On Decompositions, Generation Methods and related concepts in the theory of Matching Covered Graphs

Bachelor of Technology Project Thesis Report
submitted by

JANMENJAYA PANDA
ME20B087

in partial fulfillment of the requirements for the award of the degree of

BACHELOR OF TECHNOLOGY
in
MECHANICAL ENGINEERING

Department of Mechanical Engineering

Indian Institute of Technology Madras

Declaration

I certify that the ideas, designs, experimental work, results, analyses and conclusions set out in this dissertation are entirely my own effort, except where otherwise indicated and acknowledged. I further certify that the work is original and has not been previously submitted for assessment in any other course or institution except where specifically stated.

Janmenjaya Panda

Name:	**Janmenjaya Panda**	Place:	Chennai
Roll No.:	ME20B087	Date:	17/05/2024

Certificate

This certificate attests that the project report titled "**On Decompositions, Generation Methods and related concepts in the theory of Matching Covered Graphs**", submitted by **Janmenjaya Panda** (ME20B087) to the Indian Institute of Technology, Madras, in partial fulfilment of the requirements for the Bachelor of Technology degree in Mechanical Engineering, constitutes bonafide work conducted by him in the Department of Computer Science and Engineering at IIT Madras. The contents of this report, in its entirety or in segments, have not been presented to any other institute or university for the conferral of any degree or diploma.

Nishad Kothari
Project Guide & Assistant Professor
Department of Computer Science and Engineering
Indian Institute of Technology Madras
Chennai, 600036

Dr. Chandramouli Padmanabhan
Professor & Head
Department of Mechanical Engineering
Indian Institute of Technology Madras
Chennai, 600036

Place: Chennai
Date: 17/05/2024

Acknowledgements

I am absolutely grateful to my guide Prof. Nishad Kothari. Throughout last one and half years, his guidance has been invaluable. He introduced me the fascinating realm of matching theory, igniting within me a desire to explore new avenues in this field. I am truly appreciative for the countless hours he has listened to my immature explanations, mentoring me and offering his ingenious solutions whenever I got stuck. During moments of despair, he supported me unwaveringly, bolstering my confidence in my abilities. He taught me the joy of mathematics, the fun of problem solving and have contributed significantly to my personal growth. I owe him an immeasurable debt of gratitude and consider myself incredibly fortunate to have had the privilege of being mentored by him.

I am deeply thankful to Amit Kumar Mallik for his precious time and expert suggestions and ideas that guided me for this project. He has been a great companion throughout this journey.

Finally, I would like to thank all my batchmates and friends who have supported me throughout my undergraduate directly or indirectly.

Limitations of Use

The Department of Mechanical Engineering at the Indian Institute of Technology Madras and the staff of IIT Madras do not accept any responsibility for the truth, accuracy, or completeness of the material contained within or associated with this dissertation.

Persons using any or all parts of this material do so at their own risk and not at the risk of the Department of Mechanical Engineering at the Indian Institute of Technology Madras or the staff of IIT Madras.

This document, along with the associated hardware, software, drawings, and other materials set out in the associated appendices, should not be used for any other purpose. If they are used in such a manner, it is entirely at the risk of the user.

Abstract

Matchings and perfect matchings have received considerable attention in graph theory as well as in other related domains (such as, but not limited to, algorithms and optimization). There still remain many open problems — such as Barnette's conjecture, Berge-Fulkerson conjecture, and so on — due to which it continues to remain an active area of research. For problems pertaining to perfect matchings, it is well-known that it suffices to solve them for matching covered graphs (that is, those connected graphs wherein each edge belongs to some perfect matching). The theory of matching covered graphs, despite its relatively recent emergence, presents an exciting landscape filled with captivating discoveries, elegant proofs, and unexpected applications. In the following paragraph, we briefly summarize some of the key developments in this field without going into the mathematical details.

Kotzig [Ein beitrag zur theorie der endlichen graphen iiiiii. Mat. Fyz. Casopis, 9:7391, 136159, and 10 (1960) 205215], in 1959, introduced the notion of canonical partition of a matching covered graph that uniquely partitions its vertex set into its maximal barriers. In 1987, László Lovász [Matching structure and the matching lattice. Journal of Combinatorial Theory, Series B, 43(2):187222] established the uniqueness of the tight cut decomposition procedure; as per this, every matching covered graph may be uniquely decomposed into a list of special matching covered graphs called *"bricks"* (nonbipartite) and *"braces"* (bipartite). The key contribution of this landmark paper was to solve the Matching Lattice Problem. Lovász and Plummer, in 1975, introduced the well-known ear decomposition of matching covered graphs; see *"Matching Theory"* by L. Lovász and M. D. Plummer. This notion of ear decomposition was further refined to that of an optimal ear decomposition by Carvalho, Lucchesi and Murty [Optimal ear decompositions of matching covered graphs and bases for the matching lattice. Journal of Combinatorial Theory, Series B, 76(1):114, 2002]. Furthermore, in their seminal paper, the same authors [Ear decompositions of matching covered

graphs. Combinatorica, 19:151174, 1999] introduced the dependency relationship in matching covered graphs that is closely tied with the ear decomposition theory. In 2001, McCuaig [Brace generation. Journal of Graph Theory, 38:124169] established a generation method for all braces; analogously, Norine and Thomas [Generating bricks. Journal of Combinatorial Theory, Series B, 97:769817], in 2007, established a generation method for all bricks. Both of these generation procedures may be viewed as a synthesis of the ear decomposition and tight cut decomposition theories, and have found applications in solving some of the major problems in matching theory — such as Pólya's Permanent Problem [Pólyas permanent problem. The Electronic Journal of Combinatorics, 11(1):R79, 2004]. All of these results — pertaining to decompositions, generation methods and related concepts — have played indispensable roles in the advancement of matching theory, and continue to do so.

It is worth noting that all of the notions discussed above are computable in poly-time. Despite this, there are no publicly available implementations. It is for this reason that researchers in this area are at a loss, and are required to implement parts of this theory by themselves. Currently, in SageMath, a few existing matching-theoretic algorithms for general graphs have been put within the module "Undirected graphs" — either under the submodule "Algorithmically hard stuff" (for instance: `matching_polynomial()`) or under "Leftovers" (for instance: `has_perfect_matching()`, `matching()`, `is_factor_critical()` and `perfect_matchings()`), whereas that concerning the bipartite graphs have been put within the module "Bipartite Graphs" (for instance: `matching()`, `matching_polynomial()` and `perfect_matchings()`). Ergo, we propose to implement efficient algorithms pertaining to the results and concepts discussed in the above paragraph in SageMath, and to make all of these available freely to students, educators as well as researchers all across the world. This proposal has been inspired by the book of Lucchesi and Murty — *"Perfect Matchings: a theory of matching covered graphs"*.

Contents

Abbreviations

Bip.	Bipartite
ILP	Integer Linear Program
LP	Linear Program
MCG	Matching Covered Graph
PM	Perfect Matching
rem.	removable

Chapter 1

Introduction

All the graphs considered in this proposal are undirected and loopless. But, they might contain multiple edges. For graph theoretical notation and terminology, the main resources that are essentially followed, are —- Graph Theory (2008, [1]) by Bondy and Murty. This proposal assumes that the reader has the basic knowledge in graph theory. The reader is requested to refer to the equivalent papers in case they require an in depth overview of the concerning concepts.

Chapter 2

Existing Methods

This section identifies and explicitly lists out all the existing functions (as of 17/01/2024) on SageMath — pertaining to the theory of matching covered graphs.

2.1 Undirected graphs

We have identified the following precisely five algorithms that are listed within the module "Undirected graphs", which implements functions and operations involving undirected graphs.

2.1.1 matching_polynomial()

The method `matching_polynomial()` listed under "Algorithmically hard stuff" computes the matching polynomial of the graph G.

```
matching_polynomial(G, complement=True, name=None)
```

Computes the matching polynomial of the graph G. If p(G,k) denotes the number of k-matchings (matchings with k edges) in G, then the matching polynomial is defined as [12]:

$$\mu(x) = \sum_{k \geqslant 0} (-1)^k p(G,k) x^{n-2k}.$$

INPUT:

- `complement` – (default: `True`) whether to use Godsils duality theorem to compute the matching polynomial from that of the graphs complement (see ALGORITHM).
- `name` – optional string for the variable name in the polynomial

Note

The `complement` option uses matching polynomials of complete graphs, which are cached. So if you are crazy enough to try computing the matching polynomial on a graph with millions of vertices, you might not want to use this option, since it will end up caching millions of polynomials of degree in the millions.

OUTPUT:

- When `value_only=False` (default), this method returns an EdgesView containing the edges of a maximum matching of G.
- When `value_only=True`, this method returns the sum of the weights (default: 1) of the edges of a maximum matching of G. The type of the output may vary according to the type of the edge labels and the algorithm used.

ALGORITHM:

The algorithm used is a recursive one, based on the following observation [12]:

- If e is an edge of G, G' is the result of deleting the edge e, and G'' is the result of deleting each vertex in e, then the matching polynomial of G is equal to that of G' minus that of G''. (the algorithm actually computes the signless matching polynomial, for which the recursion is the same when one replaces the subtraction by an addition. It is then converted into the matching polynomial and returned)

Depending on the value of `complement`, Godsils duality theorem [12] can also be used to compute $\mu(x)$:

$$\mu(\overline{G}, x) = \sum_{k \geqslant 0} p(G, k) \mu(K_{n-2k}, x)$$

Where $\overline{G}$ is the complement of G, and K_n the complete graph on n vertices.

EXAMPLES:

```
sage: g = graphs.PetersenGraph()
sage: g.matching_polynomial()
x^10 - 15*x^8 + 75*x^6 - 145*x^4 + 90*x^2 - 6
sage: g.matching_polynomial(complement=False)
x^10 - 15*x^8 + 75*x^6 - 145*x^4 + 90*x^2 - 6
sage: g.matching_polynomial(name='tom')
tom^10 - 15*tom^8 + 75*tom^6 - 145*tom^4 + 90*tom^2 - 6
sage: g = Graph()
sage: L = [graphs.RandomGNP(8, .3) for i in range(1, 6)]
```

```
sage: prod([h.matching_polynomial() for h in L]) == sum(L, g).matching_polynomial()
↪  # long time (up to 10s on sage.math, 2011)
True
```

2.1.2 has_perfect_matching()

The method `has_perfect_matching()` listed under "Leftovers" returns whether the graph has a perfect matching.

```
has_perfect_matching(algorithm='Edmonds', solver=None, verbose=0,
↪  integrality_tolerance)
```

Returns whether this graph has a perfect matching.

INPUT:

- `algorithm` – string (default: '`Edmonds`')
  - '`Edmonds`' uses Edmonds algorithm as implemented in NetworkX to find a matching of maximal cardinality, then check whether this cardinality is half the number of vertices of the graph.
  - '`LP_matching`' uses a Linear Program to find a matching of maximal cardinality, then check whether this cardinality is half the number of vertices of the graph.
  - '`LP`' uses a Linear Program formulation of the perfect matching problem: put a binary variable $b[e]$ on each edge e, and for each vertex v, require that the sum of the values of the edges incident to v is 1.
- `solver` – string (default: `None`); specify a Mixed Integer Linear Programming (MILP) solver to be used. If set to `None`, the default one is used. For more information on MILP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity: set to `0` by default, which means quiet (only useful when `algorithm == 'LP_matching'` or `algorithm == 'LP'`)
- `integrality_tolerance` – float; parameter for use with MILP solvers over an inexact base ring; see `MixedIntegerLinearProgram.get_values()`.

OUTPUT:

A boolean.

EXAMPLES:

```
sage: graphs.PetersenGraph().has_perfect_matching()  # needs networkx
True
sage: graphs.WheelGraph(6).has_perfect_matching()  # needs networkx
True
sage: graphs.WheelGraph(5).has_perfect_matching()  # needs networkx
False
sage: graphs.PetersenGraph().has_perfect_matching(algorithm="LP_matching")
 # needs sage.numerical.mip
True
sage: graphs.WheelGraph(6).has_perfect_matching(algorithm="LP_matching")  # needs
↪  sage.numerical.mip
True
sage: graphs.WheelGraph(5).has_perfect_matching(algorithm="LP_matching")
False
sage: graphs.PetersenGraph().has_perfect_matching(algorithm="LP_matching")
 # needs sage.numerical.mip
True
sage: graphs.WheelGraph(6).has_perfect_matching(algorithm="LP_matching")  # needs
↪  sage.numerical.mip
True
sage: graphs.WheelGraph(5).has_perfect_matching(algorithm="LP_matching")
False
```

2.1.3 is_factor_critical()

The method `is_factor_critical()` listed under "Leftovers" checks whether this graph is factor-critical.

```
is_factor_critical(matching=None, algorithm='Edmonds', solver=None, verbose=0,
↪  integrality_tolerance)
```

Checks whether this graph is factor-critical.

A graph of order n is factor-critical if every subgraph of $n-1$ vertices have a perfect matching, hence n must be odd. See Wikipedia article: Factor-critical graph for more details.

This method implements the algorithm proposed in [14] and we assume that a graph of order one is factor-critical. The time complexity of the algorithm is linear if a near perfect matching is given as input (i.e., a matching such that all vertices but one are incident to an edge of the matching). Otherwise, the time complexity is dominated by the time needed to compute a maximum matching of the graph.

INPUT:

- `matching` – (default: `None`); a near perfect matching of the graph, that is a matching such that all vertices of the graph but one are incident to an edge of the matching. It can be given using any valid input format of Graph.

If set to None, a matching is computed using the other parameters.

- algorithm – string (default: 'Edmonds'); the algorithm to use to compute a maximum matching of the graph among
  - 'Edmonds' selects Edmonds algorithm as implemented in NetworkX,
  - 'LP_matching' uses a Linear Program to find a matching of maximal cardinality, then check whether this cardinality is half the number of vertices of the graph.
  - 'LP' uses a Linear Program formulation of the matching problem.
- solver – string (default: None); specify a Mixed Integer Linear Programming (MILP) solver to be used. If set to None, the default one is used. For more information on MILP solvers and which default solver is used, see the method solve of the class MixedIntegerLinearProgram.
- verbose – integer (default: 0); sets the level of verbosity: set to 0 by default, which means quiet (only useful when algorithm == 'LP')
- integrality_tolerance – float; parameter for use with MILP solvers over an inexact base ring; see MixedIntegerLinearProgram.get_values().

EXAMPLES:

Odd length cycles and odd cliques of order at least 3 are factor-critical graphs:

```
sage: [graphs.CycleGraph(2*i + 1).is_factor_critical() for i in range(5)]
 # needs networkx
[True, True, True, True, True]
sage: [graphs.CompleteGraph(2*i + 1).is_factor_critical() for i in range(5)]   #
↪ needs networkx
[True, True, True, True, True]
```

More generally, every Hamiltonian graph with an odd number of vertices is factor-critical:

```
sage: G = graphs.RandomGNP(15, .2)
sage: G.add_path([0..14])
sage; G.add_edge(14, 0)
sage: G.is_hamiltonian()
True
sage: G.is_factor_critical()  # needs networkx
True
```

Friendship graphs are non-Hamiltonian factor-critical graphs:

```
sage: [graphs.FriendshipGraph(i).is_factor_critical() for i in range(1, 5)]   #
↪ needs networkx
[True, True, True, True]
```

Bipartite graphs are not factor-critical:

```
sage: G = graphs.RandomBipartite(randint(1, 10), randint(1, 10), .5)  # needs numpy
sage: G.is_factor_critical()  # needs numpy
False
```

Graphs with even order are not factor critical:

```
sage: G = graphs.RandomGNP(10, .5)
sage: G.is_factor_critical()
False
```

One can specify a matching:

```
sage: F = graphs.FriendshipGraph(4)
sage: M = F.matching()  # needs networkx
sage: F.is_factor_critical(matching=M)  # needs networkx
True
sage: F.is_factor_critical(matching=Graph(M))  # needs networkx
True
```

2.1.4 matching()

The method `matching()` listed under "Leftovers" returns a maximum weighted matching of the graph represented by the list of its edges.

```
matching(value_only=False, algorithm='Edmonds', use_edge_labels=False, solver=None,
↪ verbose=0, integrality_tolerance)
```

Returns a maximum weighted matching of the graph represented by the list of its edges.

For more information, see the Wikipedia article: Matching(graph theory).

Given a graph G such that each edge e has a weight w_e, a maximum matching is a subset S of the edges of G of maximum weight such that no two edges of S are incident with each other.

As an optimization problem, it can be expressed as:

$$\begin{array}{llll} \text{Maximize} & \sum_{e \in G.edges()} & w_e b_e & \\ \text{such that} & \sum_{(u,v) \in G.edges()} & b_{(u,v)} \leqslant 1 & \forall\, v_i \in V(G) \\ & & b_x \in \{0,1\} & \forall\, x \in E(G) \end{array}$$

INPUT:

- `value_only` – boolean (default: `False`); when set to `True`, only the cardinal (or the weight) of the matching is returned.
- `algorithm` – string (default: `'Edmonds'`)
  - `'Edmonds'` selects Edmonds algorithm as implemented in NetworkX,
  - `'LP'` uses a Linear Program formulation of the matching problem.
- `use_edge_labels` – boolean (default: `False`)
  - when set to `True`, computes a weighted matching where each edge is weighted by its label (if an edge has no label, 1 is assumed),
  - when set to `False`, each edge has weight 1.
- `solver` – string (default: `None`); specify a Mixed Integer Linear Programming (MILP) solver to be used. If set to `None`, the default one is used. For more information on MILP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity: set to `0` by default, which means quiet (only useful when `algorithm == 'LP'`)
- `integrality_tolerance` – float; parameter for use with MILP solvers over an inexact base ring; see `MixedIntegerLinearProgram.get_values()`.

OUTPUT:

- When `value_only=False` (default), this method returns an EdgesView containing the edges of a maximum matching of G.
- When `value_only=True`, this method returns the sum of the weights (default: 1) of the edges of a maximum matching of G. The type of the output may vary according to the type of the edge labels and the algorithm used.

ALGORITHM:

The problem is solved using Edmonds algorithm implemented in NetworkX, or using Linear Programming depending on the value of `algorithm`.

