On the Implementation of the $\mathcal{O}(|E| \cdot \sqrt{|V|})$ Micali-Vazirani Algorithm for Maximum Cardinality Matching in Undirected Graphs

Proposer/ Contributor: Janmenjaya Panda

Organization: SageMath

Contents

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. At the heart of all this research lies the bottleneck of finding a maximum cardinality matching in undirected graphs. Edmonds' Blossom algorithm [2], introduced in 1965, was groundbreaking in providing the first polynomial-time solution by effectively handling odd-length cycles (blossoms) in $\mathcal{O}(|E| \cdot |V|^2)$ time. There are algorithms, with the Linear Programming backbone, that exploit the properties of the matching polytope for computing a maximum cardinality matching. For nonbipartite graphs, it either suffers from nontight Linear Programming relaxation, leading to fractional solutions or otherwise the inclusion of exponential numbers of the odd-set constraint.

Overtime, there have been several attempts in order to effectively compute the maximum cardinality matching in general undirected graphs. In 1980, Silvio Micali and Vijay V. Vazirani presented an algorithm [6] that calculates a maximum cardinality matching in $\mathcal{O}(|E| \cdot \sqrt{V})$ time. It is worth noting that the Micali-Vazirani algorithm, by far, offers the best theoretical runtime known for the concerned problem. Despite this, there are no publicly available sagemath implementations. It is for this reason that researchers in this area are at a loss, and are required to implement this by themselves. Currently, in SageMath, for general undirected graphs, in the method "Graph.matching()", it computes a maximum cardinality matching in $\mathcal{O}(|E| \cdot |V|^2)$ either through Edmonds' algorithm or by using Linear Programming formulation. Ergo, we propose to implement the $\mathcal{O}(|E| \cdot \sqrt{|V|})$ Micali-Vazirani algorithm for maximum cardinality matching in undirected graphs in SageMath, and to make all of that available freely to students, educators as well as researchers all across the world.

This implementation draws upon the work of Prof. Vazirani [7] and the study by Michael Huang and Clifford Stein [5]. In addition, a presentation [8] delivered by Prof. Vazirani at the Simons Institute — *A Theory of Alternating Paths and Blossoms, from the Perspective of Minimum Length* — was consulted to obtain a more comprehensive understanding of the algorithm.

1 Introduction

All the graphs considered in this proposal are undirected and unweighted. (For directed or weighted, or possibly both, graphs, we may consider the underlying undirected unweighted graph for this problem). 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.

2 Existing Methods in SageMath

For computing a maximum cardinality matching in general undirected graphs, as of September 7, 2025, in sagemath, in the method "Graph.matching()", it either uses the Edmonds' algorithm through the networkX implementation, or the Linear Programming formulation. The other methods, such as "Graph.has_perfect_matching()", internally use `matching()` for computing the a maximum cardinality matching. It's worth mentioning that for specific cases, for instance bipartite graph, the method "BipartiteGraph.matching()", implements $\mathcal{O}(|E| \cdot \sqrt{|V|})$ Edmonds', Hopcroft-Karp's, Linear Programming formulation or using Eppstein's algorithm. But, our focus shall be on general undirected graphs.

```
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:

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

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 Edmond's 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
```

