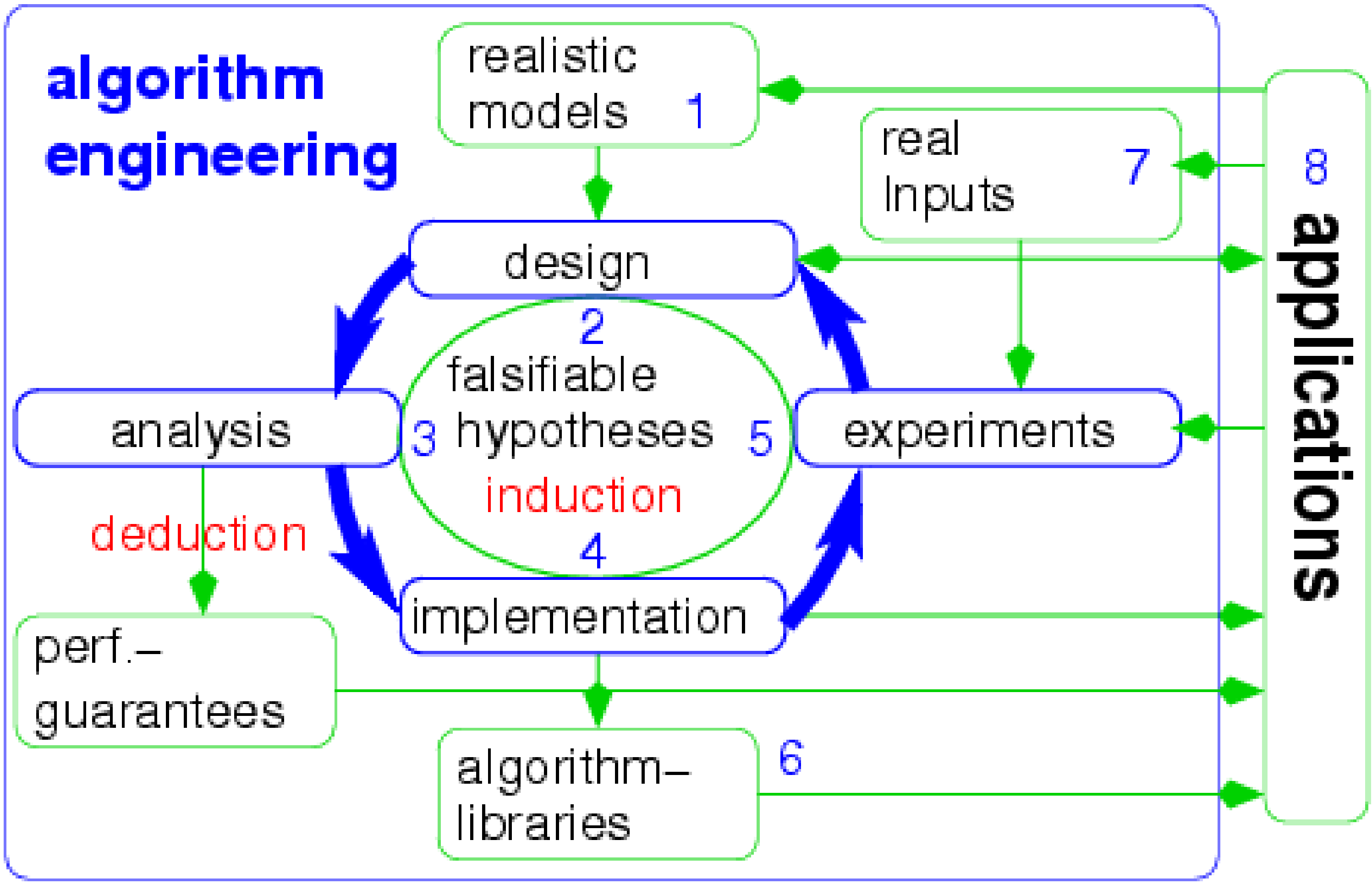
# Experimental Study of Directed Feedback Vertex Set Problem

