# ⟮𝔐⟯FST: A Python OpenFst Wrapper with Support for Custom Semirings and Jupyter Notebooks

Matthew Francis-Landau
mfl@cs.jhu.edu

**Abstract**

This paper introduces ⟮𝔐⟯FST, a new Python library for working with Finite-State Machines based on OpenFst. ⟮𝔐⟯FST is a thin wrapper for OpenFst and exposes all of OpenFst's methods for manipulating FSTs. Additionally, ⟮𝔐⟯FST is the only Python wrapper for OpenFst that exposes OpenFst's ability to define a *custom* semirings. This makes ⟮𝔐⟯FST ideal for developing models that involve learning the weights on a FST or creating *neuralized* FSTs. ⟮𝔐⟯FST has been designed to be easy to get started with and has been previously used in homework assignments for a NLP class as well in projects for integrating FSTs and Neural Networks. In this paper, we exhibit ⟮𝔐⟯FST's API and how to use ⟮𝔐⟯FST to build a simple neuralized FST with PyTorch.

## 1 Finite-State Machine

A Finite-State Machine (FSM) represents a computation as a finite number of states labeled $[0, N)$ and transitions between those states. A FSM starts at an initial state—represented as green in this paper—and performs a series of transitions following the directed edges until it reaches a final state—represented as red in this paper. Transitions in a FSM can be labeled with symbols (e.g. a character in a string). In general, a Finite-State Transducer (FST) contains two symbols on each edge. One symbol represents what is *read* from the FST's input, and the other represents what is *written* to the output (figure 2). In the special case that both symbols are the same along all edges, a FST can also be called a Finite-State Acceptor (FSA) (figure 1). Furthermore, edges in a FST can also be weighted. This allows for a FST to score different paths that accept a given string. If an input can be mapped to multiple outputs, then each output will have a different weight assigned (figure 3).

In the remainder of this paper, we will assume the reader is familiar with the theory and algorithms behind FSTs. Instead, we will focus on ⟮𝔐⟯FST and how it makes working with FSTs from Python easy. The examples in this paper are intended to demonstrate how easy it is to get started constructing simple models rather than demonstrating "real" models that work with real scenarios.

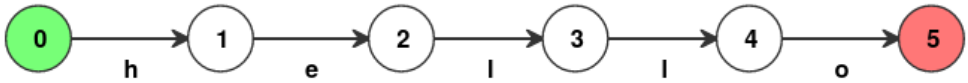



Figure 1: Finite-State Acceptor (FSA) for the string *"hello."* State #0, is colored green as it is the initial state of the FSA. State #5 is marked red as it is a terminal state. Strings that do not exactly match *"hello"* will not reach the final state (red), thus are not accepted.

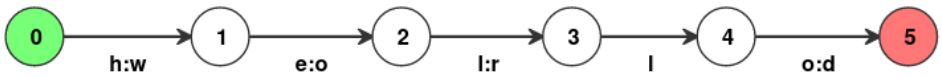



Figure 2: Finite-State Transducer (FST) that transforms the string *"hello"* → *"world."* During each transition, a character of the string *"hello"* is read from the input string—represented on the left side of the colon. On the right hand side of the colon, the output "world" is generated one character at a time as each arc is transversed. When the characters are the same on both the input and output side (e.g. between states $3 \rightarrow 4$), the colon is omitted for readability.

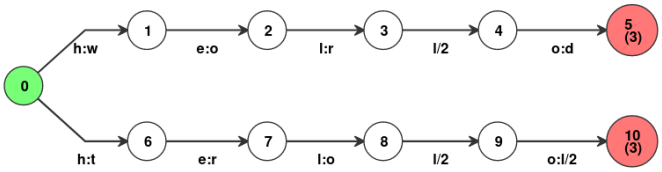



Figure 3: Weighted Finite-State Transducer where *"hello"* is transduced non-deterministically into both the words *"world"* and *"troll."* Each output word is weighted with the product of the weights along a given path. Here, *"hello"* is given the weight $1 * 1 * 1 * 2 * 1 * 3 = 6$, and *"troll"* is given $1 * 1 * 1 * 2 * 2 * 3 = 12$. The weight 1 is omitted from the figure for readability. If there exist multiple paths that generate the same output, the weight would be summed across the different paths.

