
iglsynth Documentation

Release 0.2.3

Abhishek N. Kulkarni

Nov 26, 2019

1	Indices and tables	3
1.1	Installation Instructions	3
1.2	IGLSynth API	4
1.3	IGLSynth Examples	9
1.4	Release Notes	14
	Python Module Index	17
	Index	19

IGLSynth is a high-level Python API for solving Infinite Games and Logic-based strategy Synthesis. It provides an easy interface to

1. Define two-player games-on-graphs.
2. Assign tasks to players using formal logic.
3. Write solvers to compute winning strategies in the game.

IGLSynth consists of 4 modules,

1. `game`: Defines classes representing deterministic/stochastic and concurrent/turn-based games as well as hyper-games.
 2. `logic`: Defines classes representing formal logic such as Propositional Logic, Linear Temporal Logic etc.
 3. `solver`: Defines solvers for different games such as ZielonkaSolver etc.
 4. `util`: Defines commonly used classes such as Graph.
-

1.1 Installation Instructions

IGLSynth requires Python (≥ 3.7) and spot (≥ 2.7). The recommended operating system is Ubuntu 19.04 or above, which ships with Python 3.7. This allows spot and iglsynth to be installed directly using apt-get and pip, respectively.

If you want to use a different operating system, docker is the way to go! Please refer to the docker installation instructions below.

1.1.1 Ubuntu 19.04 or above

First, install spot using instructions given at [install spot](#).

Then, install IGLSynth:

```
$ pip3 install iglsynth
```

1.1.2 Docker Image (other OS)

When using some other OS where either spot or iglsynth is not installed, it is recommended to use [Docker](#) images with an Python IDE such as [PyCharm](#). Note that using PyCharm is not necessary, but definitely makes life easy!!

Assuming Docker client is already installed on your OS, IGLSynth docker image can be downloaded by running:

```
$ docker pull abhibp1993/iglsynth
```

The instructions to set up remote interpreter are given at [Configure a Remote Interpreter using Docker](#).

The above image also has `numpy`, `matplotlib` pre-installed.

1.1.3 Docker Image for Developers

This section is only for developers of IGLSynth.

IGLSynth development requires additional tools such as pytest (testing), pytest-cov (coverage checking), sphinx (automatic documentation), sphinx_rtd_theme (theme for sphinx documentation). These all tools are installed in the developer image of docker (tagged `dev`), in addition to spot.

Note: IGLSynth is **NOT** installed in developer's docker image. Developers are expected to mount the necessary folders for development.

You can download the developer image of IGLSynth by running:

```
$ docker pull abhibp1993/iglsynth:dev
```

and configure the remote interpreter with PyCharm using instructions given at [Configure a Remote Interpreter using Docker](#).

1.2 IGLSynth API

1.2.1 Game Module API

Overview

Will be written..

Game Module

Global Variables

```
iglsynth.game.core.TURN_BASED = 'Turn-based'
```

Macro to define concurrent transition system and game.

Action

```
class iglsynth.game.core.Action (name=None, func=None)
```

Bases: object

Represents an action. An action acts on a state (of TSys or Game etc.) to produce a new state.

Parameters

- **name** – (str) Name of the action.
- **func** – (function) An implementation of action.

Note: Acceptable function templates are,

- `st <- func(st)`

- `st <- func(st, *args)`
 - `st <- func(st, **kwargs)`
 - `st <- func(st, *args, **kwargs)`
-

Game Module

Two-Player Deterministic Game

1.2.2 Logic Module API

Logic Module

in progress...

Logic Module

Global Variables

```
iglsynth.logic.core.TRUE = AP(name=true)
iglsynth.logic.core.FALSE = AP(name=false)
```

ILogic (Interface Class)

SyntaxTree

Atomic Propositions

Alphabet

Propositional Logic Formulas

Linear Temporal Logic

1.2.3 Utilities API

Utilities

in progress..

Utilities

Graph Class

class `iglsynth.util.graph.Graph` (*vtype=None, etype=None, graph=None, file=None*)

Base class to represent graph-based objects in IGLSynth.

- `Graph` may represent a Digraph or a Multi-Digraph.
- `Graph.Edge` may represent a self-loop, i.e. *source = target*.
- `Graph` stores objects of `Graph.Vertex` and `Graph.Edge` classes or their sub-classes, which users may define.
- `Graph.Vertex` and `Graph.Edge` may have attributes, which represent the vertex and edge properties of the graph.

