Raymond BISDORFF

Algorithmic Decision Making with Python Resources

Tutorials and Advanced Topics

July 4, 2021

Springer

Contents

Chapter 1
About split, interval and permutation graphs

Abstract

Contents

1.1 A multiply perfect graph

Following Martin Golumbic (see [GOL-2004] p. 149), we call a given graph g:

- *Comparability* graph when *g* is *transitively orientable*;
- *Triangulated* graph when *g* does not contain any *chordless cycle* of length 4 and more;
- *Interval* graph when *g* is *triangulated* and its dual $-g$ is a *comparability* graph;
- *Permutation* graph when *g* and its dual $-g$ bothe are *comparability* graphs;
- *Split* graph when *g* and its dual $-g$ bothe are *triangulated* graphs.

To illustrate these *perfect* graph classes, we will generate from 8 intervals, randomly chosen in the default integer range $[0, 10]$, a *RandomIntervalIntersectionsGraph* instance *g* (see Line 2 below).

```
1  >>> from graphs import\
2  ...          RandomIntervalIntersectionsGraph
3  >>> g = RandomIntervalIntersectionsGraph(\
4  ...                          order=8,seed=100)
5  >>> g
6   *------- Graph instance description ------*
7    Instance class   : RandomIntervalIntersectionsGraph
8    Instance name    : randIntervalIntersections
9    Seed             : 100
10   Graph Order      : 8
11   Graph Size       : 23
12   Valuation domain : [-1.0; 1.0]
13   Attributes       : ['seed', 'name', 'order',
14         'intervals', 'vertices', 'valuationDomain',
15         'edges', 'size', 'gamma']
16 >>> print(g.intervals)
17   [(2,7),(2,7),(5,6),(6,8),(1,8),(1,1),(4,7),(0,10)]
```

With `seed = 100`, we obtain here an interval graph, in fact apperfect graph, which is *conjointly* a triangulated, a comparability, a split and a permutation graph (see Listing 1.1 Lines 6, 10, 14).

Listing 1.1 Testing perfect graph categories.

```
>>> g.isPerfectGraph(Comments=True)
  Graph randIntervalIntersections is perfect !
>>> g.isIntervalGraph(Comments=True)
  Graph 'randIntervalIntersections' is triangulated.
  Graph 'dual_randIntervalIntersections' is transitively orientable.
  => Graph 'randIntervalIntersections' is an interval graph.
>>> g.isSplitGraph(Comments=True)
  Graph 'randIntervalIntersections' is triangulated.
  Graph 'dual_randIntervalIntersections' is triangulated.
  => Graph 'randIntervalIntersections' is a split graph.
>>> g.isPermutationGraph(Comments=True)
  Graph 'randIntervalIntersections' is transitively orientable.
  Graph 'dual_randIntervalIntersections' is transitively orientable.
  => Graph 'randIntervalIntersections' is a permutation graph.
>>> print(g.computePermutation())
  ['v5', 'v6', 'v4', 'v2', 'v1', 'v3', 'v7', 'v8']
  ['v8', 'v6', 'v1', 'v2', 'v3', 'v4', 'v7', 'v5']
  [8, 2, 6, 5, 7, 4, 3, 1]
>>> g.exportGraphViz('randomSplitGraph')
  *---- exporting a dot file for GraphViz tools ---------*
   Exporting to randomSplitGraph.dot
   fdp -Tpng randomSplitGraph.dot -o randomSplitGraph.png
```

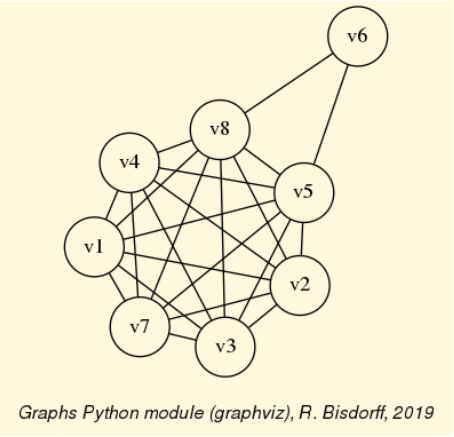


Fig. 1.1 A conjointly triangulated, comparability, interval, permutation and split graph

In Fig. 1.1 we may readily recognize the essential characteristic of split graphs, namely being always splitable into two disjoint sub-graphs: an *independent choice* 'v6' and a *clique* {'v1', 'v2', 'v3', 'v4', 'v5', 'v7', 'v8'}; which explains their name.

