Aufgabe 3: Traumdreiecke

1 Lösungsidee

Für die Lösung dieses Problems kann man einen evolutionären Algorithmus verwenden. Ein evolutionärer Algorithmus enthält im Allgemeinen folgende Bestandteile: Initialisierung, Fitnessfunktion, Selektion, Rekombination, Mutation und Abbruchkriterium. Diese Bestandteile müssen lediglich auf dieses Problem angepasst werden:

Die Traumdreiecke stellen die Individuen in dieser Evolution dar. Mehrere Traumdreiecke werden als Population angesehen.

Die Farben der Murmeln sind die Eigenschaften der Traumdreiecke, welche sich durch Mutationen verändern können. Die Traumdreiecke werden durch eine Fitnessfunktion bewertet und dann selektiert. Aus den selektierten Individuen entsteht durch Rekombination die neue Generation. Jede neue Generation kommt dem Ziel näher: In dem Traumdreieck gibt es keine gleichseitigen Dreiecke von Murmeln, die aus denselben Farben bestehen (Abbruchbedingung).

Vorteil von evolutionären Algorithmen ist, dass man diese parallelisieren kann. Ein Nachteil ist jedoch, dass man nie genau sagen kann, ob die gefundene Lösung die optimale ist. Dies könnte nur ein Brute-Force-Algorithmus beweisen. Da das Durchprobieren aller Möglichkeiten jedoch zu viel Zeit benötigen würde, habe ich mich für den evolutionären Algorithmus entschieden.

1.1 Initialisierung

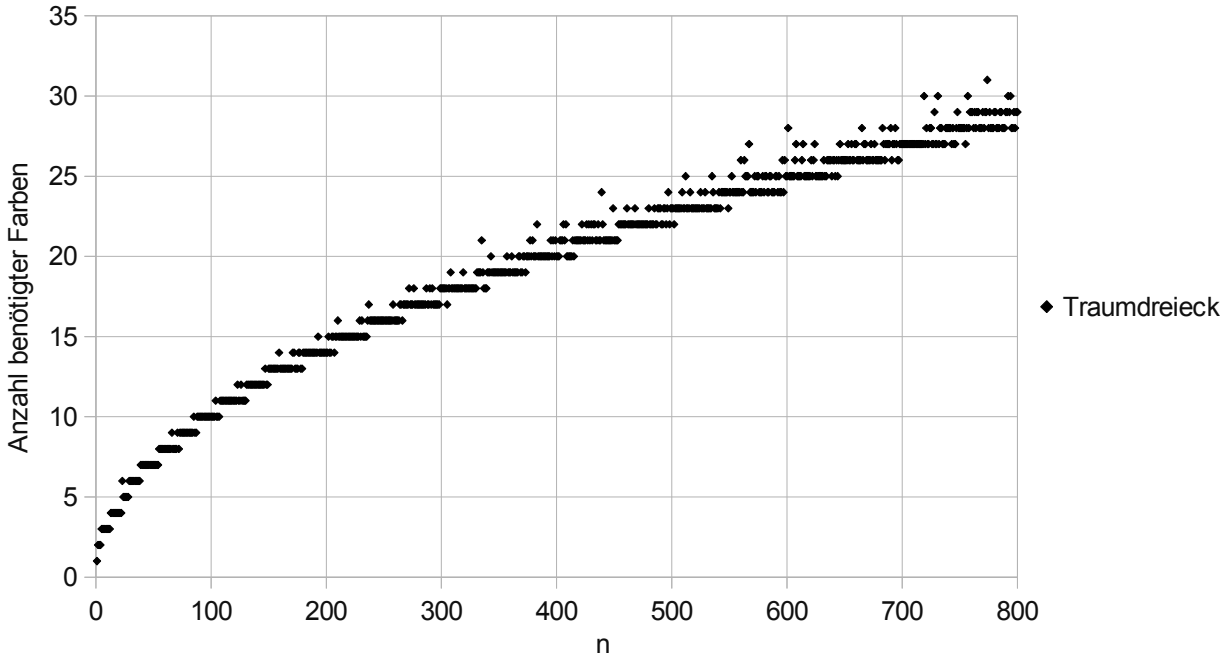
Dieser Schritt wird nur einmal und zwar vor dem „Beginn der Evolution“ ausgeführt.

