Klausurblatt GDI3 Thorsten Wink

Graphen
- bestehen aus Knoten (V) und Kanten (E)
- zwei durch eine Kante verbundene Knoten heißen adjazent
- Außengrad d+ = Anzahl der Kanten, die V als Startknoten haben
- Innengrad d- = Anzahl der Kanten, die V als Endknoten haben
- Haben alle Knoten den selben Grad, so spricht man von n-regulär
- Vollständiger Graph = alle Knoten sind miteinander verbunden
- Teilgraph: Ausschnitt aus einem Graphen, muss nicht alle Kanten enthalten
- Untergraph: muss alle Kanten entsprechend der Knoten enthalten
- Knoteneinfacher Weg: jeder Knoten wird nur einmal besucht, Hamiltonscher Kreis
- Kanteneinfacher Weg: jede Kante wird nur einmal besucht, Eulerscher Kreis
- Wegegraph G^+: alle möglichen Wege direkt eintragen
- Azyklische Graphen: Graphen ohne Zyklus
- Bei ungerichteten Graphen existiert ein Eulerscher Kreis, wenn G zusammenhängend und alle Knotengrade gerade sind.
- Trennknoten/Schnittkante: Wenn man ihn entfernt zerfällt der Graph in 2 Teilgraphen
- Aufspannender Teilgraph: nur nötige Kanten um Verbindungen zu erhalten
- Planarer Graph: er lässt sich ohne Überschneidung darstellen

---

Speicherung von Graphen
Adjazenzmatrix: 1 wenn Verbindung zwischen den Knoten existiert, sonst 0. Algorithmen, die mit der Adjazenzmatrix arbeiten haben immer einen Aufwand von $O(V^2)$.

Adjazenztabelle: 3 Spalten: 1. Knotennr, 2. Nachfolger, 3. nächste Zeile mit Nachfolger oder -1 wenn keiner mehr

Adjazenzliste: Darstellung mittels verketteter Liste: erst Knoteninfo, dann Nachfolger, dann Verweis auf nächsten Knoten, effizienter als Matrix da hier nur die wirklich vorhanden Kanten gespeichert werden

---

Verkettete Listen:
Speicherstruktur, einfach/doppelt verkette Listen, speichert immer Verweis zum Vorgänger/Nachfolger + Speicherinhalt, Operationen: einfügen, löschen, Listeninspektion
Weitere datenstrukturen: Queue, Stack

Traversieren von Graphen
DFS: Suche in die Tiefe, rekursiv für alle Nachfolger aufrufen, Implementierung mit Stack, O(max(|V|, |E|))
BFS: Suche in die Breite, Implementierung mit Queue, erst alle Nachfolger, O(max(|V|, |E|))

Minimale aufspannende Bäume
Kruskal: Kanten nach Kosten sortieren, dann die geringste übernehmen wenn kein Zyklus entsteht. O(|E|log|E|)
Prim: Von einem Startknoten die billigste Kante nehmen, dann von diesem usw. bis Baum fertig, O(|E|log|E).

Minimale Wege
Moore: Startknoten, markiert alle Nachfolger mit 1, dessen Nachfolger (wenn noch nicht besucht) mit 2 usw., nur bei ungewichteten Kanten anwendbar, wie BFS; O(max(|V|, |E|))

Dijkstra (die Katastrophe, die jeder INF angeblich braucht ;)
Zuerst alle Nachfolger vom Startknoten besuchen + in M2, Entfernung in Tabelle, dann mit dem Knoten aus M2 weitermachen der die geringste Entfernung hat, $O(V^2)$

Allgemeines Wegeproblem
Kleene-Algorithmus $O(V^3)$:
Immer Matrixdarstellung, vielseitig verwendbares Schema, wird über den Semiring konkretisiert. Zwei Wege von i nach j werden über (+) verknüpft, zwei Teilstrecken über (·).
S=(M, (+), (·), I0, I1)

