
Python

Gareth Walley and Aditi Shenvi

Sep 26, 2023

CONTENTS

I Quick Start	3
1 Creation of a Staged Tree	5
1.1 EventTree Class	5
1.2 StagedTree Class	7
2 Creating a Chain Event Graph	17
2.1 Example 1: Using a Stratified Dataset	17
2.2 Example 2: Chain Event Graph from Non-Stratified Dataset	19
3 Reducing a Chain Event Graph	23
4 How to make visual changes to the graphs?	27
4.1 Changing the colour palette	27
4.2 Modifying graph, node, and edge attributes	28
II API	31
5 EventTree	33
6 StagedTree	37
7 ChainEventGraph	41
8 ChainEventGraphReducer	43
Python Module Index	47
Index	49

Cegpy (/segpai/) is a Python package for working with Chain Event Graphs (CEG). It supports learning the graphical structure of a Chain Event Graph from data, encoding of parametric and structural priors, estimating its parameters, and performing inference.

It is built on top of the Python network modelling package NetworkX.

Why use cegpy?

CEGs are a flexible family of graphical models that are suitable for exploratory analysis of processes with asymmetries.

This is the first python package for modelling with CEGs. It can handle processes exhibiting context-specific conditional independence relationships and/or structural asymmetries.

It is built with extensibility in mind and is open-source, so new models built on top of CEGs can be built on top of it, and be included into the package. Our aim is to enable researchers to work with CEGs without having to start from scratch. As such, contributions are welcome!

Installation

To use cegpy, first install it using pip:

```
$ pip install cegpy
```

The package is hosted on [PyPi](#)!

Table of Contents

- Quick Start
 - *Creation of a Staged Tree*
 - *Creating a Chain Event Graph*
 - *Reducing a Chain Event Graph*
 - *How to make visual changes to the graphs?*
- API
 - *EventTree*
 - *StagedTree*
 - *ChainEventGraph*
 - *ChainEventGraphReducer*

Part I

Quick Start

CREATION OF A STAGED TREE

1.1 EventTree Class

The first starting point in constructing a Chain Event Graph (CEG) is to create an event tree describing the process being studied. An event tree is a directed tree graph with a single root node. The nodes with no emanating edges are called *leaves*, and the non-leaf nodes are called *situations*.

In this example we work with a data set which contains 4 categorical variables; *Classification*, *Group*, *Difficulty*, and *Response*.

Each individual is given a binary classification; *Blast* or *Non-blast*. Each group is rated on their experience level: *Experienced*, *Inexperienced*, or *Novice*. The classification task they are given has a difficulty rating of *Easy* or *Hard*. Finally, their response is shown: *Blast* or *Non-blast*.

We begin by importing the data set and initializing the `EventTree` object, as shown below:

```
from cegpy import EventTree
import pandas as pd

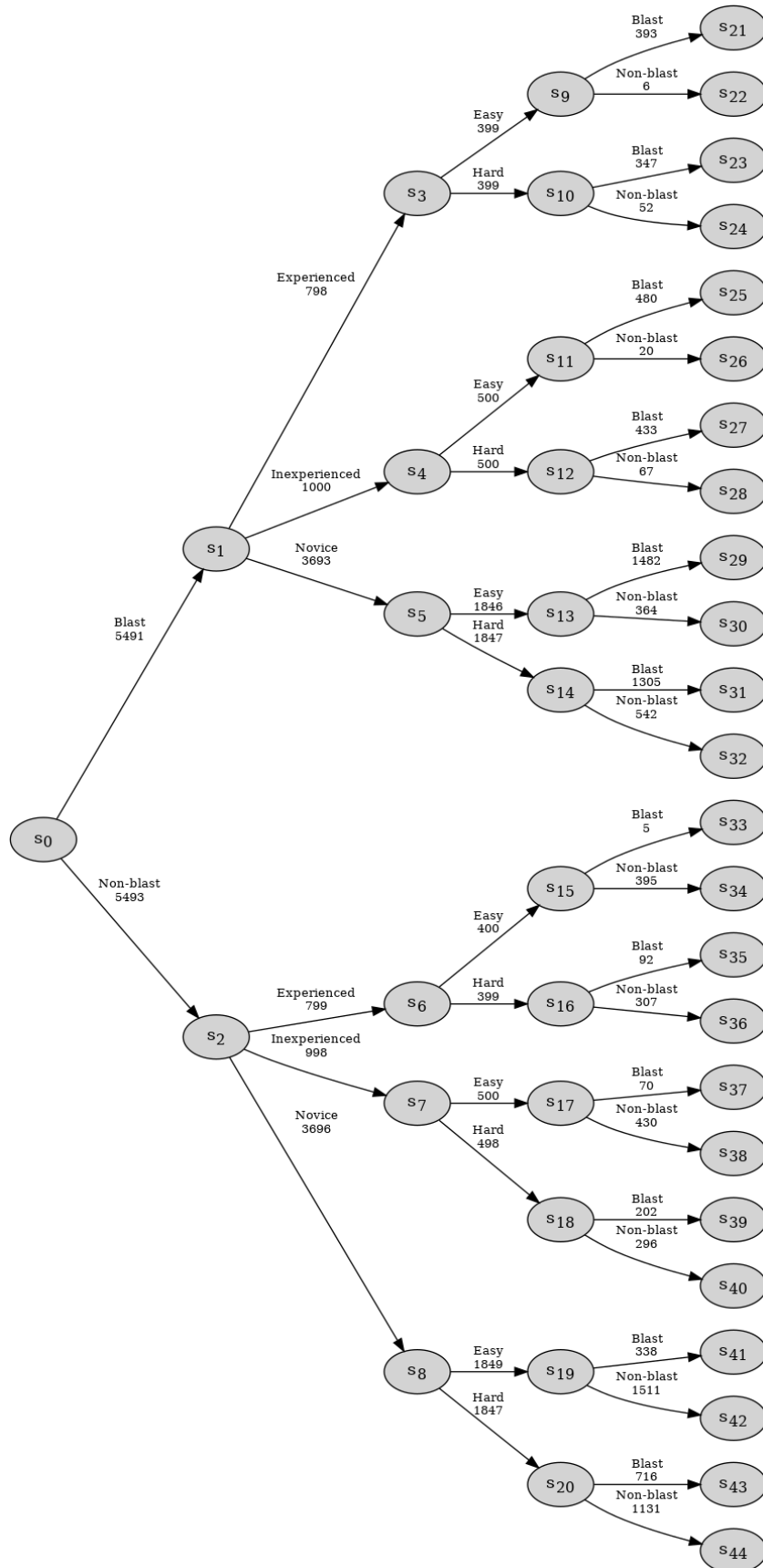
df = pd.read_excel('../data/medical_dm_modified.xlsx')
print(df.head())

#initialize the event tree
et = EventTree(df)
```

	Classification	Group	Difficulty	Response
0	Blast	Experienced	Easy	Blast
1	Non-blast	Experienced	Easy	Non-blast
2	Non-blast	Experienced	Hard	Blast
3	Non-blast	Experienced	Hard	Non-blast
4	Blast	Experienced	Easy	Blast

In order to display the `EventTree`, we can use the method `create_figure()`. The numbers above the edges of the event tree represent the number of individuals who passed through the given edge.

```
et.create_figure()
```



1.2 StagedTree Class

In an event tree, each situation is associated with a transition parameter vector which indicates the conditional probability of an individual, who has arrived at the situation, going along one of its edges. In order to create a CEG, we first need to elicit a *staged tree*. This is done by first partitioning situations into *stages*, which are collections of situations in the event tree whose immediate evolutions, i.e. their associated conditional transition parameter vectors, are equivalent. To indicate this symmetry, all situations in the same stage are assigned a single colour.

Identification of the stages in the event tree can be done using any suitable model selection algorithm. Currently, the only available selection algorithm in `ceppy` is the *Agglomerative Hierarchical Clustering (AHC)* algorithm ([Freeman and Smith, 2011](#)).

In order to create a staged tree in `ceppy` we first initialize a `StagedTree` object from the dataset and then run the AHC algorithm using the `create_AHC_transitions` method, as displayed below. The output of the AHC algorithm is a dictionary containing the following information:

- Merged Situations - a list of tuples representing the partition of the nodes into stages
- Log Likelihood - the log likelihood of the data under the model selected by AHC

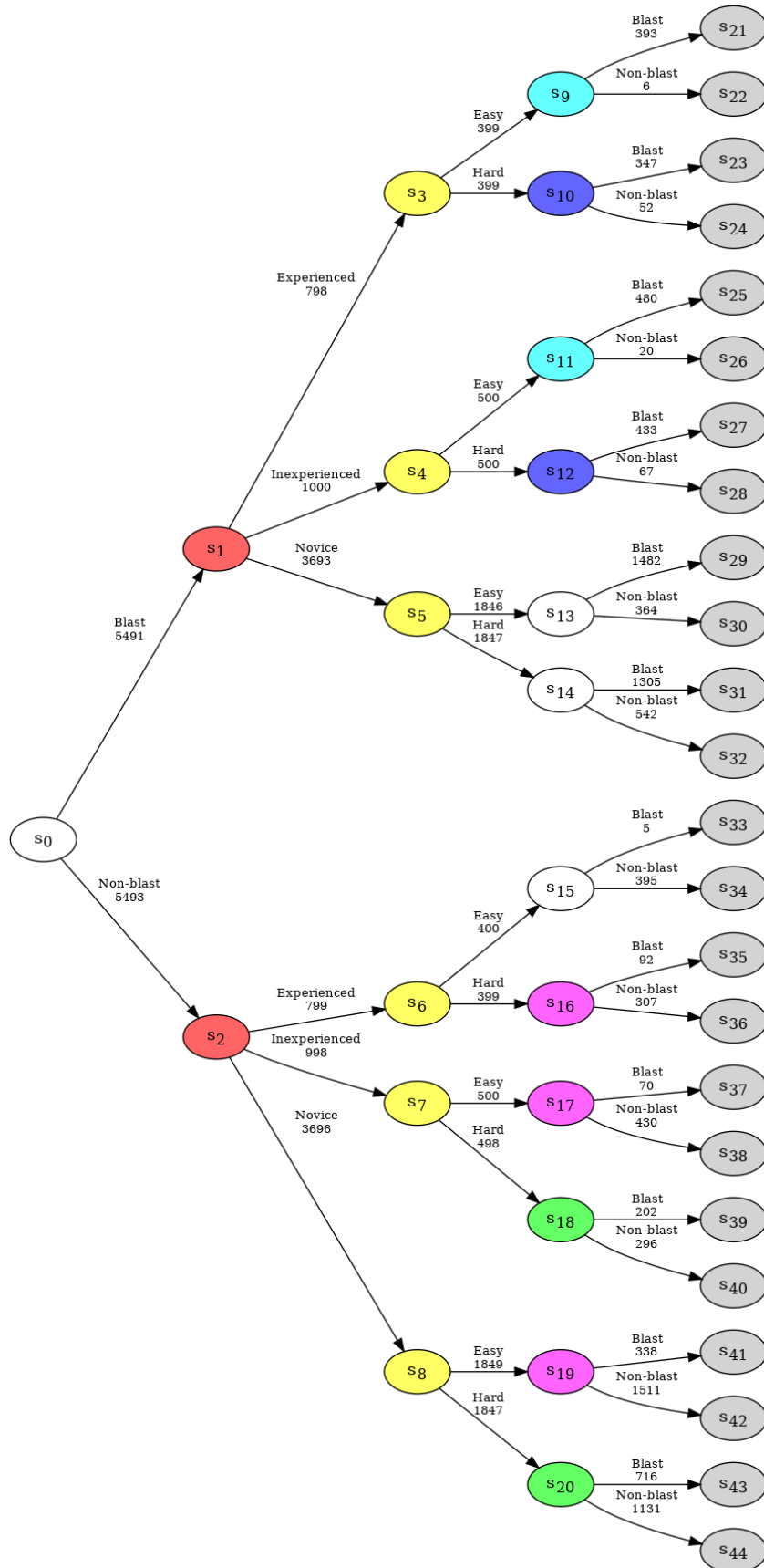
```
from ceppy import StagedTree
```

```
st = StagedTree(df)
st.calculate_AHC_transitions()
```

```
{'Merged Situations': [('s2', 's1'),
 ('s20', 's18'),
 ('s12', 's10'),
 ('s4', 's5', 's8', 's3', 's7', 's6'),
 ('s11', 's9'),
 ('s16', 's17', 's19'),
 ('s0',),
 ('s13',),
 ('s14',),
 ('s15',)],
 'Log Likelihood': -30091.353114865367}
```