Parameters

- **vtype** – (class) Class representing vertex objects.
- **etype** – (class) Class representing edge objects.
- **graph** – (`Graph`) Copy constructor. Copies the input graph into new Graph object.
- **file** – (str) Name of file (with absolute path) from which to load the graph.

Todo: The copy-constructor and load-from-file functionality.

class `Edge` (*u: iglsynth.util.graph.Graph.Vertex, v: iglsynth.util.graph.Graph.Vertex*)

Base class for representing a edge of graph.

- `Graph.Edge` represents a directed edge.
- Two edges are equal, if the two `Graph.Edge` objects are same.

Parameters

- **u** – (`Graph.Vertex` or its sub-class) Source vertex of edge.
- **v** – (`Graph.Vertex` or its sub-class) Target vertex of edge.

source

Returns the source vertex of edge.

target

Returns the target vertex of edge.

class `Vertex`

Base class for representing a vertex of graph.

- `Graph.Vertex` constructor takes no arguments.
- Two vertices are equal, if the two `Graph.Vertex` objects are same.

add_edge (*e: iglsynth.util.graph.Graph.Edge*)

Adds an edge to the graph. Both the vertices must be present in the graph.

Parameters **e** – (`Graph.Edge`) An edge to be added to the graph.

Raises

- **AttributeError** – When at least one of the vertex is not in the graph.
- **AssertionError** – When argument *e* is not an `Graph.Edge` object.

add_edges (*ebunch*: `Iterable[Graph.Edge]`)

Adds a bunch of edges to the graph. Both the vertices of all edges must be present in the graph.

Raises

- **AttributeError** – When at least one of the vertex is not in the graph.
- **AssertionError** – When argument *e* is not an `Graph.Edge` object.

add_vertex (*v*: `iglsynth.util.graph.Graph.Vertex`)

Adds a new vertex to graph. An attempt to add existing vertex will be ignored, with a warning.

Parameters **v** – (`Graph.Vertex`) Vertex to be added to graph.

add_vertices (*vbunch*: `Iterable[Graph.Vertex]`)

Adds a bunch of vertices to graph. An attempt to add existing vertex will be ignored, with a warning.

Parameters **vbunch** – (Iterable over `Graph.Vertex`) Vertices to be added to graph.

edges

Returns an iterator over edges in graph.

get_edges (*u*: `iglsynth.util.graph.Graph.Vertex`, *v*: `iglsynth.util.graph.Graph.Vertex`)

Returns all edges with source *u* and target *v*.

Parameters

- **u** – (`Graph.Vertex`) Vertex of the graph.
- **v** – (`Graph.Vertex`) Vertex of the graph.

Returns (iterator(`Graph.Edge`)) Edges between *u*, *v*.

has_edge (*e*: `iglsynth.util.graph.Graph.Edge`)

Checks whether the graph has the given edge or not.

Parameters **e** – (`Graph.Edge`) An edge to be checked for containment in the graph.

Returns (bool) True if the graph has the given edge, False otherwise.

has_vertex (*v*: `iglsynth.util.graph.Graph.Vertex`)

Checks whether the graph has the given vertex or not.

Parameters **v** – (`Graph.Vertex`) Vertex to be checked for containment.

Returns (bool) True if given vertex is in the graph, else False.

in_edges (*v*: `Union[Graph.Vertex, Iterable[Graph.Vertex]]`)

Returns an iterator over incoming edges to given vertex or vertices. In case of vertices, the iterator is defined over the union of set of incoming edges of individual vertices.

Parameters **v** – (`Graph.Vertex`) Vertex of graph.

Raises **AssertionError** – When *v* is neither a `Graph.Vertex` object nor an iterable of `Graph.Vertex` objects.

in_neighbors (*v*: `Union[Graph.Vertex, Iterable[Graph.Vertex]]`)

Returns an iterator over sources of incoming edges to given vertex or vertices. In case of vertices, the iterator is defined over the union of set of incoming edges of individual vertices.

Parameters **v** – (`Graph.Vertex`) Vertex of graph.

Raises `AssertionError` – When v is neither a `Graph.Vertex` object nor an iterable of `Graph.Vertex` objects.

num_edges

Returns the number of edges in graph.

num_vertices

Returns the number of vertices in graph.

out_edges (v : `Union[Graph.Vertex, Iterable[Graph.Vertex]]`)

Returns an iterator over outgoing edges to given vertex or vertices. In case of vertices, the iterator is defined over the union of set of incoming edges of individual vertices.

