8. Programmieraufgabe

Programmierparadigmen

LVA-Nr. 194.023
2023/2024 W
TU Wien

Themen:

Funktionale Programmierung

Ausgabe:

04. 12. 2023

Abgabe (Deadline):

18. 12. 2023, 14:00 Uhr

Abgabeverzeichnis:

`Aufgabe8`

Programmaufruf:

`java Test`

Grundlage:

Skriptum, Schwerpunkt auf Abschnitt 5.1 und 5.2

Kontext

Ameisenalgorithmen (Ant Colony Optimization, ACO) sind Metaheuristiken[1], die das modellhafte Verhalten von Ameisen bei der Futtersuche zum Vorbild haben. Die Algorithmen sind auf die Lösung von Optimierungsproblemen zugeschnitten und weichen erheblich vom Verhalten natürlicher Ameisen ab. Wir wollen einen Ameisenalgorithmus einsetzen, um Traveling-Salesman-Probleme (TSP)[2] zu lösen. Ein Vergleich von Ameisenalgorithmen für TSP ist z. B. unter

https://staff.washington.edu/paymana/swarm/stutzle99-eaecs.pdf

zu finden, wobei wir uns auf ACS (Ant Colony System, Abschnitt 1.3.3, eine Variante von ACO) beziehen, da der Algorithmus effizient und einfach ist. Ein TSP lässt sich folgendermaßen beschreiben: Gegeben ist ein vollständiger gewichteter Graph $G = (N, A)$, wobei N eine Menge von Knoten (Städten) und A eine Menge von Kanten (engl. arcs) ist, die je zwei Städte verbinden. Für jede Kante $(i, j) \in A$ bezeichnet d_{ij} (distance) die Länge des Wegs (im Sinne einer kürzesten oder schnellsten Straßenverbindung, nicht zu verwechseln mit einem Pfad im Graphen) zwischen den Städten $i, j \in N$. Wir gehen von symmetrischen Weglängen aus, sodass $d_{ij} = d_{ji}$. ACO-Algorithmen beruhen auf Pheromonstärken (Stärken der Duftspuren) auf den Kanten: $\tau_{ij}(t)$ steht für einen numerischen Wert (`double`), der die Pheromonstärke der Kante zwischen i und j in der Iteration t (Zähler der Iterationen des Algorithmus' vom Typ `int`) darstellt. Aufgrund der Symmetrie gilt $\tau_{ij}(t) = \tau_{ji}(t)$.

Zu Beginn des Ablaufs wird eine vorgegebene Anzahl m von Ameisen zufällig auf die Städte verteilt. In jeder Iteration t besucht jede Ameise jede Stadt. Dabei bewegt sich eine Ameise, die derzeit in der Stadt i ist, zufällig (aber mit einer Wahrscheinlichkeit abhängig von $\tau_{ij}(t)$ und d_{ij}) in eine von dieser Ameise noch nicht besuchte Stadt j. Das wiederholt die Ameise, bis sie alle Städte besucht hat. Jede Ameise führt eine eigene Liste der in dieser Iteration schon besuchten Städte mit, die nicht noch einmal besucht werden dürfen. Nach jeder Bewegung von einer Stadt i zur Stadt j wird $\tau_{ij}(t)$ sowie $\tau_{ji}(t)$ etwas abgeändert, damit andere Ameisen dazu angehalten werden, möglichst unterschiedliche Rundreisen zu erkunden. Die Ameisen bewegen sich überlappend: Zuerst macht die erste Ameise einen Schritt, dann die zweite und so weiter bis zur $m$ten,

[1] Während Algorithmen zur näherungsweisen Lösung bestimmter Optimierungsprobleme *Heuristiken* genannt werden, sind *Metaheuristiken* Algorithmen, die auf viele unterschiedliche Optimierungsprobleme anwendbar sind. Die einzelnen Schritte in Metaheuristiken müssen aber auf bestimmte Optimierungsprobleme zugeschnitten sein.

