Two Monoids for Approximating $\mathbb{NP}$-Complete Problems

by Mike Izbicki ⟨mike@izbicki.me⟩

August 7, 2013

As a TA, I was confronted with a real life instance of the $\mathbb{NP}$-complete Scheduling *problem. To solve the problem, I turned to the classic Least Processing Time First (LPTF) approximation algorithm. In this article, we'll see that because LPTF is a monoid homomorphism, we can implement it using HLearn's* `HomTrainer` *type class. This gives us parallel and online versions of LPTF "for free." We'll also be able to use these same techniques to solve a related problem called* BinPacking. *Hopefully, at the end of the article you'll understand when the* `HomTrainer` *class might be a useful tool, and how to use and build your own instances.*

Framing The Problem

I enjoy TAing the introduction to C++ course at my university. Teaching pointer arithmetic can be immensely frustrating, but it's worth it to see the students when it all finally clicks. Teaching is even better when it causes you to stumble onto an interesting problem. Oddly enough, I found this cool Haskell problem because of my C++ teaching assistanceship.

The professor wanted to assign a group project, and I had to pick the groups. There had to be exactly five groups, and the groups needed to be as fair as possible. In other words, I was supposed to evenly distribute the best and worst students.

After a little thought, I realized this was an instance of the $\mathbb{NP}$-complete Scheduling problem in disguise. This problem was first formulated in the context of concurrent computing. In the textbook example of Scheduling, we are given p processors and n tasks. Each task t_i has some associated time it will take to complete it. The goal is to assign the tasks to processors so as to complete the tasks as quickly as possible.

The SCHEDULING problem is shown graphically in Figure 1. Processors are drawn as bins, and tasks are drawn as green blocks inside the bins. The height of each task represents the length of time required to process it. Our goal is to minimize the processing time given a fixed number of processors.

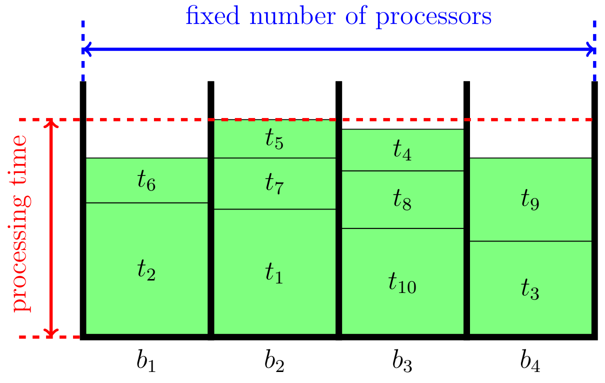

Figure 1: The SCHEDULING problem

What does this have to do with the problem my professor gave me? Well, we can think of the number of groups as the number of processors, each student is a task, and the student's current grade is the task's processing time. Then, the problem is to find a way to divvy up the students so that the sum of grades for the "best" group is as small as possible. We'll see some code for solving this problem in a bit.

There is another closely related problem called BINPACKING that is easily confused with SCHEDULING. In BINPACKING, instead of fixing the number of bins and minimizing the amount in the bins, we fix the bin size and minimize the total number of bins used. Compare Figure 2 below and Figure 1 above to see the difference. The BINPACKING problem was originally studied by shipping companies, although like SHEDULING it occurs in many domains.