Die Murmeln der Traumdreiecke werden mit zufälligen Farben initialisiert. Die Anzahl der benötigten Farben ist von n abhängig:

$$Anzahl\,benötigter\,Farben = \begin{cases} \lceil \sqrt{n} \rceil & \text{für } n < 9 \\ \lfloor \sqrt{n} \rfloor & \text{für } n \geq 9 \end{cases}$$

Die Traumdreieck können also mit dieser Anzahl der benötigten Farben initialisiert werden. Die Murmeln eines Traumdreiecks mit z. B. zehn Ebenen bekommen jeweils einen zufälligen Farbwert zwischen 1 und 3, weil $\sqrt{10} \approx 3{,}2$ und $\lfloor \sqrt{10} \rfloor = 3$.

Ich hatte zunächst ein Programm geschrieben, dass die Anzahl der Farben stetig erhöht, bis eine Lösung gefunden wird. Die ermittelten Wert habe ich dann in einem Diagramm darstellen lassen:



Da der Verlauf der Punkte Ähnlichkeiten mit der Wurzelfunktion hat, lag die Vermutung nahe, dass die Funktion für die benötigten Farben durch $\lceil\sqrt{n}\rceil$ und $\lfloor\sqrt{n}\rfloor$ beschrieben werden kann.

Da man bei zwei Ebenen nicht mehr mit einer Farbe auskommt und es eine Lösung bei zehn Ebenen mit nur drei Farben gibt, bin ich auf die obige Formel für die benötigten Farben gekommen.

Aus Effizienzgründen, habe ich festgelegt, dass für $n \geq 9$ die Formel $\lfloor\sqrt{n+1}\rfloor$ für die Anzahl der benötigten Farben gilt. Da evolutionäre Algorithmen meist mit dem Zufall arbeiten, dauert es auch meist lange, bis eine optimale Lösung gefunden ist. Gerade bei solchen „Grenzfällen“ kann die Laufzeit sehr hoch sein.

Da ich nun wusste wie viele Farben benötigt werden, konnte ich mein Programm darauf abstimmen, sodass immer Färbungen mit der geringsten Anzahl benötigter Farben gefunden werden.

1.2 Fitnessfunktion

Dies ist der erste Schritt im evolutionären Algorithmus: Die Individuen werden danach bewertet, wie gut sie an die Umwelt angepasst sind. Ein Traumdreieck, welches keine Fehler enthält ist optimal angepasst und bekommt eine Fitness von 0.

Eine schlechte Fitness wird durch eine negative Zahl dargestellt. Ein Traumdreieck erhält eine schlechte Fitness, wenn es gleichseitige Dreiecke von Murmeln enthält, die dieselbe Farbe besitzen. Je mehr Fehler ein Traumdreieck enthält, desto schlechter ist die Fitness.

Eine Färbung für das Traumdreieck ist gefunden, wenn ein Individuum der Population eine Fitness von 0 besitzt.

Ändert sich die Fitness über einige Generationen hinweg nicht oder wird diese sogar schlechter, dann muss eventuell die Anzahl der Farben erhöht werden, um schneller ein Ergebnis zu bekommen.

1.3 Selektion

In der Selektion werden die Traumdreiecke anhand ihrer Fitness für die Rekombination ausgewählt:

Es werden nur die zwei Traumdreiecke zur Rekombination ausgewählt, die die höchste Fitness besitzen.

1.4 Rekombination

Die ausgewählten Traumdreiecke erzeugen die nächste Generation. Die Traumdreiecke der neuen Generation erhalten Murmeln vom besten und zweitbesten Traumdreieck der alten Generation. Wie viele Murmeln sie jeweils vom Besten und Zweitbesten bekommen wird dabei zufällig gewählt:

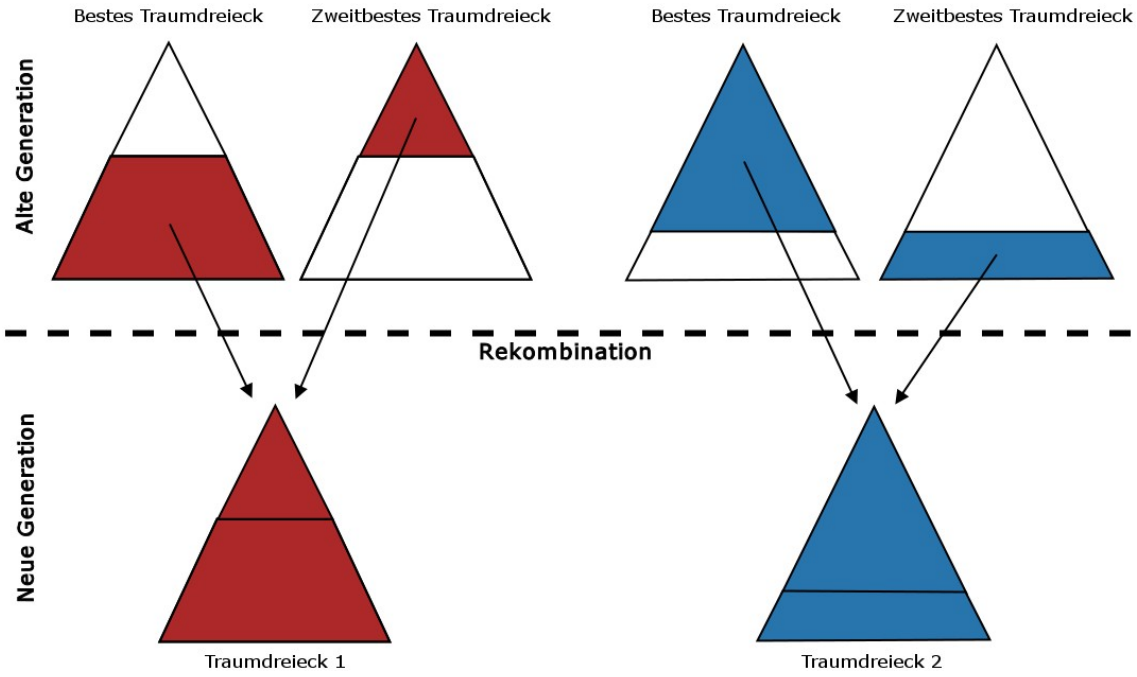


Abbildung 1: Rekombination der besten Traumdreiecke der alten Generation

Diese Art der Rekombination ist dem Crossing Over von Chromosomen nachempfunden.

Nach der Rekombination wird die alte Generation verworfen.

1.5 Gezielte Mutation

Nach der Rekombination verändern sich die Traumdreiecke aus der neuen Generation. Durch eine gezielte Mutation sollen ihre Fehler korrigiert werden, damit sie dadurch eventuell besser an die Umwelt angepasst sind: Eine zufällige Murmel in den gleichseitigen Dreiecken von Murmeln derselben Farbe bekommt eine neue zufällige Farbe:

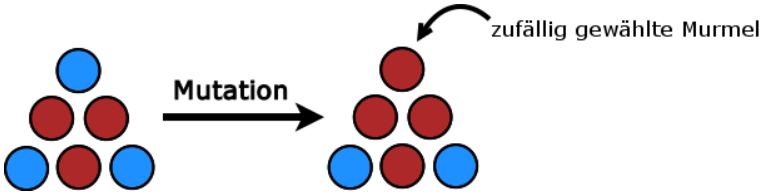


Abbildung 2: gezielte Mutation

Es kann vorkommen, dass bei der Mutation wieder ein gleichseitiges Dreieck mit Murmeln derselben Farbe entsteht oder dass der Farbwert der Murmel sich nicht verändert. Trotzdem werden

die gleichseitigen Dreiecke früher oder später durch den Zufall eliminiert.

Nachdem die Murmeln der Traumdreiecke verändert wurden, fängt alles wieder bei der Bewertung der Traumdreiecke der neuen Generation an (siehe 1. 2). Dies wiederholt sich, bis es ein Traumdreieck gibt, welches eine Fitness von 0 besitzt (Abbruchkriterium).

2 Programm-Dokumentation

Die Lösungsidee wurde von mir in der Programmiersprache C umgesetzt. Mein Wissen über B**ash/Shell-Programmierung** und **make** kam auch zum Einsatz.

