252-0027-00: Einführung in die Programmierung

Übungsblatt 8 (ohne Bonus)

Abgabe: 21. November 2023, 23:59

Checken Sie mit Eclipse wie bisher die neue Übungs-Vorlage aus. Importieren Sie beide Eclipse-Projekte (das Projekt für den Bonus und das Projekt für die restlichen Aufgaben).

Aufgabe 1: Loop Invariante

Gegeben ist eine Postcondition für das folgende Programm

```
public int compute(String s, char c) {
  int x;
  int n;

  x = 0;
  n = 0;

  // Loop Invariante:
  while (x < s.length()) {
    if (s.charAt(x) == c) {
      n = n + 1;
    }
    x = x + 1;
  }

  // Postcondition: count(s, c) == n
  return n;
}
```

Die Methode `count(String s, char c)` gibt zurück wie oft der Character `c` im String `s` vorkommt. Schreiben Sie die Loop Invariante in die Datei "LoopInvariante.txt". **Tipp:** Benutzen Sie die `substring` Methode.

Aufgabe 2: Executable Graph

In dieser Aufgabe verwenden wir gerichtete azyklische Graphen, um Programme zu repräsentieren. Der Programmzustand ist dabei immer durch ein Tupel $(sum, counter)$ gegeben, wobei *sum* und *counter* ganze Zahlen sind. Programmzustände werden durch `ProgramState`-Objekte modelliert, wobei `ProgramState.getSum()` (bzw. `ProgramState.getCounter()`) dem ersten Element (bzw. dem zweiten Element) des Tupels entspricht.

Eine Ausführung des Programms manipuliert den Programmzustand und das Resultat eines Programms ist gegeben durch den erreichten Programmzustand, nachdem alle Operationen im Programm ausgeführt wurden. Programme können nichtdeterministisch sein: Das heisst, für ein einzelnes Programm kann es für den gleichen Startzustand mehrere Programmausführungen geben, welche zu unterschiedlichen Resultaten führen.

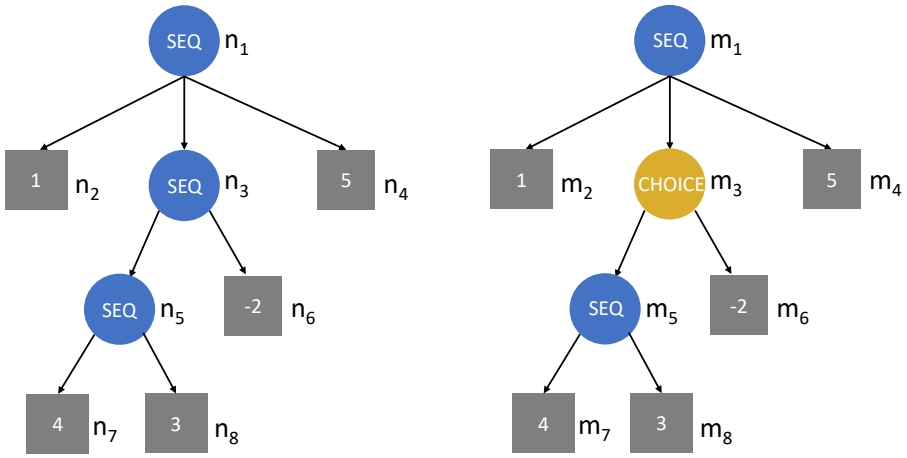
Knoten in Graphen werden durch `Node`-Objekte modelliert. `Node.getSubnodes()` gibt die Kinderknoten als ein Array zurück (m ist genau dann ein Kinderknoten von n, wenn es eine ausgehende gerichtete Kante von n zu m gibt). Wir unterscheiden drei Arten von Knoten, wobei die Methode `Node.getType()` die Knotenart als `String` zurückgibt. Um ein Programm, welches durch den Knoten `n` repräsentiert wird, auszuführen, muss man den "Knoten `n` ausführen". Wir beschreiben nun die drei Knotenarten und jeweils die Ausführung der Knoten:

1. Additionsknoten (`Node.getType()` ist "ADD"): Solche Knoten besitzen einen Additionswert a gegeben durch `Node.getValue()` (eine ganze Zahl) und bei der Ausführung dieses Knotens wird der Programmzustand von $(sum, counter)$ zu $(sum + a, counter + 1)$ aktualisiert. Die Kinderknoten von solchen Knoten werden bei der Ausführung ignoriert.

