

# API Documentation

API Documentation

December 10, 2015

## Contents

<b>Contents</b>	<b>1</b>
<b>1 Package datk</b>	<b>4</b>
1.1 Modules . . . . .	4
1.2 Variables . . . . .	4
<b>2 Package datk.core</b>	<b>5</b>
2.1 Modules . . . . .	5
2.2 Variables . . . . .	5
<b>3 Module datk.core.algs</b>	<b>6</b>
3.1 Variables . . . . .	6
3.2 Class LCR . . . . .	6
3.2.1 Methods . . . . .	6
3.3 Class AsyncLCR . . . . .	7
3.3.1 Methods . . . . .	7
3.4 Class FloodMax . . . . .	8
3.4.1 Methods . . . . .	8
3.5 Class SynchBFS . . . . .	9
3.5.1 Methods . . . . .	9
3.6 Class SynchBFSAck . . . . .	10
3.6.1 Methods . . . . .	10
3.7 Class SynchConvergecast . . . . .	11
3.7.1 Methods . . . . .	11
3.8 Class AsyncConvergecast . . . . .	12
3.8.1 Methods . . . . .	12
3.9 Class SynchConvergeHeight . . . . .	13
3.9.1 Methods . . . . .	13
3.10 Class AsyncConvergeHeight . . . . .	14
3.10.1 Methods . . . . .	14
3.11 Class SynchBroadcast . . . . .	15
3.11.1 Methods . . . . .	15
3.12 Class SynchLubyMIS . . . . .	16
3.12.1 Methods . . . . .	17
<b>4 Module datk.core.distalgs</b>	<b>18</b>
4.1 Variables . . . . .	18

4.2	Class Message . . . . .	18
4.2.1	Methods . . . . .	18
4.3	Class Process . . . . .	18
4.3.1	Methods . . . . .	18
4.4	Class Network . . . . .	19
4.4.1	Methods . . . . .	19
4.5	Class Algorithm . . . . .	20
4.5.1	Methods . . . . .	20
4.6	Class Synchronous_Algorithm . . . . .	21
4.6.1	Methods . . . . .	21
4.7	Class Do_Nothing . . . . .	22
4.7.1	Methods . . . . .	22
4.8	Class Asynchronous_Algorithm . . . . .	22
4.8.1	Methods . . . . .	22
4.9	Class Compose . . . . .	23
4.9.1	Methods . . . . .	23
4.10	Class Chain . . . . .	24
4.10.1	Methods . . . . .	24
<b>5</b>	<b>Module datk.core.imports</b>	<b>25</b>
5.1	Variables . . . . .	25
<b>6</b>	<b>Module datk.core.networks</b>	<b>26</b>
6.1	Variables . . . . .	26
6.2	Class Unidirectional_Ring . . . . .	26
6.2.1	Methods . . . . .	26
6.3	Class Bidirectional_Ring . . . . .	26
6.3.1	Methods . . . . .	27
6.4	Class Unidirectional_Line . . . . .	27
6.4.1	Methods . . . . .	27
6.5	Class Bidirectional_Line . . . . .	27
6.5.1	Methods . . . . .	28
6.6	Class Complete_Graph . . . . .	28
6.6.1	Methods . . . . .	28
6.7	Class Random_Line_Network . . . . .	28
6.7.1	Methods . . . . .	29
<b>7</b>	<b>Module datk.core.testers</b>	<b>30</b>
7.1	Functions . . . . .	30
7.2	Variables . . . . .	30
<b>8</b>	<b>Package datk.tests</b>	<b>31</b>
8.1	Modules . . . . .	31
8.2	Variables . . . . .	31
<b>9</b>	<b>Module datk.tests.helpers</b>	<b>32</b>
9.1	Functions . . . . .	32
9.2	Variables . . . . .	32
<b>10</b>	<b>Module datk.tests.networks_tests</b>	<b>33</b>
10.1	Variables . . . . .	33
<b>11</b>	<b>Module datk.tests.tests</b>	<b>34</b>

---

11.1 Functions . . . . .	34
11.2 Variables . . . . .	35
<b>Index</b>	<b>36</b>

# 1 Package datk

## 1.1 Modules

- **core**: A Python Toolkit for Distributed Algorithms  
(Section 2, p. 5)
  - **algs** (Section 3, p. 6)
  - **distalgs** (Section 4, p. 18)
  - **imports** (Section 5, p. 25)
  - **networks** (Section 6, p. 26)
  - **tester** (Section 7, p. 30)
- **tests** (Section 8, p. 31)
  - **helpers**: Helper functions for tests in tests.py  
(Section 9, p. 32)
  - **networks\_tests**: Network Test Suite  
(Section 10, p. 33)
  - **tests**: Algorithm Test Suite  
(Section 11, p. 34)

## 1.2 Variables

Name	Description
<code>__package__</code>	<b>Value:</b> None

## 2 Package datk.core

A Python Toolkit for Distributed Algorithms

Authors:

- Amin Manna <manna@mit.edu>
- Mayuri Sridhar <mayuri@mit.edu>

### 2.1 Modules

- **algs** (*Section 3, p. 6*)
- **distalgs** (*Section 4, p. 18*)
- **imports** (*Section 5, p. 25*)
- **networks** (*Section 6, p. 26*)
- **tester** (*Section 7, p. 30*)

### 2.2 Variables

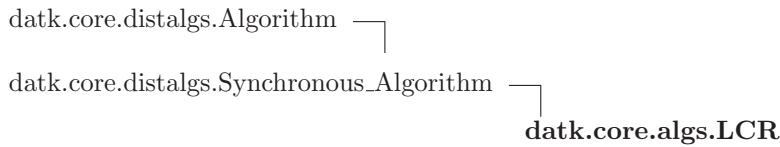
Name	Description
<code>--package--</code>	<b>Value:</b> None

### 3 Module *datk.core.algs*

#### 3.1 Variables

Name	Description
<code>--package--</code>	<b>Value:</b> <code>'datk.core'</code>

#### 3.2 Class LCR



The LeLann, Chang and Roberts algorithm for Leader Election in a Synchronous Ring Network

Each Process sends its identifier around the ring. When a Process receives an incoming identifier, it compares that identifier to its own. If the incoming identifier is greater than its own, it keeps passing the identifier; if it is less than its own, it discards the incoming identifier; if it is equal to its own, the Process declares itself the leader.