# 2 ᨆFST: Getting Started

ᨆFST was initially developed for teaching advanced FST techniques in a classroom setting [4]. This has motivated ᨆFST's simple interface, which makes it quick to get started. ᨆFST includes sensible defaults all accessible from Python, while not eliminating power user features of OpenFst.

## 2.1 One Command Install

ᨆFST is installable with a single command. ᨆFST includes and automatically compiles OpenFst, requiring no additional steps from a user[1].

```
pip install 'git+https://github.com/matthewfl/openfst-wrapper.git'
```

## 2.2 First Simple FSAs

Once ᨆFST is installed, it makes the first steps of interacting with a FSTs from Python easy. Here, a FSA is constructed for the string *"hello."* The FSA is constructed using ᨆFST's default semiring—more on that later in section 4.

```
In [1]: from mfst import FST

        FST('hello')
```

Out[1]:

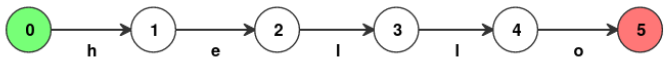


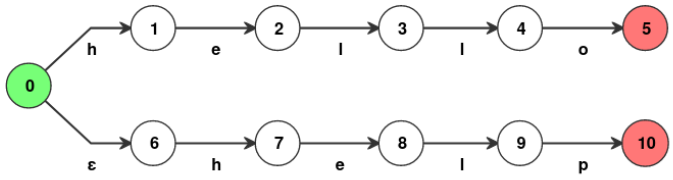

Figure 4: The simplest way to construct a FSA for the string *"hello."* When a FST is returned from a Jupyter notebook cell, it is drawn into the notebook. All figures in this paper are from ᨆFST's Jupyter drawings. Here we are exemplifying how any Python iterable (such as a string) is converted into the labels on a FSA.

ᨆFST exposes all of OpenFst's operations[2] for interacting and manipulating a FST. Whenever an instance of the `FST` class is returned from a cell in a Jyputer notebook, the FST is automatically drawn into the notebook. This makes ᨆFST ideal for learning how FSTs work, visualizing how the different operations transform a FST without requiring any additional steps to draw the FST (shown in figure 5). ᨆFST, being a Python package, includes documentation on all of its methods (adapted from OpenFst's documentation). This documentation can be easily accessed in an interactive Python environment using Python's `help()` function (figure 6).

[1]Note: The install command takes around 10 minutes to run without printing any indication it is running to the terminal.

[2]OpenFst included operations: `http://www.openfst.org/twiki/bin/view/FST/FstQuickTour#Available%20FST%20Operations`.

```
In [2]: FST('hello').union(FST('help'))
Out[2]:
```




```
In [3]: _.remove_epsilon().determinize()
Out[3]:
```

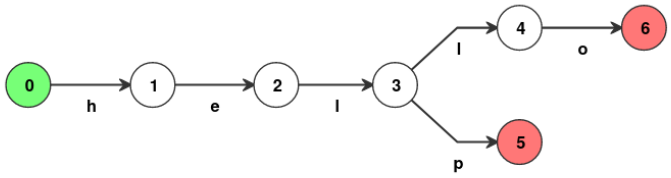


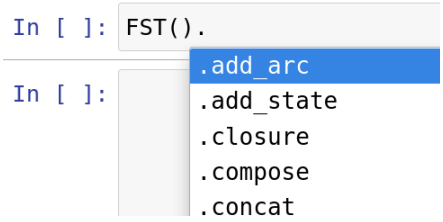

Figure 5: Simple FSA constructed by taking the union of the FSA for the string *"hello"* and *"help."* In cell #3, the FSA is referenced using Python's underscore variable (_) to reference the output of the previous computation. The operations `remove_epsilon` and `determinize` are chained together here to create a FSA where the prefixes *"hel"* is merged.

```
In [6]: help(FST().determinize)
Help on method determinize in module mfst:

determinize(delta=0.0009765625, weight_threshold=None, *, allow_
hod of mfst.FST instance
    This operation determinizes a weighted transducer. The resul
    equivalent FST that has the property that no state has two t
    with the same input label. For this algorithm, epsilon trans
    treated as regular symbols (cf. RmEpsilon).

    http://www.openfst.org/twiki/bin/view/FST/DeterminizeDoc
```

(a) Using Python's `help()` function to access documentation for a method. The documentation is largely adapted from OpenFst online documentation[2].