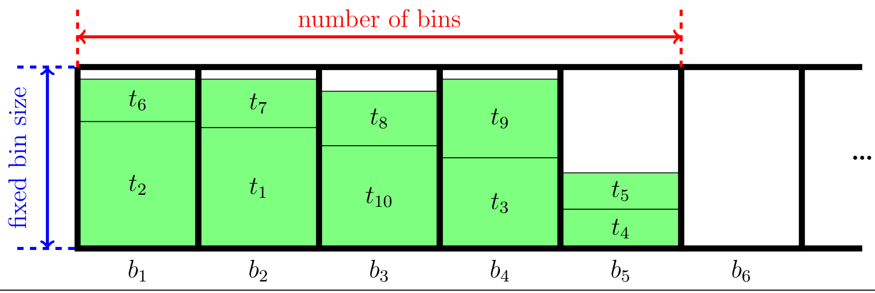

Figure 2: The BINPACKING problem

Both Scheduling and BinPacking are ℕℙ-complete. Therefore, I had no chance of creating an optimal grouping for my professor—the class had 100 students, and 2^{100} is a *big* number! So I turned to approximation algorithms. One popular approximation for Scheduling is called Longest Processing Time First (LPTF). When analyzing these approximation algorithms, it is customary to compare the quality of their result with that of the theoretical optimal result. In this case, we denote the total processing time of the schedule returned by LPTF as *LPTF*, and the processing time of the optimal solution as *OPT*. It can be shown that the following bound holds:

$$LPTF \leq \left(\frac{4}{3} - \frac{1}{3n}\right) OPT$$

This bound was proven in the late 1960's, but the original paper remains quite readable today [1]. By making this small sacrifice in accuracy, we get an algorithm that runs in time $\Theta(n \log n)$ instead of $\Theta(2^n)$. Much better! In the rest of this article, we'll take a detailed look at a Haskell implementation of the LPTF algorithm, and then briefly use similar techniques to solve the BinPacking problem.

The Scheduling HomTrainer

When implementing an algorithm in Haskell, you always start with the type signature. LPTF takes a collection of tasks and produces a schedule, so it's type might look something like:

```
:: [Task] → Schedule
```

Anytime I see see a function of this form, I ask myself, "Can it be implemented using HLearn's `HomTrainer` type class?" `HomTrainers` are useful because the compiler automatically derives online and parallel algorithms for all instances. In this section, we'll get a big picture view of how this class will help us solve the Scheduling problem. We start by looking at the format of a `HomTrainer` instance as shown graphically in Figure 3 below.

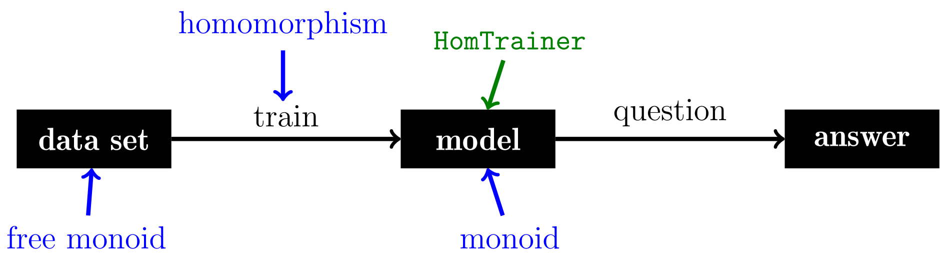

Figure 3: Basic requirements of the HomTrainer type class

There's a lot going on in Figure 3, but we'll look at things piece-by-piece. The black boxes represent data types and the black arrows represent functions. In our case, the data set is the collection of tasks we want scheduled and the model is the schedule. The train function is the LPTF algorithm, which generates a model from the data points. Finally, our model is only useful if we can ask it questions. In this case, we might want to ask, "Which processor is assigned to task t_{10}?" or "What are all the tasks assigned to processor 2?"

The blue arrows in the diagram impose some constraints on the data types and functions. These requirements are a little trickier: they specify that our training algorithm must be a **monoid homomorphism** from the **free monoid**. These are scary sounding words, but they're pretty simple once you're familiar with them. We'll define them and see some examples.

In Haskell, the `Monoid` type class is defined as having an identity called `mempty` and a binary operation called `mappend`:

```
class Monoid m where
    mempty  :: m
    mappend :: m → m → m
```

Sometimes, we use the infix operation `(◇) = mappend` to make our code easier to read. All instances must obey the identity and associativity laws:

```
mempty ◇ m = m ◇ mempty = m
(m1 ◇ m2) ◇ m3 = m1 ◇ (m2 ◇ m3)
```

Lists are one of the simplest examples of monoids. Their identity element is the empty list, and their binary operation is concatenation:

```
instance Monoid [a] where
    mempty = []
    mappend = ++
```

Lists are an example of free monoids because they can be generated by any underlying type. For the `HomTrainer`, when we say that our data set must be a free monoid, all we mean is that it is a collection of some data points. For the Scheduling problem, it is just a list of tasks.