Xi Wu

With Rudolf Fleischer and Liwei Yuan

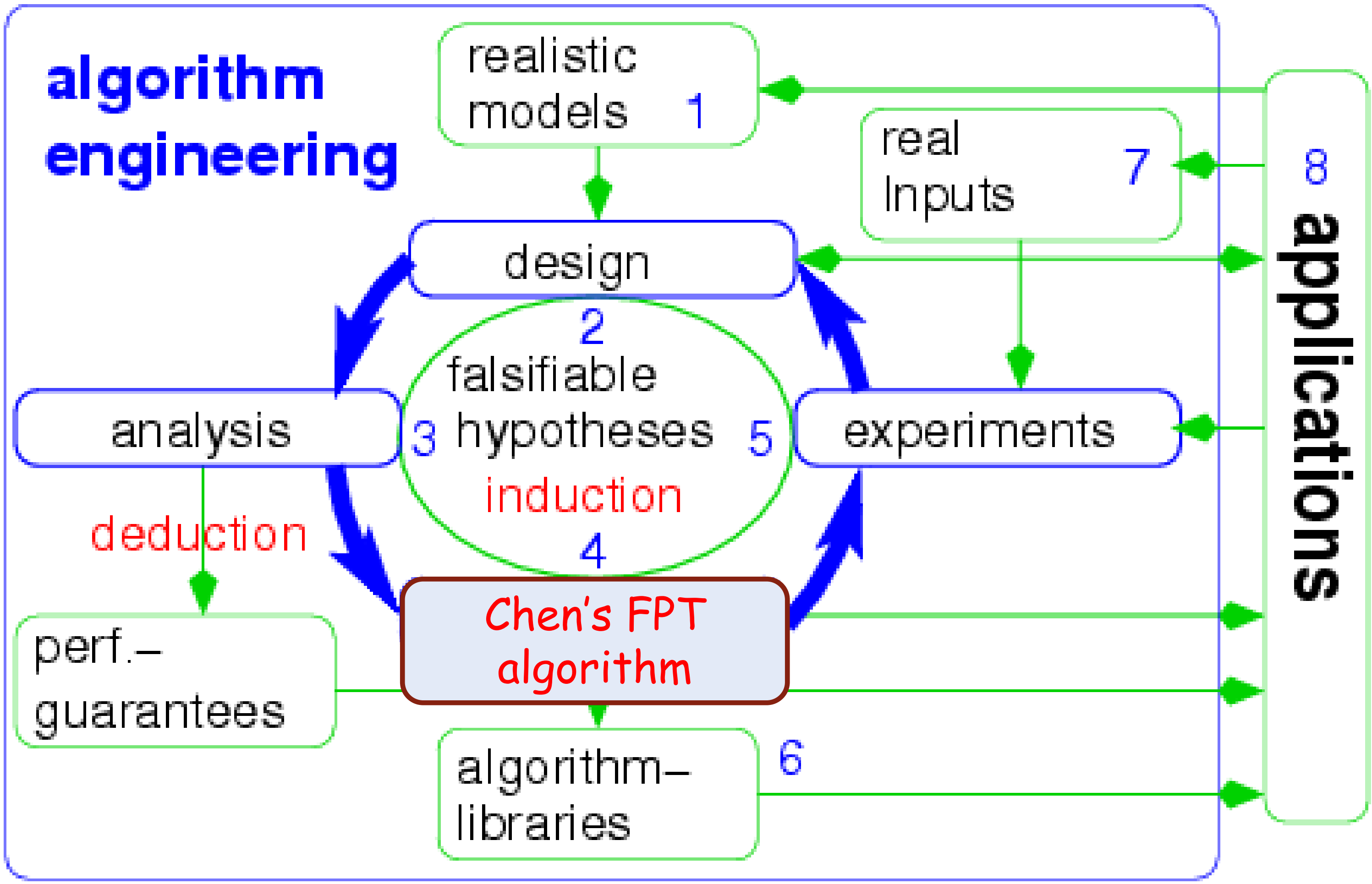
Fudan University, Shanghai

# Algorithm Engineering for DFVS



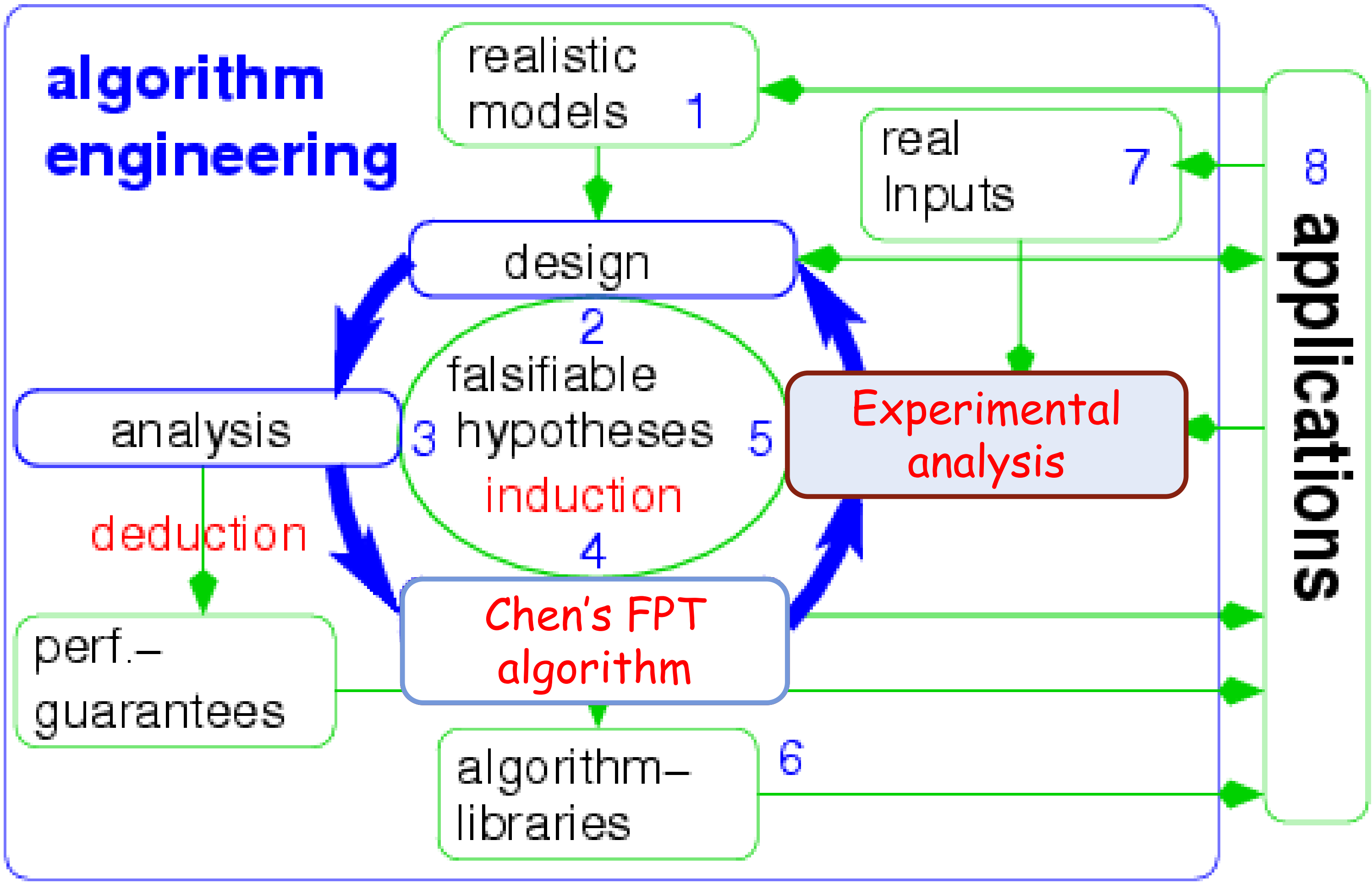

Peter Sanders, Festschrift Mehlhorn 60th, 2009

# Algorithm Engineering for DFVS



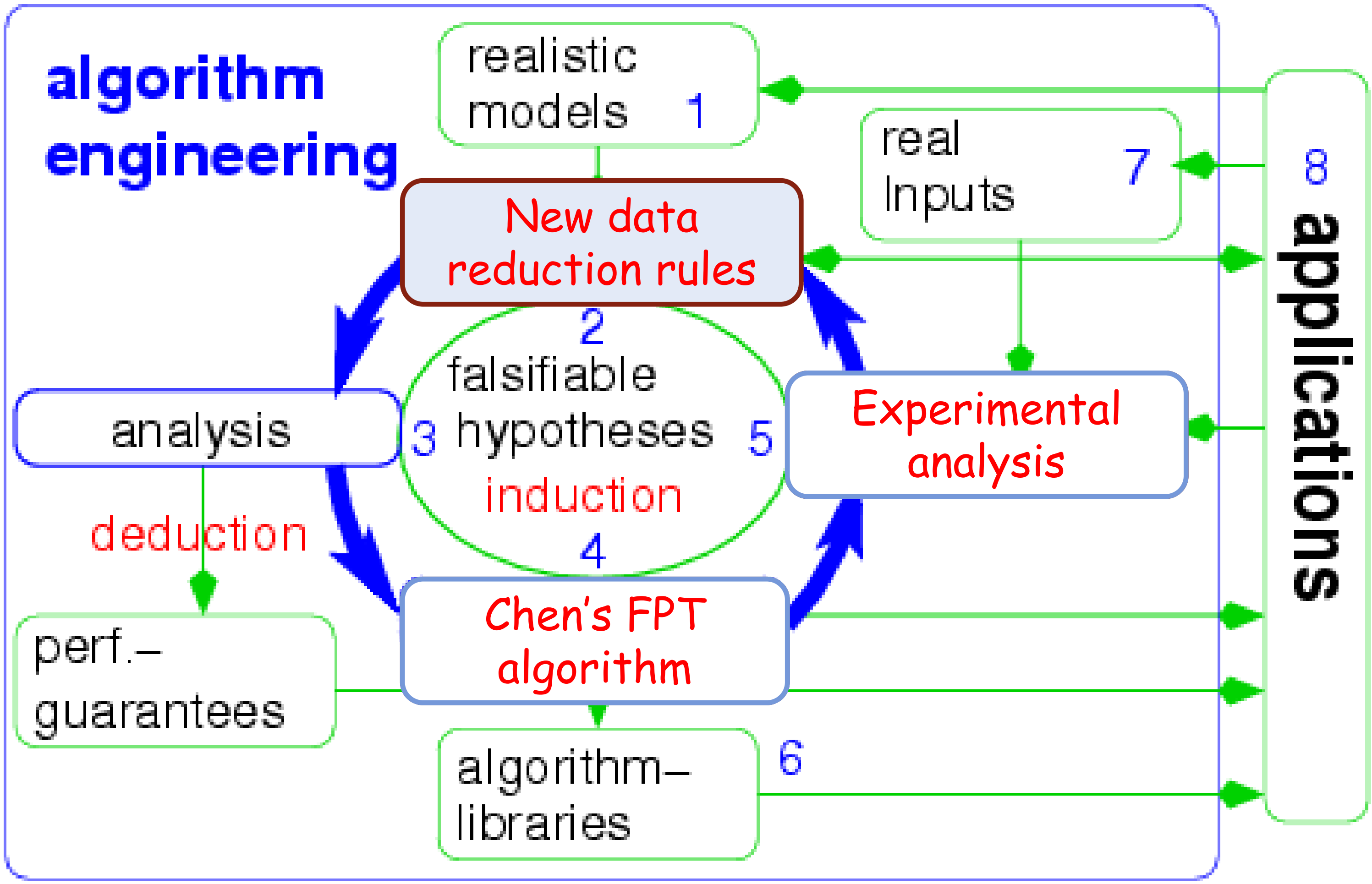

Peter Sanders, Festschrift Mehlhorn 60th, 2009

# Algorithm Engineering for DFVS



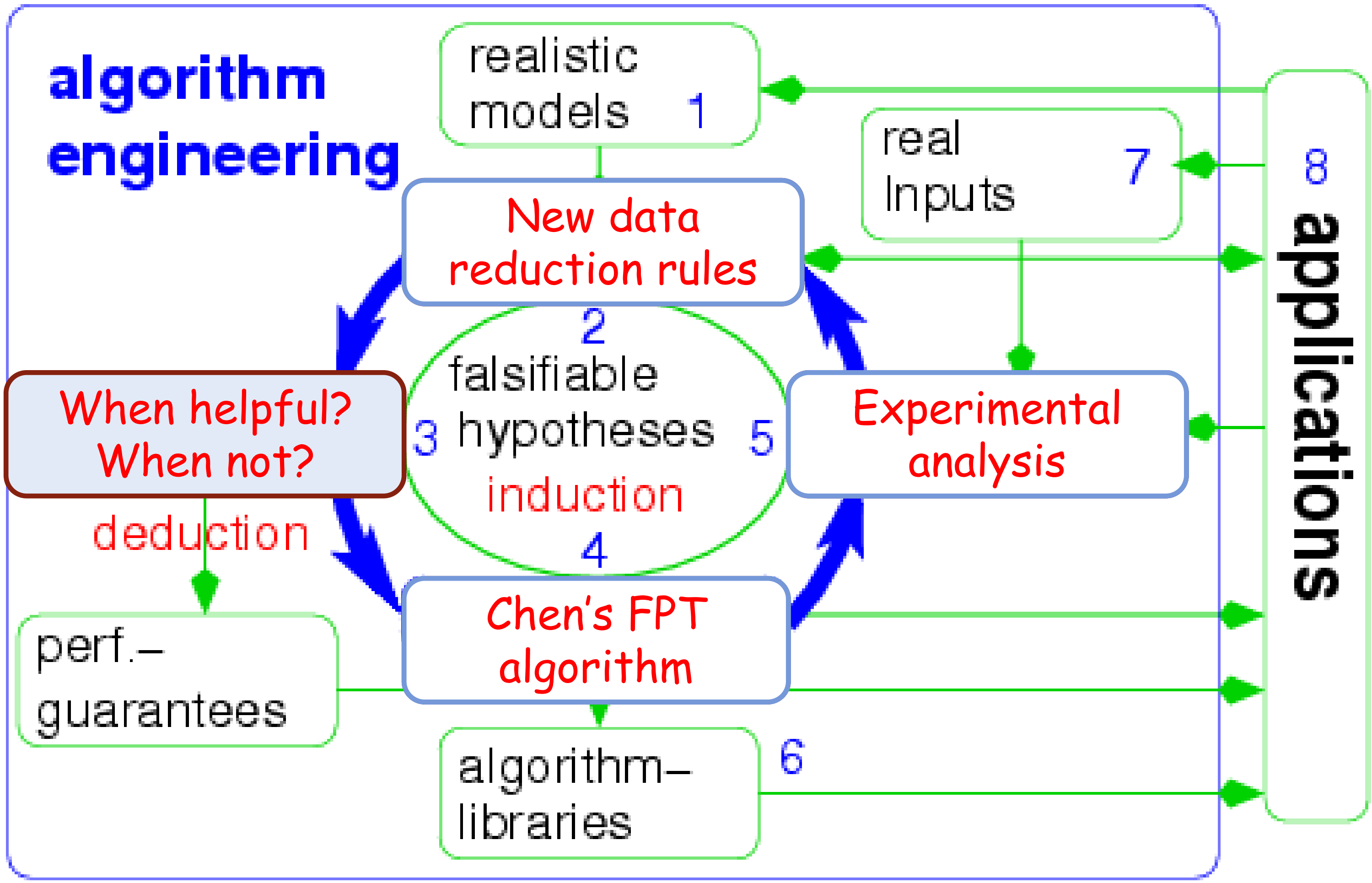

Peter Sanders, Festschrift Mehlhorn 60th, 2009

# Algorithm Engineering for DFVS



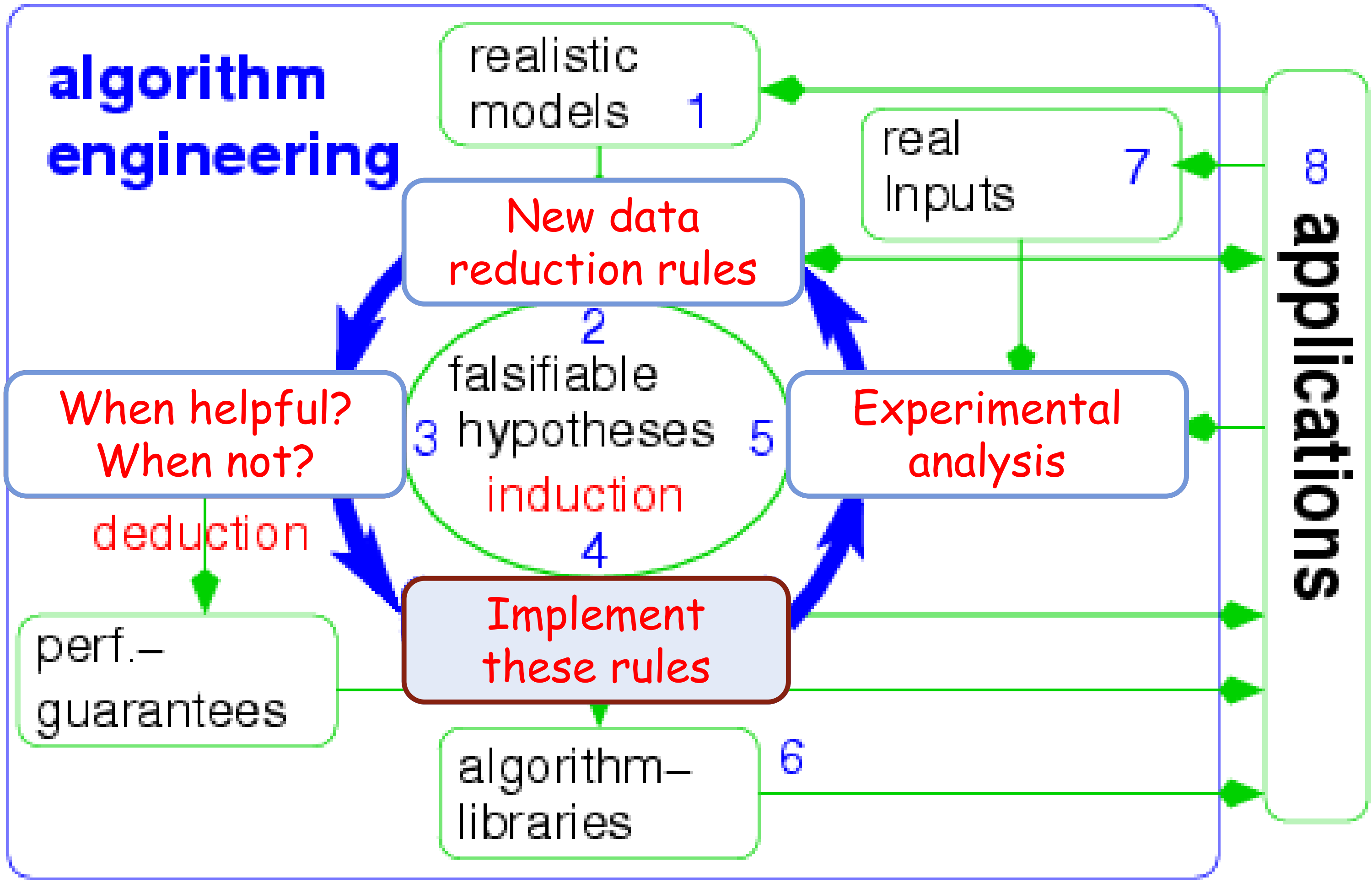

Peter Sanders, Festschrift Mehlhorn 60th, 2009

# Algorithm Engineering for DFVS



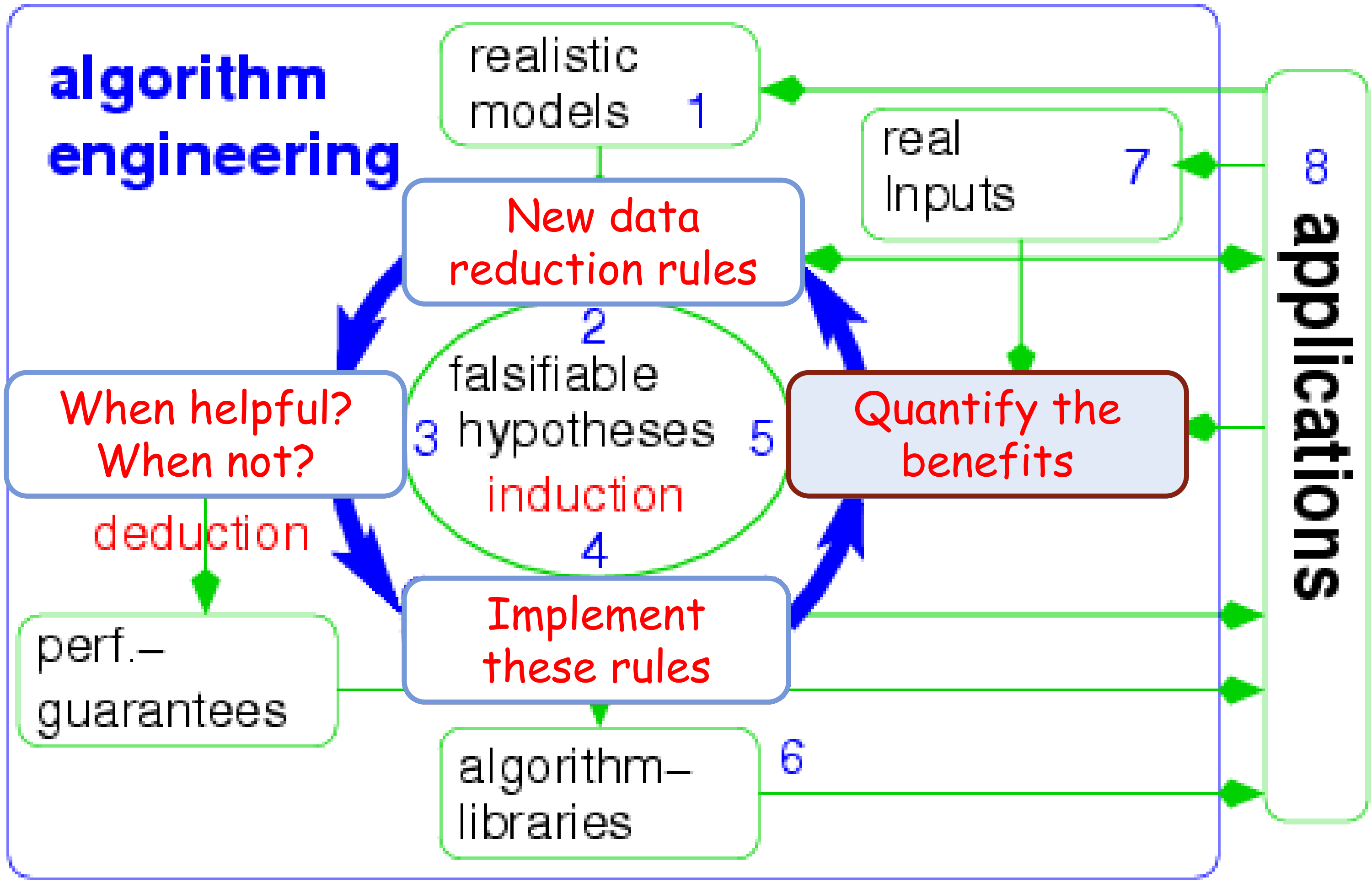

Peter Sanders, Festschrift Mehlhorn 60th, 2009

# Algorithm Engineering for DFVS



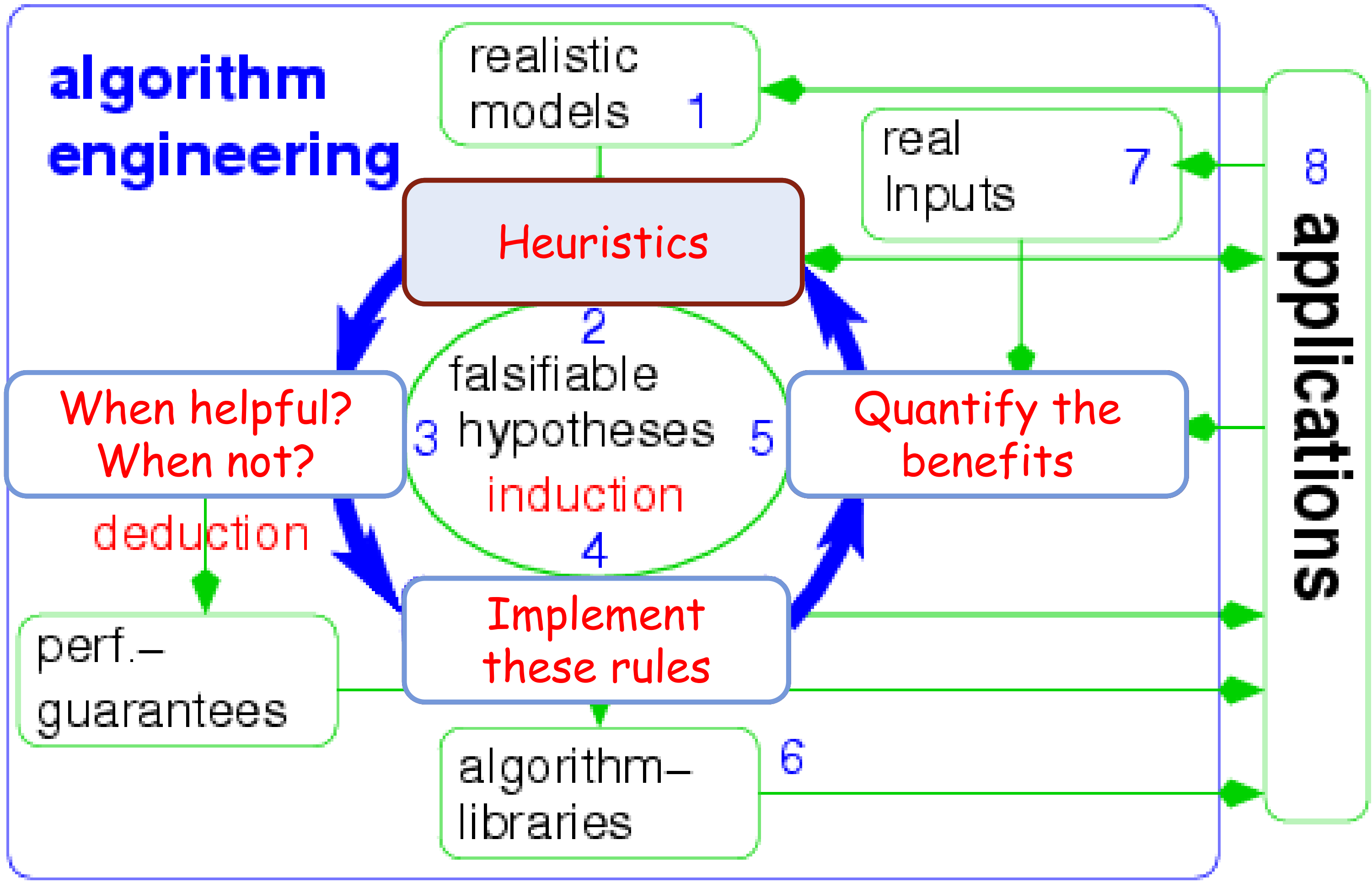

