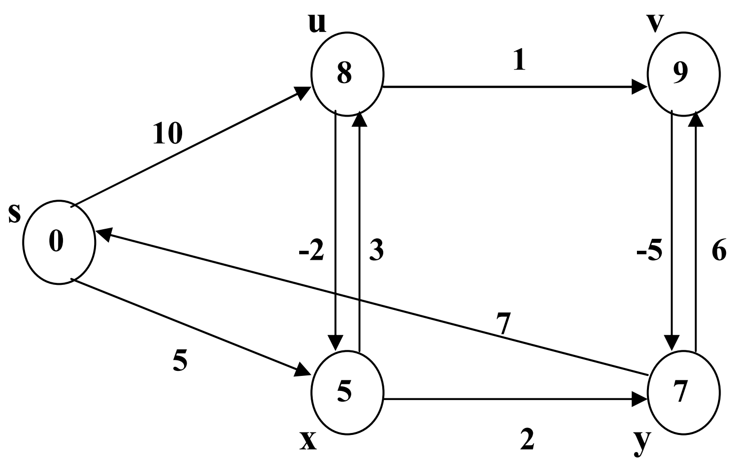
Algorithmen und Programmierung III
WS 03/04

Übung 11

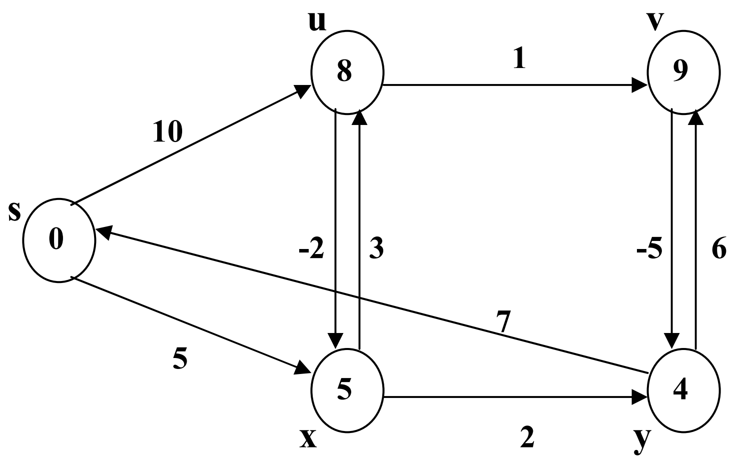
Karima Habassi **Tutor: Till Zoppke**
Ahmad Reza Nosrati

64. (6 Punkte) Konstruieren Sie einen Graphen, der auch Kanten mit negativer Länge enthält, und bei dem der Algorithmus von Dijkstra die kürzesten Wege nicht richtig bestimmt. Der Graph soll keine Kreise negativer Länge enthalten.

mit dem Algorithmus von Dijkstra würde für y den Pfad (s,x,y) gewählt, weil bei der Bearbeitung von v y schon bearbeitet worden ist, daher wird auch nicht das beste Ergebnis geliefert:



für y soll den Pfad (s,x,u,v,y) gewählt werden, und richtig sollte es so aussehen (z.B. nach Bellman-Ford-Algorithmus):



66. (9 Punkte) Betrachten Sie die Variante einer Halde, bei der jeder innere Knoten $d \geq 2$ Kinder hat. (Für d = 2 ergeben sich die gewöhnlichen Halden aus der Vorlesung.) Wie verändert sich die Laufzeit für die Methoden zugroß beziehungsweise zuklein in Abhängigkeit von d? Wie wirkt sich das auf die Operationen einfügen, entferneMin und verkleinereSchlüssel aus?
Wenn man eine solche "d-Halde" für den Algorithmus von Dijkstra verwendet, wie muss man dann d in Abhängigkeit von m und n wählen, dass man die optimale asymptotische Laufzeit erhält? Welche Laufzeit ergibt sich dann für das kürzeste-Wege-Problem? Was ergibt sich in den Extremfällen m = $\theta(n)$ und m = $\theta(n^2)$?

