Contents lists available at ScienceDirect

Knowledge-Based Systems

journal homepage: www.elsevier.com/locate/knosys

cegpy: Modelling with chain event graphs in Python

Gareth Walley [a,1], Aditi Shenvi [b,*,1], Peter Strong [c,d], Katarzyna Kobalczyk [b,e]

[a] *Kenilworth, CV8 1JY, UK*
[b] *Statistics Department, University of Warwick, Coventry, CV4 7AL, UK*
[c] *Centre for Complexity Science, University of Warwick, Coventry, CV4 7AL, UK*
[d] *Alan Turing Institute, London, NW1 2DB, UK*
[e] *Department of Pure Mathematics and Mathematical Statistics, University of Cambridge, Cambridge, CB3 0WB, UK*

A R T I C L E   I N F O

Article history:
Received 25 November 2022
Received in revised form 1 May 2023
Accepted 2 May 2023
Available online 11 May 2023

Keywords:
Chain event graphs
Staged trees
Event trees
Graphical models
Asymmetric processes
Context-specific independence
Python

A B S T R A C T

Chain event graphs (CEGs) are a recent family of probabilistic graphical models that generalise the popular Bayesian networks (BNs) family. Crucially, unlike BNs, a CEG is able to embed, within its graph and its statistical model, asymmetries exhibited by a process. These asymmetries might be in the conditional independence relationships or in the structure of the graph and its underlying event space. Structural asymmetries are common in many domains, and can occur naturally (e.g. a defendant vs prosecutor's version of events) or by design (e.g. a public health intervention). Whilst two CEG packages exist in R for modelling processes with asymmetric conditional independencies, there currently exists no software that allows a user to leverage the theoretical developments of the CEG model family in modelling processes with structural asymmetries. In this paper, we present cegpy: the first Python implementation of CEGs and the first across all languages to support structurally asymmetric processes. cegpy contains an implementation of Bayesian learning and probability propagation algorithms for CEGs. We illustrate the functionality of cegpy using a structurally asymmetric dataset.

1. Introduction

A probabilistic graphical model (PGM) is composed of a statistical model and a graph representing the conditional independence relationships between the defining random variables or events of the underlying model. The graph of a PGM gives a compact visual representation of the factorisation of the joint probability distribution of a statistical model, and provides a way to perform efficient inference using local computations [1]. A key benefit of PGMs is that the gist of its graph can typically be understood by those without formal statistical or mathematical training, thereby facilitating interactions between statisticians, domain experts and decision makers.

Bayesian networks (BNs) [1], currently the most popular family of PGMs, have been successfully applied to a wide range of domains. Notwithstanding the great success of BNs, they do have some shortcomings. In particular, BNs are unable to fully describe processes that exhibit asymmetries either in their conditional independence relationships or in their structure. The former indicates the presence of context-specific conditional independencies which are independence relationships that only hold for certain values of the conditioning variables. The latter refers to the presence of structural missing values, i.e. missing values that have no underlying meaningful value, and/or structural zeros,[2] i.e. observations of a zero frequency for a category of a categorical variable where a non-zero observation is logically restricted. Such asymmetries are common in many real-world processes especially in domains such as medicine, risk analysis, policing, forensics, law, ecology and reliability engineering where processes are best described through an unfolding of events (see Zhang and Poole [2], Boutilier et al. [3],Shenvi et al. [4] and examples in Section 2.1).

Chain event graphs were introduced in Smith and Anderson [5] as an alternative to BNs for asymmetric processes. Indeed, CEGs generalise BNs. They are built from event trees which provide a natural and intuitive framework for describing the unfolding of a process through a sequence of *events*.[3] A CEG is constructed by leveraging the probabilistic symmetries existing within its corresponding event tree to reduce the number of nodes and edges required in the representation of the process. Therefore, CEGs ensure the resultant statistical model and graph are no more complex than they need to be to represent the process. Since their relatively recent introduction, several

* Corresponding author.
E-mail address: aditi.shenvi@gmail.com (A. Shenvi).
[1] Both authors contributed equally to this research.

[2] These are in contrast to sampling zeros that occur due to limitations of data sampling.
[3] An event is an element or a subset of elements of the state space of a variable.

methodological developments have now been made for the CEG family. These include learning algorithms [6–8], probability propagation algorithms [9,10] and a d-separation theorem [11] as well as tools for causal inference [8,12,13] and diagnostics in a CEG [11]. Applications of CEGs have also been explored in: public health and medicine [4,14,15], educational studies [16], asymmetric Bayesian games [17], migration studies [18], and policing [19,20].

Despite the prevalence of asymmetric processes and the proven flexibility offered by CEGs in modelling such processes, CEGs are yet to be widely adopted by applied statisticians. A major hindrance to the wider application of CEG methodologies is the lack of existing software, particularly when it comes to modelling structurally asymmetric processes. There currently exist two R packages for modelling with CEGs, namely `ceg` [21] and `stagedtrees` [22]. However, neither of these supports processes with structural asymmetries. In contrast, for modelling with BNs, there exist several well-developed and maintained softwares such as Netica [23], Weka [24], BARD [25], GeNIe [26], and Hugin [27], as well as packages such as `bnlearn` [28] and `deal` [29] in R; and `BayesPy` [30], `GOBNILP` [31] and `BayeSuites` in Python. In this paper, we present `cegpy`,[4,5] the first Python package for modelling with CEGs and the first package across all languages to support modelling of processes with structural asymmetries.

2. Modelling with chain event graphs

2.1. Asymmetric processes

Before reviewing CEGs, we discuss the generality of asymmetric processes. Asymmetric structures may occur naturally or by design. Below we list several examples of such processes.

- *A public health intervention:* To maximise their effectiveness given limited resources, interventions are designed to offer support to those most at risk or most in need (e.g. annual flu vaccines provided by the UK's National Health Services only to those most at risk).
- *A medical diagnosis process:* Diagnosis involves identifying the patient's ailment based on their background covariates (e.g. age, sex, health history), the sequence of symptoms already exhibited by them, and the additional symptoms that they may exhibit in the future according to the possible diagnoses. Different diseases might have an overlap of symptoms, and further, not all patients suffering from a certain disease exhibit all associated symptoms.
- *A forensic proceeding:* The contrasting explanations for forensic evidence given by a defendant and prosecutor typically result in asymmetric developments in the sequence of events in the two arguments.
- *A policing process:* Depending on the current preparedness of an individual intent on committing a crime, the sequence of preparatory tasks undertaken by the individual can be very varied; e.g. training, help of like-minded criminals, target identification acquisition of resources.
- *A machinery failure process:* Several different sequences of events can lead to faults in machinery, involving different failing components.

Asymmetric processes are common in several domains (see Díez et al. [32] and various examples in Shenvi [10],Collazo et al. [33]). However, a large number of statistical methods and techniques rely on using variables as building blocks in their description of the process and therefore implicitly expect processes to have symmetric structures.

[4] `cegpy` is hosted on https://pypi.org/project/cegpy/; install with pip.

[5] Documentation for `cegpy` is hosted at https://cegpy.readthedocs.io

2.2. Chain event graphs

Like BNs, CEGs are typically used for explanatory modelling and therefore, a key focus is on computing the posterior probabilities of a CEG and revising these in the presence of new information. `cegpy` is designed for such explanatory CEGs. For a discussion of alternative explanatory PGMs for asymmetric processes, see Section 2.2.1. Note here that whilst CEGs can also be used for decision analysis as in Thwaites and Smith [17], this is currently beyond the scope of `cegpy`.