A homomorphism from the free monoid is a function that "preserves the free monoid's structure." More formally, if the function is called `train`, then it obeys the law that for all `xs` and `ys` of type `[a]`:

```
train (xs ++ ys) = (train xs) ◇ (train ys)
```

The LPTF algorithm turns out to have this property. Figure 4 shows this in picture form with a commutative diagram. This means that it doesn't matter whether we take the **orange** path (first train schedules from our data sets, then combine the schedules with `mappend`) or the **purple** path (first concatenate our data sets, then train a schedule on the result). Either way, we get the exact same answer.

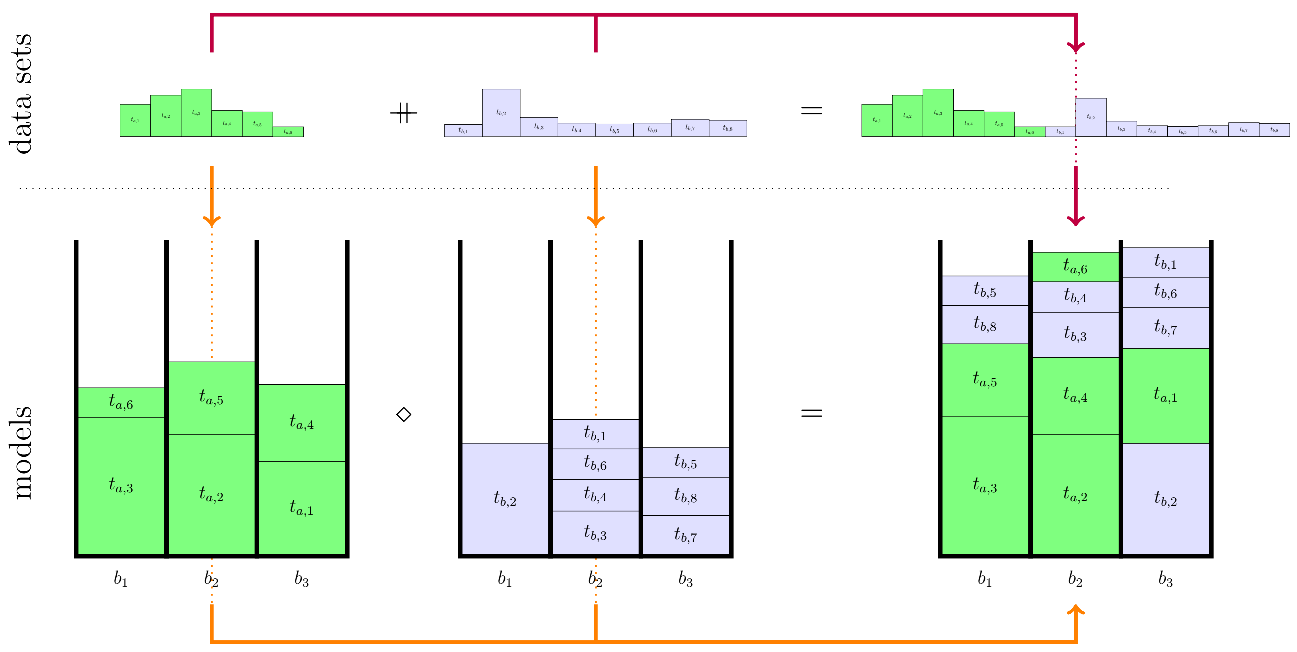

Figure 4: The LPTF is a monoid homomorphism because this diagram commutes

Now that we understand what `HomTrainer` is, we can look at what it gives us. Most importantly, it gives us a simple interface for interacting with our models. This interface is shown in Figure 5. In the class, we associate a specific `Datapoint` type to our model and get four functions for training functions. The most important training function is the batch trainer, called `train`. This is the homomorphism that converts the data set into a model. In our case, it will be the LPTF algorithm. The second most important function is the online trainer `add1dp`. This function takes a model and a datapoint as input, and "adds" the data point to the model. Developing new online functions is an important research area in approximation algorithms. As we will see later, the compiler generates these two functions automatically for all instances of `HomTrainer`.

