Kassiopeias Weg

Aufgabe 1, Runde 1, 34. Bundeswettbewerb Informatik

Marian Dietz, Johannes Heinrich, Erik Fritzsche

1 Lösungsidee

Quadratien ist ein Graph. Jedes *weiße* Feld ist ein Knoten und hat Kanten zu den maximal vier weißen Feldern, die in den vier Himmelsrichtungen um ihn liegen. n ist die Anzahl der weißen Felder und damit Knoten in Quadratien.

1.1 Teilaufgabe 1

Zunächst muss bestimmt werden, ob Kassiopeia von ihrem Startpunkt aus alle weißen Felder erreichen kann, ohne ein schwarzes Feld zu betreten. Dies ist er Fall, wenn der Graph (Quadratien) zusammenhängend ist, d. h., wenn jedes weiße Feld mit jedem anderen weißen Feld über irgendeinen Weg verbunden ist. Der Zusammenhang eines Graphen lässt sich mittels Tiefensuche überprüfen.

Immer wenn man ein weißes Feld besucht, angefangen mit dem Startpunkt Kassiopeias, wird n um eins verringert und die Farbe des Feldes auf schwarz gesetzt. Dadurch kann das Feld nicht erneut besucht werden.

Ist $n = 0$, nachdem dies getan wurde, dann gibt es keine weißen Felder mehr. Das bedeutet wiederum, dass alle weißen Felder besucht werden konnten und Kassiopeia alle weißen Felder erreichen kann.

Ist dies nicht der Fall, dann muss noch weiter gesucht werden, und zwar in allen vier Himmelsrichtungen der Reihe nach (N, O, S, W). Existiert dort ein weißes Feld, dann besucht man dieses. Dabei wird wieder n verringert etc. Wurden von einem Feld aus alle angrenzenden Felder besucht, geht man wieder auf das vorherige Feld zurück und sucht weiter in den restlichen Himmelsrichtungen.

Wenn $n > 0$, nachdem vom Startpunkt aus alle vier Himmelsrichtungen abgesucht wurden, ist der Graph nicht zusammenhängend und Kassiopeia kann nicht alle weißen Felder besuchen.

Der Pseudocode 1 verdeutlicht den rekursiven Algorithmus. Die Funktion sollte am Anfang mit dem Startpunkt von Kassiopeia aufgerufen werden (wobei jedes andere weiße Feld auch funktionieren würde). Es wird true zurückgegeben, wenn Kassiopeia alle weißen Felder besuchen kann und false, wenn nicht.

In Bild 1 wird sichtbar, in welcher Reihenfolge die Felder abgerufen werden, wenn das Beispiel des Aufgabenblatts verwendet wird.

Algorithm 1 Kassiopeia

```
1:  function KASSIOPEIA(field)
2:      field.state ← BLACK
3:      n ← n − 1
4:      if n ≤ 0 then
5:          return true
6:      end if
7:      for all neighbor in field.neighbors do
8:          if neighbor.state is WHITE and KASSIOPEIA(neighbor) is true then
9:              return true
10:         end if
11:     end for
12:     return false
13: end function
```

