Documentation for Transducer Groups Interface Python module

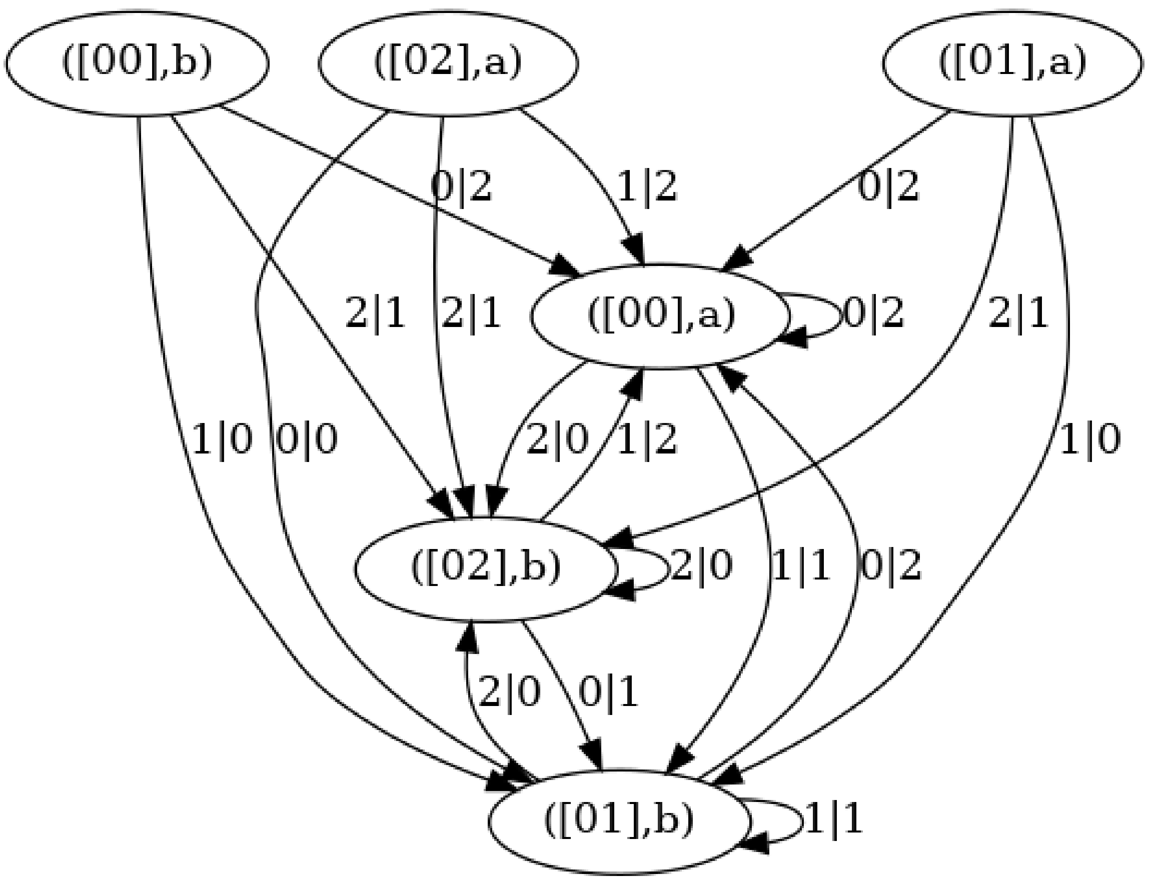
Elliott Cawtheray
University of St Andrews
October 2020

Abstract

The 'Transducer Groups Interface' is a Python module for working with automata and transducers, especially those relating to the transducer monoids and groups relating to automorphism groups of the full shift, as discussed in the papers by Bleak, Cameron and Olukoya.

There is a particular emphasis on making this software accessible, readable, and easily used by all. To this end, the software functions as a command line interface that the user can interact with to create automata and transducers, and play with them as they choose. Transducers may be created interactively, visualised using Python module pygraphviz, and saved and loaded to/from the user's machine in a variety of formats.