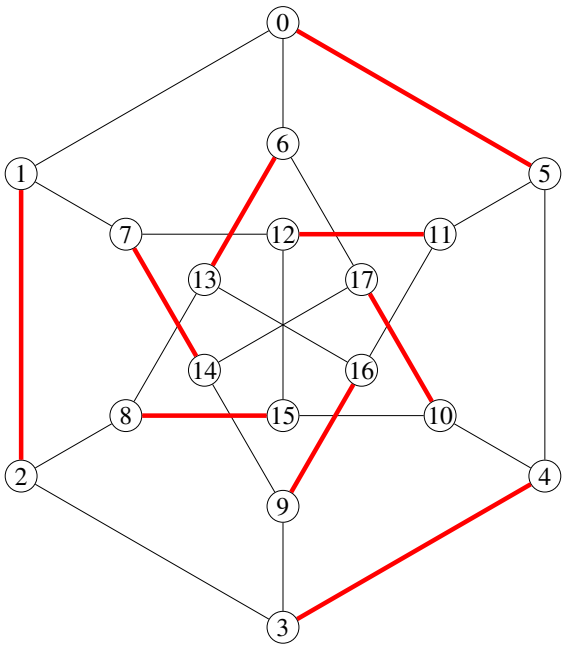

Figure 1: Pappus Graph and its maximum (perfect) matching
— shown in bold red

2.1 Edmonds' algorithm

In the following section, the internal implementation of Edmonds' algorithm for general undirected (weighted) graphs is described, which is designed to compute a maximum weighted integer matching — and, consequently, a maximum cardinality integer matching.

```
def matching(G, value_only=False, algorithm='Edmonds',
             use_edge_labels=False, solver=None, verbose=0,
             *, integrality_tolerance=1e-3):
    r"""
    Return a maximum weighted matching of the graph represented by the list
    of its edges.
    """
    from sage.rings.real_mpfr import RR

    def weight(x):
        if x in RR:
            return x
        else:
            return 1

    W = {}
    L = {}
    for u, v, l in G.edge_iterator():
        if u is v:
            continue
        fuv = frozenset((u, v))
        if fuv not in L or (use_edge_labels and W[fuv] < weight(l)):
            L[fuv] = l
```

```
            if use_edge_labels:
                W[fuv] = weight(l)

    if algorithm == "Edmonds":
        import networkx
        g = networkx.Graph()
        if use_edge_labels:
            for (u, v), w in W.items():
                g.add_edge(u, v, weight=w)
        else:
            for u, v in L:
                g.add_edge(u, v)
        d = networkx.max_weight_matching(g)
        if value_only:
            if use_edge_labels:
                return sum(W[frozenset(e)] for e in d)
            return Integer(len(d))

        from sage.graphs.graph import Graph
        return EdgesView(Graph([(u, v, L[frozenset((u, v))]) for u, v in d],
                               format='list_of_edges'))

    raise ValueError('algorithm must be set to either "Edmonds" or "LP"')
```

2.2 Linear Programming Formulation

Below, the internal implementation for computing a maximum weight fractional matching in general undirected (weighted) graph, is presented.

```
def matching(G, value_only=False, algorithm='Edmonds',
             use_edge_labels=False, solver=None, verbose=0,
             *, integrality_tolerance=1e-3):
    r"""
    Return a maximum weighted matching of the graph represented by the list
    of its edges.
    """
    from sage.rings.real_mpfr import RR

    def weight(x):
        if x in RR:
            return x
        else:
            return 1

    W = {}
    L = {}
    for u, v, l in G.edge_iterator():
        if u is v:
            continue
        fuv = frozenset((u, v))
        if fuv not in L or (use_edge_labels and W[fuv] < weight(l)):
            L[fuv] = l
            if use_edge_labels:
```

```
                W[fuv] = weight(l)

    if algorithm == "LP":
        g = G
        from sage.numerical.mip import MixedIntegerLinearProgram
        # returns the weight of an edge considering it may not be
        # weighted ...
        p = MixedIntegerLinearProgram(maximization=True, solver=solver)
        b = p.new_variable(binary=True)
        if use_edge_labels:
            p.set_objective(p.sum(w * b[fe] for fe, w in W.items()))
        else:
            p.set_objective(p.sum(b[fe] for fe in L))
        # for any vertex v, there is at most one edge incident to v in
        # the maximum matching
        for v in g:
            p.add_constraint(p.sum(b[frozenset(e)] for e in G.edge_iterator(vertices=[v],
             ↪  labels=False)
                                   if e[0] != e[1]), max=1)

        p.solve(log=verbose)
        b = p.get_values(b, convert=bool, tolerance=integrality_tolerance)
        if value_only:
            if use_edge_labels:
                return sum(w for fe, w in W.items() if b[fe])
            return Integer(sum(1 for fe in L if b[fe]))

        from sage.graphs.graph import Graph
        return EdgesView(Graph([(u, v, L[frozenset((u, v))])
                                for u, v in L if b[frozenset((u, v))]],
                                format='list_of_edges'))

    raise ValueError('algorithm must be set to either "Edmonds" or "LP"')
```

3 The Proposed New Method

Followingly, we propose to implement the $\mathcal{O}(|E| \cdot \sqrt{|V|})$ Micali-Vazirani algorithm [6] for maximum cardinality matching in a general undirected graph. A proof of the algorithm has been provided by Vijay V. Vazirani [7] in 2020. We adopt the version of the algorithm as described by Michael Huang1 and Clifford Stein in 2017 [5].