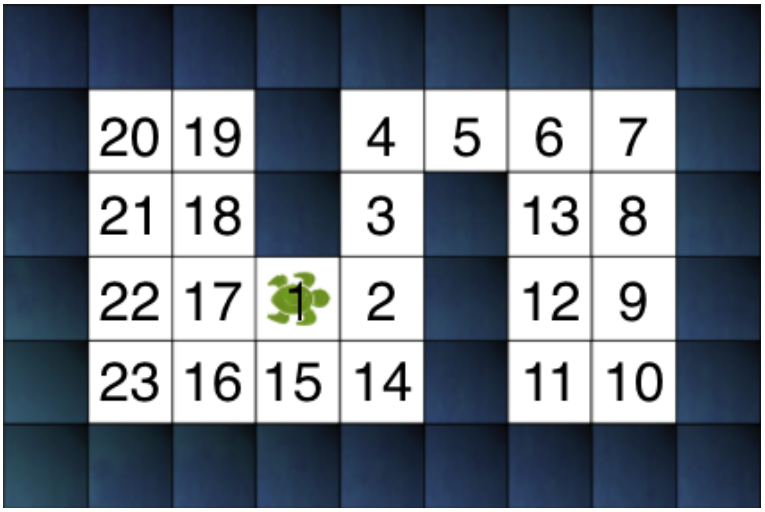

Abbildung 1: Kassiopeia

1.1.1 Laufzeitanalyse

Eine Tiefensuche in einem Graphen $G = (V, E)$ benötigt normalerweise die Zeit $\mathcal{O}(|V| + |E|)$ mit $|V|$ als Anzahl der Knoten und $|E|$ als Anzahl der Kanten von G.

Die benötigte Zeit ist in der gesamten Funktion jedoch konstant, bis auf Zeile 8, da dort die Funktion erneut aufgerufen wird. Da dies jedoch nur n-Mal geschehen kann, wird hier nur die Zeit $\mathcal{O}(n)$ benötigt.

1.2 Teilaufgabe 2

Mit dem Prinzip des Back Tracking kann jetzt bestimmt werden, wie Kassiopeia gehen muss, damit sie kein weißes Feld zweimal betreten muss (bzw. ob dies überhaupt möglich ist). Vorher muss Quadratien wieder zurückgesetzt werden, denn bei der ersten Tiefensuche wurden alle besuchten Felder schwarz markiert. Dasselbe gilt für n.

Wenn bei Teil 1 herausgefunden wurde, dass Kassiopeia nicht alle weißen Felder besuchen kann, muss dieser Teil gar nicht ausgeführt werden, denn dann kann Kassiopeia auch keinen Weg finden, durch den sie jedes Feld nur einmal betreten muss.

Da Back Tracking durch Tiefensuche funktioniert, kann Teil 1 wiederverwendet werden und muss nur an einigen Stellen abgewandelt werden.

- Es muss der Weg zum Ziel aufgezeichnet werden. Daher wird immer, wenn ein weiteres Feld besucht wird, die Himmelsrichtung, in die Kassiopeia dafür gehen müsste, zu einer Zeichenkette hinzugefügt. Diese Zeichenkette wird bei jedem weiteren besuchten Feld erweitert. Führt der ausprobierte Weg zum Misserfolg, so wird der letzte Buchstabe wieder entfernt.

- Statt *true* bzw. *false* (siehe Pseudocode) muss jetzt der aufgezeichnete Weg in Form der Zeichenkette (sofern er existiert) zurückgegeben werden.

- Wurde ein Feld besucht, von dem aus kein passender Weg gefunden wurde, so muss das Feld auch wieder als weiß markiert werden, da andere Wege natürlich auch dieses Feld miteinbeziehen müssen. Außerdem muss n wieder um eins erhöht werden. Bei der Tiefensuche der Teilaufgabe 1 war dies nicht nötig, denn dort musste nur überprüft werden, *ob* alle weißen Felder miteinander verbunden sind, und nicht, wie man gehen muss, um alle weißen Felder *einmal* zu besuchen.

Algorithmus 2 zeigt den abgeänderten Pseudocode. Die Funktion sollte anfänglich mit dem Startpunkt von Kassiopeia sowie einer leeren Zeichenkette aufgerufen werden.

Möglich wäre auch, die Teilaufgabe 1 mit der Teilaufgabe 2 zu verknüpfen, sodass das Back Tracking in Teil 2 auch speichert, welche weißen Felder alle besucht wurden, sodass zum Schluss überprüft werden kann, ob alle weißen Felder erreichbar sind. Da bei größeren Feldern jedoch die Teilaufgabe 2 deutlich mehr Zeit benötigen kann als Teilaufgabe 1, kann es passieren, dass in Teil 2 ein Feld sehr oft besucht wird, sodass unnötige Aktionen ausgeführt werden. Daher trennen wir Teil 1 und 2.