[2] Das *Traveling-Salesman-Problem* ist eine häufig verwendete Optimierungsaufgabe, bei der vorgegebene Städte, die jeweils paarweise über Wege miteinander verbunden sind, in einer solchen Reihenfolge besucht werden sollen, dass die gesamte dabei zurückgelegte Weglänge möglichst kurz ist. Dieses Problem ist im Allgemeinen NP-schwer, das heißt, um eine optimale Lösung zu finden, müssen alle möglichen Reihenfolgen miteinander verglichen werden, was durch die kombinatorische Explosion schon bei einer relativ kleinen Anzahl an Städten zu einem gewaltigen Aufwand führt.

erst danach macht die erste Ameise den nächsten Schritt, dann die zweite und so weiter, bis jede Ameise jede Stadt besucht hat. Danach werden die Rundreisen analysiert und neue $\tau_{ij}(t+1)$ berechnet, die als Grundlage für die nächste Iteration $t+1$ dienen. In der nächsten Iteration besuchen wieder alle Ameisen alle Städte. Im Idealfall finden die Ameisen in jeder Iteration bessere Rundreisen, bis eine annähernd optimale Lösung des TSP gefunden wurde. Wir gehen von folgendem vereinfachten Algorithmus aus:

procedure ACS für TSP
  Parameter setzen, Pheromonstärken initialisieren
  while (gegebene Anzahl an Iterationen nicht erreicht) **do**
    jede Ameise erzeugt Lösung durch Besuch aller Städte
    Pheromonstärken an bisher beste Lösung anpassen
  end while
end ACS für TSP

Eine Reihe von Parametern fließt in Details des Algorithmus' ein:

- m (häufig 25) ist die Anzahl der Ameisen.
- q_0 (häufig 0.9) ist die vorgegebene Wahrscheinlichkeit dafür, dass der beste Weg gewählt wird, nicht irgendeiner.
- α (häufig 1) gibt den Einfluss der Pheromonstärken auf die Auswahl der nächsten zu besuchenden Stadt an.
- β (häufig 2) gibt an, wie stark Entfernungen der Städte in die Auswahl der nächsten zu besuchenden Stadt einfließen.
- ρ ($0 < \rho \leq 1$, häufig zwischen 0.1 und 0.2) gibt an, um welchen Faktor sich Pheromonstärken abschwächen.

Eine wichtige Entscheidung ist, welche Stadt eine Ameise, die gerade in der Stadt i ist, als nächste besucht. Mit Wahrscheinlichkeit q_0 (vorgegeben) wird die Stadt j gewält, für die $\tau_{ij}(t) \cdot (1/d_{ij})$ maximal ist. Mit Wahrscheinlichkeit $1 - q_0$ wählt die Ameise k die nächste Stadt j zufällig nach folgenden Wahrscheinlichkeiten $p_{ij}^k(t)$:

$$p_{ij}^k(t) = \frac{(\tau_{ij}(t))^\alpha \cdot (1/d_{ij})^\beta}{\sum_{l \in N_i^k} (\tau_{il}(t))^\alpha \cdot (1/d_{il})^\beta} \quad (j \in N_i^k)$$

wobei N_i^k die Menge der Städte ist, die k in dieser Iteration noch nicht besucht hat.

Direkt nachdem eine Ameise den Weg von i nach j gewählt hat, wird $\tau_{ij}(t)$ und $\tau_{ji}(t)$ folgendermaßen angepasst (lokaler Pheromonupdate):

$$\tau_{ij}(t) = \tau_{ji}(t) = (1 - \rho) \cdot \tau_{ij}(t) + \rho \cdot \tau_0$$

Für τ_0 kann eine beliebige sehr kleine Konstante größer 0 verwendet werden. Häufig wird für τ_0 der Wert $1/(n \cdot L^{nn})$ verwendet, wobei n die Anzahl der Städte und L^{nn} die Gesamtlänge einer auf einfache Weise berechneten Rundreise ist. Beispielsweise kann L^{nn} berechnet werden, indem (in einer

beliebigen Stadt beginnend) als nächste Stadt jeweils die nächstgelegene noch nicht besuchte Stadt gewählt wird.

