Assignment 3

Due: 26. November 2021, 10:15 CET

General information: This assignment should be completed in groups of **three to four people**. Submit your assignment via the corresponding LernraumPlus activity. You can only upload one file, therefore make sure to place all your files required to run your solution (.py, .ipynb, .pdf etc) into **one archive** and just upload that archive.

Python exercise information: You are only allowed to use the external libraries that we provide or explicitly mention on the assignment sheet (or that are already within the provided skeleton files). Your solutions' functionality will all be tested against automated tests before being reviewed further. **You need to ensure** that your solution can directly be **imported as a Python3 module** for automatic tests and at least contains the name of the skeleton file. The easiest way to achieve this, is to directly develop your solution in the assignment3.py file and submit that. The LernraumPlus contains information about the test environment. You will also find an example test-file alongside the skeleton file that you can use to ensure that your solution works in the evaluation environment.

Exercise information: In this assignment, you will build upon the graph class from the first assignment and implement functions required for performing independence checks for variables on directed and undirected graphs. You will find a reference graph implementation in the *ccbase* package in the *assignment3.zip* alongside the skeleton file. The graph implementation can be found in *ccbase/networks.py*, while the used node implementation is located in *ccbase/nodes.py*. The *docs* folder further contains an automatically generated documentation of the provided modules based on the docstrings. The *assignment3.py* file contains the skeletons for the functions that you should implement for this assignment. For any function that can take either a Node object or a Node's name as string as an argument, you only need to ensure that your solution works with one or the other, although you can also implement it in such a way that works with both. You are free to use your own graph class from the first assignment instead, however, incorrect behavior of the assignment's tasks due to bugs in your graph class may still lead to point reductions.

Exercise 1: (3 Points)

Causal structures like *forks, chains* and *colliders* can be used in a Bayesian network to test for conditional independence given evidence. Implement the following methods:

Task 1: (1 Point) *find_forks(dg)*: Returns a list of nodes that are forks in the directed graph *dg*.

Task 2: (1 Point) *find_chains(dg)*: Returns a list of nodes that are chains in the directed graph *dg*.

Task 3: (1 Point) *find_collider(dg)*: Returns a list of nodes that are collider in the directed graph *dg*.

Exercise 2: (4 Points)

As mentioned in the lecture, two different directed graphs can make the same (conditional) independence statements. Implement the following functions in order to check if two graphs are *Markov equivalent*:

Task 1: (2 Points) *find_immoralities(graph)*: Returns all immoralities (i.e. two nodes with the same child but no direct connection) of the given graph.

Task 2: (1 Point) *same_skeleton(graph1, graph2)*: Returns true if the two graphs have the same skeleton as one another.

Task 3: (1 Point) *markov_equivalent(graph1, graph2)*: Returns true if the two graphs are Markov equivalent and false otherwise.

Independence Tests The 4th and 5th lectures introduce two different graphical tests for independence in graphical networks for belief representation. The first one is called d-separation and is more strongly connected to directed (Bayesian) networks. The 2nd is a general graphical test, also applicable to other networks, such as Markov Networks.

Exercise 3: (2 Points)

Implement the function *get_paths(graph, node_x, node_y)* which computes all *undirected* paths (where each node on the path is only visited once) between node X and node Y in the given (directed or un-directed) graph and returns these as a list of lists.

Exercise 4: (5 Points)

Implement the functions required to perform a (conditional) independence check using *d-separation*:

Task 1: (1 Point) Implement *is_collider(dg, node, path)* which returns *True* only if the given node is a collider with respect to the given (undirected) path within the directed graph.

Task 2: (2 Points) Implement the function *is_path_open(dg, path, nodes_z)* that returns *True* only if the given path is *open* (according to d-separation) when conditioned on a (potentially empty) set of variables **Z**.

Task 3: (1 Point) Implement the function *unblocked_path_exists(dg, node_x, node_y, nodes_z)* which checks (and returns *True*) if there exists an unblocked undirected path between two variables X and Y, given a (potentially empty) set of variables **Z**.

Task 4: (1 Point) Implement *check_independence(dg, nodes_x, nodes_y, nodes_z)* which tests if two *sets* of variables **X** and **Y** are conditionally independent given a (potentially empty) variable set **Z**.

Exercise 5: (6 Points)

Now implement the following functions for the general test. You may want to re-use some of the functions you implemented for the last exercise.

Task 1: (2 Points) Implement the *make_ancestral_graph(graph, nodes)* function which takes a (directed or undirected) *graph* and sets of nodes and returns the corresponding ancestral graph for these nodes.

Task 2: (1 Point) Implement the *make_moral_graph(graph)* function which takes a directed (ancestral) graph and returns its (undirected) moral graph.

Task 3: (1 Point) Implement the *separation(graph, nodes_z)* function which seperates all links from nodes in the set *nodes_z* within the (undirected) graph *g*.

Task 4: (2 Points) Implement *check_independence_general(graph, nodes_x, nodes_y, nodes_z)* which again tests if two *sets* of variables **X** and **Y** are conditionally independent given a (potentially empty) variable set **Z**. Contrary to the d-separation version above, this should work for both directed and un-directed graphs using the general graphical test.

You should pay attention to the provided docstrings as they further specify the required behavior of the different functions.
Of course, you are allowed to add any auxiliary functions you find helpful for your implementation.
Hint: You may want to use the functions of earlier tasks when solving later ones. You could hard code solutions for some of these functions for simple examples, if you want to test later ones. Additionally, the automatic tests for each task will also use correct versions of earlier tasks.