Documentation for Storage_tree.py

Yili Yang

June 2017

1 Introduction

This script creates structures that provide a convenient way to store different information that is used in the optimization procedure. First, we need to define the concept of a decision period, decision interval, and inter-decision-period:

- **decision period**: the point in time when we can make a new decision[1]: how much we want to mitigate. In the paper it is assumed that we have 7 decision periods [0,15,45,85,185,285,385] (however, in the last decision period 385 we can no longer mitigate and consumption is deterministic growing from that period on).
- **decision interval**: the time span between 2 decision periods. For example, if decision periods are [0,15,45,85,185,285,385] then the 1th decision interval is {0} and the 2nd decision interval is (0,15].
- **inter-decision-period**: the subperiods that are in a decision interval. Although the cost/tax and the mitigation levels do not change between two decision periods, the consumption is still assumed to increase every year. It is because of this assumption that we need to consider inter-decision-periods. To decrease the computational burden we only consider 5-year subperiods to calculate the agents utility based on interpolated consumption flows at 5-year sub-periods and the inter decision period is [5,10...], instead of, for example, 1-year periods where the inter-decision-period is [1,2,3,...,14].

Storage_tree.py is a file containing three classes: one abstract class (*BaseStorageTree*) and two subclasses (*SmallStorageTree*, *BigStorageTree*) that inherit from the abstract class. SmallStorageTree and BigStorageTree are used to create storage trees for the Litterman's model. We can have an object that stores only the information available at decision periods (SmallStorageTree) and an object that stores the information on every period including inter-decision-periods (BigStorageTree). And this is the only difference between the two objects. Although BigStoreTree can store the information that the SmallStorageTree stores, it is not in the code for the sake of resource management. The storage trees are mainly used as dictionaries (in fact, it is calling the 'tree' attr in the class. But a custom defined inner method **__getitem__** makes the class works like a dictionary) storing various information for each

[1] Starting time of a 'decision period' described in the paper.

node within a tree object (an instance of TreeModel). For example, the class mainly provides a dictionary with keys as decision times (i.e. [0,15,45,85,100]) and items of information (e.g., mitigation level, utility function values at each node, etc.) for each decision period.

The main difference between a storage tree object (either SmallStorageTree or BigStorageTree) and a tree object is that the storage tree does not have index for nodes and states, which makes periods more 'independent' of each other (i.e. Although the Storage object is mainly using its 'tree' attr, you can not know if 15 the followed by 45 in the storage_tree.tree attr since they are keys of dictionary, to get the info, you need to go back to the decision tree attr stored in the big class. but in tree object, you can always know that 15 is followed by 45) . In spite of modelling the nodes as a sequence, the storage trees take nodes as an attribute of a certain period and the uniqueness of a node (So there are nodes in those classes, but they are not indexed) is always obtained by the period and the position of a node within a certain period. For example, node in tree object is denoted by indexes like 0,1,2 and etc. While in storage tree it is located by period such as '15' and position within period, such as '0', the '0' node in period '15' is indexed as 1 in tree object. And consequently you can not find path or reachable nodes using this class. All the information in this kind of tree is specific (like mitigation level or utility of a node while the tree object take care of the relationship between two nodes such as parent or child) and it is merely for storage usage.

In damage.py ,cost.py and utility.py, the SmallStorgeTree is only used for cost and the other entities are stored in BigStorageTree.

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 to be made. For example, [0, 15, 45, 85, 185, 285, 385].

Outputs:
There are no instances of the class because it is an abstract class.

2.1.2 Attributes

- **decision_times**:(ndarray) array of years from start where decisions about mitigation levels are to be made. For example, [0, 15, 45, 85, 185, 285, 385].