(b) Tab competition in a Jupyter notebook to list the available methods on the FST class.

Figure 6: The documentation on ᗰFST methods is included as Python docstrings and can also be easily auto-completed using any Python IDE.

## 3 Building Transducers

FST can also be built using the `add_state` and `add_arc` methods in addition to the simplified constructor we have demonstated so far. These methods makes it possible to build more complicated FSTs which may have loops or may define input and output labels and weights on all of the edges. The main methods for constructing a FST are as follows and in figure 7.

- `add_state()`→`state_id`: Modifies the FST in place to add a new state.
- `add_arc(from_state, to_state, weight, input_label, output_label)`→`unit`: Modifies the FST in place to add a new arc between two states. If `from_state` and `to_state` are the same, then this will create a self-loop. The `weight` is cast to the FST's semiring automatically. By default `weight` is set to the semiring's identity element of $\bar{1}$. `input_label` and `output_label` are 64 bit integers. The value 0 is used to represent the special $\epsilon$ character in the FST, which indicates that no symbol is read/written when traversing an arc. If a single character of a string is used as a label, then it is automatically converted into a integer using its character code (via the built-in function `ord` in Python).
- `set_initial_state(state_id)`→`unit`: Modifies the FST to set its initial state. A FST can only have a single initial state, and a FST without an initial state will not be usable.
- `set_final_weight(state_id, weight)`→`unit`: Modifies the FST to set the final state weight. There can be multiple final states, and a well defined FST should have at least one final state.

```
In [4]: transducer = FST()
        s = transducer.add_state()
        transducer.set_initial_state(s)
        for letter_from, letter_to in zip('hello', 'world'):
            next_state = transducer.add_state()
            transducer.add_arc(from_state=s,
                               to_state=next_state,
                               weight=1,
                               input_label=letter_from,
                               output_label=letter_to)
            s = next_state
        transducer.set_final_weight(s, weight=1)
        transducer
```

Out[4]:


Figure 7: In Jupyter cell #4, an empty FST is first constructed using `FST()` with the default semiring. The edges are added one at a time using `add_arc`, where most edges have different labels on the input and output side.

## 4 Weighted FSTs

Weighted FSTs allow for scoring a given sequence rather than just accepting or transducing an input. This is done by assigning a weight to every edge in the FST. The weights on a FST must all be instances of the same semiring, meaning that they share a multiplication and addition operator as well as a multiplicative and additive identity $\langle +, \times, \bar{0}, \bar{1} \rangle$. To determine the weight for a given input/output sequence, the weights along a given path are multiplied together. When there are multiple paths that accept an input or transduce a given input/output pair, the weight will be summed

```
In [5]: Out[3].compose(Out[4]).project('output')
```

Out[5]:

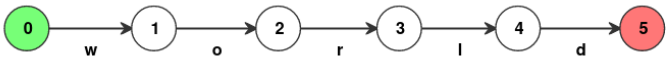



Figure 8: In cell #5, the FST is composed with the FSA from figure 5 that was the union of the word *"hello"* and *"help."* Only the word *"hello"* is transduced as there is no path that accepts the word *"help."* The `project('output')` method turns the FST into a FSA using the labels on the output side.

across all paths. Changing the semiring and defining different actions for multiplication and addition can allow the same algorithm operating on the FST to compute different results—such as in figure 11.

MFST includes the standard semirings. This includes: Python values, real numbers, min, max, tropical, and boolean. Each of these semirings is implemented in approximately 20 lines of Python each. Custom semirings can be defined by extending the `AbstractSemiringWeight` class.