Notice however that the four properties:

1. g is a *comparability* graph;
2. g is a *co-comparability* graph, i.e. $-g$ is a comparability graph;
3. g is a *triangulated* graph;
4. g is a *co-triangulated* graph, i.e. $-g$ is a comparability graph;

are independent of one another (see [GOL-2004] p. 275).

1.2 Who is the liar?

Claude BERGE's famous mystery story (see [GOL-2004] p.20) may well illustrate the importance of being an *interval* graph.

Suppose the file `berge.py` Footnote[18] contains the following `Graph` instance data:

```
vertices = {
  'A': {'name': 'Abe', 'shortName': 'A'},
  'B': {'name': 'Burt', 'shortName': 'B'},
  'C': {'name': 'Charlotte', 'shortName': 'C'},
  'D': {'name': 'Desmond', 'shortName': 'D'},
  'E': {'name': 'Eddie', 'shortName': 'E'},
  'I': {'name': 'Ida', 'shortName': 'I'},
  }
valuationDomain = {'min':-1,'med':0,'max':1}
edges = {
  frozenset(['A','B']) : 1,
  frozenset(['A','C']) : -1,
  frozenset(['A','D']) : 1,
  frozenset(['A','E']) : 1,
  frozenset(['A','I']) : -1,
  frozenset(['B','C']) : -1,
  frozenset(['B','D']) : -1,
  frozenset(['B','E']) : 1,
  frozenset(['B','I']) : 1,
  frozenset(['C','D']) : 1,
  frozenset(['C','E']) : 1,
  frozenset(['C','I']) : 1,
  frozenset(['D','E']) : -1,
  frozenset(['D','I']) : 1,
  frozenset(['E','I']) : 1,
  }
```

Six professors (labeled 'A', 'B', 'C', 'D', 'E' and 'I') had been to the library on the day that a rare tractate was stolen. Each entered once, stayed for some time, and then left. If two professors were in the library at the same time, then at least one of them saw the other. Detectives questioned the professors and gathered the testimonies that 'A' saw 'B' and 'E'; 'B' saw 'A' and 'I'; 'C' saw 'D' and 'I'; 'D' saw 'A' and 'I'; 'E' saw 'B' and 'I'; and 'I' saw 'C' and 'E'. This data is gathered in the previous file, where each positive edge $\{x,y\}$ models the testimony that, either x saw y, or y saw x.

```
>>> from graphs import Graph
>>> g = Graph('berge')
>>> g.showShort()
  *---- short description of the graph ----*
   Name             : 'berge'
   Vertices         :  ['A', 'B', 'C', 'D', 'E', 'I']
   Valuation domain :  {'min': -1, 'med': 0, 'max': 1}
   Gamma function   :
    A -> ['D', 'B', 'E']
```

```
    B -> ['E', 'I', 'A']
    C -> ['E', 'D', 'I']
    D -> ['C', 'I', 'A']
    E -> ['C', 'B', 'I', 'A']
    I -> ['C', 'E', 'B', 'D']
>>> g.exportGraphViz('berge1')
  *---- exporting a dot file for GraphViz tools ---------*
   Exporting to berge1.dot
   fdp -Tpng berge1.dot -o berge1.png
```

