Algorithmen und Datenstrukturen 1

ALGO1 · SoSe-2021 · tcs.uni-frankfurt.de/algo1

⭐-Aufgabe: 3-Farben Algorithmus

Revision 393abf7 (2021-06-08)

For an English version of this exercise, see [Erickson, page 210].

Eine Klasse von Suchalgorithmen auf Graphen, die Breitensuche und Tiefensuche verallgemeinern, wurde 1975 von Edsger Dijkstra, Leslie Lamport, Alain Martin, Carel Scholten und Elisabeth Steffens beschrieben. Die Autor:innen haben diese Algorithmen untersucht, um damit einen automatischen *garbage collector* zu entwerfen (siehe wikipedia). Anstatt markierte und unmarkierte Knoten zu verwalten, verwaltet ihr Algorithmus eine Farbe für jeden Knoten, entweder weiß, grau oder schwarz. Im Folgenden stellen wir uns einen als Adjazenzliste gegebenen, ungerichteten Graphen G vor.

```
procedure ThreeColorSearch(s)
    färbe alle Knoten weiß
    färbe s grau
    while mindestens ein Knoten ist grau do
        ThreeColorStep
```

```
procedure ThreeColorStep
    v ← irgendein grauer Knoten
    if v hat keine weißen Nachbarn then
        färbe v schwarz
    else
        w ←irgendein weißer Nachbar von v
        w.π ← v
                        ▷ v ist der Elternknoten von w.
        färbe w grau
```

a) Beweise, dass ThreeColorSearch zu jedem Zeitpunkt die folgende Invariante erhält: Kein schwarzer Knoten ist zu einem weißen Knoten benachbart. *(Hinweis: Das sollte einfach sein.)*

b) Beweise Folgendes: Wenn ThreeColorSearch(s) terminiert, dann sind alle von s erreichbaren Knoten schwarz, alle nicht von s erreichbaren Knoten weiß, und die zu den Eltern zeigenden Kanten $(v, v.\pi)$ definieren einen Baum, der alle Knoten der Zusammenhangskomponente von s aufspannt. *(Hinweis: Wenn man den Algorithmus mit DFS/BFS vergleicht, kann man sich intuitiv vorstellen, dass die schwarzen Knoten „markiert" sind und die grauen Knoten „auf dem Stapel/in der Warteschlange". Ein Unterschied ist, dass ThreeColorStep nicht im selben Aufruf* alle *Kanten abarbeiten muss, die aus einem Knoten v rausgehen.)*

c) Die folgende Variante von ThreeColorSearch verwaltet die grauen Knoten auf einem Stapel. Beweise, dass diese Variante äquivalent zu DFS ist, das heißt, die Knoten werden in genau derselben Reihenfolge entdeckt und die Elternbeziehungen sind identisch. *Hinweis: Die Reihenfolge der letzten zwei Zeilen von ThreeColorStackStep ist wichtig!*

```
procedure ThreeColorStackSearch(s)
    färbe alle Knoten weiß
    färbe s grau
    lege s auf den Stapel
    while mindestens ein Knoten ist grau do
        ThreeColorStackStep
```

```
procedure ThreeColorStackStep
    nimm v vom Stapel
    if v hat keine weißen Nachbarn then
        färbe v schwarz
    else
        w ←irgendein weißer Nachbar von v
        w.π ← v
        färbe w grau
        lege v auf den Stapel
        lege w auf den Stapel
```

d) Die folgende Variante von ThreeColorSearch verwaltet die grauen Knoten in einer Warteschlange. Beweise, dass diese Variante ***nicht*** äquivalent zu BFS ist. *Hinweis: Die Reihenfolge der letzten zwei Zeilen von ThreeColorQueueStep ist nicht wichtig!*

```
procedure ThreeColorQueueSearch(s)
    färbe alle Knoten weiß
    färbe s grau
    schiebe s in die Warteschlange
    while mindestens ein Knoten ist grau do
        ThreeColorQueueStep
```

```
procedure ThreeColorQueueStep
    ziehe v aus der Warteschlange
    if v hat keine weißen Nachbarn then
        färbe v schwarz
    else
        w ←irgendein weißer Nachbar von v
        w.π ← v
        färbe w grau
        schiebe v in die Warteschlange
        schiebe w in die Warteschlange
```

