Educational Data Synthesizer

Hoang-Trieu Trinh, Behzad Beheshti, Michel C. Desmarais

2016-09-04

Load the package

```
library(edmsyn)
```

Modifying `edmsyn`

`edmsyn` comes with a pre-defined set of parameters and relationships amongst themselves. These relationships are rules that help `edmsyn` derive values for one or more parameters from some others. Specifically, these rules are represented as functions in the package. For example, a function that takes two integers and randomly produce a binary matrix with its dimensions being the two inputted integers can be used as the rule to derive `M` (skill mastery matrix) from `students` and `concepts`. Rules that derive value for "data parameter" such as `poks`, `dina`, or `dino` encode POKS, DINA, DINO models respectively. Similarly, rules that derive value in the opposite direction encode the corresponding learning algorithms.

The choice of built-in parameters, models, and learning algorithms is made independently at the time of development for `edmsyn` and thus, it may or may not satisfy users' need. That is why `edmsyn` also comes with a set of tools that allow its users to re-define all these components to the extent of building a whole new set of parameters and models, while still retaining all the original benefits that it offers. All possible changes on `edmsyn` will be made on a single graphical structure that `edmsyn` creates at the loading time, with vertices representing parameters and edges encoding the respective relationships.

Functions that allow these modifications have names starting with `edmtree.` There are just a few of such functions and they provide everything you need to work with `edmsyn` at its internal level. This tutorial will also walk you through the building blocks of them so that an in-depth understanding of `edmsyn` mechanism is provided. By the end of this section, you are expected to handle `edmtree.` functions properly and efficiently.

If you do not feel going that deep, feel free to skip to **A toy model** at your own risk. Reading **A toy model** before going through the whole tutorial is also a good suggestion.

Fetching a node using `edmtree.fetch`

Firstly, let's start with the skill mastery matrix `M`

```
M.node <- edmtree.fetch('M')
class(M.node)
```

```
## [1] "list"
```

```
names(M.node)
```

```
## [1] "tell"   "gen"    "f.tell" "f.gen"
```

The representation of `M` in `edmsyn` is essentially a list with four components `tell`, `f.tell`, `gen`, and `f.gen`. The first one, `tell`, is a set of names of parameters that receive information if the value of `M` is known.

```
M.node$tell
```

```
## [1] "concepts"    "students"    "concept.exp"
```

In this case, the value of `M` *tells* `edmsyn` the values of `concepts`, `students` and `concept.exp` (expected mastery rate for each concept). This is quite straightforward since `concepts`, `students`, and `concept.exp` are respectively the row dimension, column dimension, and row means of `M`. At this point, it is reasonable to look at the third component `f.tell`

```
M.node$f.tell
```

```
## function (x)
## {
##     list(nrow(x), ncol(x), rowMeans(x))
## }
## <environment: 0x3944018>
```

The third component, `f.tell`, is the function to derive values for each parameter listed in `tell` from value of `M`. As can be seen, this function does exactly what we expected: taking the row dimension, column dimension and row means value of its input `M`, assemble these results into a list as its return value.

The second component, `gen`, is a list of generating methods for `M`.

```
M.node$gen
```

```
## [[1]]
## [1] "S"
##
## [[2]]
## [1] "students"    "skill.space" "skill.dist"
##
## [[3]]
## [1] "students"    "concept.exp"
```

In this example, `edmsyn` knows that there are three different methods to reach `M`: either using (`S`), (`students`,`skill.space`,`skill.dist`), or (`students`,`concept.exp`). If the generating process of `M` is somehow determined to use `S` (the skill matrix, or the probabilistic version of `M`), then one way to proceed can be rounding `S` element-wise to obtain `M`. In fact, this is the chosen method in `edmsyn`, let's have a look at the last component

```
M.node$f.gen
```

```
## [[1]]
## function (x)
## {
##     round(x[[1]])
## }
## <environment: 0x3944018>
##
## [[2]]
## function (x)
```

```
## {
##     x[[2]][, sample(1:length(x[[3]]), size = x[[1]], prob = x[[3]],
##         replace = TRUE)]
## }
## <environment: 0x3944018>
##
## [[3]]
## function (x)
## {
##     conexp <- x[[2]]
##     sapply(1:x[[1]], function(dum) {
##         sapply(conexp, function(p) {
##             sample(0:1, 1, prob = c(1 - p, p))
##         })
##     })
## }
## <environment: 0x3944018>
```

As can be seen, `f.gen` is a list of three functions correspond to three generating methods in `gen`, and the first one is somehow rounding its input for the return value. The details of these functions will be discussed in later part, at this point it is sufficient to just realise what is the representation of a parameter in `edmsyn` and which tasks it is trying to achieve.