EXAMPLES:

Maximum matching in a Pappus Graph:

```
sage: g = graphs.PappusGraph()
sage: g.matching(value_only=True)  # needs sage.networkx
9
```

Same test with the Linear Program formulation:

```
sage: g = graphs.PappusGraph()
g.matching(algorithm="LP", value_only=True)  # needs sage.numerical.mip
9
```

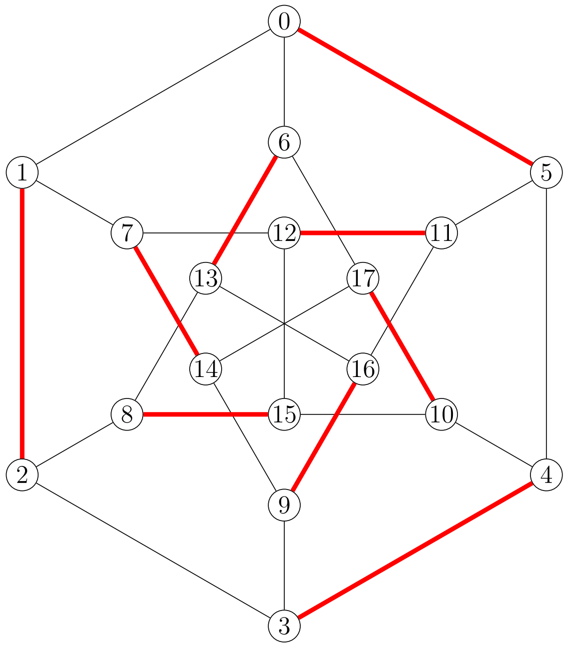


Figure 2.1: Pappus Graph and its maximum (perfect) matching — shown in bold red

2.1.5 perfect_matchings()

The method `perfect_matchings()` listed under "Leftovers" returns an iterator over all perfect matchings of the graph.

```
perfect_matchings(labels=False)
```

Returns an iterator over all perfect matchings of the graph.

INPUT:

- `labels` – boolean (default: `False`); when `True`, the edges in each perfect matching are triples (containing the label as the third element), otherwise the edges are pairs.

ALGORITHM:

Choose a vertex v, then recurse through all edges incident to v, removing one edge at a time whenever an edge is added to a matching.

EXAMPLES:

```
sage: G=graphs.GridGraph([2,3])
sage: for m in G.perfect_matchings():
....:     print(sorted(m))
[((0, 0), (0, 1)), ((0, 2), (1, 2)), ((1, 0), (1, 1))]
[((0, 0), (1, 0)), ((0, 1), (0, 2)), ((1, 1), (1, 2))]
[((0, 0), (1, 0)), ((0, 1), (1, 1)), ((0, 2), (1, 2))]

sage: G = graphs.CompleteGraph(4)
sage: for m in G.perfect_matchings(labels=True):
....:     print(sorted(m))
[(0, 1, None), (2, 3, None)]
[(0, 2, None), (1, 3, None)]
[(0, 3, None), (1, 2, None)]

sage: G = Graph([[1,-1,'a'], [2,-2, 'b'], [1,-2,'x'], [2,-1,'y']])
sage: sorted(sorted(m) for m in G.perfect_matchings(labels=True))
[[(-2, 1, 'x'), (-1, 2, 'y')], [(-2, 2, 'b'), (-1, 1, 'a')]]

sage: G = graphs.CompleteGraph(8)
sage: mpc = G.matching_polynomial().coefficients(sparse=False)[0]  # needs
↪ sage.libs.flint
sage: len(list(G.perfect_matchings())) == mpc  # needs sage.libs.flint
True

sage: G = graphs.PetersenGraph().copy(immutable=True)
sage: [sorted(m) for m in G.perfect_matchings()]
[[(0, 1), (2, 3), (4, 9), (5, 7), (6, 8)],
 [(0, 1), (2, 7), (3, 4), (5, 8), (6, 9)],
 [(0, 4), (1, 2), (3, 8), (5, 7), (6, 9)],
 [(0, 4), (1, 6), (2, 3), (5, 8), (7, 9)],
 [(0, 5), (1, 2), (3, 4), (6, 8), (7, 9)],
 [(0, 5), (1, 6), (2, 7), (3, 8), (4, 9)]]

sage: list(Graph().perfect_matchings())
[[]]
```

```
sage: G = graphs.CompleteGraph(5)
sage: list(G.perfect_matchings())
[]
```

2.2 Bipartite graphs

This subsection explicitly mentions all the existing funtions listed within the module "Bipartite graphs" (as of Monday 27$^{\text{th}}$ May, 2024) concerning matching theory.

2.2.1 matching()

The method `matching()` returns a maximum matching of the graph represented by the list of its edges.

```
matching(value_only=False, algorithm='Hopcroft-Karp', use_edge_labels=False,
 ↪  solver=None, verbose=0, integrality_tolerance)
```

Returns a maximum matching of the graph represented by the list of its edges.

Given a graph G such that each edge e has a weight w_e, a maximum matching is a subset S of the edges of G of maximum weight such that no two edges of S are incident with each other.

INPUT:

- `value_only` – boolean (default: `False`); when set to `True`, only the cardinal (or the weight) of the matching is returned.
- `algorithm` – string (default: `'Hopcroft-Karp'` if `use_edge_lables==false`, otherwise `'Edmonds'`); algorithm to use among:
  - `'Hopcroft-karp'` selects the default bipartite graph algorithm as implemented in NetworkX,
  - `'Eppstein'` selects Eppsteins algorithm as implemented in NetworkX,
  - `'Edmonds'` selects Edmonds algorithm as implemented in NetworkX,
  - `'LP'` uses a Linear Program formulation of the matching problem.
- `use_edge_labels` – boolean (default: `False`)
  - when set to `True`, computes a weighted matching where each edge is weighted by its label (if an edge has no label, 1 is assumed); only if `algorithm` is `'Edmonds`, `'LP'`
  - when set to `False`, each edge has weight 1.

- `solver` – string (default: `None`); specify a Mixed Integer Linear Programming (MILP) solver to be used. If set to `None`, the default one is used. For more information on MILP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity: set to `0` by default, which means quiet (only useful when `algorithm == 'LP'`)
- `integrality_tolerance` – float; parameter for use with MILP solvers over an inexact base ring; see `MixedIntegerLinearProgram.get_values()`.

EXAMPLES:

Maximum matching in a cycle graph:

```
sage: G = BipartiteGraph(graphs.CycleGraph(10))
sage: G.matching()  # needs networkx
[(0, 1, None), (2, 3, None), (4, 5, None), (6, 7, None), (8, 9, None)]
```

The size of a maximum matching in a complete bipartite graph using Eppstein:

```
sage: G = BipartiteGraph(graphs.CompleteBipartiteGraph(4,5))
sage: G.matching(algorithm="Eppstein", value_only=True)  # needs networkx
4
```

2.2.2 matching_polynomial()

The method `matching_polynomial()` computes the matching polynomial.

`matching_polynomial(algorithm='Godsil', name=None)`

Computes the matching polynomial.

The matching polynomial is defined as in [12], where $p(G,k)$ denotes the number of k-matchings (matchings with k edges) in G:

$$\mu(x) = \sum_{k \geqslant 0} (-1)^k p(G,k) x^{n-2k}.$$

INPUT:

- `algorithm` – string (default: `'Godsil'`); either `'Godsil'` or `'rook'`; 'rook' is usually faster for larger graphs.
- `name` – string (default: `None`); name of the variable in the polynomial, set to x when name is `None`.

EXAMPLES:

```
sage: BipartiteGraph(graphs.CubeGraph(3)).matching_polynomial()  # needs
↪ sage.libs.flint
x^8 - 12*x^6 + 42*x^4 - 44*x^2 + 9
```

```
sage: x = polygen(QQ)
sage: g = BipartiteGraph(graphs.CompleteBipartiteGraph(16, 16))
sage: bool(factorial(16) * laguerre(16, x^2)  # needs sage.symbolic
....:      == g.matching_polynomial(algorithm='rook'))
True
```

Compute the matching polynomial of a line with 60 vertices:

```
sage: from sage.functions.orthogonal_polys import chebyshev_U  # needs
↪ sage.symbolic
sage: g = next(graphs.trees(60))
sage: (chebyshev_U(60, x/2)  # needs sage.symbolic
....:   == BipartiteGraph(g).matching_polynomial(algorithm='rook'))
True
```

The matching polynomial of a tree is equal to its characteristic polynomial:

```
sage: g = graphs.RandomTree(20)
sage: p = g.characteristic_polynomial()  # needs sage.modules
sage: p == BipartiteGraph(g).matching_polynomial(algorithm='rook')  # needs
↪ sage.modules
True
```

2.2.3 perfect_matchings()

The method `perfect_matchings()` returns an iterator over all perfect matchings of the bipartite graph.

```
perfect_matchings(labels=False)
```

Returns an iterator over all perfect matchings of the bipartite graph.

INPUT:

- `labels` – boolean (default: `False`); when `True`, the edges in each perfect matching are triples (containing the label as the third element), otherwise the edges are pairs.

ALGORITHM:

Choose a vertex v, then recurse through all edges incident to v, removing one edge at a time whenever an edge is added to a matching.

EXAMPLES:

```
sage: B = BipartiteGraph({0: [5, 7], 1: [4, 6, 7], 2: [4, 5, 8], 3: [4, 5, 6], 6:
 ↪ [9], 8: [9]})
sage: len(list(B.perfect_matchings()))
6
sage: G = Graph(B.edges(sort=False))
sage: len(list(G.perfect_matchings()))
6
```

The algorithm ensures that for any edge of a perfect matching, the first vertex is on the left set of vertices and the second vertex in the right set:

```
sage: B = BipartiteGraph({0: [5, 7], 1: [4, 6, 7], 2: [4, 5, 8], 3: [4, 5, 6], 6:
 ↪ [9], 8: [9]})
sage: m = next(B.perfect_matchings(labels=False))
sage: B.left
{0, 1, 2, 3, 9}
sage: B.right
{4, 5, 6, 7, 8}
sage: sorted(m)
[(0, 7), (1, 4), (2, 5), (3, 6), (9, 8)]
sage: all((u in B.left and v in B.right) for u, v in m)
True
```

Multiple edges are taken into account:

```
sage: B = BipartiteGraph({0: [5, 7], 1: [4, 6, 7], 2: [4, 5, 8], 3: [4, 5, 6], 6:
 ↪ [9], 8: [9]})
sage: B.allow_multiple_edges(True)
sage: B.add_edge(0, 7)
sage: len(list(B.perfect_matchings()))
10
```

Empty graph:

```
sage: list(BipartiteGraph().perfect_matchings())
[[]]
```

Bipartite graph without perfect matching:

```
sage: B = BipartiteGraph(graphs.CompleteBipartiteGraph(3, 4))
sage: list(B.perfect_matchings())
[]
```

Check that the number of perfect matchings of a complete bipartite graph is consistent with the matching polynomial:

```
sage: B = BipartiteGraph(graphs.CompleteBipartiteGraph(4, 4))
sage: len(list(B.perfect_matchings()))
24
sage: B.matching_polynomial(algorithm='rook')(0)  # needs sage.modules
24
```

2.3 Common graphs

There are several graphs/ families of graphs, that play significant role in the theory of matching covered graphs. This sections lists out some of the existing such crucial graphs (for instance Petersen graph, Hexahedral graph and so on), and graph families (for instance Ladder graph, Circular ladder graph, aka Prism graph, Wheel graph and so on).

2.3.1 PetersenGraph()

This subsection visits `PetersenGraph()` implemented in SageMath.

`static PetersenGraph()`

Returns the Petersen Graph.

The Petersen Graph is a named graph that consists of 10 vertices and 15 edges, usually drawn as a five-point star embedded in a pentagon.

The Petersen Graph is a common counterexample. For example, it is not Hamiltonian.

PLOTTING: See the plotting section for the generalized Petersen graphs.

EXAMPLES: We compare below the Petersen graph with the default spring-layout versus a planned position dictionary of (x, y) tuples:

```
sage: petersen_spring = Graph({0:[1,4,5], 1:[0,2,6], 2:[1,3,7],
                              3:[2,4,8], 4:[0,3,9], 5:[0,7,8],
                              6:[1,8,9], 7:[2,5,9], 8:[3,5,6],
                              9:[4,6,7]})
sage: petersen_spring.show()  # long time  # needs sage.plot
sage: petersen_database = graphs.PetersenGraph()
```

```
sage: petersen_database.show()  # long time  # needs sage.plot
```

2.3.2 HexahedralGraph()

This static method visits `HexahedralGraph()` implemented in SageMath.

> `static HexahedralGraph()`
>
> Returns the Hexahedral graph (with 8 nodes).
>
> A regular hexahedron is a 6-sided cube. The hexahedral graph corresponds to the connectivity of the vertices of the hexahedron. This graph is equivalent to a 3-cube.

PLOTTING: The Hexahedral graph should be viewed in 3 dimensions. We choose to use a planar embedding of the graph. We hope to add rotatable, 3-dimensional viewing in the future. In such a case, an argument will be added to select the desired layout.

EXAMPLES:

Construct and show a Hexahedral graph:

```
sage: g = graphs.HexahedralGraph()
sage: g.show()  # long time  # needs sage.plot
```

Create several hexahedral graphs in a Sage graphics array. They will be drawn differently due to the use of the spring-layout algorithm:

```
sage: # needs sage.plot
sage: g = []
sage: j = []
sage: for i in range(9):
....:     k = graphs.HexahedralGraph()
....:     g.append(k)
sage: for i in range(3):
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)
sage: G.show()  # long time
```

2.3.3 LadderGraph()

The static method `LadderGraph(n)` returns a ladder graph with $2 \times n$ nodes.

```
static LadderGraph(n)
```

Returns the Ladder graph with $2 \times n$ nodes.

A ladder graph is a basic structure that is typically displayed as a ladder, i.e.: two parallel path graphs connected at each corresponding node pair.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, each ladder graph will be displayed horizontally, with the first n nodes displayed left to right on the top horizontal line.

EXAMPLES:

Construct and show a ladder graph with 14 nodes:

```
sage: g = graphs.LadderGraph(7)
sage: g.show()  # long time  # needs sage.plot
```

Create several ladder graphs in a Sage graphics array:

```
sage: # needs sage.plot
sage: g = []
sage: j = []
sage: for i in range(9):
....:     k = graphs.LadderGraph(i+2)
....:     g.append(k)
sage: for i in range(3):
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)
sage: G.show()  # long time
```

2.3.4 CicularLadderGraph()

The static method `CircularLadderGraph(n)` returns a circular ladder graph with $2 \times n$ nodes.

```
static CircularLadderGraph(n)
```

Returns the Circular ladder graph with $2 \times n$ nodes.

A Circular ladder graph is a ladder graph that is connected at the ends, i.e.: a ladder bent around so that top meets bottom. Thus it can be described as two parallel cycle graphs connected at each corresponding node pair.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the circular ladder graph is displayed as an inner and outer cycle pair, with the first nodes drawn on the inner circle. The first (0) node is drawn at the top of the inner-circle, moving clockwise after that. The outer circle is drawn with the (n+1)-th node at the top, then counter-clockwise as well. When $n == 2$, we rotate the outer circle by an angle of $\frac{\pi}{8}$ to ensure that all edges are visible (otherwise the 4 vertices of the graph would be placed on a single line).

EXAMPLES:

Construct and show a circular ladder graph with 26 nodes:

```
sage: g = graphs.CircularLadderGraph(13)
sage: g.show()  # long time  # needs sage.plot
```

Create several circular ladder graphs in a Sage graphics array:

```
sage: g = []
sage: j = []
sage: for i in range(9):
....:     k = graphs.CircularLadderGraph(i+3)
....:     g.append(k)
sage: for i in range(3):  # needs sage.plot
....:     n = []
....:     for m in range(3):
....:         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:     j.append(n)
sage: G = graphics_array(j)  # needs sage.plot
sage: G.show()  # long time  # needs sage.plot
```

2.3.5 WheelGraph()

The static method `WheelGraph(n)` returns a Wheel graph with n nodes.

`static LadderGraph(n)`

Returns the Wheel graph with n nodes.

A Wheel graph is a basic structure where one node is connected to all other nodes and those (outer) nodes are connected cyclically.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, each wheel graph will be displayed with the first (0) node in the center, the second node at the top, and the rest following in a counterclockwise manner.

With the wheel graph, we see that it doesnt take a very large n at all for the spring-layout to give a counter-intuitive display. (See Graphics Array examples below).

EXAMPLES:

We view many wheel graphs with a Sage Graphics Array, first with this constructor (i.e., the position dictionary filled):

```
sage: # needs sage.plot
sage: g = []
sage: j = []
sage: for i in range(9):
....:  k = graphs.WheelGraph(i+3)
....:  g.append(k)
...
sage: for i in range(3):
....:  n = []
....:  for m in range(3):
....:      n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:  j.append(n)
...
sage: G = graphics_array(j)
sage: G.show()  # long time
```

Next, using the spring-layout algorithm:

```
sage: # needs networkx sage.plot
sage: import networkx
sage: g = []
sage: j = []
sage: for i in range(9):
....:  spr = networkx.wheel_graph(i+3)
....:  k = Graph(spr)
....:  g.append(k)
...
sage: for i in range(3):
....:  n = []
....:  for m in range(3):
....:      n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
....:  j.append(n)
...
sage: G = graphics_array(j)
sage: G.show()  # long time
```

Compare the plotting:

```
sage: # needs networkx sage.plot
sage: n = networkx.wheel_graph(23)
sage: spring23 = Graph(n)
sage: posdict23 = graphs.WheelGraph(23)
sage: spring23.show()  # long time
sage: posdict23.show()  # long time
```

Chapter 3

Implemented Methods

This section mentions the lists of all the new functions, that are proposed to be implemented on SageMath — pertaining to the theory of matching covered graphs.

3.1 Perfect matchings and matching covered graphs

For a graph $G := (V, E)$, a *matching* is any subset of the edge set E, say M, such that $|M \cap \partial(v)$[1]$| \leqslant 1$ for each vertex $v \in V$. Here, for a vertex v, the notation $\partial(v)$ denotes the set of edges incident at that vertex. A matching M is a *maximum matching* if $|M| \geqslant |N|$ for each matching N of G. This leads us to the following problem:

> **Problem 3.1.1.** *Given a graph G, find a maximum matching.*

As discussed in the Section 'Existing Functions in SageMath', currently, the function `matching()` under the module "Undirected graphs" and the function `matching()` under the module "Bipartite graphs" compute the maximum (weighted) matching for a general graph and a bipartite graph respectively. Note that for the general case, there are two algorithms, namely the Edmonds algorithm (that utilizes Edmonds blossom algorithm [11] and works in $\mathcal{O}(|E| \cdot |V|^2)$), and the LP (that uses a Linear Programming Formulation) that have been implemented in the former mentioned method `matching()`. For bipartite graphs, the latter function `matching()` implements several algorithms, the efficient among which is the Hopcroft-Karp algorithm[13] ($\mathcal{O}(|E| \cdot \sqrt{|V|})$).

