Institut für Informatik

Lehrstuhl für Technische Informatik
Prof. Dr. Björn Scheuermann

Digitale Systeme 2021
C-Programmierprojekt

1 Aufgabenstellung

Im diesjährigen C-Programmierprojekt ist es Ihre Aufgabe, die zufällig aussehenden Bewegungen einer Ameise nach bestimmten Regeln zu simulieren. Die Ameise läuft über Brücken und Wege aus Stöckchen, welche auf dem Waldboden verteilt sind. Bei jeder Kreuzung hinterlässt die Ameise ein Duftspur, welche sich mit den Gerüchen vermischt, die schon zuvor da waren. Anhand dieser Duftspur entscheidet die Ameise, wohin sie als nächstes laufen will. Nach einiger Zeit ist sie erschöpft und kann sich nicht weiter bewegen. Ihr Programm soll anhand eines festen Netzes an Wegen und einer Startposition die Bewegungen der Ameise simulieren um deren Endposition zu bestimmen, ebenso wie die Duftspuren, welche sie hinterlassen hat.

Das Netz an Bewegungsmöglichkeiten der Ameise lässt sich als (ungerichteter) Graph darstellen. Stöckchen, über welche die Ameise laufen kann, sind Kanten des Graphen. Kreuzungen, an denen mehrere Kanten zusammenkommen, nennt man Knoten. Jedem Knoten wird eine eindeutige ID zugeordnet. Kanten verbinden stets genau zwei Knoten.

Die Duftspur, mit der die Ameise Knoten kennzeichnet, wird durch eine Zahl ausgedrückt, welche wir im Folgenden als Markierung bezeichnen. Jeder Knoten hat eine Markierung, deren Wert im Bereich von 0 bis $2^{32}-1$ liegt. Nachdem die Ameise über einen Knoten geht, wird dessen Markierung um eins hochgezählt. Wenn der Maximalwert erreicht ist, beginnt die Zählung wieder bei 0.

Die Ameise fängt an einem festgelegten Startknoten A an. Um zu entscheiden, wohin sie gehen soll, schaut sie sich die Markierung des momentanen Knotens und die lexikographische Reihenfolge[1] der IDs der Nachbarknoten an. Die Ameise folgt von einem Knoten K aus als nächstes der Kante zu demjenigen Nachbarknoten von K, dessen Position in der lexikographischen Ordnung der IDs aller Nachbarn von K der momentanen Markierung von K modulo der Anzahl von Nachbarknoten von K entspricht. Wenn also zum Beispiel der Markierungswert von K derzeit gleich 2 ist und es K drei oder mehr Nachbarn hat, dann bewegt sich die Ameise zu demjenigen Nachbarknoten mit der drittniedrigsten ID. (Achtung: auch die Werte der Markierungen fangen bei 0 an, deshalb entspricht ein Markierungswert von 2 dem dritten Nachbarknoten!)

Der Graph ist ungerichtet, das heißt die Ameise kann jede Kante in beide Richtungen verwenden. Die Markierung eines Knotens wird erst verändert, wenn die Ameise ihn verlässt. Das gilt auch für den Startknoten. Nach einer vorgegebenen Anzahl I an Schritten ist die Ameise

[1] `https://de.wikipedia.org/wiki/Lexikographische_Ordnung`. Allgemein gilt: Ziffern vor Buchstaben; Buchstaben werden alphabetisch sortiert; kürzere Zeichenketten kommen vor längeren, deren Präfix sie sind (Beispiel: „aa“ vor „aa5“ vor „aa5x“ vor „aaa“).

erschöpft und hört auf sich zu bewegen. Die Markierung des Knotens, auf dem der Weg der Ameise endet, wird dann nicht noch einmal aktualisiert. Ihr Programm soll dann alle Knoten des Graphen, deren Markierungen sowie die Endposition E der Ameise in der unten vorgegebenen Weise ausgeben.

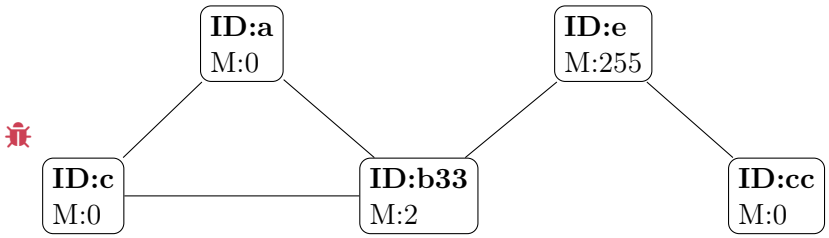

Abbildung 1: Beispielgraph mit Markierungen und KnotenIDs.