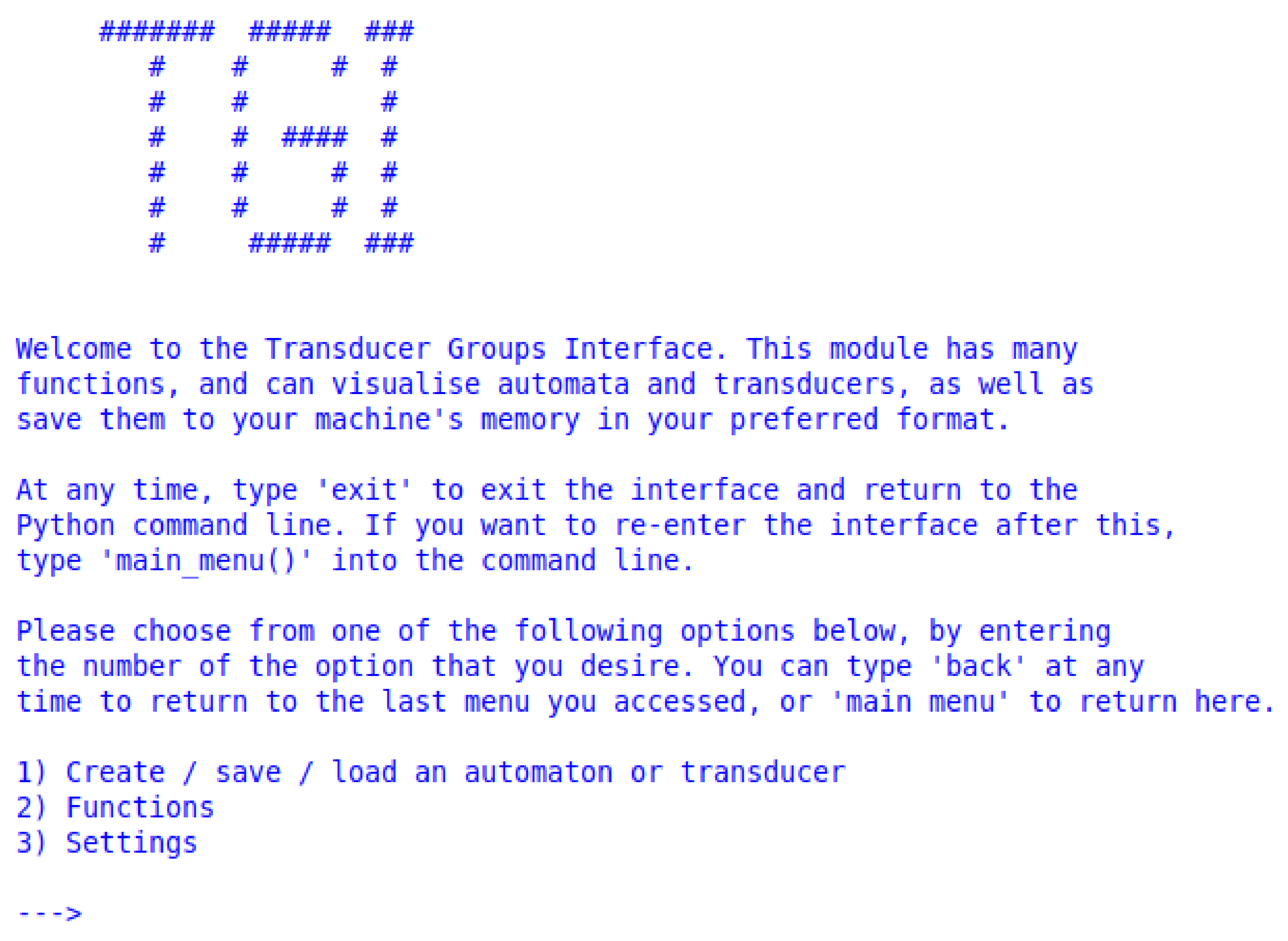
This software was created during an undergraduate summer project, supervised by Dr Bleak at the University of St Andrews and funded by an EMS Student Research Bursary in Mathematics and Statistics.



An image generated using the software, of a 6-state transducer over the alphabet $\{0, 1, 2\}$, which arises from Step 9 of the Decomposition Algorithm of [1], as the transducer product of an element T of $\mathcal{H}_3 \cong \mathrm{Aut}(X_3^{\mathbb{N}}, \sigma_3)$ and a transducer constructed from an automorphism of the underlying graph of T's range automaton.

1 How the interface works

The software loads up on the 'main menu', pictured below.



```
#######  #####  ###
   #    #     #  #
   #    #        #
   #    #  ####  #
   #    #     #  #
   #    #     #  #
   #     #####  ###


Welcome to the Transducer Groups Interface. This module has many
functions, and can visualise automata and transducers, as well as
save them to your machine's memory in your preferred format.

At any time, type 'exit' to exit the interface and return to the
Python command line. If you want to re-enter the interface after this,
type 'main_menu()' into the command line.

Please choose from one of the following options below, by entering
the number of the option that you desire. You can type 'back' at any
time to return to the last menu you accessed, or 'main menu' to return here.

1) Create / save / load an automaton or transducer
2) Functions
3) Settings

--->
```

From there, the menus are navigated by typing the number of the desired option. At any time, typing 'back' brings the interface back to the previous menu, typing 'main menu' brings the interface back to the main menu, and typing 'exit' exits the interface and returns the user to the usual Python command line, and from there typing 'main_menu()' re-opens the interface.