e) (sehr schwer) Hier sei G ein gerichteter Graph. Wir nehmen nun an, dass ein zweiter Prozess Kanten zu G hinzufügt, während ThreeColorSearch noch läuft. Diese neuen Kanten könnten die Farbinvariante zerstören, die in Teil a) beschrieben ist. Daher könnte es jetzt sein, dass ThreeColorSearch nicht mehr korrekt ist. Das heißt, es könnte sein, dass der Algorithmus zwar terminiert, aber es trotzdem noch Knoten gibt, die von s erreichbar sind und weiß sind. Wenn wir einen *garbage collector* implementieren wollen, wäre das fatal, denn hier würden wir „weiß“ mit „unerreichbar und daher löschbar“ gleichsetzen wollen. Wenn der andere Prozess auf die Farbinvariante Rücksicht nimmt und diese explizit wiederherstellt, wann immer sie verletzt würde, dann können wir den ThreeColorSearch Algorithmus trotzdem sicher verwenden. Das möchten wir jetzt zeigen. Um die zwei parallelen Algorithmen zu modellieren, verwenden wir die *either/or* Syntax in GarbageCollect; wie bei *if/then/else* verzweigt das Programm hier, aber welcher Zweig verfolgt wird, wird nicht vom Programm selbst entschieden, sondern vom Betriebssystem.[1]

[1] Das ist eine dramatische Vereinfachung, sowohl von paralleler Programmierung als auch von *garbage collection*. Mehrfädige Programmiersprachen wie Lua und Go benutzen einen viel komplexeren *mark and sweep* Algorithmus als *garbage collector*. Mathematisch wichtig für *either/or* ist, dass das Betriebssystem zwar nicht garantiert, wie oft hintereinander und in welcher Reihenfolge die zwei Programmzweige gewählt werden. Aber jeder Zweig wird immer wieder *irgendwann* gewählt, das heißt, nach endlicher Zeit. — Das Betriebsystem darf also ***nicht*** für immer den einen Zweig wählen und den anderen „vergessen“.

```
procedure GarbageCollect(s)
    färbe alle Knoten weiß
    färbe s grau
    while mindestens ein Knoten ist grau do
        either
            CollectStep
        or
            Mutate
```

```
procedure CollectStep
    v ← irgendein grauer Knoten
    if v hat keine weißen Nachbarn then
        färbe v schwarz
    else
        w ←irgendein weißer Nachbar von v
        färbe w grau
```

```
procedure Mutate
    u ← irgendein Knoten
    w ← irgendein Knoten
    if (u, w) ist keine Kante then
        füge (u, w) als Kante hinzu
        if u ist schwarz und w ist weiß then
            färbe u grau
        if u ist weiß und w ist schwarz then
            färbe w grau
```

Beweise, dass GarbageCollect irgendwann terminiert, und dass dann jeder von s erreichbare Knoten schwarz gefärbt ist und jeder von s nicht erreichbare Knoten weiß gefärbt ist.

f) (sehr schwer) Hier sei G wieder ein gerichteter Graph. Anstatt schwarze Knoten grau zu färben, soll Mutate jetzt die Farbinvariante aufrechterhalten, indem manche *weißen* Knoten grau gefärbt werden:

```
procedure Mutate
    u ← irgendein Knoten
    w ← irgendein Knoten
    if (u, w) ist keine Kante then
        füge (u, w) als Kante hinzu
        if u ist schwarz und w ist weiß then
            färbe w grau
        if u ist weiß und w ist schwarz then
            färbe u grau
```

Beweise, dass GarbageCollect irgendwann terminiert, und dass dann s schwarz gefärbt ist, jeder von einem schwarzen Knoten erreichbare Knoten wiederum schwarz ist, und jeder nicht von einem schwarzen Knoten erreichbare Knoten weiß ist.

Hinweise zur Abgabe. Die Abgabe soll wie immer per PDF erfolgen, allerdings sind diesmal ausschließlich mathematische Beweise gefordert. Um einen ⭐ zu erhalten, müssen a) bis d) zielgerichtet, nachvollziehbar, lesbar, vollständig, und korrekt gelöst sein. Zielgerichtet heißt, dass alle Sätze und Satzteile in der Abgabe aktiv zum Ziel hinführen, nämlich der Lösung der jeweiligen Aufgabe.