Finally, HLearn comes with a higher order function for making all batch trainers run efficiently on multiple cores. The function

```
parallel :: (...) ⇒
    (container datapoint → model) → (container datapoint → model)
```

takes a batch trainer as input and returns a parallelized one as output. In the next section, we'll see an example of its use in practice.

```
class (Monoid model) ⇒ HomTrainer model where
    type Datapoint model

    -- The singleton trainer
    train1dp :: Datapoint model → model

    -- The batch trainer
    train :: (Functor container, Foldable container) ⇒
        container (Datapoint model) → model

    -- The online trainer
    add1dp :: model → Datapoint model → model

    -- The online batch trainer
    addBatch :: (Functor container, Foldable container) ⇒
        model → container (Datapoint model) → model
```

Figure 5: The `HomTrainer` type class

Using the Scheduling HomTrainer

Before we look at implementing a `HomTrainer` to solve SCHEDULING, we'll take a look at how it's used. In particular, we'll look at the Haskell code I used to solve the problem of grouping my students. In order to run the code, you'll need to download the latest `HLearn-approximation` library:

```
cabal install HLearn-approximation-1.0.0
```

Let's begin by doing some experiments in GHCi to get ourselves oriented. The `Scheduling` type is our model, and here we ask GHCi for it's kind signature:

```
ghci> import HLearn.NPHard.Scheduling
ghci> :kind Scheduling
Scheduling :: Nat → ∗ → ∗
```

`Scheduling` takes two type parameters. The first is a type-level natural number that specifies the number of processors in our schedule. In HLearn, any parameters to our training functions must be specified in the model's type signature. In this case, the type-level numbers require the `DataKinds` extension to be enabled. The second parameter to `Scheduling` is the type of the task we are trying to schedule.

Next, let's find out about `Scheduling`'s `HomTrainer` instance:

```
ghci> :info Scheduling
...
```

```
instance (Norm a,...) ⇒ HomTrainer (Scheduling n a) where
...
```

We have a constraint on our task parameter specifying that it must be an instance of `Norm`. What does that mean? In mathematics, a type has a norm if it has a "size" of some sort. In HLearn, the `Norm` type class is defined in two parts. First we associate a specific number type with our model using the `HasRing` type class. Then, the `Norm` type class defines a function `magnitude` that converts a model into the number type.

```
class (Num (Ring m)) ⇒ HasRing m where
    type Ring m

class (HasRing m, Ord (Ring m)) ⇒ Norm m where
    magnitude :: m → Ring m
```

Usually the associated ring will be a `Double`. But we might make it a `Rational` if we need more accuracy or an `Int` if we don't want fractions.

Figure 6 shows the code for solving the student groups problem. In this case, I defined a new data type called `Student` to be the data points and defined the `magnitude` of a `Student` to be its `grade`. Notice that since I am deriving an instance of `Ord` automatically, I must define `grade` before `name` and `section`. This ensures that the ordering over students will be determined by their `grade`s, and so be the same as the ordering over their `magnitude`s.

The `main` function is divided up into three logical units. First, we load a CSV file into a variable `allStudents::[Student]` using the `Cassava` package [2]. The details of how this works aren't too important.

In the middle section, we divide up the students according to which section they are in, and then `train` a `Schedule` model for each section. We use the function:

```
getSchedules :: Scheduling n a → [[a]]
```

to extract a list of schedules from our `Schedule` type, then print them to the terminal. Just for illustration, we train the third section's `Scheduling` model in parallel. With only about 30 students in the section, we don't notice any improvement. But as the data sets grow, more processors provide drastic improvements, as shown in Table 0.1.