However, there is no implementation of the famous Micali-Vazirani algorithm [20] (described below) for an (unweighted) maximum matching of a general graph even though it has a time complexity of $\mathcal{O}(|E| \cdot \sqrt{|V|})$, which is (at least theoretically) significantly better than that of of the best among existing implemented algorithms.

3.1.1 matching() [Inclusion of Micali-Vazirani algorithm]

We propose the following modification to the existing method `matching()` under both "Undirected graphs" and "Bipartite graphs" to include the Micali-Vazirani algorithm to compute a maximum (unweighted) matching.

```
matching(value_only=False, algorithm='Edmonds', use_edge_labels=False,
↪  solver=False, verbose=None, integrality_tolerance=0)
```

Returns a maximum weighted matching of the graph represented by the list of its edges.

INPUT:

- `value_only` – boolean (default: `False`); when set to `True`, only the cardinal (or the weight) of the matching is returned
- `algorithm` – string (default: `'Edmonds'`)
  - `'Edmonds'` selects Edmonds algorithm as implemented in NetworkX
  - `'LP'` uses a Linear Program formulation of the matching problem
  - **'Micali-Vazirani' uses Micali-Vazirani algorithm [20] (Only for unweighted maximum matching)**
- `use_edge_labels` – boolean (default: `False`)
  - when set to `True`, computes a weighted matching where each edge is weighted by its label (if an edge has no label, 1 is assumed)
  - when set to False, each edge has weight 1
- `solver` – string (default: `None`); specify a Mixed Integer Linear Programming (MILP) solver to be used. If set to `None`, the default one is used. For more information on MILP solvers and which default solver is used, see the method solve of the class MixedIntegerLinearProgram.
- `verbose` – integer (default: `0`); sets the level of verbosity: set to 0 by default, which means quiet (only useful when algorithm == 'LP')
- `integrality_tolerance` – `float`; parameter for use with MILP solvers over an inexact base ring; see MixedIntegerLinearProgram.get_values().

OUTPUT:

- When `value_only=False` (default), this method returns an EdgesView containing the edges of a maximum matching of G.

- When `value_only=True`, this method returns the sum of the weights (default: 1) of the edges of a maximum matching of G. The type of the output may vary according to the type of the edge labels and the algorithm used.

ALGORITHM:

The problem is solved using Edmonds algorithm implemented in NetworkX, or using Linear Programming depending on the value of `algorithm`. If `algorithm` is set to `'Micali-Vazirani'`, then the problem is solved using the following algorithm:

Algorithm 1 : Micali-Vazirani algorithm [20]

if `use_edge_labels` is `True` **then**
  return 'Micali Vazirani computes the maximum unweighted matching. Please set either `use_edge_labels` to False or `algorithm` to anything valid apart from `'Micali-Vazirani'`.'
end if
$M \leftarrow$ a maximum matching found using Micali-Vazirani algorithm ▷ $\mathcal{O}(|E| \cdot \sqrt{|V|})$
return M
▷ Please refer to [20] for the pseudocode of the algorithm; it has not been explicitly written down here because of its length.

TIME COMPLEXITY: $\mathcal{O}(|E| \cdot \sqrt{|V|})$

3.1.2 has_perfect_matching() [Inclusion of a certificate]

A matching M of a graph G is a *perfect matching* if $|M \cap \partial(v)| = 1$ for each vertex v of G. A graph is *matchable* if it has a perfect matching and an edge e of a graph G is a `matchable edge` if there exists some perfect matching of M containing e. This raises the following decision problem:

Decision Problem 3.1.2. *Given a graph G, decide whether it is matchable.*

The method `has_perfect_matching()`, listed in the Section 'Existing Functions in SageMath', provides a poly-time algorithm for the above decision problem.

In 1947, Tutte [22] showed the necessary and sufficient condition for the existence of a perfect matching in a graph G that is stated in the following theorem.

Tuttes Theorem [22]

Theorem 3.1.3. *A graph G has a perfect matching if and only if*

$$o(G - S) \leqslant |S|$$

for every subset S *of* V*, where* $o(G - S)$ *refers to the number of odd components of* $G - S$*.*

If G does not have a perfect matching, then there exists a subset S of the vertex set V, such that

$$o(G - S) > |S|.$$

Such a set S is referred to as a *Tutte set*. However, there is no implementation in SageMath that finds the Tutte set of a graph, that is not matchable. We propose the the incorporation of a boolean argument `cetificate` in the existing method , that shall be `False` by default and when set to `True` shall output:

- an arbitrary perfect matching (if the graph is matchable), or otherwise
- an arbitrary Tutte set.

```
has_perfect_matching(algorithm='Edmonds', certificate=False, solver=None,
↪ verbose=0, integrality_tolerance)
```

Returns whether this graph has a perfect matching.

INPUT:

- `algorithm` – string (default: `'Edmonds'`)
  - `'Edmonds'` uses Edmonds algorithm as implemented in NetworkX to find a matching of maximal cardinality, then check whether this cardinality is half the number of vertices of the graph.
  - `'LP_matching'` uses a Linear Program to find a matching of maximal cardinality, then check whether this cardinality is half the number of vertices of the graph.
  - `'LP'` uses a Linear Program formulation of the perfect matching problem: put a binary variable $b[e]$ on each edge e, and for each vertex v, require that the sum of the values of the edges incident to v is 1.
  - `'Micali-Vazirani'` uses Micali-Vazirani algorithm [20] (Only for unweighted maximum matching)
- **certificate – boolean (default: False); when set to True, it outputs:**
  - **an arbitrary perfect matching (if the graph is matchable), or otherwise**

– **an arbitrary Tutte set.**

- `solver` – string (default: `None`); specify a Mixed Integer Linear Programming (MILP) solver to be used. If set to `None`, the default one is used. For more information on MILP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity: set to `0` by default, which means quiet (only useful when `algorithm == 'LP_matching'` or `algorithm == 'LP'`)
- `integrality_tolerance` – float; parameter for use with MILP solvers over an inexact base ring; see `MixedIntegerLinearProgram.get_values()`.

OUTPUT:

A boolean.

3.1.3 is_bicritical()

A matchable graph G is said to be *bicritical* if $G-u-v$ is matchable for every pair of distinct vertices u and v. Bicritcal graphs play a significant role in the theory of matching covered graphs as we will see in further sections. Consequently, this raises a decision problem.

Decision Problem 3.1.4. *Given a matchable graph G, decide whether it is bicritical.*

We state and prove the following theorem.

Theorem 3.1.5. *Given a matchable graph G and a matching M. Let u and v be two distinct vertices of G. The following statements are equivalent.*

1. *$G-u-v$ is matchable.*
2. *There exists an M-alternating odd length uv-path in G with the starting and the ending edge in M.*

Here the length of a path refers to the number of edges in that path.

Proof. Let G be a matching covered graph. Consider two distinct vertices u and v. Suppose $G-u-v$ is matchable. Let N be a perfect matching of $G-u-v$. Observe that each vertex in G distinct from u and v is matched to precisely one vertex in M and precisely some other vertex in N. u and v are the only vertices in G, that are N-exposed and M-matched. Thus, the symmetric difference of M and N in the graph G, generates a set of M-N alternating even cycles with precisely one M-N alternating odd-length path. This path has u and v as its ends and clearly, it starts and ends with edges that are in M, as u and v are M-matched. This is the required path.

Conversely, let P denote an M-alternating odd length uv-path that starts and ends with edges in M. Observe that $M \oplus P$ is a perfect matching of $G - u - v$, where $M \oplus N$ refers to the symmetric difference of the two (edge) subsets M and N in G. This completes the proof. □

Observe that since M is matchable, for distinct vertices u and v in G if there exist an M alternating odd-length uv-path in G with both starting and ending edges in M, there exists an M alternating odd-length uv-path in G with both starting and ending edges not in M. We shall use this observation and the method described in [15] to check whether a given matchable graph is bicritical or not.

```
is_bicritical(perfect_matching=None, algorithm='Micali-Vazirani', solver=None,
 ↪  conp_certificate=False, verbose=0, integrality_tolerace)
```

Checks whether the graph is bicritical.

INPUT:

- `perfect_matching` – (default: `None`); a perfect matching of the graph. It can be given using any valid input format of Graph.

  If set to `None`, a matching is computed using the other parameters.

- `algorithm` – string (default: `'Micali-Vazirani'`); the algorithm to use to compute a maximum matching of the graph among
  - `'Micali-Vazirani'` uses Micali-Vazirani algorithm [20] (Only for unweighted maximum matching)
  - `'Edmonds'` selects Edmonds algorithm as implemented in NetworkX,
  - `'LP_matching'` uses a Linear Program to find a matching of maximal cardinality, then check whether this cardinality is half the number of vertices of the graph.
  - `'LP'` uses a Linear Program formulation of the matching problem.
- `conp_certificate` – boolean (default: `False`); when set to `True` outputs an edge e such that e is not matchable in G, if G is not matching covered.
- `solver` – string (default: `None`); specify a Mixed Integer Linear Programming (MILP) solver to be used. If set to `None`, the default one is used. For more information on MILP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: `0`); sets the level of verbosity: set to `0` by default, which means quiet (only useful when `algorithm == 'LP'`)

- `integrality_tolerance` – float; parameter for use with MILP solvers over an inexact base ring; see `MixedIntegerLinearProgram.get_values()`.

OUTPUT:

a boolean

ALGORITHM:

Algorithm 2 : Decide whether G is bicritical [15]

if G has at most 1 vertex **then** ▷ $\mathcal{O}(1)$
  if `certificate` is `True` **then** ▷ $\mathcal{O}(1)$
    return False, 'G has at most one vertex.' ▷ $\mathcal{O}(1)$
  end if
  return `False` ▷ $\mathcal{O}(1)$
end if
if G is not connected **then** ▷ $\mathcal{O}(|E| + |V|)$
  if `certificate` is `True` **then** ▷ $\mathcal{O}(1)$
    return False, 'G is not connected.' ▷ $\mathcal{O}(1)$
  end if
  return `False` ▷ $\mathcal{O}(1)$
end if
Let exist ← `True` ▷ $\mathcal{O}(1)$
Construct an adjacency list for G ▷ $\mathcal{O}(|E|)$
if `pefect_matching` is `None` **then** $M \leftarrow G$`.matching()` ▷ $\mathcal{O}(|E| \cdot \sqrt{|V|})$
end if
if M is not a perfect matching of G **then**
  if `conp_certificate` is `True` **then**
    $e \leftarrow$ an arbitrary edge in M ▷ $\mathcal{O}(1)$
    $u, v \leftarrow$ ends of e ▷ $\mathcal{O}(1)$
    return `False`, $\{u$ and $v\}$ ▷ $\mathcal{O}(1)$
  end if
  return `False` ▷ $\mathcal{O}(1)$
end if
$u \leftarrow$ an arbitrary vertex in G. ▷ $\mathcal{O}(1)$
for each u in $V(G) - u$ **do** ▷ $\mathcal{O}(|V|)$
  `exist` ← M-Alternating Path Search(u) ▷ $\mathcal{O}(|E|)$ ▷ Please refer to [15] for how to perform an efficient M-alternating Path Search by constructing an M-alternating tree for given perfect matching M for a vertex v in $\mathcal{O}(|E|)$.
  if `exist` is `False` **then**
    if `certificate` is `True` **then**

$v \leftarrow$ a vertex that is not reachable from u via an M-alternating odd-length path starting and ending with edges not in M $\triangleright$ $\mathcal{O}(1)$
return `False`, $\{u$ and $v\}$ $\triangleright$ $\mathcal{O}(1)$
end if
return `False`, $\{u$ and $v\}$ $\triangleright$ $\mathcal{O}(1)$
end if
end for
return `True` $\triangleright$ $\mathcal{O}(1)$

TIME COMPLEXITY: $\mathcal{O}(|V| \cdot |E|)$.

EXAMPLE:

- Any matchable bipartite graph is not bicritical.
- All Wheel graphs, that are matchable, are bicritical.

3.1.4 is_matching_covered()

A graph is nontrivial if its order is at least two. A *matching covered graph* is a connected nontrivial graph in which each edge participates in some perfect matching. The reader may easily verify the fact that a graph is matching covered if and only if its underlying simple graph is matching covered. This immediately brings us to the subsequent decision problem.

Decision Problem 3.1.6. *Given a connected nontrivial graph G, decide whether it is matching covered.*

For a nonbipartite graph, we shall use Theorem 3.1.5 and the method of constructing an M-alternating tree as described in [14] to check whether a graph is matching covered or not. It shall work in with a worst time complexity of $\mathcal{O}(|E|^2)$ provided we have a perfect matching M of G. Interestingly, if we have a perfect matching M of G, for a bipartite graph G, we can check if G is matching covered in linear time using the following theorem. This theorem has been adopted from the book [18].

Theorem 3.1.7. *Let $G[A,B]$ be a bipartite graph and let M be a perfect matching of G. Let H be the graph obtained from G by the addition of a parallel edge to each edge of M . Let D be the directed graph obtained from H by directing every edge of $E(H) - M$ from A to B and by directing every edge of M from B to A. It holds that*

G is matching covered if and only if D is strongly connected.

Proof. Suppose that D is strongly connected. Clearly, H is connected; hence G is connected. Note that the edges of M are matchable. Thus, it suffices to show that each edge in $E(G)-H$ is matchable.

Let ab denote an edge of $E(G) - M$, where $a \in A$ and $b \in B$. As D is strongly connected, there is in D a directed path, P, from b to a. Thus, P plus ab is a directed cycle in D. The corresponding cycle in G is M-alternating and contains the edge ab, hence ab is matchable. This conclusion holds for each edge ab in $E(G) - M$. We deduce that every edge of G is matchable. As G is connected, in fact G is matching covered.

Conversely, suppose that G is matching covered. Clearly, G is connected; hence so too are H and D. To prove that D is strongly connected we must prove that every arc uv of D is in a directed cycle. If uv is in M or if uv corresponds to an edge added to H as a parallel edge of an edge in M then uv is in a directed cycle of order two. Assume then that uv corresponds to an edge of $E(G) - M$. As G is matching covered, the edge uv is matchable, hence uv is in an M-alternating cycle, Q. The corresponding cycle in D is directed. Thus, uv is in a directed cycle of D. We conclude that every arc of D is in a directed cycle. As D is connected, we conclude that D is strongly connected. □

```
is_matching_covered(perfect_matching=None, algorithm='Micali-Vazirani',
 ↪  solver=None, conp_certificate=False, verbose=0, integrality_tolerace)
```

Checks whether the graph is matching covered.

INPUT:

- `perfect_matching` – (default: `None`); a perfect matching of the graph. It can be given using any valid input format of Graph.

  If set to `None`, a matching is computed using the other parameters.

- `algorithm` – string (default: `'Micali-Vazirani'`); the algorithm to use to compute a maximum matching of the graph among
  - `'Edmonds'` selects Edmonds algorithm as implemented in NetworkX,
  - `'LP_matching'` uses a Linear Program to find a matching of maximal cardinality, then check whether this cardinality is half the number of vertices of the graph.
  - `'LP'` uses a Linear Program formulation of the matching problem.

- `conp_certificate` – boolean (default: `False`); when set to `True` outputs either a comment or two vertices u and v such that $G - u - v$ is not matchable, if G is not bicritical.

- `solver` – string (default: `None`); specify a Mixed Integer Linear Programming (MILP) solver to be used. If set to `None`, the default one is used. For more information on

MILP solvers and which default solver is used, see the method solve of the class MixedIntegerLinearProgram.

- `verbose` – integer (default: `0`); sets the level of verbosity: set to `0` by default, which means quiet (only useful when `algorithm == 'LP'`)
- `integrality_tolerance` – float; parameter for use with MILP solvers over an inexact base ring; see `MixedIntegerLinearProgram.get_values()`.

OUTPUT:

a boolean

ALGORITHM:

Algorithm 3 : Check if G is matching covered

```
if G is not connected nontrivial then                                   ▷ O(1)
    return                    ▷ A matching covered graph is connected and nontrivial
end if
if perfect_matching is None then
    M ← G.matching()                                                    ▷ O(|E| · √|V|)
end if
if M is not a perfect matching of G then
    if conp_certificate is True then
        e ← an arbitrary edge in M                                      ▷ O(1)
        return False, e                                                 ▷ O(1)
    end if
    return False                                                        ▷ O(1)
end if
if G is bipartite then                                                  ▷ O(|E| + |V|)
    Construct the digraph D, as defined in Theorem 3.1.7                ▷ O(|E| + |V|)
    Determine whether D is strongly connected                           ▷ O(|E| + |V|); see [8]
    if D is not strongly connected then
        if conp_certificate is True then
            e ← an arbitrary edge in D, that does not participate in a directed cycle
  ▷ Alearly found while checking if D is strongly connected or not
            return False, e                                             ▷ O(1)
        end if
        return False                                                    ▷ O(1)
    else
        return True
    end if
```

```
else
    for each vertex u in G do                                              ▷ O(|V|)
        Let e ← ∂(u) ∩ M and let v ← M(u)
        exist ← True if there exists an M-alternating uw-path in G − e to each
w ∈ ∂(u) − v                                                    ▷ O(|E|); see [14]
        if not exist for some w then
            if conp_certificate is True then
                return False, uw                                           ▷ O(1)
            end if
            return False                                                   ▷ O(1)
        else
            continue
        end if
    end for
    return True
end if
```

TIME COMPLEXITY:

- $\mathcal{O}(|E| \cdot |V|)$ if G is nonbipartite, or otherwise
- $\mathcal{O}(|E| + |V|)$.

EXAMPLE:

- K_{nm} for $n > m > 1$ is not matchable, hence is not matching covered.
- C_4 plus the (unique) edge, that is not a multiple/ parallel edge, is matchable, but not matching covered.
- All Wheel graphs are matching covered.

3.2 Barriers and Canonical Partition

For a graph G, a subset B of the vertex set is a *barrier* if $|U| = o(G - B) - |B|$, where $|U| = |V(G)| - 2|M|$. Here $|M|$ denotes the cardinality of the maximum matching of G and $o(G - B)$ denotes the number of odd components in $G - B$. For a matchable graph G, note that $|U|$ is precisely zero; thus, $o(G - B) = |B|$. The empty set and all singletons are barriers for every matchable graph. Such a barrier is known as a *trivial barrier*. The reader may easily verify the following proposition (which will be used in proving the theorem next).

