



Datenstrukturen, Algorithmen und Programmierung 2 (DAP2)

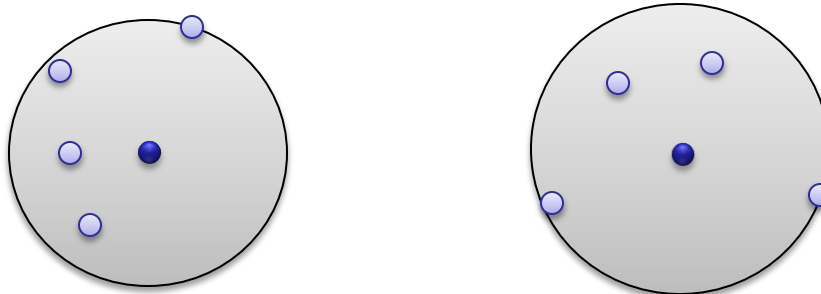
Approximationsalgorithmen

Das (diskrete) k-Center Clustering Problem

- Gegeben: Menge P von n Punkten in der Ebene
- Gesucht: Menge $C \subseteq P$ von k Zentren, so dass die maximale Distanz der Punkte zum nächstgelegenen Zentrum, d.h.
$$\text{cost}(P,C) = \max_{p \in P} d(p,C)$$
 minimiert wird, wobei
- $\text{dist}(p,C) = \min_{q \in C} \text{dist}(p,q)$ und $\text{dist}(p,q)$ bezeichnet den Abstand von p und q (Euklidische Distanz)

Approximationsalgorithmen

Beispiel



Alternative Formulierung

Finde k Scheiben mit Zentrum aus P , die alle Punkte abdecken und deren Maximaler Radius minimiert wird.

Approximationsalgorithmen

Typische Anwendung

- Punkte symbolisieren Städte
- Will Mobilfunkmasten mit möglichst geringer Leistung aufstellen, so dass alle Städte versorgt sind

Allgemeiner

- Punkte (typischerweise in höheren Dimensionen) sind „Beschreibungen von Objekten“
- Will Objekte in Gruppen von ähnlichen Objekten unterteilen

Approximationsalgorithmen

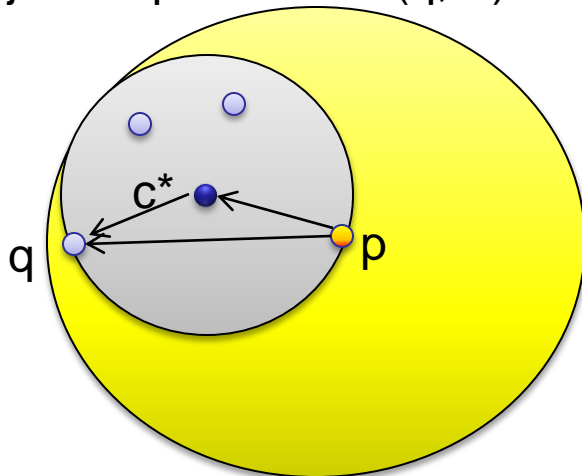
Ein Gedankenexperiment

- Nehmen wir an, wir kennen die Kosten r einer optimalen Lösung, d.h. wir wissen, dass man mit Scheiben mit Radius r und Zentrum aus P die Punkte abdecken kann
- Wir werden zeigen, dass wir dann einen einfachen 2-Approximationsalgorithmus finden können

Approximationsalgorithmen

Idee

- Wir nutzen Existenz von Lösung C^* mit Radius (Kosten) r
- Betrachte Punkt $p \in P$
- Dann gibt es Zentrum $c^* \in C^*$ mit $\text{dist}(p, c^*) \leq r$
- Nehmen wir nun p als Zentrum anstelle von c^* und verdoppeln wir den Radius, so decken wir jeden Punkt q ab, der von c^* mit Radius r abgedeckt wurde, d.h.
- für jedes $q \in P$ mit $\text{dist}(q, c^*) \leq r$ gilt $\text{dist}(p, q) \leq \text{dist}(p, c^*) + \text{dist}(c^*, q) \leq 2r$



Approximationsalgorithmen

k-Center1(P,k)

1. $C = \emptyset$; $P' \leftarrow P$
2. **while** $P' \neq \emptyset$ **do**
3. Wähle beliebigen Punkt $p \in P'$
4. $C = C \cup \{p\}$
5. Lösche alle Punkte aus P' mit Distanz höchstens $2r$ von p
6. **if** $|C| \leq k$ **then return** C
7. **else return** „Es gibt keine Menge von k Zentren mit Radius r “

Offensichtlich gilt

Jede Menge von k Zentren, die der Algorithmus zurückgibt hat Kosten höchstens $2r$.