Within `ceppy`, singleton stages, i.e. stages containing a single situation, are coloured white, leaves and their corresponding sink node are coloured in light-grey. Running AHC on our data set results in the following staged tree.

```
st.create_figure()
```



1.2.1 Custom Hyperstages

`ceppy` allows the user to specify which situations are allowed to be merged by the AHC algorithm. This is done by specifying a *hyperstage* (Collazo et al., 2017) which is a collection of sets such that two situations cannot be in the same stage unless they belong to the same set in the hyperstage. Under a default setting in `ceppy`, all situations which have the same number of outgoing edges and equivalent set of edge labels are in the same set within the hyperstage. The default hyperstages of a given tree can be displayed by accessing the `hyperstage` property, which returns a list of lists, where each sublist contains situations belonging to the same hyperstage.

```
st.hyperstage
```

```
[['s0',
  's9',
  's10',
  's11',
  's12',
  's13',
  's14',
  's15',
  's16',
  's17',
  's18',
  's19',
  's20'],
 ['s1', 's2'],
 ['s3', 's4', 's5', 's6', 's7', 's8']]
```

In this example, situations s_1 and s_2 belong to the same hyperstage. Each of them has three emanating edges with labels *Experienced*, *Inexperienced*, and *Novice*. However, stages s_6 and s_{15} belong to different hyperstages. They both have two emanating edges, yet different labels: *Easy*, *Hard* and *Blast*, *Non-blast*.

We can specify a different hyperstage at the point of running the AHC algorithm by passing a list defining the hyperstage partition as a parameter to the `calculate_AHC_transitions` method, for example:

```
new_hyperstage = [
    ['s0'],
    ['s3', 's4', 's5', 's6', 's7', 's8', 's9', 's10', 's11', 's12',
     's13', 's14', 's15', 's16', 's17', 's18', 's19', 's20'],
    ['s1', 's2'],
]
st.calculate_AHC_transitions(hyperstage=new_hyperstage)
```

```
{'Merged Situations': [('s2', 's1'),
 ('s20', 's18'),
 ('s12', 's10'),
 ('s4', 's5', 's8', 's3', 's7', 's6'),
 ('s11', 's9'),
 ('s16', 's17', 's19'),
 ('s0',),
 ('s13',),
 ('s14',),
 ('s15',)],
 'Log Likelihood': -30091.353114865367}
```

1.2.2 Structural and sampling zeros / missing values

The package, by default, treats all blank and NaN cells as *structural* missing values, i.e. data that is missing for a logical reason. However, sometimes these might occur due to sampling limitations; *sampling* missing values. We may also not observe a certain value for a variable in our data set (given its ancestral variables) not because that value is a structural zero but because of sampling limitations, in which case we are dealing with *sampling zeros*.

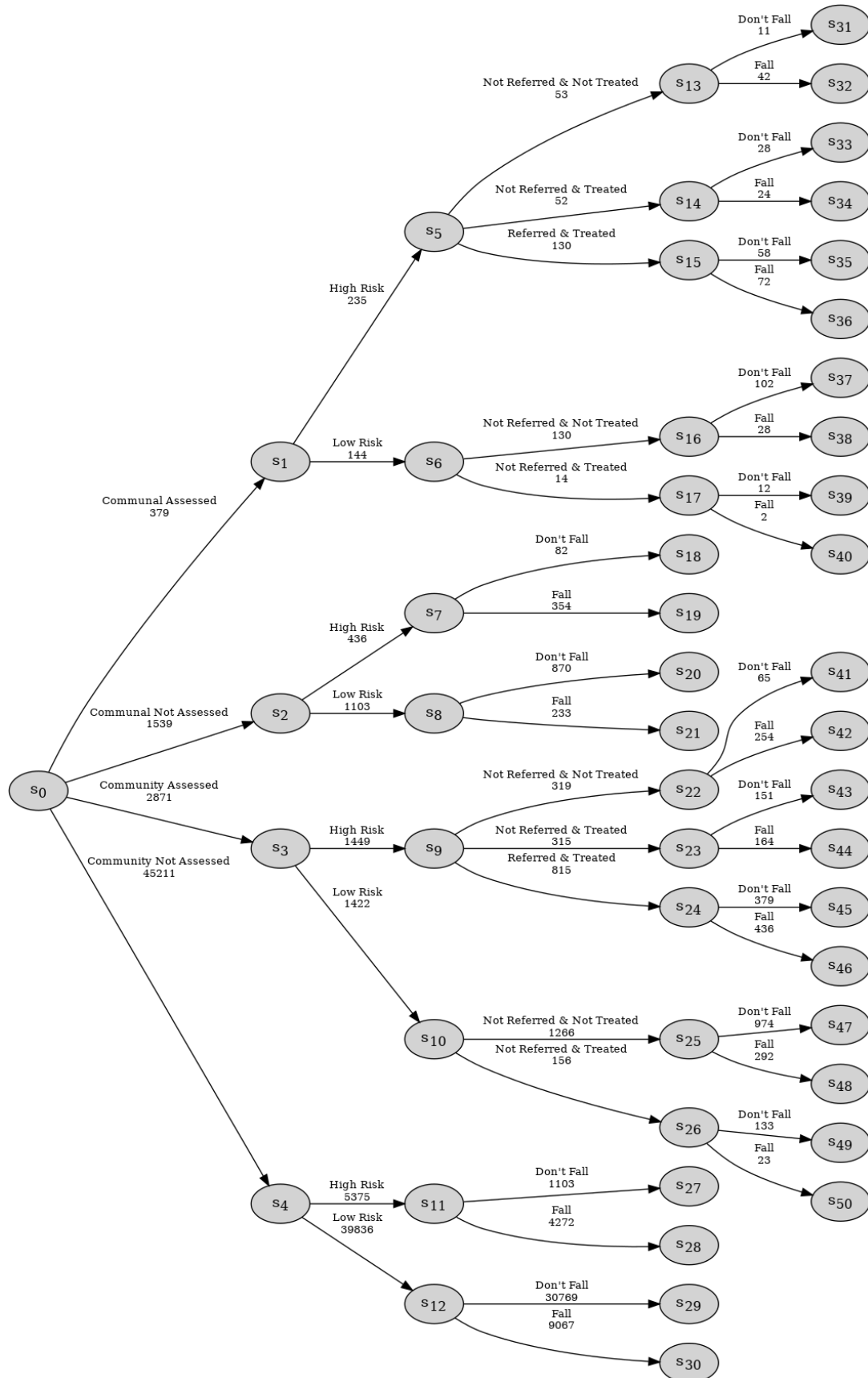
Consider the following example of the `falls.xlsx` data set which provides information concerning adults over the age of 65, and includes four categorical variables as given below with their state spaces:

- **Housing Assessment:** Living situation and whether they have been assessed, state space: {"Communal Assessed", "Communal Not Assessed", "Community Assessed", "Community Not Assessed"};
- **Risk:** Risk of a future fall, state space: {"High Risk", "Low Risk"};
- **Treatment:** Referral and treatment status, state space: {"Not Referred & Not Treated", "Not Referred & Treated", "Referred & Treated"};
- **Fall:** the outcome, state space: {"Fall", "Don't Fall"}.

```
df = pd.read_excel('../data/Falls_Data.xlsx')
df.head()
```

	HousingAssessment	Risk	Treatment	Fall
0	Community Not Assessed	Low Risk	NaN	Fall
1	Community Not Assessed	High Risk	NaN	Fall
2	Community Not Assessed	Low Risk	NaN	Don't Fall
3	Community Not Assessed	Low Risk	NaN	Don't Fall
4	Community Not Assessed	Low Risk	NaN	Fall

```
et = EventTree(df)
et.create_figure()
```



Observe that this process has structural asymmetries. None of the individuals assessed to be low risk are referred to the falls clinic and thus, for this group, the count associated with the '_Referred & Treated' category is a structural zero:

```
df[df.Risk == "Low Risk"]['Treatment'].value_counts()
```

```
Treatment
Not Referred & Not Treated    1396
Not Referred & Treated        170
Name: count, dtype: int64
```

Moreover, for individuals who are not assessed, their responses are structurally missing for the Treatment variable:

```
# Missing values in each column
print(df.isna().sum())

# Missing values for Treatment are structural,
# they are missing due to the lack of assessment:
df[df.HousingAssessment.isin([
    'Community Not Assessed', 'Communal Not Assessed'
])] ['Treatment'].isna().sum()
```

```
HousingAssessment    0
Risk                  0
Treatment             46750
Fall                  0
dtype: int64
```

```
46750
```

In `ceppy` any paths that should logically be in the event tree description of the process but are absent from the dataset due to sampling limitations would need to be manually added by the user using the `sampling zero paths` argument when initialising the `EventTree` object. Further, not all missing values in the dataset will be structurally missing.

How to distinguish between structural and sampling missing values?

e.g. Falls example: Suppose that some individuals in communal establishments who are not formally assessed but are known to be high risk were actually either "Not Referred & Treated" or "Not Referred & Not Treated" but that these observations were missing in the `falls.xlsx` dataset due to sampling limitations. All the other blank/NaN cells are structurally missing.

```
idx = (df.HousingAssessment == 'Communal Not Assessed') & (df.Risk == 'High Risk')
df[idx]
```

	HousingAssessment	Risk	Treatment	Fall
67	Communal Not Assessed	High Risk	NaN	Fall
72	Communal Not Assessed	High Risk	NaN	Fall
95	Communal Not Assessed	High Risk	NaN	Fall
102	Communal Not Assessed	High Risk	NaN	Fall
132	Communal Not Assessed	High Risk	NaN	Fall
...
49065	Communal Not Assessed	High Risk	NaN	Fall
49087	Communal Not Assessed	High Risk	NaN	Fall

(continues on next page)

(continued from previous page)

```

49135  Communal Not Assessed  High Risk      NaN  Don't Fall
49461  Communal Not Assessed  High Risk      NaN      Fall
49905  Communal Not Assessed  High Risk      NaN      Fall

[436 rows x 4 columns]

```

To demarcate the difference between structural and sampling missing values, a user can give different labels to the structural and sampling missing values in the dataset and provide these labels to the `struct_missing_label` and `missing_label` arguments respectively when initialising the `EventTree` or `StagedTree` object.

