Aufgabe 1: LaTeX-Dokument

Teilnahme-ID: ?????

Bearbeiter/-in dieser Aufgabe:
Vor- und Nachname

30. März 2023

Inhaltsverzeichnis

1 Lösungsidee **1**
1.1 Notation . . . 1
1.2 Sortieren . . . 1
1.3 PWUE-Zahl . . . 3

2 Zeit- und Speicherkomplexität **4**
2.1 Sortieren . . . 4

3 Umsetzung **4**

4 Beispiele **4**

5 Quellcode **5**

1 Lösungsidee

1.1 Notation

Die Menge der möglichen Stapel der Höhe n wird mit P_n bezeichnet. Die Möglichen Pfannkuchen-Wende-und-Ess-Operationen für einem Stapel mit n Pfannkuchen wird mit W_n bezeichnet. Die Menge der möglichen Umkehroperationen für solch einen Stapel wird mit W_n^{-1} bezeichnet. Wenn der Stapel $S \in \mathsf{P}_n$ durch die Operation $w \in \mathsf{W}_n$ verändert wird, so wird der neue Stapel $S' = wS$ bezeichnet. Operationen assoziieren nach rechts, d.h. $w_1 w_2 S = w_1(w_2 S)$. Die Funktionen A und P werden aus der Aufgabenstellung übernommen. Wird ein Stapel im Text dargestellt, dann steht der erste Pfannkuchen für den obersten, der zweite für den zweitobersten und so weiter. Pfannkuchenstapel werden entweder in Klammern durch Kommas getrennt (z.B. $(2, 7, 1, 8)$) oder als nicht getrennte Ziffern dargestellt (z.B. 2718). Bei der zweiten Notation können Pfannkuchen dann maximal die Breite 9 haben, um die Eindeutigkeit der Darstellung zu gewährleisten.

1.2 Sortieren

Die möglichen Stapel können in einem gerichteten Graphen dargestellt werden (Siehe Abbildung 1). Die Knoten des Graphen sind die Stapel, die Kanten sind die Operationen. Die Kanten sind gerichtet, denn eine PWUE-Operation wandelt einen Stapel in einen anderen um. Die Identität eines Stapels wird durch die Reihenfolge der Pfannkuchen bestimmt, nicht deren genauen Größen. Das heißt, dass zum Beispiel die Stapel $(10, 12, 3)$ und $(2, 3, 1)$ gleich sind, denn die Elemente haben die gleiche Reihenfolge. Das hängt damit zusammen, dass der Zielstapel nur durch seine aufsteigende Reihenfolge definiert ist. Equivalente Stapel können in eine kanonische Form gebracht werden, in dem die kleinste Größe durch 1, die zweitkleinste durch 2 und so weiter ersetzt werden. Dadurch ist $(2, 3, 1)$ in kanonischer Form.
Um die Operationen zu bestimmen, die einen Stapel optimal sortieren, muss ein kürzester Pfad im

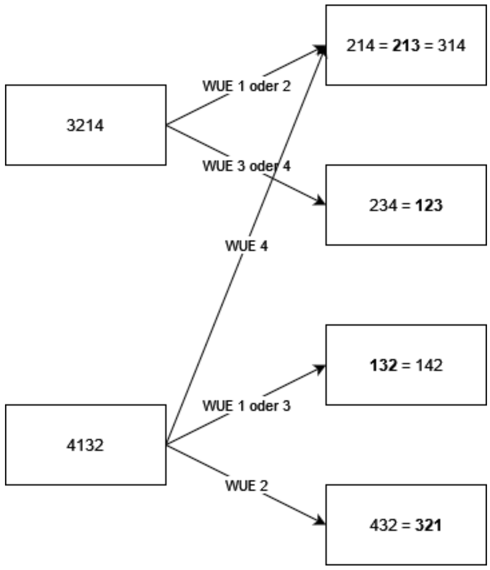

Abbildung 1: Ausschnitt aus dem Graph der Pfannkuchenstapel. Die Kanten sind beschriftet mit der zugehörigen Operation.