Approximationsalgorithmen

Lemma 86

- Wenn Algorithmus k-Center1 mehr als k Zentren auswählt, dann gilt für jede Menge $C^* \subseteq P$ von k Zentren, dass $\text{cost}(P, C^*) > r$ ist.

Beweis (durch Widerspruch)

- Annahme: Es gibt C^* mit $\text{cost}(P, C^*) \leq r$ und $|C^*| \leq k$ und $|C| > k$.
- Sei C die Menge der Zentren, die k-Center1 auswählt
- Da $C \subseteq P$ gibt es für jedes $p \in C$ (mindestens) ein $c^* \in C^*$ mit $\text{dist}(p, c^*) \leq r$
- Wir nennen c^* nah zu p
- Behauptung: Kein c^* kann nah zu zwei $p \in C$ sein (Beweis später)
- Damit folgt: $|C^*| \geq |C|$ und somit Widerspruch zu $|C^*| \leq k$ und $|C| > k$.

Approximationsalgorithmen

Lemma 86

- Wenn Algorithmus k-Center1 mehr als k Zentren auswählt, dann gilt für jede Menge $C^* \subseteq P$ von k Zentren, dass $\text{cost}(P, C^*) > r$ ist.

Beweis (durch Widerspruch)

- Behauptung: Kein c^* kann nah zu zwei $p \in C$ sein
- Beweis der Behauptung:
- Alle Paare von Zentren p, q aus C haben Abstand $> 2r$
- Wäre nun für ein Zentrum c^* $\text{dist}(p, c^*) \leq r$ und $\text{dist}(q, c^*) \leq r$, so würde $\text{dist}(p, q) \leq \text{dist}(p, c^*) + \text{dist}(c^*, q) = \text{dist}(p, c^*) + \text{dist}(c^*, q) \leq 2r$ gelten. Widerspruch!

Approximationsalgorithmen

Was, wenn wir r nicht kennen?

- Wir wissen nicht, welche Punkte Distanz größer als $2r$ von den bisher ausgewählten Zentren haben
- Idee: Wähle immer den am weitesten entfernten Punkt, d.h. der $\text{dist}(p,C)$ maximiert
- Gibt es einen Punkt mit $\text{dist}(p,C) > 2r$, dann ist es dieser

Approximationsalgorithmen

k-Center2(P,k)

1. Wähle beliebigen Punkt p aus P und setze $C=\{p\}$
3. **while** $|C|<k$ **do**
3. Wähle Punkt $p\in P$, der $\text{dist}(p,C)$ maximiert
4. $C=C\cup\{p\}$
5. **return** C

Approximationsalgorithmen

Satz 87

- Algorithmus k-Center2 ist ein 2-Approximationsalgorithmus für das diskrete k-Center Clustering Problem. Algorithmus k-Center2 kann mit Laufzeit $O(nk)$ implementiert werden.

Beweis

- Zunächst zur Laufzeit:
- Um den Algorithmus in $O(nk)$ Zeit zu implementieren, müssen wir jeden Schleifendurchlauf in $O(n)$ Zeit erledigen können. Dazu speichern wir uns für jeden Punkt p den Wert $\text{dist}(p,C)$.
- Wird nun ein neues Zentrum c in C eingefügt, so müssen wir nur für jeden Punkt überprüfen, ob $\text{dist}(p,c) < \text{dist}(p,C)$ ist und ggf. $\text{dist}(p,C)$ aktualisieren. Dies geht insgesamt in $O(n)$ Zeit.

Approximationsalgorithmen

Satz 87

- Algorithmus k-Center2 ist ein 2-Approximationsalgorithmus für das diskrete k-Center Clustering Problem. Algorithmus k-Center2 kann mit Laufzeit $O(nk)$ implementiert werden.

Beweis

- Wir bezeichnen nun mit r die Kosten einer optimalen Lösung C^* .
Annahme: Algorithmus liefert Menge C mit Kosten $>2r$.
- Dann gibt es einen Punkt p mit $\text{dist}(p,C) > 2r$
- Da der Algorithmus immer den Punkt auswählt, der maximalen Abstand zu den bisher ausgewählten Zentren hat, haben alle ausgewählten Zentren Abstand $>2r$ zur den bisher ausgewählten Zentren
- Somit würde Algorithmus k-Center1 auf dieser Eingabe mehr als k Zentren zurückgeben. Damit gilt nach Lemma 86 $\text{cost}(P,C^*) > r$. Widerspruch!