Algorithm 2 Kassiopeias Weg

```
1: function KASSIOPEIAS-WEG(field, path)
2:     field.state ← BLACK
3:     n ← n − 1
4:     if n ≤ 0 then
5:         return path
6:     end if
7:     for all neighbor in field.neighbors do
8:         if neighbor.state is WHITE then
9:             newPath ← KASSIOPEIAS-WEG(neighbor, path + Himmelsrichtung von field nach neighbor)
10:            if newPath is not null then
11:                return newPath
12:            end if
13:        end if
14:    end for
15:    field.state ← WHITE
16:    n ← n + 1
17:    return null
18: end function
```

1.2.1 Laufzeitanalye

Von jedem Feld aus gibt es maximal vier Möglichkeiten, weiterzugehen. Daher beträgt die Laufzeit im schlimmsten Fall $\mathcal{O}(4^n)$.

2 Umsetzung

Das Programm wurde in der Programmiersprache Swift geschrieben und ist lauffähig auf OS X 10.9 Mavericks und neuer. Zunächst muss ein Befehl wie „chmod 111 PFAD_ZUM_PROGRAMM“ im Terminal ausgeführt werden. Danach kann das Programm gestartet werden, indem der Pfad im Terminal eingegeben und durch die Eingabetaste bestätigt wird. Alternativ kann das Icon des Programmes in den Terminal gezogen werden, wodurch der Pfad automatisch eingegeben wird.

2.1 Programmstruktur

- Datei *Task.swift* - Implementation des Algorithmus.
- Datei *TaskReader.swift* - Liest die Aufgabe ein.
- Datei *main.swift* - Startet den `TaskReader` und übergibt diese Eingaben `Task`. Gibt schließlich die Lösung aus.

Das Wichtige geschieht also in *Task.swift*. Dort sind folgende Komponenten enthalten:

- `Coordinates` - Koordinaten mit der Zeilen- und Spaltenangabe.
- `Dimensions` - Eine Größe mit der Höhe und der Breite.
- `Neighbor` - Steht für einen Nachbarn. Besitzt die Koordinaten des Nachbarn sowie die Richtung als String, in der sich der Nachbar befindet (N, O, S oder W).
- `FieldType` - Enum mit den möglichen Arten eines Feldes. `Black`, `White` oder `Kassiopeia`, wobei letzteres für das Startfeld von Kassiopeia steht und demnach auch weiß ist.
- `Quadratien` - Stellt ganz Quadratien als `struct` (value type, wird bei der Übergabe kopiert) dar. Quadratien besitzt eine Größe in Form von `Dimensions`, den Startpunkt Kassiopeias als `Coordinates`, die Anzahl der verbleibenden weißen Felder sowie die `FieldType`s aller Felder als zweidimensionales Array.
- `CheckConnectivityTask` - Beinhaltet den Algorithmus, der prüft, ob alle weißen Felder miteinander verbunden sind.
- `SearchWayTask` - Beinhaltet den Algorithmus, der den Weg für Kassiopeia heraussucht.
- `Task` - Startet `CheckConnectivityTask` sowie `SearchWayTask`.

2.2 `Task`

`Task` wird mit einer Instanz von Quadratien erstellt. Mit der Methode `run()` werden die Algorithmen ausgeführt. Zunächst wird `CheckConnectivityTask` erstellt und gestartet. Nur wenn dies erfolgreich ist, wird auch noch `SearchWayTask` erstellt, sonst wäre das sinnlos. `run()` hat zwei Rückgabewerte: der erste besagt, ob alle weißen Felder miteinander verbunden sind, der zweite ist ein Optional und beinhaltet wenn vorhanden den Weg für Kassiopeia als String. Da Quadratien ein `struct` ist, kann es beiden Aufgaben übergeben werden, ohne dass es nach dem ersten Teil zurückgesetzt werden muss, denn es wird beim Übergeben immer automatisch kopiert.

