Paths and cycles enumeration methods in graphs — GSoC 2025

Ziad Tarek

Contents

List of Algorithms

1 Personal

- Name: Ziad Tarek Abdelaziz
- Contact Information:
  - Email: ziad20037891@gmail.com
  - Discord: zezoo050
- Github: zezoo050
- Location/Timezone: Cairo — Egypt (UTC/GMT +2 hours)
- University/Employment: Student at Faculty of Computer and Information Sciences at Ain Shams University

2 Background

2.1 Education, Technical Background and experience

I am currently a senior, set to graduate in a few months. Over the past few years, I have studied various courses and gained experience in many areas. I have mainly focused on competitive programming and have achieved several successes:

- Champion of ECPC 2024 (ICPC Egyptian Collegiate Programming Contest)
- Silver medalist with Rank 3 at ACPC 2024 (ICPC Africa & Arab Collegiate Programming Championship)
- Qualified for the ICPC World Finals 2025 in Baku, Azerbaijan
- Finalist at the ICPC World Finals 2024 in Astana, Kazakhstan

You can view all my competitive programming achievements on my ICPCID, or Codeforces: zezoo050.

I also joined acmascis, a competitive programming community that trains and prepares students for contests. In my junior year, I led the problem-setting and testing teams. We created and tested problems not only for regional contests in Africa and the Arab region but also for the community to help train students. I managed about 15 members, ensuring that we met the community's needs for problems. Additionally, I trained team members so they could continue leading the community in the future.

Regarding mathematics, I am comfortable with the level needed to understand algorithms and the math used in computer science. As part of my training for competitive programming, I have studied various fields such as linear algebra, discrete math, graph theory, and geometry.

For programming, I have experience at different levels. I have worked on many projects, from building websites to developing low-level projects like database engines and operating systems.

2.2 About me & motivation

I have been interested in computer science since high school. When I started college, people suggested I try competitive programming. As a naturally competitive person, I found it enjoyable and learned a lot from it. After a couple of years, I noticed improvements in my speed and problem-solving skills, even for problems outside competitive programming. This training also made a big difference in my ability to handle projects and my understanding to theories.

My motivation for this project comes from two main sources. First, my passion for graph theory drives me to explore its many applications. Second, I have a strong desire to give back to the open source community. I have been using Linux and other open source software for several years, and these resources have helped me learn and grow. Now, I feel it is my turn to contribute. Google summer of code is a very good opportunity to get into open source. I am still a beginner and have a lot to learn, but I will do my best to contribute and continue growing in this field.

2.3 Operating system & Development environment

I have been using Linux for almost four years and have gained a lot of experience with it. Currently, my main operating system is Ubuntu.

2.4 Open source contributions

SageMath is my first open-source contribution. I have been using open-source software as much as possible and have explored the code of multiple open-source projects. However, my first direct engagement has been with SageMath. I was able to build SageMath from source code and familiarized myself with the source code. I was able to find a fix for an issue and opened a pull request (#39754) and it got approved. Additionally, reported a bug (#39756).

2.5 Pet projects

I do code on my own pet projects from time to time when I get interested in something, but currently I am more focused on GSoC and being able to contribute to open source projects. Here are some of the projects that could be of an interest:

- ImageEncryptCompress: Compression and Decompression of images using Huffman tree, and also encryption and decryption of images.
- spacechickensfmlgame: space chicken game built using C++ and SFML.
- GSoC25-SageMath-proposal: I actually consider this proposal as one of my pet projects that I am proud of. While I have been using *Latex* for technical writing for competitive programming problems, I have never used it on such a scale as this proposal before.

2.6 SageMath knowledge

My knowledge of SageMath does not exceed a couple of months. I have been using it for graph theory purposes, applying algorithms and visualizing graphs.

3 Project

- **Title:** Paths and cycles enumeration methods in graphs
- **Length:** Medium (175 hours)

3.1 Project Synopsis

The project is aimed to add some functionalities related to graph theory in SageMath. Specifically, it aims to add the following objectives:

- Add a method for the enumeration of simple cycles in undirected graphs, as there is already an implemented method for directed graphs. Then extend these methods to support enumeration of cycles by increasing weight.
- Implement recent and more efficient methods for finding the k shortest simple paths in (un) weighted (di) graphs.
- Unify the input and output of these methods to ensure similar behaviors.