Note that any commands to be typed into the Python command line will not be accepted when inside the interface. You must instead first type 'exit' to exit the interface. Any variables saved, e.g. a created transducer, within the interface will persist after 'exit' is typed, so that it is possible to exit the interface for more freedom with the automata/transducers that were made within the interface.

Automata and transducers that are created by or loaded into the interface are stored as variables in the edgeList format, though can be in loaded in from files or saved to files as 2 other formats (see Formats section of this document).

Outputs for asynchronous transducers are as lists of letters, while outputs for synchronous transducers are simply letters.

2 Functions within the interface

The below lists the functionality that can be accessed when inside the interface, with exposition on each as well as the function name that each has, so that this can be called in the Python command line for a handy shortcut.

For documentation on the underlying functions that are utilised in this module, see the ”Functions for use in the Python command line” section of this document.

2.1 Creating automata and transducers interactively

An interactive, dialog-based function to create automata and transducers. The function asks whether you want an automaton or a transducer, then for the size of the alphabet and the size of the state set (note: on input n to these prompts, the alphabet/state set will be the set $\{0, 1, ..., n-1\}$), and finally the function will ask for where the automaton / transducer goes from each state after reading each letter. If creating a transducer, outputs will also be asked for. In this case, the transition and output must be inputted in the following format: ”state, word”. That is, the number of the state transitioned to, and the associated output word. For this input, any whitespace is ignored, but a comma must separate the state and word. Finally, the function asks what variable name is to be given to the automaton/transducer in the current Python session.

This function can be accessed from the main menu by typing ’1’ and then ’1’, or by typing create_transducer_interface() into the Python command line.

2.2 Giving outputs to automata

Takes an automaton that is currently saved as a variable in the Python session, asks for an output for each possible transition, and generates the transducer that this gives.

This function can be accessed from the main menu by typing ’1’ and then ’2’, or by typing give_outputs_interface() into the Python command line.

2.3 Forgetting outputs of a transducer

Takes a transducer that is currently saved as a variable in the Python session, and generates the underlying automaton.

This function can be accessed from the main menu by typing ’1’ and then ’3’, or by typing forget_outputs_interface() into the Python command line.

2.4 Saving automata and transducers to the directory

Takes an automaton or transducer that is currently saved as a variable in the Python session and saves it as a .txt file to the directory, in any of the formats listed in the Formats section of this document.

This function can be accessed from the main menu by typing ’1’ and then ’4’, or by typing save_transducer_interface() into the Python command line.

2.5 Loading automata and transducers from a directory

Takes an automaton or transducer that is currently saved as a .txt file to the directory, in any of the formats listed in the Formats section of this document, and saves it as a variable in the Python session with a name of the user’s choice.

This function can be accessed from the main menu by typing '1' and then '5', or by typing load_transducer_interface() into the Python command line.

2.6 Draw automata and transducers

Takes an automaton/transducer that is currently saved as a variable in the Python session, and generates a .png image file in the directory that is a picture of the automaton/transducer. The pygraphviz Python module is utilised for this purpose; the automaton/transducer is converted into a pygraphviz labeled directed graph, and then the pygraphviz function 'draw' is called. For transducers, the labels of the edges are "input letter—output word", with the vertical line symbol "|" separating the two, and with the output word a list of letters unless it is a one-letter word.

By default, this function uses the graphviz layout 'dot' when generating an image. This can be changed in the settings (see section "The settings menu" of this document).

This function can be accessed from the main menu by typing '2' and then '1', or by typing generate_image_interface() into the Python command line.

2.7 Transitions and outputs

Takes an automaton/transducer that is currently saved as a variable in the Python session, and asks for a start state and an input word. The interface then gives the final state of the automaton/transducer on reading the input word, as well as the output word in the transducer case.

This function can be accessed from the main menu by typing '2' and then '2', or by typing transitions_outputs_interface() into the Python command line.

2.8 Transducer product

Takes two transducers, over the same alphabet, that are currently saved as variables in the Python session, and generates their transducer product as a Python variable with a name of the user's choice.

This function can be accessed from the main menu by typing '2' and then '3', or by typing product_interface() into the Python command line.

2.9 Find forcing words for a state

Takes a strongly synchronizing automaton/transducer that is currently saved as a variable in the Python session, and asks for a state and an integer that is to be the length of the returned words. The interface then gives all the words that force the state.

This function can be accessed from the main menu by typing '2' and then '4', or by typing forcing_words_interface() into the Python command line.

2.10 Generate the dual

Takes a synchronous transducer that is currently saved as a variable in the Python session, and asks which level of the dual is desired. The interface then generates that level dual, and saves it as a variable with a name of the user's choice.

This function can be accessed from the main menu by typing '2' and then '5', or by typing dual_interface() into the Python command line.

