Inhaltsverzeichnis

1 Begriffe

1.1 Online-Probleme

Online-Probleme sind Probleme, bei denen die Eingabe zu Beginn nicht vollständig bekannt sind, sondern laufend hinzukommen. Sie sind eine wichtige Klasse von Problemen und treten beispielsweise bei der Planung von (CPU-)-Jobs / Scheduling auf.

Definition: Online Maximierungs-Problem

Online-Probleme bestehen aus Menge von Eingaben $\mathscr{I}$, Eingabe $I \in \mathscr{I}$ mit $I = (x_1, ..., x_n)$, sowie einer $(gain)$-Funktion. Zu jeder Eingabe kann eine Menge von Ausgaben mit jeweils O=(y_1, . . . , y_n) zugeordnert werden. $gain(I, O)$ ordnet dabei zu jeder Eingabe und passender Ausgabe eine positive, reelle Zahl zu. Für jede Eingabe I nennen wir diejenige Lösung O eine *optimale Lösung von I*, für welche $gain(I, O)$ maximal ist. Wir schreiben diese als Opt(I)

Definition: Online-Algorithmus mit *Advice*

1.2 Advice-Komplexität

2 Das einfache Online-Rucksackproblem

2.1 Greedy-Ansatz

$1 - \beta$

2.2 Untere Grenze für Optimalität

n-1 advice-bits

2.3 AONE - 2-competitive

2.4 Grenzen für log(n-1) Adice bits

$2 - \varepsilon$

2.5 SLOG

3 Vergleich mit randomisierten Algorithmen

3.1 1 Random bit: RONE - 4-competive

3.2 1 Random bit: RONE' - 2-competive

3.3 Mehr Random-bits: keine Verbesserung

4 Ausblick

4.1 Rucksackproblem mit Überfüllen

4.2 Das gewichtete Online-Rucksackproblem

5 todo:

- graph von bits / n, sprung-marken

6 Online-Knapsack-Problem

```
Constants =
    SCALE: 300
    KNAPSACK_SIZE: 1

roundValue = (value) -> Math.round(value*100)/100
```

6.1 Knapsack

The Knapsack will contain the objects and can return its gain (= the total of items' value)

```
Knapsack = class
    constructor: ->
        @size = Constants.KNAPSACK_SIZE
        @dep = new Tracker.Dependency
        @reset()

    reset: ->

        @items = []
        @dep.changed()

    fits: (item) ->  @gain() + item.value <= Constants.KNAPSACK_SIZE

    addItem: (item) ->
        if @fits item
            @items.push item
            @dep.changed()
    gain: ->
        roundValue _.reduce @getItems(), ((total, item) -> total+item.
           value), 0
    getItems: ->
        @dep.depend()
        @items


experiments = []

Algorithm = class
    constructor: ->
        @adviceBits = new ReactiveVar
        @act = new ReactiveVar
        @_knapsack = new Knapsack
```

```
    knapsack: -> @_knapsack
    askOracle: (items) ->
        if @oracle?
            @adviceBits.set @oracle items
            delete item.isPartOfSolution for item in items
    readAdviceBit: (index) ->
        @adviceBits.get()?[index]
    reset: ->
        @_knapsack.reset()
        @adviceBits.set null
        @act.set null
    handle: (item) ->
        if @decide item
            @_knapsack.addItem item
            yes
        else
            no
    doAct: (like) -> @act.set like
    acts: (like) -> @act.get() is like
```

Lets start with the greedy aproach. Here, we just take every item we get, if it fits:

```
decideGreedy = (item) -> if @knapsack().fits item then yes else no
```

and we define an algorithm with it:

```
Greedy = class extends Algorithm
    decide: decideGreedy
```

The gain of this algorithm is at least 1-b, where beta is the size of the item with the highest value (weight). Lets do some experiments with it:

```
experiments.push
    name: -> "Greedy G"
    description: -> "G archieves at least 1-beta, where beta is here
       #{@beta}"
    beta: 0.5
    Algorithm: Greedy

experiments.push
    name: -> "Greedy G"
    description: -> "G archieves at least 1-beta, where beta is here
       #{@beta}"
    beta: 0.2
    Algorithm: Greedy

experiments.push
    name: -> "Greedy G"
    description: -> "G archieves at least 1-beta, where beta is here
       #{@beta}"
    beta: 0.8
    Algorithm: Greedy
```

6.2 Advice bits

Imaging you had an oracle, that would know all items that will come. How many bits of information from this oracle would you need to get an optimal solution? And for a given amount of these advice bits, how good would your algorithm perform?