MFST's default semiring—that we have been using so far—is the boolean semiring. In the boolean semiring, all weights along the edges are $\bar{1}$ (as the $\bar{0}$ weights are omitted). The boolean semiring is given special precedence in MFST as the boolean semiring will be automatically cast to another semiring as necessary. All operations in MFST which involve two or more FSTs—such as compose or union—require that all of its arguments implement the same semiring. The boolean semiring's automatic casting is useful as it allows us to construct FSTs and compose them with weighted FSTs without too much concern for which semiring is in use, such as in figure 10. MFST also provides the method `lift(semiring_class, cast_function)→FST` to convert a FST from one semiring to another, such as in figure 11.

```
In [21]: from mfst import RealSemiringWeight
         fst = FST(RealSemiringWeight)
         s = fst.add_state()
         read_a = fst.add_state()
         fst.set_initial_state(s)
         fst.set_final_weight(s, weight=1)
         fst.add_arc(from_state=s,to_state=s,
             input_label='a', output_label='a', weight=1)
         fst.add_arc(from_state=s,to_state=read_a,
             # 0 represents epsilon (nothing) read or written
             input_label='a', output_label=0, weight=1)
         fst.add_arc(from_state=read_a,to_state=s,
             input_label='a', output_label='b', weight=.5)
         fst.add_arc(from_state=s,to_state=s,
             input_label='b', output_label='b', weight=1)

         fst
```

Out[21]:

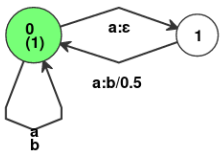



Figure 9: Here we define a weighted FST using the `RealSemiringWeight` $\langle +, \times \rangle$. This FST transduces between the alphabet $\{a, b\}$ and itself. Both symbols can be read unchanged (as they transverse the edge from state $0 \to 0$, which has weight 1). If the FST encounters two *"a"*s in a row, then it is transduced non-deterministically into the letter *"b"* with a weight of 0.5. The weight 0.5 comes from the fact that it multiplies the weights along the edges from state $0 \to 1$ and $1 \to 0$ when reading the sequence *"aa."*

```
In [22]: FST('aaa').compose(fst)
```

Out[22]:

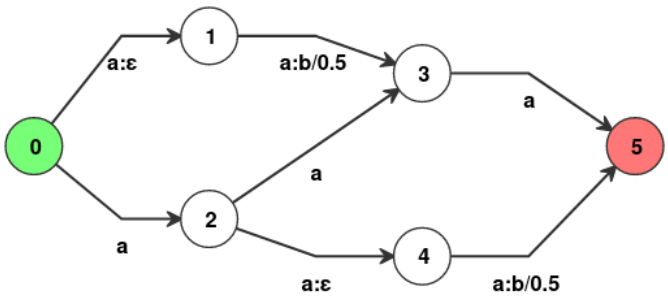
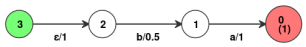



Figure 10: Here the FST defined in figure 9 is composed on its input side with the FSA which represents *"aaa"* (constructed in the same way as figure 4). The `FST('aaa')` is using the default semiring, so it is automatically cast to the `RealSemiringWeight` which was used to define `fst`. This FST has three different paths which can generate outputs *"aaa," "ba,"* or *"ab."* The weight associated with each path is the result of multiplying the weights from both FSTs which are composed together. The `FST('aaa')` only has a single path and the FST from figure 9 non-deterministically maps this and scores the transduction *"aa"*→*"b"* with weight 0.5.

```
In [30]: from mfst import MinPlusSemiringWeight
         FST('aaa').compose(fst).project('output') \
           .lift(MinPlusSemiringWeight, lambda w: w.value) \
           .shortest_path()
```

Out[30]:




```
In [31]: from mfst import MaxPlusSemiringWeight
         FST('aaa').compose(fst).project('output') \
           .lift(MaxPlusSemiringWeight, lambda w: w.value) \
           .shortest_path()
```

Out[31]:

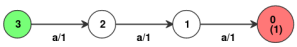



Figure 11: The semiring is cast to the min-plus and max-plus semirings and then we use the shortest path algorithm to identify the shortest path. In these semirings, the multiplication ($\odot$) operator is summing the length of the path and the addition ($\oplus$) operator is defined as min/max. This allows us to find the shortest path using the "$\leq$" operator where it is defined as $a \leq b \iff a \oplus b = a$ for idempotent semirings.