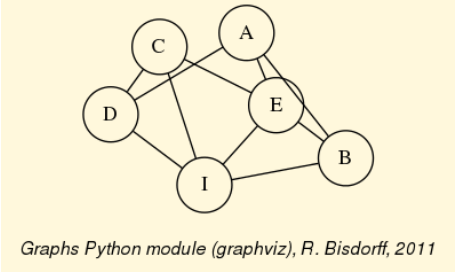


Fig. 1.2 Graph representation of the testimonies of the professors

From graph theory we know that time interval intersections graphs must in fact be interval graphs, i.e. *triangulated* and *co-comparative* graphs. The testimonies graph should therefore not contain any chordless cycle of four and more vertices. Now, the presence or not of such chordless cycles in the testimonies graph may be checked as follows:

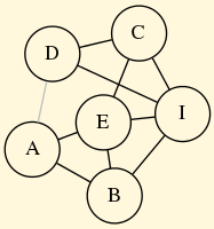
```
>>> g.computeChordlessCycles()
  Chordless cycle certificate: ['D','C','E','A','D']
  Chordless cycle certificate: ['D','I','E','A','D']
  Chordless cycle certificate: ['D','I','B','A','D']
  [(['D','C','E','A','D'],frozenset({'C','D','E','A'})),
  (['D','I','E','A','D'],frozenset({'D','E','I','A'})),
  (['D','I','B','A','D'],frozenset({'D','B','I','A'}))]
```

We see in the Listing above three intersection cycles of length 4, which is impossible to occur on the linear time line. Obviously one professor lied!

And it is *D* ; if we put to doubt his testimony that he saw 'A' (see Line 1 below), we obtain indeed a triangulated graph instance whose dual is a comparability graph.

```
>>> g.setEdgeValue( ('D','A'), 0)
>>> g.showShort()
  *---- short description of the graph ----*
   Name             : 'berge'
   Vertices         :  ['A','B','C','D','E','I']
   Valuation domain :  {'med': 0,'min': -1,'max': 1}
   Gamma function   :
    A -> ['B', 'E']
    B -> ['A', 'I', 'E']
    C -> ['I', 'E', 'D']
    D -> ['I', 'C']
```

```
    E -> ['A', 'I', 'B', 'C']
    I -> ['B', 'E', 'D', 'C']
>>> g.isIntervalGraph(Comments=True)
  Graph 'berge' is triangulated.
  Graph 'dual_berge' is transitively orientable.
  => Graph 'berge' is an interval graph.
>>> g.exportGraphViz('berge2')
  *---- exporting a dot file for GraphViz tools ---------*
   Exporting to berge2.dot
   fdp -Tpng berge2.dot -o berge2.png
```



Graphs Python module (graphviz), R. Bisdorff, 2011

Fig. 1.3 The triangulated testimonies graph

1.3 Generating permutation graphs

A graph is called a *permutation* or *inversion* graph if there exists a permutation of its list of vertices such that the graph is isomorphic to the inversions operated by the permutation in this list (see [GOL-2004] Chapter 7, pp 157-170). This kind is also part of the class of perfect graphs.

```
>>> from graphs import PermutationGraph
>>> g = PermutationGraph(permutation = [4,3,6,1,5,2])
>>> g
  *------- Graph instance description ------*
   Instance class   : PermutationGraph
   Instance name    : permutationGraph
   Graph Order      : 6
   Permutation      : [4, 3, 6, 1, 5, 2]
   Graph Size       : 9
   Valuation domain : [-1.00; 1.00]
   Attributes       : ['name', 'vertices', 'order',
                  'permutation', 'valuationDomain',
                  'edges', 'size', 'gamma']
>>> g.isPerfectGraph()
  True
>>> g.exportGraphViz()
  *---- exporting a dot file for GraphViz tools ---------*
   Exporting to permutationGraph.dot
```

```
   fdp -Tpng permutationGraph.dot\
       -o permutationGraph.png
```