2.11 Generate de Bruijn automata

Generates a de Bruijn automaton and saves it as a variable with a name of the user's choice. The interface asks for the n value, and then the m value, so that the de Bruijn graph $G(n, m)$ is generated, with alphabet $X_n = \{0, 1, ..., n-1\}$ and state set X_n^m.

This function can be accessed from the main menu by typing '2' and then '6', or by typing de_bruijn_interface() into the Python command line.

2.12 Construct an automorphism of an automaton's graph

Takes an automaton that is saved as a variable in the Python session, and allows the user to build an automorphism of the underlying digraph of the automaton, by asking where each vertex and edge is mapped to. It is then checked for being a valid graph automorphism, and saved in the format that this module uses for graph automorphisms, as described below.

The graph automorphism is required to have a very specific format. It must be a 2-tuple (V, E), where V and E are dicts that represent mappings from either the vertex set or the edge set to themselves. Vertices that are being mapped in V must be all the states of the automaton. Edges that are being mapped in E must be in the form of the 3-tuple (p, x, q), where p is the vertex the edge is coming from, q is the vertex the edge is going to, and x is the label of the edge.

This function can be accessed from the main menu by typing '2' and then '7', or by typing create_graph_aut_interface() into the Python command line.

2.13 Check if a map from an automaton's graph to itself is an automorphism

Takes an automaton that is currently saved as a variable in the Python session, and a map from the automaton's graph to itself, and tells the user if the maps meets the criteria to be a graph automorphism.

The graph automorphism is required to have a very specific format. It must be a 2-tuple (V, E), where V and E are dicts that represent mappings from either the vertex set or the edge set to themselves. Vertices that are being mapped in V must be all the states of the automaton. Edges that are being mapped in E must be in the form of the 3-tuple (p, x, q), where p is the vertex the edge is coming from, q is the vertex the edge is going to, and x is the label of the edge.

This function can be accessed from the main menu by typing '2' and then '8', or by typing check_graph_aut_interface() into the Python command line.

2.14 Construct the transducer $H(A, \phi)$

Takes an automaton A that is currently saved as a variable in the Python session, and an automorphism ϕ of the automaton's graph, and generates the transducer $H(A, \phi)$, as in [1], by 'gluing' the automaton to itself along the automorphism in order to obtain outputs.

The graph automorphism is required to have a very specific format. It must be a 2-tuple (V, E), where V and E are dicts that represent mappings from either the vertex set or the edge set to themselves. Vertices that are being mapped in V must be all the states of the automaton. Edges that are being mapped in E must be in the form of the 3-tuple (p, x, q), where p is the vertex the edge is coming from, q is the vertex the edge is going to, and x is the label of the edge.

This function can be accessed from the main menu by typing '2' and then '9', or by typing H_A_phi_interface() into the Python command line.

2.15 Partition states into ω-equivalence classes

Takes a transducer that is currently saved as a variable in the Python session, and prints a list of lists that represents a partition of the transducer's states into ω-equivalence classes. For example, if the transducer has state set $\{0, 1, 2\}$, and the states 0 and 1 are ω-equivalent, then the interface will print the list [['0', '1'], ['2']].

This function can be accessed from the main menu by typing '2' and then '10', or by typing omega_partition_interface() into the Python command line.

2.16 Weakly minimal check

Takes a transducer that is currently saved as a variable in the Python session, and tells the user if the transducer is weakly minimal or not. If not, the interface asks if the user wants to see which states are ω-equivalent.

This function can be accessed from the main menu by typing '2' and then '11', or by typing check_weakly_min_interface() into the Python command line.

2.17 Identify ω-equivalent states

Takes a transducer that is currently saved as a variable in the Python session, and generates a transducer that is ω-equivalent to the input and is weakly minimal, by identifying all ω-equivalent states with each other.

This function can be accessed from the main menu by typing '2' and then '12', or by typing equiv_states_interface() into the Python command line.

2.18 Terms in the synchronizing sequence

Takes an automaton that is currently saved as a variable in the Python session, and an integer 'k', and generates the k^{th} term in the synchronizing sequence of the automaton, which is itself an automaton, and saves it as a variable in the Python session with a name of the user's choice. If a transducer is given to the interface, the synchronizing sequence will be that of the transducer's underlying automaton.

This function can be accessed from the main menu by typing '2' and then '13', or by typing synch_seq_interface() into the Python command line.

2.19 Strongly synchronizing check

Takes an automaton or transducer that is currently saved as a variable in the Python session, and tells the user whether or not it is strongly synchronizing.

This function can be accessed from the main menu by typing '2' and then '14', or by typing check_strong_synch_interface() into the Python command line.

2.20 Minimal synchronizing level

Takes an automaton or transducer that is currently saved as a variable in the Python session, and tells the user the minimal integer 'k' such that the automaton/transducer is synchronizing at level k. If the automaton/transducer is not strongly synchronizing, the interface will instead inform the user of this.