In our example, we can replace the NaN values for the `Treatment` variable among the considered subset of data with a new label, e.g. `samp_miss`:

```
df.loc[idx, 'Treatment'] = 'samp_miss'
```

Next step is to tell the `EventTree` or `StagedTree` object about these missing value arguments as shown below. This will generate a new path along `Communal Not Assessed', High Risk', 'missing')}$`:

```

et2 = EventTree(df,
    missing_label='samp_miss',
)
et2.create_figure()

```

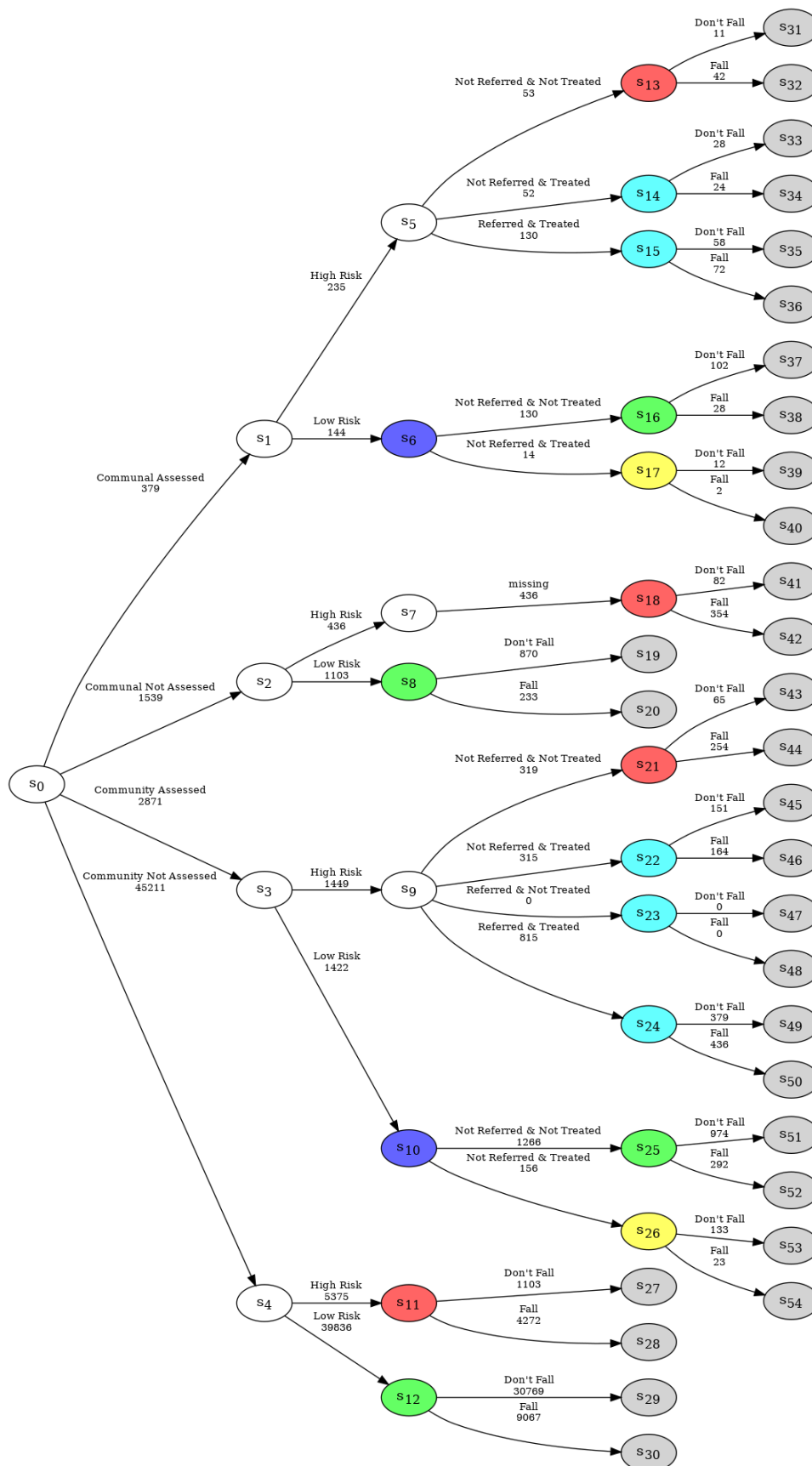


How to add sampling zeros?

e.g. Falls example: Suppose that some individuals in the community who were assessed and high risk were referred and not treated. Suppose that our observations are still the same as in the `falls.xlsx` dataset. Here, by design, this was allowed, but was not observed in the dataset. So we need to add this value in manually as a path (`"Community Assessed", "High Risk", "Referred & Not Treated"`). We also need to add in the values that follow it: i.e. (`"Community Assessed", "High Risk", "Referred & Not Treated", "Fall"`) and (`"Community Assessed", "High Risk", "Referred & Not Treated", "Don't Fall"`).

In `cegy` any paths that should logically be in the event tree description of the process but are absent from the dataset due to sampling limitations would need to be manually added by the user using the `sampling_zero_paths` argument when initialising the `EventTree` or `StagedTree` object. No changes need to be made to the dataset, as shown below:

```
st2 = StagedTree(df,
    sampling_zero_paths=[
        ('Community Assessed', 'High Risk', 'Referred & Not Treated'),
        ('Community Assessed', 'High Risk', 'Referred & Not Treated', 'Fall'),
        ('Community Assessed', 'High Risk', 'Referred & Not Treated', "Don't Fall")
    ])
st2.calculate_AHC_transitions()
st2.create_figure()
```



CREATING A CHAIN EVENT GRAPH

2.1 Example 1: Using a Stratified Dataset

This example builds a **Chain Event Graph (CEG)** from a discrete dataset showing results from a medical experiment. The dataset used is symmetrical, built from a rectangular dataset. These **CEGs** are known as *stratified* in the literature.

The **Agglomerative Hierarchical Clustering (AHC)** algorithm is used to maximise the log marginal likelihood score of the staged tree/CEG model to determine its stages. The package functions under a Bayesian framework and priors can be supplied to the **AHC** algorithm to override the default settings,

The example `medical.xlsx` dataset contains 4 categorical variables; `Classification`, `Group`, `Difficulty`, `Response`.

Each individual is given a binary classification; `Blast` or `Non-blast`. Each group is rated on their experience level; `Experienced`, `Inexperienced`, or `Novice`. The classification task they are given has a difficulty rating of `Easy` or `Hard`. Finally, their response is shown; `Blast` or `Non-blast`.

Firstly, a staged tree object is created from a data source, and calculate the AHC transitions.

```
from cegpy import StagedTree, ChainEventGraph
import pandas as pd
```

```
dataframe = pd.read_excel("medical.xlsx")
dataframe
```

	Classification	Group	Difficulty	Response
0	Blast	Experienced	Easy	Blast
1	Non-blast	Experienced	Easy	Non-blast
2	Non-blast	Experienced	Hard	Blast
3	Non-blast	Experienced	Hard	Non-blast
4	Blast	Experienced	Easy	Blast
...
10979	Blast	Novice	Easy	Non-blast
10980	Blast	Novice	Easy	Blast
10981	Non-blast	Novice	Easy	Blast
10982	Blast	Novice	Easy	Non-blast
10983	Non-blast	Novice	Hard	Non-blast

[10984 rows x 4 columns]

```
# Descriptive statistics for the dataset
dataframe.describe()
```

	Classification	Group	Difficulty	Response
count	10984	10984	10984	10984
unique	2	3	2	2
top	Non-blast	Novice	Easy	Blast
freq	5493	7389	5494	5863

The AHC algorithm is executed on the event tree, and the nodes are assigned a colour if they are found to be in the same stage as each other. Note that the `calculate_AHC_transitions` method is only available from the `StagedTree` class and not the `EventTree` class.

Effectively, nodes in the same stage share the same parameter set; in other words, the immediate future of these nodes is identical. Note that singleton stages are not coloured in the staged tree and its corresponding CEG to prevent visual cluttering.

When the CEG is created, equivalent nodes (precisely, those whose complete future is identical) in a stage will be combined to compress the graph.

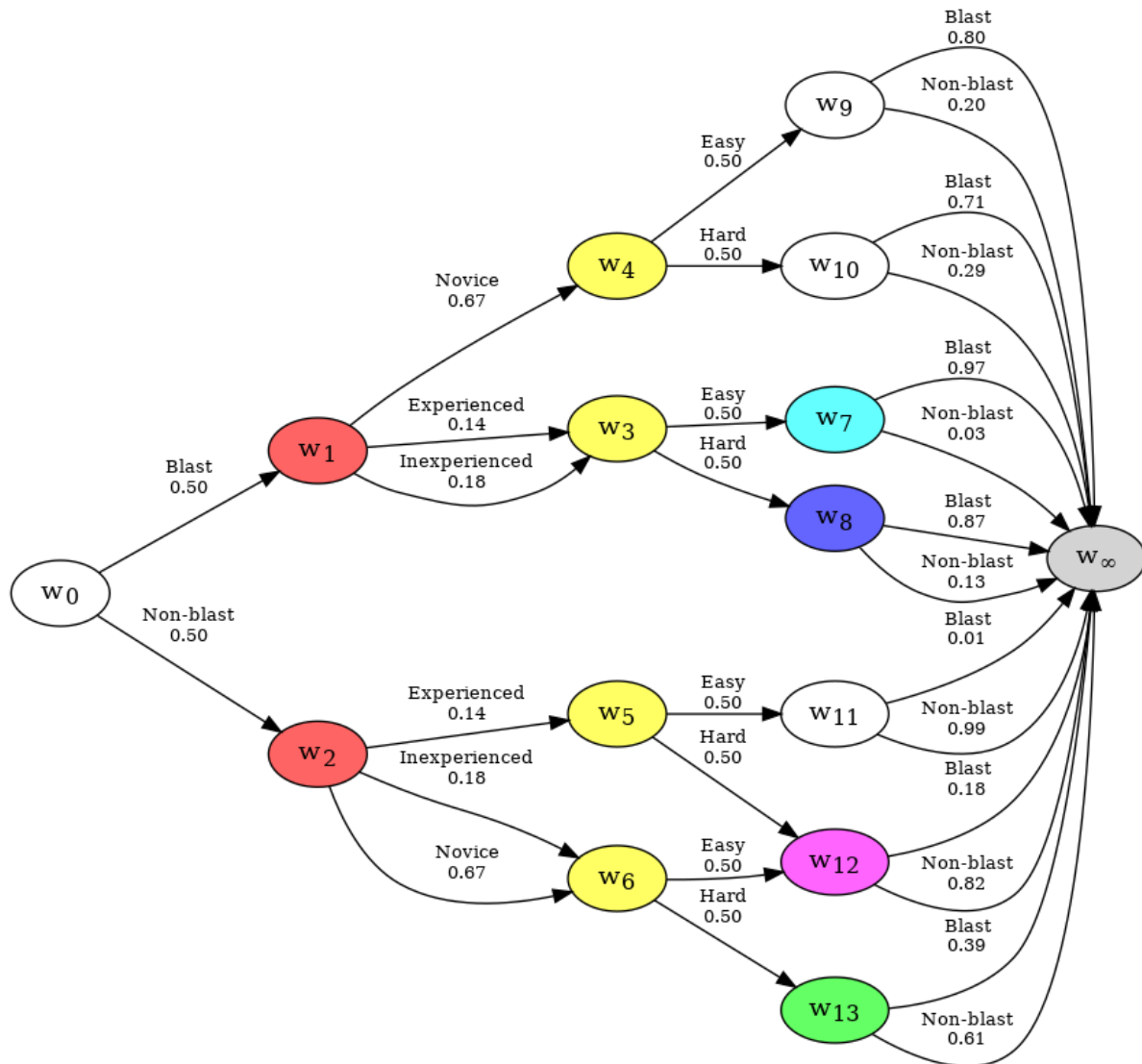
```
staged_tree = StagedTree(dataframe)
staged_tree.calculate_AHC_transitions();
```

