Praktikum 3 **Algorithmik SS 2007** 14.05.2007

Aufgabe 9: Das Skyline-Problem

Aufgabe 10: Union-Find-Strukturen

weitere Aufgaben folgen …

Name: ..... Matr-Nr: .....

Datum: …… Unterschrift des Dozenten (wenn bestanden): ……..……

Aufgabe 9 (Skyline-Problem)

Problembeschreibung: Gegeben sind die Koordinaten und Umrisse von mehreren rechteckigen Gebäuden in einer Stadt. Die Aufgabe besteht darin, eine zweidimensionale Skyline dieser Gebäude zu zeichnen. Das bedeutet, verdeckte Linien dürfen bei der Skyline-Zeichnung nicht sichtbar sein.

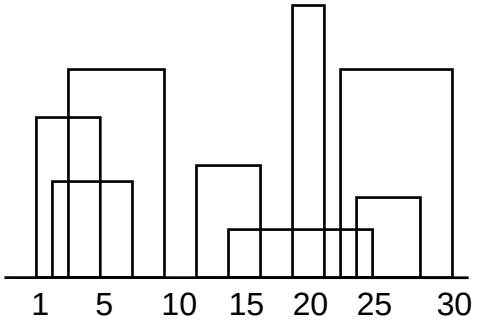


Bild 1 a): Eingabe: einzelne Gebäude als Rechtecke

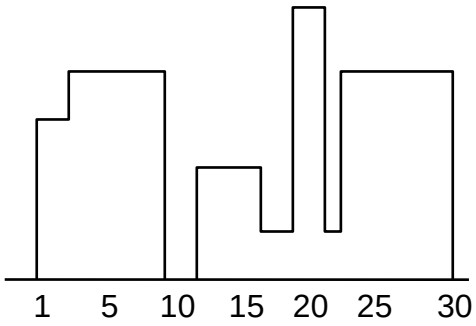


Bild 1 b): Ausgabe: Skyline

Bild 1(a) zeigt eine Eingabe für dieses Problem als Grafik. Bild 1(b) zeigt die gesuchte Ausgabe, also die Skyline. Wir lösen das Problem nur für 2-dimensionale Bilder (Lösungen für 3D-Bilder gibt es auch). Dabei nehmen wir an, dass alle Gebäude auf einer festen, gemeinsamen horizontalen Linie stehen. Ein Gebäude B_i ist repräsentiert durch ein Tripel (L_i,H_i,R_i). L_i und R_i bezeichnen die linken und rechten x-Koordinaten, H_i die Höhe des Gebäudes. Eine ***Skyline*** ist eine Liste von x-Koordinaten. Die Höhen verbinden diese Koordinaten von links nach rechts. Die Gebäude in Bild 1(a) sind durch folgende Liste repräsentiert:

(1,**11**,5), (2,**6**,7), (3,**13**,9), (12,**7**,16), (14,**3**,25), (19,**18**,22), (23,**13**,29), (24,**4**,28)

Die fett gedruckten Zahlen sind die Höhen. Die Skyline in Bild 1(b) ist durch folgende Liste exakt repräsentiert:

(1,**11**,3,**13**,9,**0**,12,**7**,16,**3**,19,**18**,22,**3**,23,**13**,29,**0**)

Eine einfache Lösung dieses Problems besteht darin, jeweils ein Gebäude zu einem Zeitpunkt der bestehenden Skyline hinzuzufügen. In diesem Fall ist die ***Induktionshypothese*** B_{n-1} einfach.

B_{n-1}: Angenommen wir wissen, wie das Problem für n-1 Gebäude zu lösen ist, und wir fügen ein weiteres Gebäude hinzu.

Für das Hinzufügen eines einzigen Gebäudes gibt es eine einfache Lösung.

Um ein Gebäude B_n der Skyline hinzuzufügen, überlagert man B_n mit der existierenden Skyline (Bild 2).