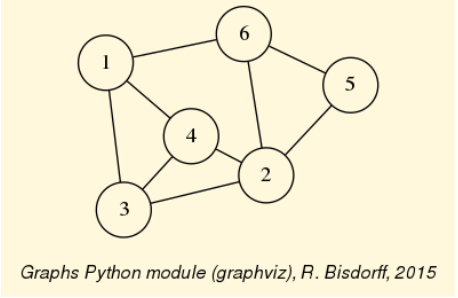


Fig. 1.4 The default DIGRAPH3 permutation graph

By using color sorting queues, the minimal vertex coloring for a permutation graph is computable in $O(nlog(n))$ (see [GOL-2004]).

```
>>> g.computeMinimalVertexColoring(Comments=True)
    vertex 1: lightcoral
    vertex 2: lightcoral
    vertex 3: lightblue
    vertex 4: gold
    vertex 5: lightblue
    vertex 6: gold
>>> g.exportGraphViz(fileName='coloredPermutationGraph',\
...                  WithVertexColoring=True)
  *---- exporting a dot file for GraphViz tools -------*
   Exporting to coloredPermutationGraph.dot
   fdp -Tpng coloredPermutationGraph.dot\
       -o coloredPermutationGraph.png
```

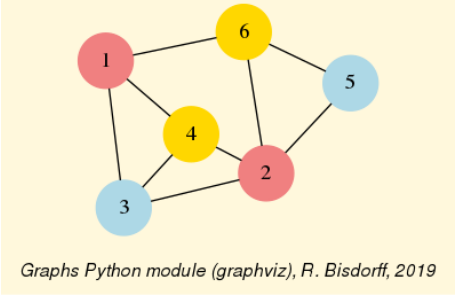


Fig. 1.5 Minimal vertex coloring of the permutation graph.

The correspondingly colored *matching diagram* of the nine inversions –the actual edges of the permutation graph–, which are induced by the given [4,3,6,1,5,2] permutation, may as well be drawn with the graphviz `neato` layout and explicitly positioned horizontal lists of vertices (see Fig. 1.6).

```
>>> g.exportPermutationGraphViz(\
...                     WithEdgeColoring=True)
  *---- exporting a dot file for GraphViz tools ------*
   Exporting to perm_permutationGraph.dot
   neato -n -Tpng perm_permutationGraph.dot\
             -o perm_permutationGraph.png
```

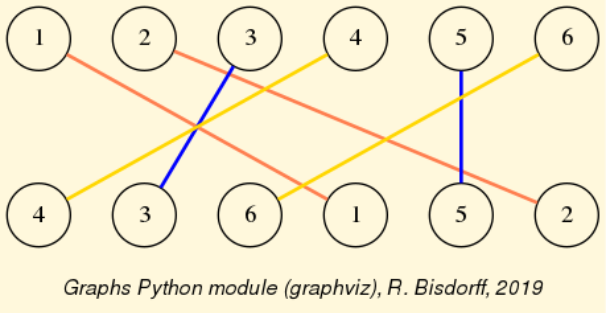


Fig. 1.6 Colored matching diagram of the permutation [4,3,6,1,5,2].

As mentioned before, a permutation graph and its dual are *transitively orientable*. The `transitiveOrientation()` method constructs from a given permutation graph a digraph where each edge of the permutation graph is converted into an arc oriented in increasing alphabetic order of the adjacent vertices' keys (see [GOL-2004]). This orientation of the edges of a permutation graph is always transitive and delivers a *transitive ordering* of the vertices.

```
>>> dg = g.transitiveOrientation()
>>> dg
  *------- Digraph instance description ------*
   Instance class   : TransitiveDigraph
   Instance name    : oriented_permutationGraph
   Digraph Order      : 6
   Digraph Size       : 9
   Valuation domain : [-1.00; 1.00]
   Determinateness  : 100.000
   Attributes       : ['name', 'order', 'actions',
                     'valuationdomain', 'relation',
                     'gamma', 'notGamma', 'size']
>>> print('Transitivity degree: %.3f' %\
...          dgd.computeTransitivityDegree() )
   Transitivity degree: 1.000
>>> dg.exportGraphViz(fileName='orientedPermGraph')
  *---- exporting a dot file for GraphViz tools -----*
   Exporting to orientedPermGraph.dot
   dot -Grankdir=TB -Tpng orientedPermGraph.dot\
                     -o orientedPermGraph.png
```

The dual of a permutation graph is again a permutation graph and as such also transitively orientable.

