien Jobs, mit einer Zeitlänge gegeben und Constraints. Außerdem ist gegeben, welche Jobs von welchen Abhängig sind. Jeder Task muss delayed werden, bis gewisse andere Tasks fertig sind (Precedence Constraints). Resourcenconstraints limitieren, wie viele Jobs gleichzeitig laufen dürfen. Die Gesamte Projektdauer soll minimiert werden.
Re-define Feasible offspring

Wir ignorieren die Ressourcen beschränkungen, und Bestrafen die Verletzung. Dadurch sollen Lösungen trotzdem die Precedence Constraints einhalten.
Jede Aufgabe wird durch eine Zahl beschrieben x_i. x_i ist die Slack TIme: Also wie viel früher eine Aufgabe erledigt werden könnte, ohne gegen Abhängigkeiten zu verstoßen. Lösungen die Precedence Constraints erfüllen, können durch **nicht-negative Werte** x_i dargestellt werden.
Crossover dieser Repräsentation

Die Gene sind die Slack-Werte. Der Crossover muss sicherstellen, dass die Werte nicht negativ bleiben. Für Crossover kann man hier verwenden: Austausch von Werten zwischen Eltern, Minimum oder Maximum der Eltern, Durchschnitt oder gewichteter Mittelwert.
Durch diese Methoden kann es jedoch passieren, dass die Ressourcen-Constraint verletzt wird.

2.3.5 Problem-spezifische Crossover Strategien

In PMX für TSP zB ist der Crossover einer der generischen. Die Reparatur ist jedoch Probelmspezifisch. Hierfür betrachten wir nun Order Crossover für TSP. Dieser stellt ebenfalls sicher dass Nachkommen zulässig sind.

Order Crossover Jede Elternlösung ist eine Permutation von Zahlen (zB Städtereihenfolge). Wir wählen ein Intervall. Dieser Intervall wird von P_1 zu O_1 übernommen. Die genutzten Werte aus

P_1 werden von P_2 entfernt ohne die relative Reihenfolge zu ändern. Die Restwerte werden in O_1 geschrieben.

2.4 Kulturelle Algorithmen

In evolutionäre Strategien ist der einzige Soziale Aspekt der Wettbewerb, zwischen Lösungen. Bei kulturellen Algorithmen reagieren die Populationsmitglieder auf Aktionen anderer Mitglieder. Wenn eine vielversprechende Aktion ausgeführt wird, reagieren die anderen Mitglieder positiv darauf.

2.4.1 Ameisenkolonie Optimierung

Jede Ameise stößt Phäromone aus. Die Ameisen versuchen der intensiveren Phäromenroute zu folgen. Je kürzer und effizienter der Pfad, desto höher die Wahrscheinlichkeit, dass eine Ameise diese genommen hat. Die Phäromonintensität gibt jedoch nur eine Wahrscheinlichkeit, dass eine Ameise dieser folgt. So vermeidet man lokale Optima.

Positives Feedback ist dann, dass eine Ameise das verhalten Simuliert. Negatives Feedback ist, dass die Phäromone nach einer Zeit verdampfen.

2.4.2 TSP als kultureller Algorithmus

Gegeben sei ein TSP mit n Städten. Eine Anzahl von Ameisen wird zufällig oder nach bestimmten Verteilungsregel über Städte verteilt.
Der Algorithmus arbeitet in Runden. Es gibt mehrere Major Rounds und jede Major Round besteht aus n Minor Rounds.
In jeder Minor Round bewegt sich eine Ameise zu einer Stadt, die sie in dieser Major Round noch nicht besucht hat. Nach n Minor Rounds hat jede Ameise eine vollständige Tour über alle Städte gemacht. Das Ziel ist es, die beste Tour zu finden, die von einer Ameise gemacht wird.
Wenn eine Ameise von einer Stadt zur anderen geht, hinterlässt es Pheromone. Um schlechte Wege nach und nach zu eliminieren, verdunstet ein Teil der Pheromone nach jeder Minor Round.
Eine Ameise wählt die nächste Stadt aus den nicht besuchten zufällig. Die Wahrscheinlichkeit, eine bestimmte Stadt j zu wählen, hängt ab von Distanz. Je kürzer desto Wahrscheinlicher und den Pheromonen auf dem Weg. Je mehr Pheromone, desto Wahrscheinlicher.
Gute Routen werden verstärkt, und schlechte verschwinden langsam.

