Online Knapsack Problem

Term Paper - Theoretical Computer Science

Marco Wettstein

2015-06-03

Contents

1 Preamble

Online problems and algorithms are problems where the inputs for the algorithm are not known at the beginning, but appear one by one. This could be a job-scheduler in an operating system where you have to decide whether to do a job immediatly after it appears or wait for other, maybe shorter jobs. It can be a memory-management that handles paging (also operation systems).

It could also be the decision whether to buy new ski-gear at the beginning of the season or rent it every day, when you do not know, how the weather will be like during the season. Every day, it could snow, rain or be a perfect powder-day, but should you buy skis on one sunny day, when you do not know if there will be another nice day to go to the mountains?

It would be nice to have some information about the future, something like an omniscient oracle, that gives us a glimpse of whats coming next.

We introduce such an oracle for online problems and try to find out, how much information do we need from this oracle to get an optimal solution.

In this interactive paper, we deal with the so called online simple knapsack problem, where we have knapsack that we want to fill with a maximum amount of value but respect the maximum capacity of it.

1.1 About this paper

This paper is written as *literate CoffeeScript*-source-code[1] of a set of experiments with these online problem running on *Meteor*[2].

It is compiled using *pandoc*[3], which enables you to compile Markdown and many other formats to latex+pdf.

A live version of the experiments is available at: http://online-knapsack.macrozone.ch, the source-code is available on github: https://github.com/macrozone/seminar_np.

[1] See http://coffeescript.org/#literate
[2] https://www.meteor.com/
[3] http://pandoc.org/

2 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.

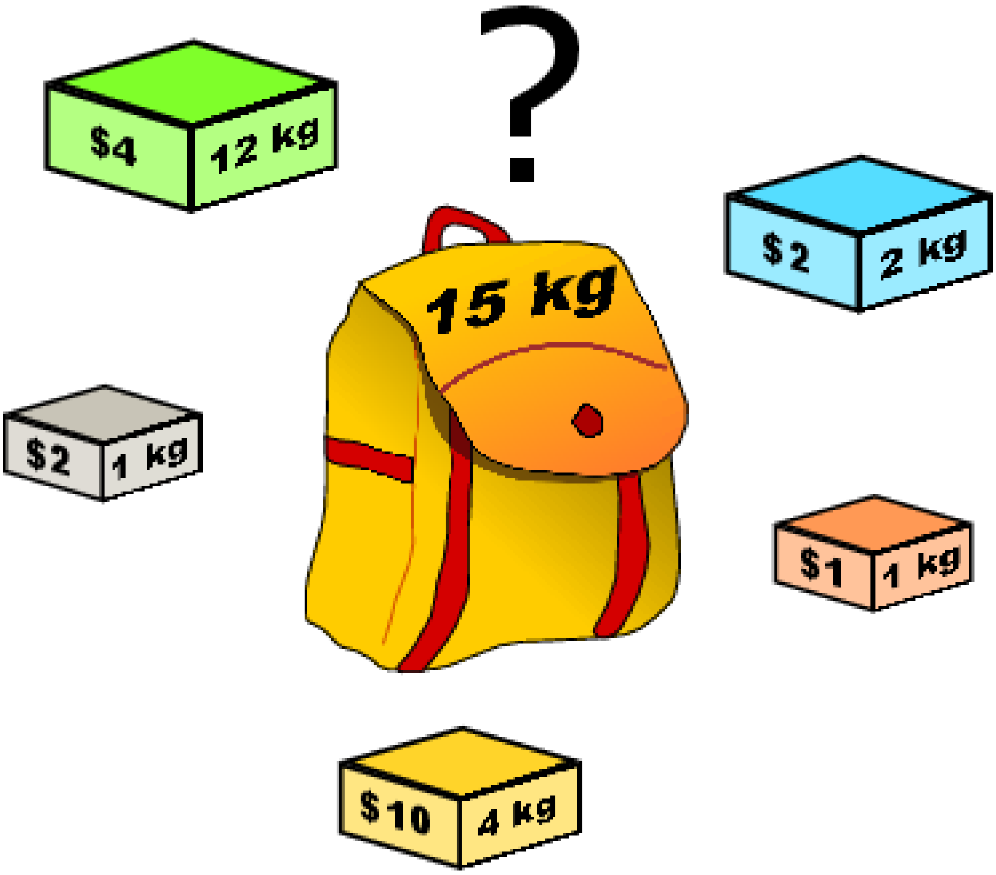


Figure 1: The knapsack problem (Source: wikipedia)