2 Ein- und Ausgabeformate

Zuerst sollen Sie den Graphen von der Standardeingabe eingelesen. Die Eingabe ist eine ASCII-codierte Zeichenfolge. Zahlenwerte werden in der Ein- und Ausgabe als Dezimalzahlen in Form einer ASCII-Ziffernfolge dargestellt.

Jede Zeile der Eingabe (mit Ausnahme der letzten beiden Zeilen, siehe unten) hat das folgende Format und beschreibt einen Knoten:

$$KnotenID : Nachbarknoten_1, ..., Nachbarknoten_m \backslash n$$

Am Beginn der Zeile steht die ID des Knotens. Sie ist eine Zeichenkette mit einer beliebigen Länge größer 0, in der alle Zeichen aus der Menge $\{0-9, a-z\}$ stammen (also keine Großbuchstaben, keine Leerzeichen, keine Umlaute, keine Sonderzeichen!). Nach dem Doppelpunkt folgt eine Aufzählung von IDs von Nachbarknoten. Diese List an Nachbarn muss nicht vollständig sein, denn die Nachbarschaftsbeziehung zwischen den Knoten kann auch nur in der Nachbarliste des anderen Knotens beschrieben sein. Um Platz zu sparen wird kommt jede Kante in der Eingabe nur einmal vor. Wenn dieselbe Kante in beide Richtungen angegeben wird, handelt es sich um einen Fehler in der Eingabe.

Die aufgezählten IDs von Nachbarknoten sind durch jeweils genau ein Komma voneinander getrennt. Die nächste Zeile ist durch ein Newline-Zeichen[2] abgetrennt. Leerzeichen sind in der Eingabe unzulässig. Es kann vorkommen, dass ein Knoten nur in Aufzählungen seiner Nachbarknoten erwähnt wird, aber keine eigene Zeile hat (siehe beim Beispiel unten der Knoten mit der ID c).

Sollte nichts anderes festgelegt sein, dann hat ein Knoten zu Beginn die Markierung 0. Knoten kann aber in der Eingabe eine andere Markierung zugeordnet werden. In diesem Fall werden ein Bindestrich und der entsprechende Wert am Ende der Zeile, wie oben definiert, angehängt. Wenn für einen Knoten keine Nachbarn aufgezählt werden, kann es trotzdem eine Zeile in der Eingabe für ihn geben, die ihm eine Markierung zuweist (dies passiert bei Knoten e im Beispiel unten).

[2] auch als Line Feed oder LF bezeichnet; das Zeichen mit der Nummer dezimal 10 in ASCII; wir schreiben das hier wie in der Spreche C üblich als „\n“

Am Ende der Eingabe werden noch zwei Startwerte für das Programm festgelegt. Zunächst gibt eine Zeile die ID des Anfangsknotens A an. An dieser Position startet die Ameise. Die letzte Zeile beinhaltet mit Anzahl I an Schritten im Graphen, die die Ameise laufen soll. Das Format der letzten beiden Zeilen lautet:

$$A : AnfangsknotenID\backslash n$$
$$I : AnzahlSchritte\backslash n$$

Hier ein Beispiel für eine Eingabe und die zugehörige Ausgabe für den Graphen aus Darstellung 1:

stdin	**stdout**
`a:b33,c`	`a:1`
`b33:c,e-2`	`b33:3`
`e:-255`	`c:1`
`cc:e`	`cc:1`
`A:c`	`e:256`
`I:5`	`E:e`

Dieser Graph enthält fünf Knoten (*a*, *b33*, *c*, *cc* und *e*). Knoten *a* ist mit *b33* und *c* verbunden. Knoten *b33* hat zusätzlich Kanten zu *c* und *e*. Knoten *cc* hat eine Kante mit *e*. Kanten sind bidirektional. Sie existieren, wenn ein Zeile sie beschreibt. Dieselben beiden Knoten dürfen nicht mehrfach über Kanten verbunden sein. Alle Knoten haben am Anfang die Markierung 0 (default), außer *b33* mit Markierung 2 und *e* mit Markierung 255. Startpunkt ist *c*. Die Ameise ist fertig, wenn sie fünf Kanten weit gegangen ist.