2.4.3 Set Covering durch kulturelle Algorithmen

Gegeben sei eine Menge F von Elementen und eine Sammlung von Mengen S, die Teilmengen von F sind.
Elemente von S sind die Knoten im Graphen. Der Graph ist vollständig, also ist jeder Knoten mit

jedem anderen Knoten verbunden. Ein geordneter Teilmengen-Pfad durch den Graphen stelle eine mögliche Lösung dar. Jede Kante hat ein Gewicht, das den Nutzen angibt. Ein mögliches Maß für diesen Nutzen ist die Anzahl neuer abgedeckter Elemente. Die Ameise muss weiter durch den Graphen laufen, bis alle Elemente von F durch die gewählten Mengen aus S abgedeckt sind.
$\Rightarrow$ das Set Covering Problem wurde als Pfad Problem reformuliert.

Proofs

Optimierungsalgorithmen
WiSe 2024/25

1 Nachbarschaftsdefinition von Matchings sind exakt

1.1 Problemdefinition

Gegeben sei ein ungerichteter Graph $G = (V, E)$. Eine Lösung ist eine Menge von Kanten, sodass keine zwei Kanten einen gemeinsamen Knoten haben. Das Ziel ist es so viele Kanten wie möglich zu finden, die diese Bedingung erfüllen.

- Ein Pfad p in G ist elementar wenn jeder der Knoten im Pfad nur einmal hervorkommt
- Ein elementarer Pfad p in G ist alternierend, wenn genaue jede zweite Kante von p zum Matching gehört.

1.2 Nachbarschaftsdefinition

Zwei Matchings M_1 und M_2 in einem ungerichteten Graph $G = (E, V)$ sind benachbart wenn die symmetrische Differenz eine einzelner alternierender Pfad von M_1 und M_2.

$$M_1 \triangle M_2 = (M_1 \backslash M_2) \cup (M_2 \backslash M_1)$$

1.3 Beweis: Nachbarschaft ist exakt

Seien M_1 und M_2 zwei Matchings, sodass gilt: $|M_2| > |M_1|$. Wir müssen zeigen, dass in diesem Fall ein alternierender Pfad p für M_1 gibt, sodass folgendes gilt:

$$(M_1 \triangle p)| > |M_1|$$

Es ist klar, dass maximal eine Kante aus jeweils M_1 und M_2 mit einem Incident Knoten verbunden ist.

- Jeder Knoten hat einen maximalen Grad von 2 in der symmetrischen Differenz $M_1 \triangle M_2$
- $M_1 \triangle M_2$ zerfallen in elementare Pfade und Zyklen, welche alle alternierend sind von M_1 und M_2.

Weil $|M_2| > |M_1|$ gilt, mindestens einer dieser Pfade p, enthält dann mehr Kanten aus M_2 als aus M_1. Dieser Pfad ist der gesuchte um die Exaktheit der Nachbarschaft zu beweisen.

1.4 Theorem von Berge

Das Theorem von Berge besagt, dass eine Verbesserung in M nur vorgenommen werden kann, wenn es ein M-augmentierenden Pfad gibt. Ein augmentierender Pfad ist ein Pfad mit ungerader Länge und beiden Endknoten die nicht gematcht sind.

2 Nachbarschaftsdefinition von Min-Cost-Flows sind exakt

2.1 Problemdefinition

Gegeben sei ein gerichteter Graph $D = (V, A)$, untere und obere Schranken für Kapazitätswerte, ein Kostenfaktor für jede Kante und ein Bilanzwert für jeden Knoten.
Eine Lösung ist ein Flusswert für jede Kante, die die Kapazitätsconstraints einhält und die Flussbilanzwerte einhält. Das Ziel ist es die Gesamtkosten des Flusses zu minimieren.