2.1 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()
```

2.2 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
```

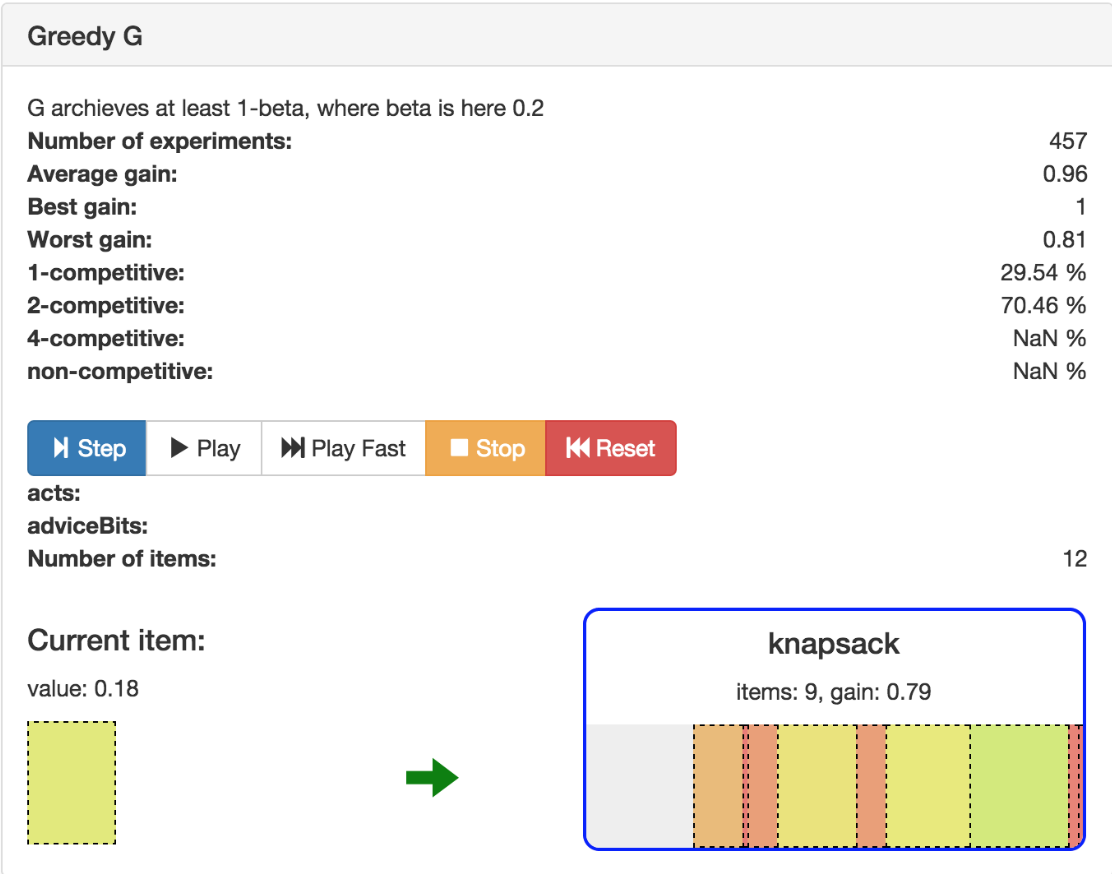


Figure 2: The greedy approach will at least gain $1 - \beta$

3 Online-Algorithm with advice

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*:

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

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

The param α is needed when the length of the input may vary and the algorithm could be bad for short, but perform well for long inputs (See also Komm, 4). For our knapsack problem, we can set $\alpha = 0$ and only consider *strict-competitiveness*, because the capacity of the knapsack is bound to 1 (Böckenhauer et al., 64).

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
```

3.1 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 = []
experimentsWithAdvice.push
    name: -> "Total Information"
    beta: 0.4
    Algorithm: TotalInformation
```

As (Böckenhauer et al., 64) states, any algorithm for the online simple knapsack problem needs at least n-1 bits to be optimal.