Graphen vom Stapel zu einem sortierten Stapel gefunden werden. Dafür lässt sich Dijkstra's Algorithmus [Dijkstra, 1959] verwenden.

procedure DIJKSTRA'S ALGORITHMUS(Stapel S)
  Intialisiere Prioritätswarteschlange Q
  Initialisiere Map V
  Initialisiere Map C
  Füge S mit Priotität 0 in Q ein
  Füge S mit Vorgänger () in V ein
  Füge S mit Kosten 0 in C ein
  while Q nicht leer **do**
    Entferne Knoten S mit der niedrigsten Priorität aus Q
    if S ist sortiert **then**
      Rekonstruiere Pfad von S zu () mit V
    end if
    for alle $w \in \mathsf{W}_n$ **do**
      $S' \leftarrow wS$
      $k \leftarrow C(S) + 1$
      if S' nicht in C oder $K < C(S')$ **then**
        Füge S' mit Priorität k in Q ein
        Füge S' mit Vorgänger S in V ein
        Füge S' mit Kosten k in C ein
      end if
    end for
  end while
end procedure

Dieser Algorithmus hat außer der Länge der Permutationskette keine Informationen über die Stapel. Da Dijkstras Algorithmus alle kürzeren erkundeten Pfade erweitert bevor ein längerer Pfad erweitert wird, ist er hier sehr langsam. Schnellere Ergebnisse lassen sich mit Hilfe vom A*-Algorithmus [Hart et al., 1968] erreichen. Dieser Algorithmus ähnelt Dijkstras Algorithmus, verwendet aber eine Heuristik, welche die Distanz zum Ziel schätzt. Die Heuristik darf die tätsächliche Entferung zum Ziel niemals überschätzen. Mit $H(S)$ als Heuristik für den Stapel S lautet der Algorithmus:

procedure A*-ALGORITHMUS(Stapel S)
  Intialisiere Prioritätswarteschlange Q
  Initialisiere Map V

```
    Initialisiere Map C
    Füge S mit Priotität 0 in Q ein
    Füge S mit Vorgänger () in V ein
    Füge S mit Kosten 0 in C ein
    while Q nicht leer do
        Entferne Knoten S mit der niedrigsten Priorität aus Q
        if S ist sortiert then
            Rekonstruiere Pfad von S zu () mit V
        end if
        for alle w ∈ W_n do
            S' ← wS
            k ← C(S) + 1 − H(S) + H(S')
            if S' nicht in C oder K < C(S') then
                Füge S' mit Priorität k in Q ein
                Füge S' mit Vorgänger S in V ein
                Füge S' mit Kosten k in C ein
            end if
        end for
    end while
end procedure
```

Eine Heuristik für das Pfannkuchensortieren ist die Anzahl der Adjazenzen [Gates and Papadimitriou, 1979]. Als Adjazenz bezeichne ich zwei Pfannkuchen die direkt nebeneinander im Stapel liegen und für die es keinen Pfannkuchen gibt, dessen Größe zwischen den beiden liegt. Mit Hilfe der Adjazenzen lässt sich eine untere Schranke für die Anzahl der Sortierschritte eines Stapels berechnen. In einer Operation können sich höchstens zwei neue Adjazenzen bilden. Weil in einer Operation sich nur die Nachbarn von zwei Pfannkuchen ändern, (nämlich des obersten, der nach unten gewendet wird, und dessen, der direkt unter dem Pfannenwender liegen bleibt) kann sich nur zwischen diesen beiden eine Adjazenz bilden. Eine weitere Adjazenz lässt sich dadurch bilden, dass der aufgegessene Pfannkuchen die Breite zwischen zwei nebeneinanderliegenden hatte, welche nach der Operation keine Pfannkuchen mit Größe zwischen ihnen haben. Als Ajdazenz wird auch gezählt, wenn der größte Pfannkuchen ganz unten liegt. Ein sortierter Stapel der Höhe n hat n Adjazenzen. Seien a_0 die Anzahl der Adjazenzen im untersuchten Stapel, h die Höhe des Stapels und n die Anzahl der Sortieroperationen. Weiterhin seien a_f und h_f die Anzahl der Adjazenzen und die Höhe des sortierten Stapels. Dann gilt:

I.	$a_f = h_f$	Adjazenzen und Höhe des Stapels müssen gleich sein
II.	$a_f \leq a_0 + 2n$	Pro Operation können höchstens zwei neue Adjazenzen entstehen
III.	$h_f = h - n$	Der Stapel wird in jedem Schritt um einen Pfannkuchen kleiner