Um die Kinder in einer d-Halde zu finden, müssen wir die Positionsformel anpassen:

Kind(i)[1] = d(i-1)+2 // 1. Kind
Kind(i)[d] = d(i-1)+(d-1) // d. Kind

Das Selbe gilt für die Position der Eltern:

Elter(i) = $\left\lfloor \frac{i+d-2}{d} \right\rfloor$

Auswirkungen auf die Laufzeit:

zuklein Keine Änderung -> O ($\log_d$ n)

zugroß -> wir brauchen d Vergleiche, um das kleinste
Kind zu finden und anschließend ($\log_d$ n) rekursive Aufrufe, bis wir die richtige Position
Gefunden haben. -> O (d*$\log_d$ n)

Anzahl der Haldenoperationen im Dijkstra:

n-mal entferne_min aufrufen, was jeweils der Operation zugroß entspricht ->
O (d*$\log_d$ n)

auch n-mal einfügen, was jeweils der Operation zuklein entspricht ->
O ($\log_d$ n)

für die Operation zuklein müssen m-mal einen Schlüssel verkleinern ->
O (m * $\log_d$ n)

Gesamtlaufzeit = ((m+n)$\log_d$ n + n*d*$\log_d$ n)

Jetzt wollen wir die Extremfälle beim Dijkstra-Algorithmus betrachten:

(d=n)
eine unsortierte Liste mit Minimum-Zeiger

Einfügen -> O(1)

Entferne_min -> O(n)

Verkl. Schl. -> O(1)

Dijkstra -> O(n+n2+m)

1. m = $\theta(n)$

Gesamtlaufzeit für die Haldenoperationen -> $O(n+n^2+n) = O(2n+n^2)$

2. m = $\theta(n^2)$ -> $O(n+n^2+n^2) = O(n+2n^2)$
(d=1)

Wir haben eine sortierte Liste.

Einfügen -> O(n)

Entferne_min -> O(1)

Verkl.Schl. -> O(n)

Dijkstra insgesamt -> O($n^2 + n + n * m$)

1. m = $\theta(n)$

Gesamtlaufzeit für die Haldenoperationen -> $O(n^2+n+n^2) = O(2n^2+n)$

2. m = $\theta(n^2)$

Es ergibt $O(n^2+n+n^3) = O(n^3+n^2+n)$

Eine asymptotische Optimierung kann man durch Ableitung der Laufzeitformeln bekommen.

68.

(b) (5 Punkte) Berechnen Sie die Verschiebefunktion der Morse-Folge
0110100110010110100101100110100110010110011010010110100110010110.

Berechnen Sie die Verschiebefunktion der Morse-Folge
0110100110010110100101100110100110010110011010010110100110010110

t = {e, 0,
01,011,0110,01101,011010,0110100,01101001,011010011,0110100110,01101001100, usw.}

Für jede Folge aus t muss jetzt überprüft werden, ob eine Teilfolge ein Präffix in t besitzt und dies wird dann hochgzählt, genauso verfährt man mit dem Suffix. Wir haben dazu ein Java-Programm geschrieben, was uns diese aufwendige Arbeit abnimmt:

Pi = { s | s ist echter Präfix von pi }

P1 = Æ

P2 = {e}

P3 = {e, 1}

P4 = {e, 1, 11}

P5 = {e, 0, 10, 110} usw.

Für jede Menge der echten Suffixe bestimmen wir den längsten echten Präfix + 1

h1 = |Æ|+1 = 0 // Sonderfall siehe java src

h2 = |e|+1 = 1

h3 = |e|+1 = 1

h4 = |e|+1 = 1

h5 = |0|+1 = 2

usw.

Ergebnis:

H = 0 1 1 1 2 3 2 2 3 4 5 2 3 2 3 4 5 6 7 8 9 2 3 4 5 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 2 3 4 5 6 7 8 9 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

```
public class knp
{
  private int[] initNext(String pattern)
  {
    int[] next = new int[pattern.length()];
    int i = 0, j = -1;
    next[0] = -1;

    while(i < pattern.length() - 1)
    {
      while(j >= 0 && pattern.charAt(i) != pattern.charAt(j))
      {
        j = next[j];
      }

      i++; j++;
      next[i] = j;
    }

    return next;
  }

  public static void main(String[] args)
  {
    String text = "01101001100101101001011001101001
                   10010110011010010110100110010110";

    knp knp1 = new knp();
    int[] so = knp1.initNext(text);

    for (int i = 0; i < so.length; i++)
    {
      System.out.print(so[i]);
      System.out.print(" ");
    }

    System.out.println();
  }
}
```