In the last section, we combine our section specific models together to get a combined model factoring in all of the sections. Because `Scheduling` is an instance of `HomTrainer`, we don't have to retrain our model from scratch. We can reuse the work we did in training our original models, resulting in a faster computation.

```
{-# LANGUAGE TypeFamilies, DataKinds #-}

import Data.Csv
import qualified Data.ByteString.Lazy.Char8  as BS
import qualified Data.Vector  as V
import HLearn.Algebra
import HLearn.NPHard.Scheduling

--------------------------------------------------------------------

data Student = Student
   { grade   :: Double
   , name    :: String
   , section :: Int
   }
   deriving (Read,Show,Eq,Ord)

instance HasRing Student where
   type Ring (Student) = Double

instance Norm Student where
   magnitude = grade

--------------------------------------------------------------------

main = do
   Right allStudents ←
      fmap (fmap (fmap (λ(n,s,g) → Student g n s) . V.toList) . decode True)
      $ BS.readFile "students.csv" :: IO (Either String [Student])

   let section1 = filter (λs → 1 == section s) allStudents
   let section2 = filter (λs → 2 == section s) allStudents
   let section3 = filter (λs → 3 == section s) allStudents
   let solution1 = train section1 :: Scheduling 5 Student
   let solution2 = train section2 :: Scheduling 5 Student
   let solution3 = parallel train section3 :: Scheduling 5 Student
   print $ map (map name) $ getSchedules solution1

   let solutionAll = solution1 ◇ solution2 ◇ solution3
   print $ map (map name) $ getSchedules solutionAll
```

Figure 6: Solution to my professor's problem

Implementing the LPTF HomTrainer

Now we're ready to dive into the details of how our model, `Scheduling` works under the hood. `Scheduling` is defined as:

```
data Scheduling (p::Nat) a = Scheduling
    { vector   :: !(SortedVector a)
    , schedule :: Map Bin [a]
    }
```

Scheduling has two member variables. The first is a `SortedVector`. This custom data type is a wrapper around the `vector` package's `Vector` type that maintains the invariant that items are always sorted. This vector will be used as an intermediate data structure while performing the LPTF computations. The second member is the actual schedule. It is represented as a `Map` with a `Bin` as the key and a list of tasks as the value. `Bin` is just a type synonym for `Int`:

```
type Bin = Int
```

and it represents the index of the processor in the range of 1 to p.

Before we look at `Scheduling`'s `Monoid` and `HomTrainer` instances, we need to take a more detailed look at the LPTF algorithm. Traditionally, LPTF is described as a two step process. First, sort the list of tasks in descending order. Then, iterate through the sorted list. On each iteration, assign the next task to the processor with the least amount of work. Figure 7 shows a single iteration of this procedure.