Peter Sanders, Festschrift Mehlhorn 60th, 2009

# Algorithm Engineering for DFVS



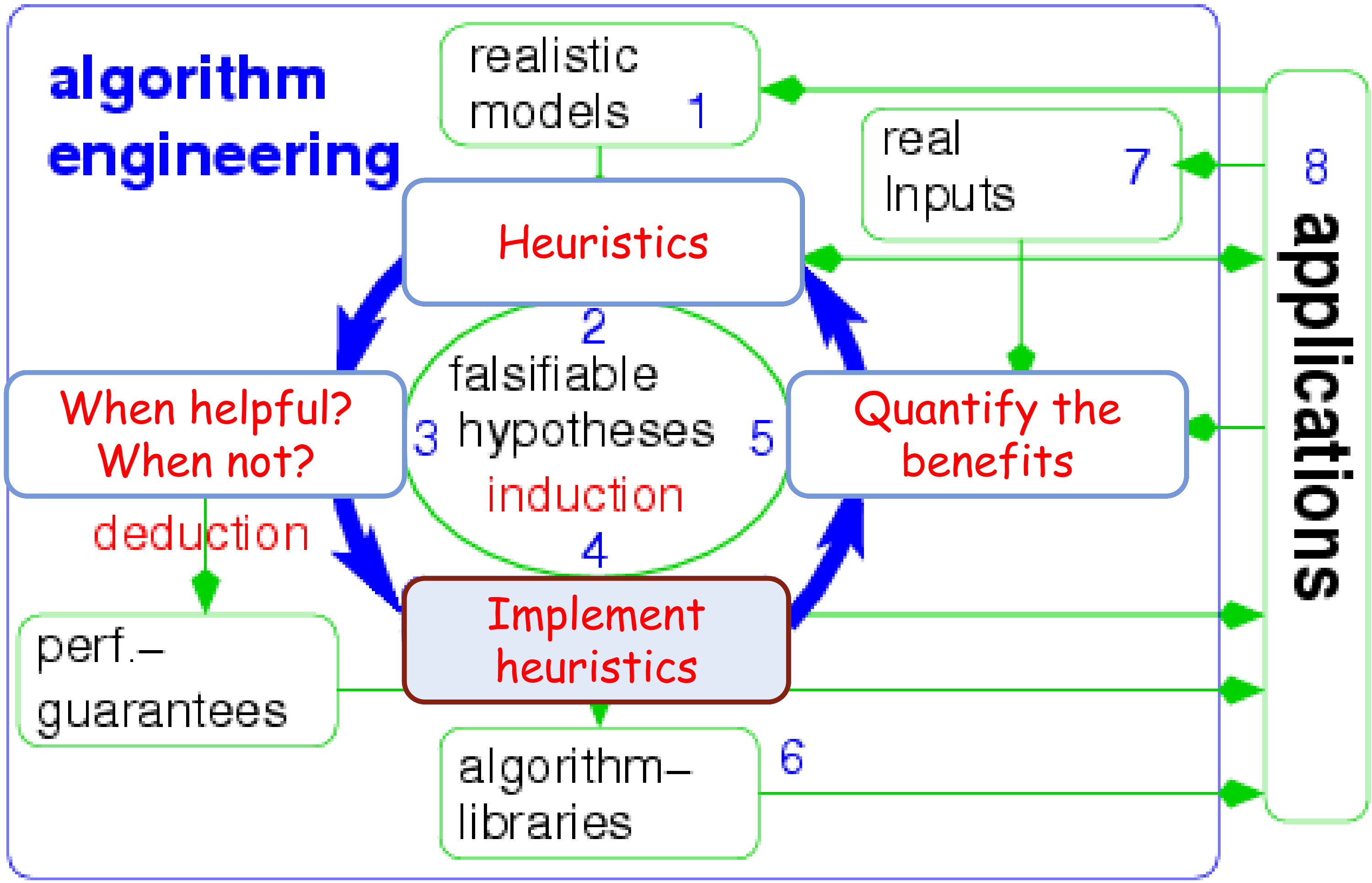

Peter Sanders, Festschrift Mehlhorn 60th, 2009

# Algorithm Engineering for DFVS



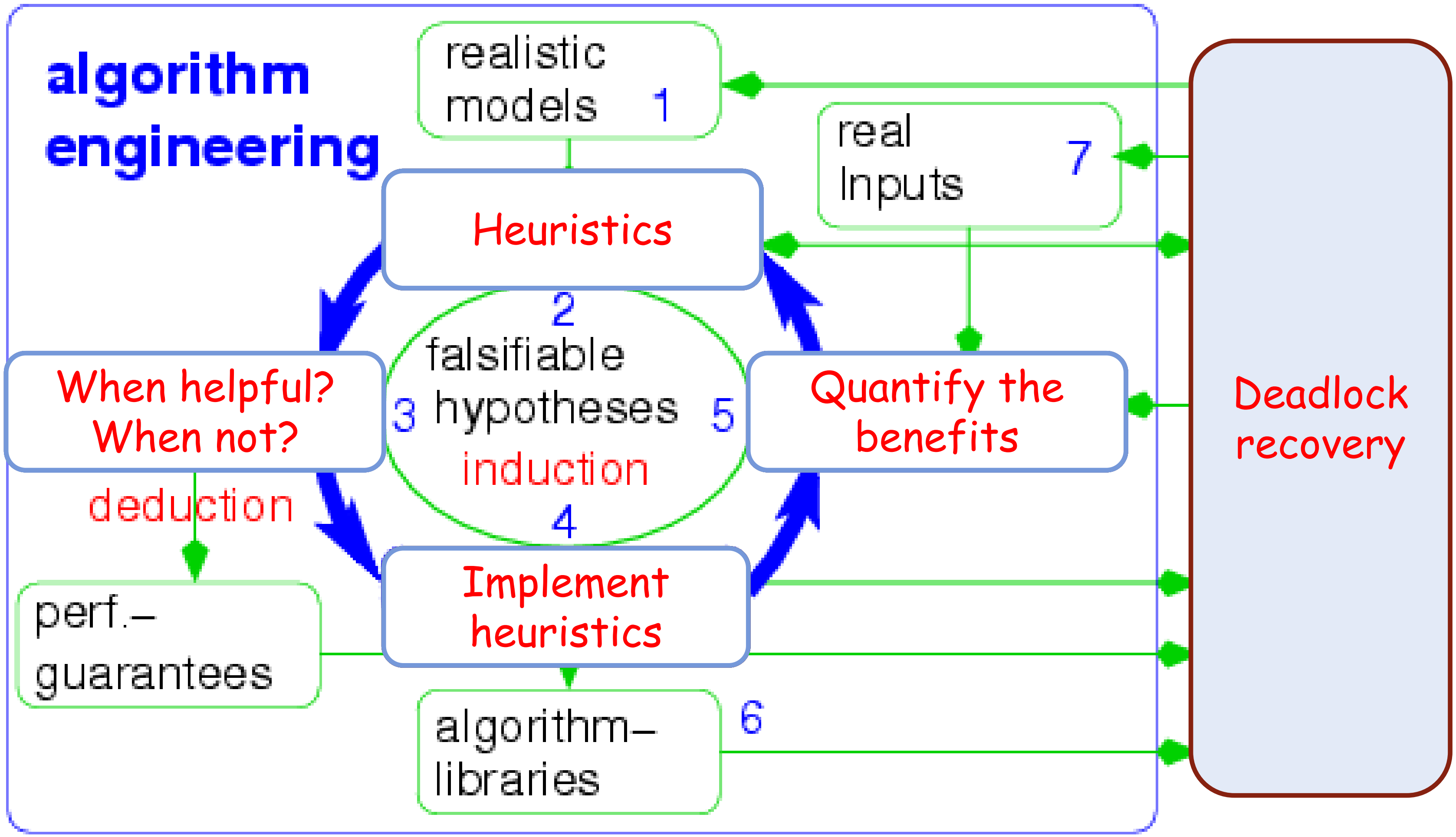

Peter Sanders, Festschrift Mehlhorn 60th, 2009

# Algorithm Engineering for DFVS




Peter Sanders, Festschrift Mehlhorn 60th, 2009

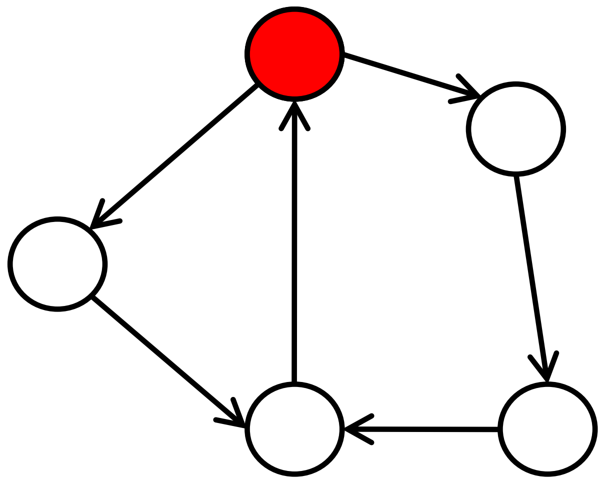
# Algorithm Engineering for DFVS




Peter Sanders, Festschrift Mehlhorn 60th, 2009

# Directed Feedback Vertex Set (DFVS)

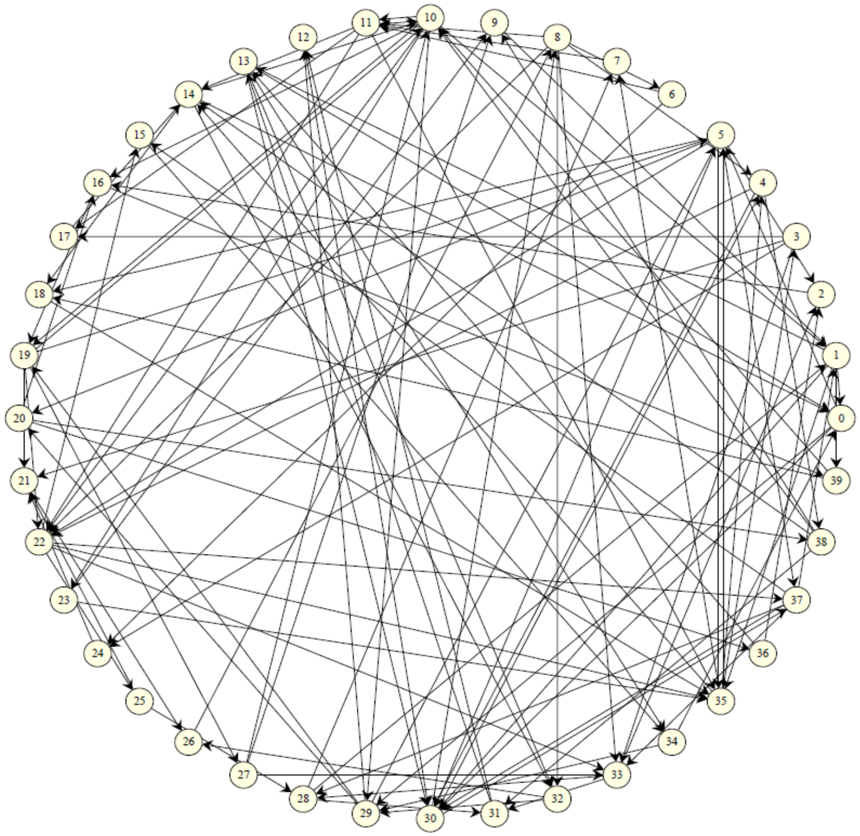
Find k vertices to destroy all cycles



1-FVS

# Directed Feedback Vertex Set (DFVS)

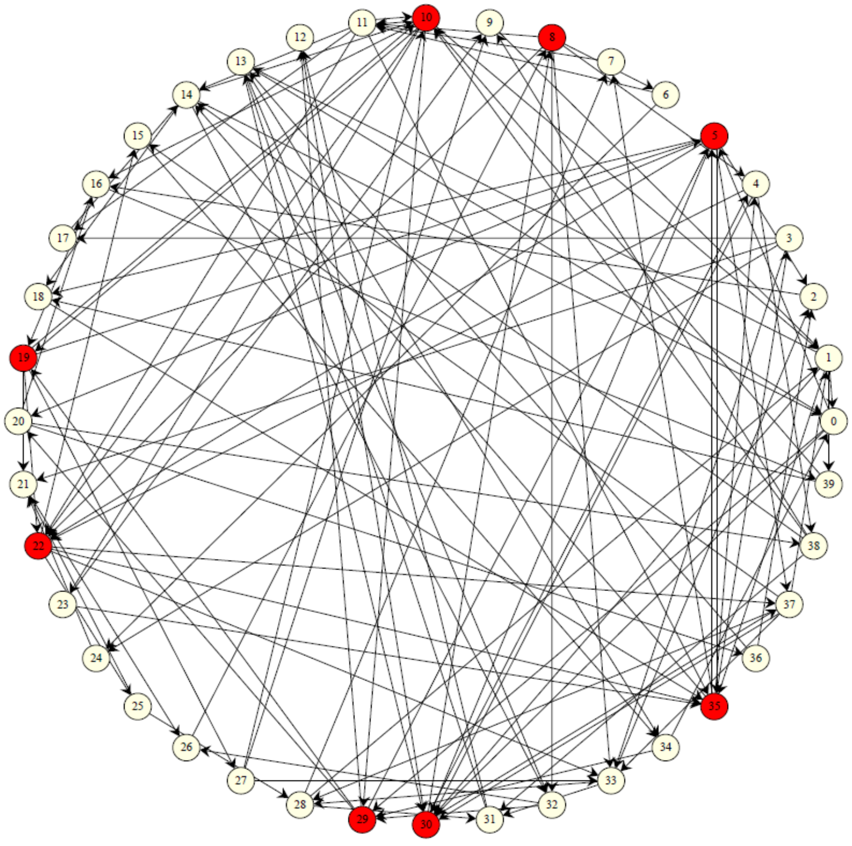
Find k vertices to destroy all cycles




What about this one with k=8?

