

Aufgabe 1: L^AT_EX-Dokument

Teilnahme-ID: ?????

Bearbeiter/-in dieser Aufgabe:
Vor- und Nachname

6. März 2023

Inhaltsverzeichnis

1	Lösungsidee	1
1.1	Notation	1
1.2	Sortieren	1
1.3	PWUE-Zahl	3
2	Umsetzung	4
3	Beispiele	4
4	Quellcode	4

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 $S \in P_n$ durch die Operation $w \in W_n$ 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 Graphen vom Stapel zu einem sortierten Stapel gefunden werden. Dafür lässt sich Dijkstra's Algorithmus [Dijkstra, 1959] verwenden.

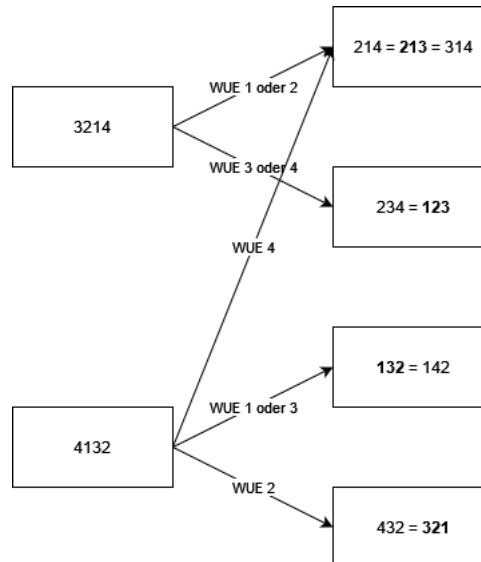


Abbildung 1: Ausschnitt aus dem Graph der Pfannkuchenstapel. Die Kanten sind beschriftet mit der zugehörigen Operation.

```

procedure DIJKSTRA'S ALGORITHMUS(Stapel  $S$ )
  Initialisiere Prioritätswarteschlange  $Q$ 
  Initialisiere Map  $V$ 
  Initialisiere Map  $C$ 
  Füge  $S$  mit Priorität 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  $w \in W_n$  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 tatsächliche Entfernung zum Ziel niemals überschätzen. Mit $H(S)$ als Heuristik für den Stapel S lautet der Algorithmus:

```

procedure A*-ALGORITHMUS(Stapel  $S$ )
  Initialisiere Prioritätswarteschlange  $Q$ 
  Initialisiere Map  $V$ 
  Initialisiere Map  $C$ 
  Füge  $S$  mit Priorität 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  $w \in W_n$  do
     $S' \leftarrow wS$ 
     $k \leftarrow C(S) + 1 - H(S) + H(S')$ 
    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 aufgegebene Pfannkuchen die Breite zwischen zwei nebeneinanderliegenden hatte, welche nach der Operation keine Pfannkuchen mit Größe zwischen ihnen haben. Als Adjazenz 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:

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

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 $K(n, a) = \{s \in P_n \mid A(s) = a\}$, die die Menge aller Stapel der Höhe n enthält,

die in mindestens a Schritten sortiert werden können. Die Funktion lässt sich rekursiv berechnen:

$$\begin{aligned} K(n, a) &= \{wS, w \in W_{n-1}^{-1}, S \in K(n-1, a-1)\} \mid \forall v \in W_n : A(vwS) \geq a-1\} \\ &= \{wS, w \in W_{n-1}^{-1}, S \in K(n-1, a-1)\} \mid \forall v \in W_n : \exists b \geq a-1 : A(vwS) = b\} \\ &= \{wS, w \in W_{n-1}^{-1}, S \in K(n-1, a-1)\} \mid \forall v \in W_n : \exists b \geq a-1 : vwS \in K(n-1, b)\} \end{aligned}$$

$K(n, a)$ 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 einen 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

$$K(n, a) = \{wS, w \in W_{n-1}^{-1}, S \in K(n-1, a-1)\} \mid \forall v \in W_n : \exists b \geq a-1 : vwS \in K(n-1, b) \wedge ((a > 1) \vee (S \notin K(n, 0)))\}$$

Dass diese Definition richtig ist, lässt sich überprüfen durch die Substitution $A(S) = k, S \in P_n \iff (\forall w \in W_n : A(ws) \geq k-1) \wedge (\exists w \in W_n : A(ws) = k-1)$. 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 $S \in P_n$ 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 $K(n, 0) = \{(1, \dots, n)\}$, denn ein sortierter Stapel kann in 0 Schritten sortiert werden.