Once the AHC algorithm has been run to identify the stages, a CEG can be created by passing the `StagedTree` object into the `ChainEventGraph` class. When the `ChainEventGraph` is created, it automatically generates the CEG from the `StagedTree` object. The process of generation compares nodes that are in the same stage to determine if they are logically compatible with one another. Once the graph has been constructed, and nodes combined, the probabilities of passing down any given edge are displayed.

Like the `StagedTree`, the graph can be displayed using the `create_figure` method as shown below.

```
from IPython.display import Image

chain_event_graph = ChainEventGraph(staged_tree)
chain_event_graph.create_figure()
```



The tree has now been compressed into a Chain Event Graph. The graph represents the system encoded in the data. All paths start at the root node w_0 , (which represents an individual entering the system), and terminate at the sink node w_∞ (which represents the point at which an individual exits the system).

2.2 Example 2: Chain Event Graph from Non-Stratified Dataset

This example builds a Chain Event Graph (CEG) from a asymmetric dataset. In simple words, a dataset is asymmetric when the event tree describing the dataset is not symmetric around its root. The class of CEGs built from asymmetric event trees is said to be non-stratified. Note that, technically, a CEG is also said to be non-stratified when the order of events along its different paths is not the same, even though its event tree might be symmetric. Whilst such processes can also be easily modelled with the `cegy` package, for this example we focus on non-stratified CEGs that are built from asymmetric event trees/datasets.

Asymmetry in a dataset arises when it has structural zeros or structural missing values in certain rows; in other words, the sample space of a variable is different or empty respectively, depending on its ancestral variables. So logically, certain values of the variable will never be observed for certain configurations of its ancestral variables, irrespective of the sample size.

In this example, we consider the falls.xlsx dataset. Here, by interventional design, individuals who are not assessed are not offered referral or treatment. In this case, we would observe individuals in our dataset who are not assessed, going down the 'Not Referred & Not Treated' path with probability 1. This is not helpful, and so we choose to condense the tree and remove this edge. The zero observations for non-assessed individuals for the categories of 'Referred & Treated' and 'Not Referred & Treated' are both structural zeros.

```
from cegpy import EventTree
import pandas as pd

dataframe = pd.read_excel("falls.xlsx")
dataframe
```

```

      HousingAssessment      Risk Treatment      Fall
0      Community Not Assessed  Low Risk      NaN      Fall
1      Community Not Assessed  High Risk      NaN      Fall
2      Community Not Assessed  Low Risk      NaN  Don't Fall
3      Community Not Assessed  Low Risk      NaN  Don't Fall
4      Community Not Assessed  Low Risk      NaN      Fall
...
49995  Community Not Assessed  Low Risk      NaN  Don't Fall
49996  Community Not Assessed  Low Risk      NaN  Don't Fall
49997  Community Not Assessed  Low Risk      NaN  Don't Fall
49998  Community Not Assessed  Low Risk      NaN      Fall
49999  Community Not Assessed  Low Risk      NaN      Fall

[50000 rows x 4 columns]
```

Note: When looking at the description of the dataset, the total count in the Treatment column is not equal to the counts for the other columns. This is the giveaway that the dataset is non-stratified. Extreme care must be taken to ensure that the dataset really is non-stratified, and doesn't simply have sampling-zeros or sampling missing values. The package has no way of distinguishing these on its own unless the user specifies them.

```
dataframe.describe()
```

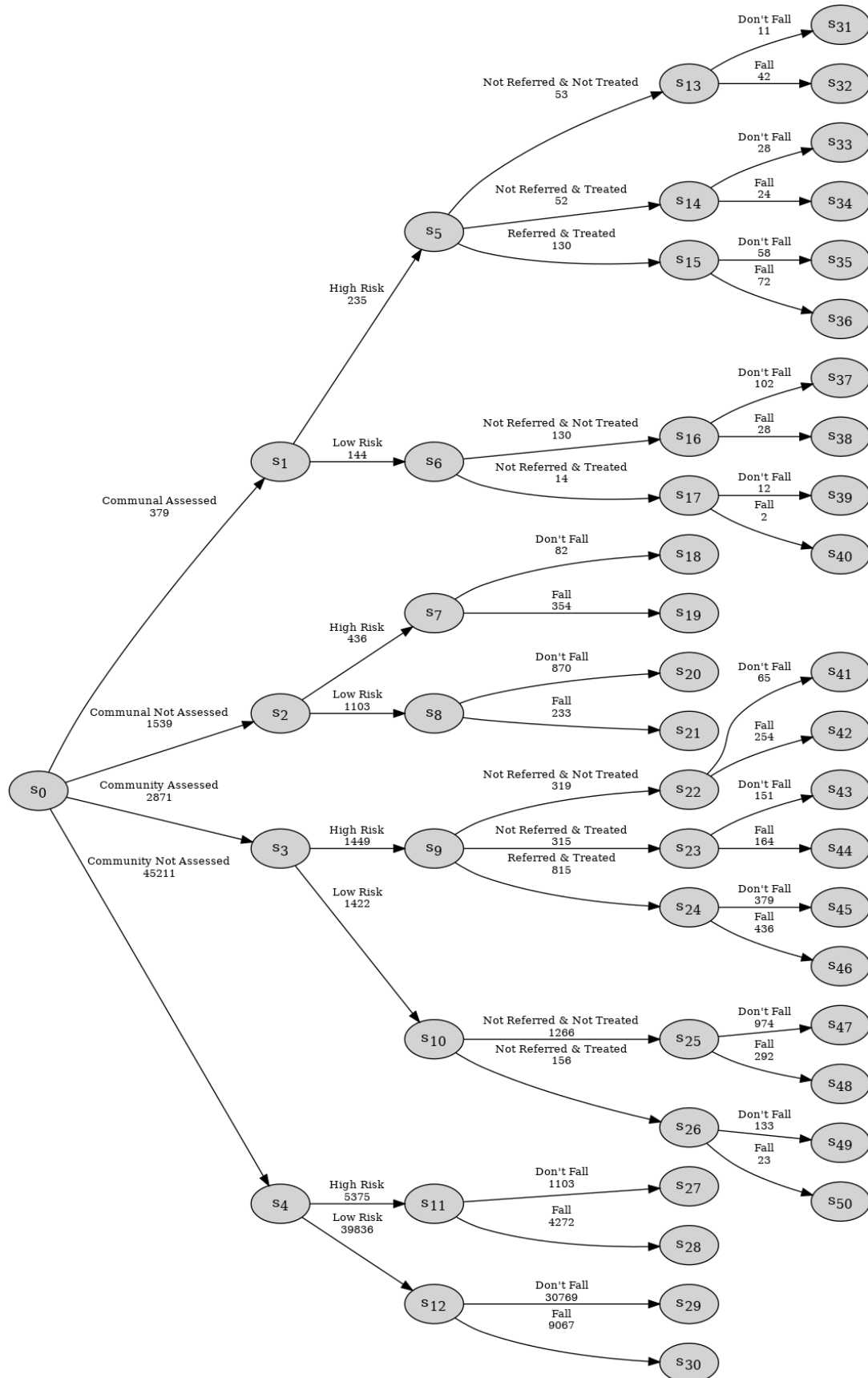
```

count      HousingAssessment      Risk      Treatment \
unique              4              2              3
top      Community Not Assessed  Low Risk  Not Referred & Not Treated
freq              45211      42505              1768

count      Fall
unique              2
top      Don't Fall
freq      34737
```

The end result of this is that in the EventTree shown below, paths such as $S0 \rightarrow S2 \rightarrow S7 \rightarrow S18$ skip the Treatment variable.

```
event_tree = EventTree(dataframe)
event_tree.create_figure()
```

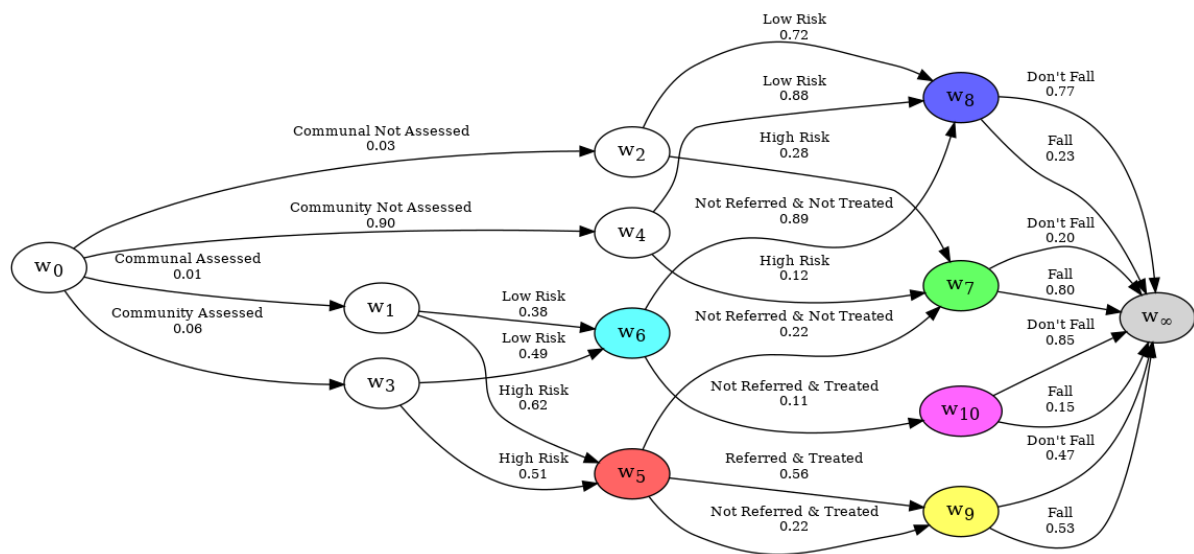
As in the stratified medical example, after initial checks on the dataset, and confirmation that the EventTree looks as expected, the next step is to identify the stages. For this, we use the StagedTree class, which first creates the EventTree internally, ready for the user to run a clustering algorithm on it. In this example we use the `.calculate_AHC_transitions()` method, which executes the agglomerative hierarchical clustering (AHC) algorithm on the EventTree. The package functions under a Bayesian framework and priors can be supplied to the AHC algorithm to override the default settings.