3.2 Personal involvement

During my competitive programming experience, I was exposed to many areas of mathematics, including graph theory, which I found especially interesting. I became the team member with the most training and strength in graph theory on my competitive programming team. When I saw this project idea, I felt it was the perfect fit for me. I am confident that I have solved variations of the problems in this project during my training and in official contests. In addition, my experience in setting and testing problems has given me the skills needed to work on graph theory challenges.

3.3 Details

3.3.1 Cycle enumeration

Current implementation The only available implementation is for directed graphs. It works by first getting SCCs (strongly connected components) and then deleting the bridges (any edge between two different SCCs) from the graph (similar to Johnson's algorithm[3]). This prunes a lot of paths that might be traversed later but will never result in a cycle (no cycle ever contains a bridge). It then starts by calling a BFS (breadth-first search) from all starting vertices. The BFS ensures that all the cycles starting at the same vertex are yielded in increasing order of length. For multiple cycles starting at different vertices, we store the shortest cycle from each vertex in a heap. When a cycle needs to be returned, the shortest cycle is extracted from the heap and yielded, and then the next shortest cycle starting from the same vertex is added to the heap again.

Algorithm 1 *simple cycles in a directed graph*

Require: A digraph $D = (V, A)$, a set of startix vertices $S \subseteq V$
Ensure: Simple cycles in D yielded by increasing length

```
function _all_cycles_iterator_vertex(v)
    queue ← [[v]]
    while queue not empty do
        Extract p from queue
        if p is a simple cycle then
            yield p
        else
            for every neighbor v' ∈ V of last vertex of p do
                p' ← p + v'
                if p' is simple path or simple cycle then
                    append p' to the end of queue
                end if
            end for
        end if
    end while
end function
function all_cycles_iterator(S)
    SCCs ← strongly connected components of D
    brideges ← []
    for each v ∈ V do
        for each edge e = vv', v' ∈ V, e ∈ A do
            if v and v' are in different SCCs then
                add e to brideges
            end if
        end for
    end for
    A ← A \ brideges
    heap ← an empty heap
    for v ∈ S do
        add iterator _all_cycles_iterator_vertex(v) to heap
    end for
    while heap not empty do
        Extract iterator from heap, containing current shortest cycle
        shortest_cycle ← from iterator
        iterator ← next element from iterator
        if iterator not empty then
            append iterator to heap
        end if
        yield shortest_cycle
    end while
end function
```

Proposed approaches

- **For undirected graphs**: SCC doesn't apply directly. We could run the above algorithm without dividing the graph first, but this would lead to many unnecessary computations, as the BFS would start visiting paths that never yield a cycle. Instead, I propose using **BCCs**

(biconnected components). Biconnected components divide the graph into multiple components such that starting the BFS from vertices within a component—without traversing between components—ensures no extra paths (which cannot form cycles) are explored.