Wenn die Eingabe gültig ist und Ihr Programm erfolgreich beendet wird, muss es das Ergebnis in einer genau vorgegebenen Weise auf die Standardausgabe schreiben. Dabei werden alle Knoten des Graphen und ihre jeweilige Markierung am Ende der Berechnung ausgegeben. Es wird ein Zeile für jeden Knoten des Graphen ausgegeben. Die Zeile enthält die ID des Knotens, dann einen Doppelpunkt und schließlich den Markierungswert des Knotens am Ende der Berechnung. Die Reihenfolge, in der die Knoten ausgegeben werden, ist irrelevant.

Zuletzt soll die Endposition der Ameise ausgegeben werden. Die entsprechende Zeile muss folgendes Format haben:

$$E : KnotenID\backslash n$$

Wenn Ihr Programm eine Eingabe erhält, die nicht der obigen Spezifikation genügt, soll es gar nichts auf die Standardausgabe schreiben. Auf die Standardfehlerausgabe (`stderr`) soll dann eine Fehlermeldung geschrieben werden; als Returncode soll in diesem Fall ein Wert ungleich null zurückgegeben werden. Falls Ihr Programm regulär beendet wird, soll als Returncode null zurückgegeben werden.

Insbesondere sollten folgende Fälle als falsche Eingabe gewertet werden:

- Ein anderes Zeichen als eine 0–9, a–z, A, I, ein Doppelpunkt, ein Komma, ein Minus, oder ein Linefeed tritt auf.
- Der Wert für die Markierung liegt nicht im gültigen Wertebereich $\{0, \ldots, 2^{32} - 1\}$.
- Der Anfangsknoten existiert nicht im Graphen.
- Die Formatierung in mindestens einer Zeile entspricht nicht der Spezifikation

Der Einfachheit halber *dürfen* Sie neben einem einzelnen Linefeed (LF) auch ein Carriage Return (CR) oder ein „CR LF“ als Zeilenumbruch akzeptieren. Wir verzichten darauf, diesbezüglich zu testen.

3 Beispieldaten

Wir stellen Ihnen in Moodle einige Beispiele für Ein- und Ausgabepaare (`example01.stdin`, `example01.stdout`) zur Verfügung, anhand derer Sie selbst überprüfen können, ob Ihr Programm gewisse Eingaben korrekt bearbeitet. Da keine bestimmte Sortierung der Ausgabedaten verlangt wird, müssen Sie Ihre Ausgabedaten ggf. mittels `sort` (GNU coreutils) sortieren, bevor Sie eine bytegenaue Übereinstimmung mit unseren Ausgabedaten mittels `diff` überprüfen können. Die Beispiel-Eingaben decken viele – aber nicht alle – Grenzfälle ab, mit denen Ihr Programm zurechtkommen muss.

Für die Überprüfung Ihrer Abgabe werden andere Datensätze zum Einsatz kommen.

Bezüglich falsch formatierter Eingaben oder fehlerhafter Aufrufe müssen Sie selbst kreativ werden. Versuchen Sie, Ihr Programm mithilfe bewusst bösartiger Eingaben zum Absturz zu bringen und vergewissern Sie sich, dass für jeden möglichen Programmdurchlauf *niemals* Speicherschutzverletzungen (eng. „Segmentation Fault“) auftreten, nur gültige Lese- und Schreiboperationen auf dem Speicher ausgeführt werden und immer der gesamte vom Heap allozierter Speicher vor Programmende explizit freigegeben wird. *Dies muss auch für beliebige fehlerhafte Eingaben sichergestellt sein.*

4 Hinweise zum Algorithmus

Die Aufgabenstellung lässt sich auf folgende Weise lösen.

1. Erzeugen Sie während des Einlesens von der Standardeingabe eine Liste aller Knoten-IDs und ordnen Sie jedem Knoten eine eindeutige fortlaufende Nummer zu.

2. Erzeugen Sie anhand der Eingabe eine Nachbarschaftsmatrix für den Graphen, in der das Programm nachsehen kann, welche Knoten miteinander verbunden sind.