A CEG for a process is constructed from the event tree describing the process which can be elicited from domain experts, data or a combination of both. An event tree is a directed tree graph with a single *root* node which has no parents and a set of *leaves* which have no children. Each non-leaf node (also known as a *situation*) represents the state an individual may be in, and its children represent the possible situations that may follow from this node. A directed edge between a situation and its child is labelled by the event leading to the transition from the situation to its child. With each situation, we can associate a *conditional probability vector* (CPV) which gives the probabilities of transitioning from the situation to its children conditional on an individual being in the state represented by the situation.

An event tree is first transformed into a staged tree that is then transformed into a CEG. A non-technical summary of these transformations is presented below.

- **Event tree to staged tree:** Situations in the event tree whose CPVs are equivalent are said to be in the same *stage* and are assigned the same node colouring to indicate this symmetry. When all the nodes of the event tree are coloured in this way, it becomes a *staged tree*.
- **Staged tree to CEG:** Situations in the staged tree whose rooted subtrees (i.e. the subtree obtained by considering that situation as the root) are isomorphic in the structure- and colour-preserving sense[6] are said to be in the same *position* and are merged into a single node, which retains the colouring of the nodes it merged. All the leaves are merged into a single *sink* node. This turns the staged tree into a CEG.

Example 1 (*Falls Intervention*). Consider a non-pharmacological intervention designed to reduce falls-related injuries and fatalities among the elderly as presented in Eldridge et al. [34]. The intervention aims to enhance assessment, referral pathways, and treatment for individuals aged over 65 years who have a substantial risk of falling, living either in the community or a communal establishment (i.e. care homes, nursing homes and hospitals). Under this intervention, a certain proportion of elderly individuals would be assessed and classified as low or high risk. Those assessed to be at a high risk are referred to a falls clinic for an advanced assessment. All those who are referred, 50% of other high risk individuals, and 10% of low risk individuals go on to receive treatment. It is assumed that those who are not assessed receive neither referral nor treatment. We can model this intervention with a CEG to analyse its effectiveness in reducing falls among the elderly.

An uncoloured version of the tree in Fig. 1(a) gives the event tree describing the above falls intervention. A dataset associated with this intervention would describe the living and assessment situation, risk status, referral and treatment status (where relevant) and fall status for each individual in the sample. Suppose that 50 individuals in our sample lived in a communal establishment, were assessed and are at a high risk of falling, then the

[6] This is equivalent to saying that the two subtrees look identical except for node labels.

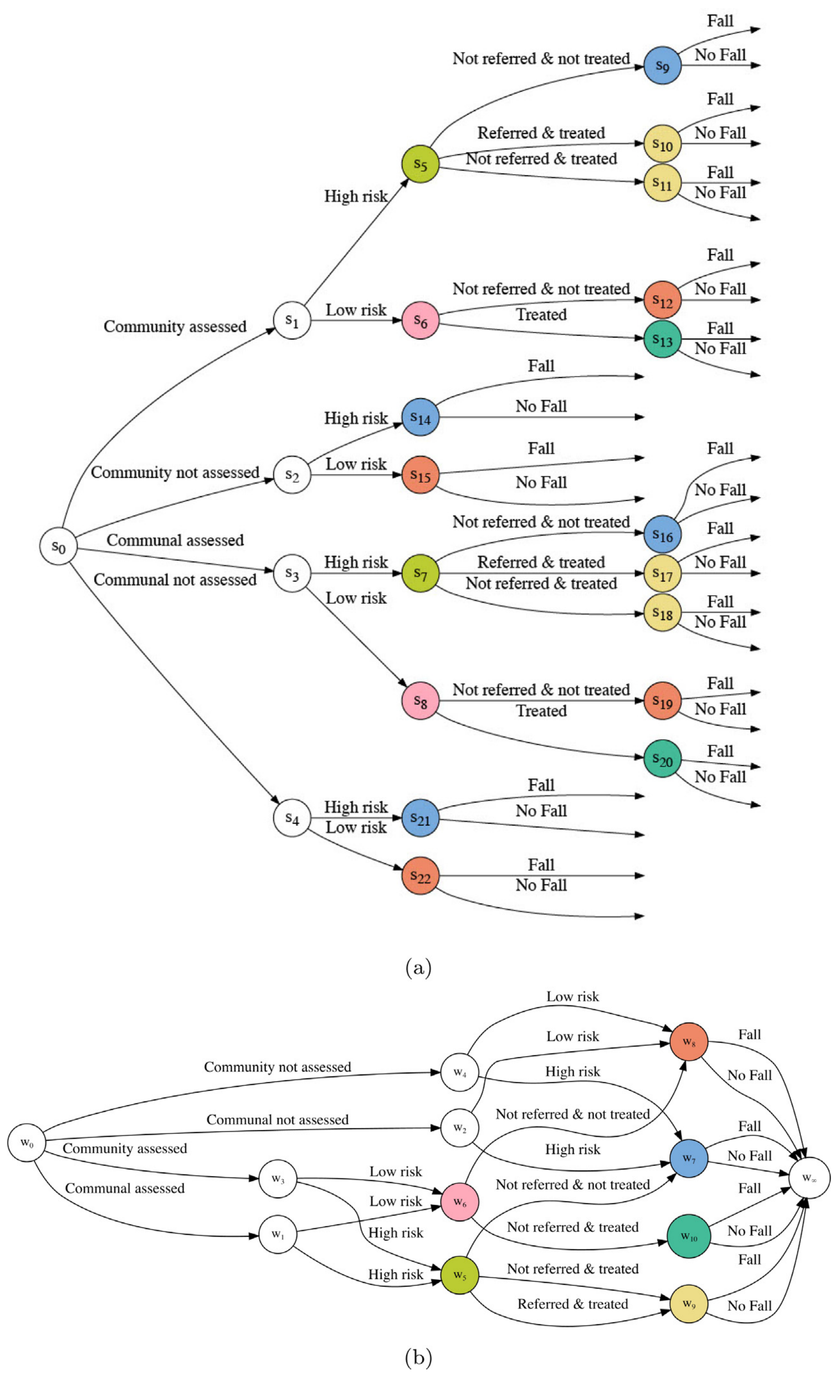


Fig. 1. (a) A hypothesised staged tree and (b) the corresponding CEG for the falls intervention described in Example 1.

edge-count associated with the edge $e_{3,7}$ in Fig. 1(a) would be 50. Further, if the edge-count for edge $e_{3,8}$ was 120, then we can associate with situation s_3 a the vector of edge-counts (50, 120). This vector would be the data used to update the prior for the CPV of situation s_3. To identify the stages in the event tree and to compute the posteriors of the CPVs of the situations in the event tree, a variety of Bayesian learning algorithms can be used. Situations which are in the same stage have equivalent CPVs.

Fig. 1 shows a hypothesised staged tree and its corresponding CEG for the falls intervention. Here, for visual clarity, situations that are in singleton stages or positions are coloured white. Nodes s_5 and s_7 are coloured light green as they are in the same stage. This is interpreted as the probability of being referred and treated (/not being referred but being treated /not being treated) is the same conditional on being in the state represented by s_5 or s_7. Further, these nodes are also in the same position as their rooted subtrees are isomorphic in the colour- and structure-preserving sense. Therefore, s_5 and s_7 are represented by the single node w_5 in the CEG in Fig. 1(b).

