Polynomial-Time Recognition Algorithm for Perfect Graphs

Algorithm by Maria Chudnovsky, Gérard Cornuéjols, Xinming Liu, Paul Seymour and Kristina Vušković
Later extended by Maria Chudnovsky, Alex D Scott, Paul D Seymour and Sophie Theresa Spirkl

IIT Palakkad

June 26, 2026

Introduction

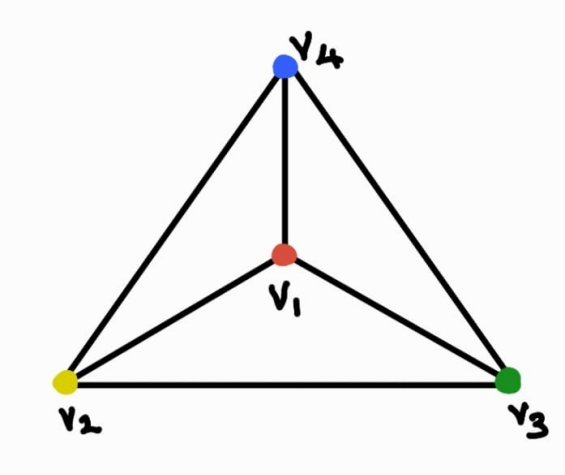
Definition (Perfect graph)

Graphs in which the chromatic number equals the size of the maximum clique, both in the graph itself and in every induced subgraph.

Introduction

Definition (Perfect graph)

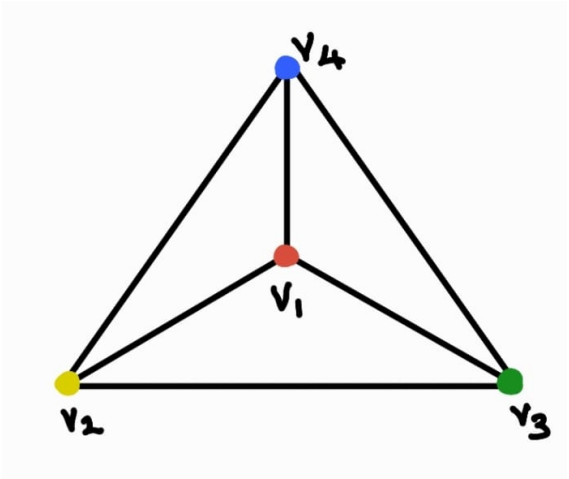
Graphs in which the chromatic number equals the size of the maximum clique, both in the graph itself and in every induced subgraph.



Introduction

Definition (Perfect graph)

Graphs in which the chromatic number equals the size of the maximum clique, both in the graph itself and in every induced subgraph.



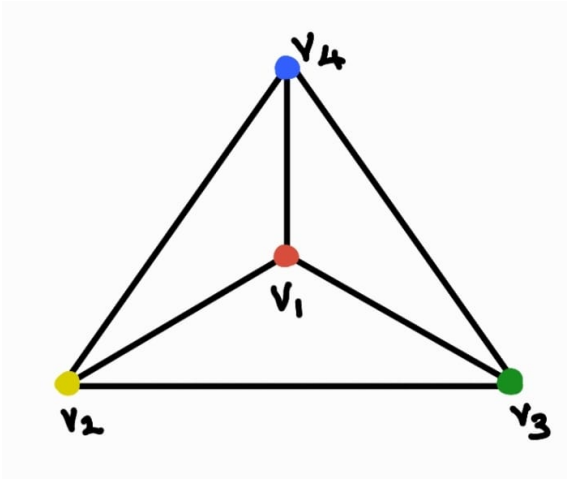
Goal of this talk

Given G, check if G is perfect in $\mathsf{poly}(|V(G)|)$ time.

Introduction

Definition (Perfect graph)

Graphs in which the chromatic number equals the size of the maximum clique, both in the graph itself and in every induced subgraph.



Goal of this talk

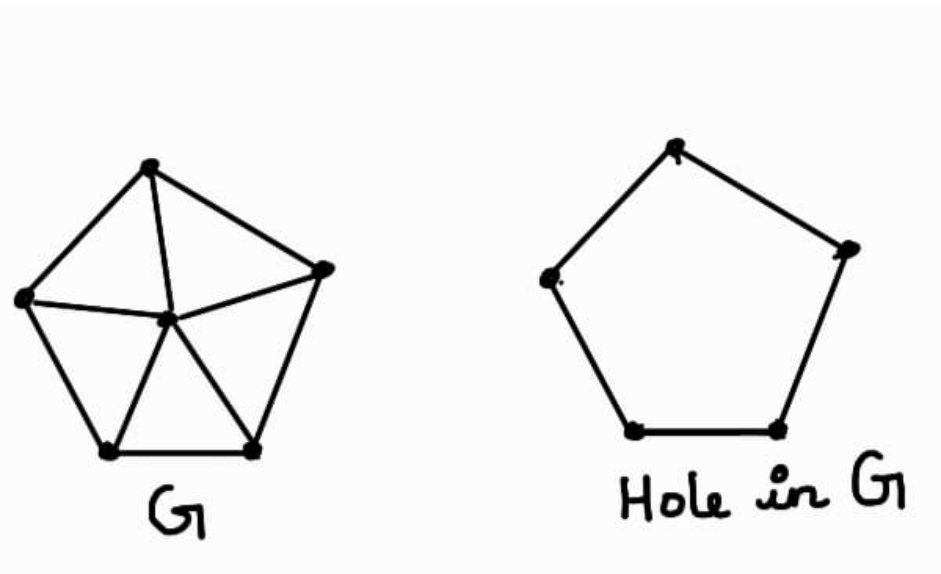
Given G, check if G is perfect in $\mathsf{poly}(|V(G)|)$ time.

Definitions

- A **hole** in G is an induced subgraph of G that is a cycle of length at least four. If the cycle length is odd, it is an odd hole.

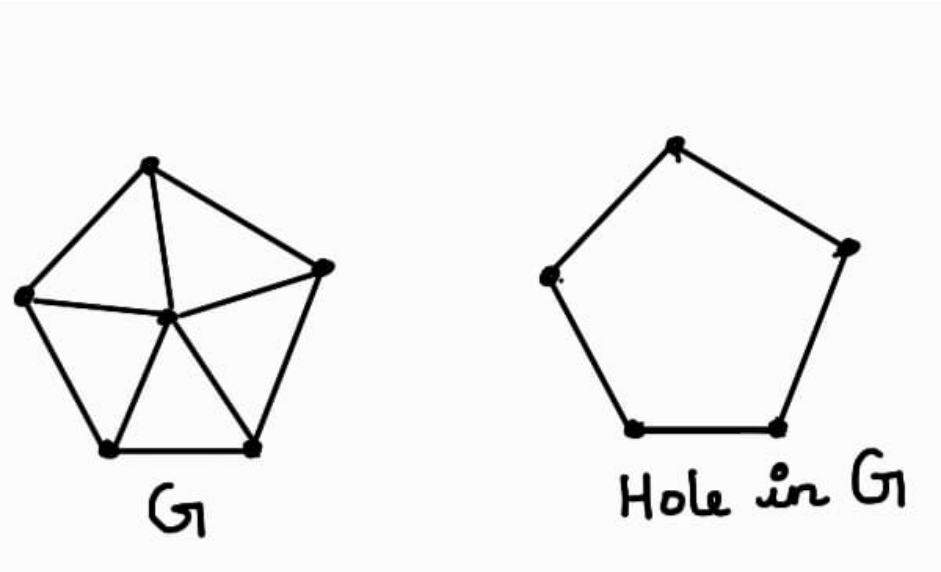
Definitions

- A **hole** in G is an induced subgraph of G that is a cycle of length at least four. If the cycle length is odd, it is an odd hole.



Definitions

- A **hole** in G is an induced subgraph of G that is a cycle of length at least four. If the cycle length is odd, it is an odd hole.



- **Berge graphs:** A graph G and its complement have no odd holes.

Examples and Approach

Strong Perfect Graph Theorem [SPG Theorem]

A graph is Berge if and only if it is perfect[a].

[a] Maria Chudnovsky, Neil Robertson, Paul Seymour, and Robin Thomas, "The Strong Perfect Graph Theorem," *Annals of Mathematics*, vol. 164, no. 1, 2006.

Examples and Approach

Strong Perfect Graph Theorem [SPG Theorem]

A graph is Berge if and only if it is perfect[a].

[a] Maria Chudnovsky, Neil Robertson, Paul Seymour, and Robin Thomas, "The Strong Perfect Graph Theorem," *Annals of Mathematics*, vol. 164, no. 1, 2006.

Approach to check for perfectness

- G is perfect iff G is Berge [SPG Theorem]

Examples and Approach

Strong Perfect Graph Theorem [SPG Theorem]

A graph is Berge if and only if it is perfect[a].

[a] Maria Chudnovsky, Neil Robertson, Paul Seymour, and Robin Thomas, "The Strong Perfect Graph Theorem," *Annals of Mathematics*, vol. 164, no. 1, 2006.

Approach to check for perfectness

- G is perfect iff G is Berge [SPG Theorem]
- Suffices to check that G has an odd Hole (FindHole).

Examples and Approach

Strong Perfect Graph Theorem [SPG Theorem]

A graph is Berge if and only if it is perfect[a].

[a] Maria Chudnovsky, Neil Robertson, Paul Seymour, and Robin Thomas, "The Strong Perfect Graph Theorem," *Annals of Mathematics*, vol. 164, no. 1, 2006.

Approach to check for perfectness

- G is perfect iff G is Berge [SPG Theorem]
- Suffices to check that G has an odd Hole ($\mathsf{FindHole}$).
- Berge(G) = $\neg\mathsf{FindHole}(G) \wedge \neg\mathsf{FindHole}(\overline{G})$.

Takeaway: Suffices to check for Odd holes

Procedure FindHole(G)

Consider the below algorithm to check for an odd hole

Algorithm 1: FindHole(G)

```
1 foreach u, v ∈ V do
2     P(u,v)=Length of shortest path between u and v;
3 foreach u, v, w of distinct vertices in V do
4     if P(u,v) + P(v,w) + P(w,u) is atleast five, is odd and induced
        then
5         return "True";
6 return "False";
```

Correctness of FindHole(G)

- If algo says "Odd Hole" $\implies$ Odd Hole in G.

Correctness of FindHole(G)

- If algo says “Odd Hole” $\implies$ Odd Hole in G.
- To show: Odd hole in G $\implies$ Algo says "Odd Hole"

Correctness of FindHole(G)

G has Odd hole $\implies$ Algo says "Odd Hole"

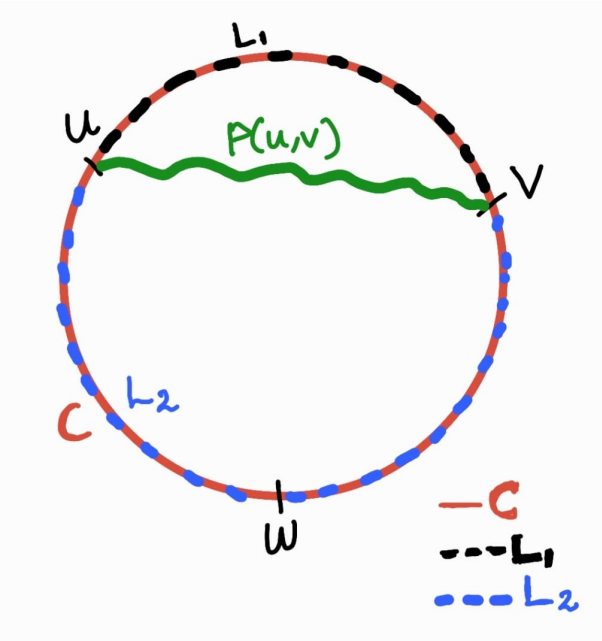


Figure: Original Odd Hole

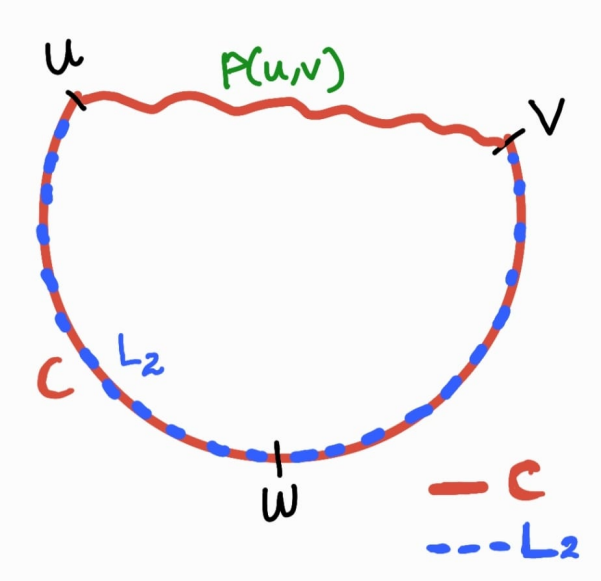


Figure: New Odd Hole

Correctness of FindHole(G)

G has Odd hole $\implies$ Algo says "Odd Hole"

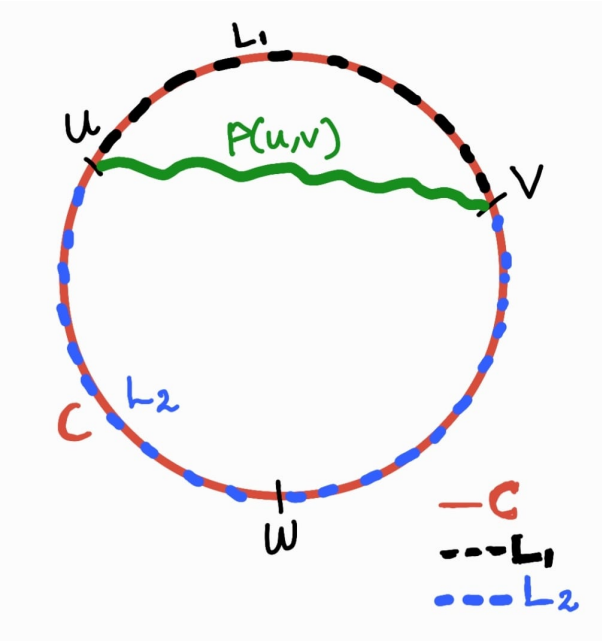


Figure: Original Odd Hole

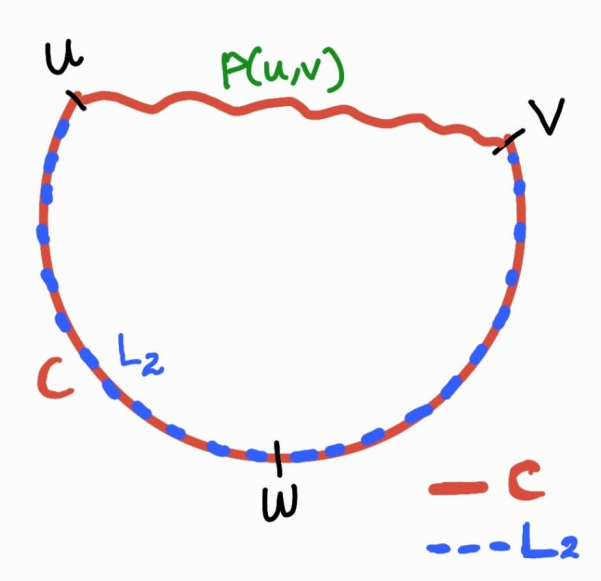


Figure: New Odd Hole

Suppose ...

C is a shortest odd hole $\implies$ $P(u,v) \cup L_2$ is a shortest odd hole.