3.2 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

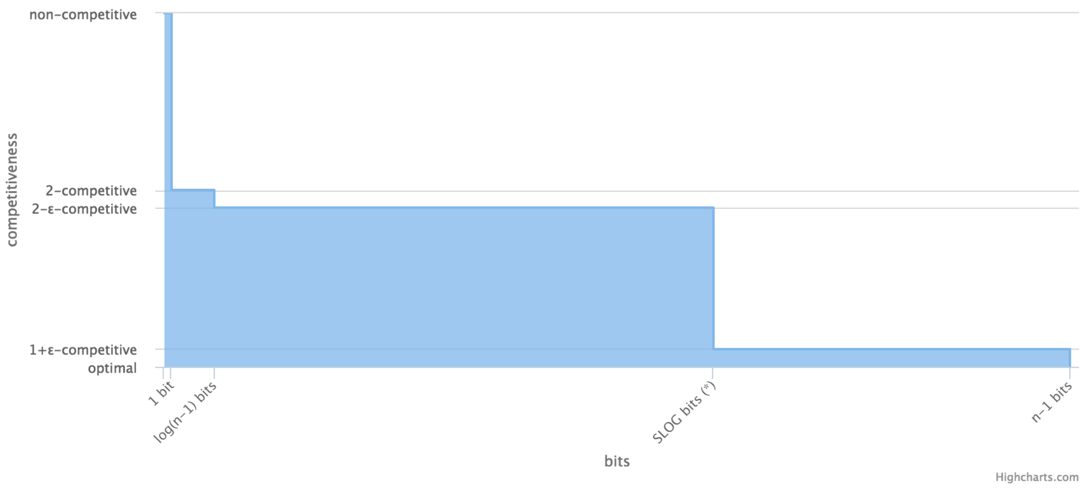
We do an experiment with a max size of one item of 0.55 to verify this:

```
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 3 shows the number of bits compared with the achieved competitive-ratio.

There is a second jump at *SLOG*-bits, where competitiveness is $1+\varepsilon$. The proof for these intervals is found in the source (Böckenhauer et al., 65).



(*) 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 3: Number of bits VS competitiveness

4 Randomized 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*.

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

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

The experiment does not show this directly, because the items are prepared in a way, that not all possible cases are evenly distributed. We expected that in 50% of the cases, the algorithm would guess wrongly and we would gain nothing, but in the experiment this probability is lower.

4.2 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.

4.3 The limit

While we can achieve different levels of competitivenesses by increasing the number of bits in online algorithms with advice, this is not the case in randomized online algorithms.

As (Böckenhauer et al.) states, there is no algorithm that performs better than 2-competitive in expecation. So 2-competiveness with 1 bit is the best we can achieve.

5 Whats next

Resource augmentation: If we allow the online algorithm to pack a little bit more (δ) in the knapsack than allowed, we can achieve up to $(2-\delta)$-competitiveness.

The weighted case: In this paper, we only considered items, where the value is equal to the weight of the item. If we introduce a different weight of each item, we will see, that online algorithms for this weighted knapsack problem is only competitive for at least a logarithmic amount of advice bits.

Randomized online algorithms for the weighted knapsack: If we create a randomized online algorithm for the weighted case, we see that these algorithms are not competitive at all, with and without resource-augmentation.

Further details and proof for these extensions can be found in the source (Böckenhauer et al.) .

6 Setup

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

```
Constants =
    SCALE: 300

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

6.1 Create items

The creation of items is done here. The items are prepared in a way, so that we now which elements are part of the solution (for experiment "Total information").

```
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()
```

6.2 Experiment templates

Add the experiments to the template:

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

Initialize it. We use some ReactiveVars to store the state of the experiment on the template instance.

```
Template.Experiment.onCreated ->
    @items = []
    @currentItem = new ReactiveVar
    @numberOfItems = new ReactiveVar
    @algorithm = new @data.Algorithm
```

Lets define a history, where we can read some stats about the experiments from:

```
    @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()
```

6.3 Initialize and reset experiment

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

    do reset = =>
```

```
        @gainHistory.reset()
        resetExperiment()
```

6.4 Running the experiment

The Ticker is a package, that can run a callback in a loop. We can run it step-by-step or fast.

```
    @ticker = new Ticker
        reset: => reset()
        turn: =>
            item = @currentItem.get()
            if item?
                @algorithm.handle item
                @currentItem.set @items.pop()
            else
                # no more items
                @gainHistory.add @algorithm.knapsack().gain()
                resetExperiment()
```

Expose the state to the template:

```
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%)"
```

6.5 Chart

The chart from 3 is created by this code:

```
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")
            ]
        ]
```

6.6 GUI for the Ticker

```
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()
```

A Additional source files

A.1 client/app.jade (template)

