PartitionGraph class – manual construction

Use the *add_node* and *add_edge* methods to create a partition graph. All nodes must be hashable (e.g. strings).

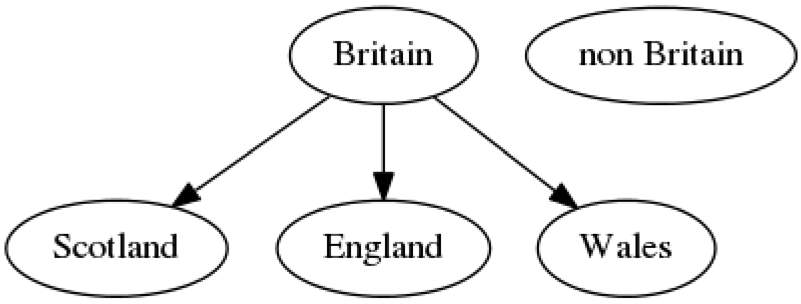
```
>>> import partition_graph
>>> g = partition_graph.PartitionGraph()
>>> for n in ['England', 'Scotland', 'Wales', 'Britain', 'non Britain']:
      g.add_node(n)


>>> for e in [('Britain','England'), ('Britain','Scotland'),
('Britain','Wales')]:
      g.add_edge(e)


('Britain', 'England')
('Britain', 'Scotland')
('Britain', 'Wales')
>>>
```

If Graphviz[1] is installed an image of the graph can be saved to file.

```
>>> g.to_image('geog.png', format='png')
>>>
```



The *del_node* and *del_edge* methods can be used to delete nodes and edges.

Structural validity of the graph can be checked using the *structure_is_valid* method. This simply checks that the subtree induced by each node and its descendants is a tree.

```
>>> g.structure_is_valid()
True
>>>
```

The graph can be converted to a JSON representation for storage.

1 http://www.graphviz.org/

```
>>> g.to_json()
'[["Scotland", "Britain", "non Britain", "England", "Wales"], [["Britain",
"Scotland"], ["Britain", "England"], ["Britain", "Wales"]]]'
>>>
```

A classmethod is provided to generate a partition graph from its JSON representation.

```
>>> acopy = partition_graph.PartitionGraph.from_json(g.to_json())
>>> g == acopy
True
>>>
```

(A better way to generate a copy is to simply use the *copy* method.) There are corresponding *to_file* and a *from_file* classmethods.

```
>>> g.to_file('geog.dat')
>>> acopy = partition_graph.PartitionGraph.from_file('geog.dat')
>>> g == acopy
True
>>>
```

The sink (leaf) nodes reachable from a node can be generated.

```
>>> gen_sinks = g.reachable('Britain')
>>> for sink in gen_sinks:
      print sink,


Scotland England Wales
>>>
```

The sink (leaf) node partition representation can be generated using the *leaf_partition* method.

```
>>> g.leaf_partition(['Britain', 'non Britain'])
[set(['England', 'Scotland', 'Wales']), set(['non Britain'])]
>>>
```

This automatically checks that the argument (list of nodes) is a valid categorisation with respect to the partition graph.

```
>>> g.leaf_partition(['Britain'])

Traceback (most recent call last):
  File "<pyshell#140>", line 1, in <module>
```

```
    g.leaf_partition(['Britain'])
  File "partition_graph.py", line 114, in leaf_partition
    raise ValueError('Not all leaf nodes are reachable from specified
nodes')
ValueError: Not all leaf nodes are reachable from specified nodes
>>>
```

There are also *remove* and *discard* methods for removing nodes. A node can only be removed if it does not render the partition graph invalid.

```
>>> g.remove('Wales')

Traceback (most recent call last):
  File "<pyshell#141>", line 1, in <module>
    g.remove('Wales')
  File "partition_graph.py", line 176, in remove
    raise ValueError('Cannot remove leaf node')
ValueError: Cannot remove leaf node
>>>
```

The *discard* method simply calls the *remove* method but suppresses the exception and leaves the graph unchanged if the node cannot be removed. Note: the *add_node*, *del_node*, *add_edge* and *del_edge* methods allow arbitrary changes to the graph. The *remove* and *discard* methods are designed for simplifying an existing, valid graph without making it invalid.