- Cut off L_1. Attach $P(u, v)$.
- Repeat for $P(v, w)$ and $P(w, u)$

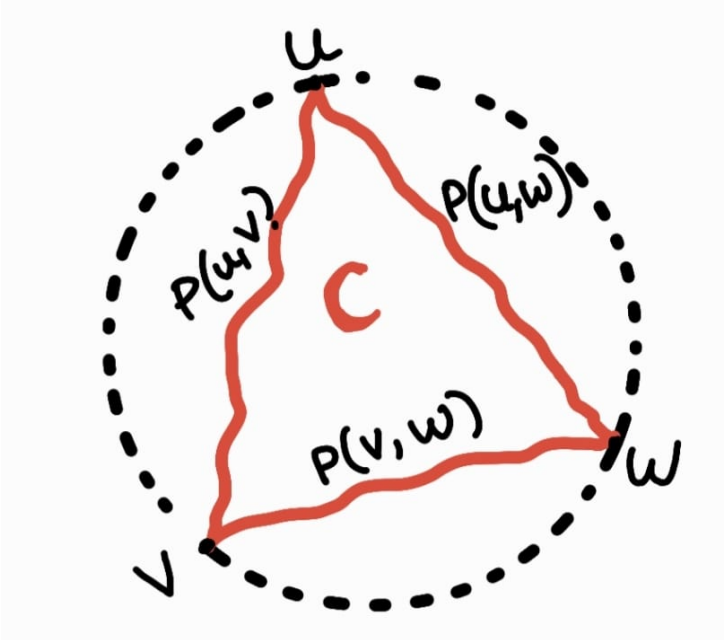


Figure: New Odd Hole C in red

- Cut off L_1. Attach $P(u, v)$.
- Repeat for $P(v, w)$ and $P(w, u)$

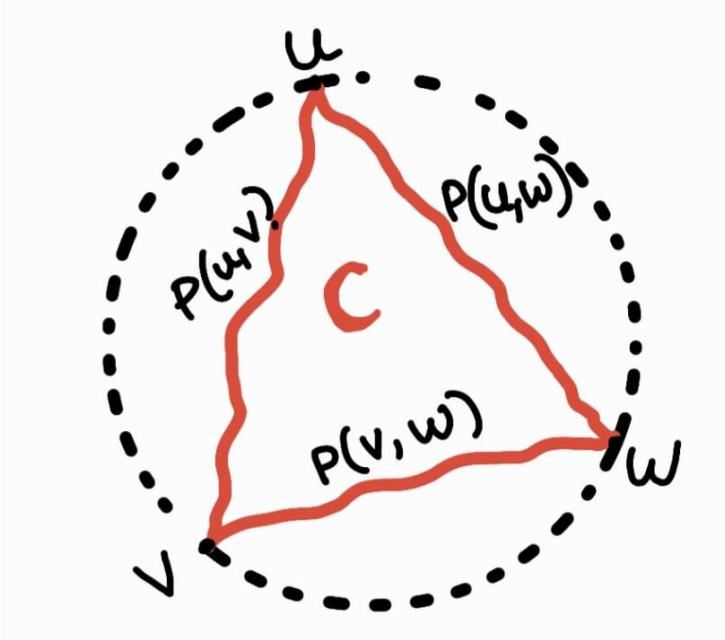


Figure: New Odd Hole C in red

G has a shortest odd hole where paths are shortest !

- Cut off L_1. Attach $P(u, v)$.
- Repeat for $P(v, w)$ and $P(w, u)$

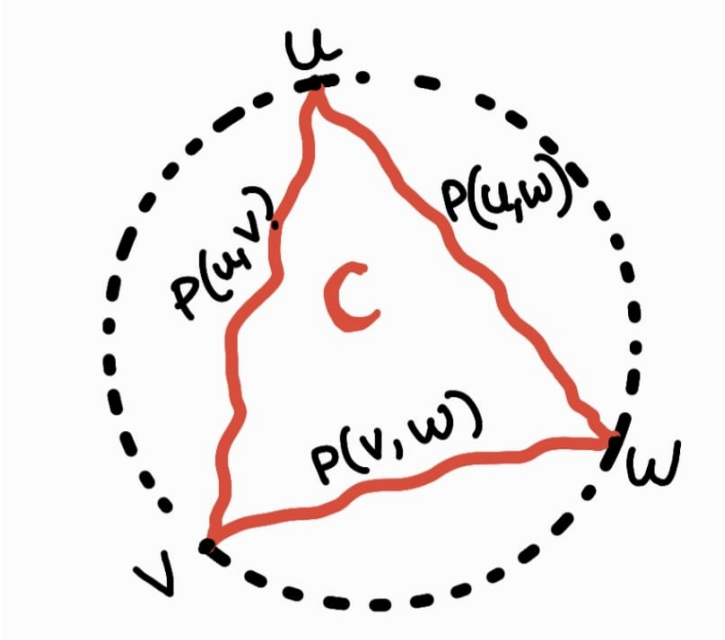


Figure: New Odd Hole C in red

G has a shortest odd hole where paths are shortest ! Algo will say "Yes".

Shortest Path Replacement Theorem

Let G be a graph without some special structures and C be the shortest odd hole in G. Let $u, v \in V(C)$ be distinct and nonadjacent, and let L_1, L_2 be the two subpaths of C joining u and v, with $|E(L_1)| < |E(L_2)|$. Then:

- L_1 is a shortest path in G between u and v, and
- for any shortest path P in G between u and v, the cycle $P \cup L_2$ is a shortest odd hole in G.

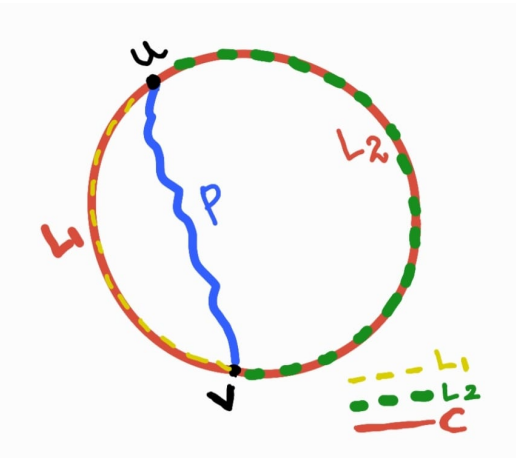


Figure: Original Odd Hole

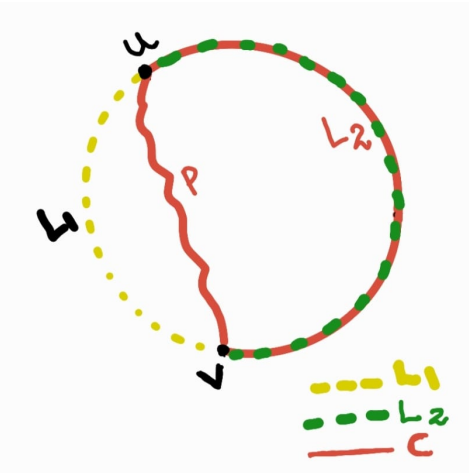
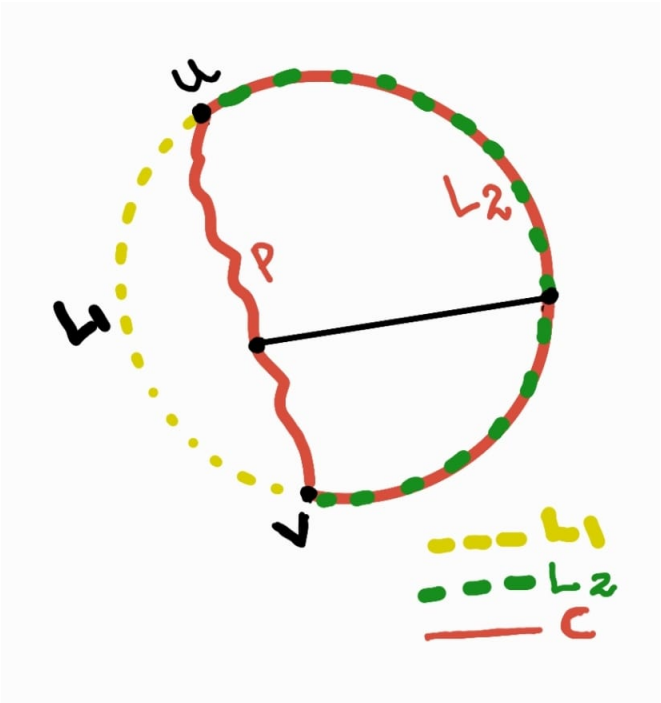


Figure: New Odd Hole

Shortest Path Replacement Theorem

Let G be a graph without some special structures, and let C be the shortest odd hole in G. Let $u, v \in V(C)$ be distinct and nonadjacent, and let L_1, L_2 be the two subpaths of C joining u and v, with $|E(L_1)| < |E(L_2)|$. Then:

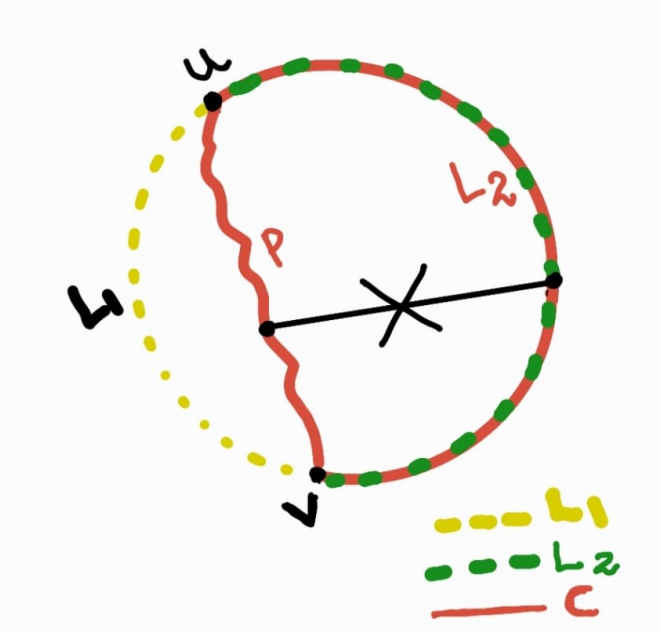
- L_1 is a shortest path in G between u and v, and
- for any shortest path P in G between u and v, the cycle $P \cup L_2$ is a shortest odd hole in G.



Shortest Path Replacement Theorem

Let G be a graph without some special structures, and let C be the shortest odd hole in G. Let $u, v \in V(C)$ be distinct and nonadjacent, and let L_1, L_2 be the two subpaths of C joining u and v, with $|E(L_1)| < |E(L_2)|$. Then:

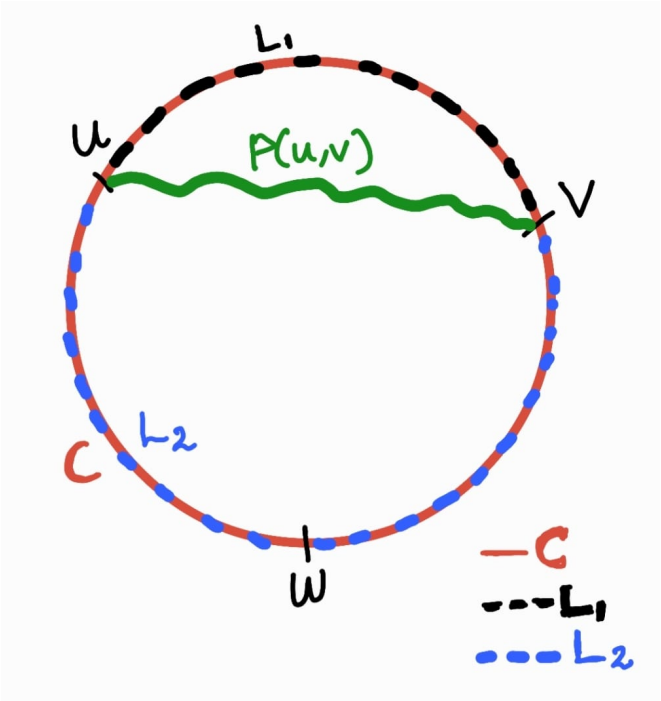
- L_1 is a shortest path in G between u and v, and
- for any shortest path P in G between u and v, the cycle $P \cup L_2$ is a shortest odd hole in G.



Correctness of FindHole(G)

G has an odd hole $\implies$ Algo says "Odd Hole"

- C: Shortest odd hole with special properties.
- Choose roughly equally spaced vertices $u, v, w \in V(C)$.
- L_1: u–v subpath of C **not** passing through w.
- $P(u, v)$: shortest u–v path



Correctness of FindHole(G)

G has an odd hole $\implies$ Algo says "Odd Hole"

Claim

C can be chosen to contain $P(u, v)$.

- By Shortest Path Replacement Theorem, L_1 and $P(u, v)$ have the same length.
- The cycle formed by $L_2 \cup P(u, v)$ is a shortest odd hole (by Shortest Path Replacement Theorem).
- Hence we may choose C that includes $P(u, v)$.

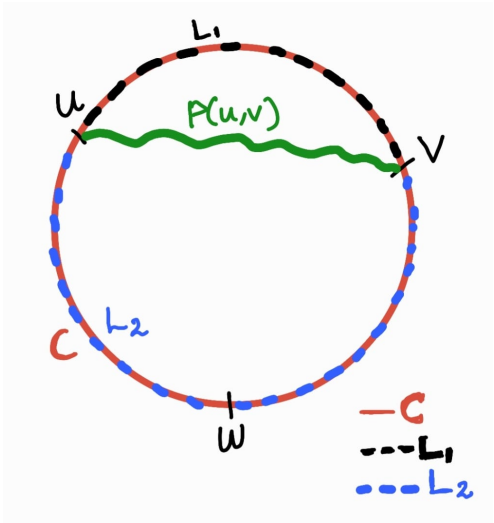


Figure: Original Odd Hole

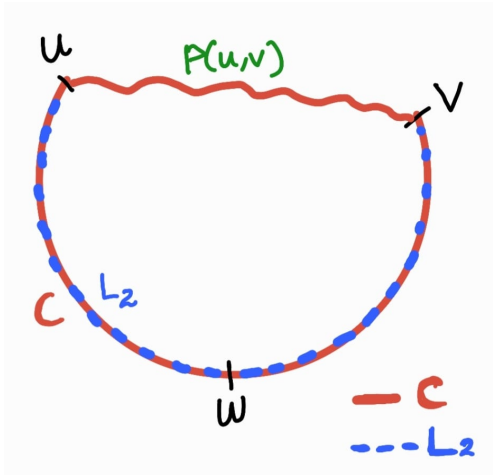


Figure: New Odd Hole

Correctness of FindHole(G)

G has an odd hole $\implies$ Algo says "Odd Hole"

- Repeat the same for (u, v), (v, w), and (w, u).
- Union of these three paths is an odd hole, so algorithm correctly outputs “Odd hole”.

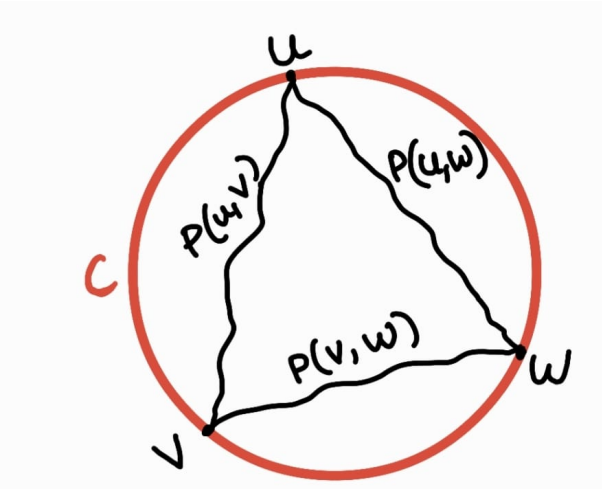


Figure: Original Odd Hole

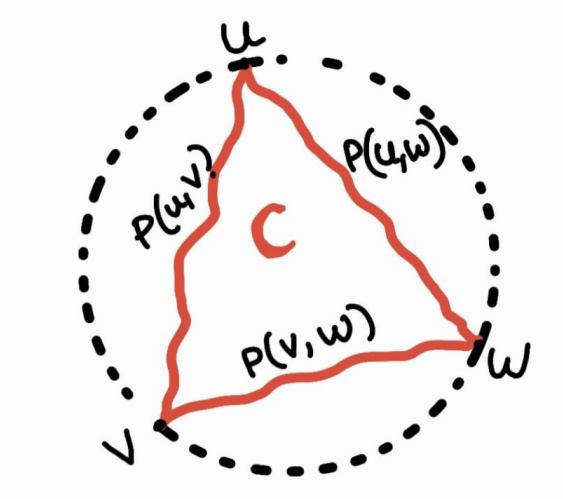


Figure: New Odd Hole

Correctness of FindHole(G)

G has an odd hole $\implies$ Algo says "Odd Hole"

- Repeat the same for (u, v), (v, w), and (w, u).
- Union of these three paths is an odd hole, so algorithm correctly outputs “Odd hole”.

