Bonus U08 Erklärung

Gamal Hassan

1 Intuition zu Hyperwürfeln

Wir sind uns nur an geometrische Figuren im 1D, 2D oder 3D gewohnt. Daher kann es schwer sein sich einen Hyperwürfel geometrisch vorzustellen. Die gute Nachricht ist dass du es dir gar nicht vorstellen musst. Wir betrachten den Hyperwürfel nicht mehr als einen visualisierbaren Würfel, sondern einfach als eine Menge von Koordinaten, wobei jede Koordinate einem Knoten/Node des Hyperwürfels entspricht. In 1D, 2D und 3D sollte das klar sein wie die Koordinaten aussehen. Im 4D kann es aber etwas verwirrend werden sich das visuell vorzustellen. Daher betrachten wir den Hyperwürfel nun als Menge von Koordinaten. Betrachte diese visualisierung der Koordinaten:

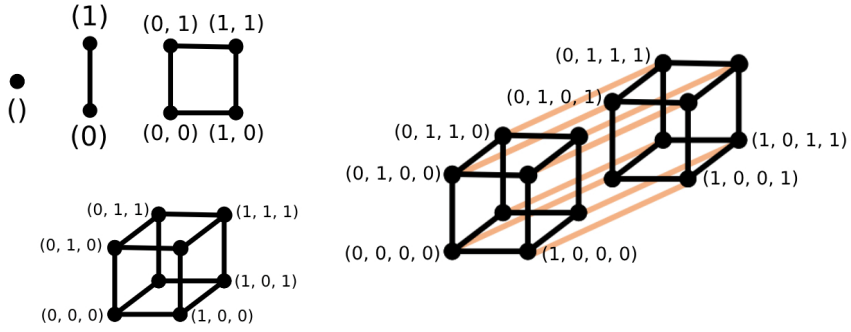

Fig. 1: Koordinaten der Knoten in 1D, 2D, 3D und 4D

Im 1D-Raum existiert nur eine Dimension, und demzufolge wird ein Punkt durch einen einzigen Wert (x) definiert. Im 2D-Raum sind es zwei Dimensionen, und somit repräsentieren zwei Werte (x, y) die Koordinaten eines Punktes. Bei 3D erweitert sich dies auf (x, y, z). Nun wird es deutlicher: Wenn wir von einer Dimension n zu einer Dimension $n' = n + 1$ übergehen, fügen wir der Koordinatenrepräsentation eine neue Dimension hinzu. Der Koordinatenvektor erhält dadurch die Länge n' anstelle von n. In diesem Prozess werden die ursprünglichen Punkte dupliziert und um die neue Dimension erweitert.

Da wir die Punkte duplizieren und nicht möchten dass zwei Kanten auf dem gleichen Punkt liegen, unterscheiden wir jedes Duplikat in der n'-ten Dimension mit 0 oder 1. Betrachten wir beispielsweise einen Punkt P im 3D-Raum P = (x, y, z), wobei $x, y, z \in \mathbb{R}$. Wenn wir in den 4D-Raum übergehen, entstehen aus dem Punkt P zwei Punkte: P = (x, y, z, 0) und P' = (x, y, z, 1). Indem wir die letzten Koordinaten unterscheiden (0 oder 1), stellen wir sicher, dass P und P' nicht identische Punkte sind. Würden wir die zwei Punkte nicht in der n'-ten Dimension unterscheiden, so hätten wir keine neue Dimension "dazu gewonnen". Denn was wäre der Sinn von einem Übergang von 2D in 3D wenn alle Punkte dann auf der z-Achse beim Punkt 0 liegen? (wir hätten einfach ein 2D Objekt im 3D Raum)

Dies bedeutet auch, dass intuitiv P' direkt "neben" P liegt da sie sich nur in einer Achse unterscheiden. Daher wird deutlich, dass zwei Knoten nur benachbart sind, wenn sie sich genau in einer Achse unterscheiden. Deshalb wurde in der Aufgabe auch gesagt dass Knoten genau dann benachbart sind wenn sie sich genau an einer Stelle im Label unterscheiden. Denn die Labels können wir einfach als die Koordinaten der Knoten interpretieren. Z.B. das Label der Koordinate (0, 0, 0, 1) wäre einfach 0001. Ich hoffe dass Hyperwürfel jetzt etwas mehr Sinn ergeben, wenn nicht dann keine Sorge. **Wichtig ist einfach dass du weisst dass zwei Nodes genau dann benachbart sind wenn sie sich im Label an genau einer Stelle unterscheiden**.

2 Approach mit Bit Manipulation

ACHTUNG: Dieser Ansatz kann verwirrend sein wenn du dich nicht besonders gut mit Bit Manipulation auskennst. Falls du lieber einen einfacheren Ansatz haben möchtest dann schau dir den Ansatz weiter unten mit Strings an!