From this point on, all `f.tell` functions will be referred to as type-1 connections. Similarly, the collections of functions in `f.gen` will be referred to as type-2 connections. These are the two types of connections along which all data flow. The main benefit `edmsyn` offers is a convenient interface that allow easy access and control to various data processes.

Replacing using `edmtree.replace`

Replacing type-2 connections

Suppose you are not satisfied with the current generation method for `M` from `S` (rounding), and expect a fully probabilistic implementation of it. To do this, the first function in `f.gen` should be replaced. Firstly, we must design a function that takes one input being the matrix `S`, sample binary result from `S`'s entries and return the value of `M` as follows:

```
new.gen.S.to.M <- function(S){
  M <- matrix(0, nrow(S), ncol(S))
  for (row in 1:nrow(S))
    for (col in 1:ncol(S)){
      p <- S[row,col]
      M[row, col] <- sample(x = 0:1, 1, prob = c(1 - p, p))
    }
  return(M)
}
```

Next, this function is replaced into the internal structure of `M`

```
edmtree.replace.gen('M','S',new.gen.S.to.M)
M.node <- edmtree.fetch('M')
M.node$f.gen
```

```
## [[1]]
## function (x)
## {
##     do.call(f.gen.copy[[1]], x)
## }
## <environment: 0x3334380>
##
## [[2]]
## function (x)
## {
##     x[[2]][, sample(1:length(x[[3]]), size = x[[1]], prob = x[[3]],
##         replace = TRUE)]
## }
## <environment: 0x3944018>
##
## [[3]]
## function (x)
## {
##     conexp <- x[[2]]
##     sapply(1:x[[1]], function(dum) {
##         sapply(conexp, function(p) {
##             sample(0:1, 1, prob = c(1 - p, p))
##         })
##     })
## }
## <environment: 0x3944018>
```

Now that `f.gen[1]` is changed. But apparently it is not the original `new.gen.S.to.M` that has been designed by us, the reason is that `edmsyn` has wrapped this function inside one or more layers to make sure it fit in perfectly with the internal working environment. We can check if the replacement is carried out successfully by simply generating `M` from `S` many times, if the result is different each time, the probabilistic version has been installed successfully.

```
S <- matrix(runif(15),3,5)
p <- pars(S = S)
M1 <- get.par('M',p)$value
M2 <- get.par('M',p)$value
# This is very unlikely to be TRUE
identical(M1, M2)
```

```
## [1] FALSE
```

```
# Let's try something else
M = matrix(0, 3, 5)
big.number = 10000
for (i in 1:big.number)
  M = M + get.par('M',p)$value
# This is likely to be TRUE
identical(round(M/big.number), round(S))
```

Similarly, we can change the functions of other generating methods by simply create a new function for each and replace them into the structure.

```
# Examples of changing generating methods
new.f.gen.2 <- function(students, skill.space, skill.dist){
  # Evaluate M here
  return(value.of.M)
}
edmtree.replace.gen('M', M.node$gen[[2]], new.f.gen.2)
```

As long as the number and the order of arguments match that of what is defined in the corresponding `gen`, the replacement will be done successfully. Otherwise, error will be detected either at replacement time or run-time depending on the error.

```
# Some more playing around with replacement
M.node$gen[[3]]
```

```
## [1] "students"    "concept.exp"
```

```
new.gen.3 <- rev(M.node$gen[[3]])
new.gen.3
```

```
## [1] "concept.exp" "students"
```

```
# Notice that the order of arguments of new.f.gen.3 below
# matches the order of new.gen.3
new.f.gen.3 <- function(concept.expectation, num.of.students){
  # evaluate M here
  return(value.of.M)
}
# Successful
edmtree.replace.gen('M', new.gen.3, new.f.gen.3)
```

Below are some examples where `edmtree.replace` fails to execute.

```
# Unsucessful examples:
# new f.gen is supposed to have 3 arguments as M.node$gen[[2]] has length 3
edmtree.replace.gen('M', M.node$gen[[2]], function(arg1, arg2) {
  return(NULL) # does not matter
})
```

```
## Error in edmtree.check(node.name, node.val, int, dat): number of arguments of f.gen[[2]] must match t
```

```
# (items, poks) is not an existing method
edmtree.replace.gen('M', c('items', 'poks'), function(arg1, arg2){
  return(NULL) # does not matter
})
```

```
## Error in edmtree.replace.gen("M", c("items", "poks"), function(arg1, arg2) {: method (items, poks) ca
```