Approximationsalgorithmen

Zusammenfassung & Kommentare

- Man kann viele Probleme approximativ schneller lösen als exakt (für die drei Beispiele sind keine Algorithmen mit Laufzeit $O(n^c)$ für eine Konstante c bekannt)
- Gierige Algorithmen sind häufig Approximationsalgorithmen

Laufzeitanalyse

Maschinenmodell

- Eine Pseudocode-Instruktion braucht einen Zeitschritt
- Wird eine Instruktion r -mal aufgerufen, werden r Zeitschritte benötigt
- Formales Modell: **Random Access Machines** (RAM Modell)

Idee

- Ignoriere rechnerabhängige Konstanten
- Betrachte Wachstum von $T(n)$ für $n \rightarrow \infty$

„Asymptotische Analyse“

Laufzeitanalyse

O-Notation

- $O(f(n)) = \{g(n) : \exists c > 0, n_0 > 0, \text{ so dass für alle } n \geq n_0 \text{ gilt } g(n) \leq c \cdot f(n)\}$
- (wobei $f(n), g(n) > 0$)

Interpretation

- $g(n) \in O(f(n))$ bedeutet, dass $g(n)$ für $n \rightarrow \infty$ höchstens genauso stark wächst wie $f(n)$
- Beim Wachstum ignorieren wir Konstanten

Korrektheitsbeweise

Ein triviales Beispiel

EinfacherAlgorithmus(n)

1. $X \leftarrow 10$
2. $Y \leftarrow n$
3. $X \leftarrow X + Y$
4. **return** X

Ein Korrektheitsbeweis vollzieht also das Programm Schritt für Schritt nach.

Behauptung

Der Algorithmus gibt der Wert $10+n$ zurück.

Beweis:

Zu Beginn des Algorithmus sind alle Variablen bis auf den Parameter n undefiniert. Der Befehl in Zeile 1 weist X den Wert 10 zu. Der Befehl in Zeile 2 weist Y den Wert n zu. Der Befehl in Zeile 3 weist X den Wert $X + Y$ zu. Da X vor der Zuweisung den Wert 10 enthielt und Y den Wert n , wird X auf $10+n$ gesetzt. Der Befehl in Zeile 4 gibt X zurück. Da X zu diesem Zeitpunkt den Wert $10+n$ hat, folgt die Behauptung.

Korrektheitsbeweise

Ein erstes nichttriviales Beispiel

Algorithmus Max-Search(Array A)

1. $\text{max} \leftarrow 1$
2. **for** $j \leftarrow 2$ **to** $\text{length}[A]$ **do**
3. **if** $A[j] > A[\text{max}]$ **then** $\text{max} \leftarrow j$
4. **return** max

Definition (Schleifeninvariante)

Eine Schleifeninvariante ist eine i.a. von der Anzahl i der Schleifendurchläufe abhängige Aussage $A(i)$, die zu Beginn des i -ten Schleifendurchlauf gilt. Mit $A(1)$ beziehen wir uns also auf den Zustand zu Beginn des ersten Durchlaufs. Dieser wird auch als Initialisierung bezeichnet.

Korrektheitsbeweise - Rekursionen

Algorithmus Sum(A,n)

1. **If** $n=1$ **then** return $A[1]$
2. **else**
3. $W = \text{Sum}(A, n-1)$
4. **return** $A[n] + W$

Beweis (Induktion über n):

(I.A.) Ist $n=1$, so gibt der Algorithmus in Zeile 1 den Wert $A[1]$ zurück. Dies ist korrekt.

(I.V.) Für $n-1 > 0$ berechnet $\text{Sum}(A, n-1)$ die Summe der ersten $n-1$ Einträge von A.

(I.S.) Wir betrachten den Aufruf von $\text{Sum}(A, n)$. Da $n > 1$ ist, wird der else-Fall der ersten if-Anweisung aufgerufen. Dort wird W auf $\text{Sum}(A, n-1)$ gesetzt. Nach I.V. ist dies die Summe der ersten $n-1$ Einträge von A. Nun wird in Zeile 4 $A[n] + W$, also die Summe der ersten n Einträge von A zurückgegeben.

Satz 3

Algorithmus Sum(A,n) berechnet die Summe der ersten n Einträge eines Feldes A.