## 4.1 Custom Weighted Semirings

All semirings in ⟑FST are defined in pure Python by extending the `AbstractSemiringWeight` class. This means that we can easily define a custom semiring that makes use of other libraries in Python. For example, we may be interested in learning the weights on a FST. One way in which this can be done is to create a featurized semiring, where we are tracking the weights associated with every feature rather than the weight of an edge itself (appendix B).

Another approach that we can use for learning the weights on a FST is to leverage a dynamic graph neural network frameworks, such as PyTorch. To construct a graph using standard neural network forward propagation, we simply use PyTorch's operations whenever we perform a semiring operation. In figure 12 we construct a custom semiring that wraps a PyTorch tensors as a weight. We can implement the `__add__` and `__mul__` methods with anything which results in a semiring. For example, if we were to implement the min semiring, then we would use `torch.min` for the `__add__` method and use `torch.add` for the `__mul__` method. For this example, however, we have instead chosen to exemplify how we could perform any neural operation. Here we are using a forward pass through a linear layer followed by a sigmoid operation. Note, the "semiring" which we define in figure 12 is not a *real* semiring as it does not follow the required distributivity and identity elements of a semiring[3]. While ⟑FST will still run, the outputs of an invalid "semiring" is unlikely to be stable or work in practice.

[3] `https://en.wikipedia.org/wiki/Semiring#Definition`

# 5 Implementation

ᗰFST is a thin wrapper around OpenFst's C++ interface. OpenFst [2] has the ability to define custom semiring by defining a custom C++ semiring class[4]. ᗰFST defines a custom semiring which wraps an opaque Python object. The Python C++ bridge makes use of PyBind11 [5], which makes passing around Python objects in C++, and exposing C++ methods to Python largely transparent. Anytime an operation is performed by OpenFst on the semiring (such as addition or multiplication), the call is redirected to the relevant method defined in Python. This means that we can easily define our own semiring from Python by implementing the relevant methods. As such, all FST operations are handled by OpenFst, hence matured and well-tuned over the years.

The images that are drawn by ᗰFST make use of Jyputer's `_repr_html_` method[5]. By defining a `_repr_html_` method, this allows a class to define how it is rendered in a Jupyter notebook by returning a string of HTML code that is used instead of the standard textual output. ᗰFST makes use of this method to generate HTML code that utilizes D3.js [3] and Dagre-d3 [1] to render the diagrams in the browser.

ᗰFST's install procedure is designed to be as easy as possible so that it can be easily accomplished by students in a class without having to follow a multi-step installation procedure. ᗰFST is installable using Python's pip with the command: `pip install 'git+https://github.com/matthewfl/openfst-wrapper.git'`. When passing pip a path to a git repository, it first git clone the repository followed by the standard package installation command: `python setup.py install`. ᗰFST makes use of this to be able to run the standard OpenFst install from source commands. In this processes, ᗰFST sets the `--prefix` argument on `./configure` command to Python's `sys.prefix` variable. This is set to the current Python interpreter's install location. When pip is run from a virtual environment[6] or anaconda, `sys.prefix` is set to the user's local environment. This is usually in the user's home directory, hence writable. This means that ᗰFST is able to install OpenFst on a system even when a user does not have root permission, without requiring a user to manage installation paths themselves.

# References


[1] Dagre-d3 - a D3-based renderer for dagre. `https://github.com/dagrejs/dagre-d3`.

[2] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. OpenFst: A general and efficient weighted finite-state transducer library. In *Proceedings of the 12th International Conference on Implementation and Application of Automata*, CIAA'07, page 11–23, Berlin, Heidelberg, 2007. Springer-Verlag.

[3] Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. D3 data-driven documents. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2301–2309, December 2011.