2. Sequenzknoten (`Node.getType()` ist "SEQ"): Bei der Ausführung eines Sequenzknoten `n` werden die Kinderknoten von `n` nacheinander ausgeführt. Die Reihenfolge in welcher die Kinderknoten ausgeführt werden spielt keine Rolle, da der erreichte Programmzustand für jede Reihenfolge gleich ist. `Node.getValue()` ist irrelevant.

3. Auswahlknoten (`Node.getType()` ist "CHOICE"): Bei der Ausführung eines Auswahlknoten `n` wird ein beliebiger Kinderknoten von `n` ausgewählt und ausgeführt. `Node.getValue()` ist irrelevant. Diese Knoten führen zu Nichtdeterminismus.

Sie dürfen davon ausgehen, dass Sequenz- und Auswahlknoten immer mindestens einen Kinderknoten haben, und dass es zwischen zwei Knoten immer höchstens einen Pfad gibt. Die folgende Abbildung zeigt zwei Beispielgraphen, wobei Knoten mit der Beschriftung "SEQ" (bzw. "CHOICE") Sequenzknoten (bzw. Auswahlknoten) entsprechen und die Zahlen in Additionsknoten den Additionswerten entsprechen.



Beim linken Graphen in der Abbildung gibt es immer nur eine mögliche Ausführung von n_1 pro Startzustand. Für den Startzustand $(1,2)$ ist das Resultat gegeben durch $(1+1+4+3-2+5, 2+5) = (12,7)$. Beim rechten Graphen gibt es zwei mögliche Ausführungen von m_1. Beim Auswahlknoten m_3 wird entweder m_5 oder m_6 ausgeführt (da m_5 und m_6 die Kinderknoten von m_3 sind). Die beiden möglichen Resultate für den Startzustand $(0,0)$ sind $(0+1+4+3+5, 0+4) = (13,4)$ (wenn m_5 gewählt wird) und $(0+1-2+5, 0+3) = (4,3)$ (wenn m_6 gewählt) wird.

Implementieren Sie `GraphExecution.allResults(Node n, ProgramState initState)`, welche für den Startzustand `initState` alle möglichen Resultate für das Programm repräsentiert durch `n` zurückgibt. Die Resultate sollten als eine Liste von `ProgramState`-Objekten zurückgegeben werden (repräsentiert durch die Klasse `LinkedProgramStateList`). Die Reihenfolge der zurückgegeben Liste spielt keine Rolle. Wenn das gleiche Resultat durch genau k verschiedene Ausführungen generiert werden kann, dann muss das Resultat k Mal in der zurückgegeben Liste vorkommen. Zwei Ausführungen sind unterschiedlich, wenn es mindestens einen Knoten gibt, der in einer aber nicht in der anderen Ausführung ausgeführt wird.

Wir geben zwei Testdateien zur Verfügung. "GraphExecutionTest.java" enthält Tests, welche wir an einer Prüfung geben würden. "GradingGraphExecutionTest.java" enthält Tests, welche wir zum Korrigieren einer Prüfung verwenden würden. Testen Sie ihre Lösung zuerst ausgiebig mit "GraphExecutionTest.java" (am besten fügen Sie selber neue Tests hinzu) und dann können Sie "GradingGraphExecutionTest.java" verwenden, um zu sehen wie ihre Lösung an einer Prüfung abgeschnitten hätte.

Aufgabe 3: Dominator

Im vorletzten Übungsblatt haben Sie Black-Box Testing kennengelernt. Wie in der letzten Aufgabe sollen Sie wieder Tests für eine Methode schreiben, deren Implementierung Sie nicht kennen. In Ihrem "U08"-Projekt befindet sich eine "blackbox.jar"-Datei, welche eine kompilierte Klasse `BlackBox` enthält. Den Code dieser Klasse können Sie nicht sehen, aber sie enthält eine Methode `int[] dominators(int[] elevations)`, welche Sie aufrufen können.