- **information_times**:(ndarray) array of years when new information is given to the agent. This is when the tree will split into two states in the next period. For example, for the decision times as above, the information times will be [0, 15, 45, 85, 185] because in the base model, we get the full knowledge on the 285 period and no information is added on 285 and 385.
- **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 an init with decision times. Also, it introduces a new concept: **information_times**, which is explained in the attributes.

```
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
```

Also, the class has a __getitem__ enableing using it as a dict (the main usage I mentioned in the introduction) and a __len__ which returns the number of keys that the tree attr have. remark,it is not pep8 and should be replaced by a public method

```
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 keys as periods and items as zero arrays (creating storage space) with the right size (for SmallStorageTree, please refer to the section 2.2.2 and for BigStorageTree please refer to section 2.3.3).

```
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 storage tree model include:

- information stored in the last decision period.
- last period (i.e., the last item of the decision time array)
- number of states that the storage tree have, equals to the sum of all the arrays' length in the storage_tree's tree (one of the attributes that is a dict) dictionary's values.

```
@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.

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

set_value : set any kind of values for all the nodes within a given period using the given values.

```
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
```

is_decision_period,**is_real_decision_period**,**is_information_period**: boolean check method to check whether a period is :

- a decision period. Continue with the above example: this check whether the period is in [0, 15, 45, 85, 185, 285, 385]
- a decision period except for the last period. This is when we can make mitigation. This method checks whether the period is in [0, 15, 45, 85, 185, 285]
- an information period. In the above example, 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, wi
            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]
```

```
        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 saves the tree's info in a row but is never used in the following code. convert problem in python 3.x version, but should work fine in py 2.x

```
        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 saves the tree's info in a csv with the following template.

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

where value0 is an abstract number, for example, it can be utility, mitigation level, consumption and etc. given what are you storing, specified as header in the inputs. Also, the next method **write_columns_existing** saves the trees info in a modified format. This kind of format is trivial and convenient to be used directly 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

This is a subclass of BaseStorageTree. In this class, decision times is the only parameter of time that we care about.

2.2.1 Inputs, Outputs and Attributes

Inputs:

- **decision_times**:(ndarray or list) array of years from start where decisions about mitigation levels are to be made. For example, [0, 15, 45, 85, 185, 285, 385].

Outputs:
A 'SmallStorageTree' object which is mainly used as an dictionary with keys including all the decision periods.

2.2.2 Attributes

- **decision_times**:(ndarray) array of years from start where decisions about mitigation levels are to be made. For example, [0, 15, 45, 85, 185, 285, 385].
- **information_times**:(ndarray) array of years when new information is given to the agent. This is when the tree will split into two states in the next period. For example, for the decision times as above, the information times will be [0, 15, 45, 85, 185]. Because in the base model, we get the full knowledge on the 285 period and no information is added on 285 and 385.)
- **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.
- **period**: (ndarray) the same as the **decision_times**.

2.2.3 Example

```
>>> sst = SmallStorageTree( [0, 15, 45, 85, 100])
>>> sst.tree
{0.0: array([ 0.]),
 15.0: array([ 0.,  0.]),
 45.0: array([ 0.,  0.,  0.,  0.]),
 85.0: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])
 100.0: array([ 0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.])}
```

2.2.4 Methods

get_next_period_array: takes a period and returns an array that 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: takes a decisition period and returns 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

This is a subclass of BaseStorageTree. This tree stores all the information on every possible interval period.

2.3.1 Inputs and Outputs

Inputs:

- **subintervals_len** : (float) the length of sub-interval that we used to calculate the approximate utility of the agent. For example, we use 5 in the base case.
- **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)

Outputs:
A 'BigStorageTree' object which is mainly used as an dictionary with keys including all the inter-decision-periods and the decision periods.

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 between the first and second decision period. For example, in the base case there are 3 subintervals :5,10,15 in (0,15]

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

```
    @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.

```
    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 interval is between and returns the index of the lower bound of the decision interval. An example for this is:

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

```
    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: Takes a period and check which decision interval, introduced in the beginning, is the period in. Return the index of the decision interval that the period is in. A decision interval is defined as (...] and thus 0 is the 1st decision interval, 1 to 15 is the 2nd and 16 to 45 is the 3rd. An example for this:

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