Da man evolutionäre Algorithmen parallelisieren kann, habe ich jeden einzelnen Schritt der Evolution in einem eigenen Programm ausgelagert. Da die Mutation und das Bewerten der Traumdreiecke immer hintereinander geschieht, habe ich diese beiden Schritte in einem Programm zusammengefasst. Die einzelnen Programme sind: **initialisieren**, **selektion**, **mutundfit** und **rekombination**. Die Programme **mutundfit** und **rekombination** verarbeiten jeweils nur ein Traumdreieck und können deshalb parallel aufgerufen werden. Die Programme „kommunizieren“ über Dateien.

2.1 generation.mk und Shell-Skript

Das parallele Aufrufen der Programme wird durch das Makefile **generation.mk** realisiert. Dieses Makefile wird wiederum durch das Shell-Skript **loesen.sh** gesteuert, welches vom Benutzer aufgerufen wird.

Das Shell-Skript **loesen.sh** ist für das Finden einer Färbung zuständig und kann entweder mit einem oder mit zwei Parametern aufgerufen werden. Über den ersten Parameter legt man die Anzahl der Ebenen der Traumdreiecke fest. Mit dem zweiten Parameter kann man zusätzlich noch die Größe der Population festlegen.

2.2 traumdreieck-Struktur

Ein Traumdreieck wird im Programm als eine **traumdreieck**-Struktur dargestellt. Diese Struktur besteht aus drei **integer**-Variablen und einem zweidimensionalen Array. In den **integer**-Variablen wird die Anzahl der Ebenen und Anzahl der Farben des Traumdreiecks gespeichert. In dem zweidimensionalen Array werden die Farbwerte der Murmeln gespeichert. Das zweidimensionale Array wird in einer Dreiecksform „aufgebaut“. Die Ebene mit dem Index 0 hat demzufolge eine Murmel, die Ebene mit dem Index 1 zwei Murmeln, usw.

2.3 Initialisierung

Das Programm **initialisieren** bekommt die Anzahl der Ebenen der Traumdreiecke und die Größe der Population als Parameter übergeben und erstellt dann genauso viele Dateien, wie es Traumdreiecke pro Generation geben soll.

Das Programm erzeugt zunächst so viele **traumdreieck**-Strukturen, wie es der Parameter angibt. Dann wird das Murmel-Array mit Hilfe der Funktion **random_traumdreieck** mit zufälligen Farbwerten gefüllt:

$$Farbe_{Murmel} = \text{zufällige Zahl aus } [1,\ \ Anzahl\,benötigter\ Farben]$$

Danach werden die einzelnen Traumdreiecke als Datei abgespeichert.

Beträgt die Populationsgröße beispielsweise drei, so werden vom Programm **initialisieren** die Dateien **1.td**, **2.td** und **3.td** erzeugt. Das Suffix **td** der Dateien ist eine Abkürzung für „Traumdreieck“.

2.4 Gezielte Mutation und Bestimmen der Fitness

Ein bisschen abweichend vom eigentlichen evolutionären Algorithmus kann man auch gleich mit der Mutation der Traumdreiecke beginnen. Nach der Mutation werden dann die Traumdreiecke bewertet, deshalb sind diese beiden Schritte in dem Programm **`mutundfit`** zusammengefasst.

Das Programm **`mutundfit`** bekommt den Dateinamen eines Traumdreiecks als Parameter übergeben. Das Traumdreieck wird dann aus der Datei in eine **`traumdreieck`**-Struktur eingelesen und mit Hilfe der Funktion **`gezielte_mutation`** mutiert.

Die Funktion **`gezielte_mutation`** durchsucht die Murmeln eines Traumdreiecks nach gleichseitigen Dreiecken von Murmeln derselben Farbe. Sobald solch ein gleichseitiges Dreieck gefunden wurde, wird zufällig eine der drei Murmeln gewählt und bekommt einen zufälligen Farbwert. Dabei wird kein Farbwert gewählt, der die Anzahl der benutzten Farben erhöhen würde.

Danach wird das mutierte Traumdreieck an die Funktion **`bestimme_fitness`** weitergegeben. Diese Funktion überprüft jede Murmel, ob diese zu einem gleichseitigen Dreieck gehört, wo alle Murmeln dieselbe Farbe haben. Wird solch ein gleichseitiges Dreieck gefunden, so wird die Fitness um eins verringert (begonnen wird bei einer Fitness von 0). Der Fitnesswert wird dann in eine Datei mit dem Suffix **`fitness`** geschrieben. Der Basisname der Datei hängt dabei von den übergebenen Dateinamen des Traumdreiecks ab.