In Appendix A in the supplementary material, we describe why a BN is less appropriate than a CEG for modelling the falls intervention.

Formally, let $\mathcal{T}$ denote an event tree with a finite node set $V(\mathcal{T})$ and edge set $E(\mathcal{T})$. Let $L(\mathcal{T})$ and $S(\mathcal{T}) = V(\mathcal{T})\backslash L(\mathcal{T})$ denote the sets of leaves and situations respectively. A directed edge $e \in E(\mathcal{T})$ from nodes s_i to s_j with label l is an ordered triple given by (s_i, s_j, l), which is also denoted by e_{ij} when there is only a single edge from s_i to s_j. Denote by $ch(s_i)$ the children of situation s_i and by $\boldsymbol{\theta}_{s_i} = \{\theta(e_{ij}) | s_j \in ch(s_i)\}$ the CPV associated with s_i where $\theta(e_{ij})$ is the probability of going along edge e_{ij} conditional on being at situation s_i. Let $\Theta_{\mathcal{T}} = \{\boldsymbol{\theta}_{s_i} | s_i \in S(\mathcal{T})\}$.

Definition 2 (*Stage*)**.** In an event tree $\mathcal{T}$, two situations s_i and s_j are said to be in the same stage whenever $\boldsymbol{\theta}_{s_i} = \boldsymbol{\theta}_{s_j}$. Additionally, for $\theta(e_i) = \theta(e_j)$ we require that $e_i = (s_i, \cdot, l)$ and $e_j = (s_j, \cdot, l)$ where edge e_i emanates from s_i and e_j emanates from s_j.

Identification of the stages in the event tree can be done using any suitable learning algorithm (see Section 2.3). Stages enable us to reduce the parameter space of the CEG model.

Definition 3 (*Position*)**.** In a staged tree $\mathcal{S}$, two situations s_i and s_j are said to be in the same position whenever we have $\Theta_{\mathcal{S}_{s_i}} = \Theta_{\mathcal{S}_{s_j}}$ where $\mathcal{S}_{s_i}$ and $\mathcal{S}_{s_j}$ are the coloured subtrees of $\mathcal{S}$ rooted at s_i and s_j respectively.

Denote the collection of positions by $\mathbb{W}$. Situations which are in the same position can be represented by a single node in the graph of the CEG as their complete future unfoldings are identical.

Definition 4 (*Chain Event Graph*)**.** A CEG $\mathcal{C} = (V(\mathcal{C}), E(\mathcal{C}))$ is defined by the triple $(\mathcal{S}, \mathbb{W}, \Theta_{\mathcal{S}})$ with the following properties:

- $V(\mathcal{C}) = R(\mathbb{W}) \cup w_\infty$ where $R(\mathbb{W})$ is the set of situations representing each position set in $\mathbb{W}$, w_∞ is the sink node and for $w \in V(\mathcal{C})$, $\boldsymbol{\theta}_{\mathcal{C}}(w) = \boldsymbol{\theta}_{\mathcal{S}}(w)$. Nodes in $R(\mathbb{W})$ retain their stage colouring.
- Situations in $\mathcal{S}$ belonging to the same position set in $\mathbb{W}$ are contracted into their representative node contained in $R(\mathbb{W})$. This node contraction merges multiple edges between two nodes into a single edge only if they share the same edge label.
- Leaves of $\mathcal{S}$ are contracted into sink node w_∞.

Staged trees and CEGs that embed context-specific independencies but not structural asymmetries are said to be *stratified*. Those that additionally embed structurally asymmetries are said to be *non-stratified*. The staged tree and CEG in Example 1 are non-stratified.

2.2.1. Alternative PGMs for asymmetric processes

CEGs provide an alternative to BNs for processes with context-specific conditional independencies and/or asymmetric structures. With major modifications (typically tree-based), a BN can represent context-specific conditional independencies. It can also be used to model processes with structural zeros; albeit with redundant parameters and some loss of graphical representation. However, generally, there is no easy way to embed structural missing values within a BN model.

PGMs such as the probability decision graph (PDG) [35] and acyclic probabilistic finite automata (APFA) [36] share some similarities with CEGs when it comes to modelling asymmetric processes. While PDGs can represent asymmetric structures and some context-specific independencies, there are some symmetric conditional independencies – that can be represented by BNs and CEGs – that they cannot represent.[7] On the other hand, APFAs can encode context-specific independencies but not asymmetric structures.

For a detailed comparison of CEGs and other asymmetric PGMs used for explanatory modelling, see Chapter 2 of Shenvi [10].

[7] For instance, no PDG can represent the independence structure implied by a BN with variables $\boldsymbol{\mathcal{X}} = \{X_1, X_2, X_3, X_4\}$ and directed edges $(X_1, X_2), (X_1, X_3), (X_2, X_4)$ and (X_3, X_4) [35]

2.3. Bayesian learning

Since the transformation of a given staged tree model into its corresponding CEG is deterministic, a CEG $\mathcal{C}$ is uniquely defined by its staged tree $\mathcal{S}$ and the parameters over its staged tree, $\Theta_{\mathcal{S}}$ [37]. Hence, learning or model selection in CEGs is equivalent to identifying the sets of stages in the event tree to obtain the staged tree. The two main approaches to learning the stages in an event tree are (i) the agglomerative hierarchical clustering (AHC) algorithm [6] and (ii) a brute-force approach using dynamic programming [7,8]. Both of these are score-based algorithms that aim to maximise a chosen score function, typically the log marginal likelihood of the model.

Within cegpy, we implemented the AHC algorithm as it is computationally efficient for moderate-sized event trees – which we envision will be what the package will be used for – and at present, it is the most popular approach for applications involving CEGs. The AHC algorithm does not rely on structural symmetry and is directly applicable to the non-stratified class.

The AHC algorithm is a greedy, bottom-up hierarchical clustering algorithm. It begins with each situation in the event tree being in its own singleton stage. Thereafter, at each step, it merges the two stages that give the highest improvement in terms of the log marginal likelihood score. The algorithm terminates when the score can no longer be improved by merging two stages. The technical details of the AHC and its associated pseudo-code are presented in Appendix B in the supplementary material.

In cegpy, the input dataset is used for creating the event tree and for identifying the edge-counts along each of its edges. The event tree, edge-counts and user-defined or default priors over the CPVs and stage structure are taken as inputs by the AHC algorithm to learn the staged tree/CEG and their parameters.

2.4. Probability propagation

Given new data or observations about an individual, the posterior distributions of the CPVs in a CEG can be updated under the Bayesian framework. Under a naïve approach, this could be accomplished by obtaining the full joint distribution of the model and then revising the probabilities using Bayes' Rule conditional on the new observations. However, this approach is inefficient and does not scale well. Probability propagation[8] refers to updating these posterior distributions through local computations directly on the graph of the CEG. The probability propagation algorithm for CEGs was given by Thwaites et al. [9].

Within the literature, new observations are also known as "evidence". For CEGs, evidence for a given individual is in terms of the state occupied or the event experienced by an individual and this directly corresponds to a node or edge in the graph of the CEG. Evidence can be further classified as "positive" or "negative" and as "certain" or "uncertain". We explain these classifications with the example below.