3. Führen Sie eine zusätzliche Tabelle, in der die Markierungen der Knoten verzeichnet sind.

4. Bewegen Sie die Ameise durch das Netz an Wegen und verändern Sie dabei die Tabelle mit den Markierungen.

Es gibt auch andere mögliche Lösungsstrategien. Jede von Ihnen eingereichte Lösung ist akzeptabel, solange sie nicht wesentlich ineffizienter als der von uns vorgeschlagene Lösungsweg ist und Sie im Abtestat begründen können, warum Ihre Lösung korrekte Resultate liefert. Beachten Sie bei Ihrer Implementierung den sorgsamen Umgang mit Arbeitsspeicher. Gehen Sie zudem davon aus, dass Sie die Eingabedaten nur einmal sequenziell lesen können, dass also `fseek` und verwandte Operationen auf `stdin` nicht möglich sind. Vielleicht ist die Eingabe ja gar keine Datei sondern wird in Ihr Programm „hineingepipet“.

5 Zeitplan

Sie können sofort mit der Bearbeitung der Aufgabe beginnen. Bis zur finalen Abgabefrist haben Sie beliebig viele Abgabeversuche, eine erste Abgabe ist ab sofort möglich. Wir bemühen uns, Ihnen nach jedem Abgabeversuch kurzfristig Rückmeldung zu geben, ob Ihre Lösung korrekt ist oder, falls nicht, inwiefern sich Ihr Programm falsch verhält. **Nach der finalen Abgabefrist werden keinerlei weitere Abgaben mehr berücksichtigt.**

Wir ermutigen Sie ausdrücklich, so früh wie möglich mit der Bearbeitung der Aufgabe zu beginnen und einen ersten sinnvollen Lösungsversuch bis Anfang Juli abzugeben. Je nach Aufkommen von Abgaben können wir wenige Wochen vor der finalen Deadline nicht mehr garantieren, kurzfristig individuelles Feedback zu geben, das Ihnen auch wirklich weiter hilft. Da eine vollständig korrekte Lösung für das Bestehen erforderlich ist, sind erfahrungsgemäß fast immer mehrere Abgaben notwendig[3].

Wenn Sie eine korrekte Lösung eingereicht haben, müssen Sie uns für die Erbringung der Modulteilleistung zusätzlich noch im Rahmen eines kurzen persönlichen Abtestats Ihre Lösung erläutern und Fragen dazu beantworten. Die Abtestate werden online über Zoom stattfinden.

Datum	Beschreibung
21.07.2021	Empfohlene Deadline für eine erste Abgabe; bei Abgabe bis zu diesem Termin können wir im Falle von Problemen mit der Abgabe gewährleisten, dass nach einer ersten Rückmeldung noch Zeit zur Nachbesserung bis zur endgültigen Deadline bleibt.
01.09.2021 (23:59 Uhr MESZ)	Spätestmögliche Deadline für die Abgabe der Endversion, die alle Kriterien vollständig erfüllen muss.
13.09.2021 – 15.09.2021	In diesem Zeitraum finden die mündlichen Testate in Einzelsitzungen statt. Eine frühere Abnahme ist nach individueller Absprache möglich.

6 Wichtige Hinweise und weitere Anforderungen

Beachten Sie bei der Erstellung Ihres Programms **unbedingt** folgende Punkte:

- Halten Sie sich strikt an das oben definierte Ausgabeformat und die Aufrufkonventionen. Wir überprüfen Ihr Programm automatisiert auf die Korrektheit der ausgegebenen Lösungen. *Eine falsch formatierte Ausgabe wird nicht als korrekt anerkannt!* Im Zweifelsfall fragen Sie *rechtzeitig vor Abgabe* nach!
- Verwenden Sie keine Bibliotheken außer der C-Standardbibliothek wie sie in der 2011er Ausgabe des ISO C Standard beschrieben ist. Insbesondere Nutzer von Softwareprodukten aus Redmond weisen wir darauf hin, dass Funktionen, die einen DromedaryCaseNamen tragen (wie `StrToInt`), sehr wahrscheinlich nicht Teil der C-Standardbibliothek sind.

[3] Die Deadlines sind ernst gemeint. Von einer Abgabe ein paar Stunden, Minuten oder Sekunden vor der finalen Abgabefrist raten wir dringend ab. Technische Probleme bei der Abgabe sind kein Grund für eine Ausnahme von der Frist.

- Ihr Programm soll als eine einzige C-Quelldatei mit dem Namen `loesung.c` in gültigem ISO C11 geschrieben sein. Ihr Programm muss sich mit folgendem einzelnen gcc-7.5.0-Compileraufruf[4] fehlerfrei übersetzen und linken lassen:
  `gcc -o loesung -O3 -std=c11 -Wall -Werror -DNDEBUG loesung.c`
  Um Fehler bei der Implementierung zu vermeiden, empfiehlt es sich, zusätzlich die Compiler-Flags `-Wextra` und `-Wpedantic` zu verwenden. Verlangt wird dies aber nicht.