3.1 Notations

Firstly, we provide concepts specific to the the concerned algorithm.

1. **Matching**: A *matching* for an undirected graph $G := (V, E)$ is a set of edges M such that no two edges meet at a vertex.
2. **Maximum cardinality matching**: A matching M is called *a maximum cardinality matching* if $|M| \geqslant |N|$ for each matching N of G, that is the number of edges in the set M is maximized.
3. **M-matched/ M-unmatched vertex/ edge**: For a matching M of G, a vertex is *M-matched* if there

exists an edge e in M such that e is in $\partial_G(v)$ (that is — e is incident at v). If no such edge exists, then v is $M\text{-}exposed$.

Similarly, for an edge e, if e belongs to M, then e is $M\text{-}matched$, otherwise, it is $M\text{-}exposed$.

4. **M-alternating path**: A path is $M\text{-}alternating$ a simple path that alternates between edges in M and in $E - M$.
5. **M-augmenting path**: An $augmenting\ path$ is an alternating path that starts and ends with unmatched vertices.
6. **Even level and Odd level of a vertex** v: An *even level/odd level* is the length of the shortest even/ odd alternating path from an unmatched vertex to v, denoted as $even\_level(v)$ and $odd\_level(v)$, respectively.
7. **Minimum level and maximum level of a vertex** v: The $minimum\ level$ of v, denoted as $minimum\_level(v)$, is the minimum of $even\_level(v)$ and $odd\_level(v)$. Analogously, the $maximum\ level$ is the maximum of $even\_level(v)$ and $odd\_level(v)$.
8. **Inner and outer vertices**: A vertex is $outer$ if $odd\_level(v) > even\_level(v)$ and $inner$ otherwise.
9. **Tenacity**: The $tenacity\ of\ a\ vertex\ v$ is defined as,

$$tenacity(v) = even\_level(v) + odd\_level(v)$$

The $tenacity\ of\ edge\ (u, v)$ is defined as,

$$tenacity((u,v)) = \begin{cases} odd\_level(u) + odd\_level(v) + 1 & \text{for an matched edge } uv \\ even\_level(u) + even\_level(v) + 1 & \text{otherwise} \end{cases}$$

10. **Predecessor, prop, and bridge**: For any edge (u, v) such that $minimum\_level(v) == minimum\_level(u) + 1$, the vertex u is defined to be $a\ predecessor\ of\ v$. Any edge that joins a vertex and its predecessor is defined as a $prop$. If an edge is not a prop, then it is a $bridge$.
11. **Support of a bridge**: If (u, v) is a bridge with tenacity t, then its support is defined as the set $\{w \mid \text{tenacity}(w) = t$ and there exists a max_level(w) path containing $(u, v)\}$.
12. The $double\ depth\ first\ search$ $(DDFS)$ algorithm is also specifically emphasized [6], since it is an essential method for finding augmenting paths and forming petals. The DDFS finds disjoint paths to the root nodes from any pair of vertices in a level graph. In the algorithm, if such paths exist then an augmenting path is found, otherwise there is a bottleneck and we form a petal.

3.2 Description

The algorithm is a non-bipartite matching algorithm that operates in phases. Each phase finds a maximal set of vertex disjoint shortest length augmenting paths. Like the Hopcroft-Karp algorithm [4] for bipartite graphs, each phase synchronously constructs a level graph using breadth-first search from unmatched vertices to find alternating paths. Every time the level graph expands, the algorithm identifies bridges and performs double depth first searches on them to check for augmenting paths and to form petals. After performing the double depth first searches at the current level, the phase ends if a path was augmented. The algorithm terminates once we search the entire graph and do not find an augmenting path.