2.3 `CheckConnectivityTask`

`CheckConnectivityTask` wird ebenfalls mit einer Instanz von Quadratien erstellt und mit `run()` ausgeführt. Die Tiefensuche wird rekursiv durch `depthFirstSearchAtCoordinates(_)` durchgeführt, angefangen in `run()` mit dem Startpunkt Kassiopeias (gespeichert in Quadratien). Bei jedem Aufruf wird die Art des Feldes an den übergebenen Koordinaten von Quadratien auf `Black` gesetzt und die Anzahl der verbleibenden weißen Felder dekrementiert. Wenn diese gleich 0 ist, wird `true` zurückgegeben. Ansonsten wird `neighborsForCoordinates(_)` auf Quadratien aufgerufen, diese Methode gibt in Form von `Neighbor`s alle (maximal vier) Nachbarn zurück. Wenn eines der Felder nicht schwarz ist, wird die Tiefensuche wieder rekursiv aufgerufen, diesmal mit dem eben ermittelten

Nachbarn. Wenn diese true zurückgibt, wird hier ebenfalls true zurückgegeben. Ansonsten werden noch die restlichen Nachbarn überprüft und schließlich wird bei Misserfolg false zurückgegeben. `run()` gibt dann ebenfalls den von der Tiefensuche produzierten Wert zurück.

2.4 `SearchWayTask`

Diese Klasse hat dieselbe Struktur wie `CheckConnectivityTask`. Folgendermaßen sehen die Unterschiede aus:

- Die Tiefensuche gibt keinen booleschen Wert zurück, sondern einen optionalen String, der den Weg beinhaltet. Dieser ist bei Misserfolg nil. Genauso sieht der Rückgabewert von `run()` aus.
- Die Tiefensuche hat einen weiteren Parameter: den Weg, der aktuell verwendet wird und zu den übergebenen Koordinaten führt. Ist irgendwann die Anzahl der verbleibenden weißen Felder 0, so wird der aktuelle Pfad zurückgegeben, was bis zu der Methode `run()` zurückgeht. Geht die Tiefensuche zu einem Nachbarn, so wird dem Aufruf der bisherige Pfad übergeben, dem die Himmelsrichtung zum Nachbarn angehängt wurde (definiert in `Neighbor`).
- Bevor nil zurückgegeben wird, wird das Feld an den gegebenen Koordinaten von Quadratien wieder auf `White` gesetzt und die Anzahl der verbleibenden weißen Felder inkrementiert.

2.5 `Quadratien`

Auf die Felder von Quadratien kann mithilfe eines `subscript`s zugegriffen werden, der als Parameter Koordinaten nimmt. Dadurch ist bspw. folgendes möglich: `quadratien[(row: 4, column: 3)] = .Black`.

Die Methode `neighborsForCoordinates(_)` mit dem Rückgabewert `[Neighbor]` arbeitet so, dass in einer Schleife die vier möglichen Nachbarn durchlaufen werden. Diese sind in der folgenden Liste zu sehen, wobei `row` und `column` die übergebenen Koordinaten sind:

- Ein Nachbar mit `(row: row-1, column: column)` und der Himmelsrichtung N.
- Ein Nachbar mit `(row: row, column: column+1)` und der Himmelsrichtung O.
- Ein Nachbar mit `(row: row+1, column: column)` und der Himmelsrichtung S.
- Ein Nachbar mit `(row: row, column: column-1)` und der Himmelsrichtung W.

Alle existierenden Nachbarn (d. h. sie liegen innerhalb Quadratiens) werden einem Array hinzugefügt, das zum Schluss zurückgegeben wird.

3 Beispiele

Hier sind alle Beispielaufgaben der Website mit den zugehörigen Lösungen aufgelistet. Die Eingabe des Dateipfades wurde bei allen Beispielen weggelassen, die Aufgabe selber ist jedoch überall zu sehen.

Alle Beispiele inklusive Lösungen befinden sich zusätzlich im Ordner dieser Einsendung.