Am Ende jeder Iteration werden Pheromonstärken (nochmals) angepasst, wobei nur die bisher beste (das ist die kürzeste) Lösung einen Beitrag zur Erhöhung der Pheromonstärke leistet:

$$\tau_{ij}(t+1) = \tau_{ji}(t+1) = (1-\rho) \cdot \tau_{ij}(t) + \rho \cdot \Delta\tau_{ij}(t)$$

Dabei ist $\Delta\tau_{ij}(t) = (1/L^{gb})$ wenn die Kante (i, j) oder (j, i) in der bisher besten Lösung vorkommt, sonst ist $\Delta\tau_{ij}(t) = 0$; der Ausdruck L^{gb} steht für die Gesamtlänge der bisher besten Rundreise (gb für global-best). Falls in der Iteration t keine so gute Lösung wie in einer früheren Iteration gefunden wurde, wird die in einer früheren Iteration gefundene beste Lösung zur Anpassung der Pheromonstärken verwendet. Zur Initialisierung der Pheromonstärken vor der ersten Iteration sollte z. B. L^{nn} (siehe oben) als „bisher beste Lösung“ verwendet werden.

bedenken, dass jede Lösung eine Rundreise ist und den Weg zurück zur ersten Stadt einschließt

Es ist nicht außergewöhnlich, dass Algorithmen für eine gute Lösung eines aufwändigen Optimierungsproblems über Stunden oder sogar Tage laufen. Wir wollen uns für diese Aufgabe mit kleineren Problemen und geringeren Anforderungen an die Qualität der Lösungen zufrieden geben, sodass ein Programmlauf nach etwa einer Minute terminiert. Die Anzahl der Iterationen und die Anzahl der Städte (vielleicht 30) ist so zu wählen, dass das einigermaßen gut hinkommt. Das Ziel besteht darin, in dieser kurzen Zeit eine möglichst gute Lösung zu finden.

Welche Aufgabe zu lösen ist

Java-8-Streams und Funktionen höherer Ordnung

Das oben beschriebene Optimierungsproblem ist algorithmisch aufwändig, sodass ein funktionaler Ansatz dafür gut geeignet ist. Verwenden Sie zur Lösung in Java einen funktionalen Ansatz, der sowohl Java-8-Streams, als auch selbst geschriebene funktionale Formen verwendet. Wesentliche Programmteile (Funktionen) sind so auszulegen, dass der Graph $G = (N, A)$, die Entfernungen d_{ij} ebenso wie die im Algorithmus verwendeten Parameter und die Anzahl der Iterationen beim Aufruf einer Funktion als Argumente frei wählbar sind.

Algorithmus als Funktion bereitstellen

Für eine Ameise sollte es relativ einfach sein, eine Liste bereits besuchter Knoten in einem funktionalen Ansatz mitzuführen, da stets nur neue Einträge in diese Liste eingefügt, aber (außer am Ende jeder Iteration) nie etwas aus der Liste entfernt wird. Es sollte auch nicht schwer sein, stets die bisher beste Lösung mitzuführen. Durch den expliziten Parameter t kann $\tau_{ij}(t+1)$ leicht aus $\tau_{ij}(t)$ berechnet werden, ohne dabei $\tau_{ij}(t)$ ändern zu müssen. Aber ein Aspekt bereitet Schwierigkeiten: $\tau_{ij}(t)$ muss auch innerhalb einer Iteration immer wieder neu berechnet werden. Dafür muss eine Lösung gefunden werden, die ohne destruktive Änderungen von Variablenwerten auskommt, oder dieser Teil der Aufgabe ist in einem prozeduralen Programmierstil zu lösen, wobei die Grenzen zwischen den unterschiedlichen Paradigmen sehr sorgfältig gestaltet werden müssen.

mit Seiteneffekten umgehen

Aufgrund der Symmetrie muss stets $d_{ij} = d_{ji}$ und $\tau_{ij}(t) = \tau_{ji}(t)$ gelten. Es ist nicht nötig, sowohl d_{ij} (oder $\tau_{ij}(t)$) als auch d_{ji} (oder $\tau_{ji}(t)$) im Speicher zu halten oder sogar mehrfach zu berechnen, je einer der Werte reicht. Die Symmetrie soll im Programm ausgenutzt werden.