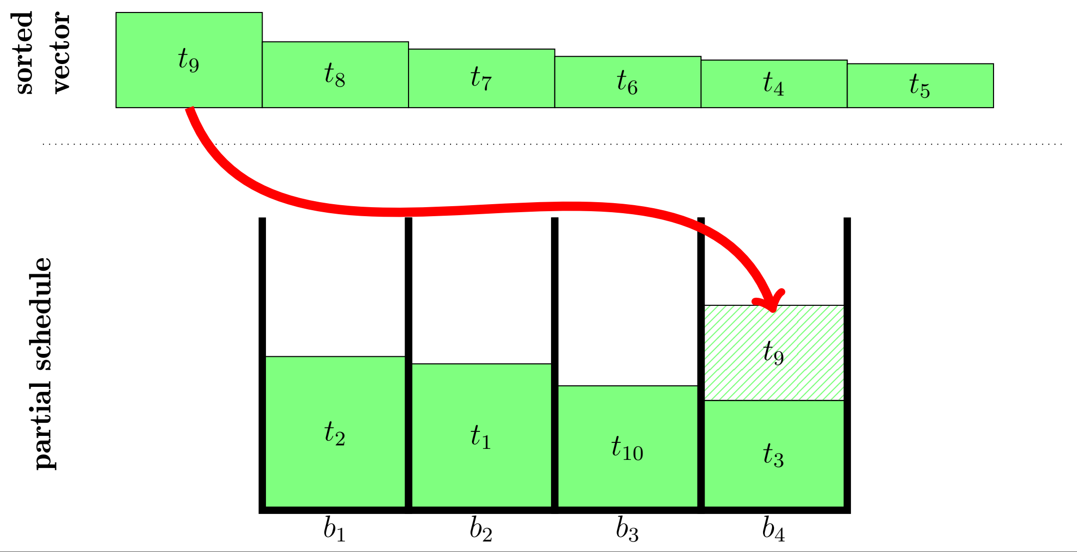

Figure 7: A single iteration of the LPTF algorithm

This conversion process is implemented with the internal function `vector2schedule`, whose code is shown in Figure 8 below. The details of this function aren't particularly important. What is important is that `vector2schedule` runs in time $\Theta(n \log p)$. This will be important when determining the run times of the `mappend` and `train` functions.

```
vector2schedule :: (Norm a) ⇒ Int → SortedVector a → Map.Map Int [a]
vector2schedule p vector = snd $ F.foldr cata (emptyheap p,Map.empty) vector
    where
        emptyheap p = Heap.fromAscList [(0,i) | i←[1..p]]
        cata x (heap,map) =
            let Just top = Heap.viewHead heap
                set = snd top
                prio = (fst top)+magnitude x
                heap' = Heap.insert (prio,set) (Heap.drop 1 heap)
                map' = Map.insertWith (++) set [x] map
            in (heap',map')
```

Figure 8: The `vector2schedule` helper function for LPTF

Our `mappend` operation will implement the LPTF algorithm internally in a way that reuses the results from the input `Schedules`. We won't be able to reuse the actual schedules, but we can reuse the sorting of the vectors. We do this by taking advantage of the `HomTrainer` instance of the `SortedVector` type. It turns out that merge sort is a monoid homomorphism, and so `SortedVector` can be made an instance of the `HomTrainer` type class. The commutative diagram for `SortedVector` is shown in Figure 9 below.

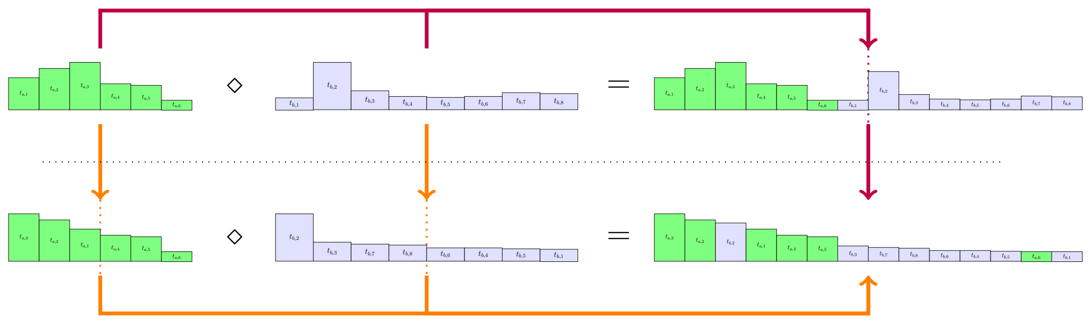

Figure 9: Constructing a `SortedVector` is a monoid homomorphism

It is important to note that `SortedVector`'s `mappend` operation does not take constant time. In fact, it takes $\Theta(n)$ time, where n is the size of both input vectors put together. The `HomTrainer` type class makes reasoning about these non-constant `mappend` operations easy. By looking up in Table 0.1, we can find the run times of the derived algorithms. Notice that if the monoid operation takes time $\Theta(n)$, then our batch trainer will take time $\Theta(n \log n)$, and this is exactly what we would expect for a sorting. Details of how these numbers were derived can be found in my TFP13 submission on the HLearn library [3].