Let's start with the first question.

Consider an algorithm with an oracle, that would give us a bit for every item coming with

- value 1 if the item is part of the solution
- value 0 if the item does not belong to the solution

We now define an algorithm for that.

Note: The items are prepared in a way, that some are allready marked as solution. That makes it easier to define the oracle here:

```
TotalInformation = class extends Algorithm
    oracle: (items) ->
        bits = []
        for item in items
            bits[item.index] = if item.isPartOfSolution then 1 else 0
        # we do not need the last (n-1)
        bits.pop()
        return bits
```

The decision is now easy. If we have a bit (yes / no), we use it:

```
    decide: (item) ->
        adviceBit = @readAdviceBit item.index
        if adviceBit? then adviceBit else yes
```

Lets do an experiment with it:

```
experiments.push
    name: -> "Total Information"
    beta: 0.4
    Algorithm: TotalInformation
```

6.3 1 Advice bit

What's the best gain if we had only 1 advice bit?

Let's do an experiment where we have an oracle that gives us one bit:

```
AONE = class extends Algorithm
    oracle: (allItems) -> [ _.some allItems, (item) -> item.value >
        0.5 ] # array with one bit
```

The bit tells us:

- 1: There exists an item with a size > 0.5
- 0: There is no such item

If the bit is 0, the algorithm acts greedy (like before). If the bit is 1, the algorithm waits until the item with size > 0.5 appears and will start acting greedyly:

```
    decide: (item)->
        adviceBit = @readAdviceBit item.index
        if adviceBit? # existance
            if adviceBit is false then @doAct "greedy" else @doAct "
                wait"
        if @acts "greedy" then decideGreedy.call @, item else @wait
            item
```

```

    wait: (item) ->
        if item?.value > 0.5
            @doAct "greedy"
            decideGreedy.call @, item
        else
            no
```

We do an experiment with a max size of one item of 0.55:

```
experiments.push
    name: "AONE - with one advice bit"
    description: "AONE is 2-competitive"
    beta: 0.55
    Algorithm: AONE
```

6.4 Random Online-Algorithms

Obviously in real online-problems, we do not have an omniscient oracle. But we can use the idea of the oracle and just guess the advice bits *randomly*.

We can then estimate the competitiveness of this *randomized online-algorithm.

6.4.1 RONE - AONE with random advice bit

Let's start with AONE from the previous experiment, but guess the adviceBit randomly:

```
RONE = class extends AONE
    oracle: ->
        [Math.random() < 0.5]
```

If we guess wrong, we might get a lower gain then 0.5 or even 0, if the adviceBit is 1 and we have no item with size > 0.5.

So while we have

```
experiments.push
    name: "RONE - one random bit"
    description: "Is 4-competitive in expectation"
    beta: 0.55
    Algorithm: RONE

A1 = Greedy
A2 = class extends Algorithm
    reset: ->
        super
        @a1 = new A1
        @doAct "simulateA1"

    decide: (item) ->
        if @acts "simulateA1"
            if @a1.handle item
                no
            else
                @doAct "greedy"
                @decide item
        else if @acts "greedy"
            decideGreedy.call @, item