# Directed Feedback Vertex Set (DFVS)

Find k vertices to destroy all cycles




What about this one with k=8?

Minimum DFVS is NP-hard

# DFVS vs. Undirected Feedback Vertex Set (UFVS)

Both NP-hard, but UFVS is better understood

| | DFVS | UFVS |
|---|---|---|
| Approximation | $O(\min\{\tau^*\log\tau^*\log\log\tau^*, \tau^*\log N \log\log N\})$-approximation [Even '98] ($\tau^*$ is the optimum fractional solution) | 2-approximation [Bafna '1999] |
| FPT algorithm | $O(k!\ 4^k\ n^{O(1)})$ [Chen STOC'2008] | $O(4^k\ k\ n)$ [Becker '2000] |
| Kernelization | Polynomial kernel? | Quadratic kernel [Thomasse SODA'2008] |

# Our Work

- Test engine: random graph generator
  - Controlling various parameters
- Experimental study of Chen's FPT algorithm for DFVS [Chen STOC'2008]
  - With various parameters
- Data reductions and heuristics
  - Quantify the benefits
- Application: deadlock recovery
  - DFVS not more helpful than cycle detection

# Random Graph Generator

- Goal: difficult, random graphs

- Parameters controlled:

    n: Number of nodes

    k: Size of the minimum FVS

    edge density: ed = #edges/n

- A nontrivial task…

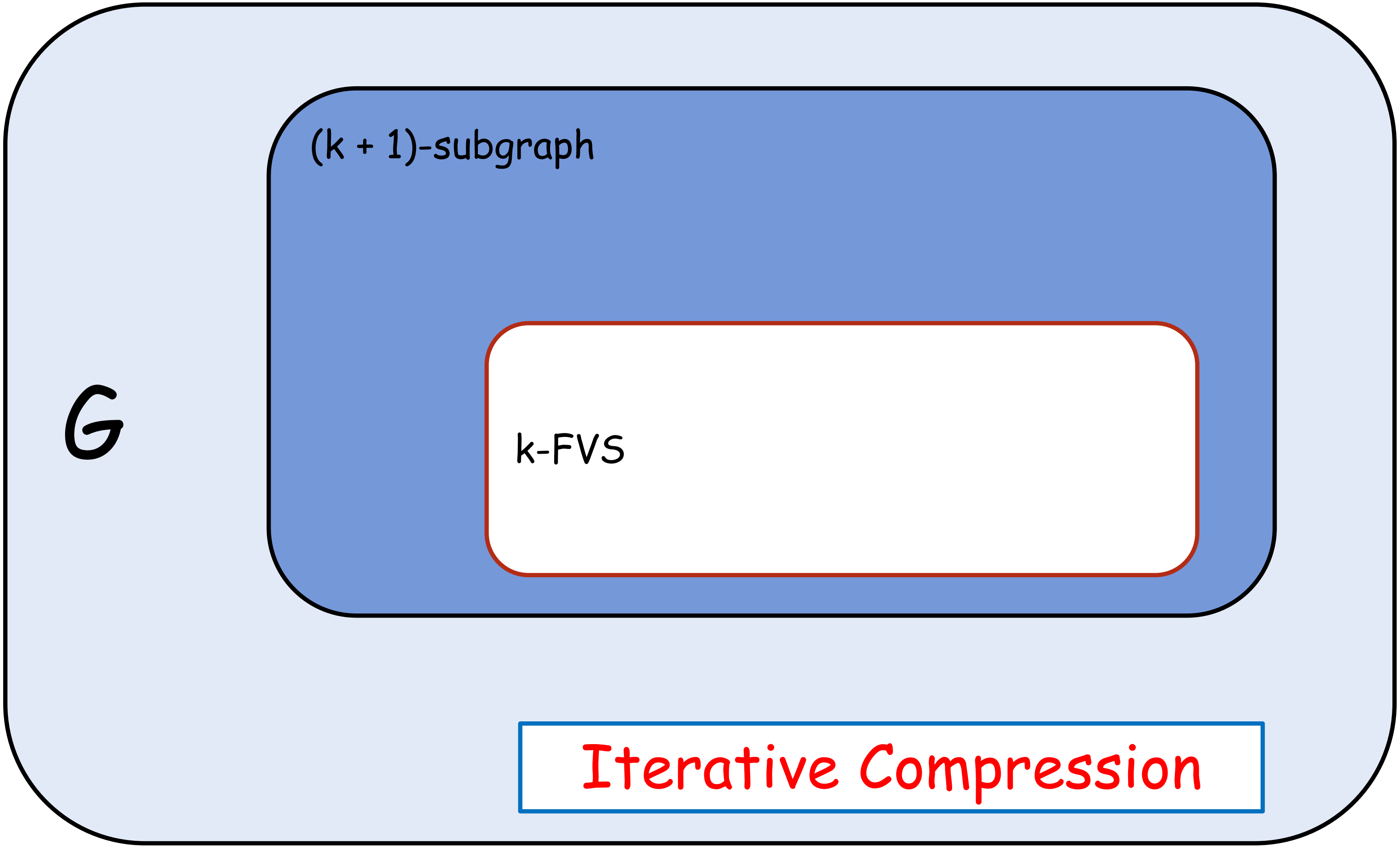
# A Non-trivial Task

- Spanning tree
  Wilson's algorithm

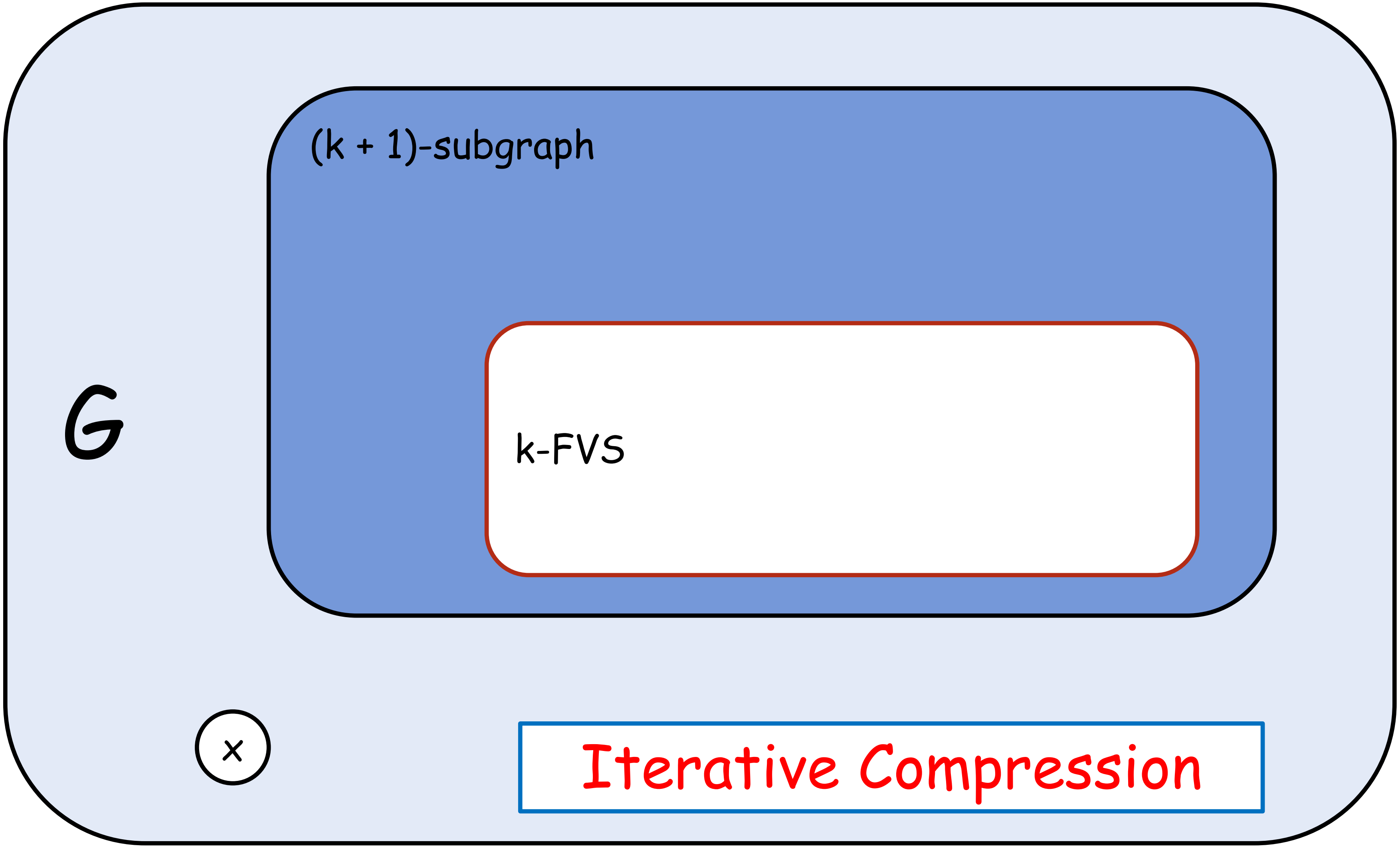
- Connected DAG
  Melancon's algorithm

- Control
  solution size, overlapping cycles, edge density …

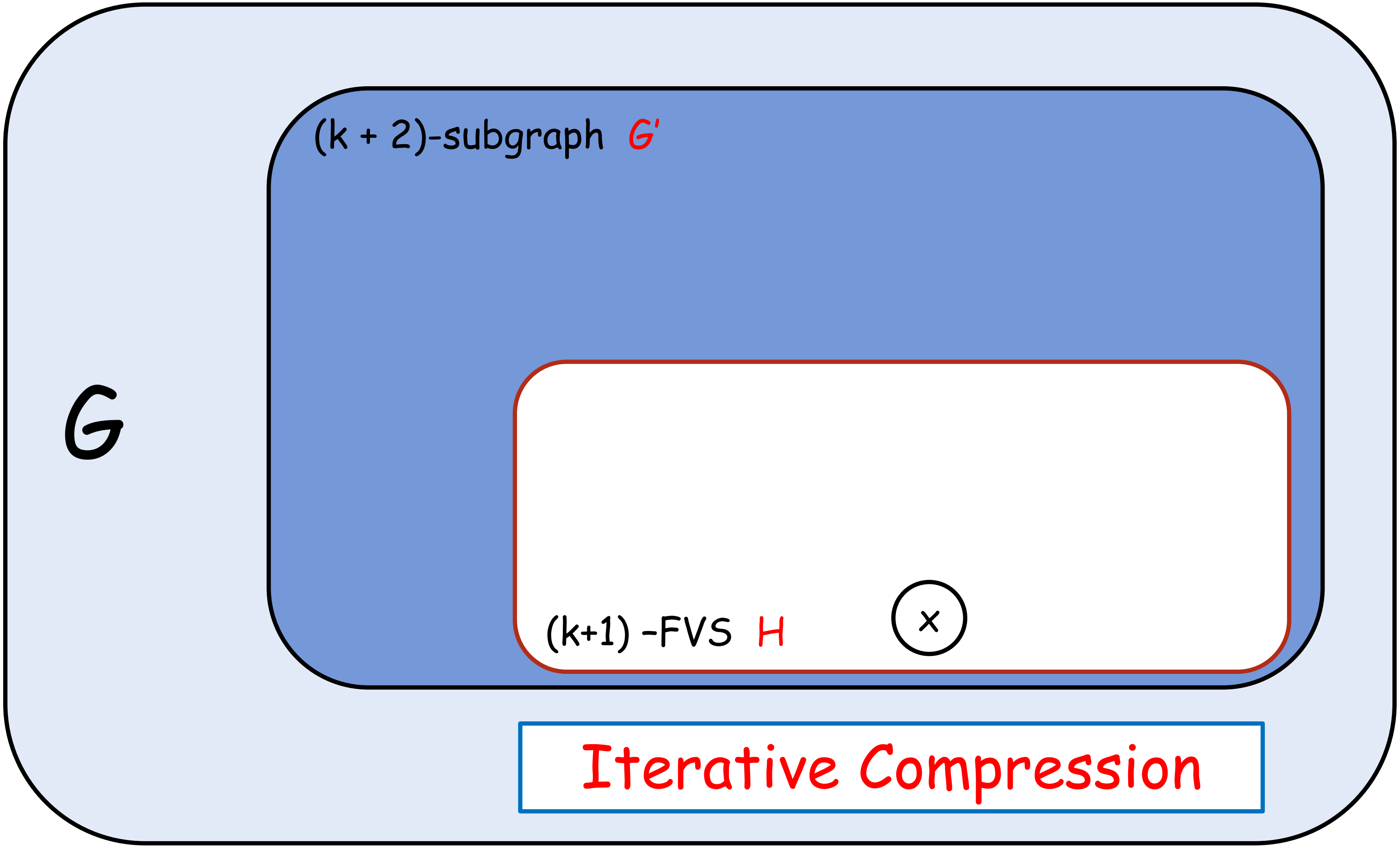
# Chen's Algorithm



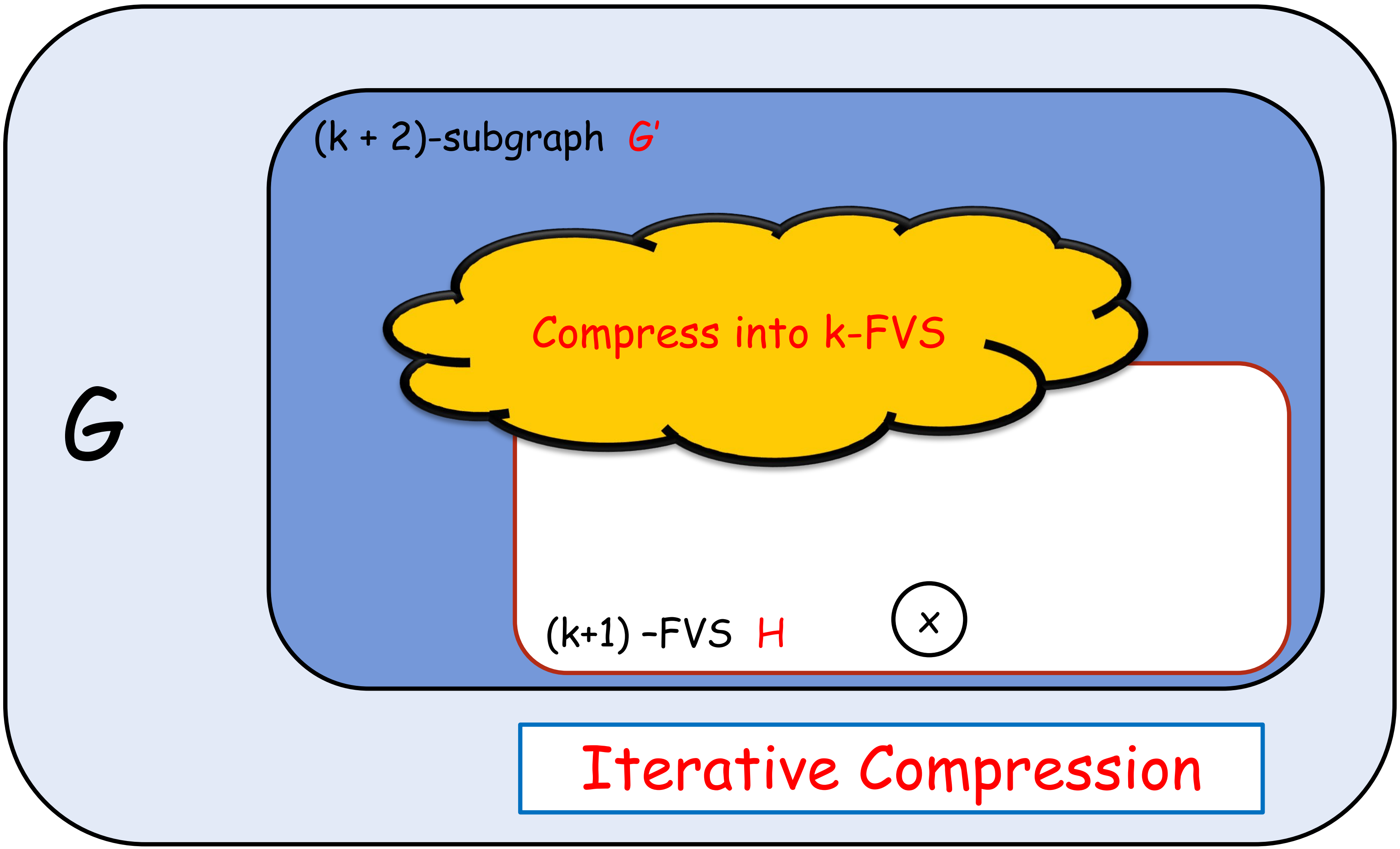
# Chen's Algorithm



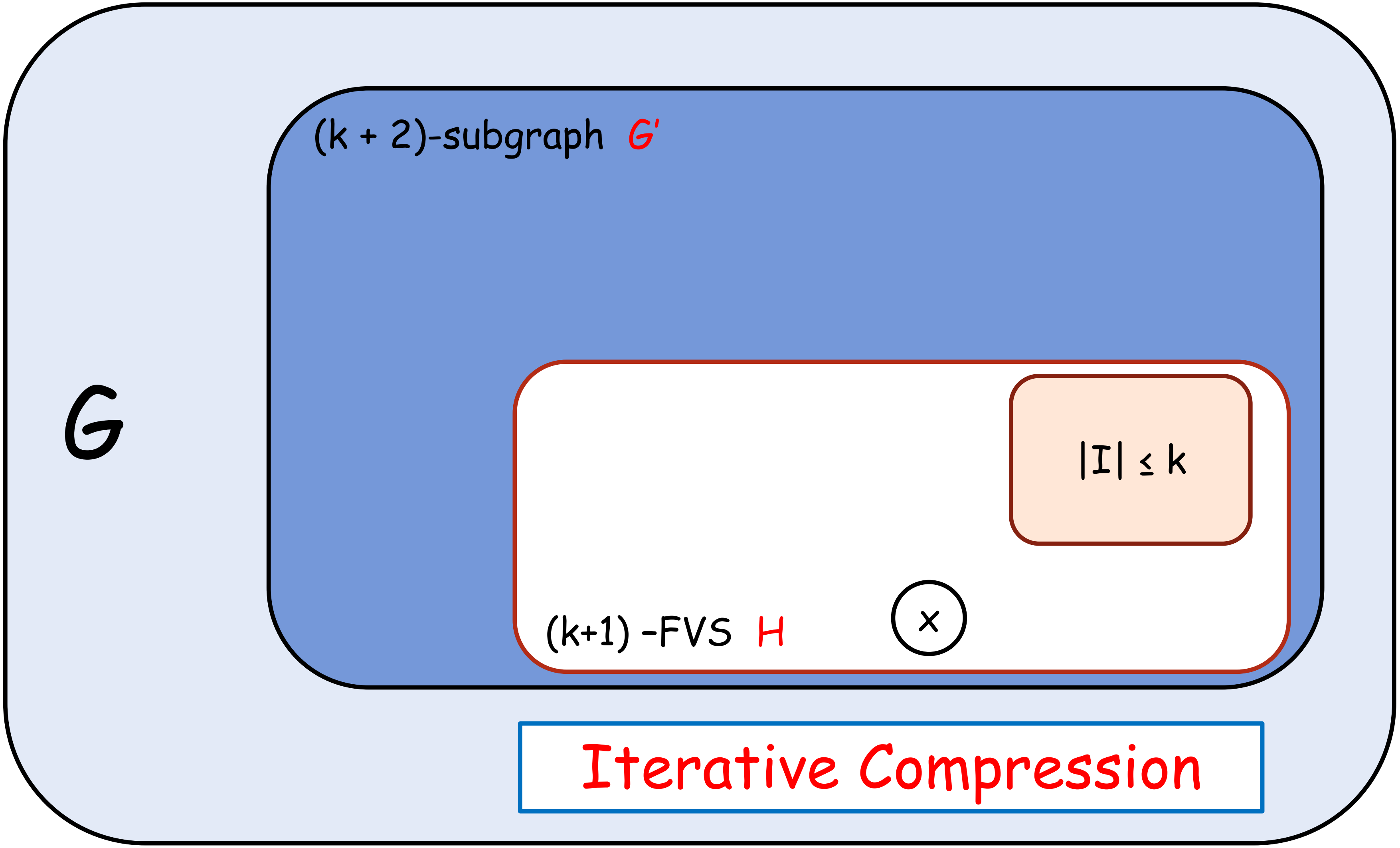
# Chen's Algorithm



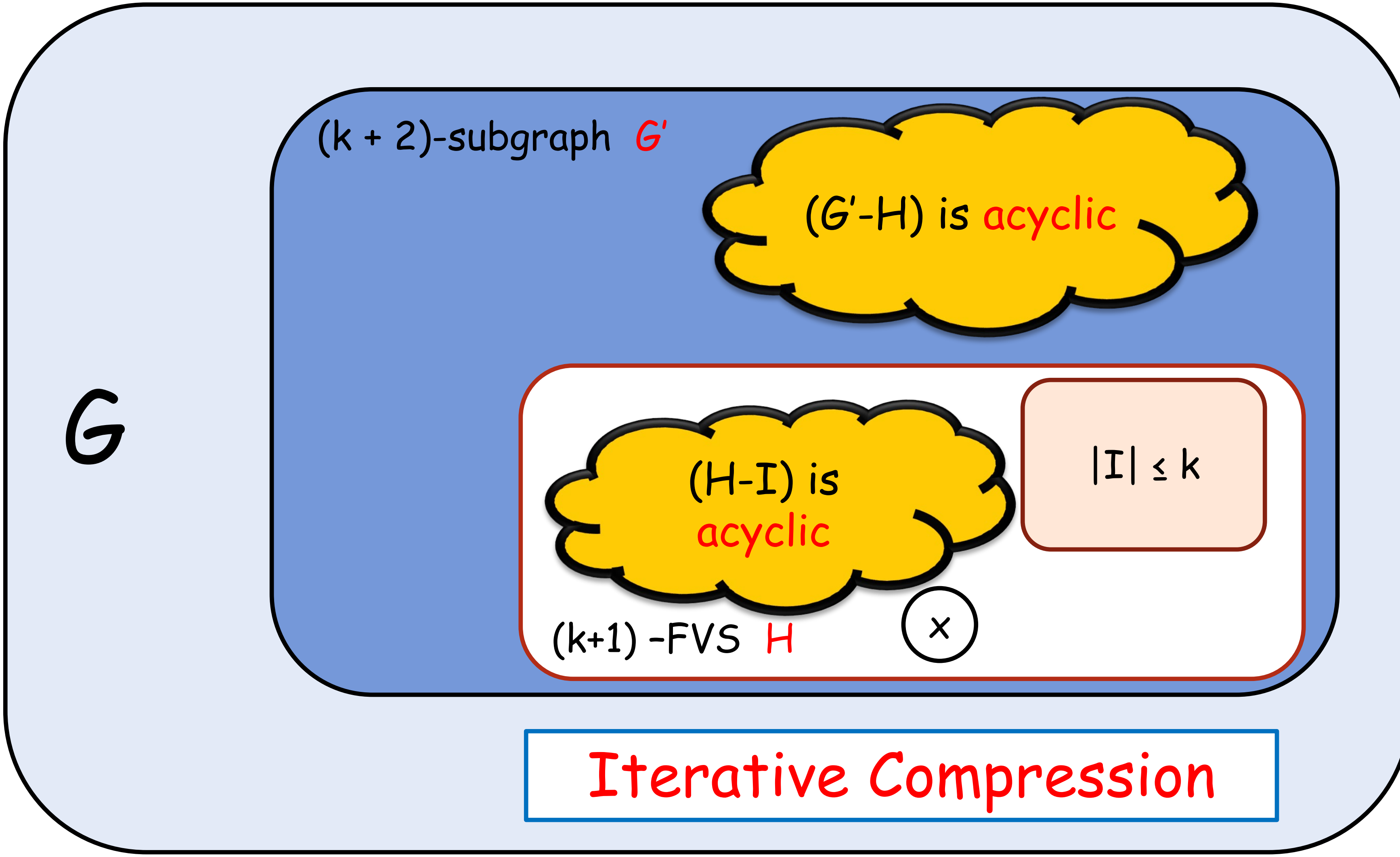
# Chen's Algorithm



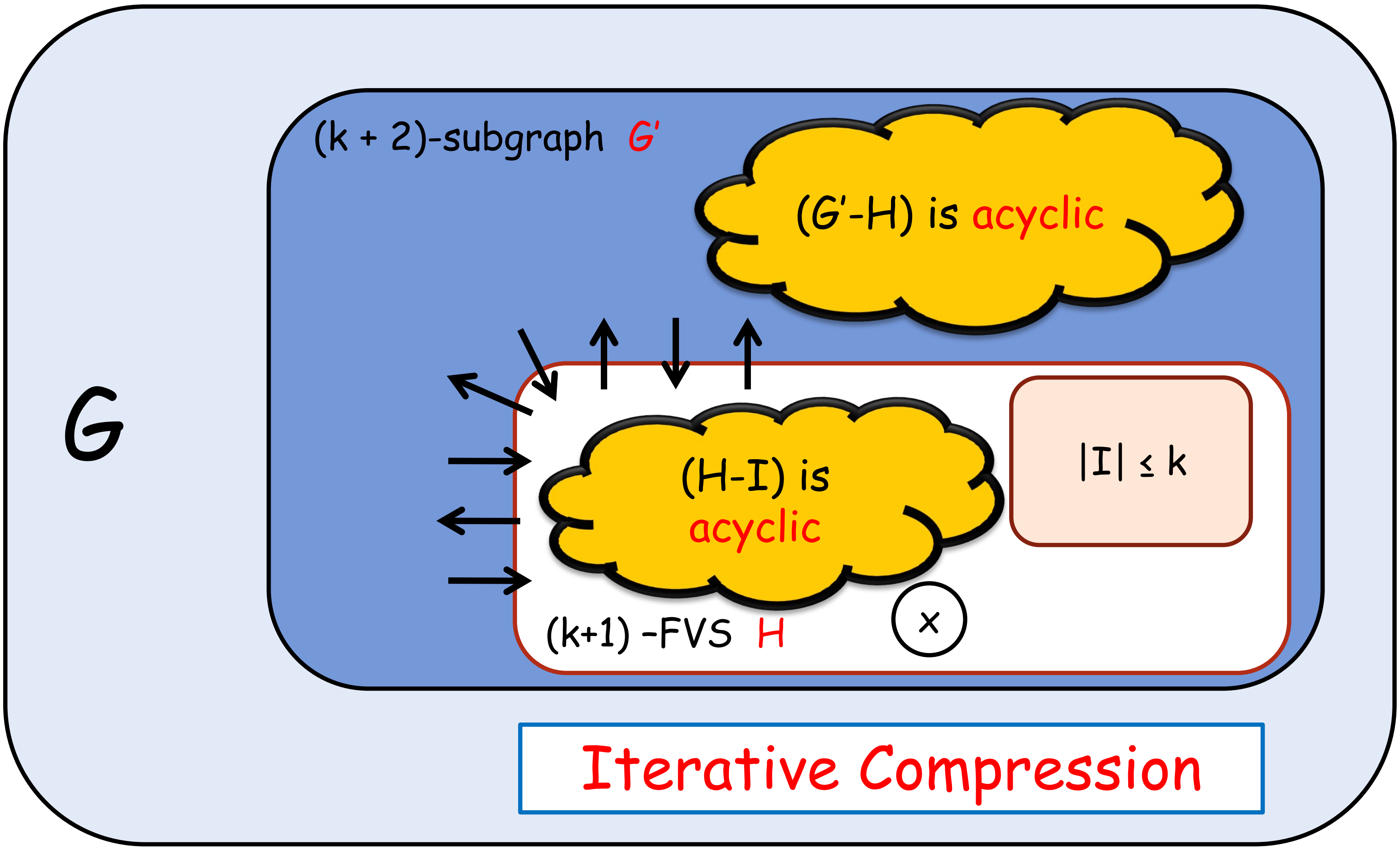
# Chen's Algorithm



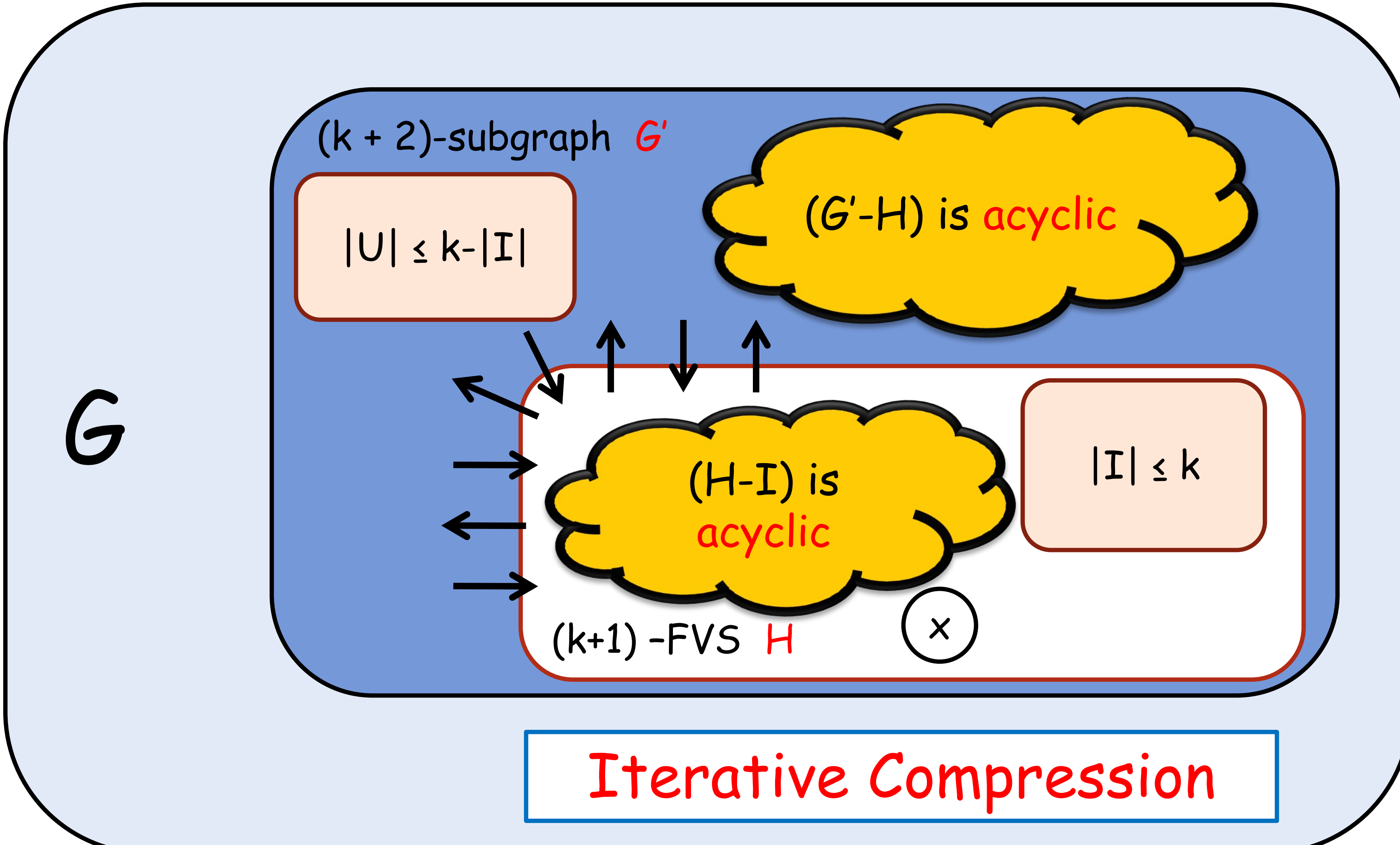
# Chen's Algorithm



# Chen's Algorithm



# Chen's Algorithm



# Start Configuration of Chen's Algorithm

- Consider heuristic solution X:
  - choose a k-subset Y of X
  - start with S = (G - (X - Y))

$\boxed{\text{If X is good, then better performance}}$

# Chen's Original Algorithm

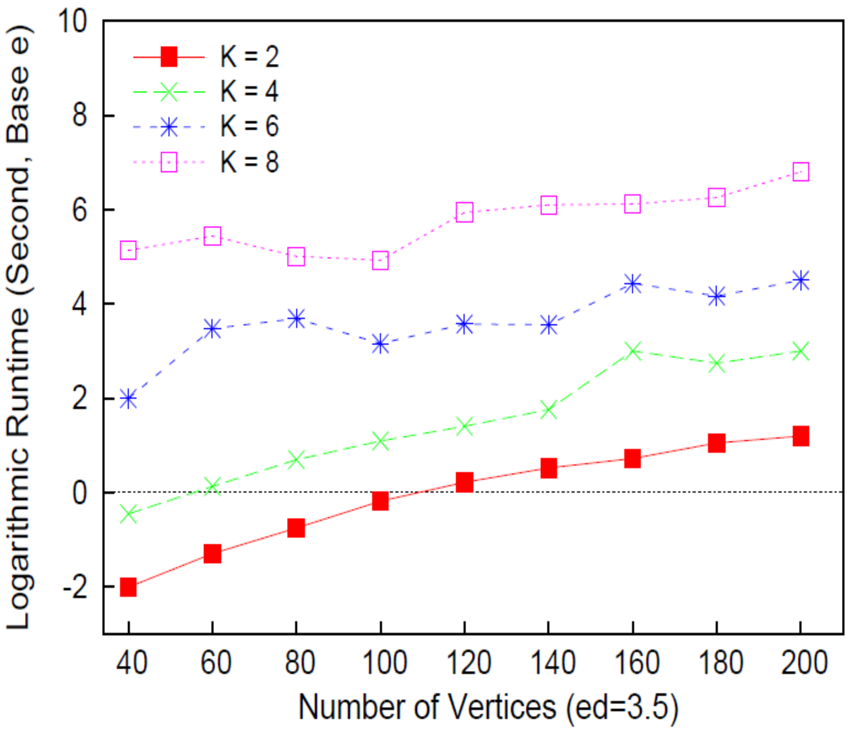
- Configuration
  - n: from 40 to 200, step by 20
  - k: in {2, 4, 6, 8}
  - ed in {2.0, 3.0, 3.5, 4.0}
- Generate 10 graphs for (n, k, ed)

  Record max, min and average
- Timeout: 3 hours

  Count as 3 hours

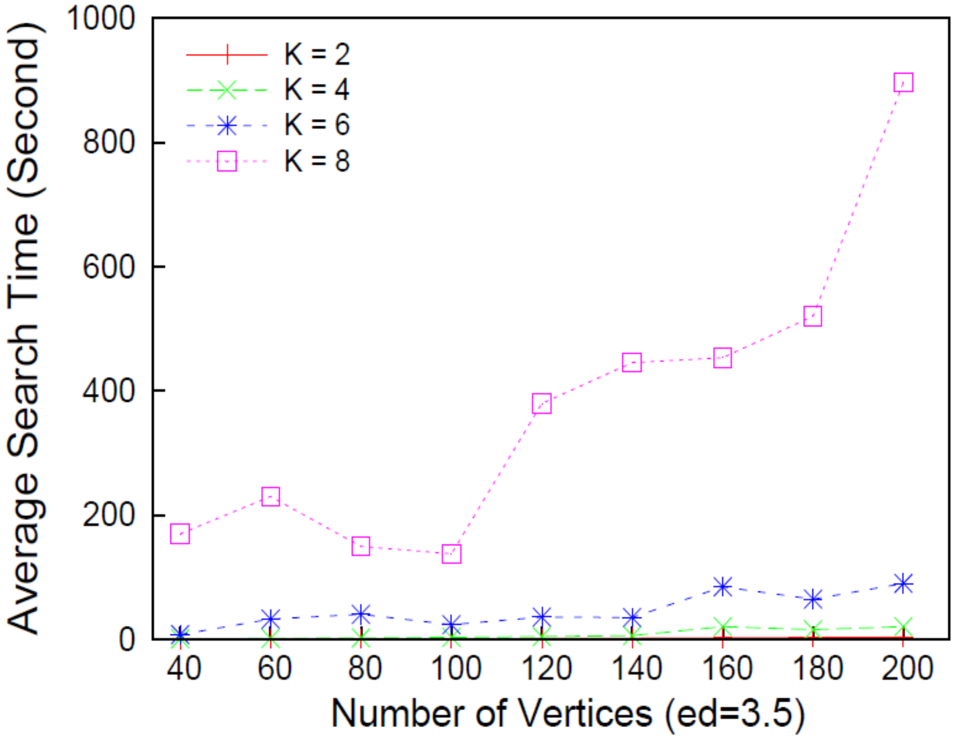
# Runtime Performance

## Logarithmic runtime (ed=3.5)




Algorithm is slow, even with small parameter k

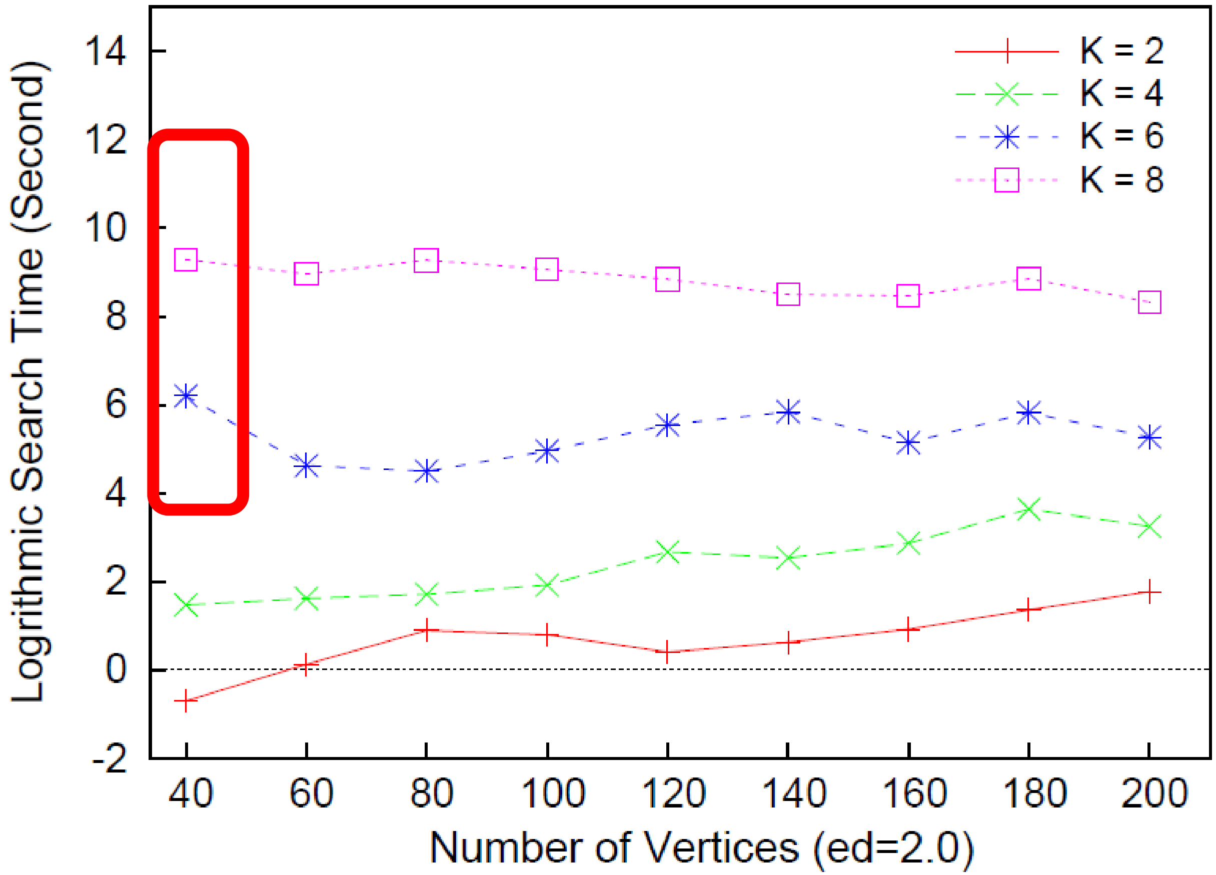
## Runtime (ed=3.5)




Performance fluctuates when n increases

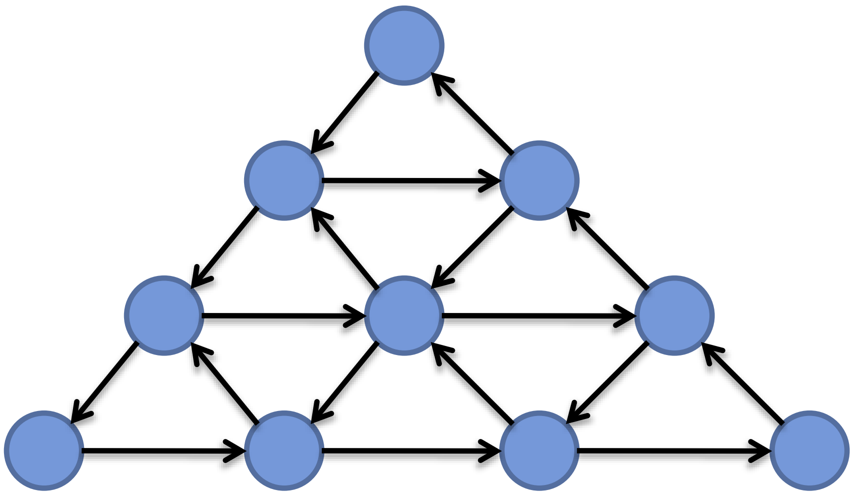
# Low Edge Density: High Runtime

Worst performance for n=40



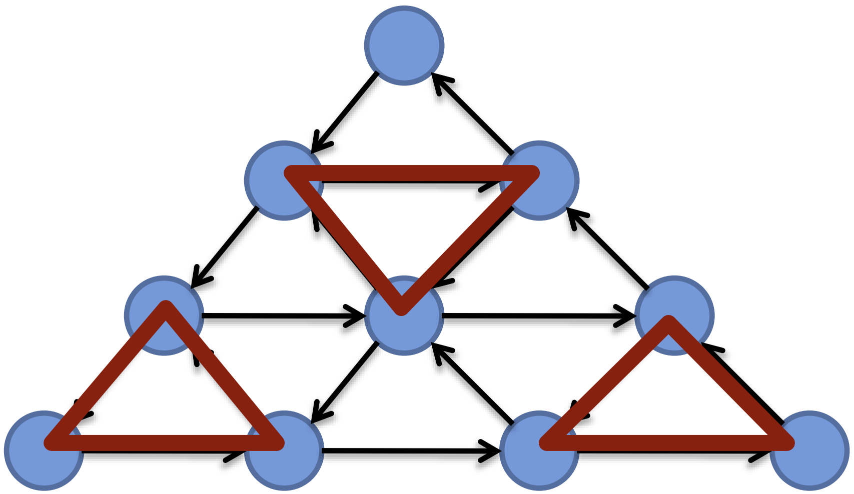
# Many Independent Cycles: High Runtime

- An intuitive example: "Pyramid"