Replacing type-1 connections

Besides `edmtree.replace.gen`, `edmsyn` also allow `edmtree.replace.tell` where users are able to modify `tell` and `f.tell` in many flexible ways. Knowing the value of `S`, it is straightforward to infer values of `students` and `concepts`, being the number of columns and rows respectively. As a toy example, let's swap the inference of `students` and `concepts` (and thus produce an incorrect result)

```
S.node <- edmtree.fetch('S')
S.node$tell
```

```
## [1] "concepts"  "students"  "S.st.var"  "S.con.exp"
```

```
p <- pars(S = S)
print(dim(S))
```

```
## [1] 3 5
```

```
# this would give correct result
c(p$concepts, p$students)
```

```
## [1] 3 5
```

```
# Now let's alter f.tell so that concepts
# is the number of columns of S and students
# is the number of rows of S.
edmtree.replace.tell('S',c('concepts','students'), function(S){
  return(list( ncol(S), nrow(S) ))
})
p <- pars(S=S)
```

```
## Error in down.stream(new.pars): 'concepts' receives different values at once
```

The reason why `pars` fails to execute is that, not only `concepts` and `students` are being inferred from `S`, `S.con.exp`, a vector of length `concepts`, is also being inferred *correctly*. This caused a conflict that `edmsyn` was able to detect. One way to force the incorrect result is as follows:

```
edmtree.replace.tell('S',c('S.con.exp','concepts','students'), function(S){
  return(list( colMeans(S), ncol(S), nrow(S) ))
})
p <- pars(S=S)
print(dim(S))
```

```
## [1] 3 5
```

```
# Now we successfully produce a wrong result.
c(p$concepts, p$students)
```

```
## [1] 5 3
```

There is two important points to make here: firstly, it is dangerous to replace such incorrect calculations into `edmsyn` structure: from what we have seen above, these false implementations may take long (sometimes forever) to be detected after the replacement and thus, your application may have operated incorrectly for many steps without being noticed. Take careful examination before you decide to change `edmsyn`'s built-in functions.

Secondly, the replacement we did above is called *partial replacement* because not all parameters in `tell` is modified. Unlike many other classes of objects in R, `f.tell`, which is essentially an R `function`, is inherently not modular, which means it can not be modified partially. So how did `edmsyn` finish the task above? It simply executes the old function *and then* the new one, after that new results will replace the old ones at appropriate positions in the returned list. This, apparently, may involve a great deal of redundant calculations and thus, the use of `edmtree.replace.tell` to modify a subset of `tell` is strongly discourage. In other words, whenever it is possible, try to use `edmtree.replace.tell` with argument `tell` being an identical set to `S$tell`.

To proceed normally, let's recover the original correct version of `f.tell` inside `S`

```
edmtree.replace.tell('S', S.node$tell, S.node$f.tell)
p <- pars(S=S)
dim(S)
```

```
## [1] 3 5
```

```
c(p$concepts, p$students)
```

```
## [1] 3 5
```

Naive replacing with `edmtree.replace`

The previous replacing functions are *smart* because they are flexible in terms of partial replacement. Namely, they retain the parts that is not declare in their function call and properly handle the integration of new parts with old parts.

If you wish to simply throw away some parts of the node and subtitude new ones, (in other words, you do not care about retaining any part of the existing functions, either they are useful to be reused or not), simply use `edmtree.replace`. See some examples below

```
# replacing tell
edmtree.replace('S', tell = new.tell)
# replacing tell and f.gen
edmtree.replace('S', tell = new.tell, f.gen = new.f.gen)
# replacing gen, f.gen and f.tell
edmtree.replace('S', f.gen = new.f.gen, f.tell = new.f.tell, gen = new.gen)
```

Removing using `edmtree.remove`

Removing type-2 connections

Let's start with a context where *students* and *concepts* are defined

```
p <- pars(students = 20, concepts = 15)
M <- get.par('M', p, progress = TRUE)
```

```
## Generate M from c("students", "concept.exp")
## Generate concept.exp from concepts
```

Assume that we desire not to produce `M` from the process that requires generating `concept.exp`. In fact, we want to remove this method of generating `M` from `edmsyn` completely. `edmtree.remove.gen` is here to help:

```
edmtree.remove.gen('M',c('concept.exp','students'))
M <- get.par('M', p, progress = TRUE)
```

```
## Generate M from c("students", "skill.space", "skill.dist")
## Generate skill.space from c("concepts", "skill.space.size")
## Generate skill.space.size from concepts
## Generate skill.dist from skill.space.size
```

```
edmtree.remove.gen('M',c('skill.dist','students','skill.space'))
M <- get.par('M', p, progress = TRUE)
```

```
## Generate M from S
## Generate S from c("concepts", "students")
```

```
edmtree.remove.gen('M', 'S')
M <- get.par('M', p, progress = TRUE)
```

```
## Error in up.stream(target, pars, FALSE, progress): Cannot reach 'M' since 'M' is missing
```

As we consecutively went through the deletion of all three generating methods for `M`, a reasonable question is raised: why does it have to be in the order of (`concept.exp`, `students`), then (`skill.dist`, `students`, `skill.space`), and lastly (`S`)? In other words, base on which criteria did `edmsyn` prioritise the method (`concept.exp`, `students`) over (`skill.dist`, `students`, `skill.space`); and (`skill.dist`, `students`, `skill.space`) over (`S`)? Note that when `students` and `concepts` are provided, all three options are possible to implement.