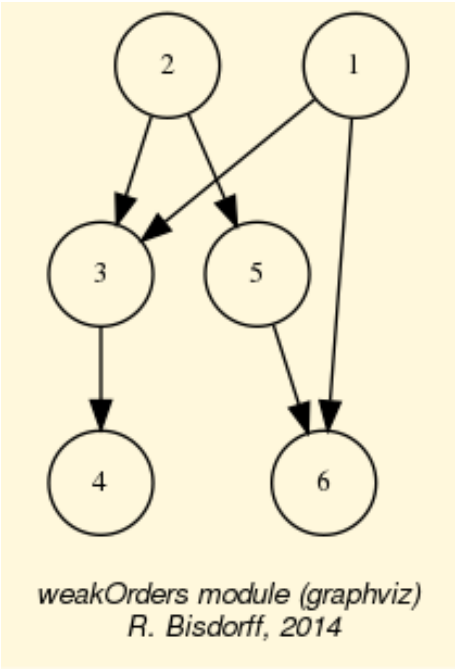


Fig. 1.7 The transitive orientation of the permutation graph.

```
>>> dgd = (-g).transitiveOrientation()
>>> print('Dual transitivity degree: %.3f' %\
...            dgd.computeTransitivityDegree() )
   Dual transitivity degree: 1.00
```

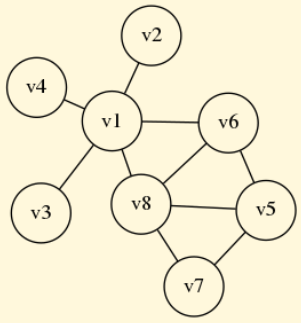
1.4 Recognizing permutation graphs

Now, a given graph g is a permutation graph if and only if both g **and** $-g$ are transitively orientable. This property gives a polynomial test procedure (in $O(n^3)$ due to the transitivity check) for recognizing permutation graphs.

Let us consider, for instance, the following random graph of order 8 generated with an edge probability of 40% and a random seed equal to 4335.

```
>>> from graphs import RandomGraph
>>> g = RandomGraph(order=8,\
...                 edgeProbability=0.4,seed=4335)
>>> g
  *------- Graph instance description ------*
   Instance class   : RandomGraph
   Instance name    : randomGraph
   Seed             : 4335
   Edge probability : 0.4
```

```
   Graph Order      : 8
   Graph Size       : 10
   Valuation domain : [-1.00; 1.00]
   Attributes       : ['name', 'order', 'vertices',
                       'valuationDomain', 'seed',
                       'edges', 'size', 'gamma',
                       'edgeProbability']
>>> g.isPerfectGraph()
  True
>>> g.exportGraphViz()
```



Graphs Python module (graphviz), R. Bisdorff, 2015

Fig. 1.8 Random graph of order 8 generated with edge probability 0.4.