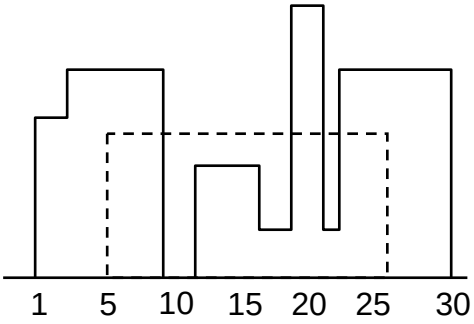


Bild 2: Integration eines Gebäudes (gestrichelt)

Sei das n-te Gebäude B_n gegeben durch (5,**9**,26). Die Skyline wird von links nach rechts geprüft, wo sich die linke Seite von B_n in die bestehenden x-Koordinaten einfügt (in diesem Fall ist die 5 die richtige x-Koordinate). Die horizontale Linie, welche die 5 überstreicht, geht von 3 bis 9, und ihre Höhe ist 13. Die Skyline wird weiter gescannt, wobei jede horizontale Linie geprüft wird; jedes mal wenn die Höhe von B_n höher als die bestehende Höhe ist, wird die Höhe von B_n übernommen. Der Algorithmus stoppt beim Erreichen einer x-Koordinate, die größer ist als die rechte Seite von B_n. In dem Beispiel wird zwischen 3 und 9 die Höhe nicht korrigiert aber zwischen 9 und 19 und ein weiteres mal zwischen 22 und 23. Die neue Skyline ergibt sich zu

(1,**11**,3,**13**,9,**9**,19,**18**,22,**9**,23,**13**,29,**0**)

Der Algorithmus ist korrekt, aber nicht effizient. Im Worst Case werden O(n) Schritte für die Integration von B_n benötigt, nämlich genau dann, wenn alle x-Koordinaten der bestehenden Gebäude geprüft werden müssen. Somit ist die Gesamtzahl der Schritte, wenn eine Skyline aus n einzelnen Gebäuden generiert wird:

$$O(n) + O(n-1) + ... + O(1) = O(n^2)$$

Es soll ein effizienterer Algorithmus nach dem Divide-and-Conquer-Prinzip entworfen werden. Statt des einfachen Induktionsprinzips - Erweiterung einer Lösung für n-1 Gebäude auf eine Lösung für n Gebäude - kann eine Lösung für n/2 Gebäude auf n Gebäude erweitert werden.

$n-1 \rightarrow n$	einfacher Algorithmus $O(n^2)$
$n/2 \rightarrow n$	Divide and Conquer Algorithmus O(?)

Divide-and-Conquer-Algorithmen teilen das Originalproblem in kleinere Teilprobleme, lösen dann jedes Teilproblem rekursiv und konstruieren die Lösung durch Mischen der Teilproblemlösungen. Meist ist es effizienter, wenn man das Originalproblem in Teilprobleme etwa gleicher Größe aufteilt.

In der Vorlesung haben wir gesehen, dass Divide-and-Conquer-Algorithmen durch Differenz-Gleichungen modelliert werden können. Mit Hilfe des in der Vorlesung bewiesenen Theorems kann die Zeitkomplexität von Differenz-Gleichungen und damit von Divide-and-Conquer-Algorithmen abgeschätzt werden.

Für die DGL $T(n) = T(n-1) + O(n)$

gilt: $T(n) = O(n^2)$

Für die Divide-and-Conquer-DGL

$T(n) = 2 * T(n/2) + O(n)$

gilt: $T(n) = O(n * \log n)$.

Aufgabenstellung

(a) Lösen Sie das Skyline-Problem nach dem Divide-and-Conquer-Prinzip, d.h. entwerfen Sie einen Algorithmus, der durch die zweite DGL modelliert wird. Beschreiben Sie die Divide- und die Conquer-Schritte ihres Algorithmus. Machen Sie Aussagen über die Zeitkomplexität Ihres Algorithmus.

(b) Programmieren Sie den Skyline-Algorithmus. Entwickeln Sie eine Experimentierumgebung, mit der man Skylines interaktiv konstruieren, modifizieren und speichern kann. Zeichnen Sie die Eingabe und die Skyline mit Hilfe der Java 2D API oder einer anderen Grafik-Bibliothek.