Proposition 3.2.1. *Let G be a graph, and let X be a subset of V. It holds that:*

$$|X| - o(G - X) \equiv |V| \mod 2.$$

The following theorem will be helpful in the implementation of the next algorithm which we will discuss soon after.

Theorem 3.2.2. *Let u and v be any two vertices in a matchable graph G. Then the graph $G-u-v$ is matchable if and only if there is no barrier of G which contains both u and v.*

Proof. Suppose that $G-u-v$ is matchable. Our goal is to show that each subset of V that contains both u and v is not a barrier of G. Let $B \subseteq V$ such that $u, v \in B$. Let S denote the set $B-u-v$. Observe that

$$\begin{aligned} o(G-B) &= o((G-u-v)-S) \\ &\leqslant |S| \ (\because G-u-v \text{ is matchable}) \\ &= |B|-2 \end{aligned}$$

Consequently, $o(G-B) < |B|$. Hence, B is not a barrier of G.

Conversely, suppose that $G-u-v$ is matchable. By Proposition 3.2.1 and Theorem 3.1.3, there exists a subset S of $V(G-u-v)$ such that

$$o(G-u-v-S) \geqslant |S|+2.$$

Let $B := S+u+v$; clearly,

$$o(G-B) \geqslant |B|.$$

Since G is matchable, by Theorem 3.1.3, the strict inequality can not hold. Thus, $o(G-B) = |B|$. Thus, $|B|$, that contains both u and v, is the required barrier of G. □

Two vertices u and v are *kotzig related* if $G-u-v$ is not matchable.

3.2.1 maximal_barrier()

For a graph G, a barrier B is a maximal barrier if C is not a barrier for each C such that $B \subset C \subseteq V$. The following beautiful result concerning matching covered graphs and maximal barriers is shown by Kotzig; see "Matching Theory" [16].

The canonical partition theorem [16]

Theorem 3.2.3. *The maximal barriers of a matching covered graph G partition its vertex set, and this partition is called its canonical partition.*

Henceforth, each vertex in a matching covered graph participates in a unique maximal barrier (this need not be true for any graph). Thus, the kotzig relation is an equivalence relation for a matching covered graph. This raises the following problem.

Decision Problem 3.2.4. *Given a matching covered graph G and vertex v of it; find the maximal barrier containing the vertex.*

We shall use 3.2.2 and an analogous approach discussed in [14] to develop an efficient algorithm that shall answer the above problem.

```
maximal_barrier_in_matching_covered_graph(v, perfect_matching=None,
 ↪ matching_covered_check=True, algorithm='Micali-Vazirani')
```

Returns the (unique) maximal barrier of a matching covered graph G containing the vertex v.

INPUT:

- the vertex v of the graph G; we shall find the (unique) maximal barrier of G containing v.
- `perfect_matching` – (default: `None`); a perfect matching of the graph. It can be given using any valid input format of Graph.

  If set to `None`, a maximum matching is computed using the other parameters.
- `matching_covered_check` – (default: `True`); Before computing the maximal barrier containing v, we shall ensure that G is matching covered.

  If set to `False`, this check is skipped.
- `algorithm` – string (default: `'Micali-Vazirani'`)
  - `'Edmonds'` selects Edmonds algorithm as implemented in NetworkX
  - `'LP'` uses a Linear Program formulation of the matching problem
  - `'Micali-Vazirani'` uses Micali-Vazirani algorithm [20] (Only for unweighted maximum matching)

OUTPUT:

A set B of vertices such that B is the (unique) maximal barrier containing v.

ALGORITHM:

Algorithm 4 : Finding the (unique) maximal barrier containing v

1: **if** `perfect_matching` is `None` **then** ▷ $\mathcal{O}(1)$
2: Let M be a maximum matching of G ▷ $\mathcal{O}(|E| \cdot \sqrt{|V|})$
3: **end if**
4: **if** M is not a perfect matching **then** ▷ $\mathcal{O}(1)$

    return ‘G is not matching covered.’ ▷ $\mathcal{O}(1)$
end if
if `matching_covered_check` is `True` **then** ▷ $\mathcal{O}(1)$
    if G is not matching covered **then** ▷ $\mathcal{O}(|E| \cdot |V|)$
        return ‘G is not matching covered.’ ▷ This method is defined for a matching covered graph
    end if
end if
$B \leftarrow V$ ▷ Initialize the set of maximal barrier containing v
$u \leftarrow M(v)$ ▷ Let u be the M-neighbor of v
$B \leftarrow B - w$, for each vertex w in $G - v$, that are reachable from u via an even-length M-alternating path with the starting edge not in M and the ending edge in M, that is done by constructing an M-alternating tree of $G - v$ ▷ $\mathcal{O}(|E|)$ for all w, for a specified u; see: [14]
return B ▷ $\mathcal{O}(1)$

TIME COMPLEXITY: $\mathcal{O}(|E|)$

EXAMPLE:

- In bipartite matching covered graphs, each of the color class is the maximal barrier.
- In bicritical graphs, each individual vertex in the maximal barrier.

3.2.2 canonical_partition()

As discussed above, we shall compute the canonical partition of a matching covered graph as follows.

```
canonical_partition(perfect_matching=None, matching_covered_check=True,
 ↪  algorithm='Micali-Vazirani')
```

Return the canonical partition of G.

INPUT:

- `perfect_matching` – (default: `None`); a perfect matching of the graph. It can be given using any valid input format of Graph.

  If set to `None`, a maximum matching is computed using the other parameters.

- `matching_covered_check` – (default: `True`); Before computing the canonical partition of G, we shall ensure that G is matching covered.

  If set to `False`, this check is skipped.

- `algorithm` – string (default: `'Micali-Vazirani'`)
  - `'Edmonds'` selects Edmonds algorithm as implemented in NetworkX
  - `'LP'` uses a Linear Program formulation of the matching problem
  - `'Micali-Vazirani'` uses Micali-Vazirani algorithm [20] (Only for unweighted maximum matching)

OUTPUT:

- It returns a list of (partition) sets of vertices, each of which sets is a maximal barrier.

ALGORITHM:

Algorithm 5 : Canonical Partition of a matching covered graph

if `perfect_matching` is `None` **then** ▷ $\mathcal{O}(1)$
    Let M be a maximum matching of G ▷ $\mathcal{O}(|E| \cdot \sqrt{|V|})$
end if
if M is not a perfect matching **then** ▷ $\mathcal{O}(1)$
    return 'G is not matching covered.' ▷ $\mathcal{O}(1)$
end if
if `matching_covered_check` is `True` **then** ▷ $\mathcal{O}(1)$
    if G is not matching covered **then** ▷ $\mathcal{O}(|E| \cdot |V|)$
        return 'G is not matching covered.' ▷ Canonical Partition is defined for a matching covered graph
    end if
end if
$Z \leftarrow V$ ▷ Make a copy of the set of vertices
$\mathcal{B} \leftarrow [\,]$ ▷ Initialize the list of maximal barriers
for v in Z **do** ▷ $\mathcal{O}(|V|)$
    $B \leftarrow$ `G.maximal_barrier_in_matching_covered_graph(`v`, perfect_matching =`M`, matching_covered_check=False)` ▷ $\mathcal{O}(|E|)$; compute the maximal barrier containing v
    Append B to $\mathcal{B}$ ▷ B is the maximal barrier containing v
    $Z \leftarrow Z - B$ ▷ Kotzig relation is an equivalence relation for a matching covered graph
end for
return $\mathcal{B}$

TIME COMPLEXITY: $\mathcal{O}(|E| \cdot |V|)$

EXAMPLE:

- In bipartite matching covered graphs, each of the color class constitute the canonical partition.
- In bicritical graphs, sets each containing an individual vertex, constitute the canonical partition.

3.3 Tight Cuts

Recall, for a graph G, the notation $\partial(v)$ is used to denote the set of edges that are incident at the vertex v. Analogously, for a set $S \subseteq V(G)$, the notation $\partial(S)$ denotes the set of edges that have one end in S and the other end in $\overline{S} := V(G) - S$. This leads us to the definition of a cut.

A cut C of a matching covered graph G is called a *tight cut* if $|M \cap C| = 1$, for each perfect matching M of G. This tight cut plays an important role in the well-known tight cut decomposition of matching covered graphs, which we will discuss subsequently.

We discuss below two special types of tight cuts in matching covered graphs. Let B be any barrier of a matching covered graph G. We state the following proposition without the proof.

Proposition 3.3.1. *Let B be a barrier in a matchable graph G, and let M be any perfect matching of G. Then:*

1. *if K is an odd component of $G - B$, then $M \cap \partial(K)$ has precisely one edge; and if v is the end of that edge in $V(K)$, then $M \cap E(K)$ is a perfect matching of $K - v$, and*
2. *if L is an even component of $G - B$, then $M \cap E(L)$ is a perfect matching of L and no edge in $\partial(L)$ is matchable in G.*

It follows from Proposition 3.3.1 that, for each (odd) component K of $G - B$, the cut $\partial(K)$ is a tight cut in G. Such a tight cut, that arises from the odd component of a barrier, is known as *a barrier cut.*

For matching covered graph G a 2-vertex cut $\{u, v\}$ of G that is not a barrier, is called *a 2-separation.* If $\{u, v\}$ is a 2-separation of G, then each component of $G - u - v$ is even. Suppose that H_1 is the union of a nonempty proper subset of the components of $G - u - v$, and H_2 is the union of the remaining components of $G - u - v$. Then H_1 and H_2 are two vertex-disjoint even order subgraphs whose union is $G - u - v$.

With any such expression of $G - u - v$ as the union of H_1 and H_2, we may associate the cuts $C := \partial(V(H_1) + v)$ and $D := \partial(V(H_2) + v)$, and it is easy to see that both C and D are tight. Such a tight cut that arises in this manner is known as *2-separation* cut of G.

Interestingly matching covered graph may have a tight cut which is neither a barrier cut, nor a 2-separation cut. Figure 3.1 shows an example for the same [18].

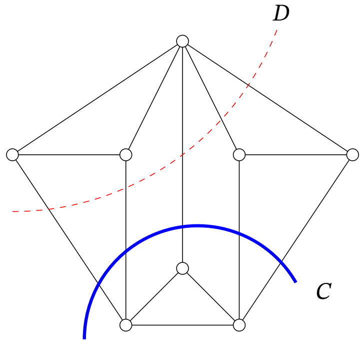


Figure 3.1: The cut C (shown in bold blue) is not an ELP cut, but D (shown in dashed red) is a 2-separation cut

In 1982, Edmonds, Lovász and Pulleyblank [10] proved the following theorem.

ELP Theorem [10]

Theorem 3.3.2. *If a matching covered graph has a nontrivial tight cut then it has a nontrivial barrier cut or a (nontrivial) 2-separation cut.*

Carvalho, Lucchesi and Murty refer to this assertion as the ELP Theorem and use the term ELP cut to mean either a barrier cut or a 2-separation cut.

3.3.1 tight_cut_decomposition()

A matching covered graph free of tight cuts is called *a brace* if it is bipartite and *a brick* if it is nonbipartite.

Tight cut decomposition: Given any matching covered graph G, we may apply to it a procedure, called a tight cut decomposition of G, which produces a list of bricks and braces. If G itself is a brick or a brace then the list consists of just G. Otherwise, let C be any nontrivial tight cut of G. Then, both C-contractions of G are matching covered. One may recursively apply the tight cut decomposition procedure to each C-contraction of G, and then combine the resulting lists to produce a tight cut decomposition of G itself.

Lovász [17] proved the following remarkable result on tight cut decomposition.

The unique decomposition theorem [17])

Theorem 3.3.3. *Any two applications of the tight cut decomposition procedure to a matching covered graph G produce the same list of bricks and braces, up to multiple edges.*

This raises the following question.

Problem 3.3.4. *Given a matching covered graph G, find one of its tight cut decomposition.*

Let's define some notations as per the paper entitled "On a Conjecture of Lovász Concerning Bricks: II. Bricks of Finite Characteristic" [3] which shall be helpful in determining the structure of the output in the method `tight_cut_decomposition()`. Let G be a matching covered graph. Let $C := \partial(X)$ and $D := \partial(Y)$ be two odd cuts of G. The four sets $X \cap Y$, $X \cap \overline{Y}$, $\overline{X} \cap Y$ and $\overline{X} \cap \overline{Y}$ are the quadrants defined by C and D. The cuts C and D *cross* if each of these four quadrants is nonnull. A collection C of cuts of G is *laminar* if no two of its members cross. A set $\mathcal{C}$ of laminar nontrivial tight cuts is called a *maximal set/ collection of laminar nontrivial tight cuts* if there does not exist any nontrivial tight cut in G that is distinct from and laminar to each of the nontrivial tight cuts in $\mathcal{C}$. The following interesting theorem concerning laminar nontrivial tight cuts conjectured by Carvalho, Lucchesi and Murty [4] and proved by Chen, Feng, Lu, Lucchesi, and Zhang [7]:

Laminar ELP theorem [7]

Theorem 3.3.5. *If C is a tight cut in a matching covered graph G, then there exists either a barrier cut or a 2-separation cut in G, that is laminar to C.*

The concerned method shall output a maximal set of laminar nontrivial tight cuts (in fact non trivial ELP cuts), so that the application of the tight cut decomposition procedure on G with these set of nontrivial tight cut (in any order) will result in a list of bricks and braces (that are unique up to multiple edges) of the graph G. The following algorithm has been adopted from the book [18].

```
tight_cut_decomposition(perfect_matching=None, algorithm='Micali-Vazirani',
 ↪  matching_covered_check=True)
```

Returns a maximal set of laminar nontrivial tight cuts of G

INPUT:

- `perfect_matching` – (default: `None`); a perfect matching of the graph. It can be given using any valid input format of Graph.

  If set to `None`, a maximum matching is computed using the other parameters.

- `matching_covered_check` – (default: `True`); Before computing the tight cut decomposition of G, we shall ensure that G is matching covered.

  If set to `False`, this check is skipped.

- `algorithm` – string (default: `'Micali-Vazirani'`)
  - `'Edmonds'` selects Edmonds algorithm as implemented in NetworkX
  - `'LP'` uses a Linear Program formulation of the matching problem
  - `'Micali-Vazirani'` uses Micali-Vazirani algorithm [20] (Only for unweighted maximum matching)

OUTPUT:

- A maximal set of laminar nontrivial tight cuts

ALGORITHM:

Algorithm 6 : Tight cut decomposition of a matching covered graph G [18]

if `perfect_matching` is `None` **then** ▷ $\mathcal{O}(1)$
  Let M be a maximum matching of G ▷ $\mathcal{O}(|E| \cdot \sqrt{|V|})$
end if
if M is not a perfect matching **then** ▷ $\mathcal{O}(1)$
  return 'G is not matching covered.' ▷ $\mathcal{O}(1)$
end if
if `matching_covered_check` is `True` **then** ▷ $\mathcal{O}(1)$
  if G is not matching covered **then** ▷ $\mathcal{O}(|E| \cdot |V|)$
    return 'G is not matching covered.' ▷ $\mathcal{O}(1)$
  end if
end if
if G is bipartite **then** ▷ $\mathcal{O}(|E| + |V|)$
  Use `G.is_brace()` algorithm to either attest that G is a brace, or otherwise find a nontrivial tight cut C. ▷ $\mathcal{O}(|E| \cdot |V|)$
else ▷ G is nonbipartite
  Use `G.is_brick()` algorithm to either attest that G is a brick, or otherwise find a nontrivial tight cut C. ▷ $\mathcal{O}(|E| \cdot |V|)$
end if
Let H and J be the C contraction of G. Repeat the above procedure to determine if H and J are brick/ brace, or otherwise obtain a nontrivial tight cut C.
Repeat the entire procedure until the graphs obtained are free of tight cuts. ▷ This repetition shall occur at most $\mathcal{O}(|V|)$ times (described below).
Let $\mathcal{C}$ denote the set of laminar nontrivial tight cuts obtained in this procedure
return $\mathcal{C}$

TIME COMPLEXITY:
Note that the number of nontrivial tight cuts of the tight cut decomposition is at most $\frac{|V|}{2}$.

Thus the algorithm runs in $\mathcal{O}(|E| \cdot |V|^2)$ time.

3.3.2 bricks_and_braces()

In the previous section, we saw that for a matching covered graph G, the list of the underlying simple graphs of each of its bricks and the braces, is an invariant. The following method shall list down the underlying simple graphs of all bricks and braces of a matching covered graph G. This algorithm has been adopted from the book [18].

```
bricks_and_braces(perfect_matching=None, algorithm='Micali-Vazirani',
 ↪ matching_covered_check=True, only_brick=False, only_brace=False)
```

Returns the list of bricks and braces of the matching covered graph G, that are invariant of G (up to multiple edges)

INPUT:

- `perfect_matching` – (default: `None`); a perfect matching of the graph. It can be given using any valid input format of Graph.

  If set to `None`, a maximum matching is computed using the other parameters.

- `matching_covered_check` – (default: `True`); Before computing the list of bricks and braces of G, we shall ensure that G is matching covered.

  If set to `False`, this check is skipped.

- `algorithm` – string (default: `'Micali-Vazirani'`)
  - `'Edmonds'` selects Edmonds algorithm as implemented in NetworkX
  - `'LP'` uses a Linear Program formulation of the matching problem
  - `'Micali-Vazirani'` uses Micali-Vazirani algorithm [20] (Only for unweighted maximum matching)

- `only_brick` – (default: `False`); if set to `True`, outputs only the list of bricks.

- `only_brace` – (default: `False`); if set to `True`, outputs only the list of braces.