Algorithm 1 : Micali-Vazirani Algorithm

Input: An unweighted undirected graph G
Output: A maximum cardinality matching M of G
Set initial greedy matching for G
Reset edge labels
Set $minimum\_level(v) \leftarrow 0$ and $maximum\_level \leftarrow \infty$ for each unmatched vertex v
Set $minimum\_level(v) \leftarrow \infty$ and $maximum\_level \leftarrow \infty$ for each matched vertex v
// Line 8 to 33: One complete phase
Set $level \leftarrow 0$
// Line 10 to 20: The synchronous breadth-first search
if there exist u such that $maximum\_level(u) == level$ or $minimum\_level(u) == level$ **then**
    continue
else
    Return current matching
end if
for each u such that $maximum\_level(u) == level$ or $minimum\_level(u) == level$ **do**
    for each unscanned (u, v) with appropriate edge parity **do**
        if $minimum\_level(v) \geqslant level + 1$ **then**
            Set $minimum\_level(v) \leftarrow level + 1$
            Add u to the list of predecessors of u
            Label (u, v) as prop
        else
            Label (u, v) as bridge
            if $tenacity(u, v)! = \infty$ **then**
                Add (u, v) to the list of bridges with the same tenacity
            end if
        end if
    end for
end for
// Line 30 to 45: Using DDFS to find augmenting paths and forming petals
for each bridge of $tenacity == 2 * level + 1$ **do**
    Find support using DDFS
    if bottleneck found **then**
        Augment alternating path
        Delete the vertices in the augmented path and all vertices that are orphanned (no predcessors) as a result
    else
        for each v in the support **do**
            Set $maximum_level(v) = 2 * level + 1 - minimum_level(v)$
            if v is an inner vertex **then**
                for all incident (v, u) which are not props **do**
                    if $tenacity(u, v)! = \infty$ **then**
                        Add (u, v) to the list of bridges with the same tenacity
                    end if

```
                end for
            end if
        end for
    end if
end for
Set level ← level + 1
// Line 49: Phase termination condition
if augmentation occured then
    Go back to line 4
else
    Go to line 10
end if
// Lines 10 : Algorithm termination condition
Return the current matching
```

Since each phase ends when the maximum set of vertex disjoint minimum length alternating path is augmented, there are at most $\sqrt{|V|}$ phases. The algorithm also guarantees $\mathcal{O}(|E|)$ runtime per phase [3] which leads to the overall runtime of $\mathcal{O}(|E| \cdot \sqrt{|V|})$.

The above pseudocode is an overview of the overall algorithm. For a detailed go through, the reader may refer to the original paper [6] and the paper by Michael Huang and Clifford Stein [5].

4 About Me

4.1 Personal Details

Parameter	Value
Name	Janmenjaya Panda
Mail	janmenjaya.panda.22@gmail.com
Phone	+916370642056
Linked In	linkedin.com/in/panda-janmenjaya/
GitHub	github.com/janmenjayap/
Website	janmenjaya-panda.web.app/
Resume	Link
Location	Bangalore, India
Timezone	IST (GMT + 05: 30)

4.2 Background

This section includes some of my details concerning me, my skills and my familiarity with open-source applications and sage.

4.2.1 Who am I

I am Janmenjaya Panda, currently a Project-Trainee, working with Prof. Nishad Kothari in the Department of Algorithms and Graphs at Indian Institute of Technology Madras. I graduated from Indian Institute of Technology Madras in 2024 with a Bachelor's degree in Mechanical Engeinnering and with two minors, one in Computing and the other in Artificial Intelligence. My current interests include maths, statistics, computer vision and theoretical computer science, in particular graph theory, with a focus on matching theory. I also play some chess in my free time.

4.2.2 Technical Skills

My coursework at IITM, pertaining to the theoretical computer science included: Design and Analysis of Algorithms, Advanced Graph Algorithms, Approximation Algorithms, Linear Programming and Combinatorial Optimization, Nonlinear Optimization: Theory and algorithms. I am well acquainted with sagemath and it's overall structure, at least wrt the graph module.

4.2.3 Platform and Operating System

I have been using Ubuntu 20.04 LTS since I attended the course Introduction to Scientific Computing in my second semester. I have the local set up of sage develop built and tested on my local system several times since March 2024.