This function can be accessed from the main menu by typing '2' and then '15', or by typing synch_level_interface() into the Python command line.

2.21 Check if an automaton or transducer is core

Takes a strongly synchronizing automaton or transducer that is saved as a variable in the Python session, and tells the user whether or not it is equal to its core.

This function can be accessed from the main menu by typing '2' and then '16', or by typing check_core_interface() into the Python command line.

2.22 Taking the core

Takes an automaton or transducer that is currently saved as a variable in the Python session, generates the automaon/transducer which is its core, and saves it as a variable with a name of the user's choosing. If the automaton/transducer is not strongly synchronizing, the interface will instead inform the user of this.

This function can be accessed from the main menu by typing '2' and then '17', or by typing core_interface() into the Python command line.

2.23 Check if a transducer is synchronous

Takes a transducer that is currently saved as a variable in the Python session, and tells the user whether or not it is synchronous.

This function can be accessed from the main menu by typing '2' and then '18', or by typing check_synchronous_interface() into the Python command line.

2.24 Check if a synchronous transducer is invertible

Takes a synchronous transducer that is currently saved as a variable in the Python session, and tells the user whether or not it is invertible (in the automata-theoretic sense).

This function can be accessed from the main menu by typing '2' and then '19', or by typing check_automata_inv_interface() into the Python command line.

2.25 Generate automata-theoretic inverse

Takes a synchronous, invertible transducer that is currently saved as a variable in the Python session, and generates its automata-theoretic inverse, by swapping the input and output labels on the edges.

This function can be accessed from the main menu by typing '2' and then '20', or by typing invert_synchronous_interface() into the Python command line.

2.26 Check if a synchronous, invertible transducer is bi-synchronizing

Takes a synchronous, invertible transducer that is currently saved as a variable in the Python session, and tells the user whether or not it is bi-synchronizing, i.e. strongly synchronizing with strongly synchronizing inverse.

This function can be accessed from the main menu by typing '2' and then '21', or by typing check_bisynch_interface() into the Python command line.

2.27 Check if a transducer is in $\mathcal{H}_n$

Takes a transducer that is saved as a variable in the Python session, and tells the user whether or not it is in the transducer group $\mathcal{H}_n \cong \mathrm{Aut}(X_n^{\mathbb{N}}, \sigma_n)$. More accurately, since $\mathcal{H}_n$ consists of ω-equivalence classes of transducers, the user is told whether or not the transducer is a representative (i.e. an element) of one of the ω-equivalence classes belonging to $\mathcal{H}_n$. Since weakly minimal representatives are the most natural to work with, the user is also told if such a transducer is a weakly minimal representative or not.

This function can be accessed from the main menu by typing '2' and then '22', or by typing check_in_Hn_interface() into the Python command line.

2.28 The $\mathcal{H}_n$ decomposition algorithm

Takes a transducer that is saved as a variable in the Python session, and is a representative of an element of $\mathcal{H}_n$, and runs the decomposition algorithm of [1] on it, generating a list of finite order elements of $\mathcal{H}_n$ whose product in $\mathcal{H}_n$ is the inputted transducer. The interface asks if the user would like to save this list as a Python variable, and also if the user would like to generate a folder consisting of .txt files of the finite order elements, and finally if the user would like to generate a folder consisting of .png images of these transducers.

This function can be accessed from the main menu by typing '2' and then '23', or by typing decomp_interface() into the Python command line.

2.29 Multiplying in $\mathcal{H}_n$

Takes two transducers that are elements of $\mathcal{H}_n$ and are saved as variables in the Python session, and returns their product by taking their transducer product, taking the core of this, and identifying ω-equivalent states.

This function can be accessed from the main menu by typing '2' and then '24', or by typing Hn_product_interface() into the Python command line.

2.30 The settings menu

2.30.1 Change the directory

Allows the user to change the current working directory, so that automata and transducers may be saved, loaded, and have images of them generated in a directory of the user's choosing.

This function can be accessed from the main menu by typing '3' and then '1', or by typing directory_interface() into the Python command line.

2.30.2 Changing graphviz layout

Allows the user to choose between the different layouts that pygraphviz uses to generate images of automata and transducers. Upon testing, the layouts that I found to be functioning were neato, dot, twopi, circo and sfdp. For more information on these, see the relevant sections of https://graphviz.gitlab.io/pdf/dot.1.pdf.

The default layout is 'dot'. By changing this through this menu, a .txt file is created that saves the user's choice for future reference. If this file is deleted, the layout will again default to 'dot'.

This function can be accessed from the main menu by typing '3' and then '2', or by typing layout_interface() into the Python command line.

3 Functions for use in the Python command line

The interface functions all make use of the following functions. Using these functions in the Python command line allows for much more freedom, and a more powerful experience especially when used in tow with e.g. the 'create_transducer()' and 'generate_image()' functions.