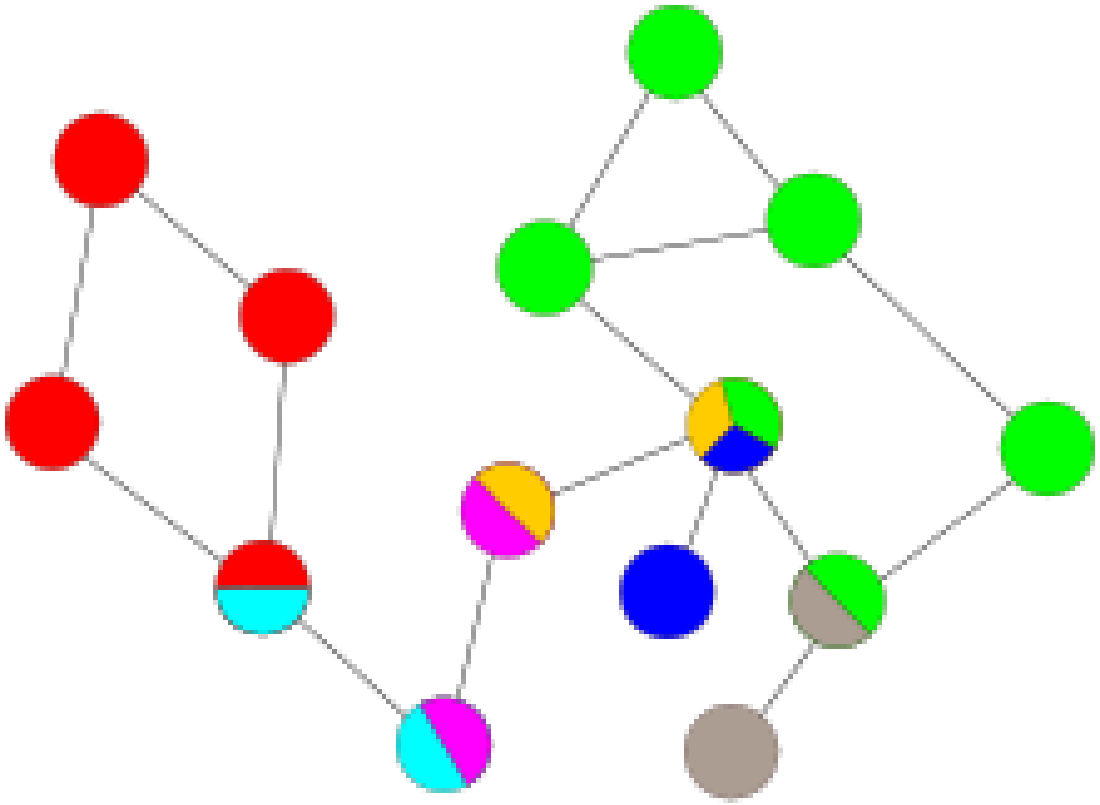

Figure 1: An example for Biconnected components in a graph

Algorithm 2 *simple cycles in a undirected graph* (modified from algorithm 1)

at line 1
function *_all_cycles_iterator_vertex*$(v, BCCs)$
...
starting from line 8
for every neighbor $v' \in V$ of last vertex of p, such that v' belong to the same biconnected component as p in $BCCs$ **do**
    $p' \leftarrow p + v'$
    if p' is simple path or simple cycle **then**
        append p' to the end of *queue*
    end if
end for
...
replaces lines 18 to 27
$BCCs \leftarrow$ biconnected components of D
...
at line 29
for $v \in S$ **do**
    add iterator *_all_cycles_iterator_vertex*$(v, BCCs)$ to *heap*
end for
...

This ensures that the shortest simple cycles are yielded without any drawbacks to efficiency compared to the method for directed graphs.

- **For weighted graphs**: the BFS finds shortest cycles first by searching by breadth, but for weighted graphs it cannot adapt to that since it never considers the weight and only considers the edges themselves. What I propose is adapting to a Dijkstra-like method, that is, by using a heap instead of a queue and storing and sorting based on the current path weight in the heap. This will ensure that the cycles with the smallest weight are retrieved first.

Algorithm 3 *simple cycles in a weighted graph* (modified from both algorithms 1 & 2)

```
...
in _all_cycles_iterator_vertex
heap ← [(0, [v])]                    ▷ a heap storing a path and its weight, while sorting on the weight
while heap not empty do
    Extract (w, p) from heap
    if p is a simple cycle then
        yield (w, p)
    else
        for every neighbor v' ∈ V of last vertex of p do
            p' ← p + v'
            w' ← w + w(vv')                                          ▷ new weight of p'
            if p' is simple path or simple cycle then
                append (w', p') to the heap
            end if
        end for
    end if
end while
...
```

Deliverables These are the tasks that are expected to be delivered for this part of the project.

- Simple cycles in undirected graph
- Extend methods to support weighted graphs

Each and every task will contain the following

- Core implementation
- Testing
- Documentation

All methods will follow a consistent input/output format:

- **Input**: Parameters specifying cycle properties (e.g., length bounds, weight thresholds).
- **Output**: Iterators yielding cycles in ascending order of weight/length.

3.3.2 K shortest simple paths

Current implementation The currently implemented methods are Yen's algorithm[5] and Feng's algorithm (node classification)[2]. The method *shortest_simple_paths* will by default uses Yen's for undirected graphs and Feng's for directed graphs.

Proposed approaches To implement efficient algorithms for the k shortest simple paths problem, we propose following the methods introduced in the paper "Finding the k Shortest Simple Paths: Time and Space Trade-offs"[1]. This is a recent paper, offering efficient optimizations for this problem. Furthermore, the fact that David Coudert, the mentor of this GSoC project, is a co-author of the paper makes this approach particularly well-suited for the project.

Algorithms to be implemented

- **PNC** (Postponed Node Classification)[1] This algorithm performs well on graphs resembling road networks, where the structure tends to be sparse. It offer a faster performance in practice over the classic NC as It delays some of the calls until necessary, in order to avoid some of them.