Parameters v – (`Graph.Vertex`) Vertex of graph.

Raises `AssertionError` – When v is neither a `Graph.Vertex` object nor an iterable of `Graph.Vertex` objects.

out_neighbors (v : `Union[Graph.Vertex, Iterable[Graph.Vertex]]`)

Returns an iterator over targets of incoming edges to given vertex or vertices. In case of vertices, the iterator is defined over the union of set of incoming edges of individual vertices.

Parameters v – (`Graph.Vertex`) Vertex of graph.

Raises `AssertionError` – When v is neither a `Graph.Vertex` object nor an iterable of `Graph.Vertex` objects.

rm_edge (e : `iglsynth.util.graph.Graph.Edge`)

Removes an edge from the graph. An attempt to remove a non-existing edge will be ignored, with a warning.

Parameters e – (`Graph.Edge`) Edge to be removed.

rm_edges ($ebunch$: `Iterable[Graph.Edge]`)

Removes a bunch of edges from the graph. An attempt to remove a non-existing edge will be ignored, with a warning.

Parameters $ebunch$ – (Iterable over `Graph.Edge`) Edges to be removed.

rm_vertex (v : `iglsynth.util.graph.Graph.Vertex`)

Removes a vertex from the graph. An attempt to remove a non-existing vertex will be ignored, with a warning.

Parameters v – (`Graph.Vertex`) Vertex to be removed.

rm_vertices ($vbunch$: `Iterable[Graph.Vertex]`)

Removes a bunch of vertices from the graph. An attempt to remove a non-existing vertex will be ignored, with a warning.

Parameters $vbunch$ – (Iterable over `Graph.Vertex`) Vertices to be removed.

vertices

Returns an iterator over vertices in graph.

Utilities

Spot Formulas

1.2.4 Solver Module API

Overview

Will be written..

Solver Module

Zielonka Solver

1.3 IGLSynth Examples

1.3.1 Constructing a Game Graph

This example demonstrates how to construct a simple game on graph. There are three ways to define a game on graph, namely

- By explicitly adding vertices and edges to the graph and marking the accepting states,
- By providing a transition system and a formal specification,
- By providing a game field and player objects.

Note: IGLSynth v0.2.2 only supports explicit construction of graph.


Define Game by Explicit Construction

The module `iglsynth.game.game` provides necessary classes to define a game on graph. Hence, first import the `game.game` module:

```
from iglsynth.game.game import Game
```

Note that `Game` class defines a deterministic two-player zero-sum game on graph.

A `Game` can be either `TURN_BASED` or `CONCURRENT`. For this example, let us consider a `TURN_BASED` game on graph shown in following image.



examples/EPFL_Problem1.png

First, instantiate a game object:

```
game = Game(kind=TURN_BASED)
```

Now, add the vertices and assign each vertex to a player. When a vertex has `turn = 1`, player 1 (circle) will make a move. When a vertex has `turn = 2`, player 2 (box) will make a move.

The vertex in a game (an instance of `Game` class) must be of type `Game.Vertex` or its derivative. Hence, it is recommended to instantiate a new game vertex as `game.Vertex`:

```
vertices = list()
for i in range(8):
    if i in [0, 4, 6]:
        vertices.append(game.Vertex(name=str(i), turn=1))
    else:
        vertices.append(game.Vertex(name=str(i), turn=2))

game.add_vertices(vertices)
```

and mark the vertices 3, 4 as final:

```
# Set the states as final
v3 = vertices[3]
v4 = vertices[4]
game.mark_final(v3)
game.mark_final(v4)
```

Finally, add the edges to the game. Similar to vertices, we instantiate new edges as `game.Edge` objects:

```
# Add edges to the game graph
edge_list = [(0, 1), (0, 3), (1, 0), (1, 2), (1, 4), (2, 4), (2, 2), (3, 0), (3, 4),
             ↪ (3, 5), (4, 3),
              (5, 3), (5, 6), (6, 6), (6, 7), (7, 0), (7, 3)]

for uid, vid in edge_list:
    u = vertices[uid]
    v = vertices[vid]
    game.add_edge(game.Edge(u=u, v=v))
```

Now, given a game we invoke the `ZielonkaSolver` from `iglsynth.solver.zielonka` module to compute the winning regions for players 1 and 2:

```
from iglsynth.solver.zielonka import ZielonkaSolver
solver = ZielonkaSolver(game)
solver.solve()
```

The `solver.solve()` runs the solver on the `Game` object `game` that encodes the game on graph. The solution of solver can be accessed using the properties:

```
print(solver.p1_win)
print(solver.p2_win)
```

which returns the winning sets for P1 and P2.

