Dokumentation der Aufgabe 5 des Bundeswettbewerb Informatik 2020

Bearbeitet von Richard Förster

Die Aufgabe

In der Aufgabe 5 bestand das Kernproblem darin, dass n Geschenke auf n Personen geschickt verteilt werden. Dabei hat jede Person Geschenke, die sie gerne haben will, einen Erstwunsch, einen Zweitwunsch und einen Drittwunsch. Es gilt, so viele Erstwünsche wie möglich zu erfüllen, danach so viele Zweitwünsche und schließlich so viele Drittwünsche wie möglich. Ein Erstwunsch ist wichtiger als alle Zweitwünsche und ein Zweitwunsch ist wichtiger als alle Drittwünsche. Heißt, Hat man 0 Erstwünsche und 1000 Zweitwünsche erfüllt, ist dennoch 1 Erstwunsch und kein anderer „mehr“ erfüllt laut Aufgabe.

Mein Lösungsansatz

Meine Herangehensweise besteht darin, zuerst alle Erstwünsche zu erfüllen. Anschließend werden alle Geschenke, die sich niemand als Erstwunsch gewünscht hat, an alle die verteilt, die ihren Erstwunsch nicht erfüllt bekommen haben (weil ein anderer sich das Geschenk auch gewünscht hat, dazu später weiteres) und deren Zweitwunsch entspricht. Selbes Prinzip wird dann auf den Drittwunsch der übrigen Personen angewendet. Gibt es dann noch Personen, die die alle drei Wünsche nicht erfüllt bekommen haben, werden den übrigen Geschenken, die niemand haben wollte, zugeordnet. Jedem Geschenk wird genau eine Person zugeordnet, denn es gibt ja genau so viele Personen wie Geschenke. Allerdings muss es immer mindestens drei Geschenke geben, denn sonst können nicht drei unterschiedliche Wünsche entstehen.

Interessant wird die Situation, wenn sich zwei Personen sich das gleiche Geschenk mit gleicher Stärke wünschen, also den selben Erstwunsch, Zweitwunsch oder Drittwunsch haben *und auch in der selben Runde um das Geschenk „kämpfen“.* Es gibt, wie schon beschrieben, 4 Runden, in der Geschenke verteilt werden. Die erste für die Erstwünsche, die zweite für die Zweitwünsche, die dritte für die Drittwünsche und die letzte für die Personen, die leer ausgegangen sind (wobei die letzte nicht von Relevanz beim „Kampf-Problem“ ist). Ein Beispiel: Wir haben 3 Personen P1, P2 und P3. P1 wünscht sich Geschenk 1 (Erst-), 2 (Zweit-), oder 3 (Drittwunsch), P2 Geschenk 2, 3, oder 1 und P3 2, 1, oder 3. P1 wird zuerst ein Geschenk zugeteilt, dann P2 und schließlich P3. P1’s Erstwunsch, Geschenk 1, ist frei, er bekommt es. Geschenk 1 ist nun vergeben. Als nächstes wird P2 ein Geschenk zugewiesen. Er bekommt Geschenk 2, es ist nun auch vergeben. Schließlich ist P3 an der Reihe. Sein Erstwunsch ist aber schon an P2 vergeben! Wer von beiden bekommt es? Wer es als erstes bekommen hat hat keinen Einfluss auf die insgesamte Zufriedenheit, denn Zufriedenheit hängt von der bestmöglichen Erfüllung der Wünsche ab, nicht von der Reihenfolge. Würden wir also P2 Geschenk 2 geben, bleibt für P3 nur das Geschenk 3 übrig, was sein Drittwunsch ist. Gibt man allerdings P3 Geschenk 2, bleibt für P2 Geschenk 3 übrig, was immerhin sein Zweitwunsch. Bei Variante 1 wurden 2 Erstwünsche und ein Drittwunsch erfüllt und bei Variante 2 2 Erstwünsche und ein Zweitwunsch. Variante 2 ist also besser (man kann gerne versuchen, die Geschenke anders zu verteilen - man wird keine bessere finden ;)). Daraus kann man schlussfolgern, dass bei solch einem Kampf immer der gewinnt, dessen Wunsch in der drunterliegenden Stärke schon vergeben ist. Bei Variante 2 war P3’s Zweitwunsch Geschenk 1, aber dies ist schon an P1 als Erstwunsch vergeben, P2’s Zweitwunsch war allerdings das noch nicht vergebene Geschenk 3, also verliert P2 und P3 bekommt Geschenk 2. Selbes gilt für die zweite Verteilungsrunde, wenn es um die Zweitwünsche geht: in Streitfällen entscheidet, ob der Drittwunsch vergeben ist oder nicht. In der dritten Verteilungsrunde gibt es dann keinen weiteren Kampf mehr; einer von beiden wird willkürlich zum Sieger erklärt.