Algorithm 4 Postponed Node Classification (PNC)[1]

Require: A digraph $D = (V, A)$, source $s \in V$, sink $t \in V$, integer k
Ensure: k shortest simple $s - t$ paths

Let $Candidate \leftarrow \emptyset$ and $Output \leftarrow \emptyset$
$T \leftarrow$ an SP in-branching of D rooted at t
Add $(P_{st}(T), \omega(P_{st}(T)), 0, 1)$ to $Candidate$
while $Candidate \neq \emptyset$ and $|Output| < k$ **do**
    $(P = (s, u_1, \cdots, t), \omega(P), i, \zeta) \leftarrow$ extract shortest element from $Candidate$
    $\pi \leftarrow (s, u_1, \cdots, u_{i-1})$
    $Dev_{old} \leftarrow \{e = (u_i, v) \mid \exists$ path in $Output$ with prefix $\pi.e\}$
    if $\zeta = 1$ (P **is simple**) **then**
        Add P to $Output$
        $\lambda \leftarrow$ vertex labeling of D with respect to P and T
        for each vertex u_j in $(u_i, \cdots, t)$ **do**
            $(u_j, v_{LB}) \leftarrow$ arc in $A \setminus Dev_{old}$ with min δ at u_j
            $P_{LB} \leftarrow (s, \cdots, u_j, v_{LB}, P^T_{v_{LB}t})$
            $\zeta' \leftarrow 0$
            if $\lambda(v_{LB}) > j$ (P_{LB} **is simple**) **then**
                $\zeta' \leftarrow 1$
            end if
            Add $(P_{LB}, \omega(P_{LB}), j, \zeta')$ to $Candidate$
        end for
    else
        Compute shortest $u_i - t$ path Q in $D' = (V \setminus \pi, A \setminus Dev_{old})$
        if Q exists **then**
            Add $(P' = \pi.Q, \omega(P'), i, 1)$ to $Candidate$
        end if
    end if
end while
return $Output$

- **PSB** (Parsimonious Sidetrack-Based)[1] This variant is significantly more efficient for complex networks (e.g., social networks, web graphs) where the degrees of the node is higher and diameter tends to be shorter. It offers a trade-off compared to the PNC as it uses more memory but is faster on this type of graphs.

Algorithm 5 Sidetrack Based (SB) Algorithm for the kSSP (which the PSB is based upon)[4]

Require: A digraph $D = (V, A)$, source $s \in V$, sink $t \in V$, integer k
Ensure: k shortest simple s-t paths

Let $Candidate \leftarrow \emptyset$ and $Output \leftarrow \emptyset$
$T_0 \leftarrow$ an SP in-branching of D rooted at t containing s
Add $\big((T_0), \omega(P_{st}(T_0)), \zeta = 1\big)$ to $Candidate$
while $Candidate \neq \emptyset$ and $|Output| < k$ **do**
    $\mu = \big(\varepsilon = ((T_0, e_0, \cdots, T_h, e_h = (u_h, v_h), T_{h+1}), lb, \zeta)\big) \leftarrow$ a shortest element in $Candidate$
    if $\zeta = 1$ **then**
        Extract μ from $Candidate$ and add ε to $Output$
        for every deviation $e = v_j v'$ with $v_j \in P_{v_i,t}^{T_{h+1}}$ **do**
            $\varepsilon' \leftarrow (T_0, e_0, \cdots, T_h, e_h, T_{h+1}, e, T_{h+1})$
            $lb' \leftarrow lb - \omega(P_{v_j,t}^{T_{h+1}}) + \omega(e) + \omega(P_{v't}^{T_{h+1}})$
            if ε' represents a simple path **then**
                Add $\mu' = (\varepsilon', lb', \zeta = 1)$ to $Candidate$
            else
                $T' \leftarrow$ the name of an SP in-branching of $D_j(P)$ ▷ T' is not computed yet
                Add $\mu'' = \big(\varepsilon'' = (T_0, e_0, \cdots, T_h, e_h, T_{h+1}, e, T'), lb', \zeta = 0\big)$ to $Candidate$
            end if
        end for
    else
        if T_{h+1} has not been computed yet **then**
            Compute T_{h+1}, an SP in-branching of $D_h(P)$
        end if
        $\mu' = \big(\varepsilon' = (T_0, e_0, \cdots, T_h, e_h, T_{h+1}), lb + \omega(P_{v_i,t}^{T_{h+1}}) - \omega(P_{v_i,t}^{T_h}), \zeta = 1\big)$
        Add μ' to $Candidate$
    end if
end while
return $Output$

Selection algorithm As discussed in the paper, the optimal algorithm depends heavily on the properties of the input graph. What I propose:

Analyzing the graph (e.g., average degree, diameter) and Automatically selecting between PNC and PSB based on the graph's properties:

- Use PNC for sparse, road-like graphs
- Use PSB for dense, complex networks

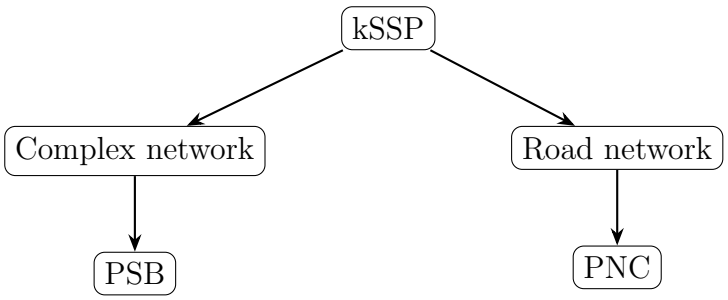

Figure 2: Selection decisions for kSSP

Both algorithms natively support weighted directed graphs, and can be easily extended to undirected graphs and unweighted graphs.

Deliverables These are the tasks that are expected to be delivered for this part of the project.

- PNC (Postponed Node Classification)
- PSB (Parsimonious Sidetrack-Based)
- Selection algorithm

Each and every task will contain the following

- Core implementation
- Testing
- Documentation

All methods will follow a consistent input/output format:

- **Input**: Parameters specifying the paths properties (e.g., starting vertices, ending vertices, weight thresholds).
- **Output**: Iterators yielding paths in ascending order of weight/length.

Note: Neither the current implementation nor the proposed one will accept a parameter k (the number of shortest paths to retrieve). Instead, the iterator will continue computing paths until no more can be found.

3.4 Schedule

Phase	**Dates (Weeks)**	**Activities**
Phase 1: Community Bonding & Simple cycles in undirected graph	May 8 — May 23 (3 wks)	• Engage with the community discuss project goals and details with the mentor. • Familiarize myself more with the source code. • Read the documentation and gain a deeper understanding of the development process. • Cycle enumeration: Simple cycles in undirected graphs
Phase 2: Extend methods to support weighted graphs	May 24 — June 07 (2 wks)	• Cycle enumeration: Extend methods to support weighted graphs • Extra testing for Cycle enumeration part.
Phase 3: PNC	June 08 — July 13 (4 wks)	• kSSP: PNC (Postponed Node Classification)
Phase 4: PSB	July 14 — August 11 (4 wks)	• kSSP: PSB (Parsimonious Sidetrack-Based)
Phase 5: Selection algorithm & Finalization	August 12 — September 1 (3 wks)	• kSSP: Selection algorithm. • Extra Testing for kSSP part. • Address any remaining issues or bugs. • Finalization.

Table 1: Project Plan and Timeline.

- Every phase is tested and documented before proceeding to the next phase.
- My availability every week ranges from 20 to 25 hours.
- I will have my final exams and some holidays in late May / early June. Although this will not affect the project schedule, I will be unavailable on certain days.
- Full-Time Job Possibility: There is a chance that I could start a full-time job around mid-July or August. Regardless of this, I will ensure that the project schedule is met and all planned milestones are completed on time.

3.5 Risk Management

While I believe I am well-suited for this project, I have identified several potential risks that could impact the project:

1. Some parts of this project depend on modules that are already implemented (like trees and other data structures). I might face problems with these modules, such as bugs or unexpected behavior. In the best case, I will be able to fix the issues in these modules and continue with the project as planned. In the worst case, I will write a temporary implementation to replace the

broken module. This might cause some code duplication, but it will let me continue working until the original module is fixed.

2. Delays might happen due to either unexpected bugs or limitations. When drafting the project schedule, I made sure to give extra time for each phase. However, in worst case scenario, the order of the phases in the schedule guarantees that a big part of the project will be done.

References

[1] Ali Al Zoobi, David Coudert, and Nicolas Nisse. Finding the k shortest simple paths: Time and space trade-offs. *ACM J. Exp. Algorithmics*, 28, December 2023.

[2] Gang Feng. Finding k shortest simple paths in directed graphs: A node classification algorithm. *Networks*, 64(1):6–17, 2014.

[3] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[4] Denis Kurz and Petra Mutzel. A sidetrack-based algorithm for finding the k shortest simple paths in a directed graph. *CoRR*, abs/1601.02867, 2016.

[5] Jin Y. Yen. Finding the K shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.