1.3.2 Gridworld Problem: Game on Graph

This example demonstrates how to construct a game on graph using a Gridworld transition system and an LTL specification for P1.

In general, there are three ways to define a game on graph, namely

- By explicitly adding vertices and edges to the graph and marking the accepting states,
- By providing a transition system and a formal specification,
- By providing a game field and player objects.

Presently, in IGLSynth v0.2.3, only first two are supported. See *Gridworld Problem: Game on Graph* for an example of constructing game on graph by explicitly adding vertices and edges.

Define Transition System

To define a game using transition system and P1's specification, first define the transition system. In general, a transition system may be defined using `TSys` class. In this example, we use a predefined `Gridworld` transition system from `iglsynth.game.gridworld` module:

```
from iglsynth.game.gridworld import *

# Define transition system
tsys = Gridworld(kind=TURN_BASED, dim=(3, 3), p1_actions=CONN_4, p2_actions=CONN_4)
tsys.generate_graph()
```

Define LTL Specification

To define P1's objective using an LTL specification, import the logic module:

```
from iglsynth.logic.ltl import *
```

An LTL formula is defined over an alphabet. Hence, we start by defining AP objects using which we will construct an alphabet.

In this demo example, we will represent P1's objective to reach to a goal. Such a reachability specification can be written as $\varphi = \Diamond a$ where a is an AP representing goal. So, define the AP as:

```
GOAL = (0, 0)
a = AP("a", lambda st, *args, **kwargs: st.coordinate[0:2] == GOAL)
```

and the specification as:

```
spec = LTL("Fa", alphabet=Alphabet([a]))
```

Define Game Object

Given a transition system and specification, we will now define a game. To define game, we require the game module:

```
from iglsynth.game.game import *
```

Now, instantiate and define a game as follows:

```
game = Game(kind=TURN_BASED)
game.define(tsys, spec)
```

Solve the Game

The final step is to invoke a solver to solve the game. We will use Zielonka's algorithm to solve the above game. Note that Zielonka's algorithm computes winning region for P1 in a deterministic two-player zero-sum game.

In IGLSynth, Zielonka's algorithm is implemented in `zielonka` module:

```
from iglsynth.solver.zielonka import ZielonkaSolver
```

To run the solver, first instantiate it and call the `solve` method:

```
solver = ZielonkaSolver(game)
solver.solve()
```

The `solver.solve()` runs the solver on the `Game` object `game` that encodes the game on graph. The solution of solver can be accessed using the properties:

```
print(solver.pl_win)
print(solver.p2_win)
```

which returns the winning sets for P1 and P2.

1.3.3 Defining Logic Formulas

This example demonstrates how to define logic formulas using `iglsynth.logic` module. Presently, IGLSynth defines three classes of logic formulas; namely atomic propositions AP, propositional logic PL and linear temporal logic LTL.

Atomic Propositions

The most common use of logic formulas in IGLSynth is to define certain properties of a game on graph. Hence, we define atomic propositions to be `Callable` objects over states (or vertices) of a graph.

An AP can be defined by providing a name:

```
a = AP(formula='a')
```

However, it is desirable to evaluate an AP at a given state. This can be achieved by instantiating an AP in the following way:

```
# Define atomic proposition with evaluation function
@ap
def is_goal(st):
    return st == 10

# Define atomic proposition with evaluation function accepting extra arguments
@ap
def is_colliding(st, *args):
    obs = args[0]
    return st in obs

# Define atomic proposition with evaluation function accepting extra keyword arguments
@ap
def is_close(st, **kwargs):
    threshold = kwargs['threshold']
    return st - threshold < 10
```

Let us understand the three AP definitions given above. The first AP, `is_goal`, accepts a state `st` as an input parameter and returns whether `st == 10` or not. It is imperative that an AP must return a `bool`.

In many cases, it is convenient to define a parameterized atomic proposition. In such cases, the extra parameters can be passed to the AP's evaluation function as optional arguments `args` or optional keyword arguments `kwargs`.