Example 5 (*Falls Intervention (continued)*)**.** Consider the CEG in Fig. 1(b). Suppose that we observe an individual who has been assessed to be at a high risk of falling. This observation or evidence can be stated in various distinct yet equivalent ways:

1. **As positive and certain evidence:** This individual is in node w_5 in the CEG.
2. **As positive and uncertain evidence:** To occupy w_5, they necessarily must have come along either edge (w_1, w_5, 'High risk') or edge (w_3, w_5, 'High risk') but it is not possible to say which one with absolute certainty.

[8] Probability propagation in CEGs is similar to the case of BNs.

3. **As negative and certain evidence:** This evidence can also be stated as observing that this individual was assessed (i.e. they are in either w_5 or w_6) and is not at a low risk of falling (i.e. they are not in w_6).
4. **As negative and uncertain evidence:** This evidence can also be stated as observing that this individual was assessed, so must have gone along one of (w_1, w_5, 'High risk'), (w_3, w_5, 'High risk'), (w_1, w_6, 'Low risk') or (w_3, w_6, 'Low risk'), and that they are not at a low risk of falling, so could not have gone along the latter two edges.

Any negative evidence (i.e. of *not* being in a particular state or *not* experiencing a particular event) can always be written as positive evidence which may be certain or uncertain depending on how it is formulated as shown above. Further, the above evidence makes the nodes w_2, w_4, w_6, w_8, w_{10} and any edges going into or out of these nodes redundant and can be safely excluded from the CEG. The resultant simplified CEG is given in Fig. 6. The propagation algorithm then redistributes the probability mass through local computations on the remaining nodes and edges.

As demonstrated above, evidence leads to simplification of the CEG under consideration. The simplified CEG obtained by excluding redundant nodes and edges is called the *reduced CEG* for that evidence. Not all evidence is compatible with being used by the probability propagation algorithm. Only *intrinsic* evidence is compatible. Evidence is said to be intrinsic if the graph of its reduced CEG contains no more information than the evidence suggests and it preserves the conditional independence relations in the original CEG. This condition is trivially met when the evidence is defined in terms of nodes and edges [33]. In Appendix C of the supplementary material, we provide an example of non-intrinsic evidence that is defined in terms of root-to-sink paths of the CEG.

In the cegpy package, once a CEG is learned, the user can specify their evidence and then run the propagation algorithm to update the graph and posteriors in the CEG. The evidence can only be specified in terms of nodes and edges, and so is always intrinsic. The evidence supplied must be positive but can certain or uncertain. The technical details of the propagation algorithm are given in Appendix D in the supplementary material.

3. The cegpy package

cegpy is the first Python implementation of CEGs, the first across all languages to support structurally asymmetric processes and the first to support probability propagation. It is developed using a Bayesian framework which provides a structured way to incorporate prior knowledge, to update the posterior as more data is received, and to perform modelling with a combination of expert knowledge and data when working on problems with limited data.

3.1. Python implementation

cegpy is built in Python, and makes use of the open source packages pandas, NetworkX, and GraphViz; see metadata in Appendix E in the supplementary material. cegpy is designed to harness the object-oriented functionalities of Python. The class diagram in Fig. 2 shows the object-oriented inheritance structure of the various classes in the package. The EventTree, StagedTree, ChainEventGraph and ChainEventGraphReducer (corresponding to the reduced CEG after incorporating the evidence, see Section 2.4) are all object classes. From the inheritance structure, we can see that, for instance, the StagedTree class inherits the features of the EventTree class and extends it. The EventTree class and ChainEventGraph class inherit from the MultiDiGraph (directed multi-graph) class of NetworkX which is a well-developed and thoroughly tested package for studying graphs and networks in Python.

The EventTree class is the entry-point for data into the package. It converts the input data into an event tree so that a learning algorithm can be run on it to then create a StagedTree. To construct the event tree, the EventTree object scans each row of a column-based data set, and counts all the unique sequences of events, i.e. paths, contained in it. This data is stored in a Python dictionary, where each key in the dictionary represents an edge (expressed as a path from the root up to that edge) in the tree, and maps to the number of times that path has been observed in the dataset which corresponds to the number of transitions along that edge.

For example, consider the example dataset for the falls intervention in Table 1. This is transformed into a dictionary like so:

```
edges = {
   ("Community Assessed", ): 2,
   ("Community Not Assessed", ): 2,
   ("Community Assessed", "High Risk"): 1,
   ("Community Assessed", "Low Risk"): 1,
   ("Community Not Assessed", "High Risk"): 1,
   ("Community Not Assessed", "Low Risk"): 1,
   ("Community Assessed", "High Risk", "Referred and Treated"): 1,
   ("Community Assessed", "Low Risk", "Don't Fall"): 1,
   ("Community Not Assessed", "High Risk", "Not Referred and Not
   ↪  Treated"): 1,
   ("Community Not Assessed", "Low Risk", "Not Referred and Not
   ↪  Treated"): 1,
   ("Community Assessed", "High Risk", "Referred and Treated",
   ↪  "Fall"): 1,
   ("Community Not Assessed", "High Risk", "Not Referred and Not
   ↪  Treated", "Fall"): 1,
   ("Community Not Assessed", "Low Risk", "Not Referred and Not
   ↪  Treated", "Fall"): 1,
}
```

By design, the paths are ordered alphabetically, which enables consistent node-naming, even when the rows in the dataset are reordered. Once the paths are determined, the edges dictionary is used to create a NetworkX MultiDiGraph object, which is then used as the data structure representation of the event tree.

As the StagedTree class inherits from the EventTree class, the former is simply a special case of the latter. A StagedTree object determines which nodes of the EventTree are in the same stage and applies a colour scheme to them to show the distinct stages. Within cegpy, staged trees and CEGs have singleton stages coloured in white, and leaves and sink nodes in light-grey. The cegpy package contains an implementation of the AHC algorithm described in Section 2.3 for identifying the stages from the EventTree. Under a Bayesian framework, the algorithm is initiated with a prior specification for each situation (i.e. each initial singleton stage) in the EventTree. With the default settings in cegpy, this is done using the approach of Collazo et al. [33] whereby an imaginary sample size for the root (called *alpha*) is set and this is then propagated uniformly through the EventTree. The default imaginary sample size is given by the maximum number of edges emanating out of any situation in the EventTree. Further, cegpy also supports specification of priors for the colouring of the StagedTree (and thus, the structure of the ChainEventGraph) by indicating which situations are allowed to be merged together. This is done by specifying a *hyperstage* [19] 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 cegpy, all situations that have the same number of outgoing edges and equivalent set of edge labels are in the same

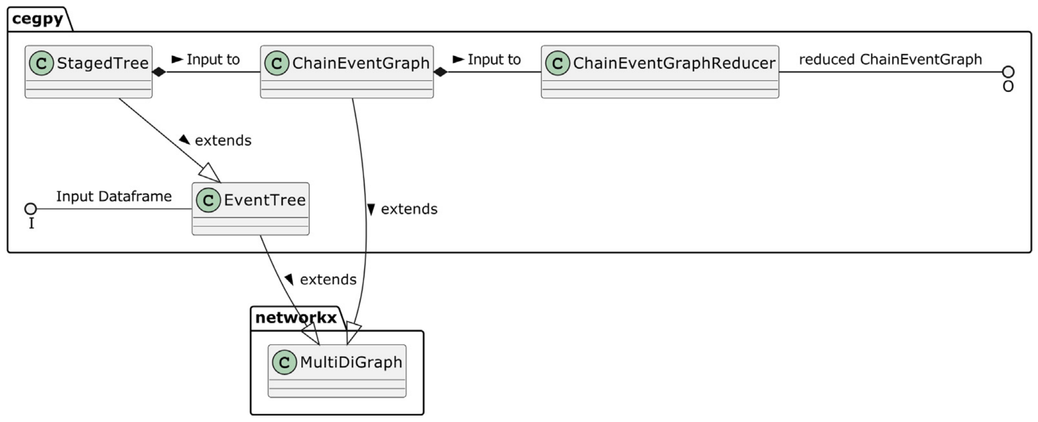


Fig. 2. Class diagram of the Python classes encoded in cegpy.

