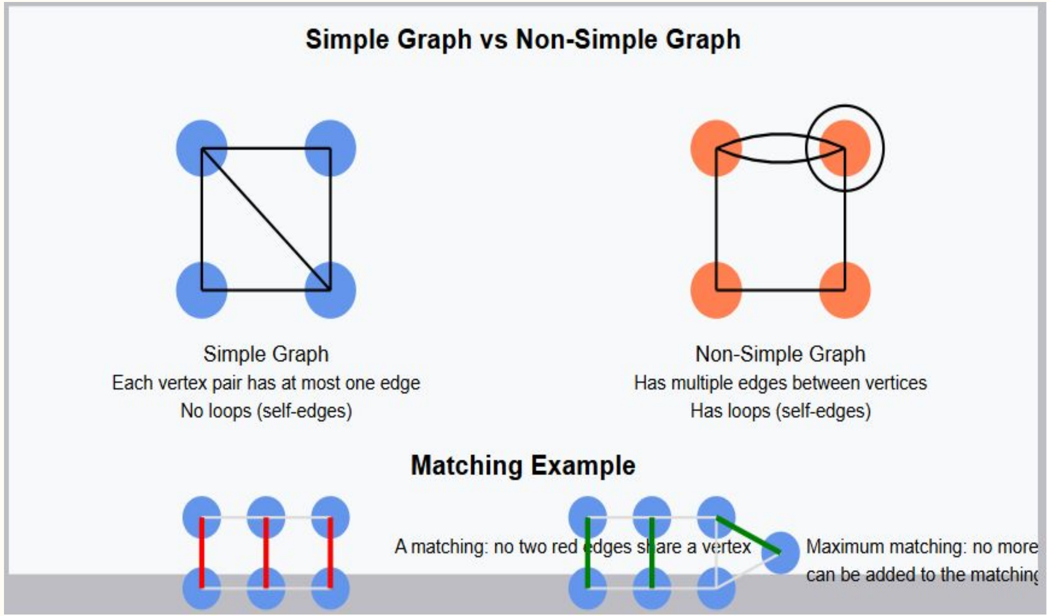
Increasing Edge-Connectivity in simple graphs

Hamza Ansari, Roshaan Khan

What is a simple Graph ?



- "A simple graph is a graph without multiple edges between the same pair of vertices and without loops (edges from a vertex to itself)"
- Example: "Facebook's friendship network is a simple graph - either you're friends with someone or not"

What is Edge-Connectivity ?

- A graph is **k-edge-connected** if it requires removing at least **k** edges to disconnect it
- The **edge-connectivity** of a graph is the minimum number of edges whose removal disconnects the graph

What is matching ?

- "A matching in a graph is a set of edges where no two share a common vertex"
- "A maximum matching is a matching with the largest possible number of edges"

What is the Problem?

- Given a k-edge-connected simple graph, find the smallest set of edges from the complement graph whose addition results in a $(k+1)$-edge-connected graph.

Ok ...but what exactly ?

Why is it important ?

- Network reliability: Higher connectivity means more resistant to failures
- Communication networks: Better connectivity means more robust communication paths
- Transportation planning: More connected road networks are more resilient

Computational Advantage

A brute force approach would be extremely costly:

- For a graph with n vertices, there are up to $O(n^2)$ edges in the complement
- We would need to check all possible subsets of these edges
- This gives $2^{O(n^2)}$ possible solutions to check
- For each candidate solution, we'd need to verify if it creates a $(k+1)$-edge-connected graph, which itself isn't trivial