Symmetrie nutzen

Auch wenn Parameter frei wählbar sind, muss das Programm doch mit konkreten Werten getestet werden. Vor allem muss ein geeigneter Graph $G = (N, A)$ mit den Entfernungen d_{ij} gefunden werden. Im Internet sind standardmäßig verwendete Instanzen von TSPs zu finden, aber vermutlich geht es leichter, die Daten des zu lösenden TSP selbst zu generieren: In einem ausreichend großen zweidimensionalen Raum wird zufallsverteilt die nötige Anzahl an Städten platziert. Wenn die Städte i und j dabei an den Koordinaten (x_i, y_i) und (x_j, y_j) zu liegen kommen, kann vereinfacht $d_{ij} = \text{abs}(x_i - x_j) + \text{abs}(y_i - y_j)$ gewählt werden, eventuell noch um einen zufälligen (kleinen) Betrag modifiziert. Die Anzahl der Städte und die Anzahl der Iterationen sind so zu wählen, dass sich ein Programmlauf dafür in etwa einer Minute ausgeht.

TSP generieren

Wie üblich soll das Programm durch einen Aufruf von `java Test` vom Abgabeverzeichnis `Aufgab8` aus ausführbar sein. Am Ende eines Programmlaufs sollen folgende Daten ausgegeben werden:

auszugebende Ergebnisse

- Die beste gefundene Lösung mit allen Städten und Entfernungen zwischen je zwei hintereinander besuchten Städten sowie die Gesamtlänge der Rundreise,
- für jede Iteration die Gesamtlänge der in dieser Iteration gefundenen besten Lösung,
- die im Testlauf verwendeten Parameter (ausgenommen die Städte und Entfernungen zwischen den Städten),
- (optional) von obigen Daten deutlich sichtbar getrennt die Ergebnisse von Tests, die durchgeführt werden, um das korrekte Funktionieren verschiedener Teile des Algorithmus' zu prüfen.

Wie üblich soll die Datei `Test.java` als Kommentar eine kurze, aber verständliche Beschreibung der Aufteilung der Arbeiten auf die einzelnen Gruppenmitglieder enthalten – wer hat was gemacht.

Aufgabenaufteilung beschreiben

Wie die Aufgabe zu lösen ist

Es gibt kaum Einschränkungen im Hinblick darauf, wie diese Aufgabe zu lösen ist. Vorausgesetzt wird jedoch, dass eine funktionale Denkweise zum Einsatz kommt, irgendwo Java-8-Streams vorkommen, aber irgendwo auch selbst geschriebene funktionale Formen (also Funktionen, die Lambdas oder andere Objekte, die vorwiegend als Funktionen verwendet werden, als Parameter haben) vorkommen. Die formale Denkweise ist unvermeidlich. Ob Sie vorwiegend mit Java-8-Streams oder eigenen funktionalen Formen arbeiten möchten, ist Ihnen überlassen, solange beides irgendwo vorkommt.

funktionale Denkweise soll klar erkennbar sein

Wichtig ist, dass durch die Lösung ein Verständnis der Ziele hinter der funktionalen Programmierung zum Ausdruck kommt. Es kommt sowohl auf Programmiereffizienz als auch eine effiziente Programmausführung an. Idealerweise werden vorgefertigte Programmteile aus Standardbibliotheken eingesetzt.

Es ist ausdrücklich nicht verboten, in kleinen Teilen des Programms auch Seiteneffekte (in Form von destruktiven Zuweisungen) zuzulassen.

klare, sichere Schnittstelle zwischen funktionalen und nichtfunktionalen Teilen

Allerdings wird dabei die Ebene der funktionalen Programmierung verlassen und eine imperative (vermutlich prozedulare) Ebene betreten. Die Schnittstelle zwischen diesen Ebenen muss so klar wie möglich gestaltet sein, etwa durch eine deutliche Abgenzung wie eine Auslagerung in eine andere Klasse sowie deutliche Hinweise in Form von Kommentaren. Bedenken Sie auch, welche Strukturierung einer Zusammenarbeit zwischen prozeduralen und funktionalen Teilen in der Regel unproblematisch ist und versuchen Sie, eine solche Strukturierung herzustellen.