The *prune* method allows the user to specify a collection of nodes that must be retained and attempts to remove the remaining nodes. For non sink nodes, they are simply removed and the relevant edges added to maintain validity. If the aggressive argument is True, then it attempts to remove sink nodes too, which requires more intricate handling (they cannot be considered individually).

```
>>> g.prune(['Britain'], aggressive=True)
>>> g
Britain []
non Britain []

>>>
```

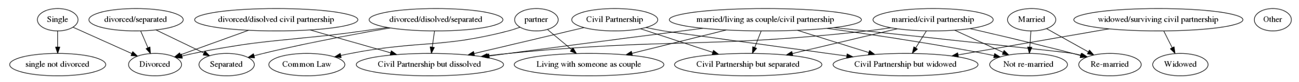
Above, England, Scotland and Wales can be removed because they have a single common parent which can become a new sink node. The non Britain node must be retained in order to maintain an exhaustive set of sink nodes.

The *deepen* and *flatten* methods adjust the layout of the graph in place. (The corresponding *deepened* and *flattened* methods return deepened and flattened copies respectively, leaving the original graph unchanged.)

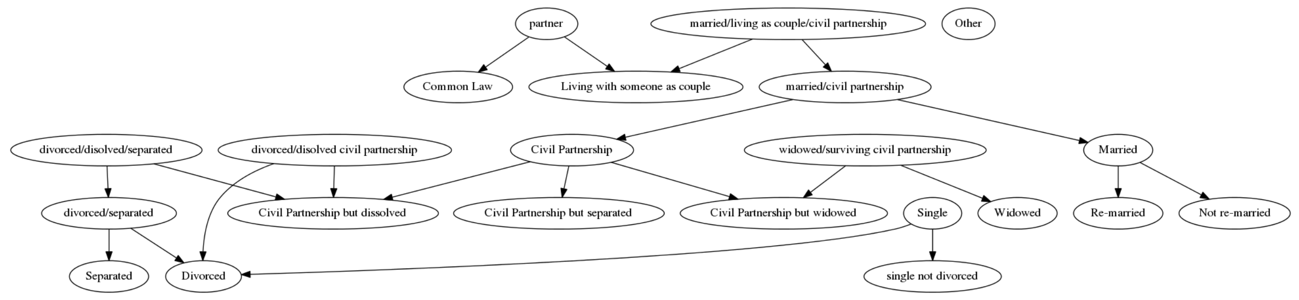
The marital status graph (constructed from the original spreadsheet KVM model) is a little more complex and is better for illustration.

```
>>> ms = partition_graph.marital_status()
>>> ms.flattened().to_image('ms_flat.png')
>>> ms.deepened().to_image('ms_deep.png')
>>>
```

The flattened structure is more efficient for purposes such as generating leaf partitions.


Single
divorced/separated
divorced/disolved civil partnership
divorced/disolved/separated
partner
Civil Partnership
married/living as couple/civil partnership
married/civil partnership
Married
widowed/surviving civil partnership
Other
single not divorced
Divorced
Separated
Common Law
Civil Partnership but dissolved
Living with someone as couple
Civil Partnership but separated
Civil Partnership but widowed
Not re-married
Re-married
Widowed

The deepened structure is easier on the eye.


partner
married/living as couple/civil partnership
Other
Common Law
Living with someone as couple
married/civil partnership
divorced/disolved/separated
divorced/disolved civil partnership
Civil Partnership
widowed/surviving civil partnership
Married
divorced/separated
Civil Partnership but dissolved
Civil Partnership but separated
Civil Partnership but widowed
Single
Widowed
Re-married
Not re-married
Separated
Divorced
single not divorced

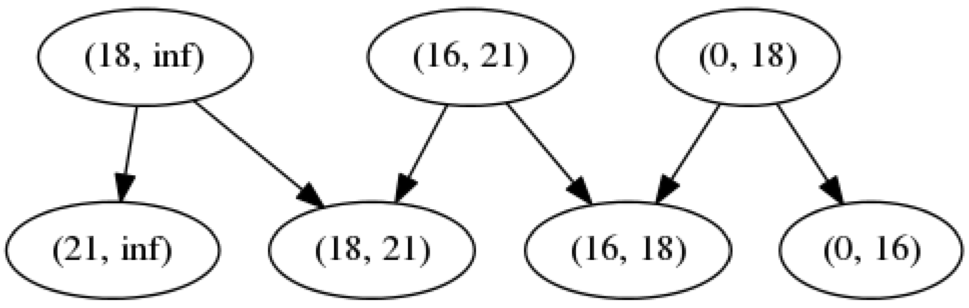
There are a few potential issues with the Marital Status graph – not least the 'Other' category. The graph implies that it denotes any form of martial status not covered by the other nodes. But as we have 'married' and 'single' it would seem to be a null category – and should not be in the graph. It is also a poor choice of name, as there will probably be several categorizations that contain 'other' categories with different meanings. Also, we have sink nodes such as 'Widowed' and 'single not divorced' which are clearly not mutually exclusive. Manual construction is not necessarily an easy task.