3.1 Beispiel 0

```
6 9
#########
#   #     #
#   # #   #
#   K #   #
#     #   #
#########
```

```
Es ist möglich, alle weißen Felder zu erreichen.
WNNWSSSOOONNNOOOSSSWNN
```

3.2 Beispiel 1

```
7 11
###########
#       #   #
#    ###    #
#    #      #
# K ###     #
#       #   #
###########
```

```
Es ist nicht möglich, alle weißen Felder zu erreichen.
```

3.3 Beispiel 2

```
5 8
########
#   K    #
# ###    #
#        #
########
```

```
Es ist möglich, alle weißen Felder zu erreichen.
Es gibt jedoch keinen Weg, mit dem kein weißes Feld mehrmals betreten
    werden muss.
```

3.4 Beispiel 3

```
5 9
#########
#   K     #
# ###     #
#         #
#########
```

```
Es ist möglich, alle weißen Felder zu erreichen.
OOOOSSWNWSWWWWWNNO
```

3.5 Beispiel 4

```
3 15
###############
#      K        #
###############
```

```
Es ist möglich, alle weißen Felder zu erreichen.
Es gibt jedoch keinen Weg, mit dem kein weißes Feld mehrmals betreten
    werden muss.
```

3.6 Beispiel 5

```
3 15
###############
#              K#
###############
```

```
Es ist möglich, alle weißen Felder zu erreichen.
WWWWWWWWWWWW
```

3.7 Beispiel 6

```
5 7
#######
#K    #
# # # #
# # # #
#######
```

```
Es ist möglich, alle weißen Felder zu erreichen.
Es gibt jedoch keinen Weg, mit dem kein weißes Feld mehrmals betreten
    werden muss.
```

3.8 Beispiel 7

```
5 12
############
#K  #     # #
# #     #   #
#     #   #  #
############
```

```
Es ist möglich, alle weißen Felder zu erreichen.
Es gibt jedoch keinen Weg, mit dem kein weißes Feld mehrmals betreten
    werden muss.
```

4 Quelltext

Listing 1: main.swift

```swift
let solution = TaskReader().read().run() // Aufgabe einlesen und ausführen

if solution.connected {
  print("Es ist möglich, alle weißen Felder zu erreichen.")
  if let way = solution.way {
    print(way)
  } else {
    print("Es gibt jedoch keinen Weg, mit dem kein weißes Feld mehrmals
        betreten werden muss.")
  }
} else {
  print("Es ist nicht möglich, alle weißen Felder zu erreichen.")
}
```