The generating critera

One complication of the `edmsyn` perspective is that, there will be times when there are more than one way to generate data. One example is the case above. There is critera that `edmsyn` follows in terms of prioritizing one option over another. Note that this criteria is chosen by the author at the time of developing `edmsyn` and there is no certain theoretical foundation for it. It is, however, beneficial for the users to be informed anyway.

If there are more than one *available* method to generate a node's value:

1. Choose the one that most utilizes inputted data. In the previous example, inputted data includes `students` and `concepts`, thus both (`concept.exp`, `students`) and (`skill.dist`, `students`, `skill.space`), utilizing a half of the input, are certainly better than (`S`).
2. If there is a tie, as pointed out in the `M$gen` example above, choose the one is that most covered by the input. In this case, one half of (`concept.exp`, `students`) is covered (by `students`), while only one third of (`skill.dist`, `students`, `skill.space`) is covered (also by `students`). This is why (`concept.exp`,`students`) is most favoured amongst the three.
3. If there is still a tie, then choose the one that most cover `tell` of the target node. For example in this case, `M$tell` consists of `concepts`, `students`, and `concept.exp`; two of them is covered by (`concept.exp`, `students`), one is covered by (`skill.dist`, `students`, `skill.space`) and none is covered by (`S`). For ties beyond this point, the choice is completely random.

Removing type-1 connections

Now we move on to removing type-1 connections. Let's say we want to discard any information about `concepts` inferred from `M`, `edmtree.remove.tell` should be used as follows:

```
edmtree.remove.tell('M', 'concepts')
p <- pars(M = M1)
print(p)
```

```
## Activated information in the context:
## [1] "M"            "default.vals" "students"     "concept.exp"
## [5] "concepts"
```

Why does `concepts` is still there in p? Actually in this case `concepts` is not inferred from `M` but from `concept.exp`, to completely eliminate `concepts`, one would need to remove `concept.exp`

```
edmtree.remove.tell('M','concept.exp')
p <- pars(M = M1)
# now concept.exp is gone, along with concepts
print(p)
```

```
## Activated information in the context:
## [1] "M"            "default.vals" "students"
```

```
# recover
edmtree.replace('M', tell = M.node$tell, f.tell = M.node$f.tell)
edmtree.fetch('M')
```

```
## $tell
## [1] "concepts"    "students"    "concept.exp"
##
## $gen
## list()
##
## $f.tell
## function (x)
## {
##     list(nrow(x), ncol(x), rowMeans(x))
## }
## <environment: 0x3944018>
##
## $f.gen
## list()
```

Again, since `f.tell` cannot be partially modified, the users should be aware of the fact that there are redundant calculations going under the implementation of the removed `concepts`. In fact, `concepts` is still being calculated, but then discarded from the returned list each time information is inferred from `M`. So it is highly recommended that the `f.tell` argument of `edmtree.remove.tell` is used whenever it is possible. Namely, the recommended way to completely remove `concepts` is as follows:

```
edmtree.remove.tell('M', c('concept.exp', 'concepts'), f.tell = function(M){
  return(ncol(M)) # since the only one left in tell is students
})
p <- pars(M = M1)
```

```
## Warning in get(var.name, envir = STRUCTURE)$f.tell(var.val): node 'M' :
## f.tell did not return a list, coerced to list of length 1
```

The warning here tells us that, the returning line inside our new `f.tell` should be `return(list(ncol(M))` instead of `return(ncol(M))`. However, `edmsyn` is smart enough to prevent this error by a function wrapper that detects and fixes many errors like these whenever it is possible. Note that since this is an error at returning of our function (in other words, it requires actual execution of the function to be presented), this error is a run-time error and cannot be detected immediately in the `edmtree.remove` step, but much later when `pars` calls for data to flow from `M` to lower level parameters in the execution of `p <- pars(M = M1)`.