Was passiert, wenn z.B. bei zwei um den Erstwunsch kämpfenden Personen die Zweitwünsche frei sind? Oder wenn beide schon vergeben sind? Oder wenn sie gleiche Zweitwünsche haben? In allen Fällen entscheidet, ob der Drittwunsch schon vergeben ist oder nicht. Hat einer von beiden einen Drittwunsch, der schon vergeben ist, muss der andere zurücktreten, denn bei ihm ist die Chance doppelt so groß wie beim anderen ein Geschenk abzukriegen. Sind in der zweiten Verteilungsrunde die Drittwünsche beide frei oder vergeben, wird willkürlich entschieden, wer der Sieger ist.

Haben zwei Personen komplett identische Erst-, Zweit- und Drittwünsche, ist egal, wer gewinnt. Ein Sieger wird willkürlich auserkoren.

Zuletzt hat sich aus Experimentieren und Ausprobieren ergeben, dass die Reihenfolge der Geschenkverteilung auch eine Rolle Spielt. Nehmen wir nochmal das Beispiel mit P1, P2 und P3,

nur diesmal beginnen wir mit P3, dann folgt P2 und P1. P3 bekommt Geschenk 2. P2 macht nun P3 sein Geschenk streitig, es kommt zum Kampf: ihre Zweitwünsche, Geschenk 1 & 3, sind beide frei, ihre Drittwünsche, auch Geschenk 1 & 3 allerdings auch. Es kommt zu willkürlichen Entscheidung. Wenn nach Zufall entschieden wird, könnte es vorkommen, dass P2 zum Sieger erklärt wird und P3 erst wieder in der zweiten Entscheidungsrunde dran ist. Zum Schluss der ersten Runde ist noch P1 dran, er bekommt sein Geschenk 1, denn es ist noch frei. Aber auch in der zweiten Runde muss P3 feststellen, dass sein Zweitwunsch, Geschenk 1, an P1 vergeben ist. Ihm bleibt also nur noch Geschenk 3. Somit haben wir die Situation erreicht, die schon im vorherigen Beispiel „unglücklicher“ war. Bekommt aber nun P3 in der ersten Entscheidungsrunde Geschenk 2, wird im Endeffekt P2 Geschenk 3 erhalten, und die bestmögliche Situation ist erreicht. Man könnte schlussfolgern, dass man sich zuerst eine Reihenfolge überlegen muss, wer nach wem sein Geschenk bekommt. Es geht aber auch einfacher, mit gleichem Ergebnis: pro Verteilungsrunde gibt es zwei Phasen: zuerst werden alle Geschenke herausgesucht, die überhaupt jemand haben will (in der jeweiligen Wunschklasse). Heißt, man gibt zuerst jedem ein Geschenk, falls es noch nicht vergeben ist, egal, ob es sich um die passenden Personen für dieses Geschenk handelt. In der zweiten Phase werden dann erneut die Geschenke verteilt, nun weiß man jedoch im Voraus, welche Geschenke in dieser Runde vergeben werden, es wird also effizient verhindert, dass solche unberechtigten Willkürsituationen wie im Beispiel weiter oben entstehen.

Zur Implementierung

Scala

Das Programm zur Simulation solcher Verteilungssituationen habe ich in Scala (zu Java-Bytecode kompiliert), einer objektorientierten und funktionalen Programmiersprache umgesetzt. Warum Scala? Mir ist bewusst, dass Scala nicht gerade eine allzu bekannte Programmiersprache ist wie Java, C++ oder Python. Ich bin allerdings bevor ich mit der Umsetzung dieser Aufgabe begonnen habe auf Scala gestoßen und musste feststellen, dass sie leicht verständlich und lesbar (zumindest für mich) und trotzdem zu komplexen, effizienten Lösungen fähig ist. Außerdem entspricht sie vielen meiner Ideale bezüglich dem Design von Programmiersprachen. Um etwas Erfahrung in Scala und ihren Programmierbibliotheken zu bekommen, entschied ich mich, Scala für die Implementierung von Aufgabe 5 zu benutzen.

Struktur

Das Projekt besteht aus der Datei „Main.scala“ mit der main-Funktion und anderen Algorithmen, aus „Person.scala“, die eine Klasse definiert, die eine Person darstellt und aus „Choice.scala“, welche eine Klasse zur Darstellung von Erst-, Zweit- und Drittwunsch definiert.

Das Programm ist Kommandozeilenbasiert, heißt, es gibt keine Fenster, sondern die Verteilung der Geschenke wird Zeile für Zeile ausgegeben, mit anschließender Zusammenfassung.

