Documentation for Storage_tree.py

Yili Yang

June 2017

1 Introduction

Storage_tree.py is a file containing two classes SmallStorgeTree and BigStorageTree of storage tree for the DLW model.The storage trees are mainly used as dictionary storing various information for each node within a tree object. For example, the class mainly provides a dict with key of period times (i.e. [0,15,45,85,100]) and item of information (i.e mitigation level on each node).

The main difference between storage tree object and tree object is that the storage tree doesn't have index for nodes and states, which makes each periods more 'independent' of each other. In spite of looking the nodes as an continues array, the storage tree take nodes as an attribute of a certain period and the uniqueness of a node is always got by the period and the position of a node within a certain period and consequently you can not find path or reachable nodes using this class. . All the information in this kind of tree is specific and it is merely for storage usage.

2 Python:Storage_tree.py

2.1 Base Class

Base Class is an abstract storage class for the EZ-Climate model.

2.1.1 Inputs and Outputs

Inputs:

- **decision_times**:(ndarray or list) array of years from start where decisions about mitigation levels are done

Outputs:
It doesn't have outputs since it's a abstract class.

2.1.2 Attributes

- **decision_times**:(ndarray) array of years from start where decisions about mitigation levels are done. For example, [0, 15, 45, 85, 185, 285, 385].
- **information_times**:(ndarray) array of years where new information is given to the agent in the model. For example, if the decision times is above,then information times will be [0, 15, 45, 85, 185].
- **periods**: (ndarray) periods in the tree. (Different from SmallStorageTree and BigStorageTree, will be explain later in the sub class)
- **tree**: (dict) dictionary where keys are 'periods' and values are nodes in period.

2.1.3 Methods

The basic components of this class is a init with decision times. Also, it introduces a new concept: **information_times**, which is an array of years where new information is given to the agent in the model. In the base model, the information time is the periods of tree excluding the final state since we get the full knowledge on the T-1 state.
Also, the class has a __getitem__ enableing using it as a dict (the main usage I mentioned in the introduction) and a __len__ get its size easily.

```
def __init__(self, decision_times):
        self.decision_times = decision_times
        if isinstance(decision_times, list):
                self.decision_times = np.array(decision_times)
        self.information_times = self.decision_times[:-2] # exclude the final p
        self.periods = None
        self.tree = None

def __len__(self):
        return len(self.tree)

def __getitem__(self, key):
        if isinstance(key, int) or isinstance(key, float):
                return self.tree.__getitem__(key).copy()
        else:
                raise TypeError('Index must be int, not {}'.format(type(key).__n
```

_init_tree: The most important method is this method which gives the class a main dictionary to work with. It is a dictionary with key of periods and items of zero arrays with the right size. (binomial sense)

```python
def _init_tree(self):
        self.tree = dict.fromkeys(self.periods)
        i = 0
        for key in self.periods:
                self.tree[key] = np.zeros(2**i)
                if key in self.information_times:
                        i += 1
```

some frequently used properties of the tree model including:

- last period's array
- last period. (i.e. the last item of the decision time array)
- number of nodes in the tree

```python
@property
def last(self):
        """ndarray: last period's array."""
        return self.tree[self.decision_times[-1]]

@property
def last_period(self):
        """int: index of last period."""
        return self.decision_times[-1]

@property
def nodes(self):
        """int: number of nodes in the tree."""
        n = 0
        for array in self.tree.values():
                n += len(array)
        return n
```

Abstract method for sub-class usage.

```python
@abstractmethod
def get_next_period_array(self, period):
        """Return the array of the next period from `periods`."""
        pass
```

set_value : set any kind of value for all the node with a period using the given value.

```python
def set_value(self, period, values):
        """If period is in periods, set the value of element to `values` (ndar
        if period not in self.periods:
                raise ValueError("Not a valid period")
        if isinstance(values, list):
                values = np.array(values)
```

```
        if self.tree[period].shape != values.shape:
                raise ValueError("shapes {} and {} not aligned".format(self.tree
        self.tree[period] = values
```

boolean check method to check whether a period is :

- a decision time. Continue with the above example: this check whether the period is in [0, 15, 45, 85, 185, 285, 385]
- a decision time besides the last period.This check whether the period is in [0, 15, 45, 85, 185, 285]
- a information time. This check whether the period is in [0, 15, 45, 85, 185]

```
def is_decision_period(self, time_period):
        """Checks if time_period is a decision time for mitigation, where
        time_period is the number of years since start.

        Parameters
        ----------
        time_period : int
                time since the start year of the model

        Returns
        -------
        bool
                True if time_period also is a decision time, else False

        """
        return time_period in self.decision_times

def is_real_decision_period(self, time_period):
        """Checks if time_period is a decision time besides the last period, wh
        time_period is the number of years since start.

        Parameters
        ----------
        time_period : int
                time since the start year of the model

        Returns
        -------
        bool
                True if time_period also is a real decision time, else False

        """
        return time_period in self.decision_times[:-1]
```

```python
def is_information_period(self, time_period):
        """Checks if time_period is a information time for fragility, where
        time_period is the number of years since start.

        Parameters
        ----------
        time_period : int
                time since the start year of the model

        Returns
        -------
        bool
                True if time_period also is an information time, else False

        """
        return time_period in self.information_times
```