Table 1
An example dataset illustrating how the falls intervention data is stored.

Housing assessment	Risk	Treatment	Fall
Community not assessed	Low risk	Not referred and not treated	Fall
Community not assessed	High risk	Not referred and not treated	Fall
Community assessed	Low risk	–	Don't fall
Community assessed	High risk	Referred and treated	Fall

Table 2
A comparison of the three packages available for modelling with CEGs.

Package	Language	CEG class	Learning algorithms	Propagation
ceg	R	Stratified only	Bayesian	✗
stagedtrees	R	Stratified only	Frequentist	✗
cegpy	Python	Stratified and non-stratified	Bayesian	✓

set within the hyperstage. Note that users can specify their own prior, imaginary sample size or hyperstage.

The AHC implementation is run within the StagedTree object and it is parallelised for efficiency. Once the stages have been identified, a ChainEventGraph object can be created by passing the StagedTree as input. The ChainEventGraph object merges the nodes that are in the same position. To do this, it uses the transformation algorithm described in Shenvi and Smith [37]. cegpy also includes an implementation of the probability propagation algorithm described in Section 2.4. In order to propagate evidence through a CEG, the ChainEventGraphReducer object is first instantiated from the ChainEventGraph object. Evidence can be specified to this object in terms of the nodes and edges as certain or uncertain evidence. Once all the evidence has been specified, the reduced CEG graph is created and the corresponding posteriors are updated within a new ChainEventGraph object.

3.2. Related work

There are two previous packages that can learn and visualise CEGs from data. The R package ceg [21] was the first package to do so. This package implements the AHC for Bayesian learning in CEGs. In 2021, the R package stagedtrees [22] was released, which included several score-based and clustering-based algorithms for non-Bayesian learning in CEGs such as hill-climbing, backward hill-climbing and k-means. Whilst both these packages are able to represent context-specific conditional independencies, neither are able to fully represent asymmetric structures, i.e. non-stratified staged trees and CEGs. Therefore, neither of these packages can be used to model processes with asymmetrical unfolding of events such as those described in Section 2.1; see examples in Appendix F in the supplementary material. cegpy fills this gap (see Table 2).

The path-based approach of cegpy is in contrast to the column-based approach of the other CEG packages, ceg and stagedtrees. In the former approach, all the data is associated with edges of the event tree and it corresponds to using events as the building blocks of the model. It can routinely handle non-stratified CEGs. On the other hand, the column-based approach associates the data to the variables of the model and to their corresponding state spaces. This approach makes it extremely difficult to model non-stratified CEGs. For instance, the structural missing values associated with the *Treatment* variable for individuals who are not assessed cannot be recorded easily within the column-based approach. An alternative is to create a new category for the *Treatment* variable called "Not referred & not treated" which then renders the counts on the other categories of this variable into structural zeros leading to redundancies in the parameters and loss of representation.

Finally, Python's object-oriented architecture lends itself well to extensibility. The functionality and CEG classes supported by cegpy can be easily built upon using inheritance. For example, to create a new class of CEGs in cegpy with arbitrary holding time distributions (such as in Barclay et al. [38]), a new TemporalEventTree class can be created which inherits from the EventTree class and extends it to handle the holding times in the input dataset. Similarly, a TemporalStagedTree class can be created such that it inherits the initialisation and functions from the TemporalEventTree class as well as just the functions from the StagedTree class.

4. An illustrative example

In this section, we illustrate the key functionalities of the cegpy package through the analysis of a structurally asymmetric process. We revisit the public health intervention to reduce falls-related injuries and fatalities among the elderly as described in

Example 1. We use the synthetic dataset simulated by Shenvi et al. [4] and provided with the supplementary material. Note that illustrations and guidance for the full range of functionalities supported by `cegpy` can be found at https://cegpy.readthedocs.io.

4.1. Creating the event tree

The Falls dataset provides information concerning adults over the age of 65, and includes the following four categorical variables:

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

Recall from the description in Example 1 that this process has structural asymmetries. None of the individuals assessed to be low risk are referred to a falls clinic and thus, for this group, the edge count associated with the 'Referred & Treated' category is a structural zero. Moreover, for individuals who are not assessed, their responses are structurally missing for the referral and treatment variable.

Observe that since `cegpy` constructs the event tree by creating a dictionary of the paths in the input dataset, there is no need to specify structural zeros as they do not occur in the dataset. On the other hand, we encode structural missing values in the dataset as NaNs. For example, a NaN value in the column relating to the referral and treatment variable is interpreted by `cegpy` as a structural missing value.

```
from cegpy import EventTree
import pandas as pd

df = pd.read_excel('Falls_Data.xlsx')
print(df.head(5))
```

```
output:
     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
```

The event tree can be constructed from the falls dataset by initialising an EventTree object using the code given below and as shown in Fig. 3.

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

Note here that 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. To demarcate the difference, 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 object.

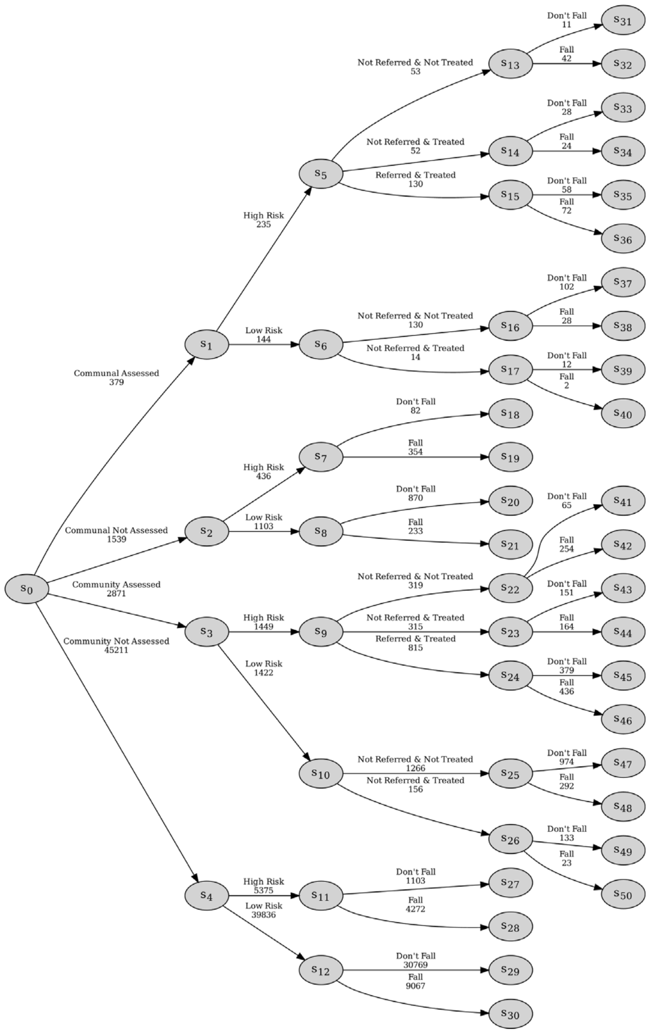


Fig. 3. Event tree output for the falls dataset. Edge labels include the edge-counts.