2 Umsetzung

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

3 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!

4 Quellcode

```

1 from queue import PriorityQueue

3 # finds the shortest path from start node to a node that fullfills target_pred. returns the path
4 def a_star(start_node, target_pred, adj_func, cost_func, heur_func, count_steps=False):
5     if count_steps:
6         steps = 0
7     i = 0
8     queue = PriorityQueue()
9     queue.put((0, heur_func(start_node), i, start_node))
10    prev = {start_node: None}
11    cost = {start_node: 0 + heur_func(start_node)}
12    while not queue.empty():
13        if count_steps:
14            steps += 1
15        _, _, _, node = queue.get()
16        if target_pred(node):
17            if count_steps:
18                return reconstruct_path(node, prev), steps
19            return reconstruct_path(node, prev)
20        for adj_node in adj_func(node):
21            new_cost = cost[node] - heur_func(node) + cost_func(node, adj_node) + heur_func(adj_node)
22            if adj_node not in cost or new_cost < cost[adj_node]:
23                i -= 1

```

```

        cost[adj_node] = new_cost
25         queue.put((new_cost, heur_func(node), i, adj_node))
        prev[adj_node] = node
27
def reconstruct_path(node, prev):
29     path = [node]
    while prev[node] is not None:
31         node = prev[node]
        path.append(node)
33     return list(reversed(path))

a_star.py

import math
2 from a_star import a_star

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

8
# Gibt alle moeglichen naechsten Reihenfolgen zurueck.
10 def next_arrs(arr):
    for i in range(1, len(arr) + 1):
12         yield normalize(flip(arr, i))

14
# Zaehlt, wie viele aufeinanderfolgende Pfannkuchen nebeneinander liegen.
16 def count_adj(arr):
    adj = 0
18     for i in range(1, len(arr)):
        if arr[i] - arr[i - 1] in (1, -1):
20         adj += 1
    if arr[-1] == max(arr):
22         adj += 1
    return adj
24

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

32
# Naehert die minimale Anzahl von flips()s mit count_adj() an.
34 def heuristic(arr):
    return math.ceil((len(arr) - count_adj(normalize(arr))) / 3)
36

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

42
# Gibt die Optimale Reihenfolge von flip()s zurueck, um die Liste zu sortieren.
44 def least_flips(arr, count_steps=False):
    return a_star(normalize(arr), is_sorted, next_arrs, lambda a, b: 1, heuristic, count_steps)
46

48 def find_flip(pre, post):
    for i in range(1, len(pre) + 1):
50         if normalize(flip(pre, i)) == post:
            return i
52

54 def main():
    path = input("Pfad: ")
56     with open(path) as f:
        n_pancakes = int(f.readline())
58         pancakes = tuple(int(x) for x in f.readlines())
        pancakes = normalize(pancakes)
        steps = least_flips(pancakes)
60         print(steps)

```

```

62     print(len(steps) - 1, "Schritte")
        print(len(pancakes), "Pfannkuchen")
64     print(heuristic(pancakes), "heuristik")
        print(math.ceil(len(pancakes) / 1.5), "upper_bound")
66     print(len(pancakes) / 2, "lower_bound")

68     print("-- schritte --")
        pre = None
        not_normalized = steps[0]
        print(not_normalized)
72     for step in steps:
            if pre is not None:
74                 ix = find_flip(pre, step)
                    print("Wende_erste", ix)
76                 not_normalized = flip(not_normalized, ix)
                    print(not_normalized)
78                 pre = step

80
81 if __name__ == "__main__":
82     main()