OUTPUT:

- A list of bricks and braces

ALGORITHM:

Algorithm 7 : Bricks and Braces of a matching covered graph G [18]

if `perfect_matching` is `None` **then** ▷ $\mathcal{O}(1)$
  Let M be a maximum matching of G ▷ $\mathcal{O}(|E| \cdot \sqrt{|V|})$
end if
if M is not a perfect matching **then** ▷ $\mathcal{O}(1)$
  return 'G is not matching covered.' ▷ $\mathcal{O}(1)$
end if
if `matching_covered_check` is `True` **then** ▷ $\mathcal{O}(1)$
  if G is not matching covered **then** ▷ $\mathcal{O}(|E| \cdot |V|)$
    return 'G is not matching covered.' ▷ $\mathcal{O}(1)$
  end if
end if
if G is bipartite **then** ▷ $\mathcal{O}(|E| + |V|)$
  Use `G.is_brace()` algorithm to either attest that G is a brace, or otherwise find a nontrivial tight cut C. ▷ $\mathcal{O}(|E| \cdot |V|)$
else ▷ G is nonbipartite
  Use `G.is_brick()` algorithm to either attest that G is a brick, or otherwise find a nontrivial tight cut C. ▷ $\mathcal{O}(|E| \cdot |V|)$
end if
Let H and J be the C contraction of G. Repeat the above procedure to determine if H and J are brick/ brace, or otherwise obtain a nontrivial tight cut C.
Repeat the entire procedure until the graphs obtained are free of tight cuts. ▷ This repetition shall occur at most $\mathcal{O}(|V|)$ times (described below).
Let Ω denote the final list of bricks and braces
return Ω

TIME COMPLEXITY:
Note that the number of nontrivial tight cuts of the tight cut decomposition is at most $\frac{|V|}{2}$. Thus the algorithm runs in $\mathcal{O}(|E| \cdot |V|^2)$ time.

3.3.3 number_of_bricks()

One particular consequence of Lovászs Theorem is that any two tight cut decompositions of a given matching covered graph G yield the same numbers of bricks and braces. Carvalho, Lucchessi and Murty [3] refer to these two invariants as the numbers of bricks, denoted as $b(G)$ and the number of braces, denoted as $b'(G)$ of G, respectively.

The following method shall output the 'number of bricks' invariant of a matching covered graph G.

```
no_of_bricks(perfect_matching=None, algorithm='Micali-Vazirani',
 ↪ matching_covered_check=True)
```

Returns the number of bricks of the matching covered graph G, aka $b(G)$, which is an invariant of G.

INPUT:

- `perfect_matching` – (default: `None`); a perfect matching of the graph. It can be given using any valid input format of Graph.

  If set to `None`, a maximum matching is computed using the other parameters.

- `matching_covered_check` – (default: `True`); Before computing the number of bricks of G, we shall ensure that G is matching covered.

  If set to `False`, this check is skipped.

- `algorithm` – string (default: `'Micali-Vazirani'`)
  - `'Edmonds'` selects Edmonds algorithm as implemented in NetworkX
  - `'LP'` uses a Linear Program formulation of the matching problem
  - `'Micali-Vazirani'` uses Micali-Vazirani algorithm [20] (Only for unweighted maximum matching)

OUTPUT:

- The number of bricks of G, aka $b(G)$

ALGORITHM:

Algorithm 8 : Number of bricks of a matching covered graph G [18]

if `perfect_matching` is `None` **then** ▷ $\mathcal{O}(1)$
  Let M be a maximum matching of G ▷ $\mathcal{O}(|E| \cdot \sqrt{|V|})$
end if
if M is not a perfect matching **then** ▷ $\mathcal{O}(1)$
  return 'G is not matching covered.' ▷ $\mathcal{O}(1)$
end if
if `matching_covered_check` is `True` **then** ▷ $\mathcal{O}(1)$
  if G is not matching covered **then** ▷ $\mathcal{O}(|E| \cdot |V|)$
    return 'G is not matching covered.' ▷ $\mathcal{O}(1)$
  end if
end if
Find the list of bricks of G using `G.number_of_bricks()` ▷ $\mathcal{O}(|E| \cdot |V|^2)$

13: $b \leftarrow$ the number of bricks of G
14: **return** b

TIME COMPLEXITY:
Note that the number of nontrivial tight cuts of the tight cut decomposition is at most $\frac{|V|}{2}$. Thus the algorithm runs in $\mathcal{O}(|E| \cdot |V|^2)$ time.

3.3.4 number_of_braces()

The following method shall output the 'number of braces' invariant of a matching covered graph G.

```
no_of_braces(perfect_matching=None, algorithm='Micali-Vazirani',
 ↪ matching_covered_check=True)
```

Returns the number of braces of the matching covered graph G, aka $b'(G)$, which is an invariant of G.

INPUT:

- `perfect_matching` – (default: `None`); a perfect matching of the graph. It can be given using any valid input format of Graph.

  If set to `None`, a maximum matching is computed using the other parameters.

- `matching_covered_check` – (default: `True`); Before computing the number of braces of G, we shall ensure that G is matching covered.

  If set to `False`, this check is skipped.

- `algorithm` – string (default: '`Micali-Vazirani`')
  - '`Edmonds`' selects Edmonds algorithm as implemented in NetworkX
  - '`LP`' uses a Linear Program formulation of the matching problem
  - '`Micali-Vazirani`' uses Micali-Vazirani algorithm [20] (Only for unweighted maximum matching)

OUTPUT:

- The number of braces of G, aka $b'(G)$

ALGORITHM:

Algorithm 9 : Number of braces of a matching covered graph G [18]

if `perfect_matching` is `None` **then** ▷ $\mathcal{O}(1)$
    Let M be a maximum matching of G ▷ $\mathcal{O}(|E| \cdot \sqrt{|V|})$
end if
if M is not a perfect matching **then** ▷ $\mathcal{O}(1)$
    return 'G is not matching covered.' ▷ $\mathcal{O}(1)$
end if
if `matching_covered_check` is `True` **then** ▷ $\mathcal{O}(1)$
    if G is not matching covered **then** ▷ $\mathcal{O}(|E| \cdot |V|)$
        return 'G is not matching covered.' ▷ $\mathcal{O}(1)$
    end if
end if
Find the list of braces of G using `G.number_of_braces()` ▷ $\mathcal{O}(|E| \cdot |V|^2)$
$b' \leftarrow$ the number of bricks of G
return b'

TIME COMPLEXITY:
Note that the number of nontrivial tight cuts of the tight cut decomposition is at most $\dfrac{|V|}{2}$. Thus the algorithm runs in $\mathcal{O}(|E| \cdot |V|^2)$ time.

3.3.5 number_of_petersen_bricks()

For a matching covered graph G, it turns out that the number of Petersen bricks, that are those bricks whose underlying simple graph is the Petersen graph, denoted as $p(G)$, which is also an invariant of G, plays an important role in the theory of matching covered graphs, for instance in determining the optimal ear decomposition of a matching covered graph; see [5]. We shall implement the following function to obtain $p(G)$ for any matching covered graph G.

```
no_of_bricks(perfect_matching=None, algorithm='Micali-Vazirani',
 ↪ matching_covered_check=True)
```

Returns the number of petersen bricks of the matching covered graph G, aka $p(G)$, which is an invariant of G.

INPUT:

- `perfect_matching` – (default: `None`); a perfect matching of the graph. It can be given using any valid input format of Graph.

  If set to `None`, a maximum matching is computed using the other parameters.

- `matching_covered_check` – (default: `True`); Before computing the number of Petersen bricks of G, we shall ensure that G is matching covered.

  If set to `False`, this check is skipped.

- `algorithm` – string (default: `'Micali-Vazirani'`)
  - `'Edmonds'` selects Edmonds algorithm as implemented in NetworkX
  - `'LP'` uses a Linear Program formulation of the matching problem
  - `'Micali-Vazirani'` uses Micali-Vazirani algorithm [20] (Only for unweighted maximum matching)

OUTPUT:

- The number of Petersen bricks of G, aka $p(G)$

ALGORITHM:

Algorithm 10 : Number of Petersen bricks of a matching covered graph G [18]

```
if perfect_matching is None then                                ▷ O(1)
    Let M be a maximum matching of G                            ▷ O(|E| · √|V|)
end if
if M is not a perfect matching then                             ▷ O(1)
    return 'G is not matching covered.'                         ▷ O(1)
end if
if matching_covered_check is True then                          ▷ O(1)
    if G is not matching covered then                           ▷ O(|E| · |V|)
        return 'G is not matching covered.'                     ▷ O(1)
    end if
end if
Find the list of bricks of G using G.number_of_bricks()         ▷ O(|E| · |V|^2)
p ← the number of Petersen bricks of G
return p
```

TIME COMPLEXITY:
Note that the number of nontrivial tight cuts of the tight cut decomposition is at most $\frac{|V|}{2}$. Thus the algorithm runs in $\mathcal{O}(|E| \cdot |V|^2)$ time.

3.4 Bricks and Braces

In this section, we shall investigate some fundamental algorithms for instance — checking if a given matching covered graph is a brick or a brace; also we shall see some interesting family of graphs that play a crucial role in generating (all) bricks and braces (which we will see in chapter 'Strictly thin edges in Bricks and Braces').

3.4.1 is_brick()

The ELP Theorem implies the following characterization of bricks, as discovered by Edmonds, Lovász and Pulleyblank [10].

Characterization of Bricks [10]

Theorem 3.4.1. *A nonbipartite matching covered graph is a brick if and only if it is 3-connected and bicritical.*

Given a matching covered nonbipartite graph G, the following method shall determine whether it is a brick or shall output a nontrivial (ELP) tight cut.

```
is_brick(perfect_matching=None, algorithm='Micali-Vazirani',
↪  matching_covered_check=True, conp_certificate=False)
```

Checks whether the matching covered nonbipartite graph G is a brick.

INPUT:

- `perfect_matching` – (default: `None`); a perfect matching of the graph. It can be given using any valid input format of Graph.

  If set to `None`, a maximum matching is computed using the other parameters.

- `matching_covered_check` – (default: `True`); Before checking if G is a brick, we shall ensure that G is matching covered.

  If set to `False`, this check is skipped.

- `algorithm` – string (default: `'Micali-Vazirani'`)
  - `'Edmonds'` selects Edmonds algorithm as implemented in NetworkX
  - `'LP'` uses a Linear Program formulation of the matching problem
  - `'Micali-Vazirani'` uses Micali-Vazirani algorithm [20] (Only for unweighted maximum matching)

- `conp_certificate` – boolean (default: `True`); when set to `True` outputs a nontrivial tight cut of G, if G is not a brick.

OUTPUT:

A boolean

ALGORITHM:

Algorithm 11 : Check if G is a brick [18]

if G is bipartite **then** ▷ $\mathcal{O}(|E|+|V|)$
    return 'G is bipartite' ▷ $\mathcal{O}(1)$
end if
if `perfect_matching` is `None` **then** ▷ $\mathcal{O}(1)$
    Let M be a maximum matching of G ▷ $\mathcal{O}(|E| \cdot \sqrt{|V|})$
end if
if M is not a perfect matching **then** ▷ $\mathcal{O}(1)$
    return 'G is not matching covered.' ▷ $\mathcal{O}(1)$
end if
if `matching_covered_check` is `True` **then** ▷ $\mathcal{O}(1)$
    if G is not matching covered **then** ▷ $\mathcal{O}(|E| \cdot |V|)$
        return 'G is not matching covered.' ▷ $\mathcal{O}(1)$
    end if
end if
Check if G is bicritical thru `G.is_bicritical()` ▷ $\mathcal{O}(|E| \cdot |V|)$
if G is not bicritical **then**
    if `conp_certificate` is `True` **then**
        Obtain $\{u, v\}$ as an output of the above function call such that $G - u - v$ is not matchable.
        $B \leftarrow$ a maximal (nontrivial) barrier of G containing both u and v ▷ $\mathcal{O}(|E|)$; obtained using `maximal_barrier_in_matching_covered_graph()`
        $X \leftarrow$ A nontrivial odd component of $G - B$.
        return `False`, a nontrivial barrier cut $\partial(X)$ of G
    end if
    return `False`
end if
Check if G is three vertex connected ▷ $\mathcal{O}(|E|+|V|)$; see `tri_connectivity()`
if G is not three vertex connected **then**
    if `conp_certificate` is `True` **then**
        Let $\{u, v\}$ be a two vertex cut of G
        Let $\mathcal{E}$ denote the set of even components of $G - u - v$.
        $X \leftarrow$ any arbitrary even component in $\mathcal{E}$
        $X \leftarrow X + u$
        return `False`, a (nontrivial) 2-separation cut $\partial(X)$ of G
    end if
    return `False`

```
end if
return True
```

TIME COMPLEXITY: $\mathcal{O}(|E| \cdot |V|)$.

3.4.2 is_brace()

We state the following theorem (without proof) from [18], which shall be useful in developing an algorithm to check whether the given bipartite matching covered graph is a brace or not.

Characterization of Braces [18]

Theorem 3.4.2. *Let G be a connected bipartite graph of order six or more and let M be a perfect matching of G. It holds that G is a brace if and only if $G - u - v$ is matching covered, for every edge uv of M.*

The following method shall either attest the given bipartite matching covered graph is a brace or shall output a nontrivial tight cut of it.

```
is_brace(perfect_matching=None, algorithm='Micali-Vazirani',
    matching_covered_check=True, conp_certificate=False)
```
Checks whether the matching covered bipartite graph G is a brace.

INPUT:

- `perfect_matching` – (default: `None`); a perfect matching of the graph. It can be given using any valid input format of Graph.

  If set to `None`, a maximum matching is computed using the other parameters.

- `matching_covered_check` – (default: `True`); Before checking if G is a brace, we shall ensure that G is matching covered.

  If set to `False`, this check is skipped.

- `algorithm` – string (default: `'Micali-Vazirani'`)
  - `'Edmonds'` selects Edmonds algorithm as implemented in NetworkX
  - `'LP'` uses a Linear Program formulation of the matching problem
  - `'Micali-Vazirani'` uses Micali-Vazirani algorithm [20] (Only for unweighted maximum matching)

- `conp_certificate` – boolean (default: `True`); when set to `True` outputs a nontrivial tight cut of G, if G is not a brace.

OUTPUT:

A boolean

ALGORITHM:

Algorithm 12 : Check if G is a brace [18]

if G is not bipartite **then** ▷ $\mathcal{O}(|E| + |V|)$
    return 'G is not bipartite' ▷ $\mathcal{O}(1)$
end if
Let A and B be the color class of G.
if `perfect_matching` is `None` **then** ▷ $\mathcal{O}(1)$
    Let M be a maximum matching of G ▷ $\mathcal{O}(|E| \cdot \sqrt{|V|})$
end if
if M is not a perfect matching **then** ▷ $\mathcal{O}(1)$
    return 'G is not matching covered.' ▷ $\mathcal{O}(1)$
end if
if `matching_covered_check` is `True` **then** ▷ $\mathcal{O}(1)$
    if G is not matching covered **then** ▷ $\mathcal{O}(|E| \cdot |V|)$
        return 'G is not matching covered.' ▷ $\mathcal{O}(1)$
    end if
end if
for each edge uv in M **do** ▷ $\mathcal{O}(|V|)$; let $u \in A$ and $v \in B$
    Check if $G - u - v$ is matching covered using `G.is_matching_covered()` ▷ $\mathcal{O}(|E| + |V|)$
    if $G - u - v$ is not matching covered **then**
        if `co_np` certificate is `True` **then**
            Construct the digraph D of $G - u - v$ as defined in Theorem 3.1.7 ▷ $\mathcal{O}(|E| + |V|)$; clearly, D is not strongly connected
            The linear time search determines a directed cut C in D. ▷ $\mathcal{O}(|E| + |V|)$
            By definition of D, it follows that $C \subseteq E(G - u - v) - M$.
            Thus, C is a cut of $G - u - v$ which has a shore X such that every edge of C is incident with a vertex in $X \cap B$.
            $\partial(X + v)$ is a nontrivial barrier cut of G.
            return `False`, a (nontrivial) tight cut $\partial(X + v)$ of G
        end if
        return `False`
    end if
end for

```
30: return True
```

TIME COMPLEXITY: $\mathcal{O}((|E| + |V|) \cdot |V|)$ or effectively $\mathcal{O}(|E| \cdot |V|)$.

3.5 Notable Families of Bricks and Braces

There are several notable families of bricks and braces which play important roles in the theory of matching covered graphs [18]. We introduce some of them in this section. These families play significant roles in the works of McCuaig [19] and Norine and Thomas [21], which are described in Section Strictly 'thin edges in bricks and braces'.

CircularLadderGraph()

`static CircularLadderGraph(n)`

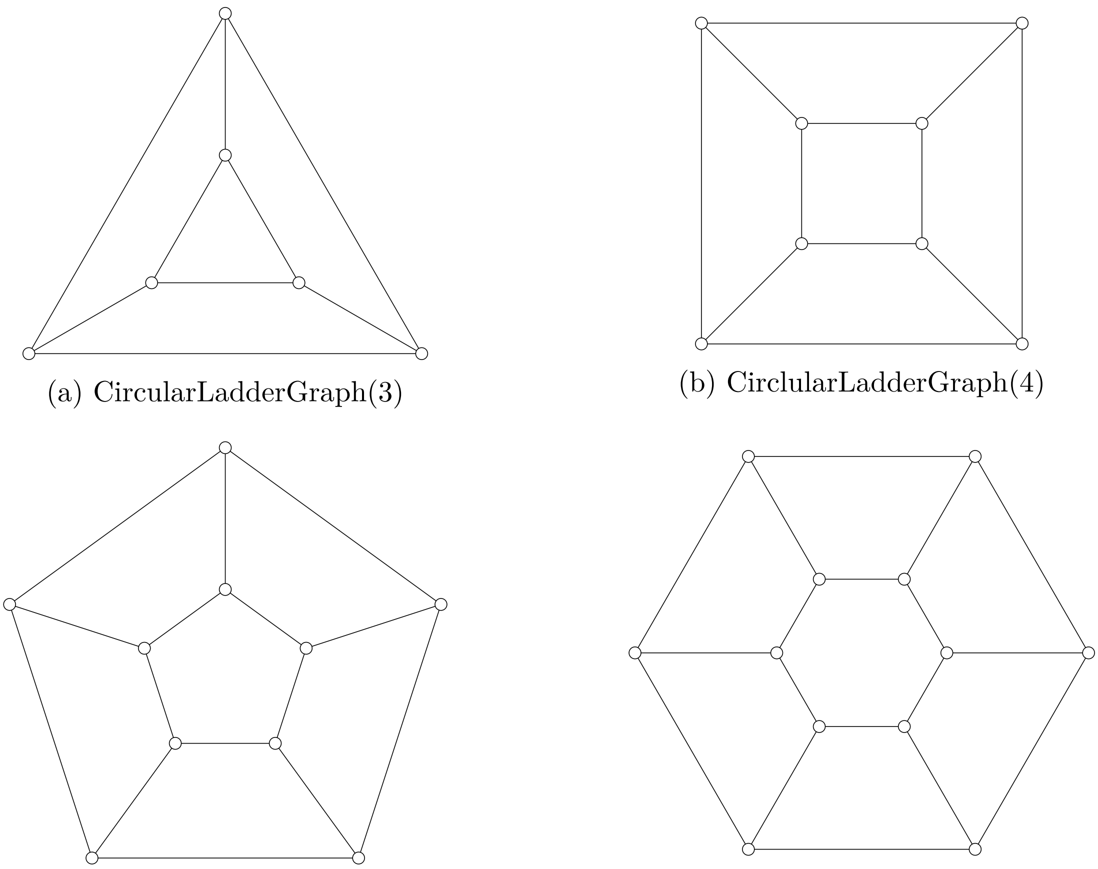
Returns the Circular ladder graph (with $2 \times n$ nodes).