3.1 get_state_list(transducer)

Takes as argument an automaton or transducer saved as a variable in Python, and returns a list of its states, ordered lexicographically.

3.2 get_alphabet_list(transducer)

Takes as argument an automaton or transducer saved as a variable in Python, and returns a list of its states, ordered lexicographically.

3.3 transducer_product(transducer1, transducer2)

Takes as argument two transducers T and U, over the same alphabet, that are saved as variables in Python, and returns their transducer product $T * U$.

3.4 transition_function(transducer, inputWord, currentNode)

Returns the final state of a transducer (that is saved as a variable in the Python session) on input 'inputWord' (of any finite length), starting at state 'currentNode'.

3.5 output_function(transducer, inputWord, currentNode)

Returns the output word of a transducer (that is saved as a variable in the Python session) on input 'inputWord' (of any finite length), starting at state 'currentNode'.

3.6 all_words_of_length_k(alphabet, length)

Takes as input a list 'alphabet' and an integer 'length', and returns a list of all the words that are the concatenation of 'length'-number of elements of 'alphabet'.

This function relies on the itertools function 'product'.

3.7 forcing_words_for_state(transducer, state, wordlength)

Takes as input a transducer saved as a variable in the Python session, a string 'state' that is a state of 'transducer', and an integer 'wordlength'. Returns all the words of length 'wordlength' which, when fed into the transducer initialised at 'state', will end at 'state'.

Note that if the transducer is strongly synchronizing at level 'wordlength', then this will return the list of words which force 'state'.

3.8 rename_states_forcing_words(transducer, synchLevel)

Takes as input a transducer saved as a variable in the Python session and an integer 'synchLevel', and returns same the transducer with each state renamed to have the set of 'synchLevel'-length words which force it as its name.

3.9 dual_level_k(transducer, k)

Takes as input a synchronous transducer saved as a variable in the Python session, and an integer 'k', and returns the level 'k' dual of the transducer.

3.10 pairwise_eq_relation_to_partition(eqRelation)

Takes as input a list consisting of length 2 lists, which represents a pairwise-defined equivalence relation, and returns the associated partition as a list of lists

3.11 omega_equivalence_partition(transducer)

Takes as input a transducer that is saved as a variable in the Python session, and returns the partition of its states into ω-equivalence classes, as a list of lists.

3.12 combine_equivalent_states(transducer)

Takes as input a transducer that is saved as a variable in the Python session, and returns a transducer that is ω-equivalent to the input and is weakly minimal, by identifying all ω-equivalent states with each other.

3.13 transducer_to_automaton(transducer)

Takes as input a transducer saved as a variable in the Python session, and returns the underlying automaton (i.e, the transducer with outputs ignored).

3.14 de_bruijn_automaton(n,m)

Takes as input two integers 'n', 'm', and returns the de Bruijn automaton $G(n, m)$ in the edgeList format

3.15 folded_de_bruijn(deBruijnAutomaton, partition)

Takes as input a de Bruijn automaton saved as a variable in the Python session, and a partition of its states (as a list/tuple of lists/tuples) which defines a folding of the automaton, and returns the folded automaton.

3.16 draw_graph(transducer, filename)

Takes an automaton, transducer or graph that is either saved as a variable in the Python session or saved as a .txt file in the directory, and a string 'filename', and generates a .png file in the directory that will display an image of the automaton/transducer/graph with the name "filename.png".

Relies on the pygraphviz function 'draw'. The layout of the graph in the image is set by default to be the 'dot' layout (see https://graphviz.gitlab.io/pdf/dot.1.pdf). This can be changed to other layouts in the settings of the interface (see Section 2.30.2).

3.17 synchronizing_sequence(automaton, term)

Takes an automaton that is saved as a variable in the Python session, and an integer 'term', and return the 'term'$^{\text{th}}$ term of the synchronizing sequence of the automaton.

3.18 synch_seq_identified_states(automaton, term)

Takes an automaton that is saved as a variable in the Python session, and an integer 'term', and returns a dictionary whose keys are the states of 'automaton'. Upon looking up a state of 'automaton', the dictionary returns the state of the 'term'th term of the synchronizing sequence of 'automaton' with which the looked-up state has been identified into.

3.19 IsStronglySynchronizing(transducer)

Takes an automaton or transducer that is saved as a variable in the Python session, and returns 'True' if it is strongly synchronizing (and 'False' if it is not).

3.20 min_synch_level(transducer)

Takes an automaton or transducer that is saved as a variable in the Python session, and returns the minimal integer 'k' such that the automaton/transducer is synchronizing at level k (if the automaton/transducer is not strongly synchronizing, ValueError is raised).

