Contents

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

6.1 The knapsack-problem

Consider a knapsack with a certain capacity of weight (or volume) and a set of items, each with a value and a weight.

Which subset of these items would you put into the knapsack to get the maximum possible total value respecting the capacity of the knapsack?

This question is the so called knapsack-problem.

6.1.1 The simple-knapsack-problem

In this paper, we only consider the so called *simple-knapsack-problem* where the value of one item is the same as its weight and where the knapsack has always a capacity of 1.

We call the total value of all items in the knapsack as the *gain*.

Let's define such a knapsack:

```
Knapsack = class
    constructor: ->
        @size = 1
        @dep = new Tracker.Dependency
        @reset()

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

    reset: ->
        @items = []
        @dep.changed()
```

6.2 The Online-Knapsack-Problem

In the former *offline*-knapsack-problem, we know all items that we want to put in the knapsack. In the *online*-version of this problem, we do not know every item, but get the items one by one. We therefore have to decide after every item, whether we put the item in the knapsack or not.

We create a base-algorithm for that:

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

    handle: (item) ->
        if @decide item
            @_knapsack.addItem item
            return yes
        else
            return no
    decide: (item) ->
        # implement me and return yes or no

    reset: ->
        @_knapsack.reset()
        @act.set null
    doAct: (like) -> @act.set like
    acts: (like) -> @act.get() is like
```

What maximum gain would can we achieve and how would an online-algorithm perform in comparison with an optimal offline-algorithm, which would know every item?

Let's try out.

```
experiments = []
```

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 $1-\beta$, where β is the size of the item with the highest value (weight). The proof is simple: if we get this item with value β, the gain is certainly higher than β. If this item does not fit anymore in the knapsack, we will have at least $1-\beta$ gain.

Lets do some experiments with it to verify this:

```
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
```

6.3 Online-Algorithm with advice

```
experimentsWithAdvice = []
```

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?

We define such an algorithm as *online algorithm with advice*.

Let I be an input of such an online algorithm A and Φ an (infinite) sequence of bits (1 or 0), called *advice bits. The online-algorithm can read a finit prefix of this sequence.

The gain of this Algorithm is $gain(A^{\Phi}(I))$.

If we have *n* items in a solution and have read *s(n)* advice-bits while computing this solution in the algorithm we call *s(n)* the *advice-complexity*.

If we compare the *gain* of this algorithm with the gain of an optimal offline algorithm OPT, we can define its *competitiveness*:

$gain(A^{\Phi}(I))* \geq 1/c * gain(OPT(I)) - \alpha$

where α is a constant and we call this algorithm *c-competitive*. If $\alpha = 0$, *A* is *strictly c-competitive*.

Let's implement a base class for such an algorithm:

```
AlgorithmWithAdvice = class extends Algorithm
    constructor: ->
        @adviceBits = new ReactiveVar
        super
    askOracle: (items) ->
        if @oracle?
            @adviceBits.set @oracle items
    oracle: (items) ->
        # implement me and return an array of advice-bits
    readAdviceBit: (index) ->
        @adviceBits.get()?[index]
    reset: ->
        super
        @adviceBits.set null
```

6.4 Optimal online algorithm with advice

Let's go back to the first question with the first question: how many advice bits do we have to read to get an optimal solution?

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

Obviously, we need n bits of advice for that, or n-1, because for the last item, we can assume that it is part of the optimal 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 AlgorithmWithAdvice
    oracle: (items) ->
        bits = []
        for item in items
            bits[item.index] = if item.isPartOfSolution then 1 else 0
        # we do not need the last bit
        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:

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

As (**???**) states, any algorithm for the online simple knapsack problem needs at least n-1 bits to be optimal.

6.5 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 AlgorithmWithAdvice
    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
```

This algorithm is 2-competitive:

- If there is no item with weight > 1/2, the gain is at least 1/2 as we have already seen in the greedy approach.
- On the other hand if such an item exists, the algorithm will wait for it and put it in, so it will get a gain of at least 1/2

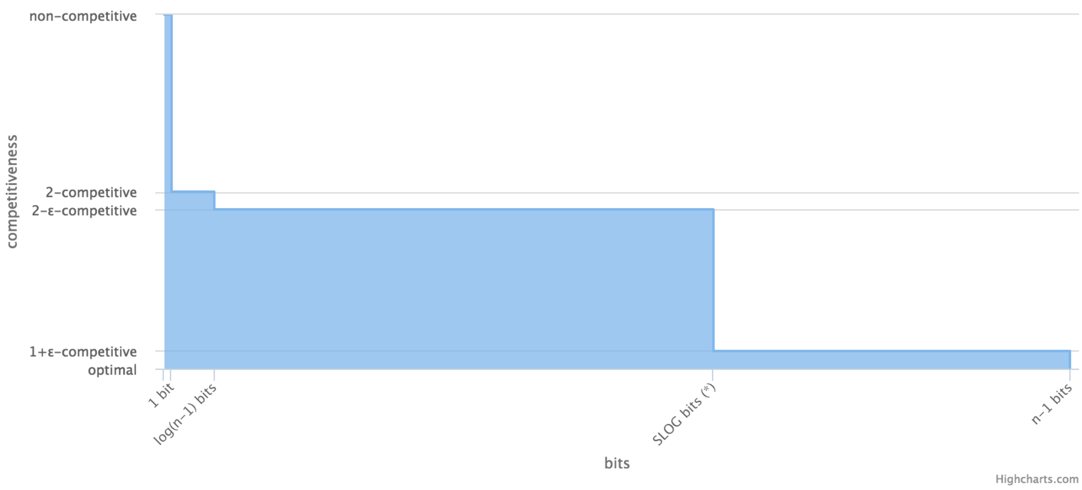
We do an experiment with a max size of one item of 0.55 to verify this:

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

This one single bit gives us an competitive-ratio of 2, but what happens if we increase the amount of bits? Can we achieve a better ratio?