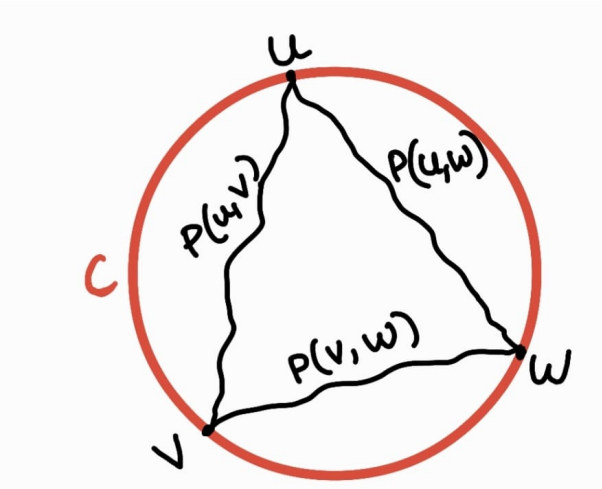


Figure: Original Odd Hole

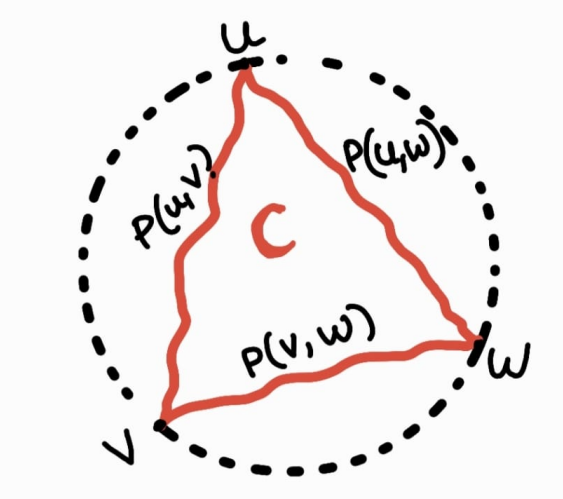


Figure: New Odd Hole

Correctness Summary

G has **clean** shortest odd hole and G does not have **Jewel** and **Pyramid**, then FindHole(G) outputs “Odd Hole”.

Steps

The algorithm has three parts:

Steps

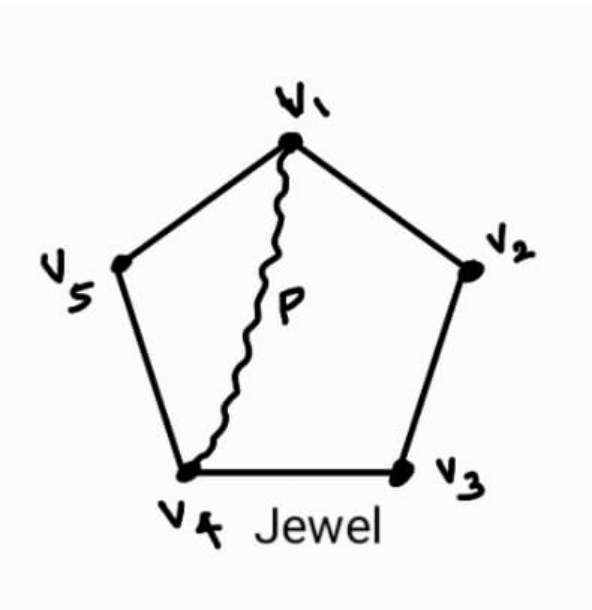
The algorithm has three parts:

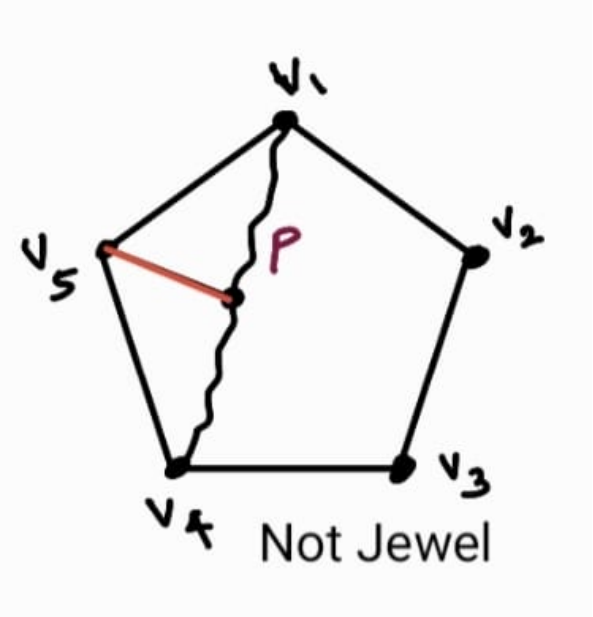
- Step 1: Detect some "easy" configurations (pyramid, jewel, 5-hole).
- Step 2: Cleaning Step: removing C-major vertices
- Step 3: Let G' be resulting graph. Run $\mathsf{FindHole}(G')$.

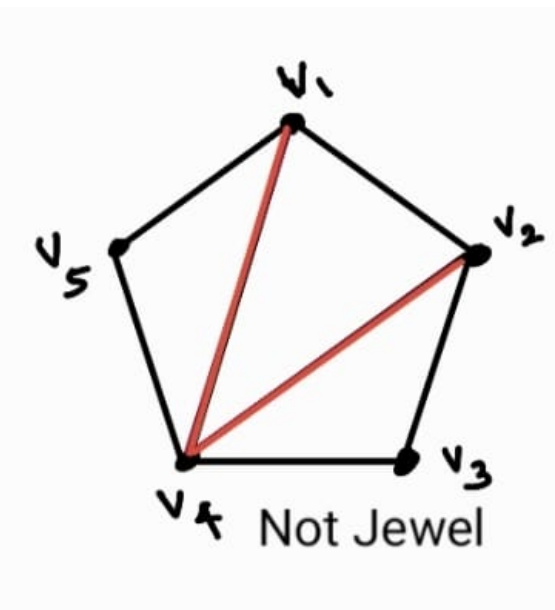
Detecting "Easy" configurations : Jewels

We say distinct vertices $v_1, \ldots, v_5$, and a path P forms a jewel in G if

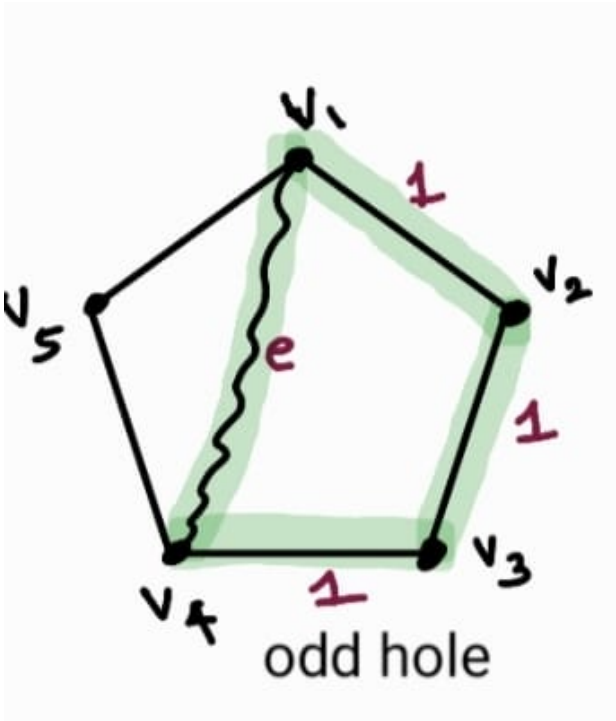
- $v_1 - v_2 - v_3 - v_4 - v_5 - v_1$ is a cycle.
- $(v_1, v_3), (v_2, v_4), (v_1, v_4)$ are non-edges.
- P is a path of G between v_1,v_4 such that v_2, v_3, v_5 have no neighbours in P^*.

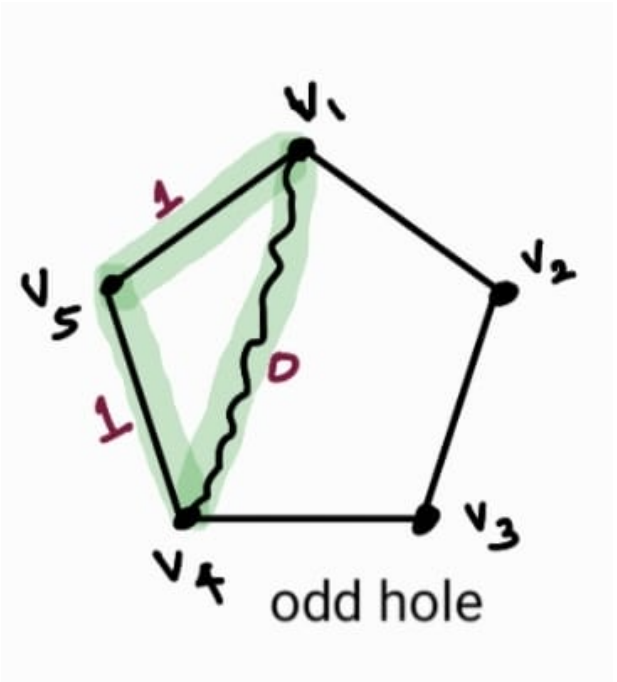






Jewel induces an odd hole

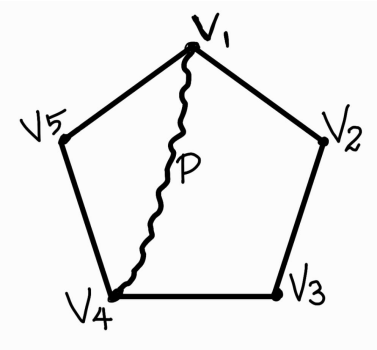




How to check for jewels in a graph?

Algorithm 2: FindJewel(G)

```
1 foreach choice of v_1, ..., v_5 satisfying edge and non-edge conditions
   do
   |  // N[v] is closed neighbourhood of v
2  |  Run BFS from v_1 in G \ (N[v_2] ∪ N[v_3] ∪ N[v_5]) keeping v_1 and
   |   v_4 unremoved;
3  |  if v_4 is reached then
4  |  |  return "Jewel found";
5  |  end
6 end
7 return "No Jewel"
```



Correctness

for FindJewel

Claim

If the algorithm outputs "Jewel found", then G contains a jewel.

- Algorithm outputs "Jewel found" for some choice of $v_1, \ldots, v_5$.

Correctness

for FindJewel

Claim

If the algorithm outputs "Jewel found", then G contains a jewel.

- Algorithm outputs "Jewel found" for some choice of $v_1, \ldots, v_5$.
- BFS from v_1 reaches v_4 avoiding $N[v_2], N[v_3], N[v_5]$.

Correctness

for FindJewel

Claim

If the algorithm outputs "Jewel found", then G contains a jewel.

- Algorithm outputs "Jewel found" for some choice of $v_1, \ldots, v_5$.
- BFS from v_1 reaches v_4 avoiding $N[v_2], N[v_3], N[v_5]$.
- In the original graph, there is a v_1–v_4 path whose internal vertices avoid $\{v_2, v_3, v_5\}$ and their neighbours.

Correctness

for FindJewel

Claim

If the algorithm outputs "Jewel found", then G contains a jewel.

- Algorithm outputs "Jewel found" for some choice of $v_1, \ldots, v_5$.
- BFS from v_1 reaches v_4 avoiding $N[v_2], N[v_3], N[v_5]$.
- In the original graph, there is a v_1–v_4 path whose internal vertices avoid $\{v_2, v_3, v_5\}$ and their neighbours.
- v_1–v_4 path with the edges $(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_5), (v_5, v_1)$ forms jewel.

Correctness

for FindJewel

Claim

If the algorithm outputs "Jewel found", then G contains a jewel.

- Algorithm outputs "Jewel found" for some choice of $v_1, \ldots, v_5$.
- BFS from v_1 reaches v_4 avoiding $N[v_2], N[v_3], N[v_5]$.
- In the original graph, there is a v_1–v_4 path whose internal vertices avoid $\{v_2, v_3, v_5\}$ and their neighbours.
- v_1–v_4 path with the edges $(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_5), (v_5, v_1)$ forms jewel.

Conclusion: if the algorithm outputs "Jewel found", a jewel exists.

Correctness

for FindJewel

Claim

If G contains a jewel, then the algorithm outputs "Jewel found".

- Suppose G contains a jewel with vertices $v_1, \ldots, v_5$ and P as v_1–v_4 path avoiding $N[v_2] \cup N[v_3] \cup N[v_5]$.

Correctness

for FindJewel

Claim

If G contains a jewel, then the algorithm outputs "Jewel found".

- Suppose G contains a jewel with vertices $v_1, \ldots, v_5$ and P as v_1–v_4 path avoiding $N[v_2] \cup N[v_3] \cup N[v_5]$.
- BFS from v_1 therefore reaches v_4 via P.

Correctness

for FindJewel

Claim

If G contains a jewel, then the algorithm outputs "Jewel found".

- Suppose G contains a jewel with vertices $v_1, \ldots, v_5$ and P as v_1–v_4 path avoiding $N[v_2] \cup N[v_3] \cup N[v_5]$.
- BFS from v_1 therefore reaches v_4 via P.
- Hence the algorithm detects the jewel for this choice of vertices.

Correctness

for FindJewel

Claim

If G contains a jewel, then the algorithm outputs "Jewel found".

- Suppose G contains a jewel with vertices $v_1, \ldots, v_5$ and P as v_1–v_4 path avoiding $N[v_2] \cup N[v_3] \cup N[v_5]$.
- BFS from v_1 therefore reaches v_4 via P.
- Hence the algorithm detects the jewel for this choice of vertices.

Conclusion: If G contains a jewel, then the algorithm outputs "Jewel found".

Odd hole detection: Overview

The algorithm has three parts:

- Step 1: Detect some "easy" configurations
  - Jewel ✓
  - Pyramid
  - 5-hole
- Step 2: Cleaning Step: removing C-major vertices
- Step 3: Let G' be resulting graph. Run $\mathsf{FindHole}(G')$.

Detecting "Easy" configurations : Pyramids

A pyramid has

- a base $\{b_1, b_2, b_3\} \subset V$ and apex $a \in V \setminus \{b_1, b_2, b_3\}$.
- Three internally disjoint path, each from b_1, b_2, b_3 to a.
- a is adjacent to at most one of b_1, b_2, b_3.

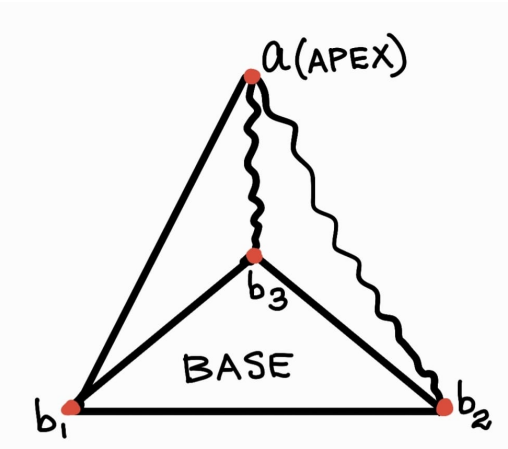


Figure: Pyramid

Detecting "Easy" configurations : Pyramids

A pyramid has

- a base $\{b_1, b_2, b_3\} \subset V$ and apex $a \in V \setminus \{b_1, b_2, b_3\}$.
- Three internally disjoint path, each from b_1, b_2, b_3 to a.
- a is adjacent to at most one of b_1, b_2, b_3.

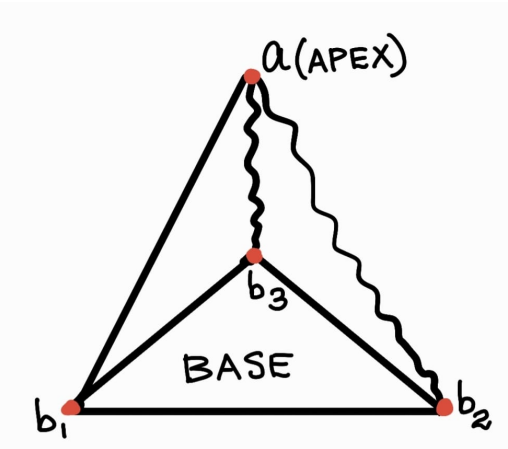


Figure: Pyramid

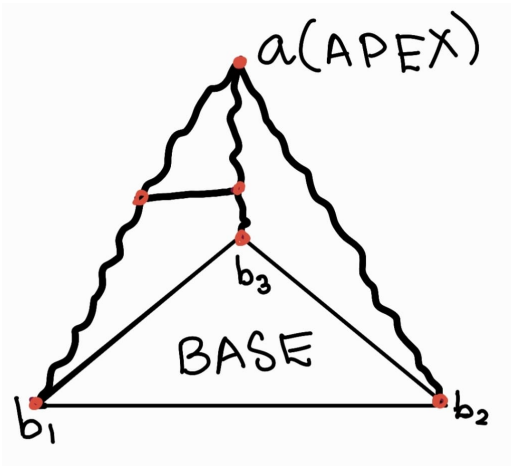


Figure: Not Pyramid

Detecting "Easy" configurations : Pyramids

A pyramid has

- a base $\{b_1, b_2, b_3\} \subset V$ and apex $a \in V \setminus \{b_1, b_2, b_3\}$.
- Three internally disjoint path, each from b_1, b_2, b_3 to a.
- a is adjacent to at most one of b_1, b_2, b_3.