Wird das Programm **`mutundfit`** beispielsweise mit dem Dateinamen **`1.td`** des Traumdreiecks aufgerufen, so wird die Datei **`1.fitness`** erstellt, die die Fitness des Traumdreiecks enthält.

Das mutierte Traumdreieck wird in die selbe Datei abgespeichert, aus der das Traumdreieck eingelesen wurde. Eine **`td`**-Datei enthält also nach dem Aufruf des Programms **`mutundfit`** das mutierte Traumdreieck.

2.5 Selektion

Erst nachdem alle Traumdreiecke mutiert und bewertet wurden, kann das Programm **`selektion`** aufgerufen werden. Das Programm **`selektion`** wählt die zwei besten Traumdreiecke, also diejenigen mit der größten Fitness, für die Rekombination aus.

Dem Programm **`selektion`** wird die Populationsgröße als Parameter übergeben. Dann öffnet das Programm alle **`fitness`**-Dateien und bestimmt die beiden besten Traumdreiecke. Die **`td`**-Dateien der beiden besten Traumdreiecke werden kopiert und unter den Dateinamen **`1.sel`** und **`2.sel`** abgespeichert.

2.6 Rekombination

Das Programm **`rekombination`** ist für die Rekombination der ausgewählten Traumdreiecke zuständig. Durch die mehrmalige Rekombination der beiden besten Traumdreiecke entstehen die Traumdreiecke der nächsten Generation.

Das Programm **`rekombination`** bekommt den Dateinamen des Traumdreiecks, welches rekombiniert werden soll, als Parameter übergeben.

Zunächst werden die ausgewählten Traumdreiecke aus den Dateien **`1.sel`** und **`2.sel`** jeweils in eine **`traumdreieck`**-Struktur eingelesen. Danach wird die Funktion **`rekombinieren`** mehrmals ausgeführt, die als Parameter die zwei besten Traumdreiecke der alten Generation übergeben bekommt. Die Funktion rekombiniert die Traumdreiecke und gibt dann ein Traumdreieck der neuen Generation zurück.

Bei jedem Aufruf der Funktion **`rekombinieren`** wird zunächst ein zufälliger Wert erzeugt, der für den Index steht, an dem die Traumdreiecke geteilt werden:

$$s = \text{zufällige Zahl aus } [0,\ n-1]$$

Ein Traumdreieck der neuen Generation bekommt dann die oberen Ebenen des einen Traumdreiecks und die restlichen Ebenen von dem anderen Traumdreieck:

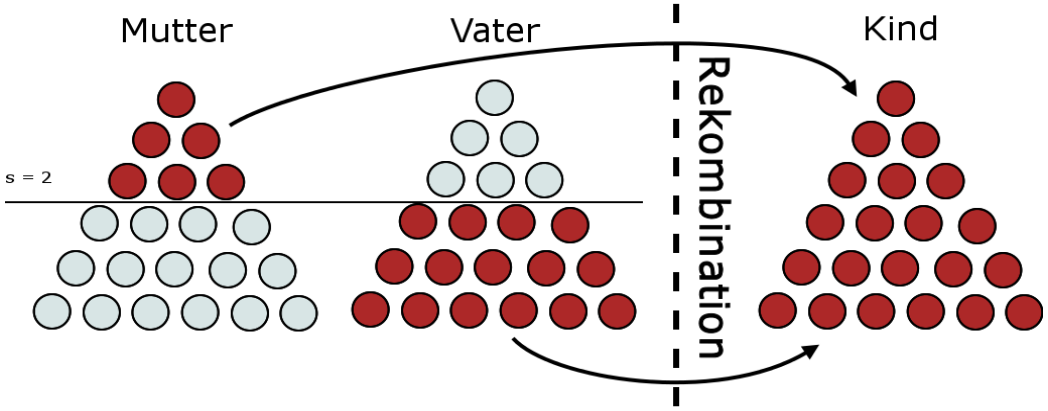


Abbildung 3: Beispiel zur Rekombination der Traumdreiecke für n = 6 und s = 2

(In dieser Abbildung sind die Farben nur zur Veranschaulichung und stehen nicht für die Farben der Murmeln.)

Das rekombinierte Traumdreieck der neuen Generation wird dann als Datei abgespeichert.