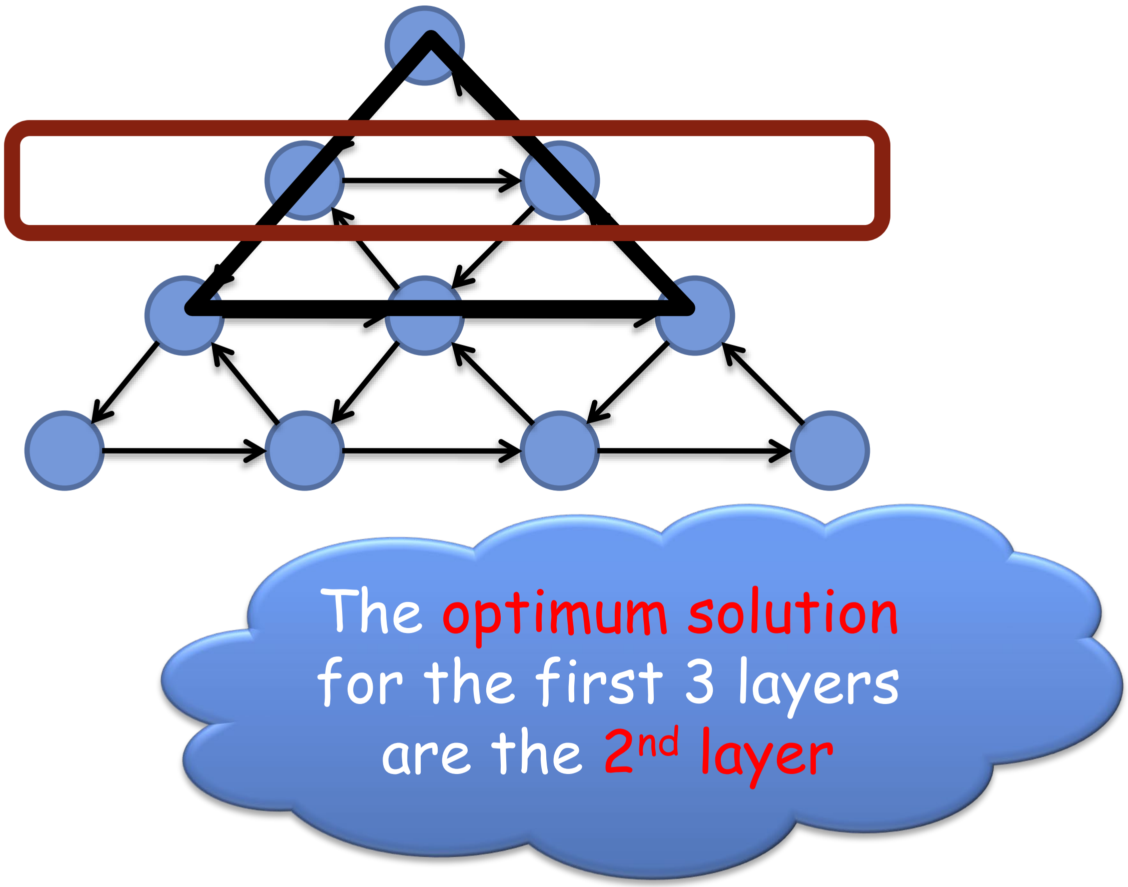
# Many Independent Cycles: High Runtime

- An intuitive example: "Pyramid"



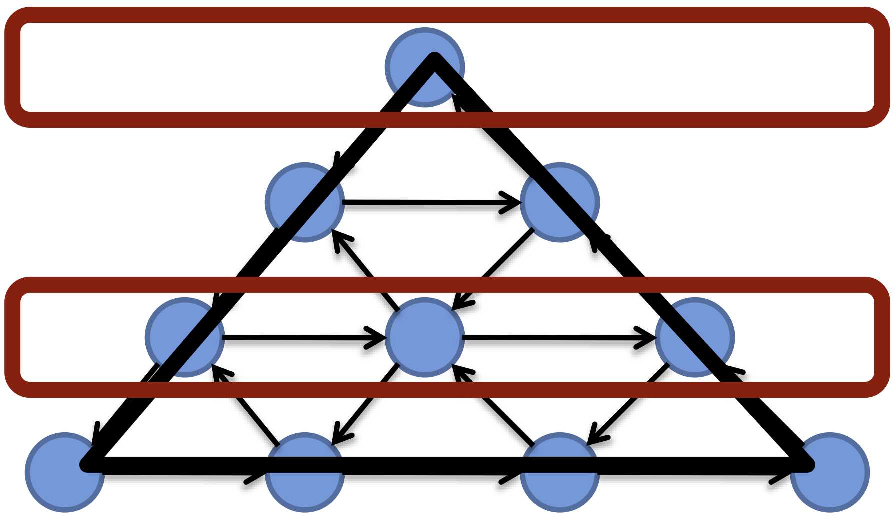
# Many Independent Cycles: High Runtime

- An intuitive example: "Pyramid"



# Many Independent Cycles: High Runtime

- An intuitive example: "Pyramid"



The optimum solution for the whole graph is the 1st and 3rd layers

# Reduction Rules

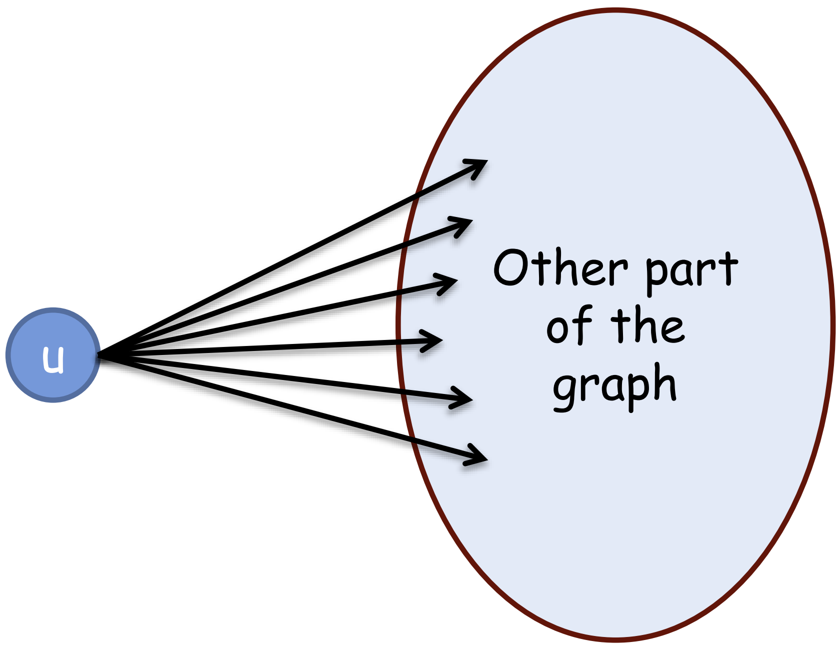
- Reduction: in polynomial time
  - reduce $(G, k)$ to $(G', k')$
  - $(G, k)$ is a YES-instance iff $(G', k')$ is a YES-instance
- Kernelization
  - $|G'|$ is bounded by $f(k)$
  - $k' \leq k$

# Reduction Rules I (Chen)

- Trivial rules:
  - Self-loops: add node to DFVS
  - Parallel-edge: delete multiple edges

# Reduction Rule II

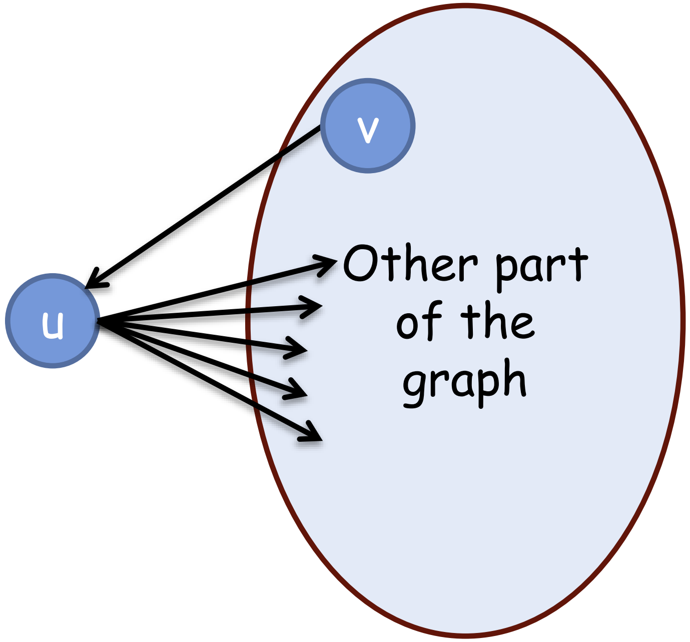
- Dummy
  - in-degree/out-degree = 0;




delete u

# Reduction Rules III

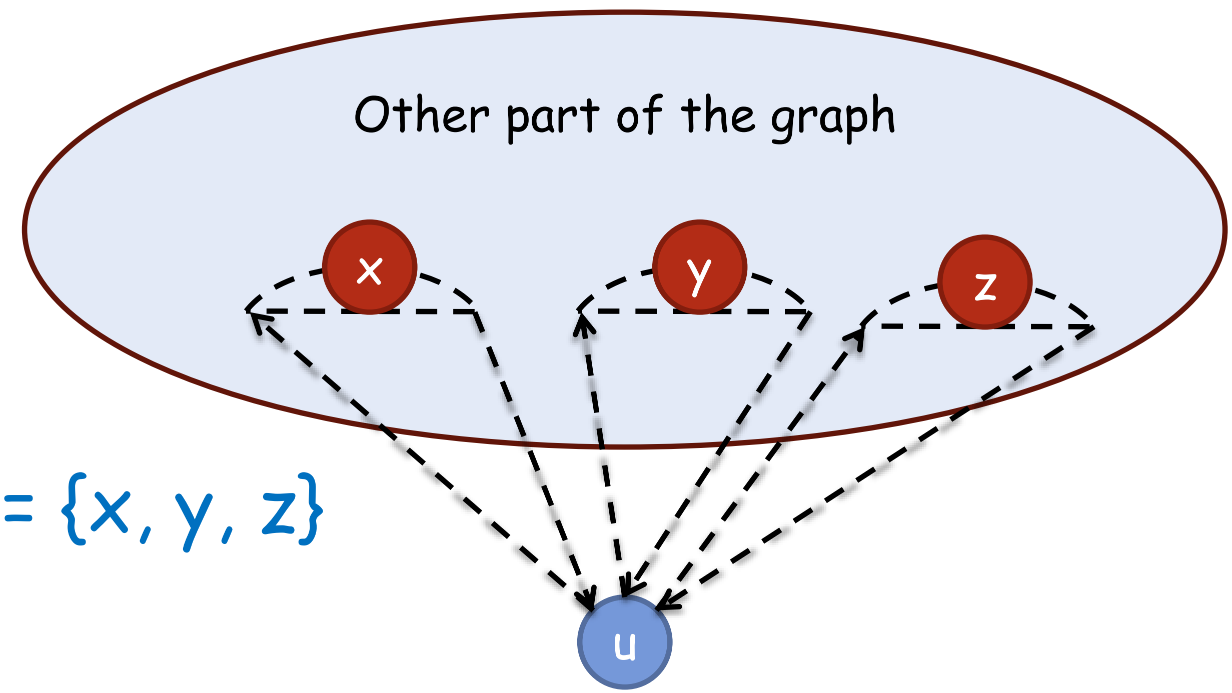
- Chain
  - in-degree/out-degree = 1;




merge u and v

# Reduction Rule IV

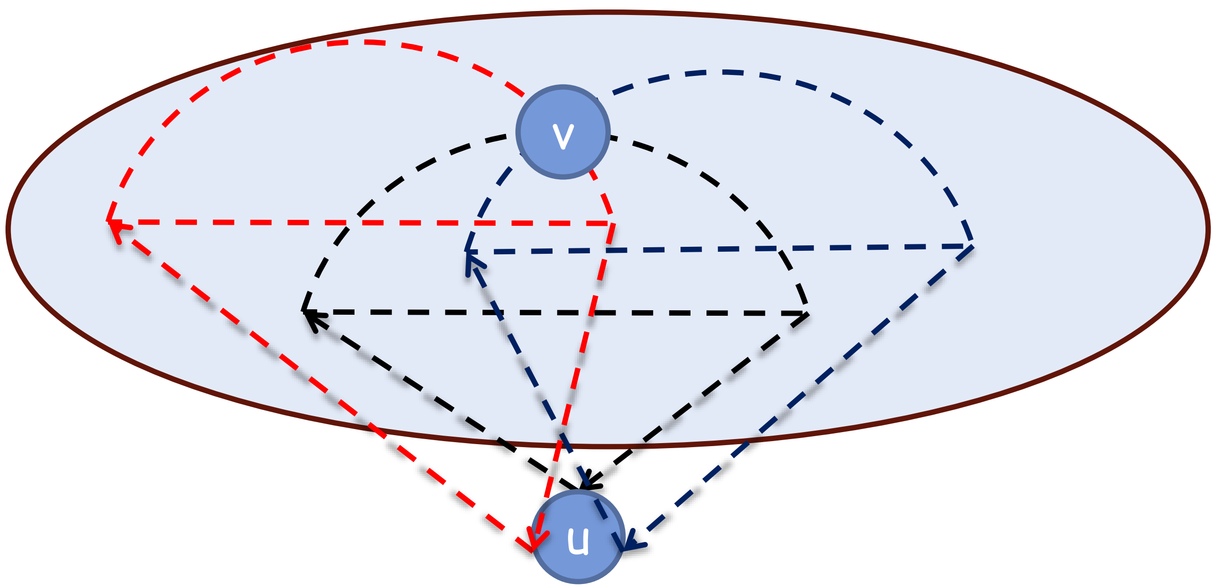
- Flower
  - (k+1) vertex-independent cycles exactly intersecting on u: add u to DFVS