If the random perfect graph instance g, shown in Fig. 1.8, is indeed a permutation graph, g and its dual $-g$ should be *transitively orientable*, i.e. comparability graphs (see [GOL-2004]). With the `isComparabilityGraph()` test, we may easily check this fact. This method proceeds indeed by trying to construct a transitive neighbourhood decomposition of a given graph instance and, if successful, stores the resulting edge orientations into a `edgeOrientations` attribute (see [GOL-2004] p.129-132).

```
>>> if g.isComparabilityGraph():
...     print(g.edgeOrientations)
 {('v1','v1'): 0, ('v1','v2'): 1, ('v2','v1'): -1, ('v1','v3'): 1,
  ('v3','v1'): -1, ('v1','v4'): 1, ('v4','v1'): -1, ('v1','v5'): 0,
  ('v5','v1'): 0, ('v1','v6'): 1, ('v6','v1'): -1, ('v1','v7'): 0,
  ('v7','v1'): 0, ('v1','v8'): 1, ('v8','v1'): -1, ('v2','v2'): 0,
  ('v2','v3'): 0, ('v3','v2'): 0, ('v2','v4'): 0, ('v4','v2'): 0,
  ('v2','v5'): 0, ('v5','v2'): 0, ('v2','v6'): 0, ('v6','v2'): 0,
  ('v2','v7'): 0, ('v7','v2'): 0, ('v2','v8'): 0, ('v8','v2'): 0,
  ('v3','v3'): 0, ('v3','v4'): 0, ('v4','v3'): 0, ('v3','v5'): 0,
  ('v5','v3'): 0, ('v3','v6'): 0, ('v6','v3'): 0, ('v3','v7'): 0,
  ('v7','v3'): 0, ('v3','v8'): 0, ('v8','v3'): 0, ('v4','v4'): 0,
  ('v4','v5'): 0, ('v5','v4'): 0, ('v4','v6'): 0, ('v6','v4'): 0,
  ('v4','v7'): 0, ('v7','v4'): 0, ('v4','v8'): 0, ('v8','v4'): 0,
  ('v5','v5'): 0, ('v5','v6'): 1, ('v6','v5'): -1, ('v5','v7'): 1,
  ('v7','v5'): -1, ('v5','v8'): 1, ('v8','v5'): -1, ('v6','v6'): 0,
  ('v6','v7'): 0, ('v7','v6'): 0, ('v6','v8'): 1, ('v8','v6'): -1,
  ('v7','v7'): 0, ('v7','v8'): 1, ('v8','v7'): -1, ('v8','v8'): 0}
```

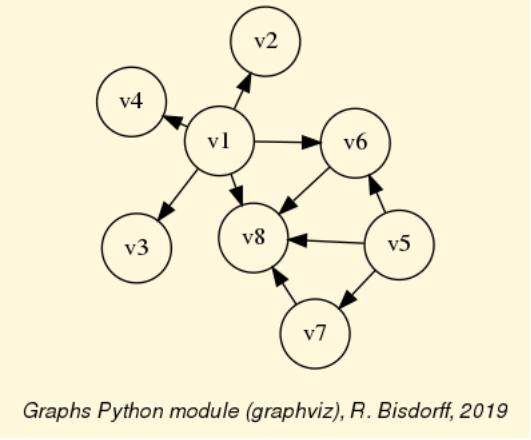


Fig. 1.9 Transitive neighbourhoods of the graph g.

The resulting orientation of the edges of g, shown in Fig. 1.9, is indeed transitive. The same procedure applied to the dual graph $gd = -g$ gives as well a transitive orientation to the edges of $-g$.

.. code-block:: pycon :linenos:

```
>>> gd = -g
>>> if gd.isComparabilityGraph():
...     print(gd.edgeOrientations)
  {('v1','v1'): 0, ('v1','v2'): 0, ('v2','v1'): 0, ('v1','v3'): 0,
   ('v3','v1'): 0, ('v1','v4'): 0, ('v4','v1'): 0, ('v1','v5'): 1,
   ('v5','v1'): -1, ('v1','v6'): 0, ('v6','v1'): 0, ('v1','v7'): 1,
   ('v7','v1'): -1, ('v1','v8'): 0, ('v8','v1'): 0, ('v2','v2'): 0,
   ('v2','v3'): -2, ('v3','v2'): 2, ('v2','v4'): -3, ('v4','v2'): 3,
   ('v2','v5'): 1, ('v5','v2'): -1, ('v2','v6'): 1, ('v6','v2'): -1,
   ('v2','v7'): 1, ('v7','v2'): -1, ('v2','v8'): 1, ('v8','v2'): -1,
   ('v3','v3'): 0, ('v3','v4'): -3, ('v4','v3'): 3, ('v3','v5'): 1,
   ('v5','v3'): -1, ('v3','v6'): 1, ('v6','v3'): -1, ('v3','v7'): 1,
   ('v7','v3'): -1, ('v3','v8'): 1, ('v8','v3'): -1, ('v4','v4'): 0,
   ('v4','v5'): 1, ('v5','v4'): -1, ('v4','v6'): 1, ('v6','v4'): -1,
   ('v4','v7'): 1, ('v7','v4'): -1, ('v4','v8'): 1, ('v8','v4'): -1,
   ('v5','v5'): 0, ('v5','v6'): 0, ('v6','v5'): 0, ('v5','v7'): 0,
   ('v7','v5'): 0, ('v5','v8'): 0, ('v8','v5'): 0, ('v6','v6'): 0,
   ('v6','v7'): 1, ('v7','v6'): -1, ('v6','v8'): 0, ('v8','v6'): 0,
   ('v7','v7'): 0, ('v7','v8'): 0, ('v8','v7'): 0, ('v8','v8'): 0}
```