Note: It is strongly recommended that AP's function template to have `*args`, `**kwargs` as parameters.

Given an AP, it is possible to evaluate it at a given state:

```
res = is_goal(10)
res = is_goal(20)

res = is_colliding(10, [5, 10])
res = is_colliding(20, [5, 10])

res = is_close(5, threshold=10)
res = is_close(50, threshold=10)
```

Propositional Logic Formulas

Propositional Logic Formulas can be defined by instantiating `PL` class. Similar to `AP`, a propositional logic formula can be defined by passing a formula string containing only `And(&)`, `Or(|)` and/or `Negation(!)` operators:

```
plf0 = PL("a & b")
plf1 = PL("a | b")
plf2 = PL("!a | b")
```

A `PL` formula is, generally, defined over an alphabet. An alphabet is a set of atomic propositions. To define a `PL` formula over alphabet, first define an alphabet:

```
@ap
def a(st, *args, **kwargs):
    return st == 10

@ap
def b(st, *args, **kwargs):
    return st in args[0]

sigma = Alphabet([a, b])
```

An important feature of alphabet is that it can be evaluated at a given state. For example,:

```
result = sigma.evaluate(10, [10, 20])
```

returns a dictionary with keys as AP's in alphabet and values as the result of evaluating the AP at given state. In above case, `result = {a: True, b: True}`.

Note: Observe that evaluating an alphabet at a state is equivalent to computing the label of that state.

Now, we can define a `PL` formula over an alphabet.

```
plf = PL("a & b", alphabet=sigma)
```

`PL` formulas can also be evaluated at a given state.

```
# PL is callable class
res = plf(10, [10, 20])

# Explicitly call evaluate function
res = plf.evaluate(10, [10, 20])
```

When a PL formula is evaluated at a given state, the alphabet is evaluated at that state. Then, the APs in PL formula are substituted with the evaluated values to get back True or False.

Warning: Calling PL object like `res = plf(10, [10, 20])` is currently failing. See [Bug Report Issue #17](#).

Temporal Logic Formulas

(Linear) Temporal Logic formulas can be defined by instantiating `LTL` class. Similar to `PL`, LTL formulas can be defined with/without providing an alphabet.

```
# Alphabet not provided.
ltlf0 = LTL(formula="Gp1 & !(p1 & X(p0 xor p1))")

# Alphabet provided
ltlf = LTL("F(a & Fb)", alphabet=sigma)
```

1.4 Release Notes

1.4.1 0.2.4 (Proposed)

- Solver module
 - [Issue #24] Define `Strategy` class to represent deterministic and stochastic strategies.

1.4.2 0.2.3 ()

- Bugs
 - [Issue #20] `Automaton.Edge` now returns an *AP* or *PL* class object for *AP*, *PL*, `LTL.translate()`.
 - [Issue #17] Unexpected `AssertionError` when calling *PL*(*st=.*) and *LTL*(*st=.*) is handled resolved.
 - [Issue #16] Bug in the printing *PL*, *LTL* objects was resolved.
- Game module
 - [Issue #6] `Gridworld` class accept `p1_action` and `p2_action` separately.
 - [Issue #6] `Gridworld` class defined with basic 4, 5, 8, 9 connectivity.
 - [Issue #23] Implement `game.define(tsys, spec)` function.
 - [Issue #4] Defined `Kripke` class to support `TSys` class. No special properties or functions are associated with `Kripke` are defined or implemented.

- [Issue #5] Updated `TSys` constructor signature to accept `p1_action`, `p2_action` instead of `actions`. `TSys.Vertex` now requires `name`, `turn` parameters. `turn = None` represents a concurrent game. `TSys.Edge` is updated to represent both concurrent and turn-based edges.
- Solver module
 - [Issue #24] `ZielonkaSolver` class defined and implemented. The solver computes winning regions only.
- Utilities
 - [Issue #11] Added `has_vertex`, `has_edge`, `get_edges` functionality to `Graph` class.

1.4.3 0.2.2 (17 November 2019)

- Docker images
 - Updated `latest` and `dev` docker images on docker hub. The latest image now ships with `iglsynth` installed.
 - Set up a GitHub - DockerHub webhook. All pushes to master branch will trigger a docker image build.
- Documentation
 - Added two examples:
 - * Game graph construction
 - * Logic formula definition
 - Added installation instructions.
 - Fixed bugs in documentation.