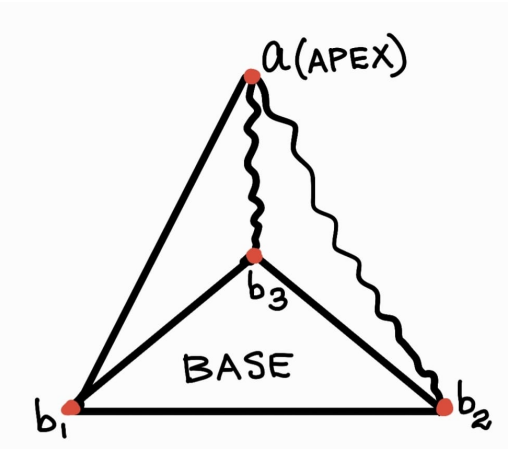


Figure: Pyramid

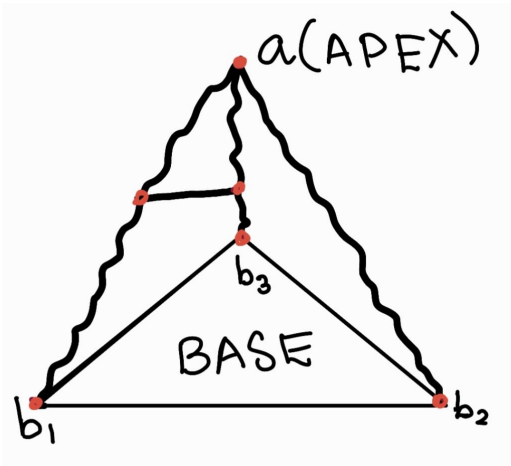


Figure: Not Pyramid

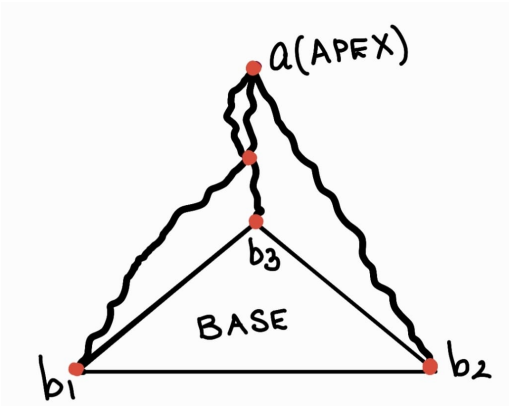


Figure: Not Pyramid

Pyramid has an odd hole

To show:

Graph containing a pyramid (as induced subgraph) has odd hole.

Pyramid has an odd hole

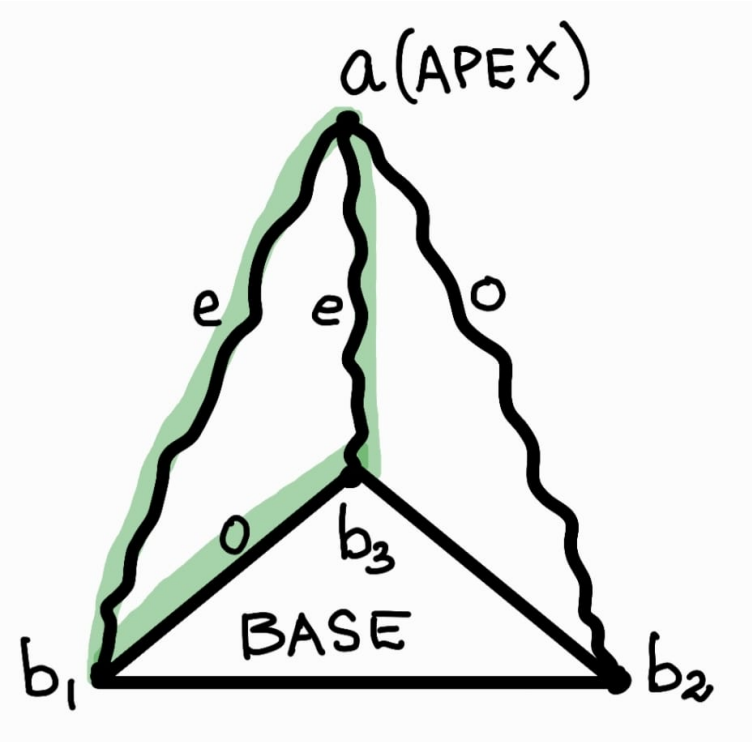
To show:

Graph containing a pyramid (as induced subgraph) has odd hole.

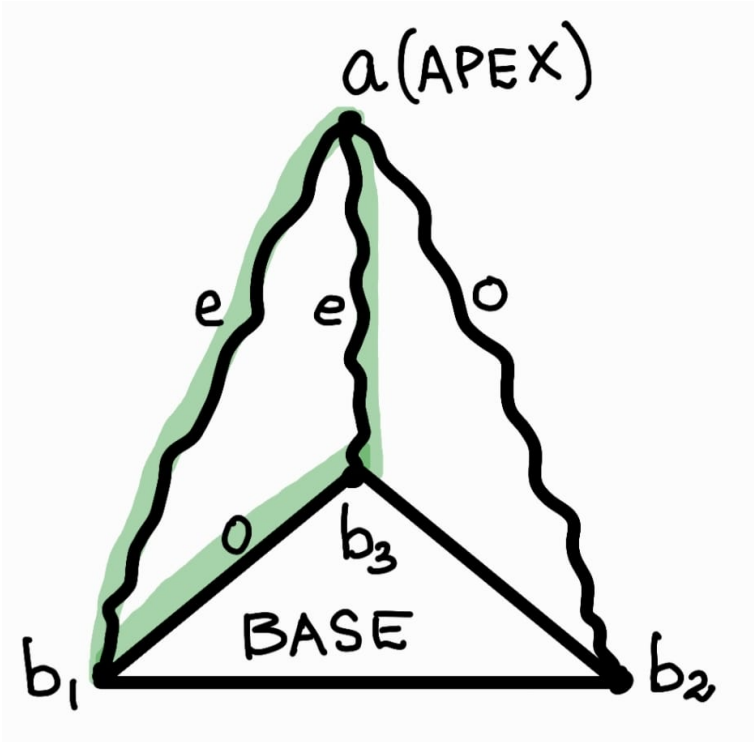
These are the following ways edges can appear on a pyramid.
Out of the three paths,

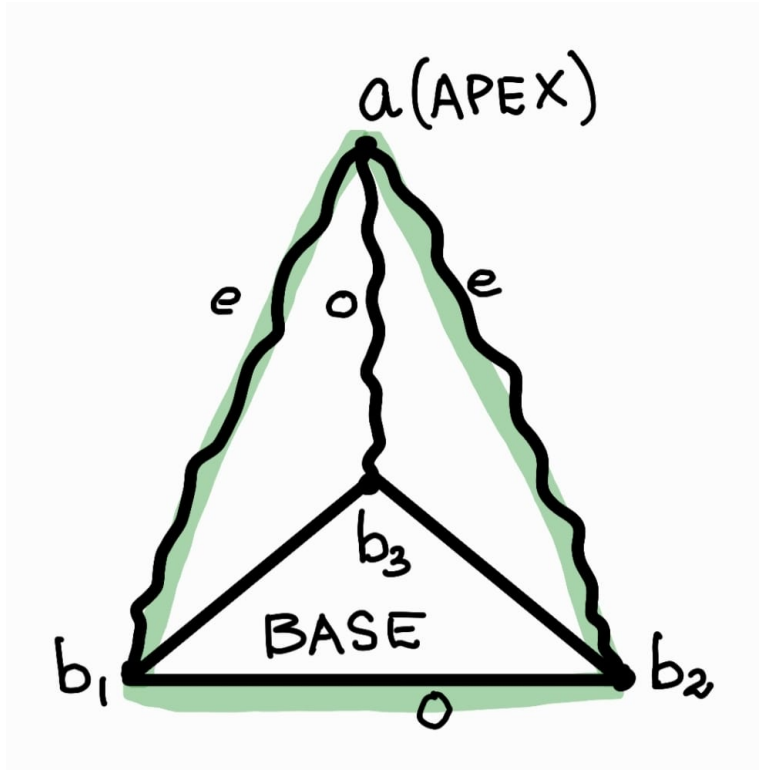
- Two paths are even and third odd.
- Two paths are odd and third even.
- All paths are even or all paths are odd.

Two paths are even and third odd.

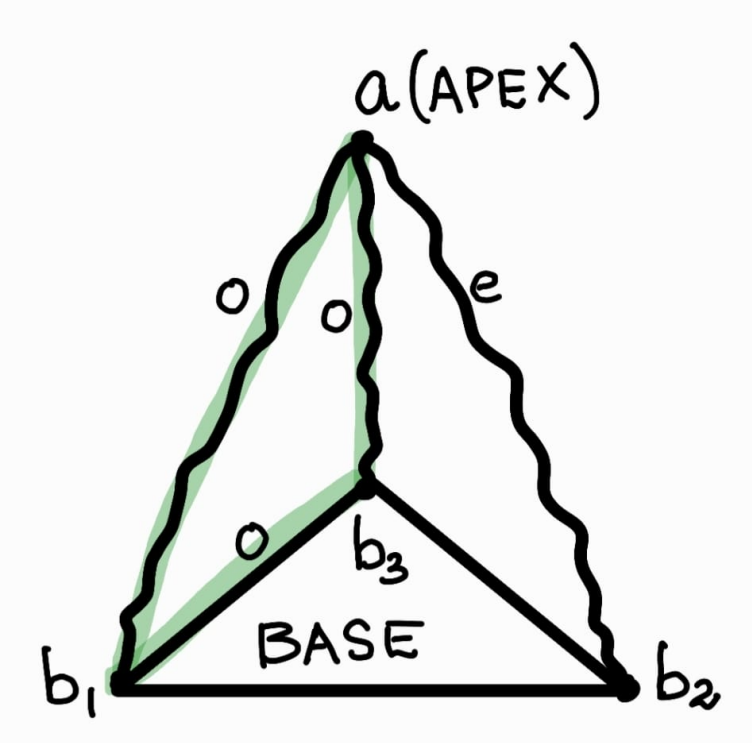


Two paths are even and third odd.

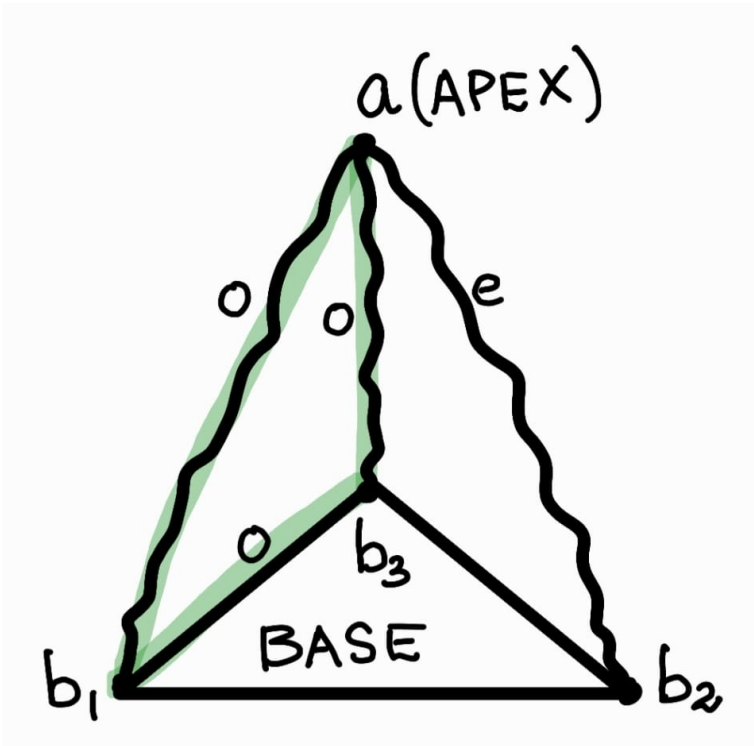


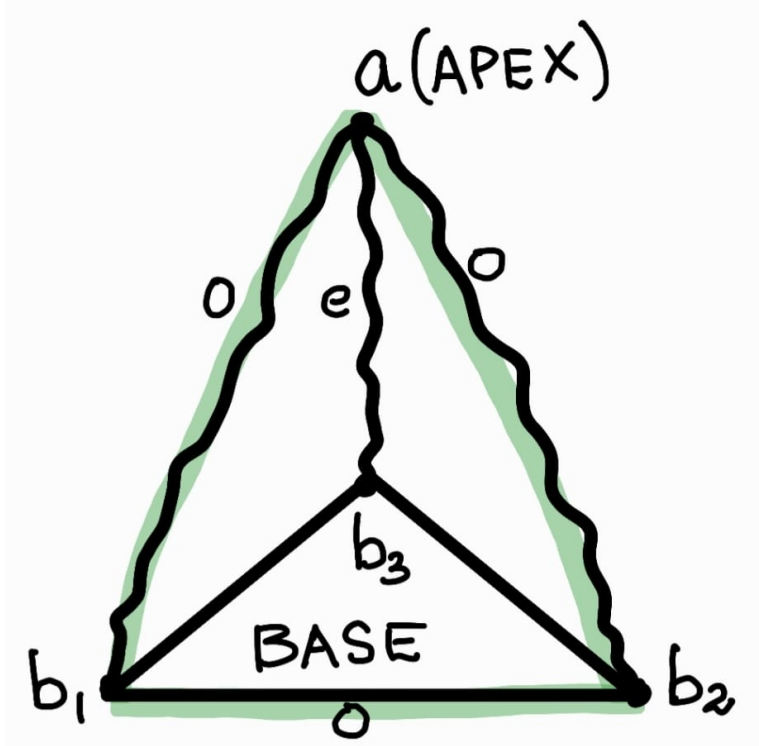


Two paths are odd and third even.

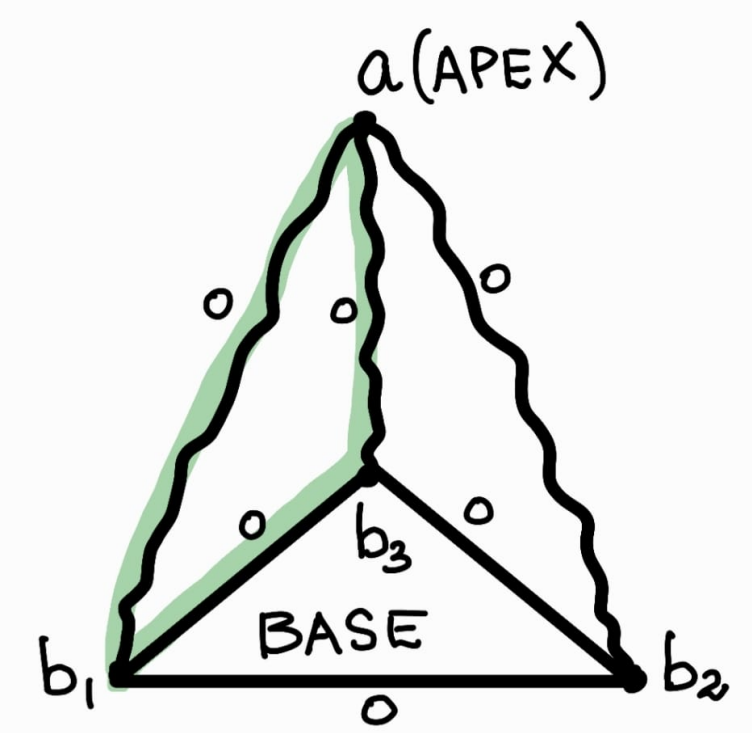


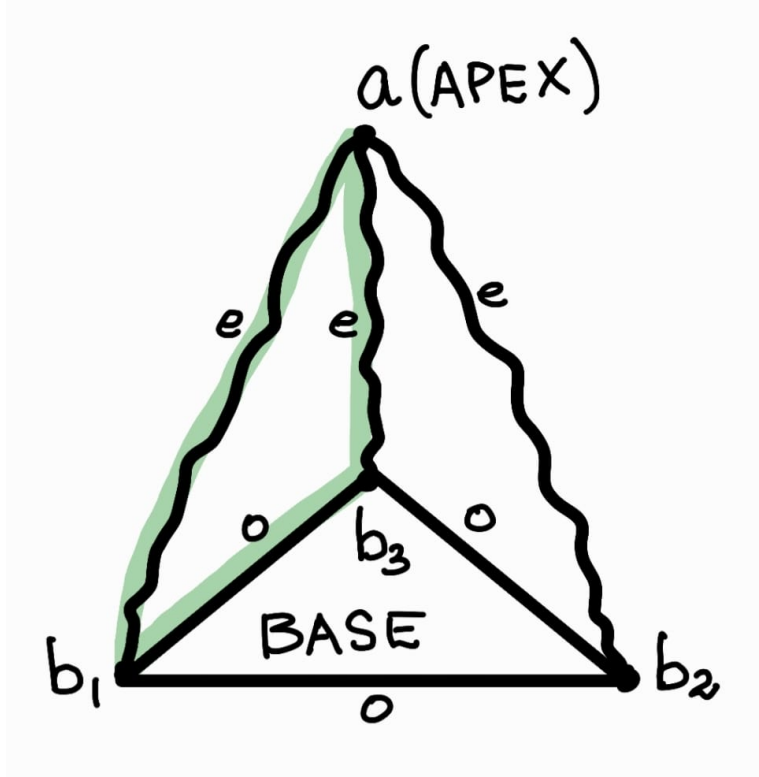
Two paths are odd and third even.