Teile & Herrsche

Teile & Herrsche (Divide & Conquer)

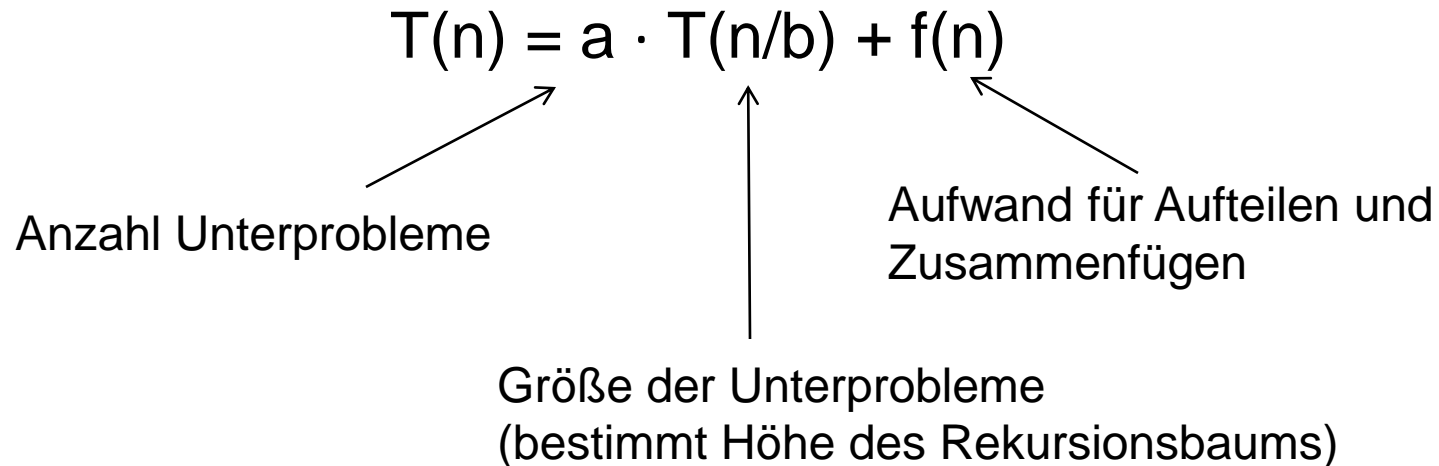
- Teile Eingabe in mehrere Teile auf
- Löse das Problem rekursiv auf den Teilen
- Füge die Teillösungen zu einer Gesamtlösung zusammen

Beispiel(Sortieren)



Teile & Herrsche

Laufzeiten der Form



- (und $T(1) = \text{const}$)

Welche unterschiedlichen Fälle gibt es?

Gierige Algorithmen

Gierige Algorithmen

- Konstruiere Lösung Schritt für Schritt
- In jedem Schritt: Optimierte ein einfaches, lokales Kriterium

Beobachtung

- Man kann viele unterschiedliche gierige Algorithmen für ein Problem entwickeln
- Nicht jeder dieser Algorithmen löst das Problem korrekt

Dynamische Programmierung

Beobachtung

- Die Funktionswerte werden bottom-up berechnet

Grundidee der dynamischen Programmierung

- Berechne die Funktionswerte iterativ und bottom-up

FibDynamischeProgrammierung(n)

1. Initialisiere Feld $F[1..n]$
2. $F[1] \leftarrow 1$
3. $F[2] \leftarrow 1$
4. **for** $i \leftarrow 3$ **to** n **do**
5. $F[i] \leftarrow F[i-1] + F[i-2]$
6. **return** $F[i]$

Dynamische Programmierung

Dynamische Programmierung

- Formuliere Problem rekursiv
- Löse die Rekursion „bottom-up“ durch schrittweises Ausfüllen einer Tabelle der möglichen Lösungen

Wann ist dynamische Programmierung effizient?

- Die Anzahl unterschiedlicher Funktionsaufrufe (Größe der Tabelle) ist klein
- Bei einer „normalen Ausführung“ des rekursiven Algorithmus ist mit vielen Mehrfachausführungen zu rechnen

Vorgehensweise bei dynamischer Programmierung

1. Bestimme rekursive Struktur einer optimalen Lösung.
2. Entwerfe rekursive Methode zur Bestimmung des Wertes einer optimalen Lösung.
3. Transformiere rekursiv Methode in eine iterative (bottom-up) Methode zur Bestimmung des Wertes einer optimalen Lösung.
4. Bestimmen aus dem Wert einer optimalen Lösung und in 3. ebenfalls berechneten Zusatzinformationen eine optimale Lösung.