S=({0,1},•,•,0,1)	ó transitive Hülle; $c_{ij}^{(k)} := c_{ij}^{(k-1)} \bullet (c_{ik}^{(k-1)} \bullet c_{kj}^{(k-1)})$
S={R∪{∞}, min,+,∞,0)	ó kürzeste Entfernung; $c_{ij}^{(k)} := \min(c_{ij}^{(k-1)}, (c_{ik}^{(k-1)} + c_{kj}^{(k-1)}))$
S={R∪{-∞}, max,+,-∞,0)	ó längster Weg; $c_{ij}^{(k)} := \max(c_{ij}^{(k-1)}, (c_{ik}^{(k-1)} + c_{kj}^{(k-1)}))$
S={R∪{∞, max,min,0,∞)	ó maximale Kapazität; $c_{ij}^{(k)} := \max(c_{ij}^{(k-1)}, \min(c_{ik}^{(k-1)}, c_{kj}^{(k-1)}))$
$S=(2^M, \cup, \bullet, 0, A)$	ó Anzahl aller einfachen Wege; NP-vollständig

---

Bäume
- ungerichtet, zyklenfrei, zusammenhängend, eine Wurzel, Knoten ohne Nachfolger = Blätter
- Anzahl Nachfolger = Ordnung des Knotens, maximal auftretende Ordnung = Ordnung des Baumes
- Nachfolgeknoten = Söhne, Vorgängerknoten = Vater
- Tiefe eines Knotens = Anzahl der kanten von der Wurzel bis zum Knoten
- Höhe eines Knotens = Anzahl der Kanten auf dem längsten weg zu einem Blatt
- Höhe des Baumes = Höhe der Wurzel
- Stufe eines Knotens = Höhe des Baumes – Tiefe des Knotens
- Hohler Baum = Baum, der seine Informationen nur in den Blättern hat (bei Suchbäumen)

Darstellungsformen:
Wortdarstellung: (Wurzel (linker Teilbaum) (rechter Teilbaum))
Verweisdarstellung (wie bei Graphen) mit Vektor, der die Nachfolger enthält

Baum à Binärer Baum
Man kann jeden Baum als binären Baum darstellen: linker Sohn = Nachfolger, rechter Sohn = Bruder. Bei binären Bäumen gilt:
Vater(i) = i/2
Linker Sohn(i) = 2i, wenn 2i < N, sonst leer
Rechter Sohn(i) = 2i+1 wenn 2i+1 < N, sonst leer

Vektordarstellung binärer Bäume: der Reihe nach, nicht vorhandene Knoten mit *,

Traversieren von binären Bäumen
Inorder (symetrische Reihenfolge): linker Teil, Wurzel, rechter Teil „runterfallen“
Preorder (Hauptreihenfolge): Wurzel, linker Teil, rechter Teil, „links vorbei“
Postorder (Nebenreihenfolge): linker Teil, rechter Teil, Wurzel, „rechts vorbei“

Bei „normalen“ Bäumen gilt folgendes:
Preorder(Baum) = Preorder(darstellender binärer Baum)
Postorder(Baum) = Inorder(darstellender binärer Baum)

Huffman
Erzeugt einen Decodierungsbaum eines optimalen Präfixcodes, Nachrichten mit Wahrscheinlichkeit, suche die 2 kleinsten Knoten, füge sie zu einem Knoten zusammen, lösche beide und füge einen neuen Knoten mit Gesamtwahrscheinlichkeit ein, O(n log n)

Hu-Tucker
Optimale vollständige sortierte hohle Suchbäume, O(n log n), 3 Phasen:
1. Schritt (Konstruktion): nur Bäume, die keine Blätter zwischendrin haben dürfen verglichen und verbunden werden. Verbinde Bäume mit geringster Wahrscheinlichkeit bei Gleichstand wird Baum mit kleinerem Index gewählt. Wiederhole so lange, bis nur ein Baum vorhanden ist.
2. Schritt (Markierung): Markiere Wurzel mit 0. Durchlaufe den Baum in Preorder und markiere Söhne um eins höher als Vater.
3. Schritt: Am Anfang wieder jeder Eintrag ein Baum. Verbinde benachbarte Bäume mit der selben Markierung, bei Gleichstand wird Baum mit kleinerem Index gewählt.

AVL-Bäume
Balancefaktor = Höhe rechter Teilbaum – Höhe linker Teilbaum, bei AVL nur 1, 0, -1, Änderung O(log n), 4 Fälle: LL, LR, RL, RR, a jeweils erster Knoten bei postorder der 2 hat

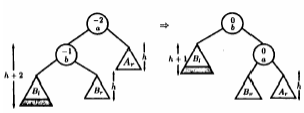

Abb. 2.26: Balancierung für LL

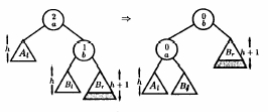

Abb. 2.27: Balancierung für RR

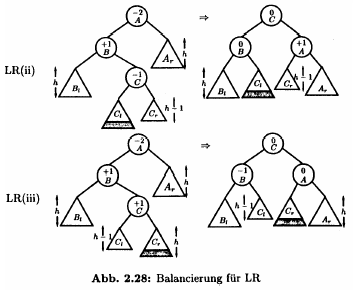

Abb. 2.28: Balancierung für LR

2-3 Bäume
jeder innere Knoten hat 2 oder 3 Söhne, alle Wege Wurzel – Blatt sind gleich lang.