Let us recheck these facts by explicitly constructing transitively oriented digraph instances with the `computeTransitivelyOrientedDigraph()` method [1].

```
>>> og = g.computeTransitivelyOrientedDigraph(\
...                       PartiallyDetermined = )
>>> print('Transitivity degree: %.3f' %\
...                        (og.transitivityDegree))
  Transitivity degree: 1.000
>>> ogd = (-g).computeTransitivelyOrientedDigraph(\
```

[1] The `PartiallyDetermined = True*` flag (see Lines 2 and 7 above) is required here in order to orient only the actual edges of the graphs. Relations between vertices not linked by an edge are put to the indeterminate characteristic value 0. This allows us to compute, later on, convenient disjunctive digraph fusions.

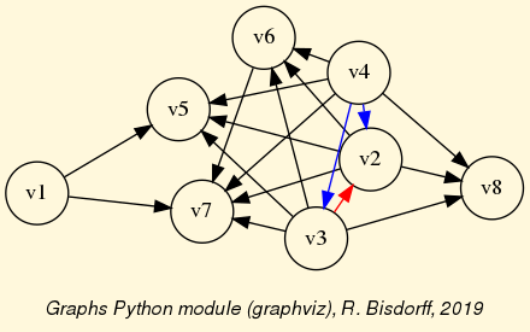


Fig. 1.10 Transitive neighbourhoods of the dual graph $-g$. It is worthwhile noticing that the orientation of g is achieved with a single neighbourhood decomposition, covering all the vertices. Whereas, the orientation of the dual graph $-g$ here needs a decomposition into three subsequent neighbourhoods marked in black, red and blue.

```
7  ...                        PartiallyDetermined = True)
8  >>> print('Transitivity degree: %.3f' %\
9  ...                        (ogd.transitivityDegree))
10   Transitivity degree: 1.000
```

As both graphs are indeed transitively orientable (see Lines 5 and 10 above), we may conclude that the given random graph g is actually a permutation graph instance. Yet, we still need to find now its corresponding permutation. We therefore implement a recipee given by Martin Golumbic [GOL-2004] p.159.

We will first *fuse* both *og* and *ogd* orientations above with an *epistemic disjunction* operated with the symmetric `o-max` operator Footnote[x].

```
1  >>> from digraphs import FusionDigraph
2  >>> f1 = FusionDigraph(og,ogd,operator='o-max')
3  >>> s1 = f1.computeCopelandRanking()
4  >>> print(s1)
5    ['v5','v7','v1','v6','v8','v4','v3','v2']
```

We obtain with the help of the `computeCopelandRanking()` method) the linear ordering ['v5','v7','v1','v6','v8','v4','v3','v2'] of the vertices (see Line 5 above).

We reverse now the orientation of the edges in *og* (see $-og$ in Line 1 below) in order to generate, again by disjunctive fusion, the *inversions* that are produced by the permutation we are looking for.