4.2. Creating the staged tree

To create a staged tree for the falls intervention, we initialise a StagedTree object with our dataset as the input. Note that it is not necessary to first initialise an EventTree object. To create a staged tree, we must first identify the stages in the event tree. We do this by running the AHC algorithm within the StagedTree object. The code and output below show the default settings of the hyperstage, alpha (imaginary sample size at the root, see Section 3.1) and prior for the falls dataset. The priors and posteriors are saved as fractions to maintain accuracy through the iterative calculations.

```
from cegpy import StagedTree
st = StagedTree(df)
print('default hyperstage:',st._create_default_hyperstage())
print('default alpha:',st._calculate_default_alpha())
print('default prior:',
      st._create_default_prior(st._calculate_default_alpha()))
```

```
Output:
default hyperstage: [['s0'], ['s1', 's2', 's3', 's4'], ['s5',
↪  's9'],
['s6', 's10'],['s7', 's8', 's11', 's12', 's13', 's14', 's15',
↪  's16',
's17', 's22', 's23', 's24', 's25', 's26']]
default alpha: 4
default prior: [[Fraction(1, 1), Fraction(1, 1), Fraction(1, 1),
Fraction(1, 1)],[Fraction(1, 2), Fraction(1, 2)], [Fraction(1,
↪  2),
Fraction(1, 2)],[Fraction(1, 2), Fraction(1, 2)], [Fraction(1,
↪  2),
Fraction(1, 2)],[Fraction(1, 6), Fraction(1, 6), Fraction(1,
↪  6)],
[Fraction(1, 4),Fraction(1, 4)], [Fraction(1, 4), Fraction(1,
↪  4)],
[Fraction(1, 4),Fraction(1, 4)], [Fraction(1, 6), Fraction(1,
↪  6),
Fraction(1, 6)],[Fraction(1, 4), Fraction(1, 4)], [Fraction(1,
↪  4),
Fraction(1, 4)],[Fraction(1, 4), Fraction(1, 4)], [Fraction(1,
↪  12),
Fraction(1, 12)],[Fraction(1, 12), Fraction(1, 12)],
↪  [Fraction(1, 12),
Fraction(1, 12)],[Fraction(1, 8), Fraction(1, 8)], [Fraction(1,
↪  8),
Fraction(1, 8)],[Fraction(1, 12), Fraction(1, 12)],
↪  [Fraction(1, 12),
Fraction(1, 12)],[Fraction(1, 12), Fraction(1, 12)],
↪  [Fraction(1, 8),
Fraction(1, 8)],[Fraction(1, 8), Fraction(1, 8)]]
```

We now run the AHC algorithm with the above default settings using the code below and generate the associated staged tree shown in Fig. 4. Additionally, a user can specify a list of colours or palette to be used in the staged tree and its corresponding CEG. In this example, we have used a colourblind-friendly palette as shown by the *colours* list below.

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

4.3. Creating the chain event graph

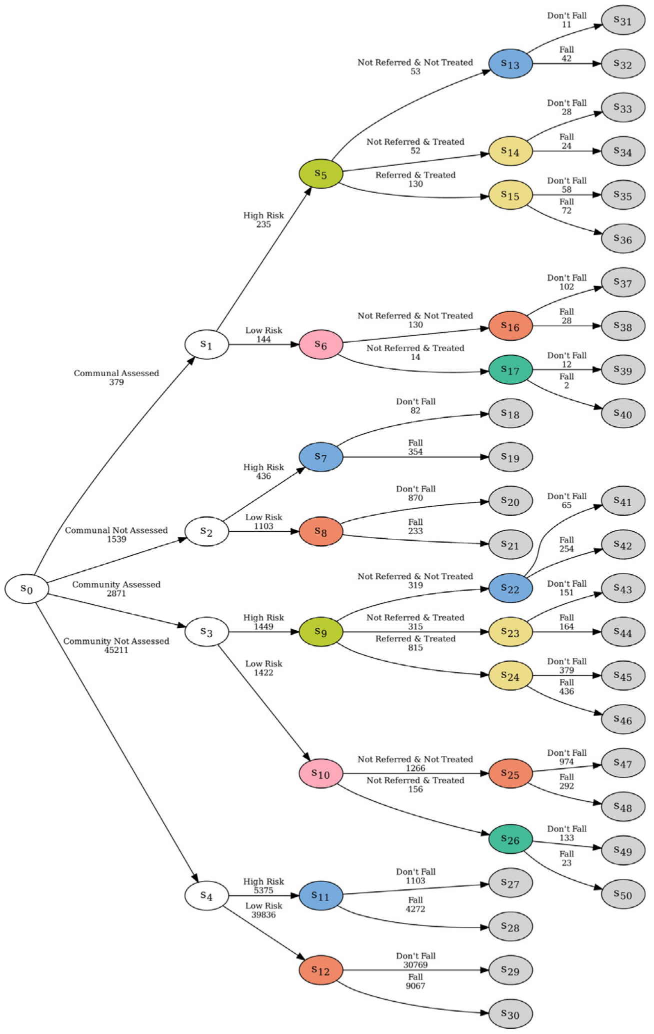
Once the stages have been identified by running the AHC algorithm on the StagedTree object, we can initialise a ChainEventGraph object that takes the StagedTree object as an input. Using this StagedTree object, the ChainEventGraph object can generate the CEG figure using the code below and as shown in Fig. 5.

```
from cegpy import ChainEventGraph
ceg = ChainEventGraph(st)
ceg.create_figure()
```

4.4. Probability propagation on the chain event graph

Finally, we demonstrate how cegpy can be used for probability propagation on a given CEG after observing some evidence associated with it. Suppose that we have observed an assessed, high risk individual. This is equivalent to observing the node w_5 in the CEG in Fig. 5 with certainty. The CPVs associated with the CEG can be updated in light of this evidence by first initialising a ChainEventGraphReducer object with the CEG as the input and then adding the certain evidence as shown in the code below. The graph and means of the updated CPVs are given in the reduced CEG in Fig. 6. We can see that based on this observation, the probability that the observed individual is from a communal establishment is updated from 0.04 (sum of communal assessed and communal not assessed) to 0.15.

```
from cegpy import ChainEventGraphReducer
rceg = ChainEventGraphReducer(ceg)
rceg.add_certain_node('w5')
rceg.graph.create_figure()
```