Hinweis: Der schnelle *Skyline*-Algorithmus hat das Divide&Conquer-Muster von *MergeSort*, d.h. es werden in jedem Induktionsschritt zwei Skylines gemischt.

Aufgabe 10 (Union-Find-Strukturen)

Viele algorithmische Problemstellungen, die mit Mengen oder Graphen zu tun haben verwenden dynamische Äquivalenzrelationen. Mit Hilfe des ADT *Union-Find* können dynamische Äquivalenzrelationen gut verwaltet werden.

Definition

Eine *Äquivalenzrelation R* auf einer Menge *S* ist eine binäre Relation auf *S*, die reflexiv, symmetrisch und transitiv ist. Das bedeutet: für alle *s, t,* und *u* in *S* gilt: *sRs*; wenn *sRt*, dann *tRs*; wenn *sRt* und *tRu*, dann *sRu*. Die Äquivalenzklasse eines Elementes *s* in *S* ist die Teilmenge von *S*, welche alle Elemente enthält, die äquivalent zu *s* sind. Diese Äquivalenzklassen bilden eine Partition von S, das bedeutet, sie sind disjunkt und ihre Vereinigung ist *S*. Das Symbol „≡“ bezeichnet eine Äquivalenzrelation.

Die Frage, um die es hier geht, lautet: Wie können Äquivalenzrelationen, die sich während der Ausführung von Algorithmen ändern, repräsentiert, modifiziert und angefragt werden. In der Vorlesung hatten wir als Beispiel den spannenden Baum mit minimalen Kosten für einen Graphen betrachtet.

Am Anfang besteht die Äquivalenzrelation aus der Gleichheitsrelation, das heißt, jedes Element ist in einer eigenen Menge. Das Problem ist, eine Folge von Operationen des folgenden Typs zu verarbeiten, wobei s_i und s_j zwei Elemente aus *S* sind:

1. IS $s_i \equiv s_j$?
2. MAKE $s_i \equiv s_j$ (wobei $s_i \equiv s_j$ nicht immer wahr sein muss)

Ob s_i und s_j in derselben Äquivalenzklasse liegen, hängt davon ab, ob vorher eine Operation MAKE $s_i \equiv s_j$ stattgefunden hat oder ob $s_i \equiv s_j$ dadurch abgeleitet werden kann, dass die Eigenschaften Reflexion, Symmetrie oder Transitivität auf Paare angewendet werden, die explizit durch MAKE äquivalent gemacht worden sind. Wird also MAKE auf Elementpaare angewendet, dann ist der nachfolgende Test $s_i \equiv s_j$? wahr.

Beispiel: Gegeben ist eine Menge S = {1, 2, 3, 4, 5}

Start der Äquivalenzklasse: {1}, {2}, {3}, {4}, {5}

1. IS 2 ≡ 4? nein
2. IS 3 ≡ 5? nein
3. MAKE 3 ≡ 5. {1}, {2}, {3, 5}, {4}
4. MAKE 2 ≡ 5. {1}, {2, 3, 5}, {4}
5. IS 2 ≡ 3? ja
6. MAKE 4 ≡ 1. {1, 4}, {2, 3, 5}
7. IS 2 ≡ 4? nein

Implementation von Union-Find-Strukturen

In Kapitel 4 der Vorlesung haben Sie die drei Operationen *Make-Set*, *Find* und *Union* des ADT *Union-Find* kennen gelernt. Um die Äquivalenz-Operationen IS und MAKE zu implementieren, verwenden Sie diese Operationen:

IS $s_i \equiv s_j$?	MAKE $s_i \equiv s_j$
$t = Find(s_i)$	$t = Find(s_i)$
$u = Find(s_j)$	$u = Find(s_j)$
$(t == u)$?	$Union(t, u)$

Um eine leere Union-Find-Struktur zu erzeugen, wird zusätzlich die Operation *Create(0)* als Abkürzung für *create(n)* eingeführt:

create(0), Make-Set(1), Make-Set(2), …, Make-Set(n)

Damit nehmen wir an, dass S={1, …, n}. Das Ergebnis ist eine Kollektion von Mengen, wobei jede ein einzelnes Element i, $1 \le i \le n$, enthält.