II. und III. in I. einsetzen:

$$\begin{aligned} & a_0 + 2n \geq h - n \\ \leftrightarrow \quad & n \geq \frac{h - a_0}{3} \end{aligned}$$

Weil $n \in \mathbb{N}^+$ kann aufgerundet werden:

$$n \geq \lceil \frac{h - a_0}{3} \rceil$$

Als untere Schranke kann diese Erkenntnis als für den A*-Algorithmus geeignete Heuristik verwendet werden, mit $a(S)$ als Anzahl der Adjazenzen im Stapel S und $h(S)$ als Höhe des Stapels S:

$$H(S) = \lceil \frac{h(S) - a(S)}{3} \rceil$$

1.3 PWUE-Zahl

Die PWUE-Zahl kann rekursiv mit Hilfe von dynamischer Programmierung berechnet werden. Dafür definieren wir die Funktion $K(n, a) = \{s \in \mathsf{P}_n \mid A(s) = a\}$, die die Menge aller Stapel der Höhe n enthält, die in mindestens a Schritten sortiert werden können. Die Funktion lässt sich rekursiv berechnen:

$$\begin{aligned}K(n,a) &= \{wS, w \in \mathsf{W}_{n-1}^{-1}, S \in K(n-1,a-1)\} | \forall v \in \mathsf{W}_n : A(vwS) \geq a-1\}\\ &= \{wS, w \in \mathsf{W}_{n-1}^{-1}, S \in K(n-1,a-1)\} | \forall v \in \mathsf{W}_n : \exists b \geq a-1 : A(vwS) = b\}\\ &= \{wS, w \in \mathsf{W}_{n-1}^{-1}, S \in K(n-1,a-1)\} | \forall v \in \mathsf{W}_n : \exists b \geq a-1 : vwS \in K(n-1,b)\}\end{aligned}$$

$K(n,a)$ enthält also alle Stapel, die durch eine Umkehroperation aus Stapeln der Höhe $n-1$ mit mindestens $a-1$ Sortieroperationen entstehen können und für die keine andere Sortieroperation eine Stapel bildet, der in weniger als $a-1$ Schritten sortiert werden kann. Nach dieser Definition würde $K(n,1)$ allerdings auch die komplett sortierten Stapel enthalten, weshalb noch die Bedingung $(a > 1) \vee (s \notin K(n,0))$ ergänzt werden muss. Die Funktion $K(n,a)$ ist also definiert als

$$K(n,a) = \{wS, w \in \mathsf{W}_{n-1}^{-1}, S \in K(n-1,a-1)\} | \forall v \in \mathsf{W}_n : \exists b \geq a-1 : vwS \in K(n-1,b) \wedge ((a>1) \vee (S \notin K(n,0)))\}$$

Dass diese Definition richtig ist, lässt sich überprüfen durch die Substitution $A(S) = k, S \in \mathsf{P}_n \iff (\forall w \in \mathsf{W}_n : A(ws) \geq k-1) \wedge (\exists w \in \mathsf{W}_n : A(ws) = k-1)$. Um jetzt die PWUE-Zahl zu berechnen, muss nur noch die Funktion $K(n,a)$ für alle a berechnet werden und überprüft werden, ob sie Elemente enthält. Da jeder Stapel $S \in \mathsf{P}_n$ in $\lceil \frac{n}{1.5} \rceil$ Schritten sortiert werden kann, reicht es aus, die Funktion $K(n,a)$ für alle $\lceil \frac{n}{1.5} \rceil$ zu berechnen. Damit lässt sich auch $\exists b \geq a-1 : vws \in K(n-1,b)$ durch $\exists \lceil \frac{n}{1.5} \rceil \geq b \geq a-1 : vwS \in K(n-1,b)$ ersetzen wodurch nicht unendlich viele Werte für b ausprobiert werden müssen. Zuletzt muss noch ein Ende der Rekursion eingeführt werden, wir setzen $K(n,0) = \{(1,\ldots,n)\}$, denn ein sortierter Stapel kann in 0 Schritten sortiert werden.

