Лабораторная работа #3

Дисциплина: "Функциональное программирование"

Дата: 2021/10

Выполнил: Федоров Сергей, P34113

Название: "Потоковая обработка данных, Замыкания"

Цель работы: Получить навыки работы с вводом/выводом, потоковой обработкой данных, замыканиями.

Вариант:

ЯП	Clojure
Алгоритм	Spline-interpolation

Требования:

1. должна быть реализована «аппроксимация» отрезками
2. настройки алгоритма аппроксимирования и выводимых данных должны задаваться через аргументы командной строки:
    - какой или какие алгоритмы использовать
    - частота дискретизации
    - и т.п.
3. входные данные должны задаваться в текстовом формате на подобии “.csv” (к примеру `x;y\n` или `x\ty\n` и т.п.) и поддаваться на стандартный ввод, входные данные должны быть отсортированы по возрастанию x
4. выходные данные должны подаваться на стандартный вывод
5. программа должна работать в потоковом режиме (пример - cat | grep 11)

Выполнение

Ссылка на репозиторий

Логика работы программы (и с программой)

Программа постоянно считает поток входящих строчек с stdin, и сразу же отвечает на них в stdout. Внутри работает все на async/chan (асинхронные каналы с буфферами и без).

Есть три сообщений которые программа обрабатывает: 1. `t,x,y` - добавление точки для выборки интерполятора 2. `d,x` - добавление точки для применения интерполятора 3. `predict` - прогнать интерполятор обученный на выборке по записанным данным

Так-же программа корректно обрабатывает сигнал к завершению. Закрывает все channel'ы и выходить с 0 `ExitCode`.

Программная реализация

Структура проекта:

```
.
├── project.clj
└── src
      └── functional_programming_itmo_2021
          └── lab_3
              ├── interpolation.clj
              └── io_streaming.clj
              └── main.clj
```

Настройка Clojure-окружения с помощью Leiningen:

`project.clj`

```
(defproject functional-programming-itmo-2021 "0.0.1"
            :description "FIXME: write description"
            :url "http://example.com/FIXME"
            :dependencies [[org.clojure/clojure "1.10.1"]
                           [nrepl/lein-nrepl "0.3.2"]
                           [org.clojure/core.async "1.4.627"]]
            :profiles {
                       :lab_1 {
                               :repl-options {
                                              :init-ns functional-programming-itmo-2021.lab-1.main
                                              :package functional-programming-itmo-2021.lab-1
                                              }
                               :main functional-programming-itmo-2021.lab-1.main
                               }
                       :lab_2 {
                               :repl-options {
                                              :init-ns functional-programming-itmo-2021.lab-2.main
                                              :package functional-programming-itmo-2021.lab-2
                                              }
                               :main functional-programming-itmo-2021.lab-2.main
                               }
                       :lab_3 {
                               :repl-options {
                                              :init-ns functional-programming-itmo-2021.lab-3.main
                                              :package functional-programming-itmo-2021.lab-3
                                              }
                               :main functional-programming-itmo-2021.lab-3.main
                               }
                       }
            )
```

Имплементация алгоритма Интерполяции:

`interpolation.clj`

```
(ns functional-programming-itmo-2021.lab-3.interpolation)

; Spline Cubic interpolation
; Implementation is transcoded from:
; github.com/Punctuality/Computational_Math_ITMO_2020/blob/master/lab4/src/main/scala/lab4/algorithm/SplineInterpolator.scala

(defn ^:private cubic-func [x-i a b c d]
   (fn [x] (let [diff (- x x-i)]
             (+ a (* b diff) (* c diff diff) (* d diff diff diff)))))

(defn ^:private cubic-spline [splines]
   (fn [x] (cond
             (< x (-> splines first first)) (apply (-> splines first last) [x]) ; x is smaller than a (extrapolation)
             (> x (-> splines last (get 1))) (apply (-> splines last last) [x]) ; x is bigger than b  (extrapolation)
             :else (apply (->> splines                      ;  x is inside [a, b]  (interpolation)
                               (filter #(let [[x-prev x-next _] %] (and (<= x-prev x) (<= x x-next))))
```

```
                              first
                              last) [x])))))

(defn spline-interpolator
   "Numerical Analysis R.L.Burden (Algorithm 3.4, p.168)"
   [points]
   (if (< (count points) 3)
      (fn [_] "Not enough data for spline interpolation")
      (let [n (dec (count points))
            xs (mapv first points)
            ys (mapv last points)
            hs (mapv #(- (get xs (inc %)) (get xs %)) (range n))
            alphas (mapv #(-
                           (/ (* 3.0 (- (get ys (inc %)) (get ys %))) (get hs %))
                           (/ (* 3.0 (- (get ys %) (get ys (dec %)))) (get hs (dec %)))
                           ) (range 1 n))

            ; tridiagonal linear system
            ; written using transient collections to be performant
            ; and also it is easier this way here
            l! (transient (vec (repeat (inc n) 1.0)))
            z! (transient (vec (repeat (inc n) 0.0)))
            mu! (transient (vec (repeat n 0.0)))]
        (dorun (map #(do
                       (assoc! l! % (-
                                     (* 2.0 (- (get xs (inc %)) (get xs (dec %))))
                                     (* (get hs (dec %)) (get mu! (dec %)))
                                     ))
                       (assoc! mu! % (/ (get hs %) (get l! %)))
                       (assoc! z! % (/ (-
                                        (get alphas (dec %))
                                        (* (get hs (dec %)) (get z! (dec %))))
                                       (get l! %)))
                       ) (range 1 n)))
        (let [z (persistent! z!) mu (persistent! mu!)
              as ys
              bs! (transient (vec (repeat n 0.0)))
              cs! (transient (vec (repeat (inc n) 0.0)))
              ds! (transient (vec (repeat n 0.0)))]
          (dorun (map #(do
                         (assoc! cs! % (- (get z %) (* (get mu %) (get cs! (inc %)))))
                         (assoc! bs! % (-
                                        (/ (- (get as (inc %)) (get as %)) (get hs %))
                                        (*
                                           (get hs %)
                                           (+ (get cs! (inc %)) (* 2.0 (get cs! %)))
                                           (/ 1.0 3.0)
                                           )))
                         (assoc! ds! % (/
                                        (- (get cs! (inc %)) (get cs! %))
                                        (* 3.0 (get hs %))
                                        ))
                         ) (reverse (range n))))
          (let [bs (persistent! bs!) cs (persistent! cs!) ds (persistent! ds!)]
             (cubic-spline (->> (range (dec (count xs)))
                                (map #(vector (get xs %)
                                              (get xs (inc %))
                                              (cubic-func
                                                 (get xs %)
                                                 (get as %)
                                                 (get bs %)
                                                 (get cs %)
                                                 (get ds %))))
                                (sort-by first)
                                )))))))
```

Самописный "фрейморк" для работы с функ-стримами:

`io_streaming.clj`

```
(ns functional-programming-itmo-2021.lab-3.io-streaming
   (:require [clojure.core.async :as a]
      [clojure.string :as str]))

(defn input-producer [inp-c]
   (a/go-loop [counter 0]
              (if-let [next-line (read-line)]
                 (if-let [_ (a/>! inp-c next-line)]
                    (recur (inc counter))
                    (println "Input channel was closed")))
              )
   inp-c)

(defn routing-channels [inp-c routing]
   (let [cs (assoc
               (into (hash-map) (mapv (fn [[key _]] [key (a/chan)]) routing))
               :else (a/chan))
         vec_rt (vec routing)]
      (a/go-loop []
                 (if-let [next-val (a/<! inp-c)]
                    (do
                       (let [channels (->> vec_rt
                                           (filter #(apply (last %) [next-val]))
                                           (mapv first)
                                           (mapv #(get cs %)))]
                          (if channels
                             (if (not (empty? channels))
                                (doseq [c channels]
                                   (a/>! c next-val))
                                (a/>! (cs :else) next-val)
                                )))
                       (recur)
                       )
                    (doseq [[_ channel] cs]
                       (println "Closing channels")
                       (a/close! channel)))
                 )
      cs
      ))

(defn output-consumer [inp-cs]
   (a/go-loop []
              (if-let [_ (a/alts! inp-cs)]
                 (recur)
                 (println "Output channel was closed"))
              ))

(defn middleware [inp-c func]
   (let [opt-c (a/chan)]
      (a/go-loop []
                 (if-let [next-val (a/<! inp-c)]
                    (do
                       (func next-val)
                       (a/>! opt-c next-val)
                       (recur))
                    (a/close! opt-c)
                    ))
      opt-c))
```