Requires:

- Every process knows `state['n']`, the size of the network

Effects:

- Every process has `state['status']` is 'leader' or 'non-leader'.
- Exactly one process has `state['status']` is 'leader'

##### 3.2.1 Methods

**msgs\_i**(*self*, *p*)

Overrides: `datk.core.distalgs.Algorithm.msgs_i`

**trans\_i**(*self*, *p*, *msgs*)

Overrides: `datk.core.distalgs.Algorithm.trans_i`

**cleanup\_i**(*self*, *p*)

Overrides: `datk.core.distalgs.Algorithm.cleanup_i`

*Inherited from `datk.core.distalgs.Synchronous_Algorithm`(Section 4.6)*

`execute()`, `msgs()`, `round()`, `run()`, `trans()`

*Inherited from `datk.core.distalgs.Algorithm`(Section 4.5)*

`__call__()`, `__init__()`, `cleanup()`, `count_msg()`, `delete()`, `get()`, `halt()`, `halt_i()`, `has()`, `increment()`, `set()`

### 3.3 Class AsyncLCR



The LeLann, Chang and Roberts algorithm for Leader Election in an Asynchronous Ring Network

Each Process sends its identifier around the ring. When a Process receives incoming identifier(s), it compares their largest to its own. If that incoming identifier is greater than its own, it keeps passing that identifier; if it is less than its own, it discards all the incoming identifiers; if it is equal to its own, the Process declares itself the leader. When a Process has declared itself Leader, it sends a Leader Declaration message around the ring, and halts. As it goes around the ring, each other Process outputs 'non-leader', and halts.

Requires:

- Every process knows `state['n']`, the size of the network

Effects:

- Every process has `state['status']` is 'leader' or 'non-leader'.
- Exactly one process has `state['status']` is 'leader'

#### 3.3.1 Methods

**`msgs_i(self, p, verbose=False)`**

Overrides: `datk.core.distalgs.Algorithm.msgs_i`

**`trans_i(self, p, verbose=False)`**

Overrides: `datk.core.distalgs.Algorithm.trans_i`

**`cleanup_i(self, p)`**

Overrides: `datk.core.distalgs.Algorithm.cleanup_i`

*Inherited from `datk.core.distalgs.Asynchronous_Algorithm`(Section 4.8)*

`run()`, `run_process()`

***Inherited from datk.core.distalgs.Algorithm(Section 4.5)***

`__call__()`, `__init__()`, `cleanup()`, `count_msg()`, `delete()`, `get()`, `halt()`, `halt_i()`, `has()`, `increment()`, `set()`

**3.4 Class FloodMax**

UID flooding algorithm for Leader Election in a general network

Every process maintains a record of the maximum UID it has seen so far (initially its own). At each round, each process propagates this maximum on all of its outgoing edges. After `diam` rounds, if the maximum value seen is the process's own UID, the process elects itself the leader; otherwise, it is a non-leader.

Requires:

- Every process, `p`, has `p.state["diam"] >= dist( p, q )`, forall `q`.

**3.4.1 Methods**

**`msgs_i(self, p)`**

Overrides: `datk.core.distalgs.Algorithm.msgs_i`

**`trans_i(self, p, msgs, verbose=False)`**

Overrides: `datk.core.distalgs.Algorithm.trans_i`

**`cleanup_i(self, p)`**

Overrides: `datk.core.distalgs.Algorithm.cleanup_i`

***Inherited from datk.core.distalgs.Synchronous\_Algorithm(Section 4.6)***

`execute()`, `msgs()`, `round()`, `run()`, `trans()`

***Inherited from datk.core.distalgs.Algorithm(Section 4.5)***

`__call__()`, `__init__()`, `cleanup()`, `count_msg()`, `delete()`, `get()`, `halt()`, `halt_i()`, `has()`, `increment()`, `set()`



### 3.5 Class *SynchBFS*



Constructs a BFS tree with the 'leader' Process at its root

At any point during execution, there is some set of processes that is "marked," initially just *i0*. Process *i0* sends out a search message at round 1, to all of its outgoing neighbors. At any round, if an unmarked process receives a search message, it marks itself and chooses one of the processes from which the search has arrived as its parent. At the first round after a process gets marked, it sends a search message to all of its outgoing neighbors.

Requires:

- `testLeaderElection`

Effects:

- every Process has `state['parent']`. Leader has `state['parent'] = None`

#### 3.5.1 Methods

<b><code>is_i0(self, p)</code></b>
------------------------------------

<b><code>msgs_i(self, p)</code></b>
-------------------------------------

Overrides: <code>datk.core.distalgs.Algorithm.msgs_i</code>
---

<b><code>trans_i(self, p, msgs)</code></b>
--

Overrides: <code>datk.core.distalgs.Algorithm.trans_i</code>
--

***Inherited from `datk.core.distalgs.Synchronous_Algorithm`(Section 4.6)***

`execute()`, `msgs()`, `round()`, `run()`, `trans()`

***Inherited from `datk.core.distalgs.Algorithm`(Section 4.5)***

`_call_()`, `_init_()`, `cleanup()`, `cleanup_i()`, `count_msg()`, `delete()`, `get()`, `halt()`, `halt_i()`, `has()`, `increment()`, `set()`

### 3.6 Class *SynchBFSAck*



Constructs a BFS tree with children pointers and the 'leader' Process at its root

Algorithm (Informal): At any point during execution, there is some set of processes that is "marked," initially just *i0*. Process *i0* sends out a search message at round 1, to all of its outgoing neighbors. At any round, if an unmarked process receives a search message, it marks itself and chooses one of the processes from which the search arrived as its parent. At the first round after a process gets marked, it sends a search message to all of its outgoing neighbors, and an acknowledgement to its parent, so that nodes will also know their children.