write_tree: A standard save method for storage trees. It save the tree's info in a row but never been use in the following code.

```python
def write_tree(self, file_name, header, delimiter=";"):
        """Save values in `tree` as a tree into file  `file_name` in the
        'data' directory in the current working directory. If there is no 'dat
        directory, one is created.

        Parameters
        ----------
        file_name : str
                name of saved file
        header : str
                first row of file
        delimiter : str, optional
                delimiter in file

        """
        from tools import find_path
        import csv

        real_times = self.decision_times[:-1]
        size = len(self.tree[real_times[-1]])
        output_lst = []
        prev_k = size

        for t in real_times:
                temp_lst = [""]*(size*2)
```

```
                k = int(size/len(self.tree[t]))
                temp_lst[k::prev_k] = self.tree[t].tolist()
                output_lst.append(temp_lst)
                prev_k = k

        write_lst = zip(*output_lst)
        d = find_path(file_name)
        with open(d, 'wb') as f:
                writer = csv.writer(f, delimiter=delimiter)
                writer.writerow([header])
                for row in write_lst:
                        writer.writerow(row)
```

write_columns: A standard save method for storage trees. It save the tree's info in a csv with the following template.

Year	Node	header
start_year	0	value0
...	...	...

where value0 is a abstract numeber, for example, it can be utility, mitigation level, consumption and etc given what are you storing. Also, the next method **write_columns_existing** save the trees info in a modified format. This kind of format is trivial and convenient to be used directed in csv.

Year	Node	other_header	header
start_year	0	other_value	value0
...	...	...	...

Where the other_value is another thing you want to store such as mitigation, utility that is different from value0.

```
def write_columns(self, file_name, header, start_year=2015, delimiter=";"):
        """Save values in `tree` as columns into file  `file_name` in the
        'data' directory in the current working directory. If there is no 'dat
        directory, one is created.

        Parameters
        ----------
        file_name : str
                name of saved file
        header : str
                description of values in tree
        start_year : int, optional
                start year of analysis
        delimiter : str, optional
                delimiter in file
```

```
        """
        from tools import write_columns_csv, file_exists
        if file_exists(file_name):
                self.write_columns_existing(file_name, header)
        else:
                real_times = self.decision_times[:-1]
                years = []
                nodes = []
                output_lst = []
                k = 0
                for t in real_times:
                        for n in range(len(self.tree[t])):
                                years.append(t+start_year)
                                nodes.append(k)
                                output_lst.append(self.tree[t][n])
                                k += 1
                write_columns_csv(lst=[output_lst], file_name=file_name, header=
                                                  index=[years, nodes], delimite

def write_columns_existing(self, file_name, header, delimiter=";"):
        """Save values in `tree` as columns into file  `file_name` in the
        'data' directory in the current working directory, when `file_name` al
        If there is no 'data' directory, one is created.

        Parameters
        ----------
        file_name : str
                name of saved file
        header : str
                description of values in tree
        start_year : int, optional
                start year of analysis
        delimiter : str, optional
                delimiter in file

        """
        from tools import write_columns_to_existing
        output_lst = []
        for t in self.decision_times[:-1]:
                output_lst.extend(self.tree[t])
        write_columns_to_existing(lst=output_lst, file_name=file_name, header=he
```

2.2 Small Storage Tree

Sub class of BaseStorageTree. In this class, the decision times are the only time that we are care about.

2.2.1 Inputs, Outputs and Attributes

The Inputs, Output and Attributes are the same as the BaseStorageTree.
In **Attributes**: **period**: here, the periods are the same with the decision times.

2.2.2 Methods

get_next_period_array: return a array consists of the stored information in the next period. A example of this method:

```
>>> sst = SmallStorageTree([0, 15, 45, 85, 185, 285, 385])
>>> sst.get_next_period_array(0)
array([0., 0.])
>>> sst.get_next_period_array(15)
array([ 0.,  0.,  0.,  0.])
```

```
def get_next_period_array(self, period):
        """Returns the array of the next decision period.

        Parameters
        ----------
        period : int
                period

        Raises
        ------
        IndexError
                If `period` is not in real decision times

        """
        if self.is_real_decision_period(period):
                index = self.decision_times[np.where(self.decision_times==period
                return self.tree[index].copy()
        raise IndexError("Given period is not in real decision times")
```

index_below: returns the key (a decision time) of the previous decision period. An example of this:

```
>>> sst = SmallStorageTree([0, 15, 45, 85, 185, 285, 385])
>>> sst.index_below(15)
```

```
            0

    def index_below(self, period):
            """Returns the key of the previous decision period.

            Parameters
            ----------
            period : int
                    period

            Raises
            ------
            IndexError
                    If `period` is not in decision times or first element in decis

            """
            if period in self.decision_times[1:]:
                    period = self.decision_times[np.where(self.decision_times==perio
                    return period[0]
            raise IndexError("Period not in decision times or first period")
```

2.3 Big Storage Tree

Sub Class of BaseStorageTree. This tree store all the information on every possible interval period.