To avoid displaying this warning each time data flow through `M`, let's fix `f.tell` by a proper one.

```
# recover
edmtree.replace('M', tell = M.node$tell, f.tell = M.node$f.tell)
# remove
edmtree.remove.tell('M', c('concept.exp', 'concepts'), f.tell = function(M){
  return(list(ncol(M)))
})
p <- pars(M = M1)
print(p)
```

```
## Activated information in the context:
## [1] "M"            "default.vals" "students"
```

Removing a node from the structure

Removing a whole node is syntactically simple but requires a high level of awareness. Removing root parameters such as `students` or `concepts` can cause the whole system to crash since everything is built up from these root parameters. In short, removing parameters at higher level creates less an impact on the whole structure than removing lower ones.

In this first example, let's remove a "data parameter", namely `bkt`:

```
bkt.node <- edmtree.remove("bkt")
```

```
## Successfully removed bkt
```

```
edmtree.fetch("bkt")
```

```
## Error in edmtree.fetch("bkt"): 'bkt' is not found in current tree
```

Later on, plugging `bkt.node` back into the structure is simple, but only because this is a special case where `bkt` is at data level. The situation is more complicated when deleting a lower level node such as `concept.exp`.

```
# For the purpose of illustration, let's first recover everything for M
edmtree.replace('M', tell = M.node$tell, f.tell = M.node$f.tell,
                gen = M.node$gen, f.gen = M.node$f.gen)
concept.exp.node <- edmtree.remove("concept.exp")
```

```
## Remove concept.exp from M$tell

## Remove method (students, concept.exp) from M$gen

## Remove concept.exp from nmf.dis$tell

## Remove concept.exp from nmf.com$tell

## Remove concept.exp from nmf.con$tell

## Successfully removed concept.exp
```

```
get.par('concept.exp', pars(concepts = 3))
```

```
## Error in edmtree.fetch(node.name): 'concept.exp' is not found in current tree
```

As can be seen, removing `concept.exp` will consequently remove several other parts of the whole structure. Let's say you want to reverse `edmsyn` back to the state before this removal, you will have to plug the `concept.exp.node` back and *manually* recover everything else that have gone along.

To do this, for example, adding method (`students`, `concept.exp`) back into `M`, `edmtree.add` is needed.

Adding with `edmtree.add`

Adding a node

This task is simple so long as you have already done the hard part: defining all four components `tell`, `f.tell`, `gen` and `f.gen` of the node. With node `concept.exp` in the removal section above, luckily we have saved these components into `concept.exp.node`. Let's reuse them for a quick illustration

```
edmtree.add('concept.exp', tell = concept.exp.node$tell,
            f.tell = concept.exp.node$f.tell,
            gen = concept.exp.node$gen,
            f.gen = concept.exp.node$f.gen)
concept.exp <- get.par('concept.exp', pars(concepts=3))$value
print(concept.exp)
```

```
## [1] 0.6769895 0.4247487 0.9317472
```

Adding type-2 connection

Now move on to adding method (`concept.exp`, `students`) into `M`. Again, we will reuse `M.node` for a quick and clean illustration

```r
# First let's see what method is being used to generate M
# When students and concept.exp is given
p <- pars(students = 10, concept.exp = concept.exp)
M <- get.par('M', p, progress = TRUE)
```

```
## Generate M from c("students", "skill.space", "skill.dist")
## Generate skill.space from c("concepts", "skill.space.size")
## Generate skill.space.size from concepts
## Generate skill.dist from skill.space.size
```

```r
# See what we have in M.node
M.node$gen
```

```
## [[1]]
## [1] "S"
##
## [[2]]
## [1] "students"    "skill.space" "skill.dist"
##
## [[3]]
## [1] "students"    "concept.exp"
```

```r
# Add the new method
edmtree.add.gen('M', gen.method = M.node$gen[[3]],
                f.gen.method = M.node$f.gen[[3]])

# Generate M again to see the change
M <- get.par('M', p, progress = TRUE)$value
```

```
## Generate M from c("students", "concept.exp")
```

Now as method (`students`, `concept.exp`) is available, and clearly has a better input utilisation than (`students`, `skill.space`, `skill.dist`) (see **The generating criteria** section above), `edmsyn` opts for this method to generate `M`.

Adding type-1 connection

Adding `concept.exp` into `M$tell` is also a necessary step in recovering the deleted `concept.exp` node. In this task, `edmtree.add.tell` is used.

```r
# For the purpose of illustration, let's say knowing M only tells us about the number of concepts
edmtree.replace('M', tell = 'concepts', f.tell = function(M){
  return(list(nrow(M)))
})

# Now see that currently knowing M tells nothing about the expected mastery rate for each concept or th
p <- pars(M = M)
print(p)
```

```
## Activated information in the context:
## [1] "M"            "default.vals" "concepts"
```

```r
# It is time to add the full set of tell to M
edmtree.add.tell('M', c('concept.exp', 'students'), function(M){
  return(list( rowMeans(M), ncol(M) ))
})
p <- pars(M = M)
print(p)
```

```
## Activated information in the context:
## [1] "M"            "default.vals" "concepts"     "concept.exp"
## [5] "students"
```

```r
# Check if all calulations are correct
identical( dim(M), c(p$concepts, p$students) )
```

```
## [1] TRUE
```

```r
identical( p$concept.exp, rowMeans(M) )
```

```
## [1] TRUE
```