Requires:

- `testLeaderElection`

Effects:

- Every process knows:
  - `state['parent']`. Leader has `state['parent'] = None`
  - `state['children']`. Leaves have `state['children'] = []`

#### 3.6.1 Methods

<code>is_i0(self, p)</code>
-----------------------------

<code>msgs_i(self, p)</code>
------------------------------

Overrides: <code>datk.core.distalgs.Algorithm.msgs_i</code>
---

<code>trans_i(self, p, msgs)</code>
-------------------------------------

Overrides: <code>datk.core.distalgs.Algorithm.trans_i</code>
--

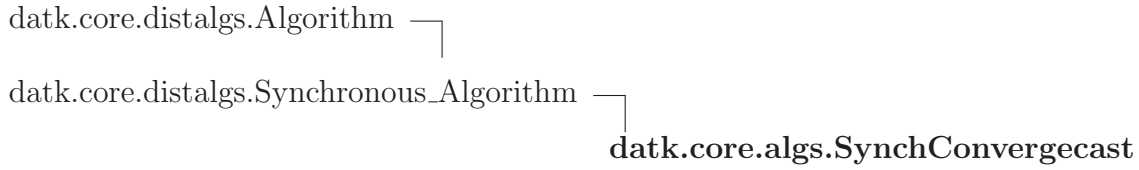
***Inherited from `datk.core.distalgs.Synchronous_Algorithm`(Section 4.6)***

`execute()`, `msgs()`, `round()`, `run()`, `trans()`

***Inherited from `datk.core.distalgs.Algorithm`(Section 4.5)***

`_call_()`, `_init_()`, `cleanup()`, `cleanup_i()`, `count_msg()`, `delete()`, `get()`, `halt()`, `halt_i()`, `has()`, `increment()`, `set()`

### 3.7 Class *SynchConvergecast*



**Known Subclasses:** *datk.core.algs.SynchConvergeHeight*

The abstract superclass of a class of Synchronous Algorithms that propagate information from the leaves of a BFS tree to its root.

Requires:

- Every Process knows state['parent']

#TODO If Processes also know state['children'] ==> Reduced Communication Complexity.

#### 3.7.1 Methods

**is\_root**(*self*, *p*)

**msgs\_i**(*self*, *p*)

Overrides: *datk.core.distalgs.Algorithm.msgs\_i*

**trans\_i**(*self*, *p*, *msgs*)

Overrides: *datk.core.distalgs.Algorithm.trans\_i*

**cleanup\_i**(*self*, *p*)

Overrides: *datk.core.distalgs.Algorithm.cleanup\_i*

**trans\_root**(*self*, *p*, *msgs*)

**output\_root**(*self*, *p*)

**initial\_msg\_to\_parent**(*self*, *p*)

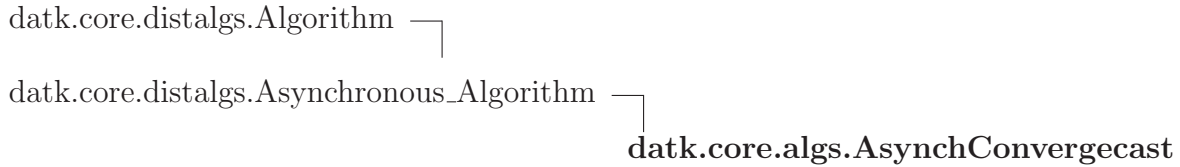
**trans\_msg\_to\_parent**(*self*, *p*, *msgs*)

*Inherited from datk.core.distalgs.Synchronous\_Algorithm(Section 4.6)*

*execute()*, *msgs()*, *round()*, *run()*, *trans()*

***Inherited from datk.core.distalgs.Algorithm(Section 4.5)***

`_call_()`, `_init_()`, `cleanup()`, `count_msg()`, `delete()`, `get()`, `halt()`, `halt_i()`, `has()`,  
`increment()`, `set()`

**3.8 Class *AsynchConvergecast***

**Known Subclasses:** `datk.core.algs.AsynchConvergeHeight`

The abstract superclass of a class of Synchronous Algorithms that propagate information from the leaves of a BFS tree to its root.

Requires:

- Every Process knows `state['parent']` and `state['children']`

**3.8.1 Methods**

<b><code>is_root(self, p)</code></b>
--------------------------------------

<b><code>msgs_i(self, p)</code></b>
-------------------------------------

Overrides: <code>datk.core.distalgs.Algorithm.msgs_i</code>
---

<b><code>trans_i(self, p, msgs)</code></b>
--

Overrides: <code>datk.core.distalgs.Algorithm.trans_i</code>
--

<b><code>cleanup_i(self, p)</code></b>
--

Overrides: <code>datk.core.distalgs.Algorithm.cleanup_i</code>
--

<b><code>trans_root(self, p, msgs)</code></b>
---

<b><code>outpout_root(self, p)</code></b>
---

<b><code>initial_msg_to_parent(self, p)</code></b>
--

<b>trans_msg_to_parent</b> ( <i>self</i> , <i>p</i> , <i>msgs</i> )
---

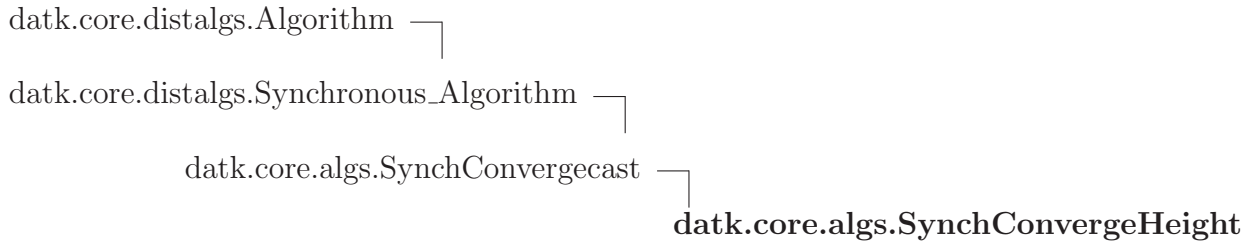
*Inherited from datk.core.distalgs.Asynchronous\_Algorithm(Section 4.8)*

run(), run\_process()

