Eine Lösung für das Problem Reversi, Version 1.0

wdjpng / Lukas Münzel

05. Oktober 2019

1 Wie funktioniert mein Algorithmus?

Es werden zwei Indexe, i_s und i_e, definiert . Nun wird i_s auf 0 und i_e auf die Länge des Inputarrays weniger eins gesetzt. Die Summe s der minimal benötigten Operationen s wird auf null gesetzt. Sei zudem n die Länge des Inputarrays, n_l die Anzahl Steine, welche links hinzugefügt wurden und n_r die Anzahl Steine, welche rechts hinzugefügt wurden. Zu Beginn ist $n_r = n_l = 0$.

1. i_s wird um eins erhöht. Wenn nun $input[i_s - 1] \neq input[i_s]$, wird s um $i_s + s + n_l$ und dann n_l um eins erhöht. Dies entspricht dem hinzulegen eines Steines an das linke Ende. Wenn nach diesen Schritten $i_s = i_e$ ist, wird s als Lösung zurückgegeben

2. i_e wird um eins niedriger gemacht. Wenn nun $input[i_e + 1] \neq input[i_e]$, wird s um $n - i_e + n_r$ und dann n_r um eins erhöht. Dies entspricht dem hinzulegen eines Steines an das rechte Ende. Wenn nach diesen Schritten $i_s = i_e$ ist, wird s als Lösung zurückgegeben.

3. Schritte 1. und 2. werden so lange wiederhohlt, bis s ausgegeben wird. Wenn in der letzten Ausführung der Schritte $input[i_s - 1] \neq input[i_s]$ war, wird 1. in der nächsten Ausführung der Schritte nicht ausgeführt. Umgekehrt wird, wenn $input[i_e + 1] \neq input[i_e]$ war, wird 2. in der nächsten Ausführung der beiden Schritte nicht ausgeführt

2 Argumentation, weshalb der Algorithmus korrekt ist

Für jede Veränderung in der Farbe des Inputs muss ein Stein auf eine Seite hinzugefügt werden, da pro hinzugefügtem Stein ein Stein mehr die selbe Farbe wie andere Steine bekommt. Nun ist die Frage, weshalb es immer optimal ist, den Stein auf die Seite zu legen, auf dem die Distanz zum ersten Stein einer anderen Farbe minimal ist.

2.1 Wird wirklich immer die aktuell mindeste Distanz zum Stein einer anderen Farbe genommen

Beide Indexe werden nacheinander um eins vom jeweiligen Ende weggeschoben, wenn ein Stein auf ihre Seite hinzugefügt wird, bleiben sie für einen Schritt am selben Ort des ursprünglichen Arrays, nur der Index Pointer schreitet also um eins von seinem Ende weg, womit der Abstand zum jeweiligen Ende immer minimal ist.

2.2 Weshalb führt das wiederhohlte wählen der kleinsten Distanz zur optimalen Lösung

Angenommen, dass die optimale Lösung mindestens ein Mal nicht den Stein auf die Seite, legen würden, bei der die Distanz vom Ende zum sich zu verändernden Stein minimal ist.

Demnach wäre ihre minimale Kostensumme s_o kleiner als das s meiner Lösung wäre. Für das Umdrehen von mindestens einem Stein hat diese aber s_0 um mindestens k mehr erhöht als s (da eine nicht-kürzeste Distanz gewählt wurde). Dadurch können aber höchstens k Kosten eingespart werden, da nach dem k Steine auf die andere Seite gelegt wurden, also k kosten gespart wurden, die Distanz zum ursprünglichen Input auf beiden Seiten gleich ist. Allerdings gilt $s_o + k - k = s_o$, was heisst dass man durch Wählen einer nicht kürzersten Distanz keine kleineren Kosten als durch das wiederhohlte Wählen einer kürzesten Distanz bekommt. Demnach ist $s_o = s$, womit meine Lösung die optimale Lösung ist.

3 Analyse zu Laufzeit und Speichernutzung

3.1 Laufzeit

Der Algorithmus führt pro Element im Array eine nicht mit der Grösse des Arrays wachsende Anzahl Operationen aus. Die Laufzeit dieses Algorithmus ist folgich $O(n)$

Algorithm 1 Pseudocode für Reversi Subtask 4

```
startIndex = addedDiscsAtTheStart = addedDiscsToTheEnd = 0
startFlag = endFlag = false
endIndex = n-1

while true do
  if !startFlag {Wenn in der letzten Runde kein Stein links dazugelegt
  wurde} then
    startIndex++
    if input[startIndex] ≠ lastStartChar {Soll links ein Stein dazugelegt
    werden} then
      operationCounter+= start + 1 + addedDiscsAtTheStart
      addedDiscsAtTheStart++
      startFlag = true
    end if
  else
    startFlag=false
  end if

  if startIndex == endIndex{Wenn sich die beiden Pointer getroffen
  haben, also alle Steine die selbe Farbe haben} then
    break
  end if

  if !endFlag {Wenn in der letzten Runde kein Stein rechts dazugelegt
  wurde} then
    endIndex++
    if input[endIndex] ≠ lastEndChar {Soll rechts
    ein Stein dazugelegt werden} then
      operationCounter+= input.size() - endIndex + addedDiscsAtT-
      heEnd
      addedDiscsAtTheEnd++
      endFlag = true
    end if
  else
    endFlag=false
  end if

  if startIndex == endIndex {Wenn sich die beiden Pointer getroffen
  haben, also alle Steine die selbe Farbe haben} then
    break
  end if

  lastStartChar = input[startIndex]
  lastEndChar = input[endIndex]
end while
```

3.2 Speichernutzung

Da keine Arrays, Vektoren oder derartiges verwendet werden, die Anzahl an benötigten Variablen nicht mit der Grösse der Eingabe steigt und auch keine Elemente der Eingabe verändert werden müssen, wird $O(1)$ Speicher benutzt.

4 Pseudocode

Anmerkung: Die Vergleiche und Gleichsetzungen der Flags am Anfang des Pseudocodes haben den Zweck, dafür zu sorgen, dass die start-/endFlag nur dann wahr ist, wenn in der vorhergegangenen Runde ein Stein auf dieser Seite hinzugefügt wurde, folglich nur der andere Iterator inkrementiert werden sollte.