Die Eingabe des Wichtel-Layouts erfolgt über das Terminal, wobei man hier entweder hier das Layout direkt und ohne Zeilenumbrüche oder als URL zu einem Layout eingibt, wie auch dem Benutzer beim Start mitgeteilt wird. Falls es einen Fehler beim parsen der URL oder sonstige Fehler auftreten, wird die Eingabe wie ein Layout behandelt.

```
val userinput = scala.io.StdIn.readLine()
val text = try {
  new String(new java.net.URL(userinput).openStream().readAllBytes())
} catch {
  case m: Exception => userinput
}
```

Falls das Layout ein ‚n' beinhaltet (was es normalerweise nicht der Fall ist), wird angenommen, dass der Benutzer das Programm beenden will.

```
if(text.toLowerCase.contains('n')) {
  println("Vielen Dank fürs Ausprobieren, einen schönen Tag noch!")
  return
}
```

Der nächste Schritt ist dann das interpretieren des Layouts zu einer „Sammlung" von Personen (Set[Person] in Scala). Die erste Zahl im Layout gibt an, wie viele 3er-Päckchen von Zahlen folgen. Anschließend wird aus jedem 3er-Päckchen ein „Choice"-Objekt aufgebaut und daraus ein Person-Objekt, welches in die Sammlung von Personen eingefügt wird. Der reguläre Ausdruck in der ersten Zeile steht für „ein oder mehr Lehrzeichen/Tabs/Zeilenumbrüche". Geht irgendetwas schief, wird nicht die Sammlung, sondern „None" bzw. Nichts zurückgegeben (Rückgabetyp „Option"). Der Fehler wird dem Benutzer auch auf Zeile 26 mitgeteilt. Während nach Außen die Geschenke von 1 bis n nummeriert sind, sind sie im Programm von 0 bis n-1, da sich damit besser umgehen lässt. „ints.next.toInt" bedeutet hier „die nächste Zeichenkette aus der Liste „ints", konvertiert zu einer Zahl".

```
val ints = prog.split( regex = "\\s+").iterator
val len = ints.next.toInt
val persons = Set.newBuilder[Person]

for(i <- 0 until len) {
  persons.addOne(Person(i, Choice(ints.next.toInt-1, ints.next.toInt-1, ints.next.toInt-1)))
}
```

In der Funktion „solve" wird eine Zuordnung von Geschenken (Zahlen) zu Personen berechnet. In „taken" sind alle festen Zuordnungen gespeichert, also wenn gerade die zweite Verteilungsrunde läuft, sind alle Zuordnungen aus der ersten Runde darin festgehalten. „queue" stell eine Sammlung von den Personen dar, die noch ein Geschenk zugeordnet bekommen müssen. Und „free" ist eine Sammlung von Geschenken, die noch zu vergeben sind.

```
def solve(persons: Set[Person]): mutable.Map[Int, Person] = {
  val taken = mutable.HashMap[Int, Person]()
  val queue = mutable.HashSet.from(persons)
  val free = mutable.HashSet.from(0 until persons.size)
```

Anschließend beginnen die 3 Verteilungsrunden. Pro Runde passiert hier folgendes: zuerst wird, wie im Lösungsansatz zum Schluss bereits festgestellt wurde, ohne weitere Beachtung der Richtigkeit jeder Person ein Geschenk entsprechend seiner Wunschklasse zugeordnet. Alle noch wartenden Personen bekommen ein Geschenk vorerst zugeordnet; anschließend werden alle die Zuordnungen entfernt, dessen Geschenke schon vergeben sind aus vorherigen Verteilungsrunden (in der ersten Runde ist „taken" leer). Alle Geschenke, die gerade vergeben wurden sind, werden aus der Sammlung der freien Geschenke entfernt via „free —=— contested.keys". Wenn im „Streitalgorithmus", den ich später noch erklären werde, geprüft wird, ob ein Geschenk noch frei ist, wird diese Sammlung benutzt.

Der darauffolgende Schritt besteht darin, nochmal alle Einträge von „contested“ zu berichtigen, also mit Beachtung von freien und vergebenen Geschenken. Hier wird „queue“, also die Warteschlange mit Personen ohne Geschenk, werden nochmal durchgegangen und kämpfen gegen denjenigen, der den selben Wunsch hat. Der Gewinner „winner“ wird dann mit der Person, die vorher das Geschenk besaß, ausgetauscht: „contested.put(wish, winner)“. Sind alle wartenden Personen durchgegangen worden, werden alle Personen, die jetzt die alle Kämpfe gewonnen haben und noch in „contested“ eingetragen sind, aus der Warteschlange entfernt; die Zuordnungen werden zu den festen Zuordnungen hinzugefügt. Anschließend beginnt die nächste Runde mit der nächst niedrigeren Wunschklasse dargestellt durch eine Zählvariable „i“ von 0 bis 2.

```
for(i <- 0 until 3) {
  // add all person to the contested list so each one gets at least one chance to wish
  val contested = mutable.HashMap.from(queue.map(p => (p.choice(i), p)))
  contested --= taken.keys
  free --= contested.keys

  // reorder the contested list
  for(person <- queue) {
    val wish = person.choice(i)

    if(contested.contains(wish)) {
      val winner = person.fight(contested(wish), i, free)
      contested.put(wish, winner)
    }
  }

  queue --= contested.values
  taken ++= contested
}
```