- Ihr Programm muss auf der Kommandozeile ohne grafische Oberfläche lauffähig sein und ohne jegliche Kommandozeilenargumente laufen, d. h. es soll mit folgendem Aufruf (in `sh`, `bash` oder `zsh`) ausführbar sein:
  `cat example.stdin | ./loesung | tee myresult.stdout` Wenn Sie die Ausgabe Ihres Programms mit unseren Beispielausgaben vergleichen wollen, können Sie zum Beispiel folgenden Aufruf verwenden:
  `cat example01.stdin | ./loesung | sort | diff <(sort example01.stdout) -`
  (Der Strich am Ende gehört zum Aufruf dazu.)

- Legen Sie Ihr Programm so aus, dass es mit beliebig großen Eingabedaten umgehen kann. Hierfür ist es unbedingt notwendig, dass Sie dynamische Speicherverwaltung einsetzen. Wir werden Ihr Programm mit *großen* Datensätzen testen!

- Ihr abgegebenes Programm wird automatisch auf Speicherlecks überprüft. Stellen Sie sicher, dass es keine Speicherlecks aufweist. Dies können Sie beispielsweise mit dem Werkzeug `valgrind`[5] bewerkstelligen. Geben Sie jeglichen dynamisch zugewiesenen Speicher vor Programmende korrekt wieder frei. Seien Sie sich bewusst, dass Ihr Programm auf Ihrem Rechner richtige Ausgaben erzeugen, aber trotzdem Speicherfehler und Speicherlecks enthalten kann, die Sie ohne Speicherdebugging sehr wahrscheinlich nicht erkennen werden.

- Stellen Sie sicher, dass Ihr Programm *niemals* eine Speicherschutzverletzung (eng. „segmentation fault“, „segfault“) auslöst, egal welche gültige oder ungültige Eingabe es erhält. Achten Sie darauf, dass Ihr Programm auch auf Systemen mit sehr wenig Speicher keine Speicherschutzverletzung auslöst. Verlassen Sie sich *niemals* darauf, dass ein Aufruf von `malloc` tatsächlich erfolgreich neuen Speicher alloziert. Falls `malloc` fehlschlägt, darf Ihr Programm natürlich mit einer Fehlerbehandlung und einem Returncode ungleich 0 abbrechen.

- Wir werden alle abgegebenen Lösungen benchmarken. Die ressourcensparsamsten Lösungen werden am Ende des Semesters mit einem kleinen Preis ausgezeichnet. Wir werden hierfür die Lösungen hinsichtlich ihrer Ausführungszeit und ihres Speicherverbrauchs untersuchen.

- Kommentieren Sie Ihren Quelltext in angemessenem Umfang. Im Testat werden Sie Ihren Quelltext detailliert erklären müssen. Dabei können Kommentare helfen. Erforderlich sind Kommentare natürlich nur an Stellen, an denen Ihr Code nicht in trivialer Weise für sich selbst spricht.

[4] Den gcc 7.5 finden Sie z.B. auf den Rechnern des Berlin-Pools ([alex|britz|buch|buckow|...|gruenau1|gruenau2|...].informatik.hu-berlin.de) installiert.
[5] `http://valgrind.org/`

- Wir werden Feedback zu Ihren Lösungsversuchen ausschließlich als persönliche Moodle-Nachrichten versenden. Achten Sie darauf, dass Sie diese Nachrichten rechtzeitig lesen.
- Wir empfehlen Ihnen, Ihr Programm vor jedem Abgabeversuch selbst mit der gegebenen Compiler-Version sowie den vorgegebenen Flags zu übersetzen und mit allen von uns zur Verfügung gestellten Testdaten und unter Verwendung von Valgrind zu testen.
- Für das Bestehen ist es nicht erforderlich, ein besonders effizientes Programm zu schreiben. Dennoch sind wir aus naheliegenden praktischen Gründen gezwungen, bei der Überprüfung sowohl die CPU-Zeit eines Prozesses als auch die Menge an gleichzeitig verwendetem RAM[6] zu begrenzen. Wir haben die Grenzen hierfür großzügig gewählt, so dass die meisten Implementierungen wahrscheinlich nicht ansatzweise in deren Nähe kommen werden. Der Transparenz halber geben wir sie hier an:
  - Ihr Programm sollte jedes der von uns bereitgestellten Beispielein- und Ausgabepaare auf einem der Pool-Rechner in jeweils maximal 50 s CPU-Zeit bearbeiten können.
  - Ihr Programm sollte zu keinem Zeitpunkt mehr als $\max\{10^6,\, 16n^2 + 500n + L\}$ Byte RAM benötigen, wobei n die Anzahl an Knoten im Graphen und L die Summe der Länge aller Knoten-IDs bezeichnet.
- Bearbeiten Sie diese Programmieraufgabe *alleine*. Gerne dürfen Sie Lösungsstrategien, auftretende Probleme, etc., *verbal* mit Ihren Kommilitonen diskutieren. Aber *teilen Sie keinen Code!*

