Knapp Coding Contest 2018

Ziel

Es gilt die Anzahl der *Schlecht*-Punkte zu minimieren.

- möglich viele Order erfüllen
  - nicht erfüllte Order geben viele Punkte
- möglichst kurze Wege
  - das Herumfahren kostet auch.

Details

Anleitung lesen!

Es gibt ein Lager mit Containern die jeweils ein Produkt enthalten.

Detail: das Lager hat mehrere Achsen (Weg zwischen den Achsen kostet auch!). Im Regal (links und rechts) stehen Container.

Es gibt eine Orderliste (Produkt, Anzahl). Diese soll abgearbeitet werden. Dazu muss man das Produkt im Lager suchen, den Container herbringen und ausladen ("pick"). Den nicht mehr benötigten Container muss man auch wieder loswerden.

Wichtig: um Details (Wieviel Punkte kostet ein Weg? Was ist in einem Container? etc.) braucht man sich nicht kümmern – man braucht *nur* die passenden Methoden aufrufen.

Info: HTL Wertung

- erster Platz: 1283315 Punkte, wäre bei TU Platz 6
- unser Bester: 1533134 Punkte, wäre bei TU Platz 8
- 10. Platz: 1718551 Punkte
- 30. Platz: 2545550 Punkte

TU 1. Platz: 960553

Anleitung

Anmerkung: Ansatz HOR, nicht optimiert, aber trotzdem recht effizient ;-)

Tipp:

- möglichst viel mit Streams
- oder Lambdas
- Fallstricke
  - viele *Dinge* sind `null` – immer vorher prüfen
    - der Container wenn das Shuttle leer ist
    - das Produkt wenn der Container leer ist
  - Achtung: man muss die Orderliste in der richtigen Reihenfolge erfüllen, man darf keine Order auslassen.

Strategie:

- immer nur eine Order betrachten – keine Optimierung über alle Order

Vorbereitung

- Sandbox auspacken, zum Laufen bringen.
  - im Zip ist ein `src` Ordner, parallel dazu ein `data`.

        * `com.knapp.codingcontest.kcc2018.solution` gibt es ein `Main.main()`.
    - TODO: Name + Organisation einfügen
    - Wieviel Punkte haben wir? Weniger ist Mehr!
    - `git`!
- Anleitung lesen!
- *Einsprungpunkt* finden
    - `Solution.java` – `runWarehouseOperations()`

Aufgabe:

Die Liste der Order abarbeiten. Dazu jeweils

- mit dem *Shuttle* an die richtige Stellen fahren
- Container aufladen
- Container zur *Workstation* bringen
- umladen (`pickOrder`)
- nicht mehr benötigte Container irgendwo *abstellen*

Klingt einfach – ist es auch ;-)

Schleife

in `Solution.runWarehouseOperations`

```
for (Order o : orders) {
    handleOrder(o);
}
```

Woher bekommt man die Orders? siehe `Solution.apis()`: da gibt es viele *Muster*.

Tipp:

- Kann man als Attribute speichern – im Konstruktor initialisieren.
- oder immer neu bestimmen – ist auch nicht viel aufwändiger

`handleOrder(o)`

Wir brauchen:

- ProductCode – das brauchen wir
- RemainingQuantity – so viele sollen es sein

Und vielleicht eine *Debug*-Ausgabe (`soutv`).

`git`!

Hilfsmethode: `getAllLocations(prodCode, allContainers)`

Alle Container `filtern`:

- location != null (`getLocation()`)
- location reachable
    - kompliziert: container.getLocation().isReachable(container)
- ProductCode != null (`getProductCode()`)
- ProductCode == prodCode (`equals` !)

liefert eine Liste der Locations (`getLocation`).

Los gehts

- Liste aller mögliche Locations erstellen (`getAllLocations`)
- Liste filtern

  - nur Container mit genügend Inhalt
  - ist etwas kompliziert: wir haben eine `Location`, brauchen die Menge:

    ```
    loc.getContainers().get(0).getQuantity()
    ```

- diese Liste sortiern
  - nach Entfernung (`calcMoveCost`)
  - `Comparator.comparingLong` und `warehouse.calcMoveCost()`
- erste Location in der Liste
  - statt sortieren und das erste: `min()`
  - shuttle zur Position fahren (`shuttle.moveToPosition`)
    - Position merken (lastPosition)!
    - shuttle beladen (shuttle.loadFrom())
  - shuttle zur workstation fahren
    - ausladen: `workStation.pickOrder`

sollte schon funktionieren – erste Order erledigt?

Bei der zweiten Oder geht es nicht mehr – wenn man die `Exceptions` in `runWarehouseOperations` bei in der Schleife über alle Order behandelt schaut es besser aus.