Monoid operation (`mappend`)	Sequential batch trainer (`train`)	Parallel batch trainer (`parallel train`)	Online trainer (`add1dp`)
$\Theta(1)$	$\Theta(n)$	$\Theta\left(\frac{n}{p} + \log p\right)$	$\Theta(1)$
$\Theta(\log n)$	$\Theta(n)$	$\Theta\left(\frac{n}{p} + (\log n)(\log p)\right)$	$\Theta(\log n)$
$\Theta(n)$	$\Theta(n \log n)$	$\Theta\left(\frac{n}{p} \log \frac{n}{p} + n\right)$	$\Theta(n)$
$\Theta(n^b)$, $b > 1$	$\Theta(n^b)$	no improvement	no improvement

Table 0.1: Given a run time for `mappend`, you can calculate the run time of the automatically generated functions using this table. The variable n is the total number of data points being trained on or combined, and the variable p is the number of processors available.

With all of these building blocks in place, the `Monoid` instance for `Scheduling` is relatively simple. The `mempty` and `mappend` operations are exactly the same as they are for `SortedVector`, except that we also call the helper function `lptf`. This function just packages the `SortedVector` into a `Scheduling` type using the `vector2schedule` function we saw earlier.

```
instance (Ord a, Norm a, SingI n) ⇒ Monoid (Scheduling n a) where
    mempty = lptf mempty
    p1 ‘mappend‘ p2 = lptf $ (vector p1) ◇ (vector p2)

lptf :: forall a p. (Norm a, SingI p) ⇒ SortedVector a → Scheduling p a
lptf vector = Scheduling
    { vector = vector
    , schedule = vector2schedule p vector
    }
    where p = fromIntegral $ fromSing (sing :: Sing n))
```

Figure 10: The `Monoid` instance for the `Scheduling` model

Since `vector2schedule` runs in linear time and `SortedVector`'s `mappend` runs in linear time, the `Scheduling`'s `mappend` runs in linear time as well. By Table 0.1 again, we have that the automatically derived batch trainer will take time $\Theta(n \log n)$. This is exactly what the traditional LPTF algorithm takes.

Of course, we still have to implement the `HomTrainer` instance. But this is easy. Inside the `HomTrainer` class is a function called the "singleton trainer":

```
train1dp :: HomTrainer model ⇒ Datapoint model → model
```

All this function does is create a model from a single data point.[1] In practice, such a singleton model is rarely useful by itself. But if we define it, then the compiler can then use this function and `mappend` to build the other functions within the `HomTrainer` class automatically. This is how we get the online and parallel functions "for free."

The resulting `Scheduling` instance looks like:

```
instance (Norm a, SingI n) ⇒ HomTrainer (Scheduling n a) where
    type Datapoint (Scheduling n a) = a
    train1dp dp = lptf $ train1dp dp
```

Figure 11: The `HomTrainer` instance is quite short and mechanical to write

That's all we *need* to do to guarantee correct asymptotic performance, but we've got one last trick that will speed up our `train` function by a constant factor. Recall that when performing the `mappend` operation on `Scheduling` variables, we can only reuse the work contained inside of `vector`. The old `schedules` must be completely discarded. Since `mappend` is called many times in our automatically generated functions, calculating all of these intermediate schedules would give us no benefit but result in a lot of extra work. That is why in the `Scheduling` type, the `vector` member was declared strict, whereas the `schedule` member was declared lazy. The `schedules` won't actually be calculated until someone demands them, and since no one will ever demand a schedule from the intermediate steps, we never calculate them.

Back to Bin Packing

Since BinPacking and Scheduling were such similar problems, it's not too surprising that a similar technique can be used to implement a `BinPacking` model.

[1] The `train1dp` function is analogous to the `pure` function in the `Applicative` class, or the `return` function in the `Monad` class.

The main difference is that we'll replace the LPTF algorithm with another simple algorithm called Best Fit Decreasing (BFD). This gives us the performance guarantee of:

$$BFD \leq \frac{11}{9} OPT + 1$$

There are some slightly better approximations for BinPacking, but we won't look at them here because they are much more complicated. Chapter 2 of *Approximation Algorithms for NP Hard Problems* gives a good overview on the considerable amount of literature for bin packing [4].

BFD is a two stage algorithm in the same vein as LPTF. First, we sort the data points by size. Then, we iteratively take the largest item and find the "best" bin to place it in. The best bin is defined as the bin with the least amount of space that can still hold the item. If no bins can hold the item, then we create a new bin and add the item there. This is shown graphically in Figure 12 below.