6.1 Valgrind

Valgrind ist ein sehr vielseitiges Werkzeug. Im einfachsten Fall testen Sie Ihr Programm mittels
`cat example01.stdin | valgrind ./loesung 1> /dev/null`

Wenn Ihr Programm keine Fehler in der Speicherverwaltung aufweist, sollten in der Valgrind-ausgabe unabhängig von der Eingabe stets folgende zwei Zeilen vorkommen:

```
==1234== All heap blocks were freed - no leaks are possible
==1234== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

(Statt „1234“ wird dort in der Regel eine andere Zahl stehen.)

6.2 SSH

Ob Ihr Code compiliert, hängt gegebenenfalls von der verwendeten gcc-Version ab. Machen Sie sich am besten mit `ssh` vertraut und testen Sie vor jeder Abgabe, ob sich Ihr Code auf einem `gruenau` oder einem Pool-Rechner übersetzen lässt. Verwenden Sie zum Beispiel folgenden Aufruf auf einer Linux-Kommandozeile, um Ihre im aktuellen Verzeichnis liegende Lösung direkt auf `gruenau6` zu compilieren:

```
cat loesung.c | ssh myHUusername@gruenau6.informatik.hu-berlin.de \
'cat > ~/loesung.c && gcc-7 -o loesung -O3 -std=c11 -Wall -Werror -DNDEBUG loesung.c'
```

[6] „maximum resident set size“, MRSS

6.3 Randfälle

- Die leere Eingabe ist keine gültige Eingabe. Das gleiche gilt für einen Graphen ohne Knoten.
- Sie dürfen annehmen, dass ein Graph nicht mehr als 2^{32} Knoten haben wird.
- Eine Markierung mit dem Wert 0 in der Eingabe ist erlaubt.
- Die Anzahl an Schritte I kann auch 0 sein, wird aber nie größer als 2^{32}.
- Keine zwei Knoten dürfen die gleiche ID haben und für jeden Knoten darf nur einmal eine ihn beschreibende Zeile in der Eingabe vorkommen. Zum Beispiel dieser Fall ist keine valide Eingabe:

$$\begin{aligned}&\dots\\&a : bb, cc\backslash n\\&a : bb, dd, zz\backslash n\\&\dots\end{aligned}$$

- Knoten dürfen keine Verbindungen zu sich selbst haben. Tritt dies auf ist es als Fehler zu bewerten.
- Dieselbe Kante darf nicht mehrfach spezifiziert werden. Für zwei Knoten X und Y, zwischen denen eine Kante existiert, darf also nur entweder X als Nachbar von Y aufgelistet werden oder Y als Nachbar von X, jedoch nicht beides. Passiert dies doch, ist es als Fehler zu bewerten.
- Wenn ein Knoten als Startknoten gewählt wird, der keine Nachbarn hat, so verweilt die Ameise auf diesem Knoten für die gesamte Ausführung. Die Farbe des Knoten wird dann trotzdem nach jedem Schritt hochgezählt.
- Wenn die Eingabe formal richtig ist, Ihr Programm aber nicht genug Speicher allozieren kann, soll es mit einer entsprechenden Fehlermeldung abbrechen und bisher allozierten Speicher freigeben. Verwenden Sie Stack-Speicher nicht exzessiv, sondern besser Heap-Speicher.

Sollten Sie allgemeine Fragen zur Aufgabenstellung haben, stellen Sie diese bitte im Moodle-Forum (`https://moodle.hu-berlin.de/mod/forum/view.php?id=2158042`) und nicht via E-Mail oder persönlicher Moodle-Nachricht, damit auch andere von der Antwort profitieren können.

Viel Erfolg!