The *Circular ladder graph*, aka *Prism graph* $\mathbb{P}_{2n}$, for $n \geqslant 3$, is the graph obtained from two disjoint cycles

$$u_1 u_2 u_3 \ldots u_n u_1 \text{ and } v_1 v_2 \ldots v_n v_1$$

of length n by the addition of the n edges $u_i v_i$, $i = 1, 2, \ldots, n$.

The static method `CircularLadderGraph`, listed in the Section Existing Functions in SageMath, generates a circular ladder graph on $2 \times n$ vertices for an input n. Followingly, we represent some members of this family.



(a) CircularLadderGraph(3)

(b) CirclularLadderGraph(4)

(c) CircularLadderGraph(5)

(d) CirclularLadderGraph(6)

Figure 3.2: The family of circular ladder graphs

3.5.1 MöbiusLadderGraph()

```
static MöbiusLadderGraph(n)
```

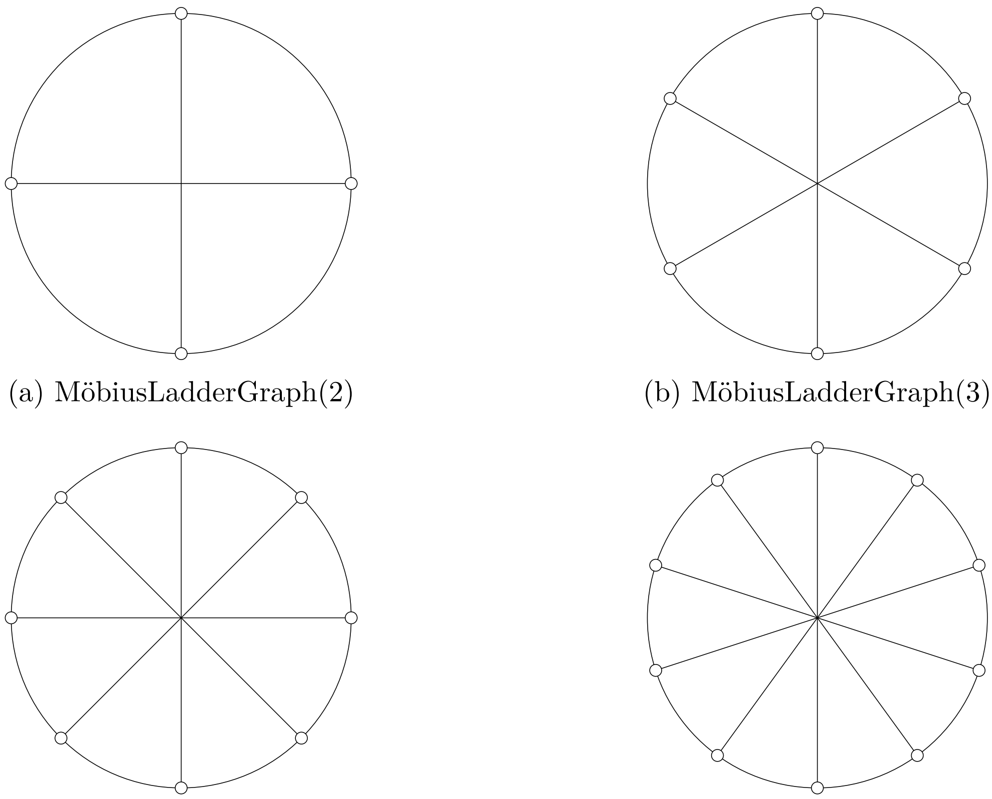
Returns the Möbius ladder graph (with $2 \times n$ nodes).

The *Möbius ladder graph* $\mathbb{M}_{2n}$, for $n \geqslant 2$, is the graph obtained from a cycle

$$v_1 v_2 \dots v_{2n} v_1$$

of length $2n$ by the addition of the n chords $v_i v_{i+n}$, for $1 \leqslant i \leqslant n$, joining antipodal pairs of vertices of the cycle.

Followingly, we represent some members of this family.



(a) MöbiusLadderGraph(2) (b) MöbiusLadderGraph(3)

(c) MöbiusLadderGraph(4) (d) MöbiusLadderGraph(5)

Figure 3.3: The family of möbius ladder graphs

WheelGraph()

```
static WheelGraph(n)
```

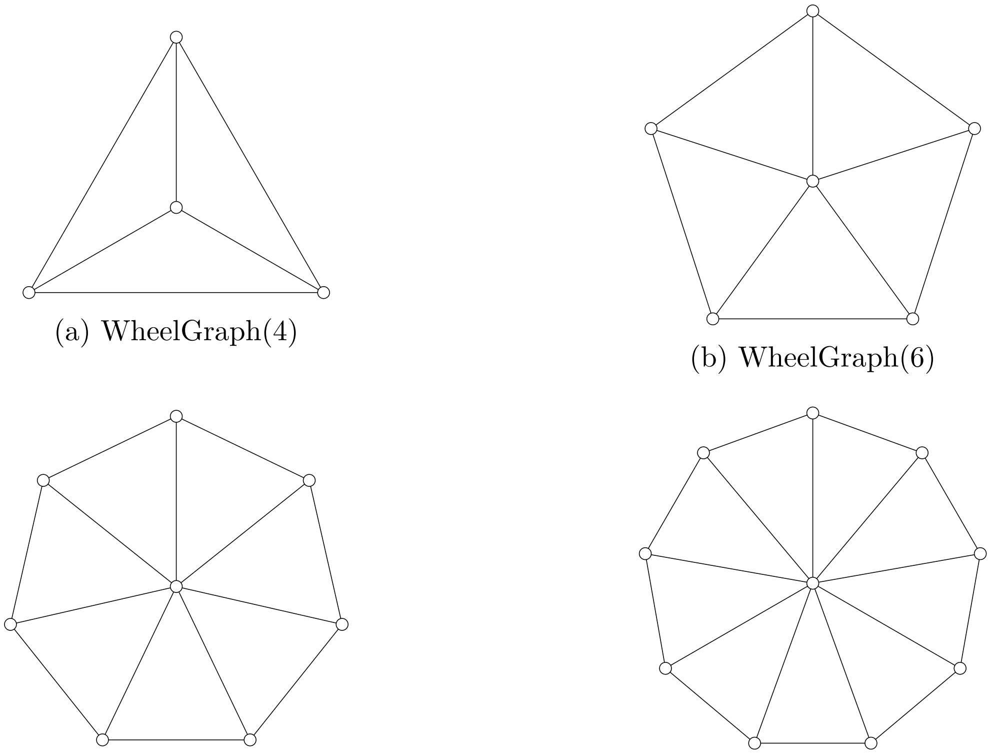
Returns the Wheel graph (with n nodes).

The *Wheel graph* $\mathbb{W}_n$, for $n \geqslant 4$, is the graph obtained from a cycle

$$v_1 v_2 \ldots v_{n-1} v_1$$

of length $n-1$, called the *rim* of $\mathbb{W}_n$, by the addition a universal vertex, called *hub* h. Note that wheels on an even number of vertices, aka W_{2k} for some $k \geqslant 2$, are matching covered — in fact, bricks.

The static method `WheelGraph`, listed in the Section Existing Functions in SageMath, generates a wheel graph on n vertices for an input n. Followingly, we represent some members of this family, that are matching covered — in fact, bricks.



(a) WheelGraph(4)

(b) WheelGraph(6)

(c) WheelGraph(8)

(d) WheelGraph(10)

Figure 3.4: The family of wheel graphs, that are matching covered

3.5.2 BiwheelGraph()

```
static BiwheelGraph(n)
```

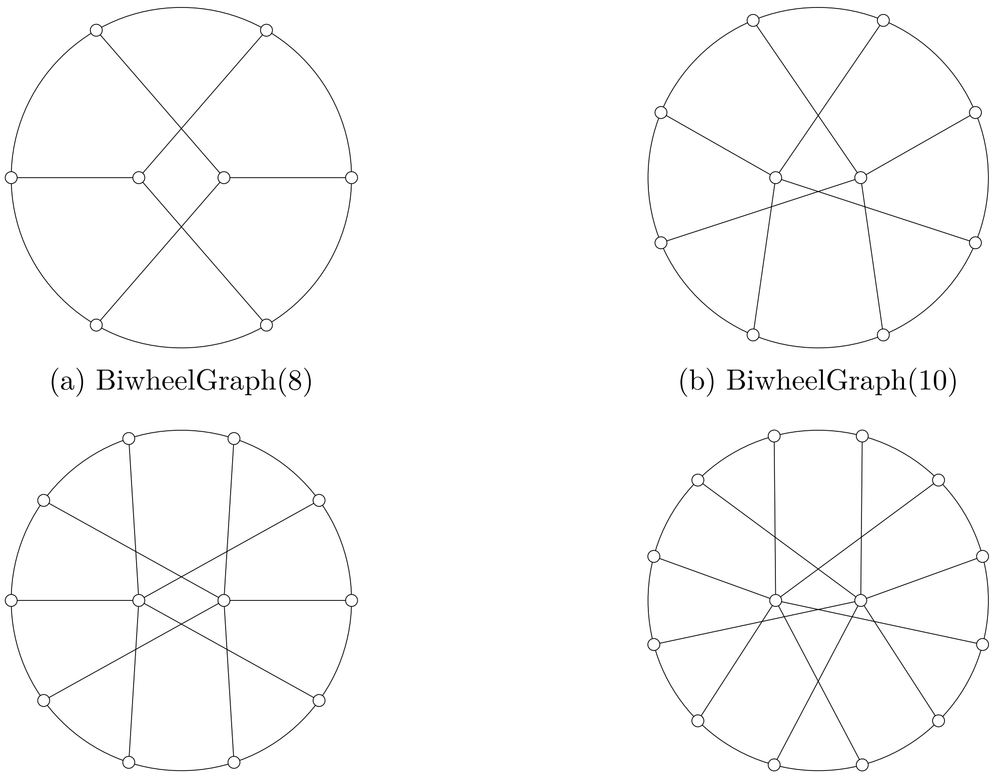
Returns the Biwheel graph with $2 \times n$ nodes.

The *Biwheel graph* $\mathbb{B}_{2n}$, for $n \geqslant 4$, is the bipartite graph obtained from a cycle

$$v_1 v_2 \dots v_{2n-2} v_1$$

of length $2n-2$, called the *rim* of $\mathbb{B}_{2n}$, by the addition of two vertices, h_1 and h_2, called the *hubs* of $\mathbb{B}_{2n}$, and by the addition of edges $h_1 v_1, h_1 v_3, \dots, h_1 v_{2n-3}$ and edges $h_2 v_2, h_2 v_4, \dots, h_2 v_{2n-2}$.

Followingly, we represent some members of this family.



(a) BiwheelGraph(8) (b) BiwheelGraph(10)

(c) BiwheelGraph(12) (d) BiwheelGraph(14)

Figure 3.5: The family of biwheel graphs

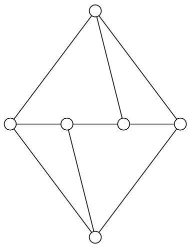
3.5.3 TruncatedBiwheelGraph()

```
static TruncatedBiwheelGraph(n)
```

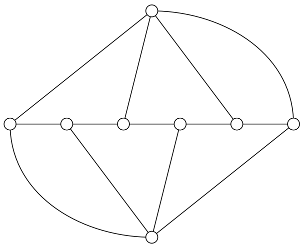
Returns the Truncated biwheel graph with $2 \times n$ nodes.

The *Truncated biwheel graph* $\mathbb{T}_{2n}$, for $n \geqslant 3$, is the graph obtained from a path $v_1v_2 \ldots v_{2n-2}$ of length $2n-3$, by the addition of two vertices, h_1 and h_2, and by the addition of edges $h_1v_1, h_1v_3, \ldots, h_1v_{2n-3}$, edges $h_2v_2, h_2v_4, \ldots, h_2v_{2n-2}$ and edges h_1v_{2n-2} and h_2v_1.

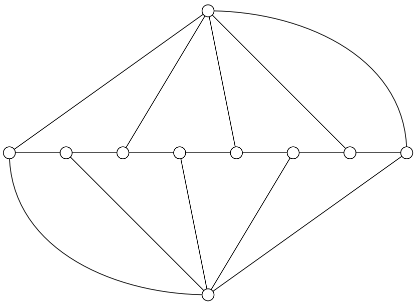
Followingly, we represent some members of this family.



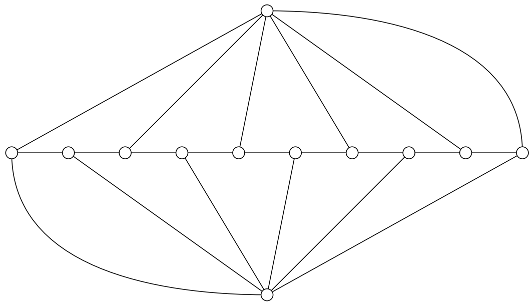
(a) TruncatedBiwheelGraph(3)



(b) TruncatedBiwheelGraph(4)



(c) TruncatedBiwheelGraph(5)



(d) TruncatedBiwheelGraph(6)

Figure 3.6: The family of truncated biwheel graphs

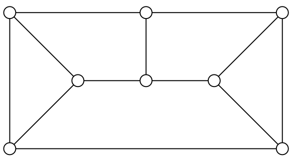
3.5.4 StaircaseGraph()

```
static StaircaseGraph(n)
```

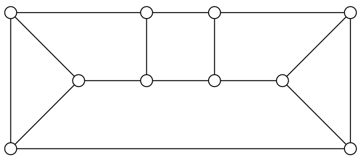
Returns the Staircase graph with $2 \times n$ nodes.

Consider the Ladder graph $\mathbb{L}_{2n-2}$ obtained from two disjoint paths $u_1u_2 \ldots u_{n-1}$ and $v_1v_2 \ldots v_{n-1}$ by adding, for $1 \leqslant i \leqslant n-1$, an edge joining u_i and v_i. For $n \geqslant 3$, the *Staircase graph* $\mathbb{S}_{2n}$ is the graph obtained from $\mathbb{L}_{2n-2}$ by adding two new vertices x and y, and then joining x to u_1 and v_1, the vertex y to u_{n-1} and v_{n-1}, and x and y to each other.

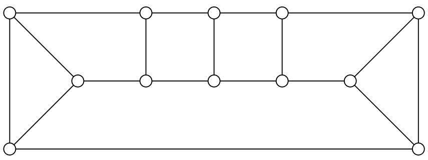
Followingly, we represent some members of this family.



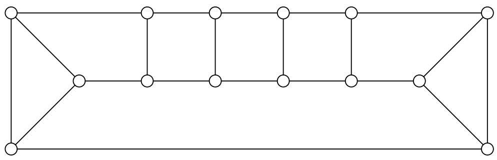
(a) StaircaseGraph(4)



(b) StaricaseGraph(5)



(c) StaircaseGraph(6)



(d) StaircaseGraph(7)

Figure 3.7: The family of staircase graphs

3.6 Dependence Relation and Removable Classes

Deletions and contractions of edges are two common inductive tools in graph theory. In 1999, through their landmark paper "Ear decompositions of matching covered graphs", Carvalho, Lucchesi and Murty [2] introduced the notation of dependency relation and removable classes in matching covered graphs, that are crucial for several significant results in the theory of matching covered graphs, for instance — computing an optimal ear decomposition of a matching covered graph (which we will see in the Section 'Ear decomposition').

3.6.1 is_removable_edge()

An edge e of a matching covered graph G is removable if the graph $G - e$ is also matching covered. For example, every edge of $K_{3,3}$ is removable, but no edge of K_4 or of $\overline{C_6}$ is. This raises a decision problem.

Decision Problem 3.6.1. *Given a matching covered graph G and an edge e; decide whether e is removable in G.*

It turns out that we may effectively use the concept of M-alternating path search, that was introduced in `is_bicritical()` to answer the above question.

```
is_removable_edge(e, perfect_matching=None, algorithm='Micali-Vazirani',
 ↪  matching_covered_check=True)
```
Checks whether the edge e is removable in the matching covered graph G.

INPUT:

- `e` – an edge of the graph G
- `perfect_matching` – (default: `None`); a perfect matching of the graph. It can be given using any valid input format of Graph.

  If set to `None`, a maximum matching is computed using the other parameters.
- `matching_covered_check` – (default: `True`); Before computing the number of Petersen bricks of G, we shall ensure that G is matching covered.

  If set to `False`, this check is skipped.
- `algorithm` – string (default: `'Micali-Vazirani'`)
  - `'Edmonds'` selects Edmonds algorithm as implemented in NetworkX
  - `'LP'` uses a Linear Program formulation of the matching problem
  - `'Micali-Vazirani'` uses Micali-Vazirani algorithm [20] (Only for unweighted maximum matching)

OUTPUT:

A boolean

ALGORITHM:

Algorithm 13 : Check if e is removable in the matching covered graph G

if `perfect_matching` is `None` **then** $\triangleright\ \mathcal{O}(1)$
    Let M be a maximum matching of G $\triangleright\ \mathcal{O}(|E| \cdot \sqrt{|V|})$
end if
if M is not a perfect matching **then** $\triangleright\ \mathcal{O}(1)$
    return 'G is not matching covered.' $\triangleright\ \mathcal{O}(1)$
end if
if `matching_covered_check` is `True` **then** $\triangleright\ \mathcal{O}(1)$
    if G is not matching covered **then** $\triangleright\ \mathcal{O}(|E| \cdot |V|)$
        return 'G is not matching covered.' $\triangleright\ \mathcal{O}(1)$
    end if
end if
check $\leftarrow$ `False`
if e is not in M **then**
    check $\leftarrow$ `True` if $G - e$ is matching covered using M $\triangleright\ \mathcal{O}(|E| + |V|)$ if $G - e$ is bipartite, otherwise $\mathcal{O}(|E| \cdot |V|)$; see `is_matching_covered()`
else
    Use M-alternating path search to find a perfect matching N of G, not containing e. $\triangleright\ \mathcal{O}(|E|)$; see `is_matching_covered()`
    check $\leftarrow$ `True` if $G - e$ is matching covered using N $\triangleright\ \mathcal{O}(|E| + |V|)$ if $G - e$ is bipartite, otherwise $\mathcal{O}(|E| \cdot |V|)$; see `is_matching_covered()`
end if
return check

TIME COMPLEXITY:

- If we check whether G is matching covered, the time complexity is $\mathcal{O}(|E| \cdot |V|)$, or otherwise
- if G is nonbipartite the time complexity is $\mathcal{O}(|E| \cdot |V|)$, or otherwise
- if a perfect matching M is given the complexity is $\mathcal{O}(|E| + |V|)$, else it is $\mathcal{O}(|E| \cdot \sqrt{|V|})$.