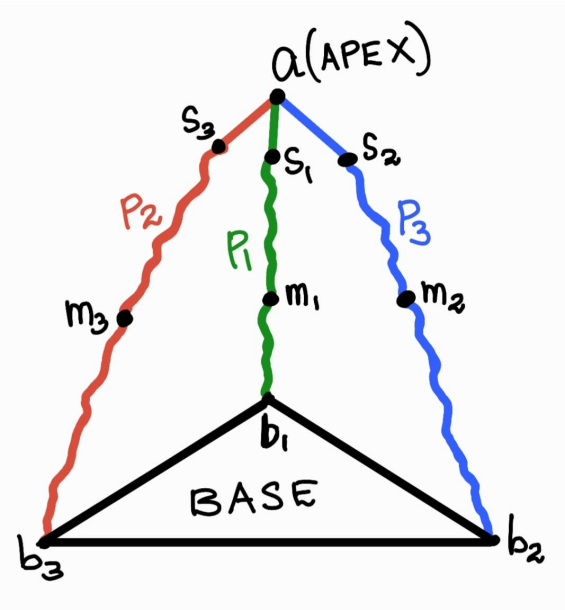


All paths are odd or all paths are even



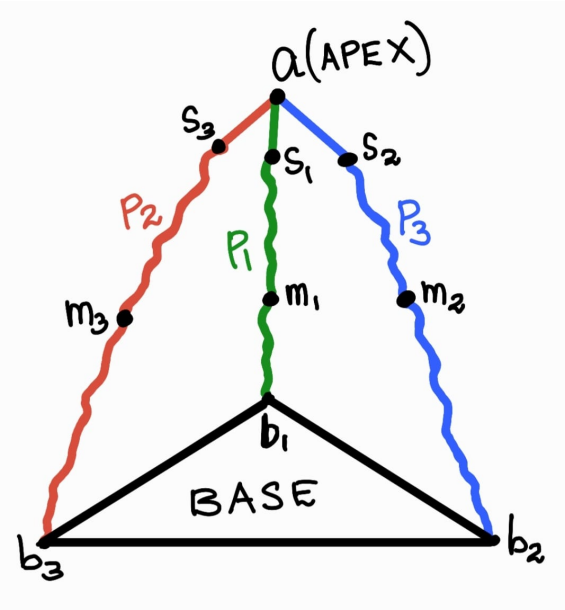


Overview of Pyramid Detection



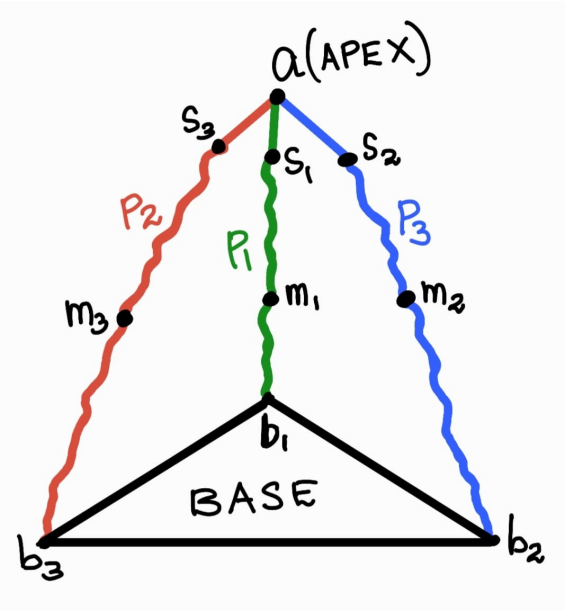
1. Enumerate base, apex and middles.

Overview of Pyramid Detection



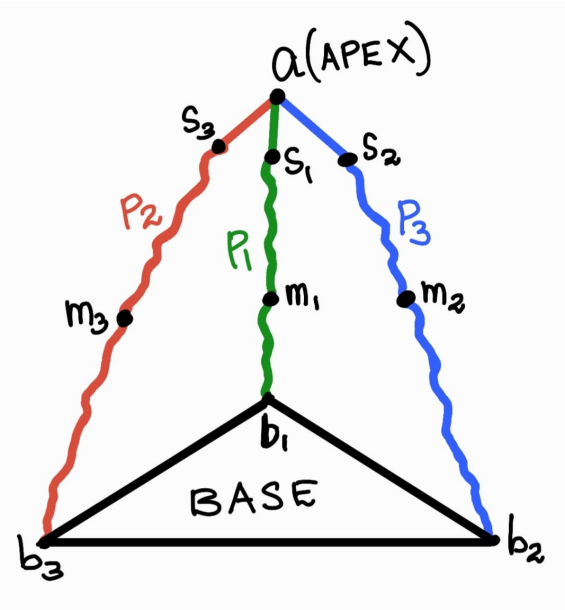
1. Enumerate base, apex and middles. S_1 $(s_1 \rightsquigarrow m_1)$ & T_1 $(m_1 \rightsquigarrow b_1) \rightarrow$ Shortest paths.

Overview of Pyramid Detection



1. Enumerate base, apex and middles. S_1 $(s_1 \rightsquigarrow m_1)$ & T_1 $(m_1 \rightsquigarrow b_1) \rightarrow$ Shortest paths.
2. Similarly: find S_2, T_2, S_3, T_3
3. Ensure no edges between paths $S_1 \cup T_1, S_2 \cup T_2, S_3 \cup T_3$.

Overview of Pyramid Detection



1. Enumerate base, apex and middles. S_1 $(s_1 \rightsquigarrow m_1)$ & T_1 $(m_1 \rightsquigarrow b_1) \rightarrow$ Shortest paths.
2. Similarly: find S_2, T_2, S_3, T_3
3. Ensure no edges between paths $S_1 \cup T_1, S_2 \cup T_2, S_3 \cup T_3$.

Why shortest path suffice?

How to check for pyramids in a graph?

foreach *Enumerate all choices of distinct* $b_1, b_2, b_3, s_1, s_2, s_3, a$ **do**
  foreach $i, j \in \{1, 2, 3\}$ *with* $i \neq j$*:* **do**
    $\{b_i, s_i\}$ is disjoint from $\{b_j, s_j\}$, and $b_i b_j$ is the unique edge between them;
    a is adjacent to s_1, s_2, s_3 and to at most one of b_1, b_2, b_3.
foreach (i, j) *with* $i \neq j$ **do**
  if $s_i = b_i$ **then**
    $P_i \leftarrow a - s_i$;
  else
    $M \leftarrow (V(G) \setminus \{b_1, b_2, b_3, s_1, s_2, s_3\}) \cup \{b_i\}$;
    pick $m \in M$ such that m is non-adjacent to both b_j and s_j;
    find S_i, the shortest path from s_i to m;
    find T_i, the shortest path from m to b_i;
    $P_i \leftarrow a - s_i - S_i - T_i - b_i$;

Why shortest path suffice?

Why this work?

- Definition:

Why shortest path suffice?

Why this work?

- Definition:
  - $V(K)$ denotes the vertex–set of pyramid K.

Why shortest path suffice?

Why this work?

- Definition:
  - $V(K)$ denotes the vertex–set of pyramid K.

Why shortest path suffice?

Why this work?

- Definition:
  - $V(K)$ denotes the vertex–set of pyramid K.
  - A pyramid K in G is **optimal** if there is no pyramid K' in G with $|V(K')| < |V(K)|$.

Why shortest path suffice?

Why this work?

- Definition:
  - $V(K)$ denotes the vertex–set of pyramid K.
  - A pyramid K in G is **optimal** if there is no pyramid K' in G with $|V(K')| < |V(K)|$.
- G has pyramid $\implies$ G has an optimal pyramid

Correctness of Algorithm

Overview

- Algo returns "Yes" $\implies$ G has a pyramid.

Correctness of Algorithm

Overview

- Algo returns "Yes" $\implies$ G has a pyramid.
- To argue : Algo returns "No" $\implies$ G has no pyramid.

Correctness of Algorithm

Overview

- Algo returns "Yes" $\implies$ G has a pyramid.
- To argue : Algo returns "No" $\implies$ G has no pyramid.
- We argue: G has a pyramid and the algorithm said "No" $\implies$ G has a smaller pyramid.
- G has a pyramid $\implies$ G has an optimal pyramid. (Seen before)

Correctness of Algorithm

Overview

- Algo returns "Yes" $\implies$ G has a pyramid.
- To argue : Algo returns "No" $\implies$ G has no pyramid.
- We argue: G has a pyramid and the algorithm said "No" $\implies$ G has a smaller pyramid.
- G has a pyramid $\implies$ G has an optimal pyramid. (Seen before)
- G has optimal pyramid + Algo says "No" $\implies$ Smaller than optimal pyramid !

Correctness of Algorithm

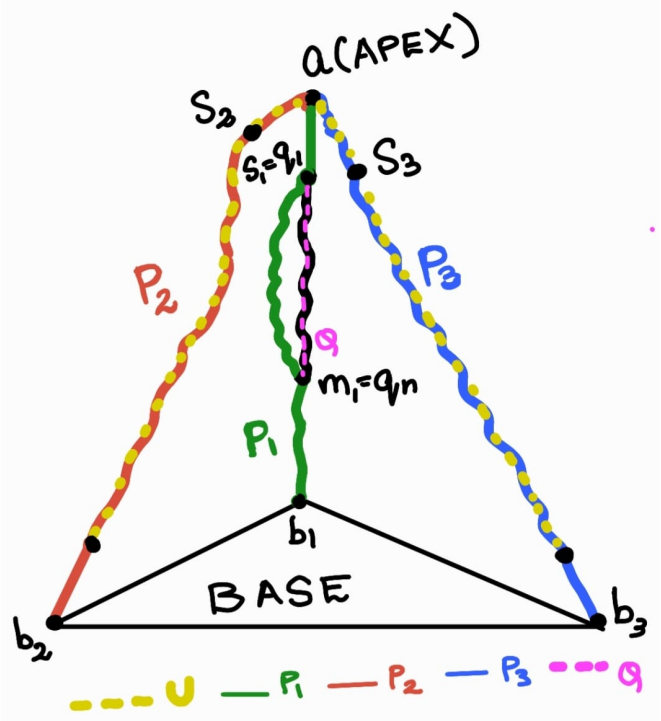
Overview

- Algo returns "Yes" $\implies$ G has a pyramid.
- To argue : Algo returns "No" $\implies$ G has no pyramid.
- We argue: G has a pyramid and the algorithm said "No" $\implies$ G has a smaller pyramid.
- G has a pyramid $\implies$ G has an optimal pyramid. (Seen before)
- G has optimal pyramid + Algo says "No" $\implies$ Smaller than optimal pyramid !
- Contradiction !

G has pyramid & algorithm said "No" $\implies$ G has a smaller pyramid.

Correctness of Algorithm

G has a pyramid and the algorithm said "No" $\implies$ G has a smaller pyramid.



- Pyramid with base b_1, b_2, b_3, apex a and three paths
  $P_1 = a - s_1 - m_1 - b_1$(not through Q),
  $P_2 = a - s_2 - u - b_2, P_3 = a - s_3 - v - b_3$.
- Let $Q = (q_1, q_2, \ldots q_n)$ be the shortest path between s_1 and m_1.
- Let $U = (P_3 \cup P_2) \setminus \{b_2, b_3\}$

Case 1: None of $q_1, \ldots q_n$ has a neighbour in U

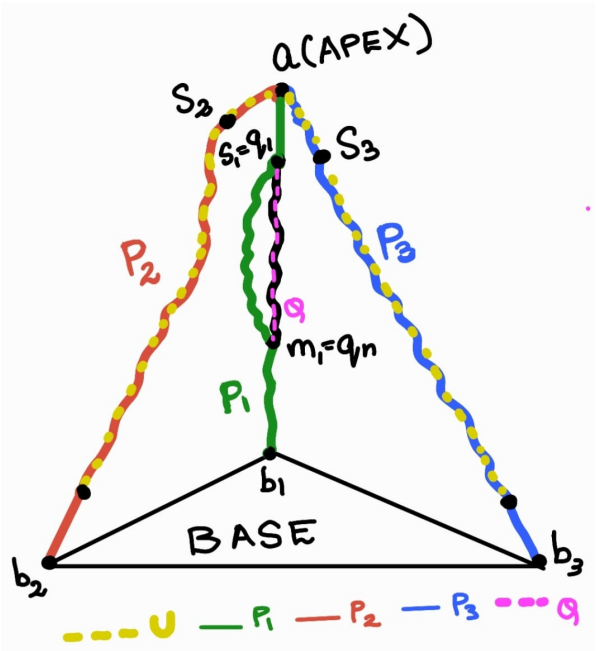


Figure: Original Pyramid

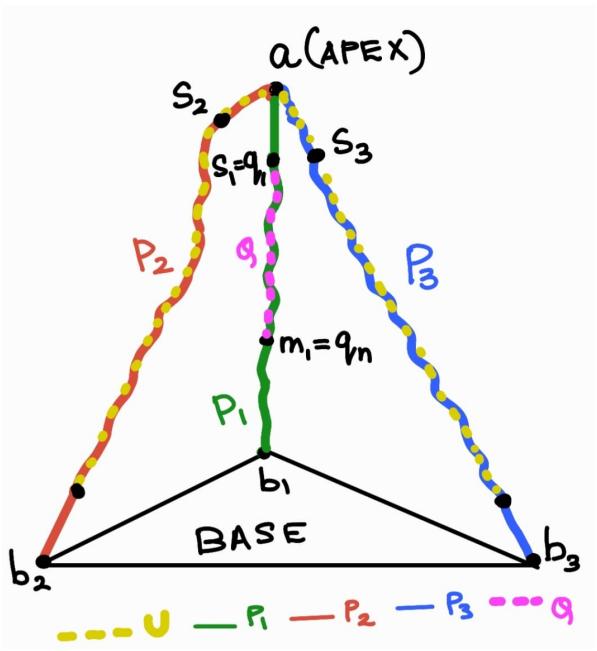


Figure: Smaller Pyramid

- Q is a shorter path, so new pyramid is smaller than the original pyramid

Case 2: q_k has unique neighbour in U where $2 \leq k \leq n-1$

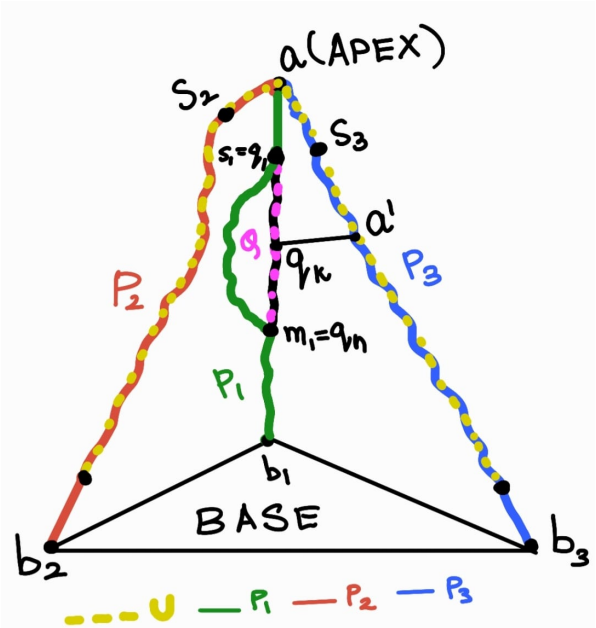


Figure: Original Pyramid

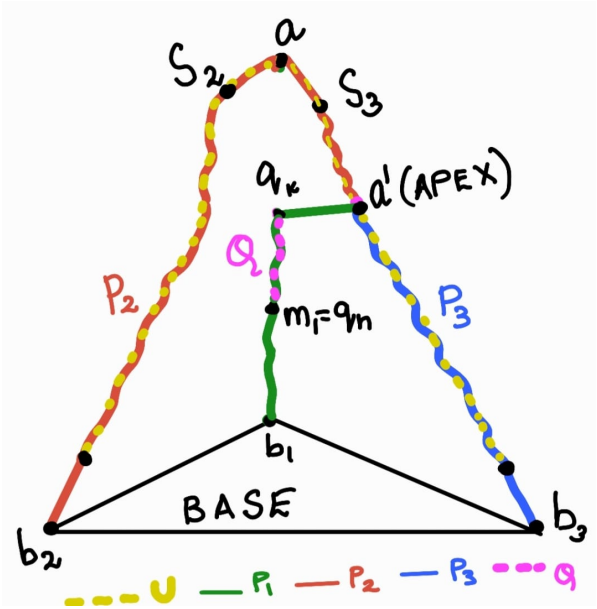


Figure: Smaller Pyramid

- P_2, P_3 is preserved but a part of P_1 is missing. So, the new pyramid is smaller than the original.
- The index k must satisfy $2 \leq k \leq n-1$.