4.2.4 My Open source contributions and GSoC 2024: Sagemath

In 2024 Google Summer of Code, I proposed the idea On Decompositions, Generation Methods and related concepts in the theory of Matching Covered Graphs and with the guidance and mentorship of Prof. David Coudert, I successfully completed the fundamental implementations of the project [Link] with 12 PRs (8 Merged and 4 currently in review), 4 Issues (3 Closed and 1 currently ongoing) and currently working on it's further extension methods.

4.2.5 My familiarity with sagemath

I have been doing my research in matching covered graph for nearly two years. Even before that while I used to learn finite element method, differential geometry, differential equations, and I can never remind myself of a moment, where I have not used SageMath, let it be in the learning aspect or in the assignment/ project aspect. It has been an wonderful journey for me to get an opportunity to pay back not only just the open source software that I have used so often in my academic voyage but also in the particular domain that piques me, that is the subject of my research: matching covered graphs.

The Sage community is a great, inclusive community that welcomes all new contributors. I remember I faced several issues while building SageMath from scratch on my local machine. I mentioned this issue on the Google Group of SageMath with all the required details. The maintainers were pretty prompt and to the point in replying to my queries. It is so good to have such a supportive and helpful community. Furthermore, my mentor is one of the best mentors I could have had in my GSoC 2024 journey. For the PRs related to my GSoC work, sometimes I also used to get reviews from other reviewers besides my mentor (for instance, I had forgotten to include the name of a newly created file ‘`src/sage/graphs/matching_covered_graph.py`‘ in ‘`src/sage/graphs/meson.build`‘, due to which the Meson test cases were failing; this was pointed out by another reviewer).

It has been an amazing journey with SageMath for me as a student, me as a researcher and me as a developer.

4.3 Availibility

I will be able to allocate 30 dedicated hours per week to the project till the end of November (in case the project gets extended).

4.4 Schedule

This subsection throws light on the milestones and deliverables that we plan to achieve:

1. **Community Bonding Period (May 8 - June 1):** During the bonding time I shall raise relevant issues, fix existing bugs, help to merge pending PRs, close issues, and create milestones on the GitHub Project. Furthermore, I shall discuss with the mentor the road map, the class and file structure and finalize the plan of action.
2. **First Phase (June 2 - July 14):** Implementing the function FindPATH [6]
3. **Second Phase (July 14 - August 25):** Implementing Subroutine Blossom-Augmentation [6]
4. **Third Phase (July 14 - September 1):** Implementing the routine Search and the main method
5. **Last Phase - Tentative (September 1 - November 10):** Implementing different variants and specific cases of initialization [5]

   PS — This division may be treated as checkpoints and milestones to be marked.

5 ACKNOWLEDGEMENTS

I am absolutely grateful to my guide Prof. Nishad Kothari. Throughout last two 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 immensely grateful to Prof. David Coudert, for his invaluable guidance, timely responses, mentorship and thorough reviews in Google Summer of Code 2024, and beyond that. His support and mentorship have been crucial throughout that journey, making the experience both rewarding and insightful. I sincerely thank him for him for his mentorship throughout this project and beyond, and I have continued having his support and guidance as we move forward with implementing some remaining and incomplete aspects of that work at sagemath.

6 REFERENCES

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

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

[3] Harold N. Gabow. Set-merging for the matching algorithm of micali and vazirani, 2014.

[4] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

[5] Michael Huang and Clifford Stein. Extending Search Phases in the Micali-Vazirani Algorithm. In Costas S. Iliopoulos, Solon P. Pissis, Simon J. Puglisi, and Rajeev Raman, editors, *16th International Symposium on Experimental Algorithms (SEA 2017)*, volume 75 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:19, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[6] Silvio Micali and Vijay V. Vazirani. An $O(\sqrt{|V|}\,|E|)$ algorithm for finding maximum matching in general graphs. In *Proceedings of the 21st Annual Symposium on Foundations of Computer Science (SFCS 1980)*, pages 17–27, 1980.

[7] Vijay V. Vazirani. A proof of the mv matching algorithm, 2020.

[8] Vijay V. Vazirani. A theory of alternating paths and blossoms, from the perspective of minimum length. Lecture, October 2023. 02:00–04:30 PM PT.