Входная точка и общая логика программы:

main.clj

```
(ns functional-programming-itmo-2021.lab-3.main
   (:require [functional-programming-itmo-2021.lab-3.io-streaming :as s]
      [functional-programming-itmo-2021.lab-3.interpolation :as i]
      [clojure.core.async :as a]
      [clojure.string :as str]))

(def train-points (ref []))
(def data-points (ref []))

(def control-rules {:train   #(str/starts-with? % "t")
                    :data    #(str/starts-with? % "d")
                    :predict #(str/starts-with? % "predict")})

(defn add-train-point [line]
   (let [split (vec (.split line ","))
         [_ x_s y_s & _] split
         x (Double/parseDouble x_s)
         y (Double/parseDouble y_s)]
      (println "Adding new train point: " x "->" y)
      (-> train-points
          (alter conj [x y])
          (dosync)
          )))

(defn add-data-point [line]
   (let [split (vec (.split line ","))
         [_ x_s & _] split
         x (Double/parseDouble x_s)]
      (println "Adding new data point: " x)
      (-> data-points
          (alter conj x)
          (dosync)
          )))

(defn predict-from-data [_]
   (let [sorted-train (sort-by first @train-points)
         interpolator (i/spline-interpolator sorted-train)
         data @data-points
         predicted (mapv interpolator data)]
      (print "Predicted: ")
      (doseq [i (range (count data))]
         (print " " (get data i) "->" (get predicted i) "|"))
      (println)
      ))

(defn error-line-handler [err-line]
   (println "Received unsupported line:" err-line))

(def stream (a/chan))

(defn shutdown-guard [effect]
   (let [shutdown-p (promise)
         hook-f (fn [] (do
                         (effect)
                         (println "Shutting down!!!")
                         (deliver shutdown-p 0)))]
      (.addShutdownHook (Runtime/getRuntime)
                        (Thread. ^Runnable hook-f))
      (System/exit @shutdown-p)))

(defn -main []
   (-> stream
       (s/input-producer)
```

```
        (s/routing-channels control-rules)
        (update :train #(s/middleware % add-train-point))
        (update :data #(s/middleware % add-data-point))
        (update :predict #(s/middleware % predict-from-data))
        (update :else #(s/middleware % error-line-handler))
        (vals)
        (s/output-consumer)
        )
    (shutdown-guard #(a/close! stream))
    )
```

Демонстрация работы программы:

```
cat input | lein with-profile lab_3 trampoline run
```

`stdin`

```
t,-5,25
t,-4,16
t,+4,16
t,5,25
d,-6
d,-5
d,5
d,6
predict
d,0
d,-2
d,0
d,2
t,0,0
predict
```

`stdout`

```
Adding new train point:  -5.0 -> 25.0
Adding new train point:  -4.0 -> 16.0
Adding new train point:  4.0 -> 16.0
Adding new train point:  5.0 -> 25.0
Adding new data point: -6.0
Adding new data point: -5.0
Adding new data point:  5.0
Adding new data point:  6.0
Predicted:   -6.0 -> 34.0 |  -5.0 -> 25.0 |  5.0 -> 25.0 |  6.0 -> 34.0 |  0.0 -> -0.6153846153846126 |  -2.0 -> 3.53846153846154 |
Adding new data point:  0.0
Adding new data point:  -2.0
Adding new data point:  0.0
Adding new data point:  2.0
Adding new train point:  0.0 -> 0.0
Predicted:   -6.0 -> 34.0 |  -5.0 -> 25.0 |  5.0 -> 25.0 |  6.0 -> 34.0 |  0.0 -> 0.0 |  -2.0 -> 3.875 |  0.0 -> 0.0 |  2.0 -> 3.875
```

Особенности реализации

Что есть: 1. Обработка сообщений в асинхронном порядке 2. После старта можем обрабатывать сообщения с использованием параллелизма (на thread-pool'e) 3. Для синхронизации состояния данных используется механизм STM 4. При подсчете аппроксимации используются transient коллекции для оптимизации производительности 5. Программа запускает асинхронный drain для функционального стрима, и в то же самое время блокируется (чтобы не завершиться) с shutdown guard, на механизме обещаний.

Чего нет *(да и не очень хочется):* 1. Не очень robust'но обрабатываются ошибки. Мне не хватило монады Try в clojure и я отлавливал некоторый специфичные ошибки, а не все. 2. Порядок сообщений истинно асинхронный, надо бы замутить какую-нибудь буффер с поумнее чем тот что у println, и пусть выводиться по времени invoke.