```

least_flips.py

```

import java.util.Arrays;
2 import java.util.HashMap;
import java.util.HashSet;
4 import java.util.List;
import java.util.Map;
6 import java.util.Optional;
import java.util.Scanner;
8 import java.util.Set;

10 public class Pwue {
    static class IntPair implements Comparable<IntPair> {
12         private int num1;
            private int num2;
14
16         public IntPair(int key, int value) {
            this.num1 = key;
18             this.num2 = value;
        }

20         public int first() {
22             return num1;
        }

24         public int second() {
26             return num2;
        }

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

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

46         @Override
48         public String toString() {
            return "(" + num1 + ", " + num2 + ")";
50     }
}

```

```

52     @Override
53     public int compareTo(Pwue.IntPair o) {
54         if (num1 < o.num1) {
55             return -1;
56         }
57         if (num1 > o.num1) {
58             return 1;
59         }
60         if (num2 < o.num2) {
61             return -1;
62         }
63         if (num2 > o.num2) {
64             return 1;
65         }
66         return 0;
67     }
68
69     public void setFirst(int num1) {
70         this.num1 = num1;
71     }
72
73     public void setSecond(int num2) {
74         this.num2 = num2;
75     }
76
77     public void swap() {
78         int temp = num1;
79         num1 = num2;
80         num2 = temp;
81     }
82 }
83
84 static Integer[] flipOp(Integer[] a, int i) {
85     Integer[] b = new Integer[a.length - 1];
86     for (int j = 0; j < i - 1; j++) {
87         b[j] = a[i - j - 2];
88     }
89     for (int j = i; j < a.length; j++) {
90         b[j - 1] = a[j];
91     }
92     return normalize(b);
93 }
94
95 static Integer[] revFlipOp(Integer[] a, int i, int n) {
96     i--;
97     Integer[] b = new Integer[a.length + 1];
98     for (int j = 0; j < i; j++) {
99         if (a[i - j - 1] >= n) {
100             b[j] = a[i - j - 1] + 1;
101         } else {
102             b[j] = a[i - j - 1];
103         }
104     }
105     b[i] = n;
106     for (int j = i; j < a.length; j++) {
107         if (a[j] >= n) {
108             b[j + 1] = a[j] + 1;
109         } else {
110             b[j + 1] = a[j];
111         }
112     }
113     return normalize(b);
114 }
115
116 static Integer[] normalize(Integer[] a) {
117     IntPair[] b = new IntPair[a.length];
118     for (int i = 0; i < a.length; i++) {
119         b[i] = new IntPair(a[i], i);
120     }
121     Arrays.sort(b);
122     for (int i = 0; i < a.length; i++) {
123         b[i].setFirst(i);