It is an NP-Hard Problem

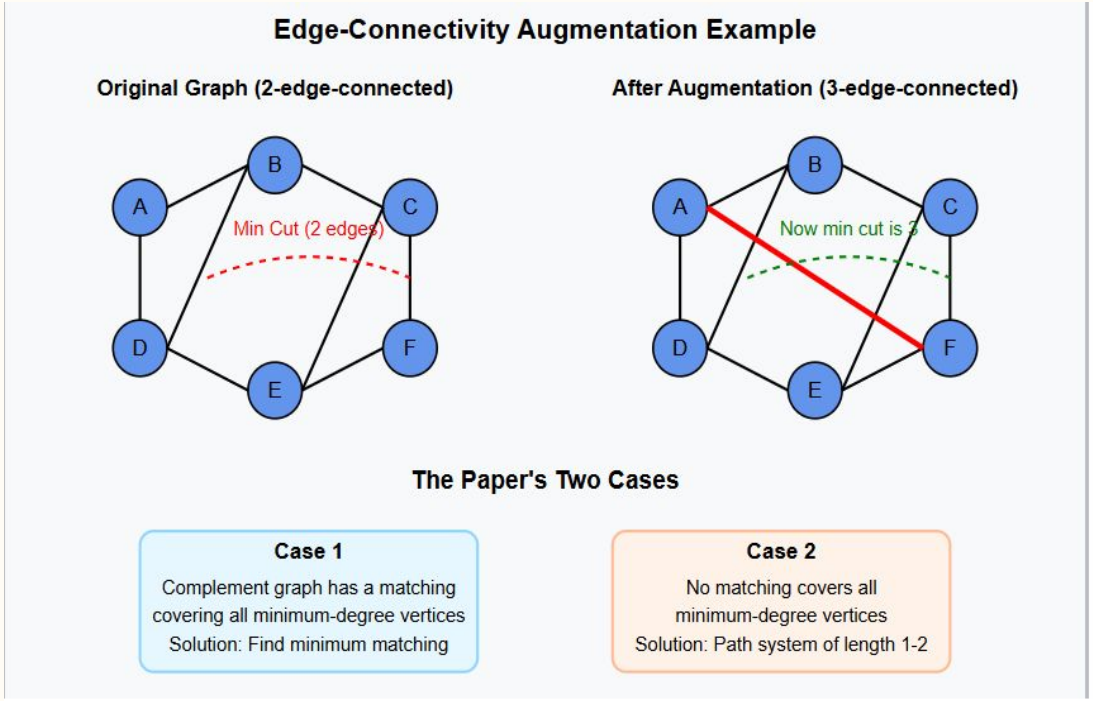
- **Combinatorially Hard:** Choosing the smallest edge set from the complement grows exponentially with graph size.

- **Augmenting a *K*-edge-connected graph to *(k+1)*-edge-connected involves choosing the smallest possible subset of edges from the complement graph. The number of possible subsets grows exponentially with the number of candidate edges—making brute-force solutions computationally infeasible.**

What does the paper deal with ?

Two main cases:

- Case 1: When the complement graph has a matching covering all minimum-degree vertices
- Case 2: When such a matching doesn't exist



Complexity

Case 1 (Matching):

Uses Edmonds' algorithm → runs in **$O(n^3)$** for maximum matching.

Case 2 (Path-Based):

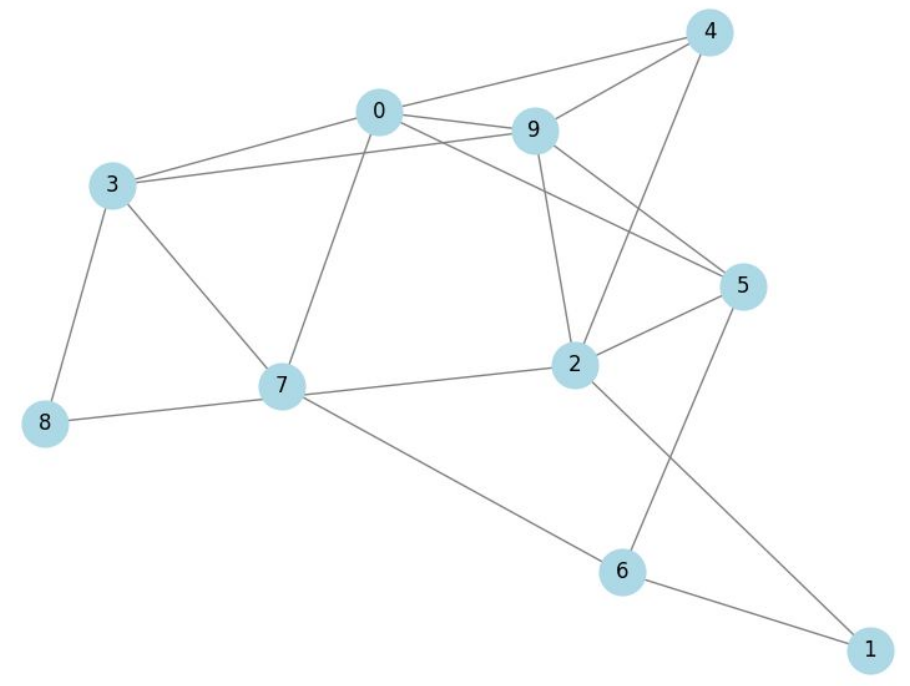
Constructs augmenting paths of length 1 or 2 → worst-case complexity **$O(n^3)$** (due to pairwise searches and modifications).