3.21 take_core(transducer)

Takes an automaton or transducer that is saved as a variable in the Python session, and returns its core (if the automaton/transducer is not strongly synchronizing, ValueError is raised).

3.22 IsSynchronous(transducer)

Takes a transducer that is saved as a variable in the Python session, returns 'True' if it is synchronous, and 'False' if it is not.

3.23 IsInvertible(transducer)

Takes a synchronous transducer that is saved as a variable in the Python session, returns 'True' if it is invertible in the automata-theoretic sense, and 'False' if not.

3.24 invert(transducer)

Takes a synchronous, invertible (in the automata-theoretic sense) transducer that is saved as a variable in the Python session, and returns its automata-theoretic inverse.

3.25 IsBiSynchronizing(transducer)

Takes a synchronous transducer that is saved as a variable in the Python session, returns 'True' if it is strongly synchronizing, invertible (in the automata-theoretic sense), and has a strongly synchronizing inverse, and 'False' if not.

Note: Currently only works for synchronous transducers. If checking for bi-synchronizing of asynchronous transducers more generally, this will not be reliable.

3.26 IsWeaklyMinimal(transducer)

Takes a transducer that is saved as a variable in the Python session, returns 'True' if it is weakly minimal, and 'False' if it is not.

3.27 IsCore(transducer)

Takes a transducer that is saved as a variable in the Python session, returns 'True' if it is strongly synchronizing and core, and 'False' if it is not.

3.28 IsInHn(transducer)

Takes a transducer that is saved as a variable in the Python session, returns 'True' if it is a weakly minimal representative of an element of $\mathcal{H}_n$, returns the string "Half true" if it is a non-weakly minimal representative of an element of $\mathcal{H}_n$, or returns 'False' otherwise.

3.29 IsValidAutomorphismForAutomaton(automaton, automorphism)

Takes an automaton that is saved as a variable in the Python session, and a Python representation of a potential automorphism of the underlying graph of the automaton (see next paragraph), returns 'True' if it is a valid automorphism, and 'False' if it is not.

The candidate automorphism must be inputted as a 2-tuple (V, E), where V is a dict that permutes the vertices, and E is a dict that permutes the edges (with each edge being a 3-tuple (p, x, q), that goes from vertex p to vertex q with label x).

3.30 transducer_from_graph_automorphism(automaton, automorphism)

Takes an automaton that is saved as a variable in the Python session and an automorphism of the automaton's underlying graph (see 2nd paragraph of IsValidAutomorphismForAutomaton above for the required format of the graph automorphism), and returns the transducer constructed by 'gluing' the automaton to itself along the automorphism in order to obtain outputs. This construction is denoted $H(A, \phi)$ (for an automaton A and graph automorphism ϕ) in the one-sided case paper.

3.31 decompose_Hn_element(transducer)

Takes a transducer that is a representative of an element of $\mathcal{H}_n$ that is saved as a variable in the Python session, and runs the decomposition algorithm of the one-sided case paper, returning a list of finite order transducers whose product in $\mathcal{H}_\backslash$ is the input transducer, where the order of the product is given by the order of the list, i.e. the first element of the list will be a single state transducer, and the final element of the list will be the first transducer found by the algorithm.

3.32 Hn_product(transducer1, transducer2)

Takes two transducers that are (representatives of) elements of $\mathcal{H}_n$, and returns their product in $\mathcal{H}_n$, i.e. the transducer obtained by identifying ω-equivalent states of the core of their transducer product.

4 Formats

Automata and transducers may be saved as 3 different formats within .txt files, which the interface can load up into Python, or save into the current working directory.

4.1 edgeList

Each line of the .txt file represents an edge of the automaton/transducer, with a semi-colon and a space seperating each piece of information, e.g. one line of the .txt file might read '0; 2; 1; [1,0]', which means the transducer goes from state 0 to state 2 when reading letter 1, and the output is [1,0]. Synchronous transducers may have their outputs written as a letter, e.g. one line could be '0; 2; 1; 1'. Automata are precisely as you'd expect, with outputs dropped, e.g. '0; 2; 1'.