A toy model

Let's wrap up this section by going through the process of adding a whole new model into `edmsyn` structure. During this process, a few useful techniques that `edmsyn` provides will also be introduced, so it is beneficial to pay attention to the details in this section. Note that:

1. None of the details in this toy model makes sense, all of them is specifically designed for illustration purpose
2. This part covers only `edmtree.add`. `edmtree.replace` and `edmtree.remove` will not be used since we are not modifying any of the existing models and parameters.

Below is an outline of what is added to `edmsyn` by this new model (named `toy`):

- a root integer parameter named `foo`
- an integer parameter named `lower.foo` being the strict lower bound of `foo` (i.e. `foo` must be greater than `lower.foo`), `lower.foo` have the default value of 1.
- an integer parameter named `upper.foo` being the upper bound of `foo` (i.e. `foo` must be less than or equal to `upper.foo`), `upper.foo` have the default value equal to the sum of default value for `min.it.per.tree` and `concepts`.
- `bar`, a matrix with dimension (`foo`,`concepts`), its entries being real numbers between 0 and 1.
- data node of this model (named `toy`) is a list with two components: the first one is a matrix obtained from the rounded multiplication of `foo` and `M`, the second is the number of concepts

```r
# foo is a root node, with no default values
edmtree.add('foo', integer = TRUE)
```

```
## 'foo' appears to be a root node
```

```
## 'foo' appears to have no default initialization
```

```
edmtree.add('lower.foo', integer = TRUE,
            tell = 'foo', f.tell = less.strict,
            gen = 'default.vals', f.gen = 1)
```

```
## 'lower.foo' appears to have a constant default value
```

`less.strict` is a special function provided by edmsyn for cases when you want to tell the structure that `lower.foo` should be strictly less than `foo`. Alternatively, `edmtree.add.tell('foo', tell = 'lower.foo', f.tell = greater.strict)` gives the same effect. There are four such special functions recognised by `edmsyn`: `less.equal`, `less.strict`, `greater.equal`, and `greater.strict`.

The presence of these four functions highlighted the fact that inferring information in `edmsyn` is not solely inferring values, but can also be inferring different aspects of this value, namely the bound of them in this case.

```
# Now add upper.foo
edmtree.add('upper.foo', integer = TRUE,
            gen = c('default.vals', 'concepts'),
            f.gen = function(default.vals, concepts){
              return(default.vals$min.it.per.tree + concepts)
            })
```

```
## 'upper.foo' appears to be a root node
```

```
## 'upper.foo' appears to have a default value that relies on at least one run-time values
```

```
# Another use of special bound function
edmtree.add.tell('upper.foo', 'concepts', function(upper.foo){
  list(greater.equal(upper.foo - default()$min.it.per.tree))
})
```

```
## 'upper.foo' appears to have a default value that relies on at least one run-time values
```

```
# Instead of upper.foo telling the bound of foo,
# we will do it in the opposite direction,
# just for the purpose of illustration
edmtree.add.tell('foo', tell = 'upper.foo', f.tell = less.equal)
```

```
## 'foo' appears to have no default initialization
```

```
# Now since lower.foo and upper.foo are both presented in the structure
# it's time to add a generating method for foo
edmtree.add.gen('foo', gen.method = c('lower.foo', 'upper.foo'),
                f.gen.method = function(lower.foo, upper.foo){
                  sample((lower.foo+1) : upper.foo, 1)
                })
# add bar
edmtree.add('bar', gen = c('foo', 'concepts'),
            f.gen = function(foo, concepts){
              matrix(runif(foo * concepts), foo, concepts)
            })
```

```
## 'bar' appears to be a root node
```

```
# dimensions of bar are foo and concepts
edmtree.add.tell('bar', tell = c('foo', 'concepts'),
                 f.tell = function(bar){
                   list(nrow(bar), ncol(bar))
                 })
```