B-Bäume
Listen-ähnlich, jeder innere Knoten mind. m Nachfolger, Wurzel mind. 2 Nachfolger, innere Knoten mind. m/2 Nachfolger, innere Knoten mit j Nachfolgern hat j-1 Eintragungen (immer zwischen 2)

LISP
(Atom X) liefert t wenn X kein Paar ist, sonst nil
(consp X) liefert t wenn X ein Paar ist
(null X) liefert t wenn X eine leere Liste ist
(quote X) liefert X selbst, X wird nicht ausgewertet
(car X) oder (first X) liefert das erste Element aus X
(cdr X) oder (rest X) liefert den hinteren Teil aus X
(cons X Y) bildet das Paar (X.Y)
(list a b c) bildet ein eListe (a b c)
(list-length X)liefert die Länger der Liste X
append fügt mehrere Listen zu einer zusammen, nicht sehr effizient
(reverse X) liefert eine umgedrehte Liste
(cond IF ((bedinung 1) (rumpf 1)) ELSE ((bedingung 2) (rumpf 2)) ELSE (t (rump3)))

Sortierverfahren
Sort by insertion
Ähnlich Bubblesort, fügt Elemente in eine neue Folge ein.

Bubblesort
Vergleiche das erste Element x mit dem Nachbarn y. Wenn x>y dann tausche und mache mit x weiter, ansonsten mache mit y weiter. $O(n^2)$, da n mal „durchbubblen“, bei sortierter Liste nur O(n).

Heapsort
Speichere die Folge in einem Heap, dann immer die Wurzel entnehmen. Blatt an Wurzel setzen und wieder Heap aufbauen. Eigenschaften Heap: Alle Söhne < Vater. O(n log n).

Quicksort
Oberstes Element als Pivotelement entnehmen, dass vergleichen mit Vorletztem, wenn kleiner weiter abwärts, wenn größer dann an ehemalige Stelle Pivot und von unten beginnen mit vergleich ob größer usw. Pivot nach Durchlauf am Endplatz, jetzt für beide Teilfolgen aufrufen. Optimal wenn immer halbiert wird (gut durchgemischt) O(n log n), Schlechtester Fall (vorsortiert) $O(n^2)$.

Sortieren durch Verteilen (lexikographisches Sortieren)
Wenn Alphabet vorher bekannt, sortiert „von vorne nach hinten“, stabiles Sortieren, O(m * (n + k)) m= Anzahl Stellen des Eintrages, n= Anzahl Einträge, k= Anzahl Alphabet

Auswahl- und Teilungsprobleme
Können im Mittel mit O(n) gelöst werden.

Hash
Hashfuntion berechnet aus Eintrag einen Schlüssel, Divisionsverfahren k%m, nur bei perfektem Hash ist eine eindeutige Zuordnung möglich, sonst Kollisionsbehandlung
Abgeschlossener Hash (Überlauf-Hash): Kollisionen werden als Überlaufliste angehängt,
Offener Hash: Rehash-Funktion, linarer Hash: k%m+a*i, Kollisionsketten können entstehen, besser Quotientenhash: ((k%m)+(((k/m)%(m-1))+1)*i)%m

---

Lösungsbäume
Um NP-vollständige Probleme zu lösen, kann man Lösungsbäume erstellen, statische Lösungsbäume enthalten alle Möglichkeiten: Brute Force, nicht sinnvoll, dynamische Lösungsbäume enthalten nur die Wege, die einer bestimmten Bedingung entsprechen, daher viel effektiver.

Backtracking
Preorder-Traversierung des Lösungsbaumes unter Berücksichtigung einer Bedingung, daher vorzeitiger Abbruch wenn weiter unten keine Lösung mehr möglich, Baum wird immer nur teilweise aufgebaut à Speicherplatzsparend

Branch-and-Bound
Erst alle Nachfolger eines Knotens besuchen und dann den „besten“ auswählen.