2.3.1 Inputs and Outputs

- **subintervals_len** : (float) periods in tree. For example, if it is 5, then every number which is divisible by 5 is a period and the final period is the last number in the dicision time.
- **decision_times** : (ndarray or list) array of years from start where decisions about mitigation levels are done (time when one state become two: up or down)

2.3.2 Attributes

- **decision_times**:(ndarray) array of years from start where decisions about mitigation levels are done.
- **information_times**:(ndarray) array of years where new information is given to the agent in the model.
- **periods**: (ndarray) periods in the tree.

- **tree**: (dict) dictionary where keys are 'periods' and values are nodes in period.
- **subintervals_len** : (float) years between periods in tree.

2.3.3 Example

```
>>> bst = BigStorageTree(5.0, [0, 15, 45, 85, 100])
>>> bst.tree
{0.0: array([ 0.]),
 5.0: array([ 0.,  0.]),
 10.0: array([ 0.,  0.]),
 15.0: array([ 0.,  0.]),
 20.0: array([ 0.,  0.,  0.,  0.]),
 25.0: array([ 0.,  0.,  0.,  0.]),
 30.0: array([ 0.,  0.,  0.,  0.]),
 35.0: array([ 0.,  0.,  0.,  0.]),
 40.0: array([ 0.,  0.,  0.,  0.]),
 45.0: array([ 0.,  0.,  0.,  0.]),
 50.0: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]),
 55.0: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]),
 60.0: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]),
 65.0: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]),
 70.0: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]),
 75.0: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]),
 80.0: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]),
 85.0: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]),
 90.0: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]),
 95.0: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.]),
 100.0: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])}
```

Here, the length of one period is **subintervals_len** which is 5. And the array $[0, 15, 45, 85, 100]$ is the time that we can make a new mitigation decision. Only at the decision time, we will know what damage we have done and decide the new mitigation level. And then, each situation is split to two (up or down). And the zeros in the output is creating the space for information storage. Each zero can be replaced by the utility, consumption, certainty equivalence and etc. of this node.
While for small trees, periods will only be [0, 15, 45, 85, 100] since there is no inter-periods and the decision times is a new period.

2.3.4 Methods

first_period_intervals: return the number of subintervals in the first period. For example:

```
>>> bst.first_period_intervals()
3
```

```python
    @property
    def first_period_intervals(self):
        """ndarray: the number of subintervals in the first period."""
        return int((self.decision_times[1] - self.decision_times[0]) / self.subi
```

get_next_period_array: The same as previously described for small storage tree.

```python
    def get_next_period_array(self, period):
        """Returns the array of the next period.

        Parameters
        ----------
        period : int
            period

        Examples
        --------
        >>> bst = BigStorageTree(5.0, [0, 15, 45, 85, 185, 285, 385])
        >>>bst.get_next_period_array(0)
        array([0., 0.])
        >>> bst.get_next_period_array(10)
        array([ 0.,  0., 0., 0.])

        Raises
        ------
        IndexError
            If `period` is not a valid period or too large

        """
        if period + self.subinterval_len <= self.decision_times[-1]:
            return self.tree[period+self.subinterval_len].copy()
        raise IndexError("Period is not a valid period or too large")
```

between_decision_times: Check which decision time the period is between and returns the index of the lower decision time. An example for this is:

```python
>>> bst = BigStorageTree(5, [0, 15, 45, 85, 185, 285, 385])
>>> bst.between_decision_times(5)
0
>>> bst.between_decision_times(15)
1
```

```python
    def between_decision_times(self, period):
        """

        Parameters
        ----------
```

```
        period : int
                period

        Returns
        -------
        int
                index

        """
        if period == 0:
                return 0
        for i in range(len(self.information_times)):
                if self.decision_times[i] <= period and period < self.decision_t
                        return i
        return i+1
```

decision_interval: Check which decision interval the period is between. Return the index of the decision interval that the period is in. An example for this:

```
>>> bst = BigStorageTree(5, [0, 15, 45, 85, 185, 285, 385])
>>> bst.decision_interval(5)
1
>>> bst.between_decision_times(15)
1
>>> bst.between_decision_times(20)
2
```

Here, 5 is within 0 and 15 which is the first decision interval, and thus it returns 1.

```
def decision_interval(self, period):
        """

        Parameters
        ----------
        period : int
                period

        Returns
        -------
        int
                index

        """
        if period == 0:
                return 0
        for i in range(1, len(self.decision_times)):
                if self.decision_times[i-1] < period and period <= self.decision
```

```
                return i
        return i
```