2.2 Nachbarschaftsdefinition

Seien f_1 und f_2 zwei Flüsse für (D, l, u, b, c). Dann sind f_1 und f_2 benachbart, wenn sie sich in nur einem elementaren Weak-Cycle unterscheiden.
Ein Weak Cycle ist sind Zyklen, die gerichtete Kanten haben, und dieser Zyklus die Richtungen innerhalb wechselt.

2.3 Beweis: Nachbarschaft ist exakt

- Nehmen wir an, dass f_1 nicht optimal ist. Dann reicht es zu zeigen, dass es einen negativen Zyklus p gibt, welcher augmentierend ist im Bezug auf f_1, den unteren Schranken und oberen Schranken.
- Nehmen wir an, dass f_2 optimal ist. Das bedeutet $c(f_2) < c(f_1)$. Dann muss einer der Zyklen die durch das Flow Decomposition Lemma entstehen, ein negativer Zyklus sein. Sei p_i dieser Zyklus und ϵ_i ihre Multiplizität in der Flow Decomposition. Weil p_i augmentierend ist, ist $f_1 + \epsilon_i * p_i$ zulässig, und p_i ist der gesuchte Zyklus.

3 Simulated Annealing konvergiert (omitted)

3.1 Erklärung: Warum konvergiert Simulated Annealing?

Der Algorithmus kann als eine Markov-Kette betrachtet werden, da der nächste Zustand nur vom aktuellen Zustand abhängt und nicht von der Vergangenheit. Die Übergangsregeln erleuben, dass

jeder Zustand des Suchraums irgendwann erreicht werden kann (mit einer positiven Wahrscheinlichkeit). Das bedeutet, dass die Markov-Kette **ergodisch** ist. Durch die Wahl der schrittweise abnehmenden Temperatur folgt die Zustandswahrscheinlichkeit der Boltzmann-Verteilung, die sich bei langer Laufzeit stabilisiert. Also benötigt man auch unendliche Iterationen pro Temperatur. Dies ist jedoch unrealistisch, aber in der Theorie in Ordnung.
Aus der ergodischen Eigenschaft folgt, dass eine Markov Kette, die die Bedingungen von Simulated Annealing erfüllt zu einer stationären Verteilung konvergiert, die sich auf optimale Lösungen konzentriert.
Die Bedingungen sind die folgenden:

- Positive und sinkende Temperatur
- unendliche Iterationen
- Übergangsbedingung

3.2 Konsequenz des Beweises

Für eine zulässige Lösung s, für eine Temperatur die gegen 0 konvergiert und für unendliche Iterationen, gilt folgendes:
wenn s keine optimale Lösung ist, dann konvergiert die Wahrscheinlichkeit $P_{kT}(s)$ gegen 0.

3.2.1 Beweis

Es werden zwei zulässige Lösungen s_1 und s_2 mit unterschiedlichen Zielfunktionswerten betrachtet, wobei $\text{obj}(s_1) < \text{obj}(s_2)$. Also s_1 ist die bessere Lösung.

Simulated Annealing wählt Lösungen mit einer Wahrscheinlichkeit, die von deren Zielfunktionswert abhängt. Besser Lösungen haben eine höhere Wahrscheinlichkeit, und schlechtere eine niedrigere Wahrscheinlichkeit.
Die Wahrscheinlichkeiten folgen einer Regel, die der sogenannten Boltzmann-Verteilung ähnelt. Diese Regel sagt:

- schlechtere Lösungen werden mit kleinerer Wahrscheinlichkeit akzeptiert
- Unterschied zwischen zwei Lösungen wird verstärkt, wenn die Temperatur sinkt

Wenn die Temperatur sehr hoch ist, sind falls alle Lösungen ungefähr gleich wahrscheinlich. Wenn die Temperatur kleiner wird, dann verstärkt sich der Unterschied zwischen guten und schlechten Lösungen. Höhere Wahrscheinlichkeit dass man bessere Lösungen akzeptiert, folgt daraus.