Datenstrukturen

Drei grundlegende Datenstrukturen

- Feld
- sortiertes Feld
- doppelt verkettete Liste

Diskussion

- Alle drei Strukturen haben gewichtige Nachteile
- Zeiger/Referenzen helfen beim Speichermanagement
- Sortierung hilft bei Suche ist aber teuer aufrecht zu erhalten

Datenstrukturen

Binäre Suchbäume

- Aufzählen der Elemente mit Inorder-Tree-Walk in $O(n)$ Zeit
- Suche in $O(h)$ Zeit
- Minimum/Maximum in $O(h)$ Zeit
- Vorgänger/Nachfolger in $O(h)$ Zeit

Dynamische Operationen?

- Einfügen und Löschen
- Müssen Suchbaumeigenschaft aufrecht erhalten
- Auswirkung auf Höhe des Baums?

Datenstrukturen

Hashing

- Hashing nutzt Vorteil der direkten Addressierung ($O(1)$ Suchzeit), reduziert aber gleichzeitig den Speicherbedarf auf $O(n)$
- Hashfunktion kann zufällig aus universeller Klasse von Hashfunktionen gewählt werden
- Dies garantiert durchschnittliche Suchzeit $O(1)$

Graphalgorithmen

SSSP (pos. Kantengewichte):

- Dijkstra; Laufzeit $O((|V|+|E|) \log |V|)$

SSSP (allgemeine Kantengewichte)

- Bellman-Ford; Laufzeit $O(|V|^2+|V| |E|)$

APSP (allgemeine Kantengewichte, keine negativen Zyklen)

- Floyd-Warshall; Laufzeit $O(|V|^3)$

Graphalgorithmen

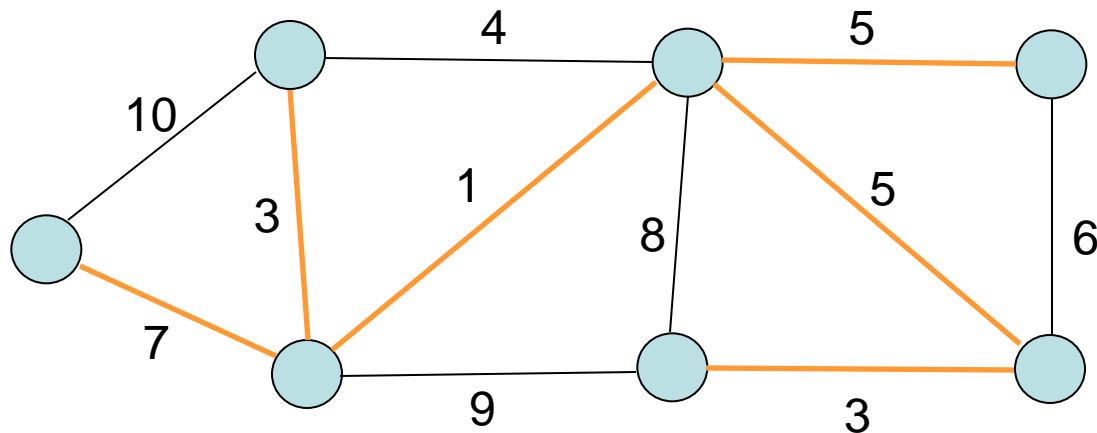
Tiefensuche und Breitensuche

- Tiefensuche bietet andere Möglichkeit (neben Breitensuche) zur Graphtraversierung in Laufzeit $O(|V|+|E|)$
- Die Sortierung der Tiefensuche kann zum topologischen Sortieren benutzt werden

Graphalgorithmen

Minimale Spannbäume

- Gegeben: Gewichteter, ungerichteter, zusammenhängender Graph $G=(V,E)$
- Gesucht: Ein aufspannender Baum mit minimalem Gewicht
- Aufspannender Baum: Inklusionsmaximaler kreisfreier Teilgraph mit Knotenmenge V



Approximationsalgorithmen

Was kann man tun, wenn man Problem nicht effizient lösen kann?

- Die Aufgabenstellung vereinfachen!

Approximationsalgorithmen

- Löse Problem nicht exakt, sondern nur approximativ
- Qualitätsgarantie in Abhängigkeit von optimaler Lösung
- Z.B.: jede berechnete Lösung ist nur doppelt so teuer, wie eine optimale Lösung

Heuristik

- Löse ein Problem nicht exakt
- Keine Qualitätsgarantie
- Können jedoch in der Praxis durchaus effizient sein

Viel Erfolg bei der Klausur und schöne Semesterferien!