Case 3.1: q_k has at least 2 non-adjacent neighbours that lie on the same P_i in U where $2 \leq k \leq n-1$

Pick the two neighbours that are the farthest(here, x and z)

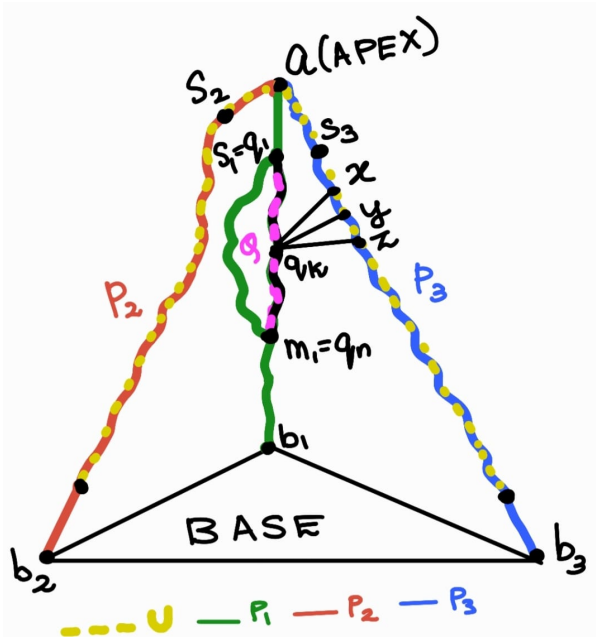


Figure: Original Pyramid

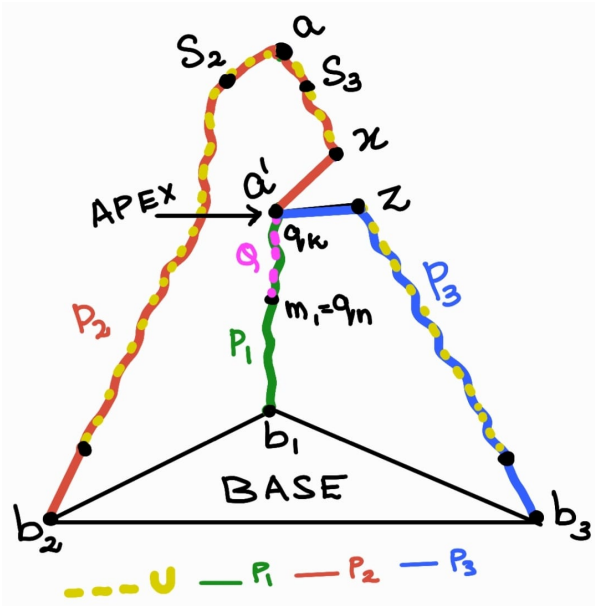


Figure: Smaller Pyramid

- P_2 is preserved but a parts of P_1, P_3 is removed. So, the new pyramid is smaller than the original.

Case 3.2: q_k has at least 2 non-adjacent neighbours that lie on P_i, P_j in U where $2 \leq k \leq n-1$ and $i \neq j$

Pick the two neighbours that are the farthest(here, x and y)

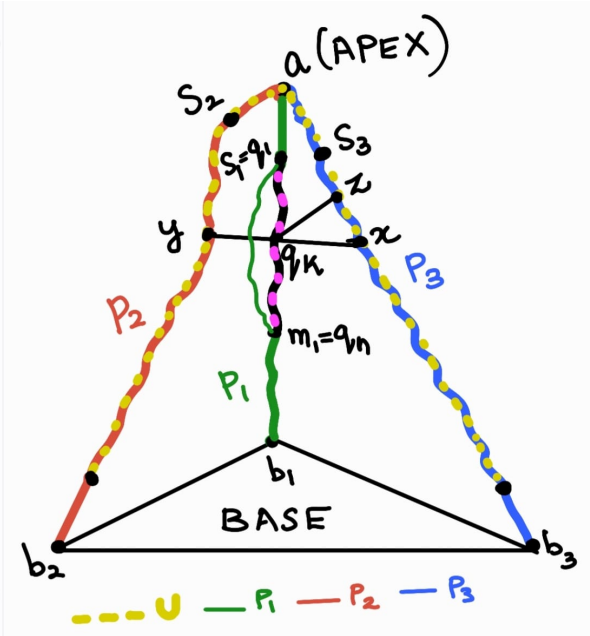


Figure: Original Pyramid

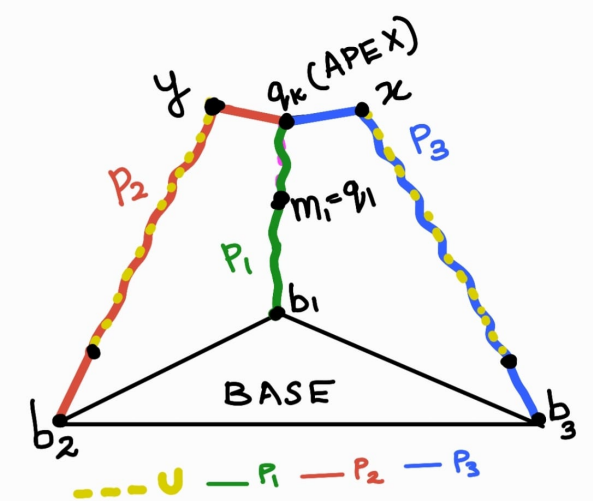


Figure: Smaller Pyramid

- Parts of P_1, P_2, P_3 is removed. So, the new pyramid is smaller than the original.

Case 4: q_k has exactly 2 adjacent neighbours in U

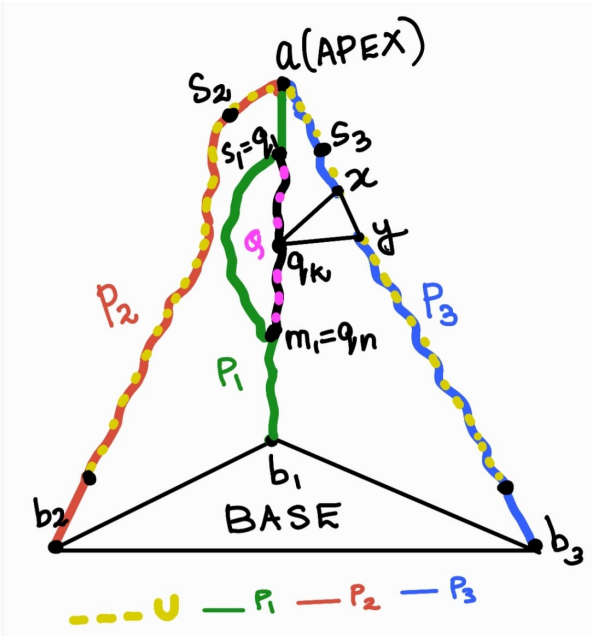


Figure: Original Pyramid

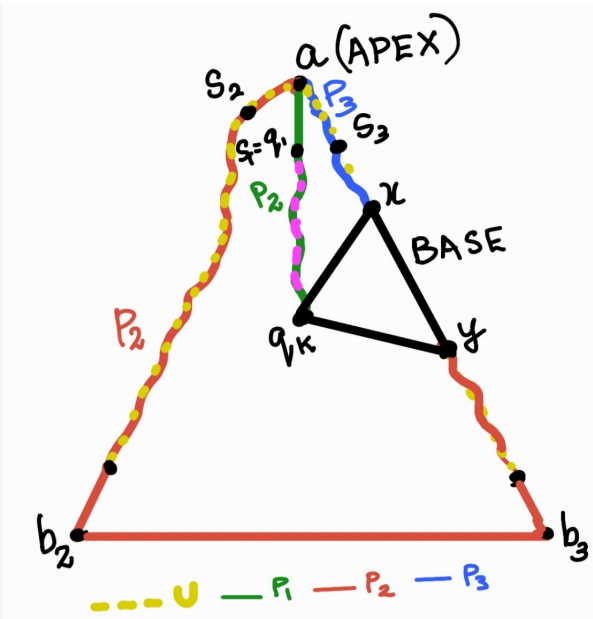
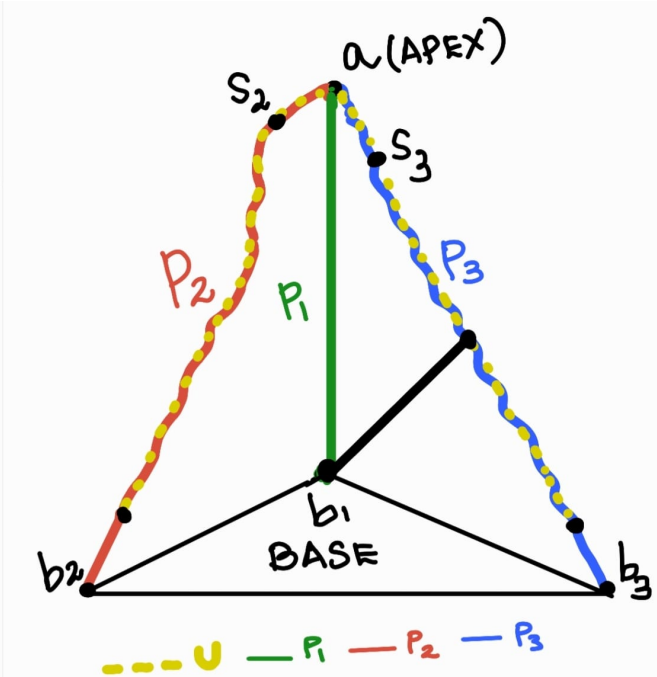


Figure: Smaller Pyramid

- P_2, P_3 is preserved but a part of P_1 is missing. So, the new pyramid is smaller than the original.

Case 5: One of the path is a single edge and an endpoint of this edge connects one of the other paths



No original pyramid

Case 6: One of the path is a single edge and there is an edge that connects the other two paths

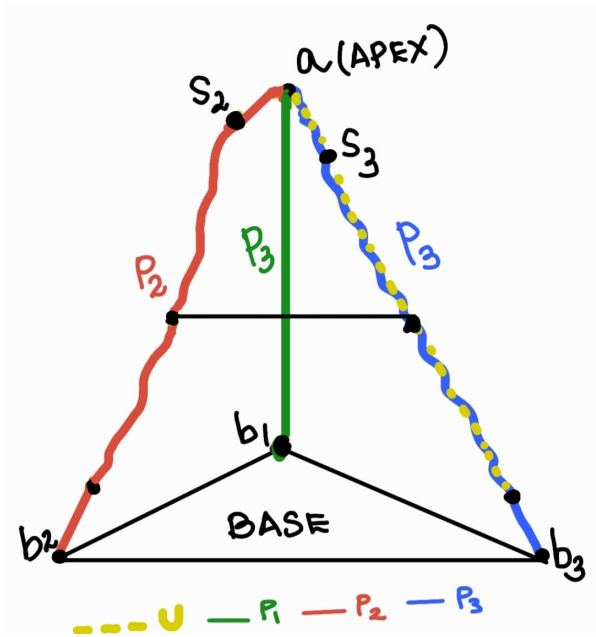


Figure: Original Pyramid

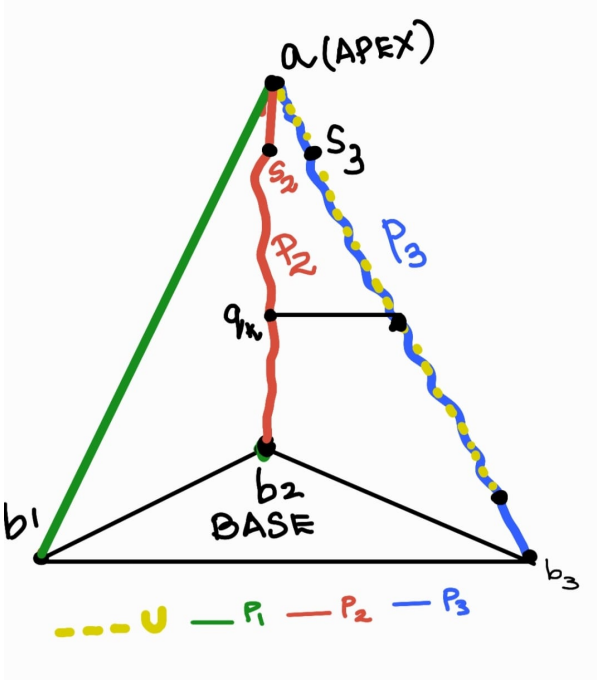


Figure: One of case 2–4

Why are the cases exhaustive?

- Say P_1, P_2, P_3 are not edges, and some vertex q_k has some number of neighbours in U.

Why are the cases exhaustive?

- Say P_1, P_2, P_3 are not edges, and some vertex q_k has some number of neighbours in U.

Why are the cases exhaustive?

- Say P_1, P_2, P_3 are not edges, and some vertex q_k has some number of neighbours in U.
  - **Case 1:** q_k has unique neighbour in U.

Why are the cases exhaustive?

- Say P_1, P_2, P_3 are not edges, and some vertex q_k has some number of neighbours in U.
  - **Case 1:** q_k has unique neighbour in U.
  - **Case 2:** q_k has at least 2 non-adjacent neighbours in U.

Why are the cases exhaustive?

- Say P_1, P_2, P_3 are not edges, and some vertex q_k has some number of neighbours in U.
  - **Case 1:** q_k has unique neighbour in U.
  - **Case 2:** q_k has at least 2 non-adjacent neighbours in U.
  - **Case 3:** q_k has exactly 2 neighbours that are adjacent in U.
  - One more case!

Why are the cases exhaustive?

- Say P_1, P_2, P_3 are not edges, and some vertex q_k has some number of neighbours in U.
  - **Case 1:** q_k has unique neighbour in U.
  - **Case 2:** q_k has at least 2 non-adjacent neighbours in U.
  - **Case 3:** q_k has exactly 2 neighbours that are adjacent in U.
  - One more case! Reduces to case 2

Why are the cases exhaustive?

- Say P_1, P_2, P_3 are not edges, and some vertex q_k has some number of neighbours in U.
  - **Case 1:** q_k has unique neighbour in U.
  - **Case 2:** q_k has at least 2 non-adjacent neighbours in U.
  - **Case 3:** q_k has exactly 2 neighbours that are adjacent in U.
  - One more case! Reduces to case 2
- **Case 4:** One of the path is a single edge and an endpoint of this edge connects one of the other paths. Then, there is no original pyramid.

Why are the cases exhaustive?

- Say P_1, P_2, P_3 are not edges, and some vertex q_k has some number of neighbours in U.
  - **Case 1:** q_k has unique neighbour in U.
  - **Case 2:** q_k has at least 2 non-adjacent neighbours in U.
  - **Case 3:** q_k has exactly 2 neighbours that are adjacent in U.
  - One more case! Reduces to case 2
- **Case 4:** One of the path is a single edge and an endpoint of this edge connects one of the other paths. Then, there is no original pyramid.
- **Case 5:** One of the path is a single edge and and the other two paths are connected by an edge. Then, this will come under one of case 2–4.

Why are the cases exhaustive?