2 Zeit- und Speicherkomplexität

2.1 Sortieren

Sowohl Dijkstra's Algorithmus als auch der A*-Algorithmus haben für einen Graphen mit Knoten V und Kanten E eine Zeitkomplexität von $\mathcal{O}(|E| + |V| \cdot log|V|)$, wenn als Prioritätswarteschlange ein binärer Heap verwendet wird. Wenn wir einen Stapel der Höhe h haben, sind die Knoten des zu untersuchenden Graphs $V = \dot\bigcup_{n=1}^{h-1} \mathsf{P}_n$. Da es $n!$ Permutationen von n Elementen gibt, ist $|V| = \Sigma_{n=1}^{h-1} n!$. Diese Summe ist in $\mathcal{O}(h!)$:

$$\begin{aligned}\Sigma_{n=1}^{h-1} n! &= (h-1)! \cdot (1 + \frac{1}{h-1} + \frac{1}{(h-1)(h-2)} + \ldots + \frac{1}{(h-1)!})\\ &= (h-1)! \cdot (1 + \mathcal{O}(1))\\ &= \mathcal{O}(h!)\end{aligned}$$

Von einem Knoten können maximal h Kanten ausgehen, denn das ist die Anzahl möglicher PWUE-Operationen des Stapels mit Höhe h. Es ist also $|E| = \mathcal{O}(h) \cdot \mathcal{O}(h!)$.

3 Umsetzung

Zur Lösung der ersten Aufgabe habe ich Python verwendet. Für die zweite Aufgabe habe ich Java verwendet.

4 Beispiele

Genügend Beispiele einbinden! Die Beispiele von der BwInf-Webseite sollten hier diskutiert werden, aber auch eigene Beispiele sind sehr gut – besonders wenn sie Spezialfälle abdecken. Aber bitte nicht 30 Seiten Programmausgabe hier einfügen!

5 Quellcode

```
from queue import PriorityQueue

# finds the shortest path from start node to a node that fullfills target_pred. returns the path
def a_star(start_node, target_pred, adj_func, cost_func, heur_func, count_steps=False):
    if count_steps:
        steps = 0
    i = 0
    queue = PriorityQueue()
    queue.put((0, heur_func(start_node), i, start_node))
    prev = {start_node: None}
    cost = {start_node: 0 + heur_func(start_node)}
    while not queue.empty():
        if count_steps:
            steps += 1
        _, _, _, node = queue.get()
        if target_pred(node):
            if count_steps:
                return reconstruct_path(node, prev), steps
            return reconstruct_path(node, prev)
        for adj_node in adj_func(node):
            new_cost = cost[node] - heur_func(node) + cost_func(node, adj_node) + heur_func(adj_node)
            if adj_node not in cost or new_cost < cost[adj_node]:
                i -= 1
                cost[adj_node] = new_cost
                queue.put((new_cost, heur_func(node), i, adj_node))
                prev[adj_node] = node

def reconstruct_path(node, prev):
    path = [node]
    while prev[node] is not None:
        node = prev[node]
        path.append(node)
    return list(reversed(path))
```

a_star.py

```
import math
from a_star import a_star

# Pfannkuchenstapel umdrehen und Pfannkuchen essen.
def flip(arr, k):
    return arr[: k - 1][::-1] + arr[k:]


# Gibt alle moeglichen naechsten Reihenfolgen zurueck.
def next_arrs(arr):
    for i in range(1, len(arr) + 1):
        yield normalize(flip(arr, i))


# Zaehlt, wie viele aufeinanderfolgende Pfannkuchen nebeneinander liegen.
def count_adj(arr):
    adj = 0
    for i in range(1, len(arr)):
        if arr[i] - arr[i - 1] in (1, -1):
            adj += 1
    if arr[-1] == max(arr):
        adj += 1
    return adj


# Veraendert die Zahlen in der Liste so, dass sie in [0, ..., n-1] liegen,
# wobei die Reihenfolge erhalten bleibt.
def normalize(arr):
    order = sorted(arr)
    return tuple(order.index(x) for x in arr)


# Naehert die minimale Anzahl von flips()s mit count_adj() an.
def heuristic(arr):
    return math.ceil((len(arr) - count_adj(normalize(arr))) / 3)
```

```python


# prueft, ob die Liste in der richtigen Reihenfolge ist.
def is_sorted(arr):
    return all(arr[i] <= arr[i + 1] for i in range(len(arr) - 1))


# Gibt die Optimale Reihenfolge von flip()s zurueck, um die Liste zu sortieren.
def least_flips(arr, count_steps=False):
    return a_star(normalize(arr), is_sorted, next_arrs, lambda a, b: 1, heuristic, count_steps)


def find_flip(pre, post):
    for i in range(1, len(pre) + 1):
        if normalize(flip(pre, i)) == post:
            return i


def main():
    path = input("Pfad:␣")
    with open(path) as f:
        n_pancakes = int(f.readline())
        pancakes = tuple(int(x) for x in f.readlines())
    pancakes = normalize(pancakes)
    steps = least_flips(pancakes)
    print(steps)
    print(len(steps) - 1, "Schritte")
    print(len(pancakes), "Pfannkuchen")
    print(heuristic(pancakes), "heuristik")
    print(math.ceil(len(pancakes) / 1.5), "upper␣bound")
    print(len(pancakes) / 2, "lower␣bound")

    print("--␣schritte␣--")
    pre = None
    not_normalized = steps[0]
    print(not_normalized)
    for step in steps:
        if pre is not None:
            ix = find_flip(pre, step)
            print("Wende␣erste", ix)
            not_normalized = flip(not_normalized, ix)
            print(not_normalized)
        pre = step


if __name__ == "__main__":
    main()
```