Wir wissen nun also dass wir für jedes Label/Koordinate einen Node erstellen müssen. Die Hauptfrage ist nun wie wir alle verschiedenen Labels generieren. Ein Ansatz ist es zu bemerken dass wir die Labels als Binärsrings dartsellen können. Wir haben 2^n Nodes, wir können nun die Zahlen im Interval $I = [0, 2^n - 1]$ verwenden um die Labels zu generieren. Das funktioniert weil wir alle Zahlen im Interval I einzigartig als Binärstring der Länge n darstellen können. Z.B. ist $Binary(0) = 0000$, $Binary(1) = 0001$, $Binary(2^n - 1) = 1111$ wenn $n = 4$. Nun wird es klar dass wir die Zahlen im Interval $I = [0, 2^n - 1]$ als Labels für die Nodes verwenden können (da wir eine bijektive Abbildung auf die Labels haben mit der Funktion Binary(i)). Das heisst aber auch dass wir uns gar nicht die Mühe machen müssen alle Labels zu generieren, sondern wir können dem i-ten Node einfach die Zahl $i \in I$ als Label zuweisen. Konkret würden wir das so implementieren:

```
//m = 2^n
Node[] nodes = new Node[m];
for (int i = 0; i < m; i++){
    Node node = new Node(i, n); //erster Parameter is label, zweiter is n
    nodes[i] = node;
}
```

Nun müssen wir für jeden Node die Nachbarn in einem Node[] Array abspeichern. Da wir die Referenzen auf die Node Objekte brauchen, muss dass in einem Schritt nach der Initialisierung aller Nodes passieren (d.h. es ist nicht möglich das im Konstruktor der Nodes zu machen). Dies ist jedoch nur ein kleines Detail.

Wir wissen bereits aus der Intuition oben und aus der Aufgabenstellung, dass zwei Nodes genau dann benachbart sind wenn sie sich nur an einer Stelle im Label, bzw. den Koordinaten, unterscheiden. Daraus folgt dass jeder Node **genau** n Nachbarn hat (da die Länge des Labels n ist gibt es genau n andere Labels welche sich an einer Stelle unterscheiden). Das heisst um die Nachbarn eines Node zu finden müssen wir einfach jede Stelle im Label einmal "flippen". Haben wir z.B. das Node mit dem Label 0001 dann wären die Nachbarn: 0000, 0011, 0101, 1001. Mit dem XOR Operator können wir ein beliebiges Bit flippen. Möchten wir das Bit an der i-ten Stelle flippen, so verwenden wir ein "mask" Bitstring welcher Länge n hat, und an jeder Stelle 0 ist ausser an der i-ten Stelle (da ist es 1). Nun können wir (label XOR mask) berechnen und wir erhalten als Resultat label mit dem geflippten Bit an der i-ten Stelle. Z.B. wenn label = 0001 und mask = 1000 dann ist (label XOR mask = 1001). Bei Mask = 0001 ist (label XOR mask = 0000). Wir sehen also das XOR das Bit an der i-ten Stelle genau dann flipped wenn im mask das Bit an i-ter Stelle 1 ist. Was wir also machen müssen um die Nachbarn für ein Node zu finden ist: Wir flippen jede Stelle im Label einmal und erhalten so alle Labels welche sich genau an einer Stelle unterscheiden. Um das zu implementieren, nutzen wir aus dass die zweier Potenzen in Binärdarstellung genau eine 1 haben und der Rest 0 ist:

- $2^0 = 0001$
- $2^1 = 0010$
- $2^2 = 0100$
- $2^3 = 1000$
- usw.

Wir können müssen also alle zweier Potenzen als mask verwenden:

```
int mask = 1;
neighbors[0] = nodes[label ^ mask]; // ^ ist der XOR operator in Java
for (int i = 1; i < n; i++){
    mask *= 2; //alternativ kann man mask auch nach links shiften
    neighbors[i] = nodes[label ^ mask];
}
```

Falls das etwas verwirrend war oder du es nicht verstehst: Keine Sorge, dieser Lösungsansatz ist nicht so einfach und es ist nicht wichtig dass du die schnellste oder "schönste" Lösung findest/verstehst. Schau dir noch den anderen Lösungsansatz an.

3 Approach mit Strings

Wir können das selbe auch mit Strings lösen. Zuerst einmal haben wir das Problem dass wir die Labels irgendwie generieren müssen. Am einfachsten geht das per Rekursion: Wir starten mit dem leeren String "" und fügen dann einmal "0" und einmal "1" an den String an. So haben wir nun zwei neue Strings "0" und "1". Nun führen wir das nochmals rekursiv auf die beiden Strings aus und erhalten die vier Strings "00", "01", "10", "11". Wir führen die Rekursion so lange fort bis alle Strings die Länge n haben. Man sieht nun dass wir so alle möglichen Bitstrings der Länge n generiert haben, und somit alle Labels gefunden haben. In Java würden wir die Rekursion so aufschreiben:

```java
public static void generateLabels(List<String> labels, String str, int n){
    //Falls die Länge des aktuellen Strings = n ist, dann haben wir
    //ein valides Label gefunden ===> speichere in Liste und brich Rekursion ab.
    if (str.length() == n) {
        labels.add(str);
        return;
    }
    generateLabels(labels, str + "0", n);
    generateLabels(labels, str + "1", n);
}
```