Wenn die Temperatur gegen 0 geht, dann wird die Wahrscheinlichkeit, eine suboptimale Lösung zu wählen sehr gering. Das bedeutet:

- Wahrscheinlichkeit für eine nicht-optimale Lösung geht gegen null
- Wahrscheinlichkeit für eine optimale Lösung geht gegen eins

4 Lagrangian Multiplikation ist eine untere Schranke

4.1 Lemma

Für beliebigen Vektor von Lagrange Multiplikatoren λ, der Wert der Lagrange Funktion $L(\lambda)$ ist eine untere Schranke des Zielwertes $z(P)$ des Originalproblems.

4.2 Beweis

Das ursprüngliche Problem hat eine lineare Nebenbedingung $Ax = b$. Für jeden Vektor von Lagrange Multiplikatoren λ, kann die Zielfunktion umgeschrieben werden als:

$$min\{c^T x | Ax = b, x \in X\} = \min\{c^T x + \lambda^T (Ax - b) | Ax = b, x \in X\}$$

Diese Umformung bestraft das nicht erfüllen der Nebenbedingung. Sonst ist es 0 und beeinflusst die Kosten nicht. Wenn die Nebenbedingung entfernt wird, kann der Zielfunktionswert nicht erhöhen. Denn eine Minimierung über eine größere Menge kann den Zielfunktionswert nur gleich lassen oder senken.
Deswegen ist es immer eine untere Schranke für die optimale Lösung.

5 Graphic Matroid erfüllt M3

5.1 Definition

Ein Matroid ist ein Unabhängigkeitssystem, wenn man aus einer größeren Unabhängigen Lösung ein Element in eine kleinere machen kann, und diese neue Menge dann immernoch unabhängig ist. Ein Grafischer Matroid ist ein ungerichteter Graph $G = (V, E)$, wobei E die Grundmenge des Matroiden ist, und F die Menge aller Teilgraphen, die Wälder (Zyklenfrei) darstellen.

5.2 Beweis: Die Unabhängigkeitsverschiebungsregel gilt

Es seien $X, Y \in F$, also sind (V, X) und (V, Y) Wälder. Es wird angenommen, dass kein Element aus $x \in X \backslash Y$ existiert, sodass $Y \cup \{x\}$ immernoch ein Wald ist.
Ziel ist es zu zeigen, dass dann $|X| \leq |Y|$ gilt.

Jede Kante $x = \{v, w\} \in X$ verbindet zwei Knoten im Graphen. Da $Y \cup \{x\} \notin F$ für jedes Element aus $X \backslash Y$, bedeutet dies, dass jede dieser Kanten in $X \backslash Y$ einen Zyklus in Y erzeugen würden. Also alle Kanten von X sind bereits in einer Zusammenhangskomponente von (V, Y). Weil jede Zusammenhangskomponent aus (V, X) vollständig in einer von (V, Y) enthalten ist, gilt $|(V, X)| \geq |(V, Y)|$.

In einem Wald mit k Kanten und n Knoten ist die Anzahl der Komponenten gegeben durch:

$$\text{Anzahl Komponenten} = n - \text{Anzahl Kanten}$$

Also:

$$|(V, X)| = |V| - |X|, |(V, Y)| = |V| - |Y|$$

Durch vorheriger Ungleichung kann man dies zu $|X| \leq |Y|$ umformen. Der Beweis ist nun vollständig.

6 Ein Unabhängigkeitssystem ist ein Matroid, wenn Best-In-Greedy für jede Kostenfunktion eines Maximierungsproblem die optimale Lösung findet

6.1 Matroid ⇒ Greedy ist optimal

Sei (E, F) ein Matroid, und wir konstruieren eine Lösung L mit Greedy. Angenommen, es existiert eine unabhängige Lösung L', die höhere Zielfunktion hat als L. Dann kann L' als Basis betrachtet werden, weil alle Elemente positive Werte haben.
Da F' eine Basis ist, hat es die gleiche Größe wie L.
Da Greedy eine schlechtere Summe liefert als L', muss es eine Kante e geben, an dem Greedy einen kleineren Wert gewählt hat, als Möglich. Da L' einen größeren Wert in einer der ELement hat, hätte Greedy diesen Wert auch gewählt, aber das hat er anscheinend nicht. Das führt zu einem Widerspruch, da Greedy niemals ein Element mit niedrigerem Wert vor einem mit höhere, Wert nimmt.