Don't Fall 11
Fall 42
Not Referred & Not Treated 53
Don't Fall 28
Not Referred & Treated 52
Fall 24
Referred & Treated 130
Don't Fall 58
Fall 72
High Risk 235
Don't Fall 102
Not Referred & Not Treated 130
Fall 28
Low Risk 144
Not Referred & Treated 14
Don't Fall 12
Fall 2
Communal Assessed 379
Don't Fall 82
Fall 354
High Risk 436
Don't Fall 870
Low Risk 1103
Fall 233
Communal Not Assessed 1539
Don't Fall 65
Fall 254
Not Referred & Not Treated 319
Don't Fall 151
Community Assessed 2871
High Risk 1449
Not Referred & Treated 315
Fall 164
Referred & Treated 815
Community Not Assessed 45211
Low Risk 1422
Don't Fall 379
Fall 436
Not Referred & Not Treated 1266
Don't Fall 974
Not Referred & Treated 156
Fall 292
Don't Fall 133
Fall 23
High Risk 5375
Low Risk 39836
Don't Fall 1103
Fall 4272
Don't Fall 30769
Fall 9067
s_0 s_1 s_2 s_3 s_4 s_5 s_6 s_7 s_8 s_9 s_{10} s_{11} s_{12} s_{13} s_{14} s_{15} s_{16} s_{17} s_{18} s_{19} s_{20} s_{21} s_{22} s_{23} s_{24} s_{25} s_{26} s_{27} s_{28} s_{29} s_{30} s_{31} s_{32} s_{33} s_{34} s_{35} s_{36} s_{37} s_{38} s_{39} s_{40} s_{41} s_{42} s_{43} s_{44} s_{45} s_{46} s_{47} s_{48} s_{49} s_{50}

Fig. 4. Staged tree output for the falls dataset with default priors.

We can also use cegpy to propagate uncertain evidence. Suppose now that we observe individuals who had been treated but they still suffered a fall. These individuals must have passed through one of the following sequences of edges in the CEG in Fig. 5: (i) (w_5, w_9, 'Not Referred & Treated') and (w_9, w_∞, 'Fall'); (ii) (w_5, w_9, 'Referred & Treated') and (w_9, w_∞, 'Fall'); or (iii) (w_6, w_{10}, 'Not Referred & Treated') and (w_{10}, w_∞, 'Fall'). To simplify, this is equivalent to having uncertain evidence about nodes w_9 and w_{10}, and about the edges (w_9, w_∞, 'Fall') and (w_{10}, w_∞, 'Fall'). As earlier, we first initialise a ChainEventGraphReducer object with the CEG as the input and add both sets of uncertain evidence as shown in the code below, and thus obtain the updated graph and CPVs shown in Fig. 7.

```
from cegpy import ChainEventGraphReducer
rceg = ChainEventGraphReducer(ceg)
rceg.add_uncertain_node_set({"w9", "w10"})
rceg.add_uncertain_edge_set_list([{('w9',ceg.sink, 'Fall'),
                              ('w10', ceg.sink, 'Fall')}])
rceg.graph.create_figure()
```

5. Discussion

cegpy is an open-source Python package that facilitates modelling with staged trees and CEGs, providing functionality for

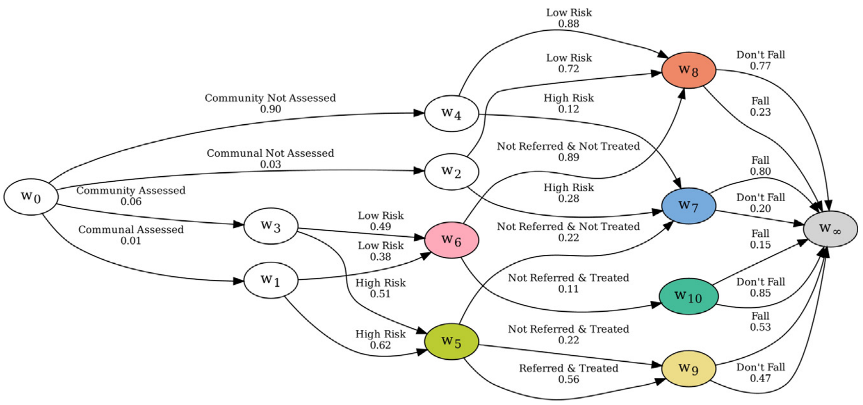


Fig. 5. CEG output for the falls dataset. Edge labels show means of the CPVs.

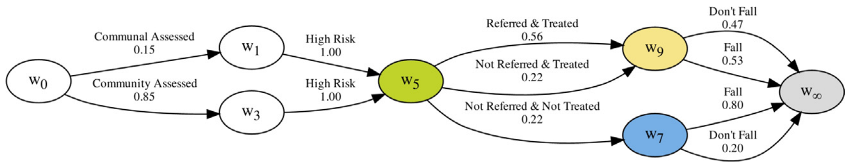


Fig. 6. CEG output for the falls dataset after propagating the observation of node w_5, i.e. individuals who are assessed and are high risk.

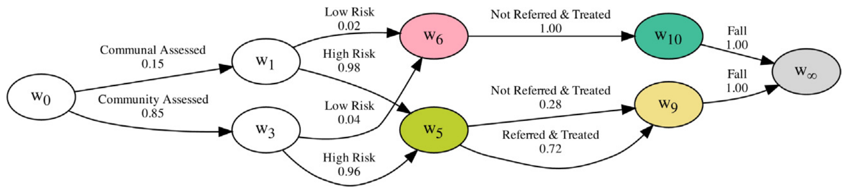


Fig. 7. CEG output for the falls dataset after propagating the uncertain evidence over the nodes: w_9, w_{10} and over the edges: (w_9, w_∞, Fall), (w_{10}, w_∞, Fall).

Bayesian learning and probability propagation. This package is the first implementation of staged trees and CEGs in Python, and, unlike previous implementations in R that focus only on the stratified class, cegpy's functionality extends to the non-stratified class. Further, it is the first package that provides support for probability propagation. Therefore, cegpy can support users with categorical data to create models of processes with structural asymmetries, which can be analysed to understand complex dependence structures. We discuss below a few avenues for greatly enhancing the current functionality of cegpy[9]

In the current version of cegpy, we have focused on Bayesian methods. However, it is straightforward to implement classical methods such as those in the stagedtree package and we plan to do this in a future version. Moreover, currently cegpy only provides support for the AHC algorithm. Other existing Bayesian learning algorithms have considerable drawbacks: dynamic programming [7] is computationally infeasible for all but the smallest of data sets. Further research is needed to explore computationally efficient Bayesian learning techniques for CEGs. Strong and Smith [39]'s work on Bayesian model averaging using a modification of the AHC algorithm has been implemented as an extension to cegpy (see https://github.com/peterrhysstrong/cegpy_BMA) and we plan to make this available in a future version of the package.

As described in Section 3.2, cegpy uses a path-based approach to construct the event tree. Thereby, sampling zeros paths are not automatically filled in for unobserved combinations of variables. Currently, users must add these paths manually. Filling in of sampling zero paths can be automated by assuming that the tree is stratified. This can be added as an argument for the EventTree class to create these paths at the point of initialisation.

Finally, for the purposes of expert elicitation, it would be extremely useful to enable a user to directly specify an event tree, staged tree or CEG structure – with colouring and possibly, with parameters – in the cegpy package. Of course, learning algorithms cannot be used due to the absence of data but it would be beneficial for visualisation and evidence propagation. We are currently looking into adding this functionality by directly importing graphs specified using the DOT language used by GraphViz.