least_flips.py

```java
import java.util.Arrays;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Optional;
import java.util.Scanner;
import java.util.Set;

public class Pwue {
    static class IntPair implements Comparable<IntPair> {
        private int num1;
        private int num2;


        public IntPair(int key, int value) {
            this.num1 = key;
            this.num2 = value;
        }

        public int first() {
            return num1;
        }

```

```
        public int second() {
            return num2;
        }

        @Override
        public boolean equals(Object o) {
            if (this == o)
                return true;
            if (o == null || getClass() != o.getClass())
                return false;
            IntPair pair = (IntPair) o;
            return (num1 == pair.num1 && num2 == pair.num2);
        }

        @Override
        public int hashCode() {
            int hash = 17;
            hash = hash * 31 + num1;
            hash = hash * 31 + num2;
            return hash;
        }

        @Override
        public String toString() {
            return "(" + num1 + ", " + num2 + ")";
        }

        @Override
        public int compareTo(Pwue.IntPair o) {
            if (num1 < o.num1) {
                return -1;
            }
            if (num1 > o.num1) {
                return 1;
            }
            if (num2 < o.num2) {
                return -1;
            }
            if (num2 > o.num2) {
                return 1;
            }
            return 0;
        }

        public void setFirst(int num1) {
            this.num1 = num1;
        }

        public void setSecond(int num2) {
            this.num2 = num2;
        }

        public void swap() {
            int temp = num1;
            num1 = num2;
            num2 = temp;
        }
    }

    static Integer[] flipOp(Integer[] a, int i) {
        Integer[] b = new Integer[a.length - 1];
        for (int j = 0; j < i - 1; j++) {
            b[j] = a[i - j - 2];
        }
        for (int j = i; j < a.length; j++) {
            b[j - 1] = a[j];
        }
        return normalize(b);
    }

    static Integer[] revFlipOp(Integer[] a, int i, int n) {
        i--;
        Integer[] b = new Integer[a.length + 1];
```

```
        for (int j = 0; j < i; j++) {
            if (a[i - j - 1] >= n) {
                b[j] = a[i - j - 1] + 1;
            } else {
                b[j] = a[i - j - 1];
            }
        }
        b[i] = n;
        for (int j = i; j < a.length; j++) {
            if (a[j] >= n) {
                b[j + 1] = a[j] + 1;
            } else {
                b[j + 1] = a[j];
            }
        }
        return normalize(b);
    }

    static Integer[] normalize(Integer[] a) {
        IntPair[] b = new IntPair[a.length];
        for (int i = 0; i < a.length; i++) {
            b[i] = new IntPair(a[i], i);
        }
        Arrays.sort(b);
        for (int i = 0; i < a.length; i++) {
            b[i].setFirst(i);
            b[i].swap();
        }
        Arrays.sort(b);
        Integer[] c = new Integer[a.length];
        for (int i = 0; i < a.length; i++) {
            c[i] = b[i].second();
        }
        return c;
    }

    static Integer[] allFlipOps(int n) {
        Integer[] a = new Integer[n];
        for (int i = 0; i < n; i++) {
            a[i] = i + 1;
        }
        return a;
    }

    static IntPair[] allRevFlipOps(int n) {
        IntPair[] a = new IntPair[n * (n + 1)];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j <= n; j++) {
                a[i * (n + 1) + j] = new IntPair(i + 1, j);
            }
        }
        return a;
    }

    static Integer[] range(int n) {
        Integer[] a = new Integer[n];
        for (int i = 0; i < n; i++) {
            a[i] = i;
        }
        return a;
    }

    static Map<IntPair, Set<List<Integer>>> memo = new HashMap<>();


    static Set<List<Integer>> k(int n, int a, int depth) {
        IntPair key = new IntPair(n, a);