Computing again a ranking with the COPELAND rule, will show the correspondingly permuted list of vertices (see Line 4 below).

```
1  >>> f2 = FusionDigraph((-og),ogd,operator='o-max')
2  >>> s2 = f2.computeCopelandRanking()
3  >>> print(s2)
4    ['v8','v7','v6','v5','v4','v3','v2','v1']
```

Vertex 'v8' is put from position 5 to position 1, vertex 'v7' is put from position 2 to position 2, vertex 'v6' from position 4 to position 3, vertex 'v5' from position 1 to position 4, etc ... . We generate these position swaps for all vertices and obtain thus the required permutation (see Line 5 below).

```
1  >>> permutation = [0 for j in range(g.order)]
2  >>> for j in range(g.order):
```

```
...     permutation[s2.index(s1[j])] = j+1
>>> print(permutation)
  [5, 2, 4, 1, 6, 7, 8, 3]
```

It is worthwhile noticing by the way that transitive orientations of a given graph and its dual are usually *not unique* and, so may also be the resulting permutations. However, they all correspond to isomorphic graphs (see [GOL-2004]). In our case here, we observe two different permutations and their reverses::

```
s1: ['v1', 'v4', 'v3', 'v2', 'v5', 'v6', 'v7', 'v8']
s2: ['v4', 'v3', 'v2', 'v8', 'v6', 'v1', 'v7', 'v5']
(s1 -> s2): [2, 3, 4, 8, 6, 1, 7, 5]
(s2 -> s1): [6, 1, 2, 3, 8, 5, 7, 4]
```

And

```
s3: ['v5', 'v7', 'v1', 'v6', 'v8', 'v4', 'v3', 'v2']
s4: ['v8', 'v7', 'v6', 'v5', 'v4', 'v3', 'v2', 'v1']
(s3 -> s4): [5, 2, 4, 1, 6, 7, 8, 3]
(s4 -> s3) = [4, 2, 8, 3, 1, 5, 6, 7]
```

The `computePermutation()` method does directly operate all these steps: - computing transitive orientations, - ranking their epistemic fusion and, - delivering a corresponding permutation.

```
>>> g.computePermutation(Comments=True)
  ['v1', 'v2', 'v3', 'v4', 'v5', 'v6', 'v7', 'v8']
  ['v2', 'v3', 'v4', 'v8', 'v6', 'v1', 'v7', 'v5']
  [2, 3, 4, 8, 6, 1, 7, 5]
```

We may finally check that, for instance, the two permutations [2, 3, 4, 8, 6, 1, 7, 5] and [4, 2, 8, 3, 1, 5, 6, 7] observed above, will generate corresponding *isomorphic* permutation graphs.

```
>>> gtesta = PermutationGraph(\
  ...       permutation=[2, 3, 4, 8, 6, 1, 7, 5])
  >>> gtestb = PermutationGraph(\
  ...       permutation=[4, 2, 8, 3, 1, 5, 6, 7])
>>> gtesta.exportGraphViz('gtesta')
>>> gtestb.exportGraphViz('gtestb')
```

.. Figure:: isomorphicPerms.png :alt: Isomorphic permutation graphs :name: isomorphicPermGraphs :width: 700 px :align: center

Isomorphic permutation graphs

And, we indeed recover in Fig. 1.11 indeed two isomorphic copies of the original random graph (compare with Fig. 1.8).

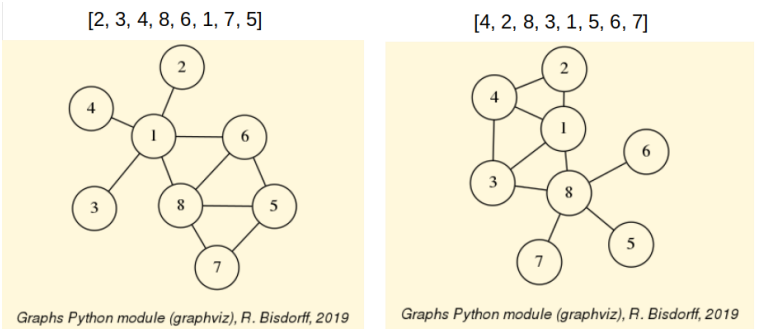


Fig. 1.11 Isomorphic permutation graphs.