Der Rekursionsbaum sieht dann etwa so aus:

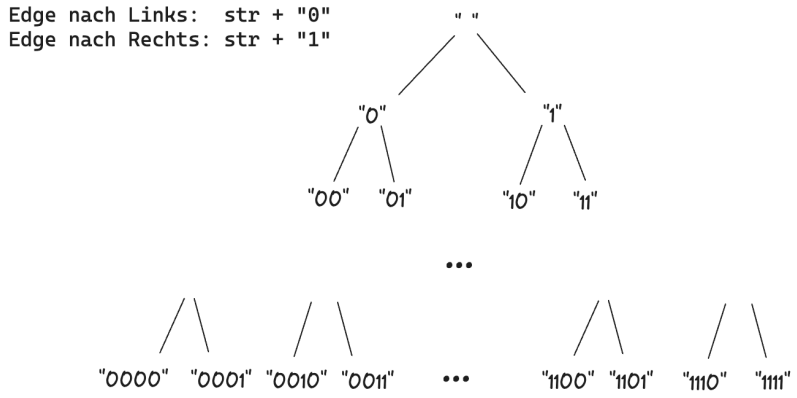

Fig. 2: Rekursionsbaum für n = 4

Nun da wir alle Labels generieren können, können wir auch alle Node Objekte erstellen:

```
//m = 2^n
Node[] nodes = new Node[m]; //Erstelle Array um alle Nodes zu speichern
List<String> labels = new ArrayList<>(); //Erstelle List um alle Labels zu speichern
generateLabels(labels, "", n); //Generiere alle möglichen Labels via Rekursion
for (int i = 0; i < m; i++){
    Node node = new Node(labels.get(i), n); //Wir geben dem i-ten Node das i-te Label.
    nodes[i] = node;
}
```

Nun da wir alle Node Objekte erstellt haben können wir schauen welche Nodes benachbart sind und welche nicht. Wie wir wissen sind zwei Nodes genau dann benachbart wenn sie sich in genau einer Stelle im Label unterscheiden. Also schreiben wir eine Helfer-Methode welche uns zurückgibt, an wie vielen Stellen sich zwei Labels unterscheiden:

```
/*
Gibt zurück, an wie vielen Stellen sich zwei Labels voneinander unterscheidet
 */
public static int distance(String label, String otherLabel){
    int count = 0;
    for (int i = 0; i < otherLabel.length(); i++){
        //Falls beide Labels an der i-ten Stelle das gleiche
        //Symbol (gleiche Zahl) haben, dann erhöhe counter um 1.
        if (label.charAt(i) != otherLabel.charAt(i)){
            count++;
        }
    }
    return count;
}
```

Wir können nun für beliebige zwei Nodes, die Methode *distance(firstNode.label, secondNode.label)* aufrufen und dann erhalten wir als Rückgabewert die Anzahl Stellen, an welchen sich die Labels der zwei Nodes unterscheiden. Das verwenden wir nun in der *createNeighbors()* Methode um herauszufinden, welche Nodes Nachbarn von einem gegebenen Node sind:

```java
public void createNeighbors(Node[] nodes){
    int foundNeighbors = 0;
    for (Node otherNode : nodes){
        //Falls sich das label von otherNode genau an
        //einer Stelle vom eigenen Node unterscheidet, dann ist es ein Nachbar
        if (distance(this.label, otherNode.label) == 1){
            neighbors[foundNeighbors] = otherNode;
            foundNeighbors++; //Inkrementiere foundNeighbors, sodass
            //der nächste Nachbar dann im nächsten Index im Array platziert wird.
        }
    }
}
```

Um herauszufinden welche Nodes Nachbarn sind müssen wir also einfach über alle Nodes iterieren und schauen ob sich das Label dort genau um eine Stelle vom eigenen Label unterscheidet. Falls das der Fall ist, fügen wir den gefundenen Node in das neighbors Array ein.

Nun ist unser Programm schon fertig! Wie du bemerkt hast ist dieser Ansatz womöglich sogar einfacher als der erste Ansatz. Daher, wenn du diesen Ansatz verstanden hast reicht das vollkommen aus! Zerbrich dir nicht den Kopf mit dem ersten Ansatz denn es gibt sehr viele mögliche Ansätze, das wichtigste ist einfach dass du einen findest der für dich Sinn macht :)

4 Noch Fragen?

Ich hoffe das hat dir ein wenig geholfen, falls du Fragen zu meinen Erklärungen hast dann melde dich bitte bei mir per E-Mail: ghassan@student.ethz.ch, oder per Discord: @ggnh
Die kompleten Lösungen zu den Ansätzen findest du auf meiner Website:
`n.ethz.ch/~ghassan`