        if (memo.containsKey(key)) {
            return memo.get(key);
        }
        if (a == 0) {
            Set<List<Integer>> result = new HashSet<>();
            result.add(Arrays.asList(range(n)));
```

```
            memo.put(key, result);
            return result;
        }
        System.out.println("|␣".repeat(depth) + "|␣k(" + n + ",␣" + a + ")␣"
                + ((int) Math.ceil(n / 1.5) - a + 1) * n * (n + 1)
                        * k(n - 1, a - 1, depth + 1).size()
                + "␣Its");
        HashSet<List<Integer>> result = new HashSet<>();
        for (IntPair rFlip : allRevFlipOps(n)) {

            for (List<Integer> seqL : k(n - 1, a - 1, depth + 1)) {
                Integer[] seq = seqL.toArray(new Integer[0]);
                Integer[] rFlipped = revFlipOp(seq, rFlip.first(), rFlip.second());
                if (!(a > 1 || !Arrays.equals(rFlipped, range(n)))) {
                    continue;
                }
                boolean r1 = true;
                boolean r2 = false;
                for (Integer flip : allFlipOps(n)) {
                    for (int b = a - 1; b < Math.ceil(n / 1.5); b++) {
                        r2 = false;
                        if (k(n - 1, b, depth + 1)
                                .contains(Arrays.asList(flipOp(rFlipped, flip)))) {
                            r2 = true;
                            break;
                        }
                    }
                    r1 = r1 && r2;
                    if (!r1) {
                        break;
                    }
                }
                if (r1) {
                    result.add(Arrays.asList(rFlipped));
                }
            }
        }
        System.out.println("|␣".repeat(depth) + "|␣k(" + n + ",␣" + a + ")␣Fertig␣␣␣␣␣␣␣␣␣␣");
        memo.put(key, result);
        return result;
    }

    static Optional<Integer[]> kHasSolution(int n, int a) {
        for (IntPair rFlip : allRevFlipOps(n)) {
            for (List<Integer> seqL : k(n - 1, a - 1, 0)) {
                Integer[] seq = seqL.toArray(new Integer[0]);
                Integer[] rFlipped = revFlipOp(seq, rFlip.first(), rFlip.second());
                if (!(a > 1 || !Arrays.equals(rFlipped, range(n)))) {
                    continue;
                }

                boolean r1 = true;
                boolean r2 = false;
                for (Integer flip : allFlipOps(n)) {
                    for (int b = a - 1; b < Math.ceil(n / 1.5); b++) {
                        r2 = false;
                        if (k(n - 1, b, 0).contains(Arrays.asList(flipOp(rFlipped, flip)))) {
                            r2 = true;
                            break;
                        }
                    }
                    r1 = r1 && r2;
                    if (!r1) {
                        break;
                    }
                }
                if (r1) {
                    return Optional.of(rFlipped);
                }
            }
        }
        return Optional.empty();
    }
```

```java

    public static void main(String[] args) {
        System.out.println(Arrays.deepToString(allRevFlipOps(5)));
        Scanner scanner = new Scanner(System.in);
        System.out.print("n: ");
        int n = scanner.nextInt();
        scanner.close();
        for (int a = (int) Math.ceil(n / 1.5); a > 0; a--) {
            Optional<Integer[]> result = kHasSolution(n, a);
            if (result.isPresent()) {
                System.out.println("max a: " + a);
                System.out.println(Arrays.deepToString(result.get()));
                break;
            }
        }

    }
}
```

Pwue.java

Literatur

[Dijkstra, 1959] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271.

[Gates and Papadimitriou, 1979] Gates, W. H. and Papadimitriou, C. H. (1979). Bounds for sorting by prefix reversal. *Discrete mathematics*, 27(1):47–57.

[Hart et al., 1968] Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107.