Die Methode `dominators` hat den Parameter `elevations`. Das Array `elevations` enthält die Höhenwerte (in Metern über Meer) für eine Serie von Punkten. Für jeden Punkt P (ein Index in `elevations`) berechnet die Methode, durch welchen anderen Punkt er *dominiert* wird. Der

Rückgabewert ist ein Array, hier mit `result` bezeichnet, welches für jeden Punkt P die Position (Index im `elevations`-Array) eines Dominators von P enthält. Falls `elevations` `null` ist, ist auch `result` `null`.

Ein *Dominator* D eines Punktes P hat von allen Punkten, welche höher als P liegen, die kleinste (horizontale) Distanz zu P. Falls zwei solche Punkte existieren, ist der höhere der beiden der einzige Dominator. Falls beide dieser Punkte gleich hoch sind, gibt es zwei mögliche Dominatoren. Falls kein Dominator für P existiert, enthält `result[`P`]` die Zahl `-1`.

Ein Beispiel finden Sie in der Datei "BlackBoxTest.java" in Ihrem Projekt. Der gegebene Test `testSimple()` übergibt als Argument die Höhenserie `{3, 1, 2, 5, 3, 6, 5, 6}`. Die Methode `dominators` sollte nun entweder `result={3, 0, 3, 5, 5, -1, 5, -1}` oder `result={3, 0, 3, 5, 5, -1, 7, -1}` zurückgeben. Andernfalls enthält die Implementierung einen Fehler.

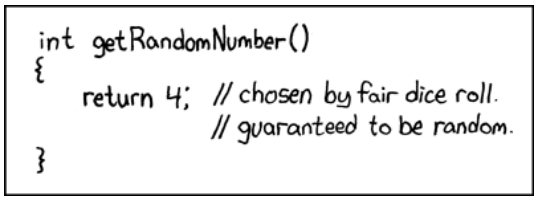
Der zweite gegebene Test `testReturnsNull()` prüft, dass die `dominators`-Methode `null` zurück gibt, wenn `null` als Argument übergeben wird. *Dieser Test schlägt für die gegebene `BlackBox`-Implementierung fehl.* Damit zeigt dieser Test, dass sich diese Implementierung nicht vollständig an die obige Spezifikation hält. Da Sie nur diese eine Implementierung zur Verfügung haben, wird dieser Test auch zum Zeitpunkt der Abgabe fehlschlagen.

Schreiben Sie nun solange weitere Tests (in "BlackBoxTest.java"), bis sie zuversichtlich sind, dass Ihre Tests die Spezifikation genügend abdecken. Um die Stärke Ihrer Tests zu beurteilen, werden wir verschiedene, teilweise fehlerhafte `BlackBox`-Implementierungen mithilfe Ihrer Tests prüfen. Je mehr Fehler Ihre Tests aufdecken, desto besser. Tests sollten fehlschlagen, falls die Implementierung fehlerhaft ist, und erfolgreich durchlaufen, falls keine Fehler vorhanden sind.

Warnung: Da Unit-Tests generell höchstens einige Sekunden dauern, werten wir Tests, die länger als eine Minute dauern, als fehlerhaft. Sie können davon ausgehen, dass keine der `BlackBox`-Implementierungen Endlosschleifen enthalten.

Aufgabe 4: Self-avoiding Random Walks

In dieser Aufgabe simulieren Sie das Schicksal eines Wolfs, der sich genau in der Mitte einer Stadt befindet und aus dieser ausbrechen will. Die Stadt besteht aus $2 \times N$ Strassen, die in einem regelmässigen Gitter angeordnet sind. N Strassen verlaufen in der West-Ost Richtung, N in der Nord-Süd Richtung. (N ist ungerade und > 1.) Die Stadt kann (für diese Aufgabe) also vollständig durch die $N \times N$ Kreuzungen der Strassen beschrieben werden.



xkcd: Random Number by Randall Munroe (CC BY-NC 2.5)

An jeder Kreuzung entscheidet der Wolf, in welche Richtung er fliehen will. Wenn der Wolf an der Kreuzung (i, j) steht, so kann er (dank seiner feinen Nase) feststellen, welche der Nachbar-Kreuzungen $(i+1, j)$, $(i-1, j)$, $(i, j+1)$, und $(i, j-1)$ er schon besucht hat[1]. Eine einmal besuchte Kreuzung wird nicht nochmal besucht, d.h., der Wolf wählt zufällig unter den Richtugen, die ihn zu einer neuen, noch nicht besuchten Kreuzung führen. Wenn der Wolf den Stadtrand (d.h. die 1. oder die N. Strasse in nord-südlicher oder west-östlicher Richtung) erreicht hat, ist die Flucht erfolgreich verlaufen. Wenn der Wolf an eine Kreuzung kommt, von der aus er