[4] Matthew Francis-Landau and Jason Eisner. Johns Hopkins University class EN.601.765, Seq2class assignment 2: Finte-State Machines, 2018. `https://github.com/seq2class/assignment2/blob/master/homework_2.ipynb`.

[5] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. PyBind11 — Seamless operability between C++11 and Python, 2016. `https://github.com/pybind/pybind11`.


[4] `http://www.openfst.org/twiki/bin/view/FST/FstAdvancedUsage#Weights`
[5] `https://iPython.readthedocs.io/en/stable/config/integrating.html#rich-display`
[6] `https://docs.python.org/3/library/venv.html`

In [7]:
```
import torch
from mfst import AbstractSemiringWeight

add_layer = torch.nn.Linear(20, 10)
mul_layer = torch.nn.Linear(20, 10)

# Note: not a "real" semiring
class TorchSemiring(AbstractSemiringWeight):
    def __init__(self, value):
        self.value = value
    def __add__(self, other):
        inp = torch.cat((self.value, other.value))
        out = torch.sigmoid(add_layer(inp))  # pytorch forward
        return TorchSemiring(out)
    def __mul__(self, other):
        inp = torch.cat((self.value, other.value))
        out = torch.sigmoid(mul_layer(inp))  # pytorch forward
        return TorchSemiring(out)
    def __str__(self): return f'TW({self.value[0]:.2f})'
    def __eq__(s, o): return all(s.value == o.value)
    def __hash__(self): return hash(self.value)
    def approx_eq(s, o, delta):
        return sum(abs(s.value - o.value)) < delta


TorchSemiring.zero = \
   TorchSemiring(torch.rand(10, requires_grad=True))
TorchSemiring.one = \
   TorchSemiring(torch.rand(10, requires_grad=True))

[weight1, weight2, weight3, weight4] = \
   [TorchSemiring(torch.rand(10, requires_grad=True))
    for _ in range(4)]

torch_fst = FST(TorchSemiring)
[a,b,c] = [torch_fst.add_state() for _ in range(3)]
torch_fst.set_initial_state(a)
torch_fst.add_arc(from_state=a,to_state=b,weight=weight1,
                  input_label='a', output_label='b')
torch_fst.add_arc(from_state=b,to_state=c,weight=weight2,
                  input_label='c', output_label='d')
torch_fst.add_arc(from_state=a,to_state=c,weight=weight3,
                  input_label='e', output_label='f')
torch_fst.set_final_weight(c, weight=weight4)
torch_fst
```

Out[7]:

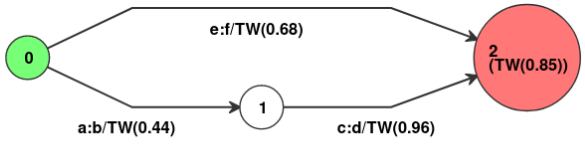
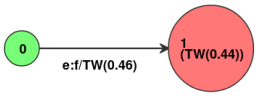



Figure 12: A custom semiring where weights are 10-element PyTorch tensors. The multiplication and addition operations of the semiring are calling PyTorch Neural Network operators. Note: This example is intended only to illustate that PyTorch tensors can be passed through OpenFst as weights. The "semiring" defined here is not, in fact, a "mathematically correct" semiring, and so actually using this as written here is ill-advised.

```
In [8]: FST('e').compose(torch_fst)
Out[8]:
```




(a) Composition with the default BooleanSemiring automatically casts to the identity elements of the TorchSemiring and then composes the two FSTs.

```
In [9]: _.sum_paths()
Out[9]: TorchSemiring(TW(0.42))
```

(b) `sum_paths` sums all of the paths in the FST and returns the resulting weight element in the semiring.