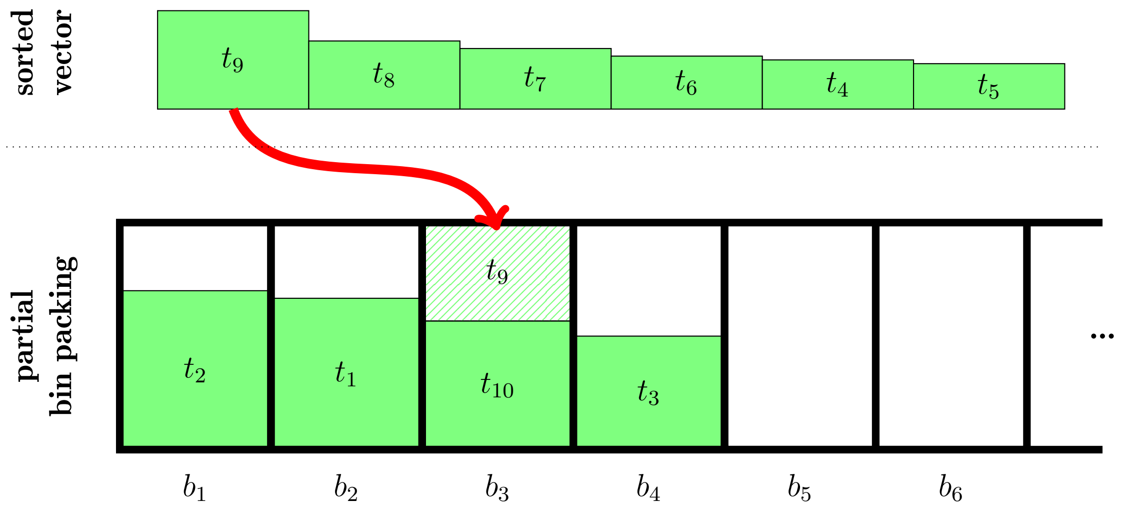

Figure 12: One iteration of the Best First Decreasing (BFD) algorithm

The data type for bin packing is:

```
data BinPacking (n::Nat) a = BinPacking
    { vector  :: !(SortedVector a)
    , packing :: Map.Map Bin [a]
    }
```

This is the exact same form as the Scheduling type had. The only difference is that we will use the BFD strategy to generate our `Map`. Therefore, by similar reasoning, the `BinPacking`'s `mappend` function takes time $\Theta(n)$ and its `train` function takes time $\Theta(n \log n)$. Again, this is exactly what the traditional description of the BFD algorithm requires.

Takeaways

Most instances of the `HomTrainer` type class are related to statistics or machine learning, but the class is much more general than that. For example, we've just seen how to use `HomTrainer`s to approximate two $\mathbb{NP}$-complete problems. So from now on, whenever you see a function that has type:

```
:: [datapoint] → model
```

ask yourself, "Could this algorithm be implemented using a `HomTrainer`?" If yes, you'll get online and parallel versions for free.

Finally, we've looked at the monoid structure for `Scheduling` and `BinPacking`, but these types also have Abelian group, $\mathbb{Z}$-module, functor, and monad structure as well. I'll let you explore the documentation available on the GitHub repository [5] (pull requests are always welcome!) to find creative ways to exploit these structures. If you have any questions or feedback, I'd love to hear it.

References

[1] R. L. Graham. Bounds on multiprocessing timing anomalies. **SIAM Journal on Applied Mathematics**, 17(2):pages 416–429 (1969).

[2] Cassava: A csv parsing and encoding library. `http://hackage.haskell.org/package/cassava`.

[3] Michael Izbicki. Hlearn: A machine learning library for haskell. **Trends in Functional Programming** (2013).

[4] E.G. Coffman Jr., M.R. Garey, and D.S. Johnson. **Approximation algorithms for NP-hard problems**, chapter Approximation Algorithms for Bin Packing: A Survey. PWS Publishing Co., Boston, MA, USA (1997).

[5] Hlearn source repository. `http://github.com/mikeizbicki/hlearn`.