Overall:

Polynomial-time in special cases, but problem remains NP-hard in general.

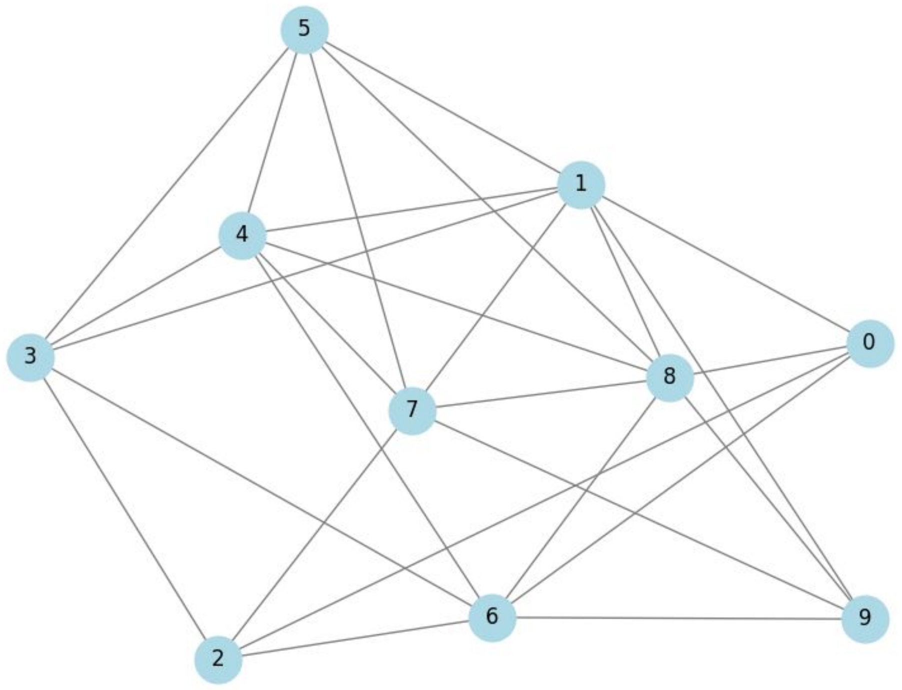
Sample Data test

G = [(0, 3), (0, 4), (0, 9), (0, 5), (0, 7), (3, 8), (3, 7), (4, 9), (9, 5), (9, 2), (5, 2), (5, 6), (2, 7), (2, 6), (7, 8), (6, 1)]