3.6.2 removable_edges()

Subsequently, we ask the following question.

Problem 3.6.2. *Given a matching covered graph G; find all of its removable edges.*

The following method shall list out all the removable edges of a matching covered graph G.

```
removable_edges(perfect_matching=None, algorithm='Micali-Vazirani',
    matching_covered_check=True, one_output=False)
```

Returns the set of removable edges in the matching covered graph G.

INPUT:

- `perfect_matching` – (default: `None`); a perfect matching of the graph. It can be given using any valid input format of Graph.

  If set to `None`, a maximum matching is computed using the other parameters.

- `matching_covered_check` – (default: `True`); Before computing the number of Petersen bricks of G, we shall ensure that G is matching covered.

  If set to `False`, this check is skipped.

- `algorithm` – string (default: `'Micali-Vazirani'`)
  - `'Edmonds'` selects Edmonds algorithm as implemented in NetworkX
  - `'LP'` uses a Linear Program formulation of the matching problem
  - `'Micali-Vazirani'` uses Micali-Vazirani algorithm [20] (Only for unweighted maximum matching)

- `one_output` – boolean (default: `False`); when set to `True`, the algorithm shall terminate as soon as a single removable edge is detected.

OUTPUT:

A set of edges consisting of all removable edges in G

ALGORITHM:

Algorithm 14 : Compute all removable edges in the matching covered graph G

1: **if** `perfect_matching` is `None` **then** ▷ $\mathcal{O}(1)$
2: Let M be a maximum matching of G ▷ $\mathcal{O}(|E| \cdot \sqrt{|V|})$
3: **end if**
4: **if** M is not a perfect matching **then** ▷ $\mathcal{O}(1)$
5: **return** 'G is not matching covered.' ▷ $\mathcal{O}(1)$
6: **end if**

```
if matching_covered_check is True then                                   ▷ O(1)
    if G is not matching covered then                                    ▷ O(|E| · |V|)
        return 'G is not matching covered.'                              ▷ O(1)
    end if
end if
R ← ∅                                        ▷ Initialize the set of all removable edges in G
for each edge e in E do                                                  ▷ O(|E|)
    check ← False
    check ← True if e is removable in G (check using M thru is_removable_edge()   ▷ O(|E| · |V|) if G is nonbipartite else O(|E| + |V|)
    if check then
        R ← R + e
    end if
end for
return R
```

TIME COMPLEXITY: $\mathcal{O}(|E|^2 \cdot |V|)$ if G is nonbipartite, or otherwise $\mathcal{O}((|E| + |V|) \cdot |E|)$.

3.6.3 is_removable_doubleton()

A pair $\{e, f\}$ of edges of a matching covered graph G is a removable doubleton if $G - e - f$ is matching covered, but neither $G - e$ nor $G - f$ is. For example, each of K_4 and $\overline{C_6}$ have three removable doubletons, and the bicorn $\mathbb{H}_8$ has two. Clearly, if $\{e, f\}$ is a removable doubleton in a matching covered graph G, then they are nonadjacent. This raises the following decision problem.

> **Decision Problem 3.6.3.** *Given a matching covered graph G and a pair of distinct nonadjacent edges e and f; decide whether e and f constitute a removable doubleton in G.*

We shall implement the following algorithm to effectively answer the above decision problem.

```
is_removable_doubleton(e, f, perfect_matching=None, algorithm='Micali-Vazirani',
↪ matching_covered_check=True, removable_edegs=Flase)
```

Checks if $\{e, f\}$ is a removable doubleton in the matching covered graph G.

INPUT:

- e and f – a pair of distinct nonadjacent edges of G
- `perfect_matching` – (default: `None`); a perfect matching of the graph. It can be given using any valid input format of Graph.

  If set to `None`, a maximum matching is computed using the other parameters.

- `matching_covered_check` – (default: `True`); Before computing the number of Petersen bricks of G, we shall ensure that G is matching covered.

  If set to `False`, this check is skipped.

- `algorithm` – string (default: `'Micali-Vazirani'`)
  - `'Edmonds'` selects Edmonds algorithm as implemented in NetworkX
  - `'LP'` uses a Linear Program formulation of the matching problem
  - `'Micali-Vazirani'` uses Micali-Vazirani algorithm [20] (Only for unweighted maximum matching)
- `removable_edges` – boolean (default: `False`); when set to `True` shall output the list of removable edges as well.

OUTPUT:

A boolean

ALGORITHM:

Algorithm 15 : Check if $\{e, f\}$ constitutes a removable doubleton in G

if e and f have at least one common incidence vertex **then** ▷ $\mathcal{O}(1)$
    return 'e and f should be distinct and nonadjacent.' ▷ $\mathcal{O}(1)$
end if
if `perfect_matching` is `None` **then** ▷ $\mathcal{O}(1)$
    Let M be a maximum matching of G ▷ $\mathcal{O}(|E| \cdot \sqrt{|V|})$
end if
if M is not a perfect matching **then** ▷ $\mathcal{O}(1)$
    return 'G is not matching covered.' ▷ $\mathcal{O}(1)$
end if
if `matching_covered_check` is `True` **then** ▷ $\mathcal{O}(1)$
    if G is not matching covered **then** ▷ $\mathcal{O}(|E| \cdot |V|)$
        return 'G is not matching covered.' ▷ $\mathcal{O}(1)$
    end if
end if
if G is bipartite **then** ▷ $\mathcal{O}(|E| + |V|)$

return False
end if
check ← False
if $e \in M$ and $f \notin M$ or $e \notin M$ and $f \in M$ **then**
    return check
else if $e, f \in M$ **then**
    Use M-alternating path search to find a perfect matching N of $G - e - f$. ▷ $\mathcal{O}(|E|)$; see `is_matching_covered()`
    check ← `True` if $G - e - f$ is matching covered using N ▷ $\mathcal{O}(|E| \cdot |V|)$; see `is_matching_covered()`
else
    check ← `True` if $G - e - f$ is matching covered using M ▷ $\mathcal{O}(|E| \cdot |V|)$; see `is_matching_covered()`
end if
return check

TIME COMPLEXITY: $\mathcal{O}(|E| \cdot |V|)$

3.6.4 removable_doubletons()

Subsequently, we ask the following question.

Problem 3.6.4. *Given a matching covered graph G; find all of its removable doubletons.*

The following method shall list out all the removable doubletons of a matching covered graph G.

```
removable_doubletons(perfect_matching=None, algorithm='Micali-Vazirani',
 ↪ matching_covered_check=True, one_output=False)
```

Returns the set of removable doubletons in the matching covered graph G.

INPUT:

- `perfect_matching` – (default: `None`); a perfect matching of the graph. It can be given using any valid input format of Graph.

  If set to `None`, a maximum matching is computed using the other parameters.

- `matching_covered_check` – (default: `True`); Before computing the number of Petersen bricks of G, we shall ensure that G is matching covered.

  If set to `False`, this check is skipped.

- `algorithm` – string (default: `'Micali-Vazirani'`)
  - `'Edmonds'` selects Edmonds algorithm as implemented in NetworkX
  - `'LP'` uses a Linear Program formulation of the matching problem
  - `'Micali-Vazirani'` uses Micali-Vazirani algorithm [20] (Only for unweighted maximum matching)
- `one_output` – boolean (default: `False`); when set to `True`, the algorithm shall terminate as soon as a single removable doubleton is detected.

OUTPUT:

A set of pair of edges consisting of all removable doubletons in G

ALGORITHM:

Algorithm 16 : Compute all removable doubletons in the matching covered graph G

```
if perfect_matching is None then                                  ▷ O(1)
    Let M be a maximum matching of G                              ▷ O(|E| · √|V|)
end if
if M is not a perfect matching then                               ▷ O(1)
    return 'G is not matching covered.'                           ▷ O(1)
end if
if matching_covered_check is True then                            ▷ O(1)
    if G is not matching covered then                             ▷ O(|E| · |V|)
        return 'G is not matching covered.'                       ▷ O(1)
    end if
end if
if G is bipartite then                                            ▷ O(|E| + |V|)
    return ∅
end if
R ←the set of removable edges obtained using removable_edges()
E' ← E − R
T ← ∅                         ▷ Initialize the set of all removable doubletons in G
for each pair of distinct and nonadjacent edges e, f in E' do     ▷ O(|E|)
    check ← False
    check ← True if {e, f} constitute a removable doubleton in G (check using M
thru is_removable_doubleton()     ▷ O(|E| · |V|) if G is nonbipartite else
O(|E| + |V|)
    if check then
        T ← T + {e, f}
```

$E' \leftarrow E' - e - f$

end if

end for

return T

TIME COMPLEXITY: $\mathcal{O}(|E|^2 \cdot |V|)$

3.7 Ear Decomposition

In this section, we will discuss about ear decomposition

3.7.1 matching_covered_ear_decomposition()

The ear decomposition procedure has played a significant role in the theory of matching covered graph, as provides us with one of the excellent induction tools to investigate the properties of matching covered graphs. Here, we state the ear decomposition theorem for matching covered graph [18]. The reader may refer to the book by Lucchesi and Murty to delve deeper into the concept.

Ear decomposition of matching covered graph [18]

Theorem 3.7.1. *Given any matching covered graph G there exists a sequence*

$$\mathcal{G} := (G_1 = G \supset G_2 \cdots \supset G_r = K_2)$$

of conformal matching covered subgraphs of G such that, for $1 \leqslant i \leqslant r-1$,

$$G_{i+1} = G_i - R_i, \text{ where } R_i \text{ is a removable ear of } G_i.$$

Reversing the order of the sequence we obtain the more traditional definition of an ear decomposition of a matching covered graph G as a sequence

$$G := (G_1 = K_2 \subset G_2 \cdots \subset G_r = G)$$

of matching covered conformal subgraphs of G such that, for $2 \leqslant i \leqslant r$,

$$G_i = G_{i-1} + R_i, \text{ where } R_i \text{ is a removable ear of } G_i.$$

The following method adopts an algorithm by Carvalho and Cheriyan [6] to implement the ear decomposition of a matching covered graph.

```
matching_covered_ear_decomposition(perfect_matching=None,
 ↪ algorithm='Micali-Vazirani', matching_covered_check=True)
```

Returns an ear decomposition of the matching covered graph G alongwith its canonical partition.

INPUT:

- `perfect_matching` – (default: `None`); a perfect matching of the graph. It can be given using any valid input format of Graph.

  If set to `None`, a maximum matching is computed using the other parameters.

- `matching_covered_check` – (default: `True`); Before computing the ear decomposition G, we shall ensure that G is matching covered.

  If set to `False`, this check is skipped.

- `algorithm` – string (default: `'Micali-Vazirani'`)
  - `'Edmonds'` selects Edmonds algorithm as implemented in NetworkX
  - `'LP'` uses a Linear Program formulation of the matching problem
  - `'Micali-Vazirani'` uses Micali-Vazirani algorithm [20] (Only for unweighted maximum matching)

OUTPUT:

A set of graphs, a set of ears and a partition of the vertex set.

ALGORITHM:

Algorithm 17 : Find the ear decomposition of a matching covered graph efficiently [6]

if `perfect_matching` is `None` **then** ▷ $\mathcal{O}(1)$
  Let M be a maximum matching of G ▷ $\mathcal{O}(|E| \cdot \sqrt{|V|})$
end if
if M is not a perfect matching **then** ▷ $\mathcal{O}(1)$
  return 'G is not matching covered.' ▷ $\mathcal{O}(1)$
end if
if `matching_covered_check` is `True` **then** ▷ $\mathcal{O}(1)$
  if G is not matching covered **then** ▷ $\mathcal{O}(|E| \cdot |V|)$
    return 'G is not matching covered.' ▷ $\mathcal{O}(1)$
  end if
end if

Let xy be any edge of M and let subgraph H correspond to xy, and let the canonical partition be initialized by $\mathcal{P}(H) \leftarrow \{\{x\},\{y\}\}$

while $H \neq G$ **do**

  If H is a spanning subgraph of G then let $F \leftarrow E(G) - E(H)$, else compute Y using the detailed explanation of this step in the text as mentioned in [6]; note that each edge $e_j \in F$ corresponds to a (single) ear P_j relative to H; finally, let $F_0 := F$

  repeat

    Let $H_0 := H$ and let $p_0 := |(H_0)|$; let F' be the set of edges in F that have their two ends in distinct classes of $\mathcal{P}(H)$; replace F by $F - F'$

    Sequentially examine the edges of F' and add each edge to H as a single ear; update $\mathcal{P}(H_o)$ to $\mathcal{P}(H)$

    If $p_0 = |\mathcal{P}(H)|$ and $F \neq \emptyset$ then find a double ear $\{e, f\} \subseteq F$ by using the method in Theorem 3.1 in [6]; remove e, f from F and add them to H; update $\mathcal{P}(H - \{e, f\})$ to get $\mathcal{P}(H)$

  until $F = \emptyset$

  For each edge $e_j \in F_0$ take the corresponding path P_j of G (see step (2.1) in [6]), and insert the internal nodes of P_j (if any) into appropriate classes of $\mathcal{P}(H)$ (see Proposition 2.4 in [6])

end while

TIME COMPLEXITY: $\mathcal{O}(|V| \cdot |E|)$.

3.7.2 retract()

Finding whether an edge/ a pair of edges are removable or not is somewhat easier that checking the same for an entire ear. In this context, the retract of a graph is useful. Let's discuss the retract as described in the book by Lucchesi and Murty [18].

Let v_o be a vertex of degree two in a matching covered graph G of order four or more, and let v_1 and v_2 be its two neighbours. Then $\partial(X)$, where $X := \{v_o, v_1, v_2\}$, is a tight cut of G, and the matching covered graph G/X is said to be obtained from G by the `bicontraction` of vertex v_o. Motivated by the above considerations, we now define the notion of the retract of a matching covered graph.

The `retract` of a matching covered graph G, denoted by $\hat{G}$, is a matching covered graph covered graph obtained from G by the following recursive procedure: If G has order two, or if G has no vertices of degree two, then $\hat{G} := G$; otherwise, $\hat{G} := \hat{G}_v$ where G_v is the graph obtained by bicontracting a vertex v of degree two in G. Clearly, if G has more than one vertex of degree two, this recursive procedure is not uniquely determined. However, it can be shown that, up to isomorphism, the retract $\hat{G}$ does not depend on the order in which the bicontractions are performed. (An inductive proof of this result first appeared in []).Since bicontraction preserves the property of matching covered, we conclude that $\hat{G}$ is also matching covered.

The following method computes the retract of a matching covered graph G in linear time.

```
retract(perfect_matching=None, algorithm='Micali-Vazirani',
 ↪  matching_covered_check=True)
```

Returns the retract of the matching covered graph G.

INPUT:

- `perfect_matching` – (default: `None`); a perfect matching of the graph. It can be given using any valid input format of Graph.

  If set to `None`, a maximum matching is computed using the other parameters.

- `matching_covered_check` – (default: `True`); Before computing the retract of G, we shall ensure that G is matching covered.

  If set to `False`, this check is skipped.

- `algorithm` – string (default: `'Micali-Vazirani'`)
  - `'Edmonds'` selects Edmonds algorithm as implemented in NetworkX
  - `'LP'` uses a Linear Program formulation of the matching problem
  - `'Micali-Vazirani'` uses Micali-Vazirani algorithm [20] (Only for unweighted maximum matching)

OUTPUT:

A graph.

ALGORITHM:

Algorithm 18 : Compute the retract of G

1: **if** G is not matching covered **then**
2: **return** 'This function is defined only for matching covered graphs.'
3: **end if**
4: **if** G has two vertices or $\delta(G) \geqslant 3$ **then**
5: **return** G
6: **end if**
7: Let $\mathcal{P} \leftarrow$ the set of maximal even length ear in G
8: Let $\mathcal{Q} \leftarrow$ the set of maximal odd length ear in G

$H \leftarrow G$ ▷ Make a copy of G
for each path P in $\mathcal{P}$ **do**
  Let u and v denote the ends of P.
  $H \leftarrow H + w$, where w is the new vertex
  for each edge $ux \in \partial_H(u)$ **do** $H \leftarrow H + wx$
  end for
  for each edge $vx \in \partial_H(v)$ **do** $H \leftarrow H + vx$
  end for
  $H \leftarrow H - P - u - v$
end for
for each path Q in $\mathcal{Q}$ **do**
  Let u and v denote the ends of Q.
  $H \leftarrow H + uv - Q$
end for
return H

TIME COMPLEXITY: $\mathcal{O}(|E| + |V|)$

3.7.3 optimal_ear()

We state the following theorem from the book of Lucchesi and Murty [18].

Theorem 3.7.2. *Let G be a matching covered graph. For any ear decomposition $\mathcal{G} := (K_2 = G_1 \subset G_2 \cdots \subset G_r = G)$ of G,*

$$d(\mathcal{G}) \geqslant (b + p)(G),$$

where $d(\mathcal{G})$ refers to the number of double ear additions in this ear decomposition procedure and $(b + p)(G)$ refers to the sum of the number of bricks and the number of Petersen bricks of G.

According to the above theorem, the number of double ears in any ear decomposition of a matching covered graph G is at least $b(G)+p(G)$. This bound is always attainable, as shown by Carvalho, Lucchesi and Murty, [9], in 2002. In other words, any matching covered graph G has an ear decomposition with precisely $b+p$ double ears. Carvalho, Lucchesi and Murty refer to such an ear decomposition as an optimal ear decomposition of G.

Followingly we present an algorithm, adopted from Exercise 16.2.8 of [18] to find an optimal ear, subsequently to find an optimal ear decomposition of a matching covered graph G.

```
optimal_ear(perfect_matching=None, algorithm='Micali-Vazirani',
 ↪ matching_covered_check=True)
```

Returns an ear decomposition of the matching covered graph G.