The resultant CEG has been reduced from the tree representation to a more compact graph.

```
from cegpy import ChainEventGraph, StagedTree

st = StagedTree(dataframe)
st.calculate_AHC_transitions()

ceg = ChainEventGraph(st)
ceg.create_figure()
```



As a CEG is a probabilistic model of a series of events, it may be desirable to view a CEG sub-graph when some or all of the variables are known. This can be especially true for graphs with lots of variables, which can balloon in size. In `cegpy`, this is done by using the `ChainEventGraphReducer` which is covered on the next page.

REDUCING A CHAIN EVENT GRAPH

A complete CEG shows all possible trajectories that an individual undergoing the process might experience. However, on observing any evidence, certain or uncertain, some edges and nodes become unvisited with probability 1. The CEG model can be reduced such that these edges and nodes are excluded, without any loss of information. Once reduced, the probabilities displayed can be also be revised.

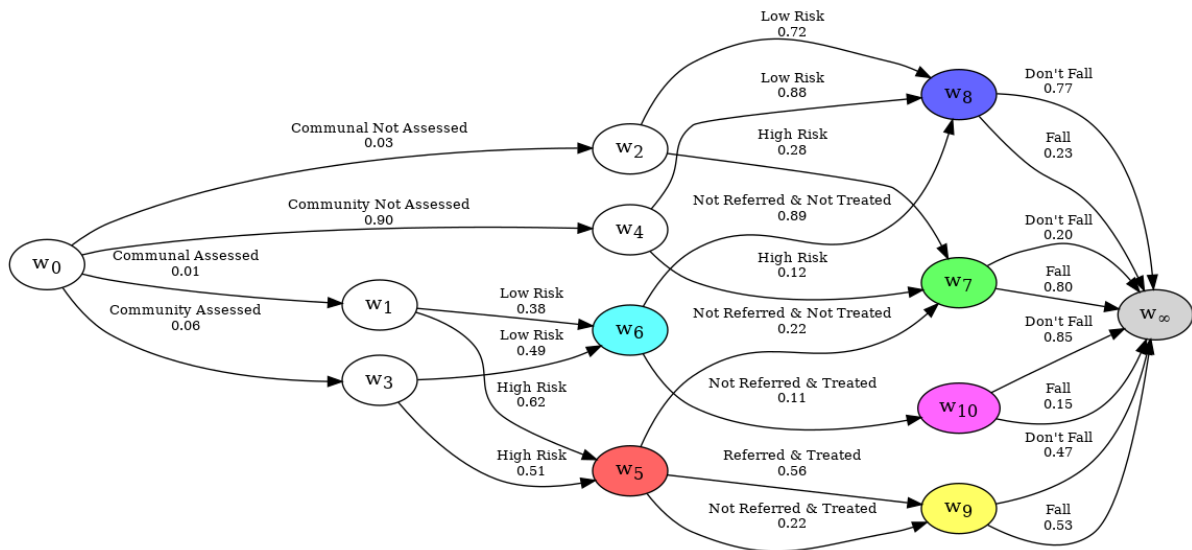
For this example, we will use the `falls.xlsx` dataset.

```
from cegpy import StagedTree, ChainEventGraph, ChainEventGraphReducer
import pandas as pd

dataframe = pd.read_excel("falls.xlsx")

staged_tree = StagedTree(dataframe)
staged_tree.calculate_AHC_transitions()

falls_ceg = ChainEventGraph(staged_tree)
falls_ceg.create_figure()
```



When examining a dataset with a CEG, you may wish to see a subset of the graph, where some events are excluded with probability zero. Consider the CEG representation of the falls dataset; It may be interesting to split the graph into two graphs, one for individuals on the `Communal` paths, and another for people on the `Community` paths. This is achieved by using uncertain evidence. In our case, we know that anyone who is community assessed will have either passed along the `Community Not Assessed` edge or the `Community Assessed` edge, which can be done like so:

```

from cegpy import ChainEventGraphReducer

reducer = ChainEventGraphReducer(falls_ceg)
reducer.add_uncertain_edge_set(
    edge_set={
        ("w0", "w4", "Community Not Assessed"),
        ("w0", "w3", "Community Assessed"),
    }
)
print(reducer)

```

The evidence you have given is as follows:

Evidence you are certain of:

Edges = []

Nodes = {}

Evidence you are uncertain of:

Edges = [

{('w0', 'w4', 'Community Not Assessed'), ('w0', 'w3', 'Community Assessed')},

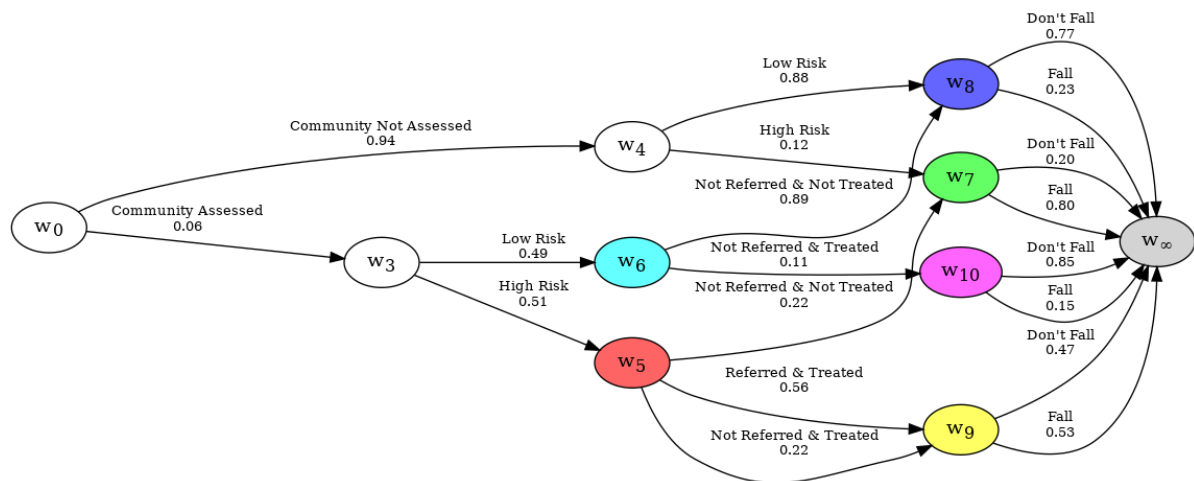
]

Nodes = {

}

The reduced graph is stored in the graph attribute, and is a ChainEventGraph object.

```
reducer.graph.create_figure()
```

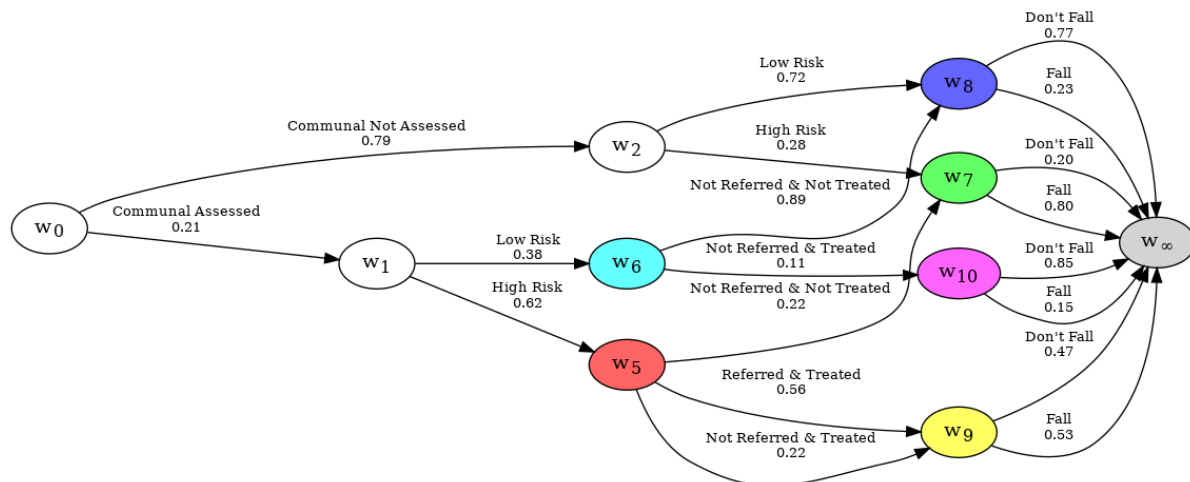


Likewise, we can do the same for the Communal graph. In this case, it could be simpler to just specify the sub-graph which contains all paths that pass through nodes w1 and w2.

```

reducer = ChainEventGraphReducer(falls_ceg)
reducer.add_uncertain_node_set({"w1", "w2"})
reducer.graph.create_figure()

```



It may also be interesting to see the sub-graph of those Communal individuals who had a Fall.

```

reducer.add_uncertain_edge_set(
    {
        (u, v, l)
        for (u, v, l) in reducer.graph.edges
        if l == "Fall"
    }
)
print(reducer)

```

The evidence you have given is as follows:

Evidence you are certain of:

Edges = []

Nodes = {}

Evidence you are uncertain of:

Edges = [

{('w10', 'w_infinity', 'Fall'), ('w9', 'w_infinity', 'Fall'), ('w8', 'w_infinity', 'Fall'), ('w7', 'w_infinity', 'Fall')},

]

Nodes = {

{'w1', 'w2'},

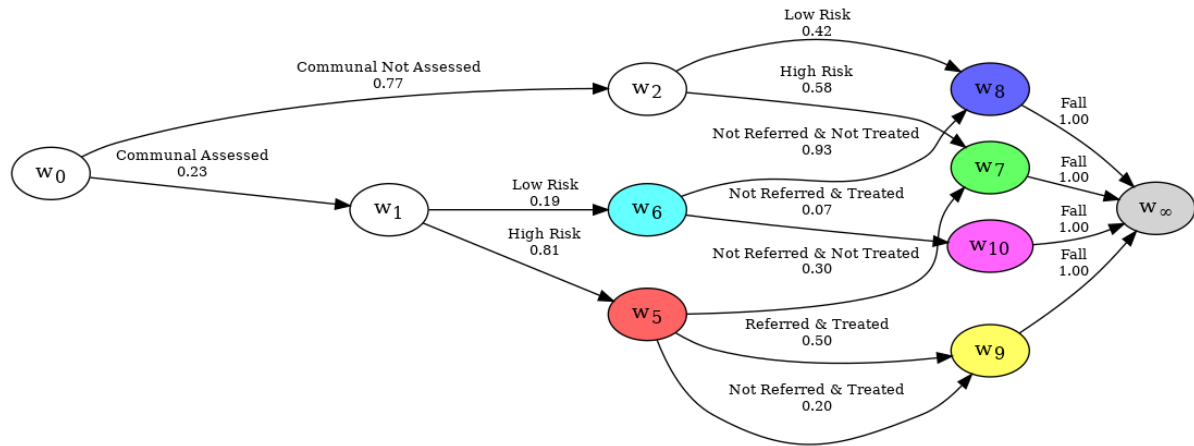
}

The probabilities are adjusted across all the edges, and back propagated through the graph automatically.

```

reducer.graph.create_figure()