1.4.4 0.2.1 (16 November 2019)

- **Game module**
 - (core) Action class defined with “@action” decorator.
 - (game) Game class defined to represent Deterministic 2-player Game.
 - * Game can be of two kinds: `TURN_BASED` or `CONCURRENT`.
 - * Game can be defined by constructing the game graph (i.e. using `game.add_vertex`, `game.add_edge` functions)
- **Logic module**
 - (core) `ILogic` class added as an interface class to define logic classes.
 - (core) `SyntaxTree` class added to represent abstract syntax tree of an `ILogic` formula.
 - (core) `AP` class added to represent atomic propositions.
 - * `AP` can be defined using decorator “@ap”.
 - * `AP.__call__`, `AP.evaluate` methods provide a way to check whether an `AP` is true/false in a given state.
 - (core) `PL` class added to represent propositional logic formulas.
 - * `PL.__call__`, `PL.evaluate` methods provide a way to check whether the formula is true in a given state.

- * PL.substitute provide a way to substitute APs in PL formula with their valuations (True/False) or other APs.
 - (core) Alphabet class added to represent a set of APs.
 - * Alphabet.__call__, Alphabet.evaluate methods provide a way to get a label of a state.
 - (ltl) LTL class is defined.
- **Util Module**
 - (graph) Graph classes redefined.
 - * Defined “Vertex” and “Edge” base classes to define vertex and edge properties.
 - * Adding and removing vertices, edges is implemented.
 - * Accessing in/out neighbors/edges is implemented.
- Documentation updated.

1.4.5 0.1.0 (07 August 2019)

- **Game module**
 - Base class for writing different types of games is ready.
 - Deterministic 2 player game is partially defined.
- **Solver module**
 - Base class for writing solvers is partially ready.
 - Zielonka attractor algorithm is implemented. Only a few configurations are supported.
- **Utility module**
 - Graph class is ready.
 - SubGraph class is ready.
- **Examples**
 - An example from [EPFL Slides](#) is added.
- First release of IGLSynth

Current release and documentation update date:

- [genindex](#)
- [modindex](#)
- [search](#)

e

examples, 9

i

iglsynth, 3

iglsynth.logic, 5

iglsynth.util, 5

A

Action (class in *iglsynth.game.core*), 4
add_edge() (*iglsynth.util.graph.Graph* method), 6
add_edges() (*iglsynth.util.graph.Graph* method), 7
add_vertex() (*iglsynth.util.graph.Graph* method), 7
add_vertices() (*iglsynth.util.graph.Graph* method), 7

E

edges (*iglsynth.util.graph.Graph* attribute), 7
examples (module), 9

F

FALSE (in module *iglsynth.logic.core*), 5

G

get_edges() (*iglsynth.util.graph.Graph* method), 7
Graph (class in *iglsynth.util.graph*), 6
Graph.Edge (class in *iglsynth.util.graph*), 6
Graph.Vertex (class in *iglsynth.util.graph*), 6

H

has_edge() (*iglsynth.util.graph.Graph* method), 7
has_vertex() (*iglsynth.util.graph.Graph* method), 7

I

iglsynth (module), 3, 4
iglsynth.logic (module), 5
iglsynth.util (module), 5
in_edges() (*iglsynth.util.graph.Graph* method), 7
in_neighbors() (*iglsynth.util.graph.Graph* method), 7

N

num_edges (*iglsynth.util.graph.Graph* attribute), 8
num_vertices (*iglsynth.util.graph.Graph* attribute), 8

O

out_edges() (*iglsynth.util.graph.Graph* method), 8

out_neighbors() (*iglsynth.util.graph.Graph* method), 8

R

rm_edge() (*iglsynth.util.graph.Graph* method), 8
rm_edges() (*iglsynth.util.graph.Graph* method), 8
rm_vertex() (*iglsynth.util.graph.Graph* method), 8
rm_vertices() (*iglsynth.util.graph.Graph* method), 8

S

source (*iglsynth.util.graph.Graph.Edge* attribute), 6

T

target (*iglsynth.util.graph.Graph.Edge* attribute), 6
TRUE (in module *iglsynth.logic.core*), 5
TURN_BASED (in module *iglsynth.game.core*), 4

V

vertices (*iglsynth.util.graph.Graph* attribute), 8