Figure 13: `torch_fst` manipulated through ⟲FST's built-in operations.

```
In [10]: # get the resulting pytorch tensor from the FST
         numerator = FST('e').compose(torch_fst).sum_paths().value
         denominator = torch_fst.sum_paths().value

         prob = torch.exp(torch.sum(numerator)) \
              / torch.exp(torch.sum(denominator))
         prob.backward()  # pytorch auto diff
         ## Here I would run the optimizer, IF I HAD ONE
         # optimizer.step()
         prob
Out[10]: tensor(0.9202, grad_fn=<DivBackward0>)
```

Figure 14: Here we exemplify how the `torch_fst` could be used with PyTorch's backpropagation. First, the numerator for the path that we are interested in is selected using `compose` and then `sum_paths` to get the value for those specific paths. We then get the normalizing constant (denominator) for all of the paths in the FST. The variable `numerator` and `denominator` are both PyTorch tensors. As a result, we can use PyTorch methods such as `torch.sum` and `torch.exp` on these variables. As with a standard PyTorch tensor, we can use `tensor.backwards()` to compute the gradient. In this example, we have not included an optimizer, though that would be included in general. Again, as mentioned above, this example is intended to illustrate how PyTorch tensors can be passed around through a FST as a semiring weight, not to illustrate something that is expected to work well.

```
In [23]: list(FST('aaa').compose(fst).iterate_paths())
Out[23]: [Path(input_path='aaa', output_path='ba', weight=RealSemiringWeight(0.5)),
          Path(input_path='aaa', output_path='ab', weight=RealSemiringWeight(0.5)),
          Path(input_path='aaa', output_path='aaa', weight=RealSemiringWeight(1))]
```

Figure 15: Using ⟲FST's `iterate_path` method to see the different path from figure 10. Every path in the FST has a sequence associated on its input side and output side (as shown above) and it associated weight. The input paths are all *"aaa"* as we composed `FST('aaa')` on the input side. If we were to use this method on a FST with cycles (such as figure 9) then it would make an infinite iterator over increasingly longer paths in the FST.

# A Abstract Semiring Class

The AbstractSemiringWeight can be found in full detail here: `https://git.io/JIWKn`. Here is a summary of the Abstract Semiring base class.

```
class AbstractSemiringWeight:
  semiring_properties : {'base', 'path'}          # if an idempotent(path) semiring
  def __add__(self, other): ...                   # semiring + operation (required)
  def __mul__(self, other): ...                   # semiring * operation (required)
  def __eq__(self, other): ...                    # if equal             (required)
  def __hash__(self): ...                         # hash code            (required)
  def __str__(self): ...                          # for displaying in figures
  def approx_eq(self, other, delta=1.0/1024): ... # approximately equal (suggested)

  def __div__(self, other): ...               # used by FST().push()
  def __pow__(self, n): ...
  def quantize(self, delta=1.0/1024): ...     # weight quantization
  def member(self): ...                       # if element of semiring (e.g. not nan)
  def reverse(self): ...                      # used by FST().reverse()
  def sampling_weight(self): ...              # used by FST().random_path()

AbstractSemiringWeight.zero = AbstractSemiringWeight(...)  # static 0 and 1 weights
AbstractSemiringWeight.one  = AbstractSemiringWeight(...)  # (required)
```

# B Basic Featurized Semiring

This exhibits how Python's Counter dictionary could be used to track weights for a feature. Here the counts of a feature are summed along a path, the max value for a given feature is selected if there are multiple paths.

```
from mfst import AbstractSemiringWeight
from collections import Counter
feature_weight = {}  # weights stored in a global dict
class FeaturizedWeight(AbstractSemiringWeight):
  def __init__(self, features):
    self._features = Counter(features)
    self._hash = hash(frozenset(self._features))
  def __add__(self, other):
    # take the max count for a given feature across different paths
    return FeaturizedWeight(self._features | other._features)
  def __mul__(self, other):
    return FeaturizedWeight(self._features + other._features)
  def __str__(self):
    return str(self._features)
  def __eq__(self, other):
    return self._hash == other._hash and self._features == other._features
  def __hash__(self):
    return self._hash
  def approx_eq(self, other, delta=1.0/1024):
    return sum(abs(self._features - other._features)) < delta
  def sampling_weight(self):
    result = 0
    for key, val in self._features.items():
      result += feature_weight.get(key, 0) * val
    return result
```