```

```
RONE2 = class extends Algorithm
    constructor: ->
        @a1 = new A1
        @a2 = new A2
        super
    oracle: -> [Math.random() < 0.5]
    reset: ->
        super
        @a1.reset()
        @a2.reset()
    knapsack: -> @algorithm().knapsack()
    handle: (item) ->
        adviceBit = @readAdviceBit item.index
        if adviceBit? # existance
            if adviceBit then @doAct "A1" else @doAct "A2"
        @algorithm().handle item

    algorithm: ->
        if @acts "A1" then @a1 else @a2




experiments.push
    name: "RONE2 - one random bit"
    description: "Is 2-competitive in expectation"
    beta: 0.55
    Algorithm: RONE2



createItems = ({beta, maxSize}) ->

    items = []
    beta ?= 0.5
    maxSize ?= 1
    totalSize = 0

    loop
        randomValue = -> roundValue Math.random()*beta
        value = randomValue()
        if totalSize+value < maxSize
            totalSize += value
            items.push {value, isPartOfSolution: yes}
        else
            # add one that fits exactly
            items.push
                value: roundValue maxSize - totalSize
                isPartOfSolution: yes
            # add the one that does not fit
            items.push {value}


            break
```

```


    items = _.shuffle items
    for item, index in items
        item.index = index
    return items.reverse() # we later pop the elements out (from the
        end) because it is faster


Template.experiments.helpers
    experiments: -> experiments



Template.Experiment.onCreated ->

    @items = []

    @currentItem = new ReactiveVar
    @numberOfItems = new ReactiveVar
    @algorithm = new @data.Algorithm
    @gainHistory =
        history: []
        dep: new Tracker.Dependency
        add: (gainValue) ->
            if gainValue > 0
                @worstGain = Math.min @worstGain ? gainValue,
                    gainValue
            @bestGain = Math.max @bestGain ? gainValue, gainValue
            @history.push gainValue
            @dep.changed()
        size: ->
            @dep.depend()
            @history.length
        worst: ->
            @dep.depend()
            @worstGain
        best: ->
            @dep.depend()
            @bestGain
        competitiveCount: ->
            @dep.depend()
            _.countBy @history, (value) ->

                if value is 1
                    "1-competitive"
                else if 0.5 <= value < 1
                    "2-competitive"
                else if 0.25 <= value < 0.5
                    "4-competitive"
                else
                    "non-competitive"


        competitivePercentage: (cGroup)->
            @dep.depend()
            if @history.length > 0
                roundValue 100 * @competitiveCount()[cGroup] /
                    @history.length
        avg: ->
            @dep.depend()
```

```
                if @history.length > 0
                    roundValue (_.reduce @history, (total, value) -> total
                        +value)/@history.length
            reset: ->
                @history = []
                @bestGain = null
                @worstGain = null
                @dep.changed()
    resetExperiment = =>

        @items = createItems beta: @data.beta
        @algorithm.reset?()
        @algorithm.askOracle? @items
        @numberOfItems.set @items.length
        @currentItem.set @items.pop()

    do reset = =>
        @gainHistory.reset()
        resetExperiment()

    @ticker = new Ticker
        reset: =>
            reset()

        turn: =>
            # 1. step: fetch new item
            # 2. step: put it in knapsack

            item = @currentItem.get()
            if item?
                @algorithm.handle item
                @currentItem.set @items.pop()
            else
                # no more items

                @gainHistory.add @algorithm.knapsack().gain()
                resetExperiment()



Template.Experiment.helpers
    adviceBits: -> Template.instance().algorithm.adviceBits.get()
    act: -> Template.instance().algorithm.act.get()
    knapsack: -> Template.instance().algorithm.knapsack()
    ticker: -> Template.instance().ticker
    currentItem: ->Template.instance().currentItem?.get()
    gainHistory: -> Template.instance().gainHistory
    numberOfItems: -> Template.instance().numberOfItems.get()
    willMatch: ->
        ctx = Template.instance()
        ctx.currentItem?.get()?.value + ctx.algorithm.knapsack().gain
            () <= ctx.algorithm.knapsack().size

Template.Knapsack.helpers
    totalWidth: ->
        @size * Constants.SCALE + 2
    items: ->
        @getItems()
Template.KnapsackItem.helpers
    width:  ->
        @value * Constants.SCALE
```

```
    color: ->
        hue = @value*360
        "hsl(#{hue}, 73%, 69%)"

Template.TickerGui.helpers
    counter: -> @ticker.getCounter()
Template.TickerGui.events
    'click .btn-step': -> @ticker.step()
    'click .btn-play': ->
        @ticker.setTimeout 100
        @ticker.play()
    'click .btn-play-fast': ->
        @ticker.setTimeout 0
        @ticker.play()

    'click .btn-stop': -> @ticker.stop()
    'click .btn-reset': -> @ticker.reset()
```