Sie sollen den ADT *Union Find* mit Hilfe des im Folgenden spezifizierten Baum-ADT *InTree* implementieren, wobei Sie als physikalische Datenstruktur Arrays verwenden dürfen (wie in Kap 4.3 der Vorlesung gezeigt).

- InTreeNode ***makeNode***(**int** *d*). { konstruiere einen Baum mit einem Knoten }
  Pre: keine
  Post: Wenn *x* = ***makeNode***(*d*), dann:
  x ist ein neu erzeugtes Objekt
  nodeData(*x*) = *d*;
  isRoot(*x*) = **true**;
- **boolean *isRoot***(InTreeNode *v*) { return True, wenn Knoten keinen Vater hat }
  Pre: keine
- InTreeNode ***parent***(InTreeNode *v*) { return Vater des Knotens }
  Pre: ***isRoot***(*v*) = **false**.
- **int *NodeData***(InTreeNode *v*) { return Datenwert }
  Pre: Knoten v ist kein Vorgänger von *p*.
  Post:
  1. ***nodeData***(*v*) bleibt unverändert;
  2. ***parent***(*v* = *p*);
  3. ***isRoot***(*v*) = **false**;
- **void *setNodeData***(InTreeNode *v*, **int** *d*) { setze einen Integer-Datenwert für diesen Knoten}
  Pre: keine
  Post:
  1. ***nodeData***(*v*) = *d*;
  2. ***parent***(*v*) = bleibt unverändert;
  3. ***isRoot***(*v*) = bleibt unverändert;

InTrees sind Bäume, deren Pfade von den Blättern zur Wurzel durchlaufen werden. Jeder Knoten hat somit nur eine Referenz auf seinen Vater und nicht auf seine Kinder. Für viele Anwendungen, wie z.B. die Repräsentation von Mengen, ist dieser Baum-ADT gut geeignet und ausreichend.

Jede Äquivalenzklasse oder Menge soll durch einen *InTree* repräsentiert werden. Jede Wurzel wird als kanonisches Element (d.h. als Name) für die Menge verwendet. Die Operation *r* = *Find*(*v*) findet die Wurzel des Baumes der *v* enthält und weist *r* die Wurzel zu.

Programmierung

a) Schreiben Sie eine Union-Find-Experimentier-Umgebung in Java, mit der Sie die ADT-Operationen testen können. Alle Daten sollen interaktiv eingegeben werden. Die Union-Find-ADT-Operationen sollen interaktiv aufgerufen werden. Der entsprechende *InTree* ist grafisch darzustellen. Eine Pfadverkürzung soll noch nicht vorgenommen werden.

Das folgende typische Union-Find-Programm **P** mit S = {1, …, n} besteht aus n-1 *Union*-Operationen gefolgt von m-n+1 *Find*-Operationen:

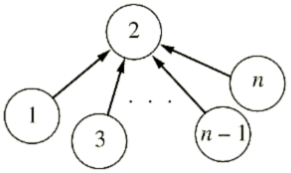
1. Union (1, 2)
2. Union (2, 3)
   .
   .
   .

n-1. Union (n-1, n)

n. Find (1)
   .
   .
   .

m. Find (1)

Baumstruktur für **P** mit *Union*



Baumstruktur für **P** bei Vereinigung nach Größe mit *wUnion*.

Beobachtung: Der *InTree*, der durch dieses Union-Find-Programm entsteht, degeneriert zu einer Liste. Es werden insgesamt n + n – 1 + (m – n + 12)n Verkettungsoperationen ausgeführt. Damit ist die WorstCase-Zeit für ein Union-Find-Programm in Ω(nm).

b) Verwenden Sie statt *Union* das Verfahren *wUnion* (weighted union), das wir in der Vorlesung als *Vereinigung nach Größe* bezeichnet haben, damit der *InTree* in der Höhe ausgeglichener wird. Stellen Sie auch hier in jedem Schritt den Baum grafisch dar, so dass man seine Struktur schrittweise verfolgen kann. Wie verändert sich die Anzahl der Link-Operationen, wenn *wUnion* statt *Union* angewendet wird?