Unfortunatly, more advice bits does not give us a better competitive-ratio, at least for a sub-logarithmic amount *s(n)* of advice bits. Figure 1 shows the number of bits compared with the achieved competitive-ratio.

There is a second jump at *SLOG*-bits, where competitiveness is $1+\varepsilon$.



(*) SLOG = $\left\lceil \frac{2\varepsilon+2}{\varepsilon} \right\rceil \cdot \lceil \log n \rceil + 2 \cdot \left\lceil \log \frac{2\varepsilon+2}{\varepsilon} \right\rceil + 2 \cdot \lceil \log \lceil \log n \rceil \rceil + 1$

Figure 1: Number of bits VS competitiveness

6.6 Randomized Online-Algorithms

```
randomExperiments = []
```

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*.

6.6.1 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 a 2-competivenes in AONE, we have here a 4-competitivenes in expectation (in 50% of the cases, we are wrong).

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

6.6.2 2-competivenes with 1 advice bit

The competitive-ratio of 4 is somewhat obvious, but suprisingly, we can also achieve a ratio of 2 with only 1 advice bit.

Consider an algorithm that choses randomly between two algorithms A1 and A2. A1 is the greedy approach we already know:

```
A1 = Greedy
```

A2 internaly simulates A1 at the beginning:

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

To decide wheter it will use the item or not, it first offers it to the simulated A1-Algorithm. As soon as A1 won't take the item anymore (A1' knapsack is full), A2 starts to act greedyly:

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

We now compose an algorithm “RONE2”, that choses randomly between A1 and A2:

```
RONE2 = class extends AlgorithmWithAdvice
    constructor: ->
        @a1 = new A1
        @a2 = new A2
        super
    oracle: -> [Math.random() < 0.5]
    reset: ->
        super
        @a1.reset()
        @a2.reset()
    knapsack: -> @algorithm().knapsack()
    # handle decides and put the item in the knapsack
    handle: (item) ->
        adviceBit = @readAdviceBit item.index
        if adviceBit? # existance of the first bit
            if adviceBit then @doAct "A1" else @doAct "A2"
        @algorithm().handle item
    algorithm: ->
        if @acts "A1" then @a1 else @a2
```

We do now an experiment with it:

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

To show that this algorithm is 2-competitive in expectation, we consider two cases:

- If the sum of all items is less than the knapsack’s capacity, A1 is optimal, while A2 gains 0. Because we chose randomly between the two algorithm, we have a 50% chance to get an optimal gain (or to get 0).
- If the sum is greater, the total gain of A1 and A2 is at least 1. Because we chose randomly between the two, we get a 0.5 gain in expecation.

Considering both cases, we get a gain of 0.5 in expecation, so the algorithm is 2-competitive.

7 Setup

The following code sets the experiments up. First, define some constants:

```
Constants =
    SCALE: 300

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

First, the creation of items:

```
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
    # we later pop the elements out (from the end) because it is faster. So we reverse here:
    return items.reverse()
```

Add the experiments to the template

```
Template.experiments.helpers
    experiments: -> experiments
    experimentsWithAdvice: -> experimentsWithAdvice
    randomExperiments: -> randomExperiments

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.competitivenessChart.helpers
    chartObject: ->
        legend: enabled: false
        title: text: ""
        yAxis:
            title: text: "competitiveness"
            tickPositioner: -> [1,1.1,1.9,2,3]
            labels:
                formatter: ->
                    switch @value
                        when 1 then "optimal"
                        when 1.1 then "1+eps-competitive"
                        when 1.9 then "2-eps-competitive"
                        when 2 then "2-competitive"
                        when 3 then "non-competitive"

        xAxis:
            title: text: "bits"
            tickPositioner: ->  [0,1,7,77,127]
            labels:
                rotation: -45
                formatter: ->
                    switch @value
                        #when 0 then "0 bits"
                        when 1 then "1 bit"
                        when 7 then "log(n-1) bits"
                        when 77 then "SLOG bits (*)"
                        when 127 then "n-1 bits"
        series: [
            type: "area"
            step: "left"
            data: [
                (x: 0, y: 3, name: "non-competitive")
                (x: 1, y: 2, name: "2-competitive")
                (x: 7, y: 1.9, name: "2-eps-competitive")
                (x: 77, y: 1.1, name: "1+eps-competitive")
                (x: 127, y: 1, name: "optimal")
            ]
        ]

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()
```