Kommentare in der funktionalen Programmierung unterscheiden sich deutlich von solchen in der objektorientierten Programmierung. Darin spiegelt sich eine andere Form der Abstraktion wider. Versuchen Sie bitte bewusst, Kommentare so einzusetzen, wie das in der funktionalen Programmierung üblich ist. Konzepte wie Design-by-Contract sind in diesem Zusammenhang nicht von Bedeutung.

Kommentare entsprechend eines funktionalen Stils

Was im Hinblick auf die Beurteilung wichtig ist

Die insgesamt 100 für diese Aufgabe erreichbaren Punkte sind folgendermaßen auf die zu erreichenden Ziele aufgeteilt:

- funktionale Prinzipien eingehalten 40 Punkte
- Algorithmus richtig und effizient implementiert 40 Punkte
- für funktionale Programmierung typische Kommentare 10 Punkte
- Lösung vollständig (entsprechend Aufgabenstellung) 10 Punkte

Schwerpunkte berücksichtigen

Der erste Punkt bezieht sich darauf, ob eine funktionale Denkweise erkennbar ist und sowohl Java-8-Streams als auch selbst geschriebene funktionale Formen in deutlich erkennbarem Ausmaß und korrekt zum Einsatz kommen. Dieser Punkt bezieht sich aber auch darauf, ob bei gleichzeitigem Einsatz eines anderen Paradigmas (in vergleichsweise kleinem Ausmaß) die Schnittstellen zwischen den Paradigmen auf eine sichere Weise strukturiert und deutlich beschrieben sind.

Anders als in früheren Aufgaben spielt die korrekte und effiziente Umsetzung des vorgegebenen Algorithmus' eine sehr große Rolle. Eine fehlerhafte oder ineffiziente Umsetzungen kann (auch bei vergleichsweise unbedeutenden Ursachen) zu bedeutendem Punkteverlust führen.

Kommentare sind extra angeführt, um nochmals darauf hinzuweisen, dass auch hierbei ein Umdenken nötig ist. Es gibt Punkteabzüge, wenn Kommentare einem objektorientierten Stil folgen oder fehlen.

Warum die Aufgabe diese Form hat

Im Unterschied zu allen bisherigen Aufgaben geht es in dieser Aufgabe darum, einen vorgegebenen, nicht-trivialen Algorithmus in einem Programm darzustellen und zur Ausführung zu bringen. Dabei sollen Techniken der funktionalen Programmierung zur Anwendung kommen. Nachdem in den letzten Aufgaben vorwiegend Techniken der objektorientierten Programmierung im Mittelpunkt standen, soll durch diese Aufgabe intuitiv klar werden, wie stark sich die funktionale Denkweise und Zielsetzung

von der objektorientierten unterscheidet. Vorgegebene algorithmische Beschreibungen müssen verstanden und möglichst effizient (sowohl im Sinne der Programmiereffizienz als auch der Laufzeiteffizienz) umgesetzt werden, wobei aber die langfistige Wartbarkeit des Programms vernachlässigt werden kann. Die Aufgabe enthält Details, die im Zusammenhang mit referenzieller Transparenz (Seiteneffektfreiheit) Schwierigkeiten verursachen. Es ist Ihnen freigestellt, wie Sie diese lösen. Sie können dafür sorgen, dass referenzielle Transparenz weitgehend erhalten bleibt (bis auf die Ausgabe der Endergebnisse), oder Sie können einen (kleinen) Teil der Aufgabe in einem nicht-funktionalen Stil lösen, müssen in diesem Fall aber für eine sinnvolle Zusammenarbeit zwischen unterschiedlichen Programmierparadigmen sorgen. Vor ähnliche Entscheidungen werden Sie in den meisten Aufgaben mit komplexeren Algorithmen gestellt werden.