- Say P_1, P_2, P_3 are not edges, and some vertex q_k has some number of neighbours in U.
  - **Case 1:** q_k has unique neighbour in U.
  - **Case 2:** q_k has at least 2 non-adjacent neighbours in U.
  - **Case 3:** q_k has exactly 2 neighbours that are adjacent in U.
  - One more case! Reduces to case 2
- **Case 4:** One of the path is a single edge and an endpoint of this edge connects one of the other paths. Then, there is no original pyramid.
- **Case 5:** One of the path is a single edge and and the other two paths are connected by an edge. Then, this will come under one of case 2–4.
- **Remark:** The same cases apply when the shortest path Q appears from m_1 to b_1.

Odd hole detection: Overview

The algorithm has three parts:

- Step 1: Detect some "easy" configurations
  - Jewel ✓
  - Pyramid ✓
  - 5-hole

Odd hole detection: Overview

The algorithm has three parts:

- Step 1: Detect some "easy" configurations
  - Jewel ✓
  - Pyramid ✓
  - 5-hole ✓
- Step 2: Cleaning Step - removing C-major vertices
- Step 3: Let G' be resulting graph. Run $\mathsf{FindHole}(G')$.

Definitions

- C is an odd hole.
- A vertex $v \in V(G) \setminus V(C)$ is C**-major** if there is no three-vertex path of C containing all the neighbours of v in $V(C)$.

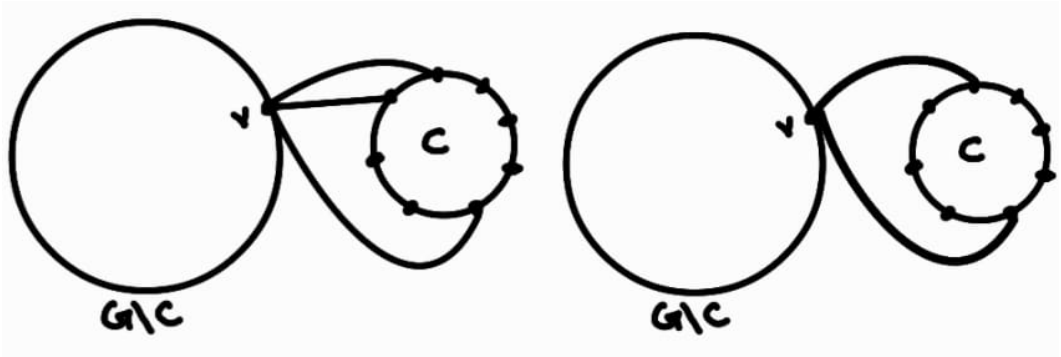


Figure: C-major

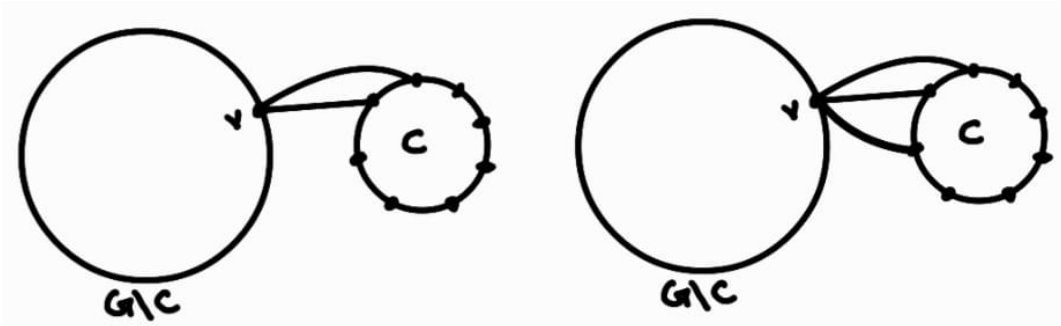
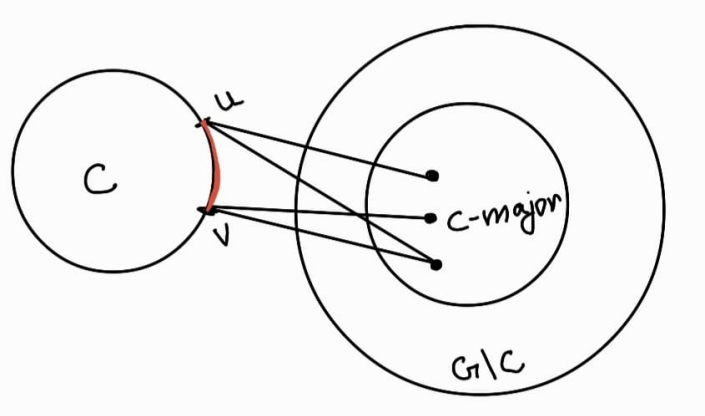


Figure: not C-major

Definitions

- **A heavy edge** is an edge (u, v) of C such that every C-major vertex is adjacent to one of u, v.



- X**-heavy edge:** If $X \subseteq V(G)$, we say an edge (u, v) of G is X-heavy if $u, v \notin X$, and every vertex of X is adjacent to at least one of u, v.
- A hole C is **clean** if no vertices of G are C-major.

Identifying and removing C-major vertices

Why are C-major vertices problematic?
Can form additional cycles by connecting distant points on C.

Identifying and removing C-major vertices

Why are C-major vertices problematic?
Can form additional cycles by connecting distant points on C.
Removing C-major vertices:

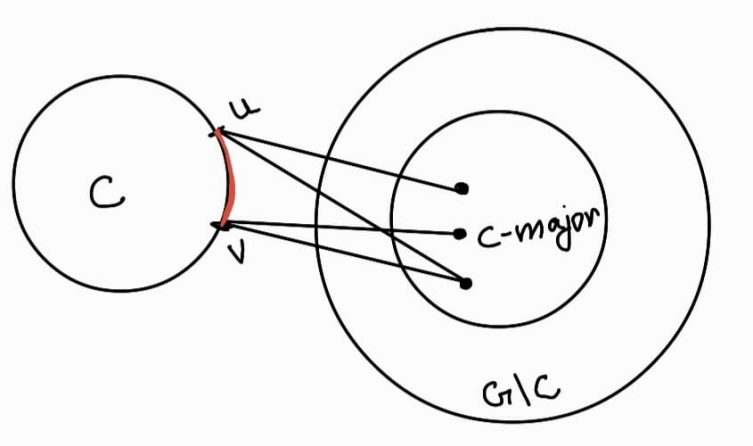
- Step 1: Finding heavy edge in C

Identifying and removing C-major vertices

Why are C-major vertices problematic?
Can form additional cycles by connecting distant points on C.
Removing C-major vertices:

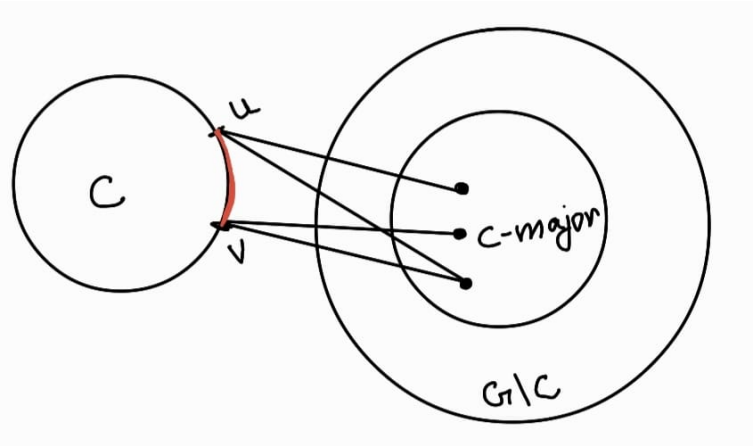
- Step 1: Finding heavy edge in C
- Step 2: "Cleaning" the graph further

Step 1: Finding heavy edge in C



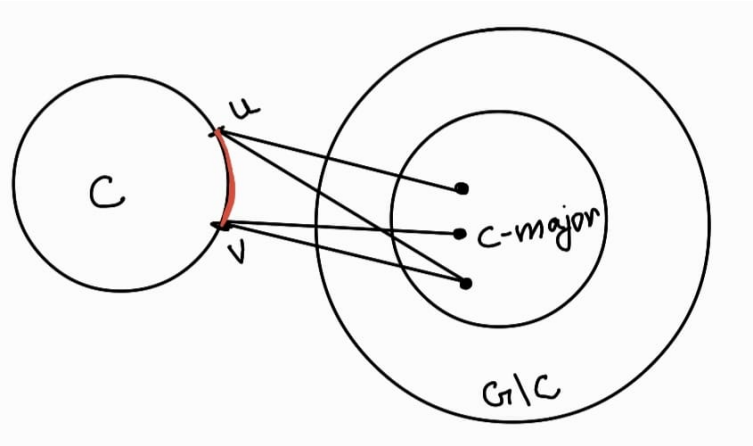
- Enumerate: $c_1 - c_2 - c_3 - c_4$ induced path.

Step 1: Finding heavy edge in C



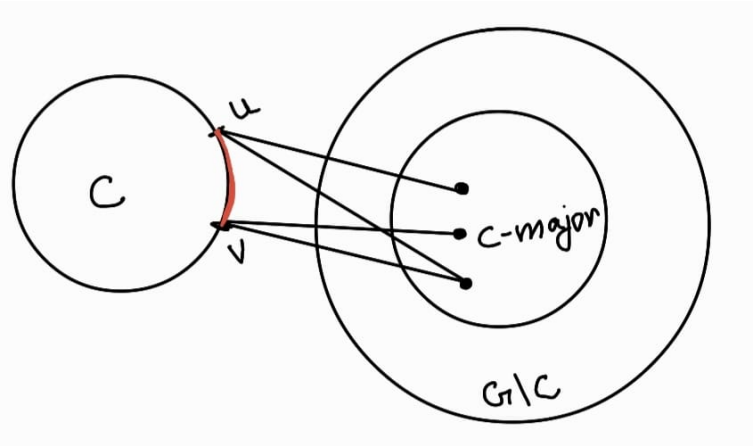
- Enumerate: $c_1 - c_2 - c_3 - c_4$ induced path.
- Remove vertices adjacent c_2 or c_3 (except c_1, c_4).

Step 1: Finding heavy edge in C



- Enumerate: $c_1 - c_2 - c_3 - c_4$ induced path.
- Remove vertices adjacent c_2 or c_3 (except c_1, c_4).
- Run FindHole on new graph and return True if it returns True. Otherwise, "clean" the graph.

Step 1: Finding heavy edge in C



- Enumerate: $c_1 - c_2 - c_3 - c_4$ induced path.
- Remove vertices adjacent c_2 or c_3 (except c_1, c_4).
- Run FindHole on new graph and return True if it returns True. Otherwise, "clean" the graph.

If $c_2 - c_3$ is a heavy edge, removing its neighbours will "clean" the hole and FindHole returns True on the cleaned graph.

Finding other C-major vertices

Theorem 3.4[1] rephrased.

Let G be a graph containing no jewel or pyramid or 5-hole, and let C be a shortest odd hole in G. Let X_2 be a set of C-major vertices, and let $x \in X_2$ be nonadjacent to all other members of X_2. Then, there is a X_2-heavy edge in C.

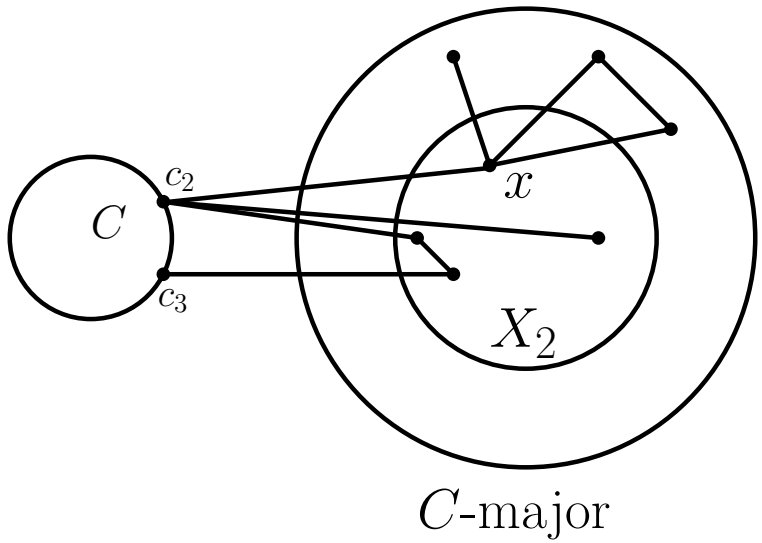


Figure: $c_2 - c_3$ is the X_2-heavy edge.

[1] Maria Chudnovsky, Alex Scott, Paul Seymour, and Sophie Spirkl. 2020. Detecting an Odd Hole. J. ACM 67, 1, Article 5 (February 2020)

Theorem 3.4[2] rephrased.

Let G be a graph containing no jewel or pyramid or 5-hole, and let C be a shortest odd hole in G. Let X_2 be a set of C-major vertices, and let $x \in X_2$ be nonadjacent to all other members of X_2. Then, there is a X_2-heavy edge in C.

- Pick $x \in V(G)$ and induced path $c_1 - c_2 - c_3 - c_4$ in G.

[2] Maria Chudnovsky, Alex Scott, Paul Seymour, and Sophie Spirkl. 2020. Detecting an Odd Hole. J. ACM 67, 1, Article 5 (February 2020)

Theorem 3.4[2] rephrased.

Let G be a graph containing no jewel or pyramid or 5-hole, and let C be a shortest odd hole in G. Let X_2 be a set of C-major vertices, and let $x \in X_2$ be nonadjacent to all other members of X_2. Then, there is a X_2-heavy edge in C.

- Pick $x \in V(G)$ and induced path $c_1 - c_2 - c_3 - c_4$ in G.
- X_2: set of C-major vertices non-adjacent to x.

[2] Maria Chudnovsky, Alex Scott, Paul Seymour, and Sophie Spirkl. 2020. Detecting an Odd Hole. J. ACM 67, 1, Article 5 (February 2020)

Theorem 3.4[2] rephrased.

Let G be a graph containing no jewel or pyramid or 5-hole, and let C be a shortest odd hole in G. Let X_2 be a set of C-major vertices, and let $x \in X_2$ be nonadjacent to all other members of X_2. Then, there is a X_2-heavy edge in C.

- Pick $x \in V(G)$ and induced path $c_1 - c_2 - c_3 - c_4$ in G.
- X_2: set of C-major vertices non-adjacent to x.
- X_2-heavy edge exists $(c_2 - c_3)$ by Theorem 3.4.

[2] Maria Chudnovsky, Alex Scott, Paul Seymour, and Sophie Spirkl. 2020. Detecting an Odd Hole. J. ACM 67, 1, Article 5 (February 2020)

Theorem 3.4[2] rephrased.

Let G be a graph containing no jewel or pyramid or 5-hole, and let C be a shortest odd hole in G. Let X_2 be a set of C-major vertices, and let $x \in X_2$ be nonadjacent to all other members of X_2. Then, there is a X_2-heavy edge in C.

- Pick $x \in V(G)$ and induced path $c_1 - c_2 - c_3 - c_4$ in G.
- X_2: set of C-major vertices non-adjacent to x.
- X_2-heavy edge exists $(c_2 - c_3)$ by Theorem 3.4.
- Remove vertices adjacent to one of $c_2 - c_3$ edge except c_1 and c_4.

[2] Maria Chudnovsky, Alex Scott, Paul Seymour, and Sophie Spirkl. 2020. Detecting an Odd Hole. J. ACM 67, 1, Article 5 (February 2020)

Theorem 3.4[2] rephrased.

Let G be a graph containing no jewel or pyramid or 5-hole, and let C be a shortest odd hole in G. Let X_2 be a set of C-major vertices, and let $x \in X_2$ be nonadjacent to all other members of X_2. Then, there is a X_2-heavy edge in C.