6.2 Greedy ist nicht optimal, wenn kein Matroid vorliegt

Wir nehmen an, dass unser System kein Matroid ist. Es gibt zwei unabhängige Mengen F_1 und F_2. Für diese gilt:

$$|F_1| = p$$
$$|F_2| = p + 1$$

- Kein Element aus $F_2 \backslash F_1$ kann zu F_1 hinzugefügt werden, ohne die Unabhängigkeit zu verlieren, weil das System kein Matroid ist.

Um zu zeigen, dass Greedy eine falsche Lösung wählt, konstruieren wir folgende Kostenfunktion:

- $\forall f_1 \in F_1 : c(f_1) = p + 2$
- $\forall f_2 \in F_2 \backslash F_1 : c(f_2) = p + 1$
- alle anderen Elemente haben den Kostenwert 0

Berechnen wir die Gesamtkosten beider Mengen:
Greedy wählt F_1, denn die Kosten für ein Element aus F_1 sind $p + 2$.

$$c(F_1) = p * (p + 2) = p^2 + 2p$$

Optimale Lösung wäre jedoch F_2:

$$c(F_2) = (p + 1)^2 = p^2 + 2p + 1$$

Greedy beginnt mit den teuersten Elementen, also wählt er zuerst alle Elemente aus F_1. Danach kann der Algorithmus nichts mehr verbessern, weil jede andere Wahl von F_2 nicht erlaubt ist, weil die Unabhängigkeitsbedingung nicht mehr gewährleistet ist, weil es kein Matroid ist.

7 Greedy Algorithmus liefert eine 2-Approximation für Maximum Weighted Matching

Betrachten wir die symmetrische Differenz $M_{opt} \triangle M_{app}$. Es gibt keine isolierten Kanten in dieser Differnez, weil jedes $e \in M_{opt}$ mit mindestens einer Kante $e \in M_{app}$ verbunden ist.
Jede Kante aus der optimalen Lösung kann nur mit höchstens zwei Kanten aus der Approximation verbunden sein, nennen wir sie e' und e''. Das liegt daran, dass ein Matching keine Knoten teilen kann, daher gibt es nur maximal zwei benachbarter Kanten.

Wenn $w(e) > w(e')$ und $w(e) > w(e'')$. Falls Greedy optimal arbeiten würde, hätte es e gewählt, statt e' oder e''. Da das nicht passiert ist, muss eines der folgenden Bedingungen gelten:

$$w(e') \geq w(e) \text{ oder } w(e'') \geq w(e)$$

Es existiert also immer eine Kante in der Approximation, die mindestens so viel wie e wiegt. Man kann dieses Gewicht von e auf eine benachbarte Kante aus der Approximation verlagern. Weil jede Kante in der optimalen Lösung mit maximal zwei Kanten aus der Approximation verbunden ist, wird jede Kente aus der Approximation höchstens zweimal belastet. Daraus resultiert:

$$w(M_{opt}) \leq 2w(M_{app})$$

8 Minimum Vertex Cover Approximation liefert eine 2-Approximation

Jede Kante im Matching, wird durch genau zwei Endpunkte abgedeckt. Da M ein maximales Matching ist, gibt es keine weiteren Kanten, die zwei ungematchte Knoten verbindet. Deswegen ist die gewählte Menge von Knoten ein gültiges Vertex Cover.

Jede Lösung für das Minimum Vertex Cover muss mindestens eine Knotenmenge der Größe $|M|$ enthalten. Denn kein Vertex Cover kann weniger als eine Knotenmenge mit $|M|$ Knoten haben, da jede Kante in M mindestens ein Endpunkt im Cover haben muss.

Unser Algorithmus wählt genau $2|M|$ Knoten, da er beide Endpunkte jeder Kante nimmt. Daher gilt:

$$2|M| \leq 2 * OPT(I)$$

Also die Approximation wählt höchstens doppelt so viele Knoten wie das Optimum.