Note that it is okay not to add the `tell` component for `bar`, (in fact, it is okay to skip defining `tell` and `f.tell` in *every* parameter, your application will still run just fine as long as the rest is properly designed). However doing so will limit the capability of `edmsyn` to recognise conflicts (**condition 2**). For example, `pars(bar = matrix(0, 3, 5), concepts = 4)` will not raise the conflict between 4 and 5 if `bar$tell` does not include the inference for `concepts`. Adding `tell` and `f.tell` is a good practice if you want to add more debugging power to a big and complicated application.

```
# to make the model a little more sophisticated,
# we add another generating method for bar
edmtree.add.gen('bar', gen = c('M', 'foo'),
                f.gen = function(M, foo){
                  concepts = nrow(M)
                  matrix(runif(foo * concepts), foo, concepts)
                })
# finally, add the data node "toy"
edmtree.add('toy', data = TRUE,
            gen = c('bar','M'), f.gen = function(bar, M){
              list(R = round(bar %*% M), concepts = nrow(M))
            },
            tell = c('bar', 'M'), f.tell = function(toy){
              # Note that the following learning algorithm makes no sense
              # it is just for the purpose of illustration
              concepts = toy$concepts
              R = toy$R
              foo = nrow(R)
              students = ncol(R)
              bar = matrix(runif(foo * concepts), foo, concepts)
              M = matrix(sample(0:1, concepts * students, TRUE),
                         concepts, students)
              list(bar, M)
            })

# Check if ALL.MODELS includes "toy" (yes it does)
edmconst$ALL.MODELS
```

```
##  [1] "exp"     "irt"     "poks"    "dina"    "dino"    "lin.avg" "nmf.con"
##  [8] "nmf.dis" "nmf.com" "toy"
```

So, that is all there is to plug a new model into `edmsyn`. The process is simple and it forces users to think about various aspects while doing so. Another benefit is that before adding `toy`, a whole system of 62 parameters with carefully-built and well-tested connections is already there. This makes the work even lighter, we saved a lot of time before moving on testing our model.

Test the toy model

```
# 1. Test the bounds
p <- pars(lower.foo = 3, foo = 3)
```

```
## Error in down.stream(new.pars): 'foo' violates bound suggested by 'lower.foo'
```

```
p <- pars(foo = 4, upper.foo = 3)
```

```
## Error in down.stream(new.pars): 'upper.foo' violates bound suggested by 'foo'
```

```
# This one requires reasoning to detect
# Thus error is not raised immediately
p <- pars(lower.foo = 3, upper.foo = 2)
# But nevertheless, when p go into use, it immediately fails
get.par('foo', p)
```

```
## Error in down.stream(new.pars): 'upper.foo' violates bound suggested by 'foo'
```

```
p <- pars(upper.foo = 5, concepts = 5)
```

```
## Error in down.stream(new.pars): 'concepts' violates bound suggested by 'upper.foo'
```

```
# 2. Test foo$gen
get.par('foo', pars())
```

```
## Error in up.stream(target, pars, FALSE, progress): Cannot reach 'foo' since 'concepts' is missing
```

```
p <- pars(p, upper.foo = 15)
get.par('foo', p)
```

```
## $value
## [1] 15
##
## $context
## Activated information in the context:
## [1] "lower.foo"    "upper.foo"    "default.vals" "foo"
```

```
p <- pars(concepts = 5)
get.par('foo', p, progress = TRUE)
```

```
## Generate foo from c("lower.foo", "upper.foo")
## Generate lower.foo from default.vals
## Generate upper.foo from c("default.vals", "concepts")
```

```
## $value
## [1] 4
##
## $context
## Activated information in the context:
## [1] "concepts"     "default.vals" "lower.foo"    "upper.foo"
## [5] "foo"
```

```
p <- pars(M = M)
p <- get.par('foo', p, progress = TRUE)
```

```
## Generate foo from c("lower.foo", "upper.foo")
## Generate lower.foo from default.vals
## Generate upper.foo from c("default.vals", "concepts")
```

```
print(p)
```

```
## $value
## [1] 2
##
## $context
## Activated information in the context:
## [1] "M"            "default.vals" "concepts"     "concept.exp"
## [5] "students"     "lower.foo"    "upper.foo"    "foo"
```

```
# 3. Test bar
get.par('bar', pars())
```

```
## Error in up.stream(target, pars, FALSE, progress): Cannot reach 'bar' since 'concepts' is missing
```

```
get.par('bar', pars(upper.foo = 15))
```

```
## Error in up.stream(target, pars, FALSE, progress): Cannot reach 'bar' since 'concepts' is missing
```

```
get.par('bar', pars(lower.foo = 3, concepts = 5), progress = TRUE)
```

```
## Generate bar from c("foo", "concepts")
## Generate foo from c("lower.foo", "upper.foo")
## Generate upper.foo from c("default.vals", "concepts")

## $value
##           [,1]      [,2]      [,3]      [,4]       [,5]
## [1,] 0.2885557 0.8686231 0.1616676 0.4339905 0.32063277
## [2,] 0.8691603 0.7307924 0.8655714 0.7821060 0.12385539
## [3,] 0.7139102 0.8556506 0.7102737 0.4463187 0.05931105
## [4,] 0.7888999 0.3807506 0.4732732 0.3584197 0.23883422
## [5,] 0.1541311 0.9848869 0.7868750 0.3417807 0.49205371
##
## $context
## Activated information in the context:
## [1] "concepts"     "lower.foo"    "default.vals" "upper.foo"
## [5] "foo"          "bar"
```

```
get.par('bar', pars(M = M), progress = TRUE)
```

```
## Generate bar from c("foo", "concepts")
## Generate foo from c("lower.foo", "upper.foo")
## Generate lower.foo from default.vals
## Generate upper.foo from c("default.vals", "concepts")
```

```
## $value
##           [,1]      [,2]       [,3]
## [1,] 0.1623706 0.7142457 0.29877457
## [2,] 0.8783880 0.6893553 0.08443889
## [3,] 0.5151362 0.5749255 0.46373986
## [4,] 0.7401097 0.8300925 0.34725072
##
## $context
## Activated information in the context:
## [1] "M"            "default.vals" "concepts"     "concept.exp"
## [5] "students"     "lower.foo"    "upper.foo"    "foo"
## [9] "bar"
```

```
# 4. Test data
toys <- gen('toy', pars(M = M, bar = matrix(0, 3, 5)))
```

```
## Error in down.stream(new.pars): 'concepts' receives different values at once
```

```
toys <- gen('toy', pars(M = M, bar = matrix(1, 3, 3)),
            n = 2, progress = TRUE)
```

```
## Generate toy from c("bar", "M")
```

```
toys <- gen('toy', pars(students = 20, concepts = 4),
            n = 3, progress = TRUE)
```

```
## Generate toy from c("bar", "M")
## Generate bar from c("foo", "concepts")
## Generate foo from c("lower.foo", "upper.foo")
## Generate lower.foo from default.vals
## Generate upper.foo from c("default.vals", "concepts")
## Generate M from c("students", "concept.exp")
## Generate concept.exp from concepts
```

```
toys <- gen('toy', pars(M = M), n = 3, progress = TRUE)
```

```
## Generate toy from c("bar", "M")
## Generate bar from c("M", "foo")
## Generate foo from c("lower.foo", "upper.foo")
## Generate lower.foo from default.vals
## Generate upper.foo from c("default.vals", "concepts")
```

```
toys.syn <- syn('toy', toys[[2]]$toy,
                keep.pars = c("foo","concept.exp"),
                students = 12, n = 3, progress = TRUE)
```

```
## Learning by 'toy' ...
## Generate toy from c("bar", "M")
## Generate bar from c("foo", "concepts")
## Generate M from c("students", "concept.exp")
```