Listing 2: Task.swift

```swift
// Koordinaten in Quadratien
typealias Coordinates = (row: Int, column: Int)

// Größe für Quadratien
typealias Dimensions = (height: Int, width: Int)

// Ein Nachbar von einem Feld hat Koordinaten und die Himmelsrichtung, in die
    der Nachbar ist
typealias Neighbor = (coordinates: Coordinates, direction: String)

// Die Art eines Feldes
enum FieldType: Character {
  case Black = "#"
  case White = " "
  case Kassiopeia = "K"
```

```
}

struct Quadratien {

  var fields: [[FieldType]] // Die Feldarten in einem zweidimensionalen Array
  var dimensions: Dimensions // Die Größe von Quadratien
  var remainingWhiteFields: Int // Anzahl der restlichen weißen Felder
  var startingPoint: Coordinates // Anfangspunkt von Kassiopeia

  // Zugriff auf die Feldarten
  subscript(coordinates: Coordinates) -> FieldType {
    get {
      return fields[coordinates.row][coordinates.column]
    } set {
      fields[coordinates.row][coordinates.column] = newValue
    }
  }

  // Berechnet die Nachbarn des Feldes mit den gegebenen Koordinaten
  func neighborsForCoordinates(coordinates: Coordinates) -> [Neighbor] {
    var neighbors = [Neighbor]() // alle Nachbarn

    // In allen vier Himmelsrichtungen nachschauen:
    for neighbor in [
      (coordinates: (row: coordinates.row - 1, column: coordinates.column),
          direction: "N"),
      (coordinates: (row: coordinates.row, column: coordinates.column + 1),
          direction: "O"),
      (coordinates: (row: coordinates.row + 1, column: coordinates.column),
          direction: "S"),
      (coordinates: (row: coordinates.row, column: coordinates.column - 1),
          direction: "W")
      ] {
        if neighbor.coordinates.row >= 0 && neighbor.coordinates.column >= 0 &&
            neighbor.coordinates.row < dimensions.height &&
            neighbor.coordinates.column < dimensions.width {
          // Der Nachbar befindet sich noch in Quadratien, also der Liste
              hinzufügen
          neighbors.append(neighbor)
        }
    }

    return neighbors
  }

}

// Prüft, ob alle weißen Felder miteinander verbunden sind
class CheckConnectivityTask {
```

```
  var quadratien: Quadratien

  init(quadratien: Quadratien) {
    self.quadratien = quadratien
  }

  func run() -> Bool {
    return depthFirstSearchAtCoordinates(quadratien.startingPoint)
  }

  // Tiefensuche an den gegebenen Koordinaten
  func depthFirstSearchAtCoordinates(coordinates: Coordinates) -> Bool {

    // Feld wurde gefunden
    quadratien[coordinates] = .Black
    quadratien.remainingWhiteFields--

    if quadratien.remainingWhiteFields <= 0 {
      // Alle Felder wurden gefunden
      return true
    }

    // Alle weißen Nachbarn durchlaufen
    for neighbor in quadratien.neighborsForCoordinates(coordinates) where
        quadratien[neighbor.coordinates] != .Black {
      if depthFirstSearchAtCoordinates(neighbor.coordinates) {
        // Tiefensuche konnte alle weißen Felder finden
        return true
      }
    }

    // Es wurden (noch) nicht alle weißen Felder gefunden
    return false
  }

}

// Prüft, ob Kassiopeia an alle weißen Felder rankommen kann, ohne ein Feld
    mehrfach zu überqueren
class SearchWayTask {

  var quadratien: Quadratien

  init(quadratien: Quadratien) {
    self.quadratien = quadratien
  }

  func run() -> String? {
```

```
    return depthFirstSearchAtCoordinates(quadratien.startingPoint, path: "")
  }

  // Tiefensuche an den gegebenen Koordinaten und dem bisherigen Pfad dorthin
  func depthFirstSearchAtCoordinates(coordinates: Coordinates, path: String)
      -> String? {

    // Feld wurde gefunden
    quadratien[coordinates] = .Black
    quadratien.remainingWhiteFields--

    if quadratien.remainingWhiteFields <= 0 {
      // Alle Felder wurden gefunden
      return path
    }

    // Alle weißen Nachbarn durchlaufen
    for neighbor in quadratien.neighborsForCoordinates(coordinates) where
        quadratien[neighbor.coordinates] != .Black {
      if let path = depthFirstSearchAtCoordinates(neighbor.coordinates, path:
          path + neighbor.direction) {
        // Tiefensuche hat einen passenden Pfad gefunden
        return path
      }
    }

    // Feld muss wieder zurückgesetzt werden, da es hier nicht funktioniert
        hat, einen passenden Weg zu finden
    quadratien[coordinates] = .White
    quadratien.remainingWhiteFields++
    return nil
  }
}

// Prüft beide Aufgaben
class Task {

  var quadratien: Quadratien

  init(quadratien: Quadratien) {
    self.quadratien = quadratien
  }

  func run() -> (connected: Bool, way: String?) {
    // Prüfen, ob alle weißen Felder verbunden sind
    let connected = CheckConnectivityTask(quadratien: quadratien).run()

    // Prüfen, ob Kassiopeia alle weißen Felder ohne mehrfachen Besuch besuchen
        kann.
```

```
    // Da Quadratien ein 'value type' ist, wird es beim Funktionenaufruf
        kopiert und kann hier ein zweites Mal verwendet werden, ohne die
        veränderten Werte zurückzusetzen.
    let way = connected ? SearchWayTask(quadratien: quadratien).run() : nil

    return (connected, way)
  }
}
```