INPUT:

- `perfect_matching` – (default: `None`); a perfect matching of the graph. It can be given using any valid input format of Graph.

  If set to `None`, a maximum matching is computed using the other parameters.

- `matching_covered_check` – (default: `True`); Before finding an optimal ear of G, we shall ensure that G is matching covered.

  If set to `False`, this check is skipped.

- `algorithm` – string (default: `'Micali-Vazirani'`)
  - `'Edmonds'` selects Edmonds algorithm as implemented in NetworkX
  - `'LP'` uses a Linear Program formulation of the matching problem
  - `'Micali-Vazirani'` uses Micali-Vazirani algorithm [20] (Only for unweighted maximum matching)

OUTPUT:

An optimal ear.

ALGORITHM:

Algorithm 19 : Find an optimal ear of a matching covered graph [18]

if G is isomorphic to K_2 **then** ▷ $\mathcal{O}(1)$
  return 'K_2 has no optimal ear.' ▷ $\mathcal{O}(1)$
end if
if G is not matching covered **then** ▷ $\mathcal{O}(|E| \cdot |V|)$
  return 'This function is defined only for matching covered graphs' ▷ $\mathcal{O}(1)$
end if
$H \leftarrow G$.retract()
if G is bipartite **then**
  $r \leftarrow$ a removable edge of H
  $p \leftarrow$ the odd lobe of G that corresponds to r in H
  return p
else
  $R \leftarrow$ the set of removable edges of H

$T \leftarrow$ the set of removable doubletons of H
if T is not $\emptyset$ **then**
Let t be a removable doubleton of H
Let p denote the pair of two odd lobes of G corresponding to the pair of edges t in H
return p ▷ A pair of removable double ear
else
Obtain $(b+p)(H)$ by performing a tight cut decomposition on H
for each edge e in R **do**
Obtain $(b+p)(H-e)$ by performing a tight cut decomposition on $H-e$ ▷ Check if e is $(b+p)-$invariant edge of H
if $(b+p)(H) == (b+p)(H-e)$ **then**
$p \leftarrow$ the odd lobe of G that corresponds to e in H
return p
end if ▷ If G has a removable edge, then G has at least one removable edge that is $(b+p)$-invariant [18]
end for
end if
end if

3.7.4 optimal_matching_covered_ear_decompostition()

This subsection presents the algorithm to obtain an optimal ear decomposition of a matching covered graph G (defined in the previous section), using the method `optimal_ear()`.

```
optimal_matching_covered_ear_decomposition(perfect_matching=None,
 ↪ algorithm='Micali-Vazirani', matching_covered_check=True)
```

Returns an optimal ear decomposition of the matching covered graph G.

INPUT:

- `perfect_matching` – (default: `None`); a perfect matching of the graph. It can be given using any valid input format of Graph.

  If set to `None`, a maximum matching is computed using the other parameters.

- `matching_covered_check` – (default: `True`); Before computing the optimal ear decomposition G, we shall ensure that G is matching covered.

  If set to `False`, this check is skipped.

- `algorithm` – string (default: ‘`Micali-Vazirani`’)

– `'Edmonds'` selects Edmonds algorithm as implemented in NetworkX

– `'LP'` uses a Linear Program formulation of the matching problem

– `'Micali-Vazirani'` uses Micali-Vazirani algorithm [20] (Only for unweighted maximum matching)

OUTPUT:

A set of graphs and a set of ears.

ALGORITHM:

Algorithm 20 : Optimal ear decomposition of a matching covered graph [9]

if G is not matching covered **then**
  return 'This function is defined only for matching covered graphs'
end if
if G is K_2 **then**
  return the sequence (G) as the ear decomposition and the empty sequence of removable ears.
else
  Recursively determine an optimal ear decomposition $\mathcal{G}$ of $G - R$ and the corresponding sequence $\mathcal{R}$ of optimal removable ears.
  Add G as the last element of the sequence $\mathcal{G}$ and R as the last element of the sequence $\mathcal{R}$.
end if

3.8 Generating Bricks and Braces

In this section, we will see a generation procedure for simple bricks and simple braces, the generalization of which were proved by Norine and Thomas [21] and McCuaig [19]. Before that let's go through some definitions that shall be helpful in the subsequent subsections.

3.8.1 is_strictly_thin_edge()

An edge e of a brick G is *thin* if the retract of $G - e$ is a brick. Analogously, an edge e in a brace G is *thin* if the retract of $G - e$ is also a brace. A thin edge e of a simple brick G is *strictly thin* if the retract of $G - e$ is a simple brick. Likewise, a thin edge of a simple brace is *strictly thin* if the retract of $G - e$ is a simple brace.

There exist infinite families of bricks and braces which do not have any strictly thin edges:

1. wheels,
2. biwheels,

3. truncated biwheels,
4. prisms, aka circular ladders,
5. möbius ladders and
6. staircases.

The reader may go through Chapters 17 and 18 of the book by Lucchesi and Murty [18] to learn more about these concepts.

The following function checks whether an edge e of a simple brick/ a simple brace is strictly thin.

```
is_strictly_thin_edge(e, simple_brick_brace_check=True)
```

Checks whether an edge e of a simple brick/ a simple brace is strictly thin.

INPUT:

- an edge e of G
- `simple_brick_brace_check` – (default: `True`); Before checking whether e is a strictly thin edge of G, we shall ensure that G is either a simple brick or a simple brace.

  If set to `False`, this check is skipped.

OUTPUT:

A boolean.

ALGORITHM:

Algorithm 21 : Check if an edge e is a strictly thin edge

if G is neither a simple brick nor a simple brace **then**
    return 'This algorithm is defined only for simple bricks/ simple braces.'
end if
if e is not a removable edge of G **then**
    return 'The edge e is not removable in G, therefore is not a strictly thin edge of G.'
end if
if G is non bipartite **then**
    if G is isomorphic to a Norine-Thomas brick **then**
        return 'G is a brick that is devoid of strictly thin edges.'
    end if
else

```
    if G is isomorphic to a McCuaig brace then
        return 'G is a brace that is devoid of strictly thin edges.'
    end if
end if
H ← G.retract()
if H is a simple brick and G is a simple brick then
    return True
else if H is a simple brace and G is a simple brace then
    return True
end if
return False
```

3.8.2 is_mccuaig_brace()

In 2001, in his pioneering paper entitled "Brace generation", McCuaig [19], identified the three families of braces that do not have strictly thin edges; those are:

1. biwheels,
2. prisms, aka circular ladders $\mathbb{P}_n$, where $n \equiv 0 \mod 4$, and
3. Möbius ladders $\mathbb{M}_n$, where $n \equiv 2 \mod 4$.

Lucchesi and Murty [18] referred to the union of these families as the McCuaig family of braces. No brace in this family has strictly thin edges. More significantly, the following assertion was established in [19]:

Strictly Thin Edge Theorem for Braces [19]

Theorem 3.8.1. *Every simple brace of order six or more that is not a member of the McCuaig family of braces has a strictly thin edge.*

The following function investigates whether a simple brace on at least six vertices is a McCuaig brace.

```
is_mccuaig_brace(simple_brace_check=True, family=False)
```

Checks whether a simple brace on at least six vertices is a McCuaig brace.

INPUT:

- `simple_brace_check` – boolean (default: `True`); Before checking whether G is a McCuaig brace, we shall ensure that G is a simple brace on at least six vertices.

  If set to `False`, this check is skipped.

- `family` – boolean (default: `False`); If set to `True`, outputs the name of the family of McCuaig braces that G belongs to.

OUTPUT:

A boolean.

ALGORITHM:

Algorithm 22 : Check if the brace G is a McCuaig brace [18]

if G is not a simple brace on at least six vertices **then**
    return 'This algorithm is defined only for simple braces on at least six vertices.'
end if
if G is isomorphic to a biwheel **then**
    $h_1, h_2 \leftarrow$ the two hubs of G
    return `True`, 'G is a biwheel with hubs h_1 and h_2.'
else if G is isomorphic to a möbius ladder **then**
    return `True`, 'G is a möbius ladder.'
else if G is isomorphic to a circular ladder **then**
    return `True`, 'G is a circular ladder.'
else
    return `False`, 'G is not a McCuaig brace.'
end if

3.8.3 is_norine_thomas_brick()

In 2008, in a paper entitled "Brick generation", Norine and Thomas [21] identified all the infinite families of simple bricks that are free of strictly thin edges. They are:

1. wheels,
2. truncated biwheels,
3. prisms, aka circular ladders $\mathbb{P}_n$, where $n \equiv 2 \mod 4$,
4. Möbius ladders $\mathbb{M}_n$, where $n \equiv 0 \mod 4$, and
5. staircases.

Lucchesi and Murty [18] refer to the union of these families of bricks as the Norine-Thomas family of bricks. The theorem below follows from the results established in [21]:

Strictly Thin Edge Theorem for Bricks [21]

Theorem 3.8.2. *Every simple brick which is different from the Petersen graph $\mathbb{P}$ and which is not a member of the Norine-Thomas family of bricks has a strictly thin edge.*

The following function investigates whether a simple brick is a Norine-Thomas brick.

```
is_norine_thomas_brace(simple_brick_check=True, family=False)
```

Checks whether a simple brick is a Norine Thomas brick.

INPUT:

- `simple_brick_check` – boolean (default: `True`); Before checking whether G is a Norine Thomas brick, we shall ensure that G is a simple brick.

  If set to `False`, this check is skipped.

- `family` – boolean (default: `False`); If set to `True`, outputs the name of the family of Norine Thomas brick that G belongs to.

OUTPUT:

A boolean.

ALGORITHM:

Algorithm 23 : Check if the brick G is a Norine Thomas brick

if G is not a simple brick **then**
    return 'This algorithm is defined only for simple bricks that are not isomorphic to the Petersen graph $\mathbb{P}$.'
end if
if G is isomorphic to the Petersen graph $\mathbb{P}$ **then**
    return 'This algorithm is defined only for simple bricks that are not isomorphic to the Petersen graph $\mathbb{P}$.'
end if
if G is isomorphic to a wheel **then**
    $h \leftarrow$ the hubs of G
    return `True`, 'G is a wheel with the hubs h.'
else if G is isomorphic to a truncated biwheel **then**
    return `True`, 'G is isomorphic to a truncated biwheel.'
else if G is isomorphic to a möbius ladder **then**
    return `True`, 'G is a möbius ladder.'
else if G is a circular ladder **then**
    return `True`, 'G is a circular ladder.'
else
    return `False`, 'G is not a Norine Thomas brick.'
end if

3.8.4 mccuaig_brace_decomposition()

We state the following result of McCuaig [19] as stated in [18].

Theorem 3.8.3. *Given any simple brace G of order six or more, there exists a sequence*

$$G_1, G_2, \dots, G_k$$

of simple braces such that:

1. *G_1 is either a biwheel, or a prism, or a Möbius ladder, and $G_k = G$, and*
2. *for $2 \leqslant i \leqslant k$, the graph G_i is obtained from G_{i-1} by an expansion operation.*

We left it as an exercise for the reader to go through the details of the above theorem. In the following method, we shall output a sequence of such simple braces for a given simple brace G of order six or more.

```
mccuaig_brace_decomposition(simple_brace_check=True)
```

Computes a McCuaig brace decomposition of a simple brace G on at least six vertices.

INPUT:

- `simple_brace_check` – boolean (default: `True`); Before computing the McCuaig brace decomposition of G, we shall ensure that G is a simple brace on at least six vertices.

  If set to `False`, this check is skipped.

OUTPUT:

A sequence of simple braces and a sequence of strictly thin edges

ALGORITHM:

Algorithm 24 : Generating simple braces

if G is not a simple brace of order six or more **then**
    return 'This algorithm is defined only for a simple brace of order six or more.'
end if
$\mathcal{G} \leftarrow \emptyset$
$\mathcal{R} \leftarrow \emptyset$
status $\leftarrow$ `False`
while not status **do**
    Let e be a strictly thin edge in G
    If such an edge does not exist, status $\leftarrow$ `True`
    $\mathcal{R} \leftarrow \mathcal{R} + e$

$\mathcal{G} \leftarrow \mathcal{G} + G$
end while
return $\mathcal{G}, \mathcal{R}$.

3.8.5 norine_thomas_brick_decomposition()

We state the following result of Norine and Thomas [21] as stated in [18].

Theorem 3.8.4. *Given any simple brick G, there exists a sequence*

$$G_1, G_2, \dots, G_k$$

of simple bricks such that:

1. *G_1 is either a wheel, or a truncated biwheel, or a prism, or a Möbius ladder, or a staircase, or the Petersen graph, and $G_k = G$, and*
2. *for $2 \leqslant i \leqslant k$, the graph G_i is obtained from G_{i-1} by an expansion operation.*

We left it as an exercise for the reader to go through the details of the above theorem. In the following method, we shall output a sequence of such simple bricks for a given simple brick G.

```
norine_thomas_brick_decomposition(simple_brick_check=True)
```

Computes a Norine Thomas brick decomposition of a simple brick G.

INPUT:

- `simple_brick_check` – boolean (default: `True`); Before computing the Norine Thomas brick decomposition of G, we shall ensure that G is a simple brick.

  If set to `False`, this check is skipped.

OUTPUT:

A sequence of simple bricks and a sequence of strictly thin edges

ALGORITHM:

Algorithm 25 : Generating simple bricks

if G is not a simple brick **then**
  return 'This algorithm is defined only for simple bricks.'
end if
$\mathcal{G} \leftarrow \emptyset$

$\mathcal{R} \leftarrow \emptyset$
status $\leftarrow$ `False`
while not status **do**
    Let e be a strictly thin edge in G
    If such an edge does not exist, status $\leftarrow$ `True`
    $\mathcal{R} \leftarrow \mathcal{R} + e$
    $\mathcal{G} \leftarrow \mathcal{G} + G$
end while
return $\mathcal{G}, \mathcal{R}$.

Chapter 4

Conclusion

In this work, most of the major milestone theorems concerning the theory of matching covered graph are implemented through cython using SageMath. Starting with the fundamentals the work covers the efficient implementational aspect of theories pertaining to the canonical partition, tight cut decomposition, notable families of bricks and braces, dependency relation and removable classes, ear decomposition ,and generation of bricks and braces.

4.1 Future work

The theory of matching in graph theory has numerous applications, with one particularly fascinating area being the theory of Pfaffian orientations. Within this realm, several intriguing results have been discovered, as highlighted in [18]:

1. Identifying the characteristic orientation of a graph G.

2. Validating an orientation.

3. Efficiently recognizing Pfaffian bipartite graphs.

4. Efficiently recognizing Pfaffian near-bipartite graphs, and so on.

In matching theory, there are the concepts of matching minors and conformal minors (analogous to the minor and topological minor in the study of planarity in graph theory) that lead to the primary ear decomposition of a matching covered graph; there are the fascinating notations of geometric objects known as perfect matching polytopes, that play an important role in results related to solid bricks and many more. Also, on the other side of the spectrum of matching theory, algorithms concerning matching under preferences pop up. In the future, I am looking forward to implementing the algorithms related to these topics in SageMath.

Bibliography

[1] Adrian Bondy and U.S.R. Murty. *Graph Theory*. Graduate Texts in Mathematics. Springer London, 2008.

[2] M. H. Carvalho, C. L. Lucchesi, and U. S. R. Murty. Ear decompositions of matching covered graphs. *Combinatorica*, 19:151–174, 1999.

[3] M. H. Carvalho, C. L. Lucchesi, and U. S. R. Murty. On a conjecture of lovász concerning bricks: Ii. bricks of finite characteristic. *Journal of Combinatorial Theory, Series B*, 85:137–180, 2002.

[4] M. H. Carvalho, C. L. Lucchesi, and U. S. R. Murty. On tight cuts in matching covered graphs. *Journal of Combinatorics*, 9:163–184, 2018.

[5] Marcelo H. Carvalho, Cláudio L. Lucchesi, and U.S.R. Murty. Optimal ear decompositions of matching covered graphs and bases for the matching lattice. *Journal of Combinatorial Theory, Series B*, 84(1):1–16, 2002.

[6] M.H. Carvalho and J. Cheriyan. An o(v e) algorithm for ear decompositions of matching-covered graphs. *ACM Trans. Algorithms*, 1(1):324–337, 2005.

[7] G. Chen, X. Feng, F. Lu, C. L. Lucchesi, and L. Zhang. Laminar tight cuts in matching covered graphs. *Journal of Combinatorial Theory, Series B*, 150:177–194, 2021.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, London, England, 2009.

[9] M. H. de Carvalho, C. L. Lucchesi, and U. S. R. Murty. Optimal ear decompositions of matching covered graphs and bases for the matching lattice. *Journal of Combinatorial Theory, Series B*, 76(1):1–14, 2002.

[10] J. Edmonds, L. Lovász, and W. R. Pulleyblank. Brick decompositions and the matching rank of graphs. *Combinatorica*, 2:247–274, 1982.

[11] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[12] Chris Godsil. *Algebraic Combinatorics.* Chapman and Hall/CRC, Boca Raton, FL, 1993.

[13] J. E. Hopcroft and R. M. Karp. An $n^{\frac{5}{2}}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2:225–231, 1973.

[14] Dingjun Lou and Dongning Rao. Characterizing factor critical graphs and an algorithm. *Mathematical Sciences*, 2004.

[15] Dingjun Lou and Ning Zhong. A highly efficient algorithm to determine bicritical graphs. In *Computing and Combinatorics*, pages 349–356. Springer, 2001.

[16] L. Lovász and M. D. Plummer. *Matching Theory.* Number 29 in Annals of Discrete Mathematics. Elsevier Science, 1986.

[17] László Lovász. Matching structure and the matching lattice. *Journal of Combinatorial Theory, Series B*, 43(2):187–222, 1987.

[18] Cláudio L. Lucchesi and U.S.R. Murty. *Perfect Matchings: A Theory of Matching Covered Graphs.* Algorithms and Computation in Mathematics. Springer Cham, 1 edition, 25 March 2024.

[19] W. McCuaig. Brace generation. *Journal of Graph Theory*, 38:124–169, 2001.

[20] Silvio Micali and Vijay V. Vazirani. An $\mathcal{O}(\sqrt{|V|} \cdot |E|)$ algorithm for finding maximum matching in general graphs. In *Proc. 21st IEEE FOCS*, pages 17–27, 1980.

[21] S. Norine and R. Thomas. Generating bricks. *Journal of Combinatorial Theory, Series B*, 97:769–817, 2007.

[22] W. T. Tutte. The factorization of linear graphs. *Journal of the London Mathematical Society*, 22(2):107–111, 1947.