IntervalGraph and SetGraph classes – non-manual construction

Categorizations based on intervals (say, age ranges) or sets can be used to construct graphs automatically. A simple *Interval* class is supplied. It does not explicitly handle open / closed endpoints, disjoint intervals or non-numeric intervals (e.g. lexicographic ordering of strings). The classes that do are still pre-alpha. The *SetGraph* class simply uses the built in Python *frozenset* type.

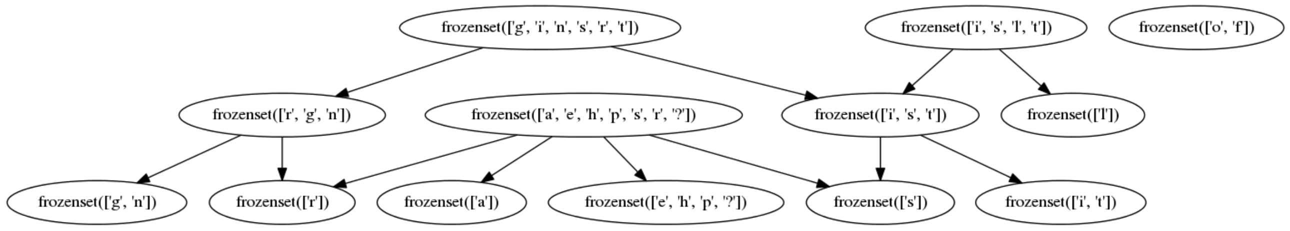
For the *IntervalGraph* class a sequence of pairs (of endpoints) can be supplied to the standard constructor. Additional pairs can be added using the *add* method (N.B. the basic *add_node* etc. methods are available, but they do not handle automatic construction).

```
>>> int_graph = partition_graph.IntervalGraph([(0,16), (16,21),
(21,float('inf'))])
>>> for n in [(0,18), (18,float('inf'))]:
      int_graph.add(n)
```



The *SetGraph* will take any iterables containing hashable items as nodes (converting them to frozensets).

```
>>> set_graph = partition_graph.SetGraph('a list of strings
perhaps?'.split())
>>> set_graph.to_image('set_graph.png')
>>>
```


frozenset(['g', 'i', 'n', 's', 'r', 't'])
frozenset(['i', 's', 'l', 't'])
frozenset(['o', 'f'])
frozenset(['r', 'g', 'n'])
frozenset(['a', 'e', 'h', 'p', 's', 'r', '?'])
frozenset(['i', 's', 't'])
frozenset(['l'])
frozenset(['g', 'n'])
frozenset(['r'])
frozenset(['a'])
frozenset(['e', 'h', 'p', '?'])
frozenset(['s'])
frozenset(['i', 't'])

The *SetGraph* is useful when we have a finite set of distinct items and categories can be expressed as collections of items. In the case of hierarchies such as geographical location this can be simplified further as these result in true trees.

The *IntervalGraph* and *SetGraph* classes also have an *is_valid* method that checks more than structure. Mutual exclusivity and exhaustiveness of child sets and sink nodes is tested. An optional cover argument allows the coverage to be tested. The value must be some object that will compare equal to the union of the sink nodes if the coverage is correct.

```
>>> int_graph.is_valid()
True
>>> int_graph.is_valid(cover=partition_graph.Interval(0,float('inf')))
True
>>> int_graph.is_valid(cover=partition_graph.Interval(1,float('inf')))
incorrect cover
False
>>>
>>> set_graph.is_valid()
True
>>> set_graph.is_valid(cover=set('alistofstringsperhaps?'))
True
>>> set_graph.is_valid(cover=set('alistofstringsperhaps?!'))
incorrect cover
False
>>>
```