[1] Er hat mindestens eine dieser Kreuzungen im vorherigen Schritt besucht um zu (i, j) zu kommen.

keine unbesuchte Kreuzung erreichen kann, dann ist die Flucht zu Ende und der Wolf wird (entsprechende Bewilligung vorausgesetzt) von den Jägern erschossen. Abbildung 1 illustriert dies anhand von 5×5 und 7×7 Städten.

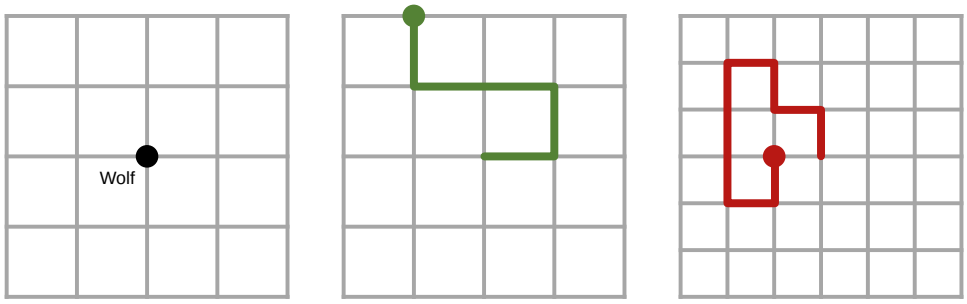


Abbildung 1: Wolf in der Stadt. Ausganglage, erfolgreiche Flucht und fehlgeschlagene Flucht

(a) Schreiben Sie ein Programm `RandomWalks`, welches die Flucht des Wolfes simuliert. Sie können die Grösse der Stadt (N) sowie die Anzahl der Simulationen von der Konsole einlesen oder fest in zwei Variablen vorgeben. Nach der Simulation soll Ihr Programm ausgeben, in wieviel Prozent der Fälle der Wolf aus der Stadt fliehen konnte. Ausserdem, wie lang im Durchschnitt der Pfad war, der es einem Wolf erlaubte zu fliehen und wie lang (im Durchschnitt) der Pfad war, wenn der Wolf nicht aus der Stadt fliehen konnte. Die Länge des Pfades wird durch die Anzahl der besuchten Kreuzungen bestimmt, inklusive der Kreuzung, an welcher der Wolf startete.

Tipps: Verwenden Sie ein 2-dimensionales Array von `booleans`, um sich die bereits besuchten Kreuzungen zu merken. Um per Zufall eine Richtung zu wählen, wird sich die Methode `Random.nextInt(int)` als nützlich erweisen.

Experimentieren Sie mit Ihrem Programm und vergleichen Sie z.B. die Wahrscheinlichkeit einer Flucht aus einer 9×9 Stadt mit der einer grösseren Stadt. Diese Simulation ist ein einfaches Beispiel für das Problem der "self-avoiding-paths" (der Pfade, die sich nicht kreuzen), welches eine Abstraktion für viele interessante Probleme aus Chemie, Pharmazie, Biologie und anderen Gebieten ist. (Nach Sedgewick & Wayne, Einführung in die Programmierung mit Java.)

(b) Nun sollen Sie die gelaufenen Pfade auf ein Fenster zeichnen. Falls ein Pfad erfolgreich war, malen Sie ihn in grün, ansonsten in rot. Dazu müssen Sie sich den Pfad während der Simulation in einer `LinkedIntList` merken und in einem zweiten Schritt (wenn Sie wissen, ob der Pfad erfolgreich war) in der richtigen Farbe zeichnen. Jeden Schritt des Wolfs können Sie als `int` kodiert der Liste hinzufügen, beispielsweise kodiert als 0=Norden, 1=Osten, usw. Es ist Ihnen überlassen, ob Sie die Pfade schrittweise (animiert) zeichnen, oder jeden Pfad auf einmal.