*Inherited from datk.core.distalgs.Algorithm(Section 4.5)*

\_\_call\_\_(), \_\_init\_\_(), cleanup(), count\_msg(), delete(), get(), halt(), halt\_i(), has(), increment(), set()

### 3.9 Class *SynchConvergeHeight*



Requires:

- BFS Tree

Effects:

- Root Process knows height of tree in state["height"]

#### 3.9.1 Methods

<b>cleanup_i</b> ( <i>self</i> , <i>p</i> )
---

Overrides: datk.core.distalgs.Algorithm.cleanup\_i

<b>initial_msg_to_parent</b> ( <i>self</i> , <i>p</i> )
---

Overrides: datk.core.algs.SynchConvergecast.initial\_msg\_to\_parent

<b>output_root</b> ( <i>self</i> , <i>p</i> )
---

<b>trans_msg_to_parent</b> ( <i>self</i> , <i>p</i> , <i>msgs</i> )
---

Overrides: datk.core.algs.SynchConvergecast.trans\_msg\_to\_parent

<b>trans_root</b> ( <i>self</i> , <i>p</i> , <i>msgs</i> )
--

Overrides: datk.core.algs.SynchConvergecast.trans_root
--

**Inherited from datk.core.algs.SynchConvergecast(Section 3.7)**

is\_root(), msgs\_i(), output\_root(), trans\_i()

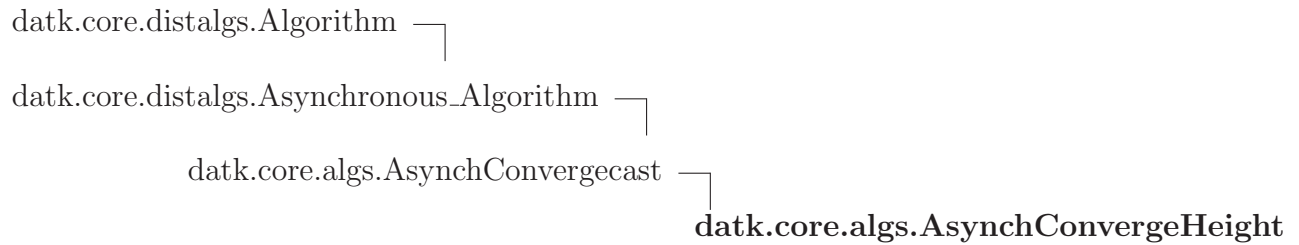
**Inherited from datk.core.distalgs.Synchronous\_Algorithm(Section 4.6)**

execute(), msgs(), round(), run(), trans()

**Inherited from datk.core.distalgs.Algorithm(Section 4.5)**

\_\_call\_\_(), \_\_init\_\_(), cleanup(), count\_msg(), delete(), get(), halt(), halt\_i(), has(), increment(), set()

### 3.10 Class AsyncConvergeHeight



Requires:

- BFS Tree

Effects:

- Root Process knows height of tree in state["height"]

#### 3.10.1 Methods

<b>cleanup_i</b> ( <i>self</i> , <i>p</i> )
---

Overrides: datk.core.distalgs.Algorithm.cleanup_i
---

<b>initial_msg_to_parent</b> ( <i>self</i> , <i>p</i> )
---

Overrides: datk.core.algs.AsyncConvergecast.initial_msg_to_parent
---

<b>output_root</b> ( <i>self</i> , <i>p</i> )
---

<b>trans_msg_to_parent</b> ( <i>self</i> , <i>p</i> , <i>msgs</i> )
---

Overrides: <i>datk.core.algs.AsyncConvergecast.trans_msg_to_parent</i>
--

<b>trans_root</b> ( <i>self</i> , <i>p</i> , <i>msgs</i> )
--

Overrides: <i>datk.core.algs.AsyncConvergecast.trans_root</i>
---

***Inherited from datk.core.algs.AsyncConvergecast(Section 3.8)***

*is\_root()*, *msgs\_i()*, *outpout\_root()*, *trans\_i()*

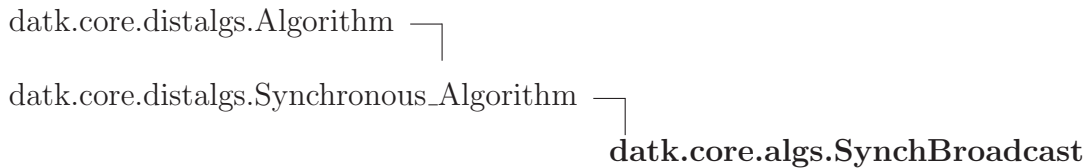
***Inherited from datk.core.distalgs.Asynchronous\_Algorithm(Section 4.8)***

*run()*, *run\_process()*

***Inherited from datk.core.distalgs.Algorithm(Section 4.5)***

*\_call\_()*, *\_init\_()*, *cleanup()*, *count\_msg()*, *delete()*, *get()*, *halt()*, *halt\_i()*, *has()*, *increment()*, *set()*

### 3.11 Class *SynchBroadcast*



Broadcasts a value stored in Process, *p*, to the BFS tree rooted at *p*

Requires:

- The attribute to be broadcasted must be specified in *self.params['attr']*
- BFS Tree with children pointers, where root node has *state[self.params['attr']]*

Effects:

- All Processes have *state[self.params['attr']] := the original value of in state[self.params['attr']]* of the root Process.