```

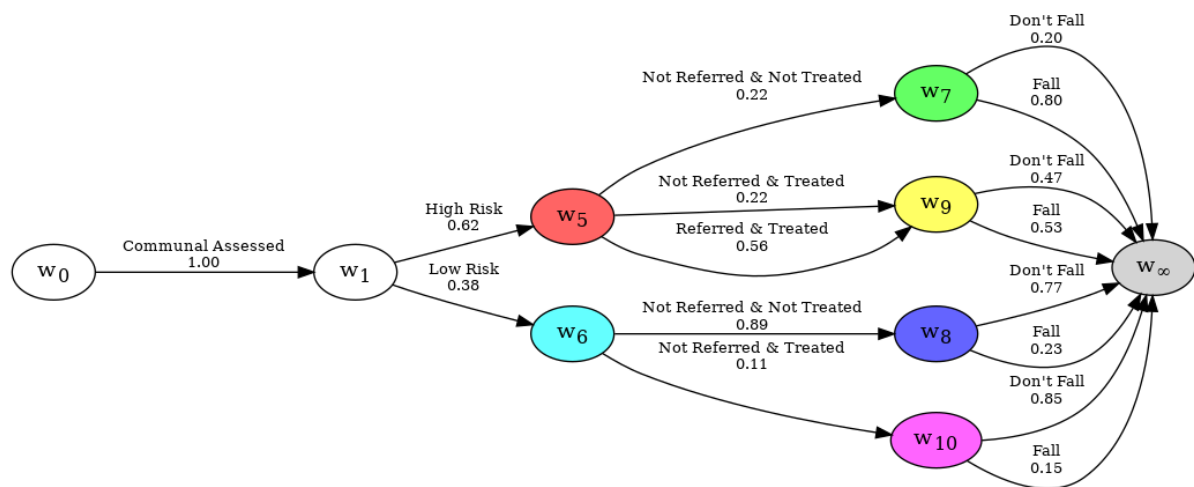


When you would like to adjust the graph to only show paths which pass through a specific edge or node, certain evidence is used.

Take the following example. You'd like to see what might have happened to an individual who was Communal Assessed. This can be done by using the `add_certain_edge` method.

```
reducer.clear_all_evidence()

reducer.add_certain_edge("w0", "w1", "Communal Assessed")
# or reducer.add_certain_node("w1")
reducer.graph.create_figure()
```



HOW TO MAKE VISUAL CHANGES TO THE GRAPHS?

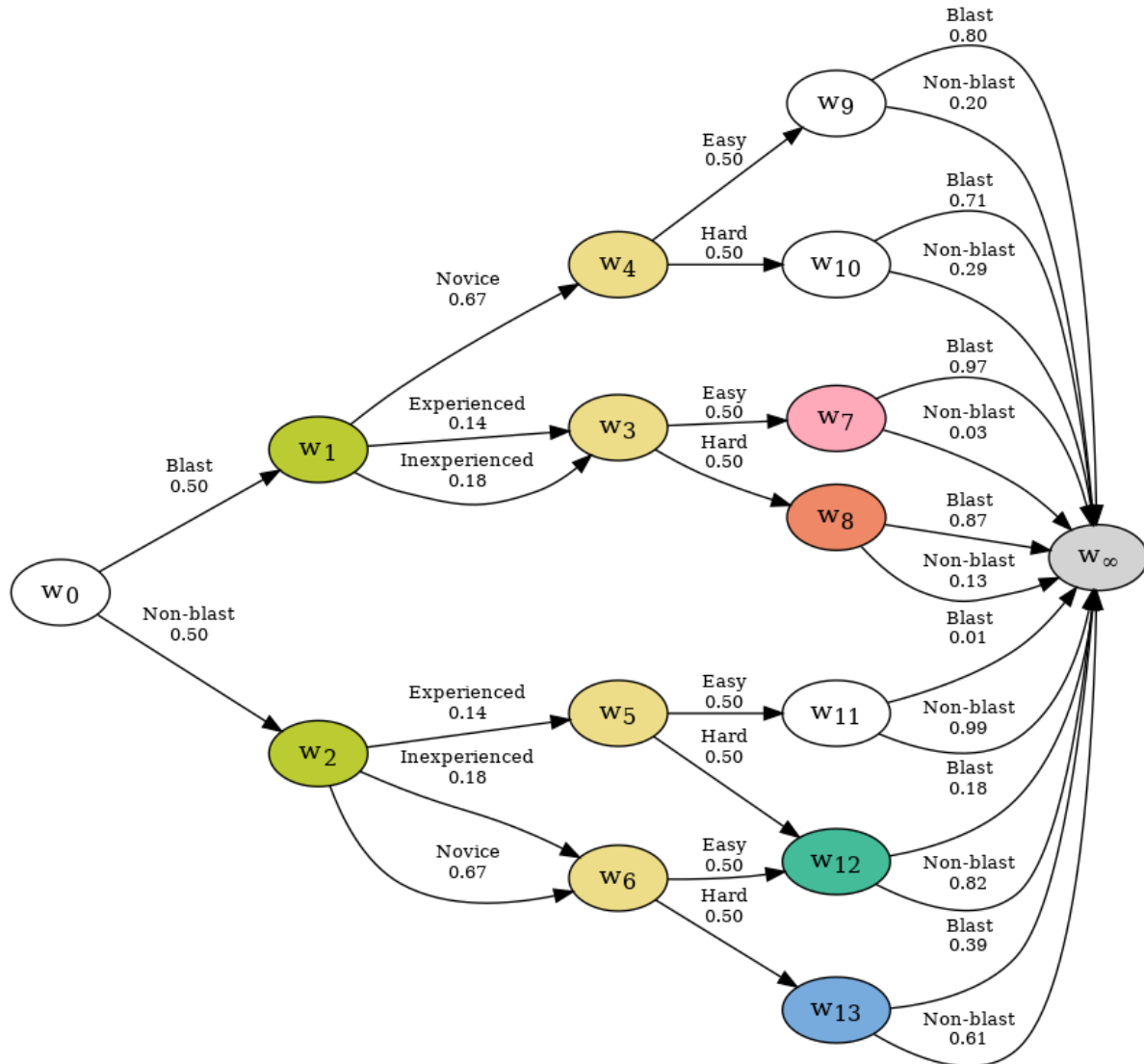
4.1 Changing the colour palette

By default, the colours of the nodes in `cegy` are selected uniformly at random from the entire spectrum of colours. If we want to use a specific colour palette, a list of colours to be used by the AHC algorithm can be specified as a parameter when calling the `create_AHC_transitions` method, for example:

```
from cegpy import StagedTree, ChainEventGraph
import pandas as pd

df = pd.read_excel('../data/medical_dm_modified.xlsx')

st = StagedTree(df)
colours = ['#BBCC33', '#77AADD', '#EE8866', '#EEDD88', '#FFAABB', '#44BB99']
st.calculate_AHC_transitions(colour_list=colours)
ceg = ChainEventGraph(st)
ceg.create_figure()
```



4.2 Modifying graph, node, and edge attributes

The graphs in `cegy` are built with `GraphViz` and `PyDotPlus`. We can access the underlying `pydotplus`. `graphviz.Dot` object by accessing the `dot_graph` property. This enables visual modifications of our event tree, staged tree, or CEG. For example, the following code modifies the distance between the nodes, changes the style of each edge labelled "Hard" from solid to dashed, and changes the shape of the root node from oval to square.

```
from IPython.display import Image

g = ceg.dot_graph()
g.set('ranksep', 0.1)
g.set('nodesep', 0.2)

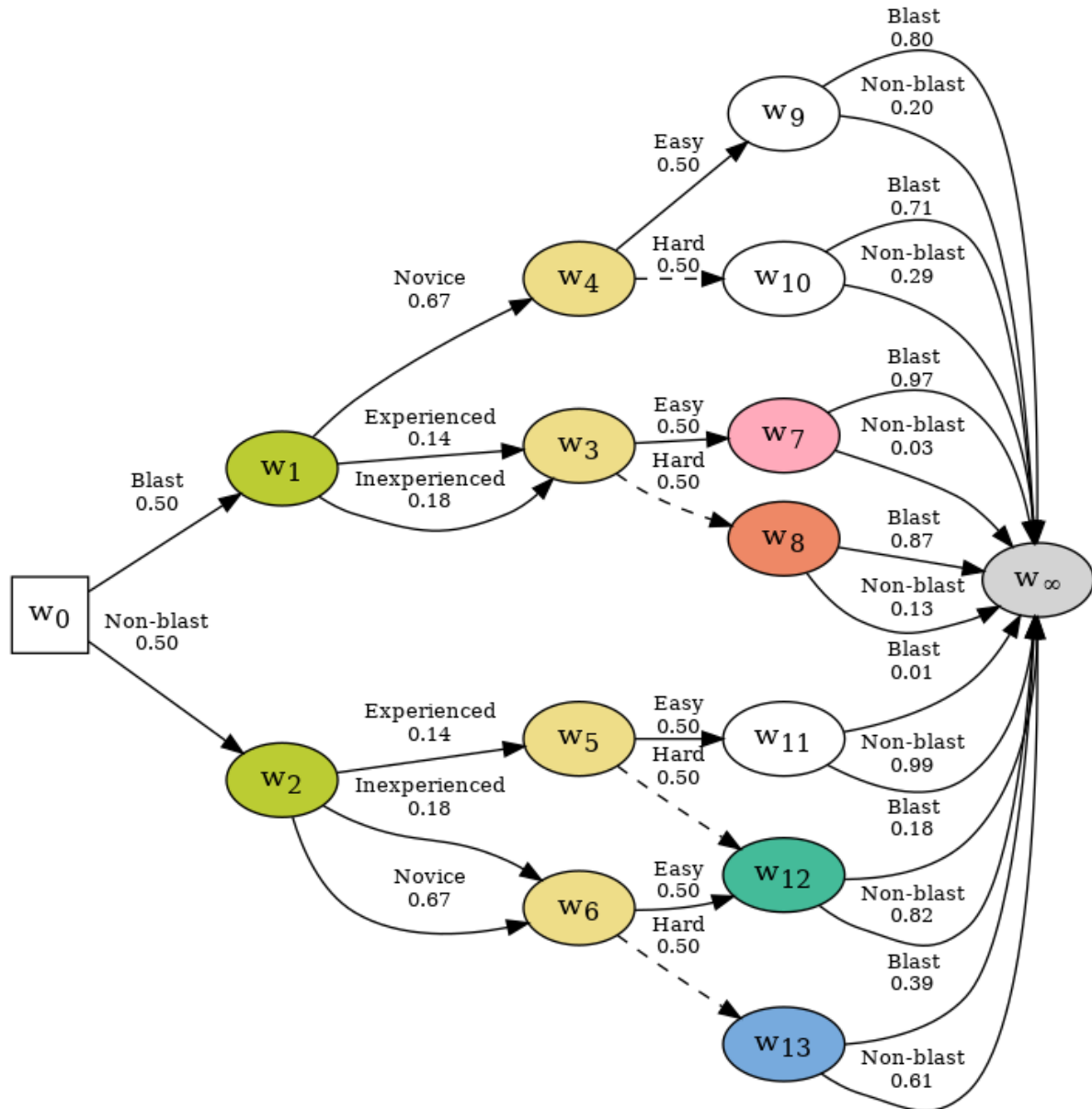
for edge in g.get_edge_list():
    if "Hard" in edge.get("label"):
        edge.set_style('dashed')
```

(continues on next page)

(continued from previous page)

```
g.get_node('w0')[-1].set_shape('square')
```

```
Image(g.create_png())
```



For more information about the available graph, node, and edge attributes refer to the [GraphViz](#) and [PyDotPlus](#) documentation.