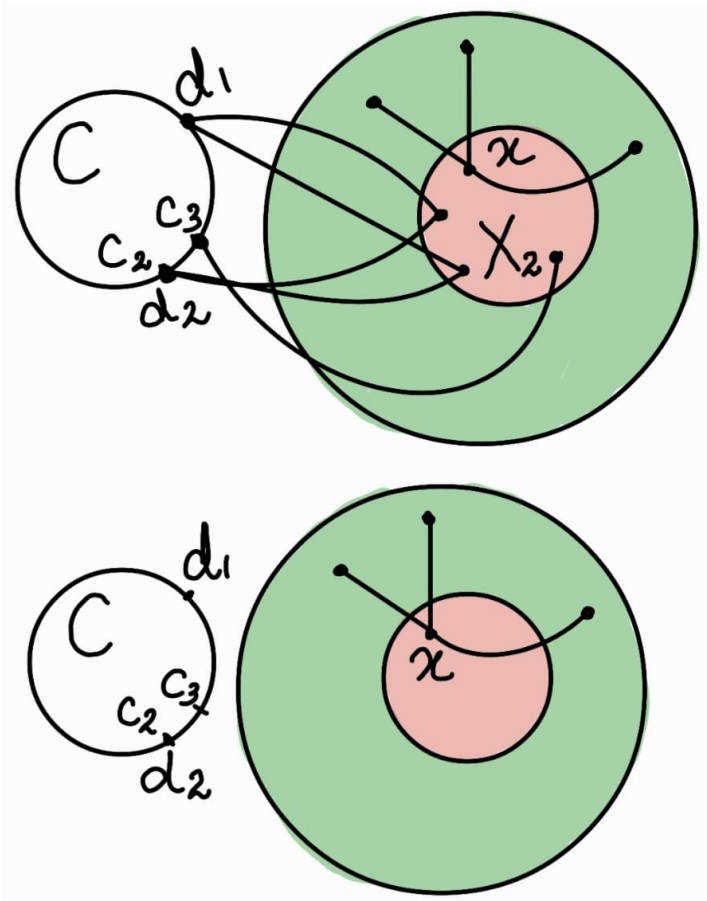
- Pick $x \in V(G)$ and induced path $c_1 - c_2 - c_3 - c_4$ in G.
- X_2: set of C-major vertices non-adjacent to x.
- X_2-heavy edge exists $(c_2 - c_3)$ by Theorem 3.4.
- Remove vertices adjacent to one of $c_2 - c_3$ edge except c_1 and c_4.

All vertices that are C-major and not adjacent to x (including vertices that are C-major but without any neighbour in the C-major set) are removed by Theorem 3.4.

[2] Maria Chudnovsky, Alex Scott, Paul Seymour, and Sophie Spirkl. 2020. Detecting an Odd Hole. J. ACM 67, 1, Article 5 (February 2020)

Removing C-major vertices (Non-adjacent to x)

- Remove vertices non-adjacent to x except $c_1, c_2, c_3, c_4, d_1, d_2$ from G to get G'.



G to G'

C-major vertices appear as

- Heavy edge

C-major vertices appear as

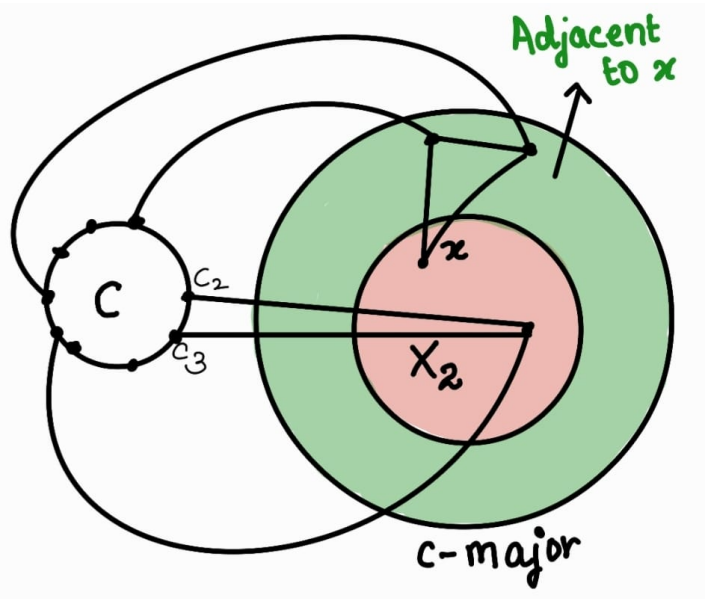
- Heavy edge ✓

C-major vertices appear as

- Heavy edge ✓
- Non-adjacent to x

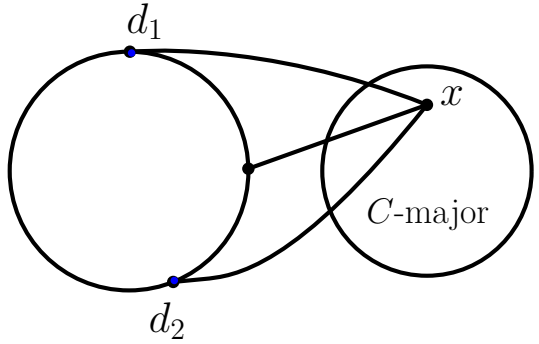
C-major vertices appear as

- Heavy edge ✓
- Non-adjacent to x ✓
- Adjacent to x



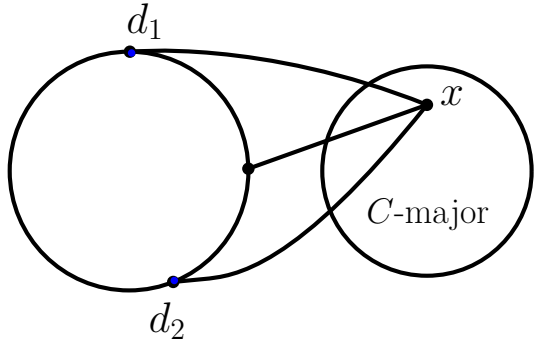
Finding C-major vertices adjacent to x

- Enumerate d_1, d_2 adjacent to x, farthest apart on C.

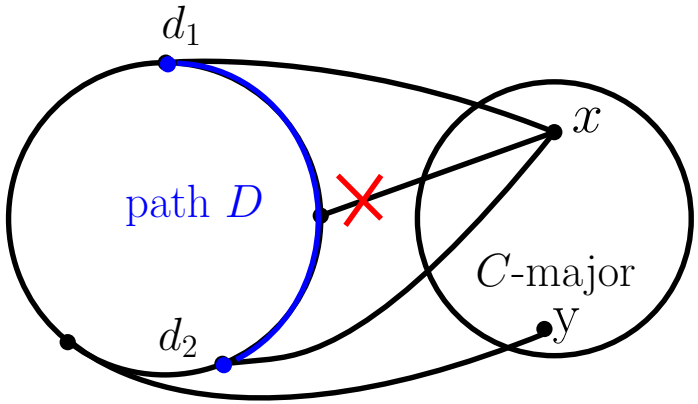


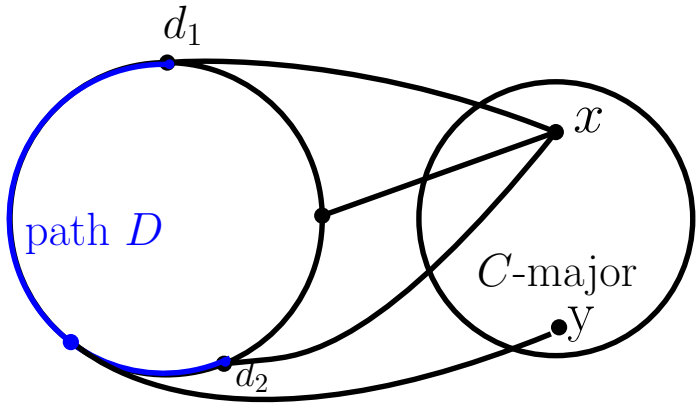
Finding C-major vertices adjacent to x

- Enumerate d_1, d_2 adjacent to x, farthest apart on C.



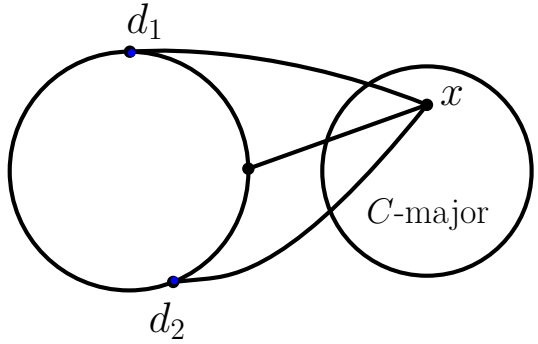
- $D : d_1 - d_2$ path on C with $N[x]$ not in interior of D and all C-major vertices have at least one neighbour in D.



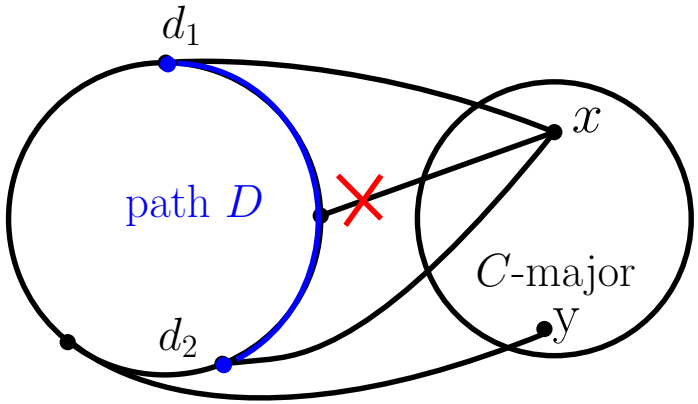


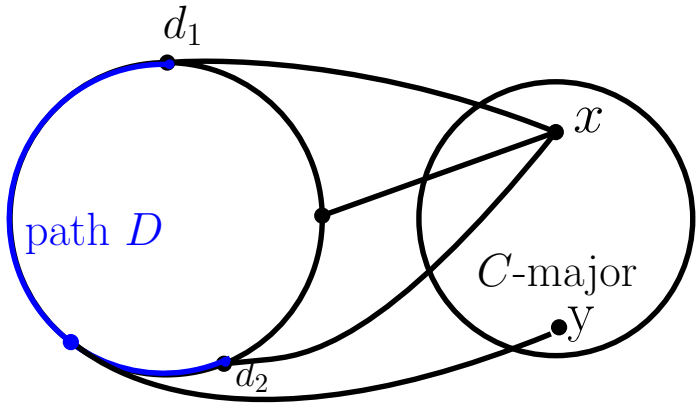
Finding C-major vertices adjacent to x

- Enumerate d_1, d_2 adjacent to x, farthest apart on C.



- $D : d_1 - d_2$ path on C with $N[x]$ not in interior of D and all C-major vertices have at least one neighbour in D.

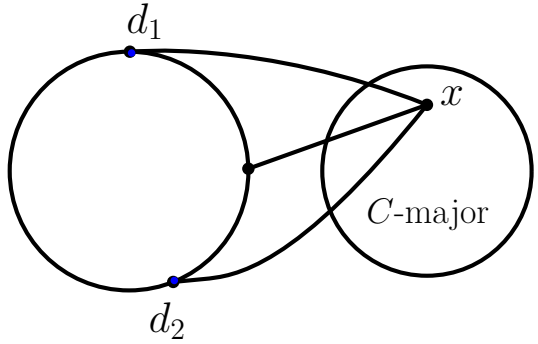




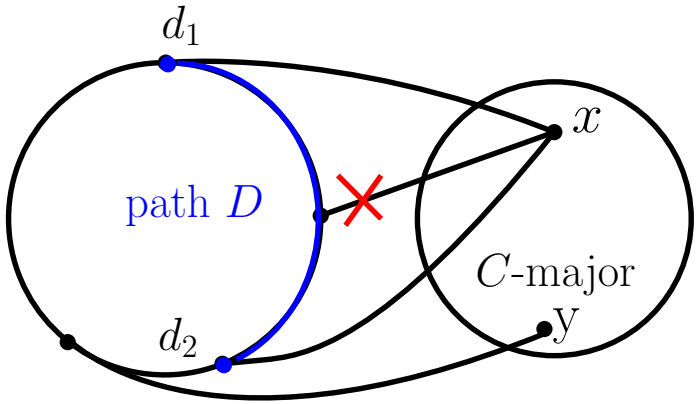
- If current choice of x does not satisfy, move to the next tuple and restart cleaning.

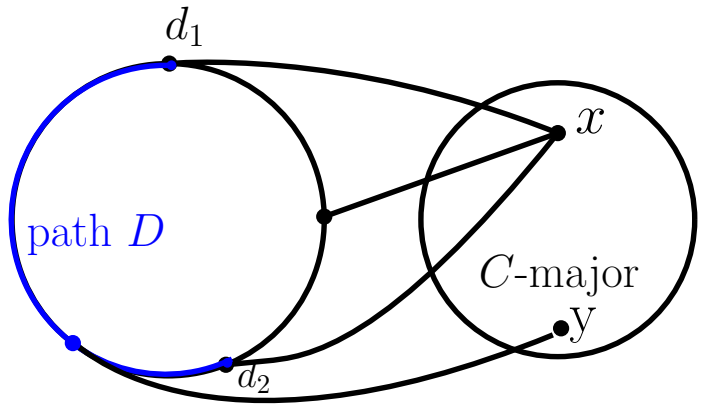
Finding C-major vertices adjacent to x

- Enumerate d_1, d_2 adjacent to x, farthest apart on C.



- $D : d_1 - d_2$ path on C with $N[x]$ not in interior of D and all C-major vertices have at least one neighbour in D.





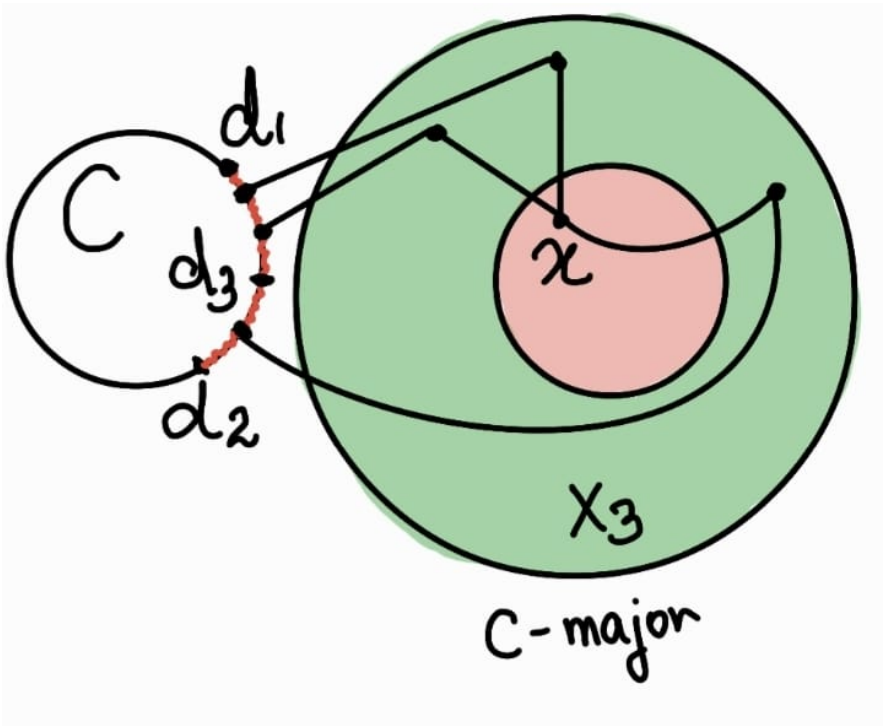
- If current choice of x does not satisfy, move to the next tuple and restart cleaning.Always possible to find such an x.
- Enumerate d_3, middle vertex between d_1, d_2.

Removing C-major vertices (Adjacent to x)

- Y: set of vertices in G' non-adjacent to x in G and ensure $d_3 \in Y$.
- Compute shortest paths from d_1 to d_3 and d_3 to d_2 in the induced graph $G[Y \cup \{d_1, d_2\}]$.

Removing C-major vertices (Adjacent to x)

- Y: set of vertices in G' non-adjacent to x in G and ensure $d_3 \in Y$.
- Compute shortest paths from d_1 to d_3 and d_3 to d_2 in the induced graph $G[Y \cup \{d_1, d_2\}]$.
- X_3 : all vertices in G' that are not in any shortest paths, but are adjacent to the shortest paths and different from x.



C-major vertices appear as

- Heavy edge

C-major vertices appear as

- Heavy edge ✓

C-major vertices appear as

- Heavy edge ✓
- Non-adjacent to x

C-major vertices appear as

- Heavy edge ✓
- Non-adjacent to x ✓
- Adjacent to x ✓

Found all C-major vertices.

Finding odd hole

- Remove vertices non-adjacent to x, adjacent to x and x from G to get G''.

Finding odd hole

- Remove vertices non-adjacent to x, adjacent to x and x from G to get G''.
- Run FindHole(G'').

Finding odd hole

- Remove vertices non-adjacent to x, adjacent to x and x from G to get G''.
- Run FindHole(G'').
- Declare not Berge if FindHole returns True. Otherwise, if after enumerating all possible $x, d_1, d_2, d_3, c_1, c_2, c_3, c_4$, FindHole, if always returned False, declare that the graph is Berge.