Working with the whole structure

Now that you have everything needed to fully manipulate the internal structure of `edmsyn`, it is time to move on working with the whole structure (as oppose to working with nodes and edges like before). The first thing to know is that, each time `library(edmsyn)` is executed, the original `edmsyn` graphical structure (without `foo`, `bar`, `toy`, etc) will be restored and everything we have built so far is lost. Thus, occasionally saving the modified structure is important.

```
toy.save <- edmtree.dump()
```

Assume a different situation where no part of the current structure is needed, you will manually build a whole new structure from scratch. This case, the first thing to do is to clear out all nodes

```
edmtree.clear()
edmtree.fetch('toy')
```

```
## Error in edmtree.fetch("toy"): 'toy' is not found in current tree
```

```
edmtree.fetch('M')
```

```
## Error in edmtree.fetch("M"): 'M' is not found in current tree
```

Lastly, you can always restore a saved structure

```
edmtree.load(toy.save)
gen('toy', pars(M = M))
```

```
## Activated information in the context:
##  [1] "M"            "default.vals" "concepts"     "concept.exp"
##  [5] "students"     "lower.foo"    "upper.foo"    "foo"
##  [9] "bar"          "toy"
```

```
gen('bkt', pars(students = 15, items = 20))
```

```
## Error in gen("bkt", pars(students = 15, items = 20)): Model 'bkt' is not available
```

```
# Leave the argument part empty
# if you want the original edmsyn structure
edmtree.load()
gen('toy', pars(M = M))
```

```
## Error in gen("toy", pars(M = M)): Model 'toy' is not available
```

```
gen('bkt', pars(students = 15, items = 20,
                concepts = 4, time = 10))
```

```
## Activated information in the context:
##  [1] "items"        "concepts"     "students"     "time"
##  [5] "default.vals" "per.item"     "S"            "L"
##  [9] "bkt.slip"     "bkt.guess"    "order"        "bkt"
```