OOPS

Ergebnis ist jetzt schlechter

- Kosten für Bewegung
- Kosten für "shuttle nicht leer"

Lösung – nach der Schleife:

- falls das shuttle nicht leer ist (`getLoadedContainer()`):
  - zur lastPosition/einer freien Stelle fahren
  - shuttle entladen

Super – endlich weniger Punkte!

Tipp:

- das `try / catch` wieder auskommentieren: dann sieht man schneller wo es hackt.
- Alternative: man kann eine Breakpoint bei Exception definieren (Run/View Breakpoints)

Weiter

bei der nächsten Order ist das Shuttle am Anfang nicht leer. Bevor wir zur Position fahren müssen wir den Container los werden – vielleicht ist es schon das richtige Produkt:

- falls ein Container am Shuttle ist (`shuttle.getLoadedContainer`)
  - zur lastPosition fahren und Container dort abstellen

Wie weit kommt man jetzt?

Zuviel

Manchmal reicht ein Container nicht aus – es sind zuwenig Produkte darin: statt der ersten Position brauchen wir eine Schleife

Wenn die Liste der gefilterten Locations (Container mit genügend Inhalt) leer ist – nehmen wir halt die ganze Liste.

Und machen eine Schleife:

```
do {
   ...
 } while(order.getRemainingQuantity() > 0) // Alternative: ! hasRemainingItems()
```

Wie weit kommt man jetzt? Das kann sich doch schon sehen lassen...

Optimierung

Glück

Manchmal ist schon das richtige Produkt am Shuttle

- falls ein Container am Shuttle ist (das gibt es schon):
  - und im Container ein Produkt ist (`isEmpty()`)
    - und es das richtige Produkt ist
    - und es mehr als 0 Stück sind (`getQuantity() > 0`)
      - kann man die komplizierte Suche nach einem Container und das Holen auslassen!
      - `pickOrder()`
      - return falls die Order fertig ist

Super - wieder besserer Score!

Glück II

Man kann bei der Bestimmung der notwendige Anzahl auch die nächsten Container anschauen – wenn das selbe Produkt brauchen dann kann man gleich mehr holen.

Hilfsmethode: `getEmptyCloseTo(position)`

Erst brauchen wir alle möglichen Positionen: für alle Gänge (`aisles`) jeweils alle Locations

Das kann man mit `flatMap` lösen: macht aus den verschachtelten `streams` einen einzigen

```
Arrays.stream(warehouse.getAisles()).flatMap(a -> a.getLocations().stream())
```

Dann ist es einfach:

- filter: nicht die aktuelle position
- filter: Anzahl der Container < 2 – an jeder Stelle können 2 Container (hintereinander) stehen
  - getContainers().size()
- sortieren – kleinste Kosten (von position zur location)
- nur die erste Location brauchen wir (`min`)

liefert: Location

leeren Platz suchen Statt zur `lastPosition` fahren wir zu nächsten leeren Stelle. Dh, `lastPosition` wird immer durch den Aufruf von `getEmptyCloseTo` ersetzt.

Tipp: `workStation` ist eine Position

Und wieder ist das Ergebnis besser!

Noch besser (aber aufwendig): der Weg von der aktueller Position `frm` zur freien Stelle und von dort weiter zur gewünschten Position `to` sollte minimal sein: `getEmptyCloseTo(Position from, Position to)` – wichtig: man sollte sich damit nicht selbst im Weg stehen

Weiter geht es

- Wie weit kommt man?
- Warum ist die Order nicht erfüllbar? der Debugger hilft!
  - Breakpoint on Exception oder
  - Breakpoint mit Bedingung

- Was könnte man tun?

Hinweise:

- jede Location hat zwei Container – hintereinander. Nur der vordere ist `reachable`.
- Also Umschlichten dh. den vorderen Container weg
  - suche einen passende Container – ignoriere reachable
  - hinfahren, container aufnehmen
  - falls es das falsche Produkt ist (weil das Richtige hinten steht):
    - leere Stelle in der Nähe suchen
    - Container dort abstellen
    - zurückfahren und wieder aufnehmen

Jetzt sollten alle Order gehen – und wir wären unter den ersten 10 gewesen!

Optimierung – echte Kosten

Die Auswahl des Containers ist nicht ideal – man sollte die *richtigen* Kosten berücksichtigen:

- alle Container
- filter: mit Location
- mit einem Produkt
- mit dem richtigen ProductCode
- sortieren cost = calcMoveCost(...) falls nicht erreichbar:

  ```
  cost += doppelter Weg zu einem freien Platz
  ```

  falls im Container zu wenig Produkte sind:

  ```
  cost *= wieviele solche container braucht es, aufgerundet
  ```

- davon nur das erste
- die Location

Punkte?

Optimierung – getEmptyCloseTo

manchmal stellen wir einen (leeren) Container irgendwo hin, später stört er weil wir das Produkt *dahinter* brauchen – vielleicht sollte man leere Container wo anders hingeben bzw. solche Plätze mit Maluspunkten belegen damit sie eher nicht genommen werden.