4.2 AAA

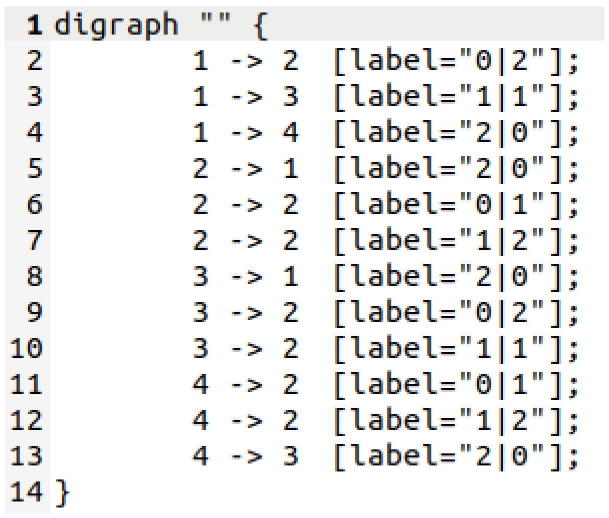
The AAA format is the format used in the AAA ('asynchronous automata algoriths') GAP package by Collin Bleak, Fernando Flores Brito, Luke Elliott and Feyishayo Olukoya. In that package, transducers are treated as 4-tuples (m, n, P, L). m is the size of the input alphabet, n is the size of the output alphabet, P is a list of lists describing the transitions of the transducer, and L is a list of list of lists describing the outputs. For example, (2, 2, [[2, 3], [2, 3], [2, 3]], [[[], []], [[0], [0]], [[1], [1]]]) represents a transducer on a 2-letter alphabet. the element of index i in the list P gives the transitions going from state i, e.g. in this transducer, there are 3 states, and reading 0 always transitions to state 2, while reading 1 always transitions to state 3. Similarly, the element of index i in the list L gives the outputs for each letter, with the outputs themselves written as a list of each letter in the output word, e.g. in this transducer, state 1 always gives empty outputs, state 2 always gives output 0, and state 3 always gives output 1.

For the purposes of this module, the automaton/transducer within a .txt file is a single line of text, which is as the tuple above is written, though it can be with square brackets instead (as a list). For the purposes of copying and pasting a transducer into GAP, the print_aaa_input function is of note - it takes as input a transducer, and returns the string of the transducer as an AAA 4-tuple, preceded by the word 'Transducer'. Don't forget to end it with a semi-colon, GAP users! Note that automata may be stored in AAA format by giving it all empty outputs.

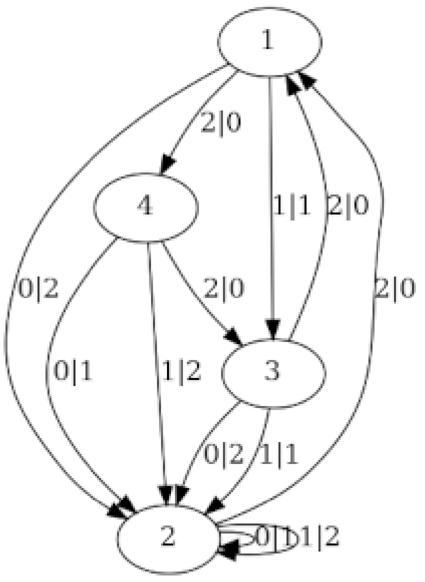
4.3 DOT

DOT is a graph description language, with a syntax designed for encoding graphs. For more information, see this excellent guide: https://www.tonyballantyne.com/graphs.html.

This program requires a .txt or .dot file in DOT format to describe a labeled directed graph that represents an automaton or transducer. For transducers, the label must be of the form ’a—b’, where ’a’ is the input letter and ’b’ is the output word (as a list, or as a letter if the transducer is synchronous) with no spaces between ’a’, ’—’ (which, for clarity, is the vertical bar symbol with unicode code point ’U+007C’), and ’b’. For automata, the label is simply the input letter ’a’. An example transducer is pictured below; the first image is a screenshot of a .txt file, and the second picture is the image generated by the module of the transducer described in the .txt file.



```
digraph "" {
        1 -> 2  [label="0|2"];
        1 -> 3  [label="1|1"];
        1 -> 4  [label="2|0"];
        2 -> 1  [label="2|0"];
        2 -> 2  [label="0|1"];
        2 -> 2  [label="1|2"];
        3 -> 1  [label="2|0"];
        3 -> 2  [label="0|2"];
        3 -> 2  [label="1|1"];
        4 -> 2  [label="0|1"];
        4 -> 2  [label="1|2"];
        4 -> 3  [label="2|0"];
}
```



4.4 Within Python

Finally, it is worth noting that all automata and transducers within Python are stored by this module as a version of the above edgeList format. Specifically, as a list of lists, with each element being a 4-length list [a, b, c, d], where a, b, c are one letter strings corresponding respectively to the start state, the end state, and the input letter, while d is a list of one letter strings that is the output. In this way, each such list represents an ’edge’ of the transducer’s labeled digraph. Synchronous transducers may have d as a letter rather than a list, and automata have d dropped, so that the lists are of length 3. For example, [[’0’, ’1’, ’0’], [’1’, ’0’, ’1’]] represents a transducer with two states 0, 1 over alphabet 0, 1, which goes from state 0 to state 1 when reading 0, or from state 1 to state 0 when reading 1. Note the letters in the lists are stored as strings.

References

[1] C. Bleak, P. Cameron, and F. Olukoya. Automorphisms of shift spaces and the Higman–Thompson groups: the one-sided case, 2020.