#### 3.11.1 Methods

<b>msgs_i</b> ( <i>self</i> , <i>p</i> )
--

Overrides: <i>datk.core.distalgs.Algorithm.msgs_i</i>
---

```
trans_i(self, p, msgs)
```

Overrides: `datk.core.distalgs.Algorithm.trans_i`

```
cleanup_i(self, p)
```

Overrides: `datk.core.distalgs.Algorithm.cleanup_i`

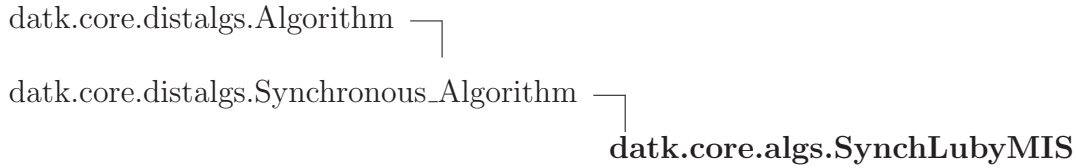
*Inherited from `datk.core.distalgs.Synchronous_Algorithm`(Section 4.6)*

`execute()`, `msgs()`, `round()`, `run()`, `trans()`

*Inherited from `datk.core.distalgs.Algorithm`(Section 4.5)*

`__call__()`, `__init__()`, `cleanup()`, `count_msg()`, `delete()`, `get()`, `halt()`, `halt_i()`, `has()`, `increment()`, `set()`

### 3.12 Class *SynchLubyMIS*



A randomized algorithm that constructs a Maximal Independent Set

The algorithm works in stages, each consisting of three rounds.

- Round 1: In the first round of a stage, the processes choose their respective vals and send them to their neighbors. By the end of round 1, when all the val messages have been received, the winners—that is, the processes in  $F$ —know who they are.
- Round 2: In the second round, the winners notify their neighbors. By the end of round 2, the losers—that is, the processes having neighbors in  $F$ —know who they are.
- Round 3: In the third round, each loser notifies its neighbors. Then all the involved processes—the winners, the losers, and the losers’ neighbors—remove the appropriate nodes and edges from the graph. More precisely, this means the winners and losers discontinue participation after this stage, and the losers’ neighbors remove all the edges that are incident on the newly removed nodes.

Requires:

- Every process knows `state['n']`, the size of the network

Effect:

- Every process knows `state['MIS']`. A boolean representing whether it is a member of the Maximal Independent Set found by Luby’s algorithm.



**3.12.1 Methods****msgs\_i**(*self*, *p*)Overrides: *datk.core.distalgs.Algorithm.msgs\_i***trans\_i**(*self*, *p*, *msgs*)Overrides: *datk.core.distalgs.Algorithm.trans\_i****Inherited from datk.core.distalgs.Synchronous\_Algorithm(Section 4.6)****execute()*, *msgs()*, *round()*, *run()*, *trans()****Inherited from datk.core.distalgs.Algorithm(Section 4.5)****\_\_call\_\_()*, *\_\_init\_\_()*, *cleanup()*, *cleanup\_i()*, *count\_msg()*, *delete()*, *get()*, *halt()*, *halt\_i()*, *has()*, *increment()*, *set()*

## 4 Module `datk.core.distalgs`

### 4.1 Variables

Name	Description
<code>__package__</code>	<b>Value:</b> <code>'datk.core'</code>

### 4.2 Class Message

**Known Subclasses:** `datk.core.algs.AsyncLCR.Leader_Declaration`, `datk.core.algs.SynchBFS.Search`, `datk.core.algs.SynchBFSAck.AckParent`, `datk.core.algs.SynchBFSAck.Search`

#### 4.2.1 Methods

```
__init__(self, algorithm, content=None)
```

```
__str__(self)
```

### 4.3 Class Process

A computing element located at a node of a network graph. Processes are identical except for their UID

#### 4.3.1 Methods

```
__init__(self, UID, state=None, in_nbrs=[], out_nbrs=[])
```

```
link_to(self, new_out_nbr)
```

Adds a new outgoing neighbor of the Process

```
bi_link(self, nbr)
```

```
output(self, key, val, silent=False)
```

```
send_to_all_neighbors(self, msg)
```

```
send_msg(self, msg, out_nbrs=None)
```

```
get_msgs(self, algorithm, in_nbrs=None)
```

```
add(self, algorithm)
```

```
terminate(self, algorithm)
```

```
__str__(self)
```

```
__repr__(self)
```

## 4.4 Class Network

**Known Subclasses:** `datk.core.networks.Bidirectional_Line`, `datk.core.networks.Bidirectional_Ring`, `datk.core.networks.Complete_Graph`, `datk.core.networks.Random_Line_Network`, `datk.core.networks.Unidirectional_Line`, `datk.core.networks.Unidirectional_Ring`

A collection of Processes that know `n`, the # of processes in the network.

### 4.4.1 Methods

```
__getitem__(self, i)
```

```
__init__(self, n, index_to_UID=None)
```

Creates a network of `n` disconnected Processes, with random distinct UUIDs, or as specified by the `index_to_UID` function

```
__iter__(self)
```

```
__len__(self)
```

```
__repr__(self)
```

```
add(self, algorithm)
```

```
clone(self)
```

```
draw(self)
```

```
index(self, p)
```

```
run(self, algorithm)
```

```
state(self)
```

## 4.5 Class Algorithm

**Known Subclasses:** `datk.core.distalgs.Asynchronous_Algorithm`, `datk.core.distalgs.Chain`, `datk.core.distalgs.Synchronous_Algorithm`

Abstract superclass for a distributed algorithm.