Nachdem (fast) alle Wünsche erfüllt worden sind, werden noch die übrigen wartenden Personen, die noch in „queue“ eingetragen sind, auf die noch freien Geschenke aufgeteilt. „free zip queue“ verbindet elegant alle freien Geschenke mit den Wartenden in „queue“ und fügt diese Zuordnungen zu „taken“ hinzu. „taken“ wird als letztes Statement zurückgegeben.

```
taken.addAll(free zip queue)
taken
```

Anschließend erfolgt ab Zeile 34 die Auswertung, auf die ich jetzt nicht weiter eingehen werde. Wichtiger ist es zu klären wie der „Kampf“ aussieht: „other“ ist die Person, die das Geschenk streitig gemacht hat, „nr“ die „Wunschklasse“, in der gekämpft wird und „free“ natürlich die Sammlung freier Geschenke. Zuerst wird geprüft, ob überhaupt gekämpft werden muss. Falls die Person gegen sich selber kämpft oder wenn um den Drittwunsch gekämpft wird, wird einfach eine der Kämpfenden zurückgegeben als Gewinner, hier ist es einfach „this“. „s“ und „o“ zeigen, ob self’s oder other’s Wunschgeschenk der nächstniederen Wunschklasse noch vorhanden ist. Erinnerung: wessen Geschenk noch

frei ist, verliert. Darunter wird in einem match-Ausdruck auf unterschiedliche Kombinationen von s und o geprüft: ist „s“ wahr und „o“ falsch, so gewinnt „other“ (der Andere), andersrum gewinnt this - und falls beide Wünsche (nicht) vorhanden sind, wird in der nächstniederen Wunschklasse gekämpft. Es handelt sich hier als um einen rekursiven Algorithmus, der dann abbricht, wenn um den Drittwunsch gekämpft wird.

```
def fight(other: Person, nr: Int, free: scala.collection.Set[Int]): Person = {
  if(nr >= 2 || other == this) // quit the fight
    return this
  val s = free.contains(choice(nr+1))
  val o = free.contains(other.choice(nr+1))

  (s, o) match {
    case (true, false) => other // the other one's taken, so he wins
    case (false, true) => this // ours is taken, we win the fight
    case (false, false) | (true, true) => fight(other, nr +1, free)
  }
}
```

Verbesserungsvorschläge & Kritik

Ich habe, nachdem ich diese Implementierung geschrieben habe, noch einen anderen Versuch gestartet, der eigentlich ein besseres Ergebnis hervorbringen sollte, da ich mir nicht sicher war, ob mein bisher funktionierender Algorithmus die optimalste Verteilung berechnet. Der Algorithmus funktionierte, brachte aber nur minimal schlechtere Ergebnisse. Deshalb wüsste ich nicht, inwiefern sich der Algorithmus bezüglich der Verteilung noch verbessern lässt. jedoch bin ich mir bewusst, dass ich durchaus etwas resourcensparendere Datenstrukturen hätte verwenden können, z.B. statt einer HashMap einen Array mit schon im Voraus fester Größe. Auch hätte ich weniger Sammlungs-Differenzen berechnen können (so habe ich es in meinem anderen Ansatz gemacht). Das Problem damit: es ist nicht besonders gut lesbar, nur mit einem Haufen Dokumentation. Außerdem sind Maps und Sets genau für solche Aufgabe wie diese gemacht - um die Performance braucht man sich heutzutage (zumindest bei solchen Aufgaben) garnicht groß kümmern. Zudem bin ich ein großer Unterstützer von guten Abstrahierungen, und eine „Menge“ von Personen und eine „Zuordnung von Geschenken zu Personen“ müssen einfach ihre entsprechenden Datenstrukturen erhalten ;).

Mir ist noch eine Möglichkeit eingefallen, eventuell eine optimalere Verteilung zu erhalten: man löst die Aufgabe mithilfe eines genetischen Algorithmus: eine Menge von Zuordnungen werden immer wieder leicht abgeändert, bewertet, aussortiert wenn sie schlechter- und dupliziert wenn sie besser als der durchschnitt sind. So besteht zumindest eine Chance, eine noch bessere Verteilung zu finden. Jedoch ist dieser Ansatz enorm resourcenaufwendig und kann Minuten bis Stunden dauern.