CRediT authorship contribution statement

Gareth Walley: Conceptualization, Software, Validation, Writing – original draft, Writing – review & editing. **Aditi Shenvi:** Conceptualization, Methodology, Project administration, Software, Validation, Writing – original draft, Writing – review & editing. **Peter Strong:** Software, Validation, Writing – original draft. **Katarzyna Kobalczyk:** Software, Validation, Visualization, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

[9] Contributions on https://github.com/g-walley/cegpy are always welcome.

Data availability

We have included the data (titled 'Falls_Data.xlsx') as a supplementary file. The code used in the analysis of this dataset is provided in the main article itself

Acknowledgments

PS was supported by the EPSRC, United Kingdom and MRC, United Kingdom grant EP/L015374/1. We thank Emma Southall for directing us to `NetworkX` in the early days of development.

Appendix. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.knosys.2023.110615.

References

[1] J. Pearl, Causality, Cambridge University Press, 2009.
[2] N.L. Zhang, D. Poole, On the role of context-specific independence in probabilistic inference, in: Proc. of the 16th Intern. Jt. Conf. on Artif. Intell., Vol. 2, 1999, pp. 1288–1293.
[3] C. Boutilier, N. Friedman, M. Goldszmidt, D. Koller, Context-specific independence in Bayesian networks, in: Proc. of the 12th Intern. Conf. on Uncertain. in Artif. Intell., 1996, pp. 115–123.
[4] A. Shenvi, J.Q. Smith, R. Walton, S. Eldridge, Modelling with non-stratified chain event graphs, in: Intern. Conf. on Bayesian Stat. in Action, Springer, 2018, pp. 155–163.
[5] J.Q. Smith, P.E. Anderson, Conditional independence and chain event graphs, Artif. Intell. 172 (1) (2008) 42–68.
[6] G. Freeman, J.Q. Smith, Bayesian MAP model selection of chain event graphs, J. Multivar. Anal. 102 (7) (2011) 1152–1165.
[7] T. Silander, T.-Y. Leong, A dynamic programming algorithm for learning chain event graphs, in: Intern. Conf. on Discov. Sci., Springer, 2013, pp. 201–216.
[8] R.G. Cowell, J.Q. Smith, Causal discovery through MAP selection of stratified chain event graphs, Electron. J. Stat. 8 (1) (2014) 965–997.
[9] P.A. Thwaites, J.Q. Smith, R.G. Cowell, Propagation using chain event graphs, in: Proc. of the 24th Intern. Conf. on Uncertain. in Artif. Intell., 2008, pp. 546–553.
[10] A. Shenvi, Non-stratified chain event graphs: Dynamic variants, inference and applications (Ph.D. thesis), The University of Warwick, 2021.
[11] R.L. Wilkerson, Customising Structure of Graphical Models (Ph.D. thesis), The University of Warwick, 2020.
[12] P.A. Thwaites, Causal identifiability via chain event graphs, Artif. Intell. 195 (2013) 291–315.
[13] X. Yu, J.Q. Smith, Causal algebras on chain event graphs with informed missingness for system failure, Entropy 23 (10) (2021) 1308.
[14] L.M. Barclay, J.L. Hutton, J.Q. Smith, Refining a Bayesian network using a chain event graph, Int. J. Approx. Reason. 54 (9) (2013) 1300–1309.
[15] C. Keeble, P.A. Thwaites, S. Barber, G. Law, P. Baxter, Adaptation of chain event graphs for use with case-control studies in epidemiology, Int. J. Biostat. 13 (2) (2017).
[16] G. Freeman, J.Q. Smith, Dynamic staged trees for discrete multivariate time series: Forecasting, model selection and causal analysis, Bayesian Anal. 6 (2) (2011) 279–305.
[17] P.A. Thwaites, J.Q. Smith, A new method for tackling asymmetric decision problems, Int. J. Approx. Reason. 88 (2017) 624–639.
[18] P. Strong, A. McAlpine, J.Q. Smith, A Bayesian analysis of migration pathways using chain event graphs of agent based models, 2021, arXiv:2111.04368.
[19] R. Collazo, The Dynamic Chain Event Graph (Ph.D. thesis), The University of Warwick, 2017.
[20] F.O. Bunnin, J.Q. Smith, A Bayesian hierarchical model for criminal investigations, Bayesian Anal. 16 (1) (2019) 1–30.
[21] R. Collazo, P. Taranti, Ceg: Chain event graph, 2017, R pkg Version 0.1.0.
[22] F. Carli, M. Leonelli, E. Riccomagno, G. Varando, The R package stagedtrees for structural learning of stratified staged trees, J. Stat. Soft. 102 (2022) 1–30.
[23] Norsys Software Corp, Netica, 2020, Version 6.08.
[24] F. Eibe, M.A. Hall, I.H. Witten, The WEKA Workbench, Morgan Kaufmann Publishers, 2016.
[25] E.P. Nyberg, A.E. Nicholson, K.B. Korb, M. Wybrow, I. Zukerman, S. Mascaro, S. Thakur, A. Oshni Alvandi, J. Riley, R. Pearson, et al., BARD: a structured technique for group elicitation of Bayesian networks to support analytic reasoning, Risk Anal. 42 (6) (2022) 1155–1178.
[26] L. BayesFusion, GeNIe Modeler, 2022, Version 4.0.
[27] HuginExpert, HUGIN, 2022, Version 9.2.
[28] M. Scutari, Learning Bayesian networks with the bnlearn R Package, J. Stat. Soft. 35 (3) (2010) 1–22.
[29] S.G. Bottcher, C. Dethlefsen, Deal: Learning Bayesian networks with mixed variables, 2018, R pkg Version 1.2-39.
[30] J. Luttinen, BayesPy: Variational Bayesian inference in python, J. Mach. Learn. Res. 17 (1) (2016) 1419–1424.
[31] J. Cussens, GOBNILP: Learning Bayesian network structure with integer programming, in: Intern. Conf. on Probab. Graph. Models, PMLR, 2020 pp. 605–608.
[32] F.J. Díez, M. Luque, I. Bermejo, Decision analysis networks, Int. J. Approx. Reason. 96 (2018) 1–17.
[33] R. Collazo, C. Görgen, J.Q. Smith, Chain Event Graphs, CRC Press, 2018.
[34] S. Eldridge, A. Spencer, C. Cryer, S. Parsons, M. Underwood, G. Feder, Why modelling a complex intervention is an important precursor to trial design, J. Health Serv. Res. Policy 10 (3) (2005) 133–142.
[35] M. Jaeger, Probabilistic decision graphs – combining verification and AI techniques for probabilistic inference, Int. J. Uncertain. Fuzziness Knowl.-Based Syst. 12 (2004) 19–42.
[36] D. Edwards, S. Ankinakatte, Context-specific graphical models for discrete longitudinal data, Stat. Model. 15 (4) (2015) 301–325.
[37] A. Shenvi, J.Q. Smith, Constructing a chain event graph from a staged tree, in: Proc. of the 10th Intern. Conf. on Probab. Graph. Models, 2020.
[38] L.M. Barclay, R.A. Collazo, J.Q. Smith, P.A. Thwaites, A.E. Nicholson, The dynamic chain event graph, Electron. J. Stat. 9 (2) (2015) 2130–2169.
[39] P. Strong, J.Q. Smith, Bayesian model averaging of chain event graphs for robust explanatory modelling, in: Proc. of the 11th Intern. Conf. on Probab. Graph. Models, 2022, pp. 61–72.