### 4.5.1 Methods

```
__init__(self, network=None, params={'draw': False, 'silent': True},  
name=None)
```

#### Parameters

**network:** [Optional] network. If specified, algorithm is immediately executed on network.

**params:** [Optional] runtime parameters.

**name:** [Optional] name of the Algorithm instance. Defaults to class name.

```
msgs_i(self, p)
```

```
trans_i(self, p, msgs)
```

```
halt_i(self, p)
```

```
cleanup_i(self, p)
```

```
cleanup(self)
```

```
__call__(self, network, params={})
```

```
run(self, network, params={})
```

```
halt(self)
```

```
count_msg(self, message_count)
```

```
set(self, process, state, value)
```

```
increment(self, process, state, inc=1)
```

```
has(self, process, state)
```

```
get(self, process, state)
```

```
delete(self, process, state)
```

## 4.6 Class *Synchronous\_Algorithm*

*datk.core.distalgs.Algorithm*

***datk.core.distalgs.Synchronous\_Algorithm***

**Known Subclasses:** *datk.core.distalgs.Compose*, *datk.core.distalgs.Do\_Nothing*, *datk.core.algs.FloodMax*, *datk.core.algs.LCR*, *datk.core.algs.SynchBFS*, *datk.core.algs.SynchBFSAck*, *datk.core.algs.SynchBroadcast*, *datk.core.algs.SynchConvergecast*, *datk.core.algs.SynchLubyMIS*

We assume that Processes take steps simultaneously, that is, that execution proceeds in synchronous rounds.

### 4.6.1 Methods

```
execute(self)
```

```
msgs(self)
```

```
round(self)
```

```
run(self, network, params={})
```

Overrides: *datk.core.distalgs.Algorithm.run*

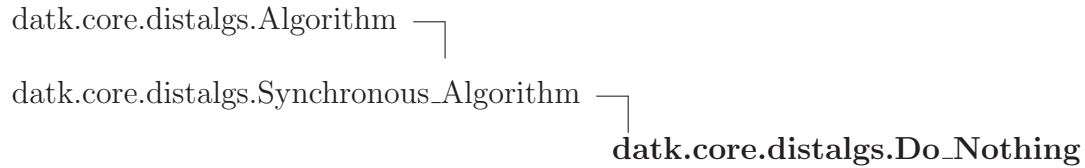
```
trans(self)
```

***Inherited from datk.core.distalgs.Algorithm(Section 4.5)***

*\_\_call\_\_()*, *\_\_init\_\_()*, *cleanup()*, *cleanup\_i()*, *count\_msg()*, *delete()*, *get()*, *halt()*,

halt\_i(), has(), increment(), msgs\_i(), set(), trans\_i()

## 4.7 Class *Do\_Nothing*



### 4.7.1 Methods

<b>trans_i</b> ( <i>self</i> , <i>p</i> , <i>messages</i> ) Overrides: datk.core.distalgs.Algorithm.trans_i
--

*Inherited from datk.core.distalgs.Synchronous\_Algorithm(Section 4.6)*

execute(), msgs(), round(), run(), trans()

*Inherited from datk.core.distalgs.Algorithm(Section 4.5)*

\_\_call\_\_(), \_\_init\_\_(), cleanup(), cleanup\_i(), count\_msg(), delete(), get(), halt(), halt\_i(), has(), increment(), msgs\_i(), set()

## 4.8 Class *Asynchronous\_Algorithm*



**Known Subclasses:** datk.core.algs.AsyncLCR, datk.core.algs.AsyncConvergecast

We assume that the separate Processes take steps in an arbitrary order, at arbitrary relative speeds.

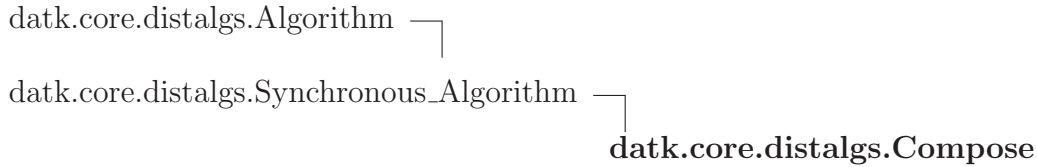
### 4.8.1 Methods

<b>run</b> ( <i>self</i> , <i>network</i> , <i>params</i> ={}) Overrides: datk.core.distalgs.Algorithm.run
---

<b>run_process</b> ( <i>self</i> , <i>process</i> )
---

***Inherited from `datk.core.distalgs.Algorithm`(Section 4.5)***

`__call__()`, `__init__()`, `cleanup()`, `cleanup_i()`, `count_msg()`, `delete()`, `get()`, `halt()`,  
`halt_i()`, `has()`, `increment()`, `msgs_i()`, `set()`, `trans_i()`

**4.9 Class Compose**

A `Synchronous_Algorithm` that is the composition of two synchronous algorithms running in parallel.

**4.9.1 Methods**

```
__init__(self, A, B, name=None, params={'draw': False, 'silent': True})
```

**Parameters**

**network:** [Optional] network. If specified, algorithm is immediately executed on network.

**params:** [Optional] runtime parameters.

**name:** [Optional] name of the `Algorithm` instance. Defaults to class name.

Overrides: `datk.core.distalgs.Algorithm.__init__`

```
msgs_i(self, p)
```

Overrides: `datk.core.distalgs.Algorithm.msgs_i`

```
trans_i(self, p, msgs)
```

Overrides: `datk.core.distalgs.Algorithm.trans_i`

```
halt_i(self, p)
```

Overrides: `datk.core.distalgs.Algorithm.halt_i`

```
cleanup_i(self, p)
```

Overrides: `datk.core.distalgs.Algorithm.cleanup_i`

```
run(self, network, params={})
```

Overrides: `datk.core.distalgs.Algorithm.run`

```
__repr__(self)
```

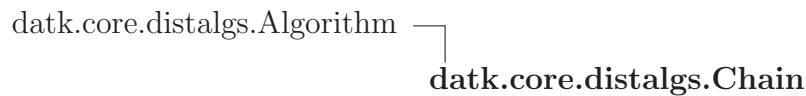
**Inherited from `datk.core.distalgs.Synchronous_Algorithm`(Section 4.6)**

`execute()`, `msgs()`, `round()`, `trans()`

**Inherited from `datk.core.distalgs.Algorithm`(Section 4.5)**

`__call__()`, `cleanup()`, `count_msg()`, `delete()`, `get()`, `halt()`, `has()`, `increment()`, `set()`