Part II

API

EVENTTREE

```
class cegpy.EventTree (dataframe: DataFrame, sampling_zero_paths=None, var_order=None,
                        struct_missing_label=None, missing_label=None, complete_case=False)
```

Bases: MultiDiGraph

This class extends the NetworkX MultiDiGraph class to allow the creation of event tree representations of data.

Parameters

- **dataframe** (*pandas.DataFrame*) – Required - DataFrame containing variables as column headers, with event name strings in each cell. These event names will be used to create the edges of the event tree. Counts of each event will be extracted and attached to each edge.
- **sampling_zero_paths** (*List[Tuple[str]] or None*) – Optional - Paths to sampling zeros.

Format is as follows: [('edge_1'), ('edge_1', 'edge_2'), ...]

If no paths are specified, default setting is that no sampling zero paths are created.

- **var_order** (*List[str] or None*) – Optional - Specifies the ordering of variables to be adopted in the event tree. Default var_order is obtained from the order of columns in dataframe. String labels in the list should match the column names in dataframe.
- **struct_missing_label** (*str or None*) – Optional - Label in the dataframe for observations which are structurally missing; e.g: Post operative health status is irrelevant for a dead patient. Label example: “struct”.
- **missing_label** (*str or None*) – Optional - Label in the dataframe for observations which are missing values that are not structurally missing. e.g: Missing height for some individuals in the sample. Label example: “miss” Whatever label is provided will be renamed in the event tree to “missing”.
- **complete_case** (*bool*) – Optional - If True, all entries (rows) with non-structural missing values are removed. Default setting: False.

property root: str

Returns

The name of the root node of the event tree, currently hard coded to 's0'.

Return type

str

property variables: List

Returns

The column headers of the dataset.

Return type

List[str]

property sampling_zeros: Optional[List[Tuple[str]]]

Setting this property will apply sampling zero paths to the tree. If different to previous value, the event tree will be regenerated.

Returns

Sampling zero paths provided by the user.

Return type

List[Tuple[str]] or None

property situations: List[str]**Returns**

The situations of the tree (non-leaf nodes).

Return type

List[str]

property leaves: List[str]**Returns**

The leaves of the tree.

Return type

List[str]

property edge_counts: Dict

The counts along edges all edges in the tree, where edges are a Tuple like so: ("source_node", "destination_node", "edge_label").

Returns

A mapping of edges to their counts.

Return type

Dict[Tuple[str], Int]

property categories_per_variable: Dict

The number of unique categories/levels for each variable (column headings in dataframe).

Returns

A mapping of variables to the number of unique categories/levels.

Return type

Dict[str, Int]

dot_graph (*edge_info: str = 'count'*) → Dot

Returns Dot graph representation of the event tree. :param edge_info: Optional - Chooses which summary measure to be displayed on edges. In event trees, only "count" can be displayed, so this can be omitted.

Returns

A graphviz Dot representation of the graph.

Return type

pydotplus.Dot

create_figure (*filename=None, edge_info: str = 'count'*) → Optional[Image]

Creates event tree from the dataframe.

Parameters

- **filename** (*str*) – Optional - When provided, file is saved to the filename, local to the current working directory. e.g. if filename = “output/event_tree.svg”, the file will be saved to: cwd/output/event_tree.svg Otherwise, if function is called inside an interactive notebook, image will be displayed in the notebook, even if filename is omitted. Supports any filetype that graphviz supports. e.g: “event_tree.png” or “event_tree.svg” etc.
- **edge_info** (*str*) – Optional - Chooses which summary measure to be displayed on edges. In event trees, only “count” can be displayed, so this can be omitted.

Returns

The event tree Image object.

Return type

IPython.display.Image or None

STAGEDTREE

```
class cegpy.StagedTree (dataframe, sampling_zero_paths=None, var_order=None,
                        struct_missing_label=None, missing_label=None, complete_case=False)
```

Bases: *EventTree*

Representation of a Staged Tree.

A staged tree is a tree where each node is a situation, and each edge is a transition from one situation to another. Each situation is given a 'stage' which groups it with other situations which have the same outgoing edges, with equivalent probabilities of occurring.

The class is an extension of EventTree.

Parameters

- **dataframe** (*pandas.DataFrame*) – Required - DataFrame containing variables as column headers, with event name strings in each cell. These event names will be used to create the edges of the event tree. Counts of each event will be extracted and attached to each edge.
- **sampling_zero_paths** (*List[Tuple[str]] or None*) – Optional - Paths to sampling zeros.

Format is as follows: [('edge_1'), ('edge_1', 'edge_2'), ...]

If no paths are specified, default setting is that no sampling zero paths are created.

- **var_order** (*List[str] or None*) – Optional - Specifies the ordering of variables to be adopted in the event tree. Default var_order is obtained from the order of columns in dataframe. String labels in the list should match the column names in dataframe.
- **struct_missing_label** (*str or None*) – Optional - Label in the dataframe for observations which are structurally missing; e.g: Post operative health status is irrelevant for a dead patient. Label example: "struct".
- **missing_label** (*str or None*) – Optional - Label in the dataframe for observations which are missing values that are not structurally missing. e.g: Missing height for some individuals in the sample. Label example: "miss" Whatever label is provided will be renamed in the event tree to "missing".
- **complete_case** (*bool*) – Optional - If True, all entries (rows) with non-structural missing values are removed. Default setting: False.

```
property prior: Dict[Tuple[str], List[Fraction]]
```

A mapping of priors keyed by edge. Keys are 3-tuples of the form: (src, dst, edge_label).

Returns

A mapping edge -> priors.

Return type

Dict[Tuple[str], List[Fraction]]

property prior_list: List[List[Fraction]]**Returns**

Priors in the form of a list of lists.

Return type

List[List[Fraction]]

property posterior: Dict[Tuple[str], List[Fraction]]

Posteriors along each edge, calculated by adding edge count to the prior for each edge. Keys are 3-tuples of the form: (src, dst, edge_label).

Returns

Mapping of edge -> edge_count + prior

Return type

Dict[Tuple[str], List[Fraction]]

property posterior_list: List[List[Fraction]]**Returns**

Posteriors in the form of a list of lists.

Return type

List[List[Fraction]]

property alpha: float

The equivalent sample size set for the root node which is then uniformly propagated through the tree.

Returns

The value of Alpha.

Return type

float

property hyperstage: List[List[str]]

Indication of which nodes are allowed to be in the same stage. Each list is a list of node names e.g. "s0".

Returns

The List of all hyperstages.

Return type

List[List[str]]

property edge_countset: List[List]

Indexed the same as situations. :return: Edge counts emanation from each node of the tree. :rtype: List[List]

property ahc_output: Dict

Contains a List of Lists containing all the situations that were merged, and the log likelihood.

Returns

The output from the AHC algorithm.

Return type

Dict

calculate_AHC_transitions (*prior=None, alpha=None, hyperstage=None, colour_list=None*) → Dict

Bayesian Agglomerative Hierarchical Clustering algorithm implementation. It returns a list of lists of the situations which have been merged together, the likelihood of the final model.

Parameters

- **prior** (*Dict*[*Tuple*[*str*], *List*[*Fraction*]]) – Optional - A mapping of priors keyed by edge. Keys are 3-tuples of the form: (src, dst, edge_label).
- **alpha** (*float*) – Optional - The equivalent sample size set for the root node which is then uniformly propagated through the tree.
- **hyperstage** (*List*[*List*[*str*]]) – Optional - Indication of which nodes are allowed to be in the same stage. Each list is a list of node names e.g. “s0”.
- **colour_list** (*List*[*str*]) – Optional - a list of hex colours to be used for stages. Otherwise, colours evenly spaced around the colour spectrum are used.

Returns

The output from the AHC algorithm, specified above.

Return type

Dict

dot_graph (*edge_info*: *str* = 'count', *staged*: *bool* = *True*)

Returns Dot graph representation of the staged tree.

Parameters

- **edge_info** (*str*) – Optional - Chooses which summary measure to be displayed on edges. Defaults to “count”. Options: [“count”, “prior”, “posterior”, “probability”]
- **staged** (*bool*) – if *True*, returns the coloured staged tree, if *False*, returns the underlying event tree.

Returns

A graphviz Dot representation of the graph.

Return type

pydotplus.Dot

create_figure (*filename*: *Optional*[*str*] = *None*, *edge_info*: *str* = 'count', *staged*: *bool* = *True*) → *Optional*[*Image*]

Draws the coloured staged tree or the underlying event tree for the process described by the dataset.

Parameters

- **filename** (*str*) – Optional - When provided, file is saved to the filename, local to the current working directory. e.g. if filename = “output/event_tree.svg”, the file will be saved to: cwd/output/staged_tree.svg Otherwise, if function is called inside an interactive notebook, image will be displayed in the notebook, even if filename is omitted.
Supports any filetype that graphviz supports. e.g: “staged_tree.png” or “staged_tree.svg” etc.
- **edge_info** (*str*) – Optional - Chooses which summary measure to be displayed on edges. In event trees, only “count” can be displayed, so this can be omitted.
- **staged** (*bool*) – if *True*, returns the coloured staged tree, if *False*, returns the underlying event tree.

Returns

The event tree Image object.

Return type

IPython.display.Image or None

CHAIINEVENTGRAPH

```
class cegpy.ChainEventGraph(staged_tree: Optional[StagedTree] = None, node_prefix: str = 'w', generate:  
                           bool = True)
```

Bases: MultiDiGraph

Representation of a Chain Event Graph.

A Chain Event Graph reduces a staged tree.

The class is an extension of NetworkX MultiDiGraph.

Parameters

- **staged_tree** (*StagedTree*) – A staged tree object where the stages have been calculated.
- **node_prefix** (*str*) – The prefix that is used for the nodes in the Chain Event Graph. Default = “w”
- **generate** (*bool*) – Automatically generate the Chain Event Graph upon creation of the object. Default = True.

```
property sink: str
```

Returns

Sink node name

Return type

str

```
property root: str
```

Returns

Root node name

Return type

str

```
property stages: Mapping[str, Set[str]]
```

Returns

Mapping of stages to constituent nodes.

Return type

Mapping[str, Set[str]]

```
property path_list: List[List[Tuple[str]]]
```

Returns

All the paths through the CEG, as a list of lists of edge tuples.