```

head
    title Advice Complexity of the Online-Knapsack-Problem
body
    .container
        .page-header
            h1 Advice Complexity of the Online-Knapsack-Problem
            img(src="knapsack.svg" style="display:block; margin: 0
               auto")
        +experiments



template(name="experiments")

    each experiments
        +Experiment
    h2 Online algorithms with advice
    each experimentsWithAdvice
        +Experiment
    +competitivenessChart
    h2 Randomized online algorithm
    each randomExperiments
        +Experiment



template(name="Experiment")
    .panel.panel-default
        .panel-heading
            h2.panel-title {{name}}
        .panel-body
            {{description}}
            .row
                .col-xs-12.col-sm-5
                    table.table
                        tr
                            th Number of experiments:
                            td {{gainHistory.size}}
                        tr
                            th Average gain:
                            td {{gainHistory.avg}}
                        tr
                            th Best gain:
                            td {{gainHistory.best}}
                        tr
                            th Worst gain:
                            td {{gainHistory.worst}}
                        tr
                            th 1-competitive:
                            td {{gainHistory.competitivePercentage "1-
                               competitive"}} %
                        tr
                            th 2-competitive:
                            td {{gainHistory.competitivePercentage "2-
                               competitive"}} %
```

```
                        tr
                            th 4-competitive:
                            td {{gainHistory.competitivePercentage "4-competitive"}} %
                        tr
                            th non-competitive:
                            td {{gainHistory.competitivePercentage "non-competitive"}} %
            .row
                .col-xs-12
                    +TickerGui ticker=ticker
            .row
                .col-xs-12.col-sm-5
                    table.table
                        tr
                            th acts:
                            td {{act}}
                        tr
                            th adviceBits:
                            td.adviceBits=adviceBits
                        tr
                            th Number of items:
                            td {{numberOfItems}}

            .row
                .currentItemContainer.col-xs-6
                    h4 Current item:

                    if currentItem
                        p value: {{currentItem.value}}

                        +KnapsackItem(currentItem)
                        if willMatch
                            .matches.yes
                                .glyphicon.glyphicon-arrow-right
                        else
                            .matches.no
                                .glyphicon.glyphicon-remove

                .knapsackContainer.col-xs-6
                    +Knapsack knapsack


template(name="Knapsack")
    .knapsack(style="width: {{totalWidth}}px")
        h4 knapsack
        p items: {{items.length}}, gain: {{gain}}
        .items

            each items
                +KnapsackItem


template(name="KnapsackItem")
    .knapsack-item(style="width: {{width}}px; background-color: {{color}}")

template(name="competitivenessChart")


    .panel.panel-default
```

```
        .panel-heading
            h2.panel-title Bits VS competitiveness
        .panel-body
            +highchartsHelper chartId="competitivenessChart" chartWidth="100%" chartHeight="400px" chartObject=chartObject
            p (*) SLOG =
                img(src="slog.png" height="40px")

template(name="TickerGui")
    .ticker
        .btn-group
            button.btn.btn-step.btn-primary
                .glyphicon.glyphicon-step-forward
                | Step
            button.btn.btn-play.btn-default
                .glyphicon.glyphicon-play
                | Play
            button.btn.btn-play-fast.btn-default
                .glyphicon.glyphicon-fast-forward
                | Play Fast
            button.btn.btn-stop.btn-warning
                .glyphicon.glyphicon-stop
                | Stop
            button.btn.btn-reset.btn-danger
                .glyphicon.glyphicon-fast-backward
                | Reset
```

A.2 client/app.styl (stylesheet)

```

body
    background-image: url("bg.png")

.knapsack
    border: 2px solid blue
    box-sizing: content-box
    border-radius: 10px
    overflow: hidden
    text-align: center
    .items
        background-color: #eee
        height: 75px
        .knapsack-item
            float: right;
            border-right: none
.knapsack-item
    border: 1px dashed black
    height: 75px
    box-sizing: border-box

.currentItemContainer
    position: relative
    .matches
        position: absolute
        right: 60px
        bottom: 16px
        border-radius: 10px
        font-size: 32px
        &.yes
            color: green
        &.no
            color: red

.adviceBits
    font-family: monospace

.table
    td
        text-align: right
    th
        text-align: left

.panel
    margin-bottom: 200px
```

References

Böckenhauer, Hans-Joachim, Dennis Komm, Richard Kralovic, and Peter Rossmanith. "The Online Knapsack Problem: Advice and Randomization." http://e-collection.library.ethz.ch/eserv/eth:5665/eth-5665-01.pdf.

Komm, Dennis. "Eine Einführung in Online-Algorithmen." http://www.ita.inf.ethz.ch/~dkomm/online-algorithmen.pdf.