```

```

124         b[i].swap();
125     }
126     Arrays.sort(b);
127     Integer[] c = new Integer[a.length];
128     for (int i = 0; i < a.length; i++) {
129         c[i] = b[i].second();
130     }
131     return c;
132 }

133
134 static Integer[] allFlipOps(int n) {
135     Integer[] a = new Integer[n];
136     for (int i = 0; i < n; i++) {
137         a[i] = i + 1;
138     }
139     return a;
140 }

141
142 static IntPair[] allRevFlipOps(int n) {
143     IntPair[] a = new IntPair[n * (n + 1)];
144     for (int i = 0; i < n; i++) {
145         for (int j = 0; j <= n; j++) {
146             a[i * (n + 1) + j] = new IntPair(i + 1, j);
147         }
148     }
149     return a;
150 }

151
152 static Integer[] range(int n) {
153     Integer[] a = new Integer[n];
154     for (int i = 0; i < n; i++) {
155         a[i] = i;
156     }
157     return a;
158 }

159
160 static Map<IntPair, Set<List<Integer>>> memo = new HashMap<>();

161
162
163 static Set<List<Integer>> k(int n, int a, int depth) {
164     IntPair key = new IntPair(n, a);
165     if (memo.containsKey(key)) {
166         return memo.get(key);
167     }
168     if (a == 0) {
169         Set<List<Integer>> result = new HashSet<>();
170         result.add(Arrays.asList(range(n)));
171         memo.put(key, result);
172         return result;
173     }
174     System.out.println("|".repeat(depth) + "| k(" + n + ", " + a + ") |"
175         + ((int) Math.ceil(n / 1.5) - a + 1) * n * (n + 1)
176         * k(n - 1, a - 1, depth + 1).size()
177         + " | Its");
178     HashSet<List<Integer>> result = new HashSet<>();
179     for (IntPair rFlip : allRevFlipOps(n)) {
180
181         for (List<Integer> seqL : k(n - 1, a - 1, depth + 1)) {
182             Integer[] seq = seqL.toArray(new Integer[0]);
183             Integer[] rFlipped = revFlipOp(seq, rFlip.first(), rFlip.second());
184             if (!(a > 1 || !Arrays.equals(rFlipped, range(n)))) {
185                 continue;
186             }
187             boolean r1 = true;
188             boolean r2 = false;
189             for (Integer flip : allFlipOps(n)) {
190                 for (int b = a - 1; b < Math.ceil(n / 1.5); b++) {
191                     r2 = false;
192                     if (k(n - 1, b, depth + 1)
193                         .contains(Arrays.asList(flipOp(rFlipped, flip)))) {
194                         r2 = true;
195                         break;
196                     }

```



```

198         r1 = r1 && r2;
199         if (!r1) {
200             break;
201         }
202     }
203     if (r1) {
204         result.add(Arrays.asList(rFlipped));
205     }
206 }
207
208 System.out.println("|_|".repeat(depth) + "|_|k(" + n + ",_|" + a + ")_|Fertiguuuuuuuuuu");
209 memo.put(key, result);
210 return result;
211 }
212
213 static Optional<Integer[]> kHasSolution(int n, int a) {
214     for (IntPair rFlip : allRevFlipOps(n)) {
215         for (List<Integer> seqL : k(n - 1, a - 1, 0)) {
216             Integer[] seq = seqL.toArray(new Integer[0]);
217             Integer[] rFlipped = revFlipOp(seq, rFlip.first(), rFlip.second());
218             if (!(a > 1 || !Arrays.equals(rFlipped, range(n)))) {
219                 continue;
220             }
221
222             boolean r1 = true;
223             boolean r2 = false;
224             for (Integer flip : allFlipOps(n)) {
225                 for (int b = a - 1; b < Math.ceil(n / 1.5); b++) {
226                     r2 = false;
227                     if (k(n - 1, b, 0).contains(Arrays.asList(flipOp(rFlipped, flip)))) {
228                         r2 = true;
229                         break;
230                     }
231                 }
232                 r1 = r1 && r2;
233                 if (!r1) {
234                     break;
235                 }
236             }
237             if (r1) {
238                 return Optional.of(rFlipped);
239             }
240         }
241     }
242     return Optional.empty();
243 }
244
245 public static void main(String[] args) {
246     System.out.println(Arrays.deepToString(allRevFlipOps(5)));
247     Scanner scanner = new Scanner(System.in);
248     System.out.print("n:");
249     int n = scanner.nextInt();
250     scanner.close();
251     for (int a = (int) Math.ceil(n / 1.5); a > 0; a--) {
252         Optional<Integer[]> result = kHasSolution(n, a);
253         if (result.isPresent()) {
254             System.out.println("max_|a:_|" + a);
255             System.out.println(Arrays.deepToString(result.get()));
256             break;
257         }
258     }
259 }
260
261 }

```

Pwue.java

Literatur

[Dijkstra, 1959] Dijkstra, E. (1959). A note on two problems in connexion with graphs. *numeriske matematik*, 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.