Return type

List[List[Tuple[str]]]

generate () → None

Identifies the positions i.e. the nodes of the CEG and the edges of the CEG along with their edge labels and edge counts. Here we use the algorithm from our paper with the optimal stopping time.

dot_graph (*edge_info*: str = 'probability') → Dot

Returns Dot graph representation of the CEG. :param *edge_info*: Optional - Chooses which summary measure to be displayed on edges. Defaults to “count”. Options: [“count”, “prior”, “posterior”, “probability”]

Returns

A graphviz Dot representation of the graph.

Return type

pydotplus.Dot

create_figure (*filename*=None, *edge_info*: str = 'probability') → Optional[Image]

Draws the coloured chain event graph for the staged_tree.

Parameters

- **filename** (*str*) – Optional - When provided, file is saved to the filename, local to the current working directory. e.g. if filename = “output/ceg.svg”, the file will be saved to: cwd/output/ceg.svg Otherwise, if function is called inside an interactive notebook, image will be displayed in the notebook, even if filename is omitted.

Supports any filetype that graphviz supports. e.g: “ceg.png” or “ceg.svg” etc.

- **edge_info** (*str*) – Optional - Chooses which summary measure to be displayed on edges. Value can take: “count”, “prior”, “posterior”, “probability”

Returns

The event tree Image object.

Return type

IPython.display.Image or None

CHAINEVENTGRAPHREDUCER

class `cecpy.ChainEventGraphReducer` (*ceg*: `ChainEventGraph`)

Bases: `object`

Reduces Chain Event Graphs given certain and/or uncertain evidence.

Parameters

ceg (`ChainEventGraph`) – Chain event graph object to reduce.

property `certain_edges`: `List[Tuple[str]]`

Returns

A list of all edges of the ChainEventGraph that have been observed.

certain_edges is a list of edge tuples of the form: *[edge_1, edge_2, ... edge_n]*

Each edge tuple takes the form: (*“source_node_name”, “destination_node_name”, “edge_label”*)

property `uncertain_edges`: `List[Tuple[str]]`

Returns

A list of sets of edges of the ChainEventGraph which might have occurred.

uncertain_edges is a list of sets of edge tuples of the form: *[{(a, b, label), (a, c, label)}, {(x, y, label), (x, z, label)}]*

Each edge tuple takes the form: (*“source_node_name”, “destination_node_name”, “edge_label”*)

property `certain_nodes`: `Set[str]`

Returns

A set of all nodes of the ChainEventGraph that have been observed.

certain_nodes is a set of nodes of the form: *{“node_1”, “node_2”, “node_3”, ... “node_n”}*

property `uncertain_nodes`: `List[Set[str]]`

Returns

A list of sets of nodes of the ChainEventGraph where there is uncertainty which of the nodes in each set happened.

uncertain_nodes is a list of sets of nodes of the form: *[{“node_1”, “node_2”, “node_3”, “node_4”, ...}]*

property `paths`: `List[List[Tuple[str]]]`

Returns

A list of all paths through the reduced ChainEventGraph.

property graph: *ChainEventGraph*

Returns

The reduced graph once all evidence has been taken into account.

Return type

ChainEventGraph

clear_all_evidence () → None

Resets the evidence provided.

add_certain_edge (*src: str, dst: str, label: str*) → None

Specify an edge that has been observed.

Parameters

- **src** (*str*) – Edge source node label
- **dst** (*str*) – Edge destination node label
- **label** (*str*) – Label of certain edge

remove_certain_edge (*src: str, dst: str, label: str*) → None

Specify an edge to remove from the certain edges.

Parameters

- **src** (*str*) – Edge source node label
- **dst** (*str*) – Edge destination node label
- **label** (*str*) – Label of certain edge

add_certain_edge_list (*edges: List[Tuple[str]]*) → None

Specify a list of edges that have all been observed.

Parameters

edges (*List[Tuple[str]]*) – List of edge tuples of the form (“src”, “dst”, “label”)

remove_certain_edge_list (*edges: List[Tuple[str]]*) → None

Specify a list of edges that in the certain edge list to remove.

Parameters

edges (*List[Tuple[str]]*) – List of edge tuples of the form (“src”, “dst”, “label”)

add_uncertain_edge_set (*edge_set: Set[Tuple[str]]*) → None

Specify a set of edges where one of the edges has occurred, but you are uncertain of which one it is.

Parameters

edge_set (*Set[Tuple[str]]*) – Set of edge tuples of the form (“src”, “dst”, “label”)

remove_uncertain_edge_set (*edge_set: Set[Tuple[str]]*) → None

Specify a set of edges to remove from the uncertain edges.

Parameters

edge_set (*Set[Tuple[str]]*) – Set of edge tuples of the form (“src”, “dst”, “label”)

add_uncertain_edge_set_list (*edge_sets: List[Set[Tuple[str]]]*) → None

Specify a list of sets of edges where one of the edges has occurred, but you are uncertain of which one it is.

Parameters

edge_set (*List[Set[Tuple[str]]]*) – List of sets of edge tuples of the form (“src”, “dst”, “label”)

remove_uncertain_edge_set_list (*edge_sets: List[Set[Tuple[str]]]*) → None

Specify a list of sets of edges to remove from the evidence list.

Parameters

edge_set (*List[Set[Tuple[str]]]*) – List of sets of edge tuples of the form (“src”, “dst”, “label”)

add_certain_node (*node: str*) → None

Specify a node in the graph that has been observed.

Parameters

node (*str*) – A node label e.g. “w4”

remove_certain_node (*node: str*) → None

Specify a node to be removed from the certain nodes list.

Parameters

node (*str*) – A node label e.g. “w4”

add_certain_node_set (*nodes: Set[str]*) → None

Specify a set of nodes that have been observed.

Parameters

nodes (*Set[str]*) – A set of node labels e.g. {“w4”, “w8”}

remove_certain_node_set (*nodes: Set[str]*) → None

Specify a list of nodes to remove from the list of nodes that have been observed.

Parameters

nodes (*Set[str]*) – A set of node labels e.g. {“w4”, “w8”}

add_uncertain_node_set (*node_set: Set[str]*) → None

Specify a set of nodes where one of the nodes has occurred, but you are uncertain of which one it is.

Parameters

nodes (*Set[str]*) – A set of node labels e.g. {“w4”, “w8”}

remove_uncertain_node_set (*node_set: Set[str]*) → None

Specify a set of nodes to be removed from the uncertain nodes set list.

Parameters

nodes (*Set[str]*) – A set of node labels e.g. {“w4”, “w8”}

add_uncertain_node_set_list (*node_sets: List[Set[str]]*) → None

Specify a list of sets of nodes where in each set, one of the nodes has occurred, but you are uncertain of which one it is.

Parameters

nodes (*List[Set[str]]*) – A collection of sets of uncertain nodes.

remove_uncertain_node_set_list (*node_sets: List[Set[str]]*) → None

Specify a list of sets nodes to remove from the list of uncertain sets of nodes.

Parameters

nodes (*List[Set[str]]*) – A collection of sets of uncertain nodes.

PYTHON MODULE INDEX

C

`cegpy`, [33](#)

A

`add_certain_edge()`
 (*cecpy.ChainEventGraphReducer* *method*),
 44
`add_certain_edge_list()`
 (*cecpy.ChainEventGraphReducer* *method*),
 44
`add_certain_node()`
 (*cecpy.ChainEventGraphReducer* *method*),
 45
`add_certain_node_set()`
 (*cecpy.ChainEventGraphReducer* *method*),
 45
`add_uncertain_edge_set()`
 (*cecpy.ChainEventGraphReducer* *method*),
 44
`add_uncertain_edge_set_list()`
 (*cecpy.ChainEventGraphReducer* *method*),
 44
`add_uncertain_node_set()`
 (*cecpy.ChainEventGraphReducer* *method*),
 45
`add_uncertain_node_set_list()`
 (*cecpy.ChainEventGraphReducer* *method*),
 45
`ahc_output` (*cecpy.StagedTree* property), 38
`alpha` (*cecpy.StagedTree* property), 38

C

`calculate_AHC_transitions()`
 (*cecpy.StagedTree* method), 38
`categories_per_variable` (*cecpy.EventTree* prop-
 erty), 34
`cecpy`
 module, 33
`certain_edges` (*cecpy.ChainEventGraphReducer*
 property), 43
`certain_nodes` (*cecpy.ChainEventGraphReducer*
 property), 43
`ChainEventGraph` (class in *cecpy*), 41
`ChainEventGraphReducer` (class in *cecpy*), 43
`clear_all_evidence()`

 (*cecpy.ChainEventGraphReducer* *method*),
 44
`create_figure()` (*cecpy.ChainEventGraph* *method*),
 42
`create_figure()` (*cecpy.EventTree* *method*), 34
`create_figure()` (*cecpy.StagedTree* *method*), 39

D

`dot_graph()` (*cecpy.ChainEventGraph* *method*), 42
`dot_graph()` (*cecpy.EventTree* *method*), 34
`dot_graph()` (*cecpy.StagedTree* *method*), 39

E

`edge_counts` (*cecpy.EventTree* property), 34
`edge_countset` (*cecpy.StagedTree* property), 38
`EventTree` (class in *cecpy*), 33

G

`generate()` (*cecpy.ChainEventGraph* *method*), 42
`graph` (*cecpy.ChainEventGraphReducer* property), 43

H

`hyperstage` (*cecpy.StagedTree* property), 38

L

`leaves` (*cecpy.EventTree* property), 34

M

`module`
 cecpy, 33

P

`path_list` (*cecpy.ChainEventGraph* property), 41
`paths` (*cecpy.ChainEventGraphReducer* property), 43
`posterior` (*cecpy.StagedTree* property), 38
`posterior_list` (*cecpy.StagedTree* property), 38
`prior` (*cecpy.StagedTree* property), 37
`prior_list` (*cecpy.StagedTree* property), 38

R

`remove_certain_edge()`
 (*cegpy.ChainEventGraphReducer* *method*),
 44

`remove_certain_edge_list()`
 (*cegpy.ChainEventGraphReducer* *method*),
 44

`remove_certain_node()`
 (*cegpy.ChainEventGraphReducer* *method*),
 45

`remove_certain_node_set()`
 (*cegpy.ChainEventGraphReducer* *method*),
 45

`remove_uncertain_edge_set()`
 (*cegpy.ChainEventGraphReducer* *method*),
 44

`remove_uncertain_edge_set_list()`
 (*cegpy.ChainEventGraphReducer* *method*),
 44

`remove_uncertain_node_set()`
 (*cegpy.ChainEventGraphReducer* *method*),
 45

`remove_uncertain_node_set_list()`
 (*cegpy.ChainEventGraphReducer* *method*),
 45

`root` (*cegpy.ChainEventGraph* property), 41

`root` (*cegpy.EventTree* property), 33

S

`sampling_zeros` (*cegpy.EventTree* property), 34

`sink` (*cegpy.ChainEventGraph* property), 41

`situations` (*cegpy.EventTree* property), 34

`StagedTree` (class in *cegpy*), 37

`stages` (*cegpy.ChainEventGraph* property), 41

U

`uncertain_edges` (*cegpy.ChainEventGraphReducer*
 property), 43

`uncertain_nodes` (*cegpy.ChainEventGraphReducer*
 property), 43

V

`variables` (*cegpy.EventTree* property), 33