9 Steiner-Tree-Approximation liefert 2-Approximation

Zu zeigen ist, dass die Länge des von der Approximation konstruierten Steiner-Baums höchstens doppelt so groß ist wie die optimale Lösung.

Der optimale Steiner Baum T^* ist ein Untergraph von G und verbindet alle Terminals. Der MST T' auf N_d hat eine Länge, die höchstens so groß ist wie T^*. Da N_d nur kürzeste Pfade zwischen Terminals enthält, kann T' nicht schlechter sein als T^*.

Beim Ersetzen der Kanten von T' in G kann die Länge höchstens verdoppelt werden. Denn jeder Kante in T' entspricht im schlimmsten Fall ein Pfad in G, der doppelt so lang ist wie die direkte Verbindung.
Der schlechteste Fall ist also dass der Steiner-Baum maximal die doppelte Länge des optimalen Baums hat. Der MST im Distanznetwerk ist bereits eine gute Näherung und das Einsetzen der kürzesten Pfade in G verdoppelt höchstens die Kosten.

10 Double Tree Algorithm ist eine Faktor 2 Approximation

Sei T ein Minimaler Spannbaum des Graphen. Die optimale Tour muss mindestens so lang sein wie ein Spannbaum. Weil durch Entfernen einer Kante in der Tour ein Spannbaum entsteht. Wenn wir alle Kanten von T verdoppeln, erhalten wir einen Eulerkreis T' (weil jetzt jeder Knoten geraden Grad hat und daher ein geschlossener Eulerkreis existiert). Die Länge des doppelten Baums ist:

$$c(E(T')) = 2 * c(E(T)) \leq 2 * OPT$$

Der Eulerkreis kann Knoten mehrmals besuchen. Um eine gültige TSP-Tour zu erhalten, entfernen wir alle doppelten Besuche und behalten nur das erste Vorkommen jedes Knotens. Wegen der Dreiecksungleichung kann das Entfernen der doppelten Besuche die Gesamtkosten nicht erhöhen.

Die durch das Entfernen der doppelten Besuche entstehende Tour hat höchstens die gleiche Länge wie der Eulerkreis. Damit ist sichergestellt, dass die vom Algorithmus gefundene Lösung höchstens doppelt so teuer wie die optimale Lösung ist.

11 Christofides Algorithmus liefert eine 1.5-Approximation für das Metric TSP

Wir wollen zeigen, dass die von Christofides konstruierte Tour höchsten $1.5 * OPT$ kostet.

Die von Christofides konstruierte Tour basiert auf zwei Hauptkomponenten:

1. Ein Spannbaum T mit minimalen Kosten. Weil das Entfernen einer Kante aus jeder TSP-Tour einen Spannbaum hinterlässt, ist klar: $c(E(T)) \leq OPT$.
2. Ein minimales perfektes Matching M auf den ungeraden Knoten. Das Matching M verbindet alle Knoten mit ungeradem Grad. Dadurch stellt es sicher, dass eine Euler Tour existiert. $c(E(T')) = c(E(T)) + c(M)$.

Betrachten wir die Reihenfolge der ungeraden Knoten W in der optimalen Tour T_{opt}. Definieren wir zwei perfekte Matchings:

- $M_1 = \{(w_1, w_2), (w_3, w_4), ..., (w_{2k-1}, w_{2k})\}$
- $M_2 = \{(w_2, w_3, ..., (w_{2k}, w_1)\}$

Weil jede Optimale Tour eine Perfekte Paarung der ungeraden Knoten enthält, gilt:

$$OPT \geq c(M_1) + c(M_2)$$

Weil M das Minimum Perfect Matching ist, gilt:

$$c(M) \leq \min(c(M_1), c(M_2)) \leq \frac{1}{2}(c(M_1) + c(M_2)) \leq \frac{1}{2} * OPT$$

Der Algorithmus nutzt den MST und das Matching um die Tour zu konstruieren. Die Kosten des MST sind höchstens die der optimalen Tour. Die Kosten des Matchings sind höchstens halb so groß wie die optimale Tour.
Daraus resultieren folgende Kosten: $OPT + \frac{1}{2} * OPT = 1.5 * OPT$