## 4.10 Class Chain



An Algorithm that is the result of sequentially running two algorithms

### 4.10.1 Methods

```
__init__(self, A, B, name=None, params={'draw': False, 'silent': False})
```

#### Parameters

**network:** [Optional] network. If specified, algorithm is immediately executed on network.

**params:** [Optional] runtime parameters.

**name:** [Optional] name of the Algorithm instance. Defaults to class name.

Overrides: `datk.core.distalgs.Algorithm.__init__`

```
run(self, network, params={})
```

Overrides: `datk.core.distalgs.Algorithm.run`

```
__repr__(self)
```

**Inherited from `datk.core.distalgs.Algorithm`(Section 4.5)**

`__call__()`, `cleanup()`, `cleanup_i()`, `count_msg()`, `delete()`, `get()`, `halt()`, `halt_i()`, `has()`, `increment()`, `msgs_i()`, `set()`, `trans_i()`



## 5 Module `datk.core.imports`

### 5.1 Variables

Name	Description
<code>TIMEOUT</code>	<b>Value:</b> 5
<code>__package__</code>	<b>Value:</b> <code>'datk.core'</code>
<code>failed_tests</code>	<b>Value:</b> <code>set([])</code>
<code>lock</code>	<b>Value:</b> <code>Lock()</code>
<code>num_tests</code>	<b>Value:</b> 0

## 6 Module **datk.core.networks**

### 6.1 Variables

Name	Description
<code>__package__</code>	Value: <code>'datk.core'</code>

### 6.2 Class **Unidirectional\_Ring**

`datk.core.distalgs.Network` └─ **`datk.core.networks.Unidirectional_Ring`**

A Network of *n* Processes arranged in a ring. Each edge is directed from a Process to its clockwise neighbor, that is, messages can only be sent in a clockwise direction.

#### 6.2.1 Methods

**`__init__(self, n, index_to_UID=None)`**

Creates a network of *n* disconnected Processes, with random distinct UUIDs, or as specified by the `index_to_UID` function

Overrides: `datk.core.distalgs.Network.__init__` extit(inherited documentation)

*Inherited from `datk.core.distalgs.Network`(Section 4.4)*

`__getitem__()`, `__iter__()`, `__len__()`, `__repr__()`, `add()`, `clone()`, `draw()`, `index()`, `run()`, `state()`

### 6.3 Class **Bidirectional\_Ring**

`datk.core.distalgs.Network` └─ **`datk.core.networks.Bidirectional_Ring`**

A Network of *n* Processes arranged in a ring. Each edge between a Process and its neighbor is undirected, that is, messages can be sent in both the clockwise and the counterclockwise directions.

### 6.3.1 Methods

```
__init__(self, n, index_to_UID=None)
```

Creates a network of *n* disconnected Processes, with random distinct UUIDs, or as specified by the *index\_to\_UID* function

Overrides: *datk.core.distalgs.Network.\_\_init\_\_* *exitit*(inherited documentation)

*Inherited from **datk.core.distalgs.Network**(Section 4.4)*

```
__getitem__(), __iter__(), __len__(), __repr__(), add(), clone(), draw(), index(), run(), state()
```

## 6.4 Class *Unidirectional\_Line*

```
datk.core.distalgs.Network └─ datk.core.networks.Unidirectional_Line
```

A Network of *n* Processes arranged in a line. Each edge is directed from a Process to its clockwise neighbor, that is, messages can only be sent in a clockwise direction.

### 6.4.1 Methods

```
__init__(self, n, index_to_UID=None)
```

Creates a network of *n* disconnected Processes, with random distinct UUIDs, or as specified by the *index\_to\_UID* function

Overrides: *datk.core.distalgs.Network.\_\_init\_\_* *exitit*(inherited documentation)

*Inherited from **datk.core.distalgs.Network**(Section 4.4)*

```
__getitem__(), __iter__(), __len__(), __repr__(), add(), clone(), draw(), index(), run(), state()
```

## 6.5 Class *Bidirectional\_Line*

```
datk.core.distalgs.Network └─ datk.core.networks.Bidirectional_Line
```

A Network of *n* Processes arranged in a line. Each edge between a Process and its neighbor is undirected, that is, messages can be sent in both the clockwise and the counterclockwise

directions.

### 6.5.1 Methods

```
__init__(self, n, index_to_UID=None)
```

Creates a network of n disconnected Processes, with random distinct UUIDs, or as specified by the `index_to_UID` function

Overrides: `datk.core.distalgs.Network.__init__` `exitit`(inherited documentation)

*Inherited from `datk.core.distalgs.Network`(Section 4.4)*

```
__getitem__(), __iter__(), __len__(), __repr__(), add(), clone(), draw(), index(), run(), state()
```

## 6.6 Class `Complete_Graph`



A Network of n Processes arranged at the vertices of a Complete undirected graph of size n.

### 6.6.1 Methods

```
__init__(self, n, index_to_UID=None)
```

Creates a network of n disconnected Processes, with random distinct UUIDs, or as specified by the `index_to_UID` function

Overrides: `datk.core.distalgs.Network.__init__` `exitit`(inherited documentation)

*Inherited from `datk.core.distalgs.Network`(Section 4.4)*

```
__getitem__(), __iter__(), __len__(), __repr__(), add(), clone(), draw(), index(), run(), state()
```

## 6.7 Class `Random_Line_Network`



A Network of  $n$  processes arranged randomly at the vertices of a connected undirected line graph of size  $n$ . Additional pairs of vertices are connected at random with a probability that is inversely proportional to the difference in their positions on the line.

For example, the Process at index 3 is guaranteed to be connected to the Process at index 4, and is more likely to be connected to the Process at index 5 than to the Process at index 8.