Wird das Programm **`rekombination`** beispielsweise mit dem Parameter **`1.td`** aufgerufen, so enthält die Datei **`1.td`** danach ein Traumdreieck der neuen Generation.

Nun fängt alles wieder bei der Mutation und Fitnessbestimmung an (siehe 2. 2).

2.7 Terminierung

Sobald eines der Traumdreiecke eine Fitness von 0 besitzt, wird dies von dem Programm **`selektion`** bemerkt. Das Traumdreieck wird dann unter der Datei **`td.loesung`** abgespeichert und das Shell-Skript gibt die Lösung dann auf dem Bildschirm aus.

2.8 Populationsgröße

Die Anzahl der Traumdreiecke pro Generation kann durch einen zweiten Parameter beim Shell-Skript angegeben werden. Dieser Parameter würde den Standardwert von 10 überschreiben.

Eine große Population lässt die Fitness in größeren Schritten gegen 0 konvergieren, weil es mehr „Vielfalt“ gibt.

2.9 Laufzeit

Ich habe die Parallelisierung der Programme abhängig von der Anzahl der Kerne der CPU gemacht. Wird das Shell-Skript also auf einem PC mit einem Dual-Core-Prozessor ausgeführt, so werden maximal zwei Programme gleichzeitig ausgeführt. Auf einem Quad-Core PC benötigte mein Algorithmus 30 Minuten um eine Färbung für n = 1000 zu erhalten. Auf einem 48-Core Rechner mit n = 1000 und einer Populationsgröße von 48 (für jeden Kern ein Traumdreieck) benötigt der Algorithmus nur noch 5 Minuten (Programm wurde in **`/dev/shm`** ausgeführt).

3 Programm-Ablaufprotokoll

Die einzelnen Programme werden durch durch das Makefile **`generation.mk`** gesteuert. Das Shell-Skript **`loesen.sh`** kümmert sich um die richtige Benutzung des **`make`**-Kommandos für das Makefile **`generation.mk`**. Nur das Shell-Skript wird vom Benutzer aufgerufen.

Es reicht das Shell-Skript mit der Anzahl der Ebenen als Parameter aufzurufen. Möchte man die Populationsgröße ändern, kann man diese als zweiten Parameter angeben.

Das Shell-Skript startet dann den evolutionären Algorithmus. Zwischendurch wird die Fitness des besten Traumdreiecks jeder Generation angezeigt. Sobald eine Fitness von 0 erreicht ist, wird die Färbung des Traumdreiecks auf dem Bildschirm ausgegeben.

Aufruf des Skripts mit n = 3:

```
$ ./loesen.sh 3
Traumdreieck fuer n = 3
1
1 2
2 2 1
Anzahl benoetigter Farben: 2
```

Erst bei n = 5 benötigt man drei Farben:

```
$ ./loesen.sh 5
Traumdreieck fuer n = 5
1
2 1
1 1 3
2 2 1 1
1 1 2 2 2
Anzahl benoetigter Farben: 3
```

Bei n = 16 benötigt man 4 Farben:

```
$ ./loesen.sh 16
Traumdreieck fuer n = 16:
4
2 3
3 3 1
3 4 2 2
2 1 1 3 3
1 3 1 3 2 4
2 4 1 4 2 1 2
3 2 4 2 2 1 4 3
1 4 1 2 3 2 3 4 4
1 3 3 1 4 2 4 1 2 2
4 2 3 1 4 3 4 2 1 1 1
4 1 2 2 3 4 1 1 3 3 4 1
3 1 4 2 1 4 3 1 4 4 2 3 3
1 2 3 1 4 2 4 3 3 3 1 4 3 4
4 3 4 1 2 4 2 2 3 2 2 4 3 1 1
3 2 3 4 1 4 3 4 1 1 3 2 1 2 2 1
Anzahl benoetigter Farben: 4
```

Bei n = 100 benötigt der Algorithmus eine halbe Minute, um die optimale Färbung zu finden.

Bei einem Aufruf des Skripts mit n = 200 benötigt der Algorithmus eine Minute um die optimale Färbung zu finden.

Bei n = 500 sind es 2 Minuten. Und für n = 1000 die bereits erwähnten 30 Minuten. Dank Quad-Core ist die Laufzeit des Algorithmus also noch akzeptabel.