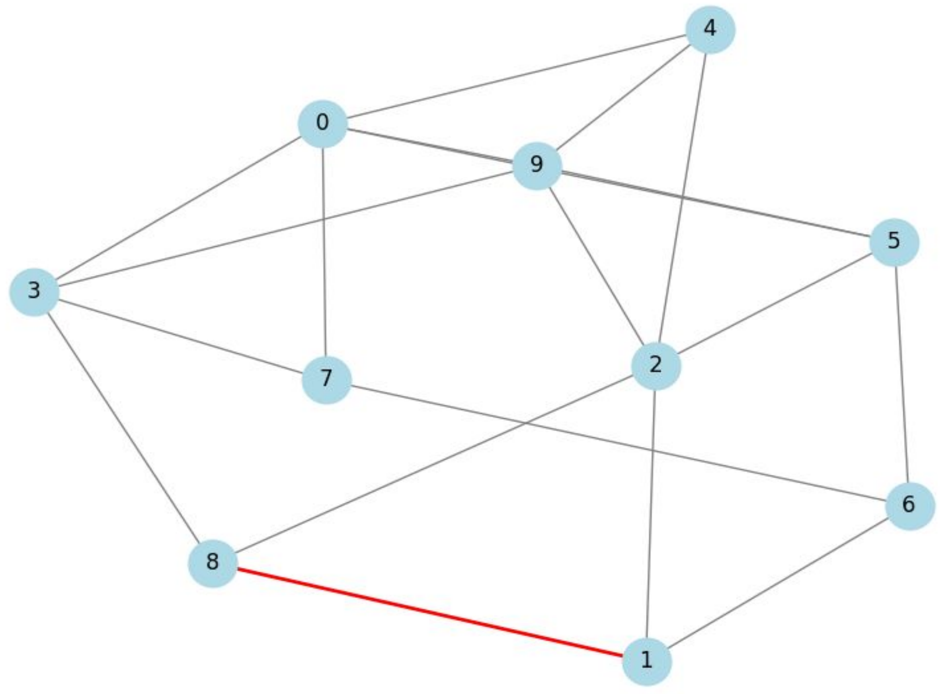
Complement Graph

G = [(0, 3), (0, 4), (0, 9), (0, 5), (0, 7), (3, 8), (3, 7), (4, 9), (9, 5), (9, 2), (5, 2), (5, 6), (2, 7), (2, 6), (7, 8), (6, 1)]



Augmented Graph

G = [(0, 3), (0, 4), (0, 9), (0, 5) (0, 7), (3, 8), (3, 7), (4, 9), (9, 5), (9, 2), (5, 2), (5, 6), (2, 7), (2, 6), (7, 8), (6, 1), (1, 8)]



Overall Improvement

By focusing on matchings and path systems in the complement:

- The problem reduces to finding maximum matchings, which can be done in $O(n^3)$ time using Edmonds' algorithm
- We avoid examining all possible edge subsets
- The results prove that these specific structures (matchings or path systems) are sufficient
- This makes the problem tractable in polynomial time for the cases studied

Understanding the Code

For the conditions: check if it is case 1 or case 2

```
vertices_k = get_vertices_of_degree_k(G, CONNECTIVITY_LEVEL)
G_k = get_subgraph_by_vertices(G, vertices_k)
complement_G_k = get_complement_graph(G_k)
matching = compute_maximum_matching(complement_G_k)

if len(matching) * 2 == len(vertices_k):
    print("Case 1: Matching covers all degree-k vertices.")
else:
    print("Case 2: Matching does NOT cover all degree-k vertices.")

# Step 4: Augment graph
G_aug, added_edges = augment_graph_via_matching(G, CONNECTIVITY_LEVEL)
```

Understanding the Code

Testing = random and test cases

```
MODE = "testcase"
#generate random number between 1 and 3
random_num = random.randint(1, 6)
if random_num == 1:
    TESTCASE_EDGES = [(0,1), (1,2), (2,3), (3,4), (4,0)]
elif random_num == 2:
    TESTCASE_EDGES = [(0,1), (1,2), (2,3), (3,0)]
elif random_num == 3:
    TESTCASE_EDGES = [(0, 1), (1, 2), (2, 0), (3, 4), (4, 5), (5, 3)] # 2 regular disjoint cycles
elif random_num == 4:
    TESTCASE_EDGES =  [(0, 1), (0, 2), (0, 3), (0, 4),
                      (1, 2), (1, 3), (1, 4),
                      (2, 3), (2, 4),
                      (3, 4)] # Complete graph K5 near complete 5
elif random_num == 5:
    TESTCASE_EDGES = [(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 0), (0, 2), (3, 5)]
elif random_num == 6:
    TESTCASE_EDGES = [(0, i) for i in range(1, 6)]
```

THANK YOU