### 6.7.1 Methods

<b><code>__init__(self, n, sparsity=1)</code></b>
---

sparsity = 0 $\rightarrow$ a Complete_Graph( $n$ ) sparsity = infinity $\rightarrow$ a Bidirectional_Line( $n$ )
---

Overrides: <code>datk.core.distalgs.Network.__init__</code>
---

*Inherited from `datk.core.distalgs.Network`(Section 4.4)*

`__getitem__()`, `__iter__()`, `__len__()`, `__repr__()`, `add()`, `clone()`, `draw()`, `index()`, `run()`, `state()`

## 7 Module *datk.core.test*er

### 7.1 Functions

```
test(f=None, timeout=5, main_thread=False, test=True)
```

```
print_with_underline(text)
```

```
summarize()
```

### 7.2 Variables

Name	Description
TIMEOUT	Value: 5
lock	Value: Lock()
num_tests	Value: 0
failed_tests	Value: set([])
__package__	Value: 'datk.core'

## 8 Package `datk.tests`

### 8.1 Modules

- **helpers**: Helper functions for tests in `tests.py`  
(Section 9, p. 32)
- **networks\_tests**: Network Test Suite  
(Section 10, p. 33)
- **tests**: Algorithm Test Suite  
(Section 11, p. 34)

### 8.2 Variables

Name	Description
<code>--package--</code>	<b>Value:</b> None

## 9 Module `datk.tests.helpers`

Helper functions for tests in `tests.py`

### 9.1 Functions

<b>testLeaderElection</b> ( <i>network</i> , <i>isLeader</i> =<function <lambda> at 0x1d1a430>, <i>isNonleader</i> =<function <lambda> at 0x1d1a470>)
---

Asserts that exactly one Process is Leader, and all other processes are Non-Leader
--

<b>testBroadcast</b> ( <i>network</i> , <i>attr</i> )
---

Asserts that <code>p.state[attr]</code> is identical for all processes <code>p</code>
---

<b>testBFS</b> ( <i>network</i> )
-----------------------------------

Asserts that every Process, <code>p</code> , knows 'parent', and there exists exactly one Process where 'parent' is None
--

<b>testBFSWithChildren</b> ( <i>network</i> )
---

Asserts that every Process, <code>p</code> , knows 'parent' and 'children', and there exists exactly one Process where 'parent' is None
---

<b>testLubyMIS</b> ( <i>network</i> )
---------------------------------------

Asserts that every process knows a boolean value, 'MIS', and that the Processes where 'MIS' is True form a set that is both independent and maximal.
--

### 9.2 Variables

Name	Description
<code>--package--</code>	Value: <code>'datk.tests'</code>



## 10 Module `datk.tests.networks_tests`

Network Test Suite

Tests Networks defined in `networks.py` by visual inspection

### 10.1 Variables

Name	Description
<code>DRAW_RANDOM</code>	<b>Value:</b> <code>None</code>
<code>DRAW_HUGE_RANDOM</code>	<b>Value:</b> <code>None</code>
<code>DRAW_UNI_RING</code>	<b>Value:</b> <code>None</code>
<code>DRAW_BI_RING</code>	<b>Value:</b> <code>None</code>
<code>DRAW_COMPLETE_GRAPH</code>	<b>Value:</b> <code>None</code>
<code>DRAW_UNI_LINE</code>	<b>Value:</b> <code>None</code>
<code>DRAW_BI_LINE</code>	<b>Value:</b> <code>None</code>
<code>TIMEOUT</code>	<b>Value:</b> <code>5</code>
<code>__package__</code>	<b>Value:</b> <code>'datk.tests'</code>
<code>failed_tests</code>	<b>Value:</b> <code>set([])</code>
<code>lock</code>	<b>Value:</b> <code>Lock()</code>
<code>num_tests</code>	<b>Value:</b> <code>0</code>

## 11 Module *datk.tests.tests*

Algorithm Test Suite

Tests algorithms defined in *algs.py*

### 11.1 Functions

**`configure_ipython()`**

Convenient helper function to determine if environment is ipython. Note that drawing is only safe in ipython qtconsole with matplotlib inline. If environment is IPython, returns True and configures IPython. Else returns False.

**`LCR_UNI_RING()`**

**`LCR_BI_RING()`**

**`ASYNC_LCR_UNI_RING()`**

**`ASYNC_LCR_BI_RING()`**

**`FLOODMAX_UNI_RING()`**

**`FLOODMAX_BI_RING()`**

**`FLOODMAX_BI_LINE()`**

**`FLOODMAX_COMPLETE_GRAPH()`**

**`FLOODMAX_RANDOM_GRAPH()`**

**`SYNCH_BFS()`**

**`SYNCH_BFS_ACK()`**

**`SYNCH_CONVERGE_HEIGHT()`**

**`SYNCH_BROADCAST_HEIGHT()`**

ASYNCH_BROADCAST_HEIGHT()
---------------------------

send_receive_msgs()
---------------------

SYNCH_DO_NOTHING()
--------------------

COMPOSE_SYNCH_LCR_AND_DO_NOTHING()
------------------------------------

COMPOSE_SYNCH_LCR()
---------------------

CHAIN_BROADCAST_HEIGHT()
--------------------------

SYNCH_LUBY_MIS_BI_RING()
--------------------------

SYNCH_LUBY_MIS()
------------------

## 11.2 Variables

Name	Description
in_ipython	Value: False
test_params	Value: {'draw': False, 'silent': True}
TIMEOUT	Value: 5
__package__	Value: 'datk.tests'
failed_tests	Value: set([])
lock	Value: Lock()
num_tests	Value: 0

## Index

- datk (*package*), 4
  - datk.core (*package*), 5
    - datk.core.algs (*module*), 6–17
    - datk.core.distalgs (*module*), 18–24
    - datk.core.imports (*module*), 25
    - datk.core.networks (*module*), 26–29
    - datk.core.testers (*module*), 30
  - datk.tests (*package*), 31
    - datk.tests.helpers (*module*), 32
    - datk.tests.networks\_tests (*module*), 33
    - datk.tests.tests (*module*), 34–35