k = 2
petal(u) = {x, y, z}

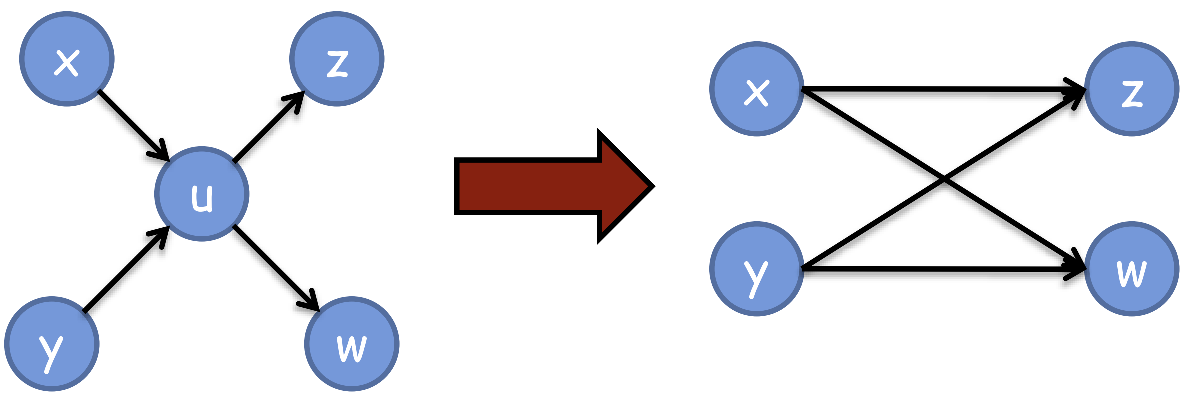
# Reduction Rule V

- Shortcut
  - petal(u) = {v}: bypass u



# Reduction Rule V

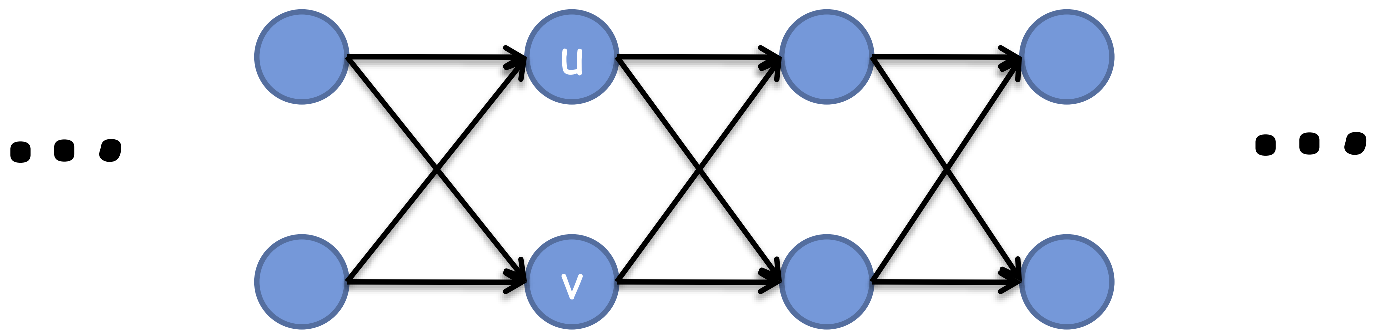
- Shortcut
  - petal(u) = {v}: bypass u



# Remarks

- These rules do NOT give a kernelization
- We need rules when $2 \leq |petal(u)| \leq k$

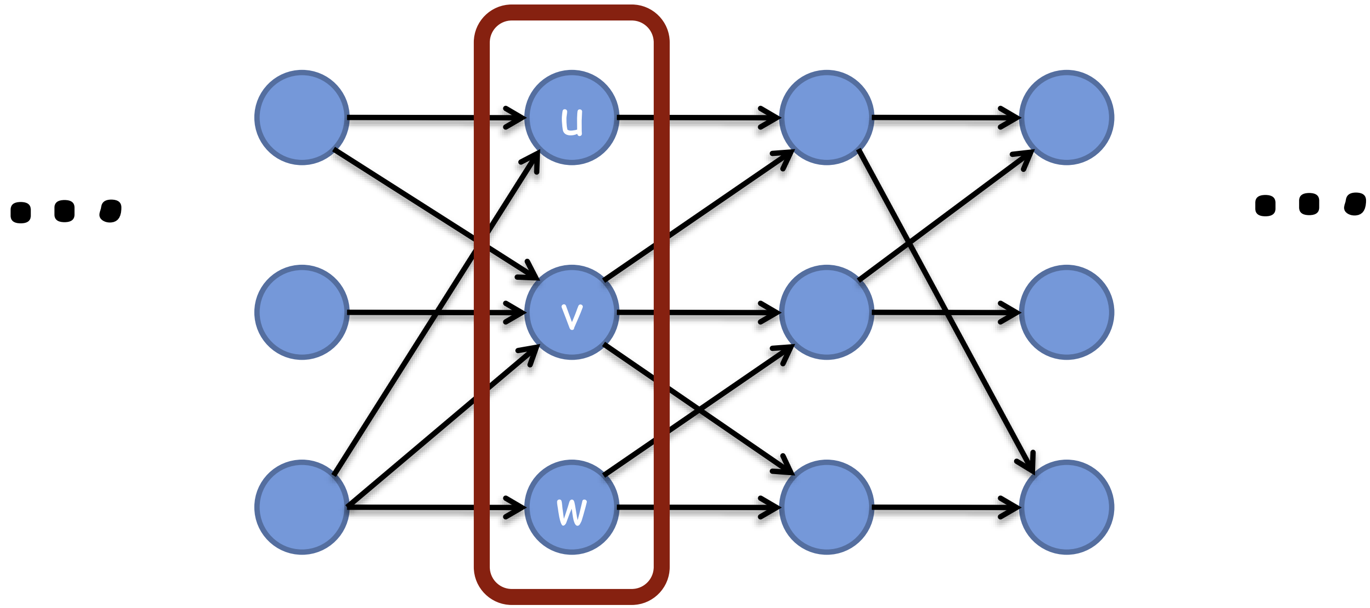
# Non-Reducible Graphs




| petal(u) | = | petal(v) | = 2

u, v are useless

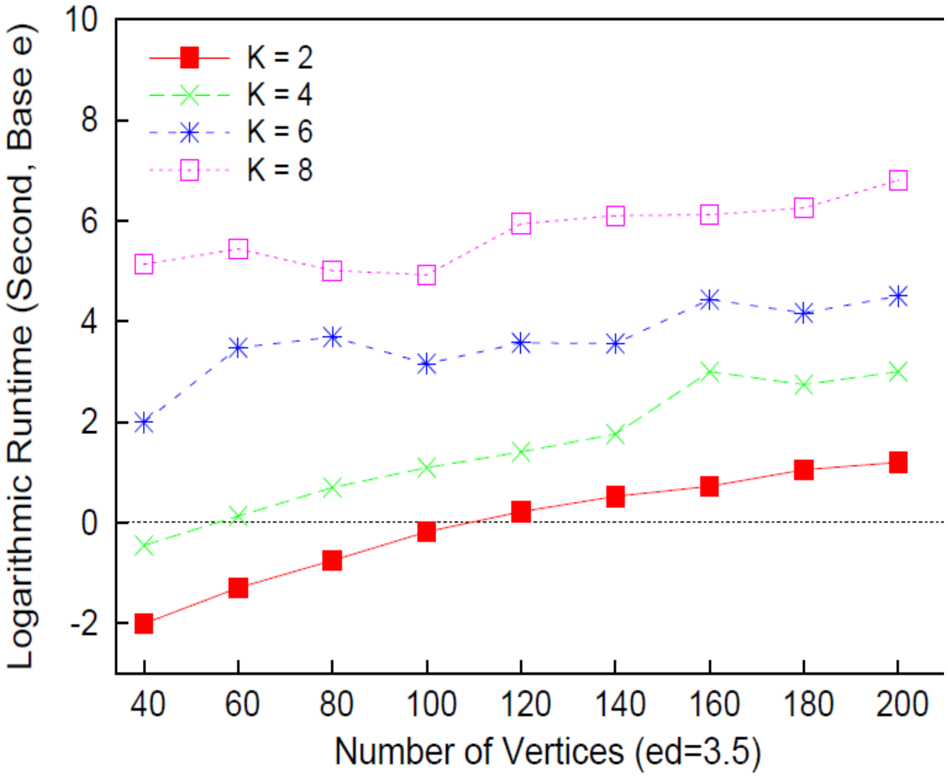
# Non-Reducible Graphs



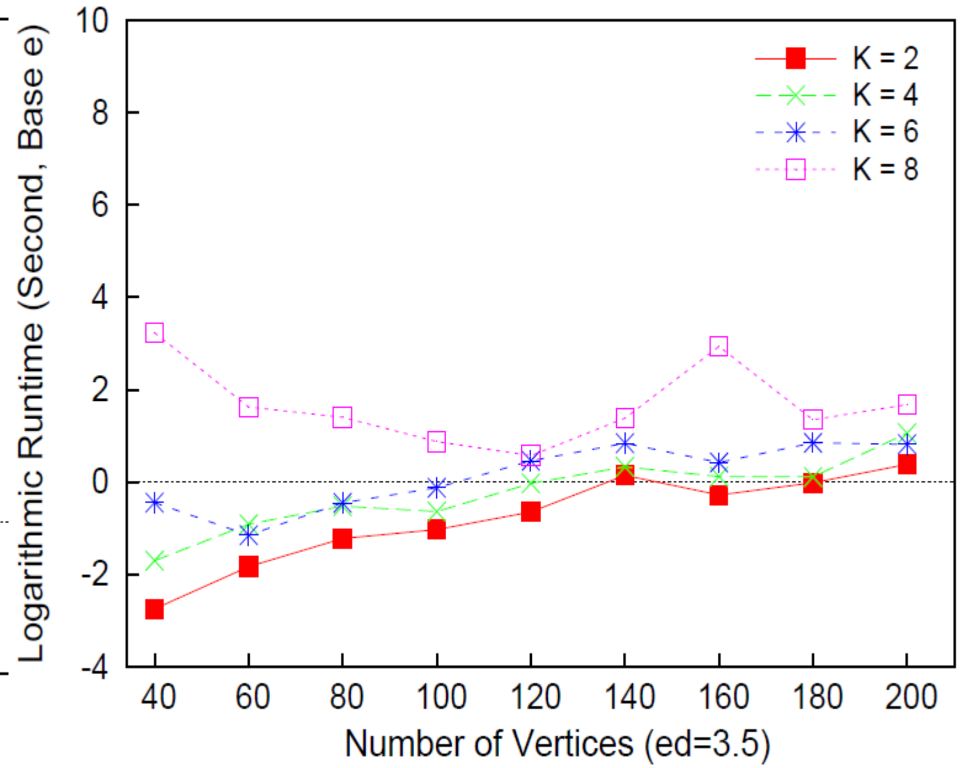

When can we safely reduce u, v, w?

# Runtime with Reductions
# (n small, k small)

No reductions



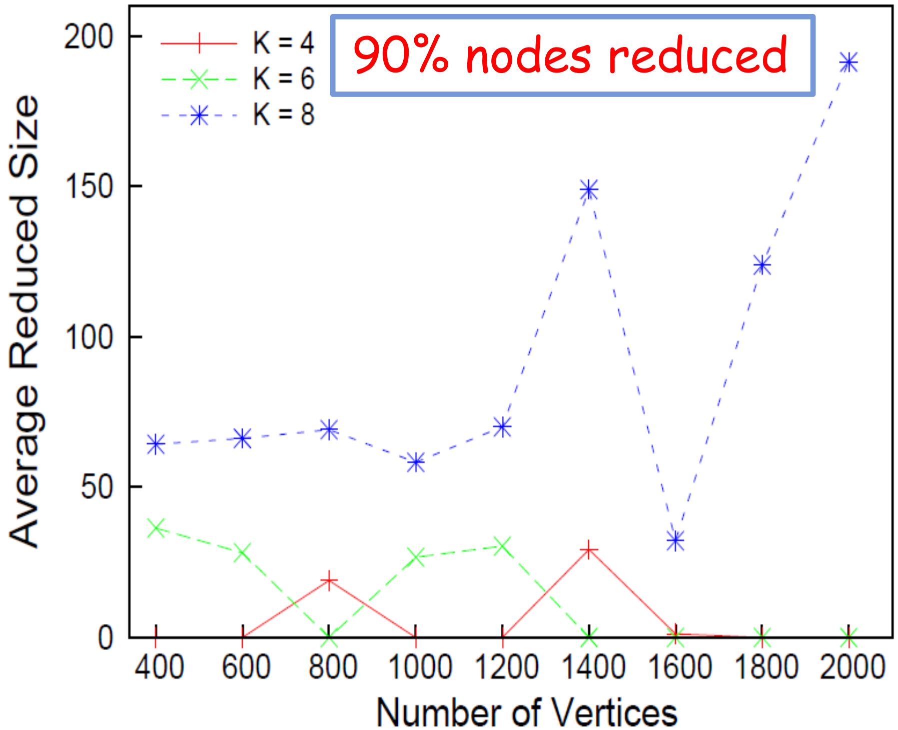

With reductions



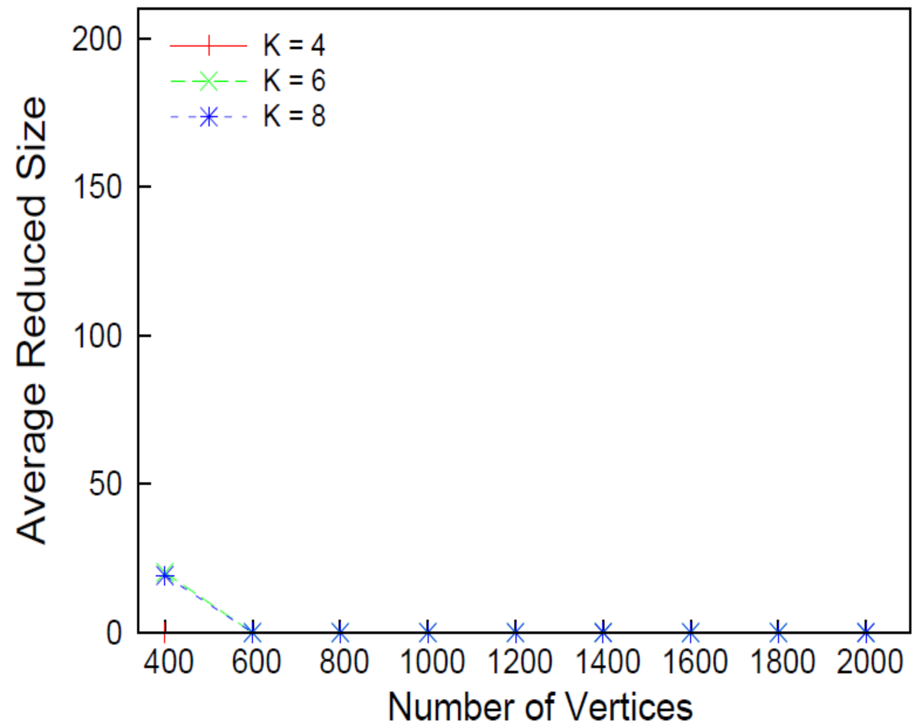

4X ~ 140X speedup

# Reduced Size
# (n large, k small)

ed = 2.0

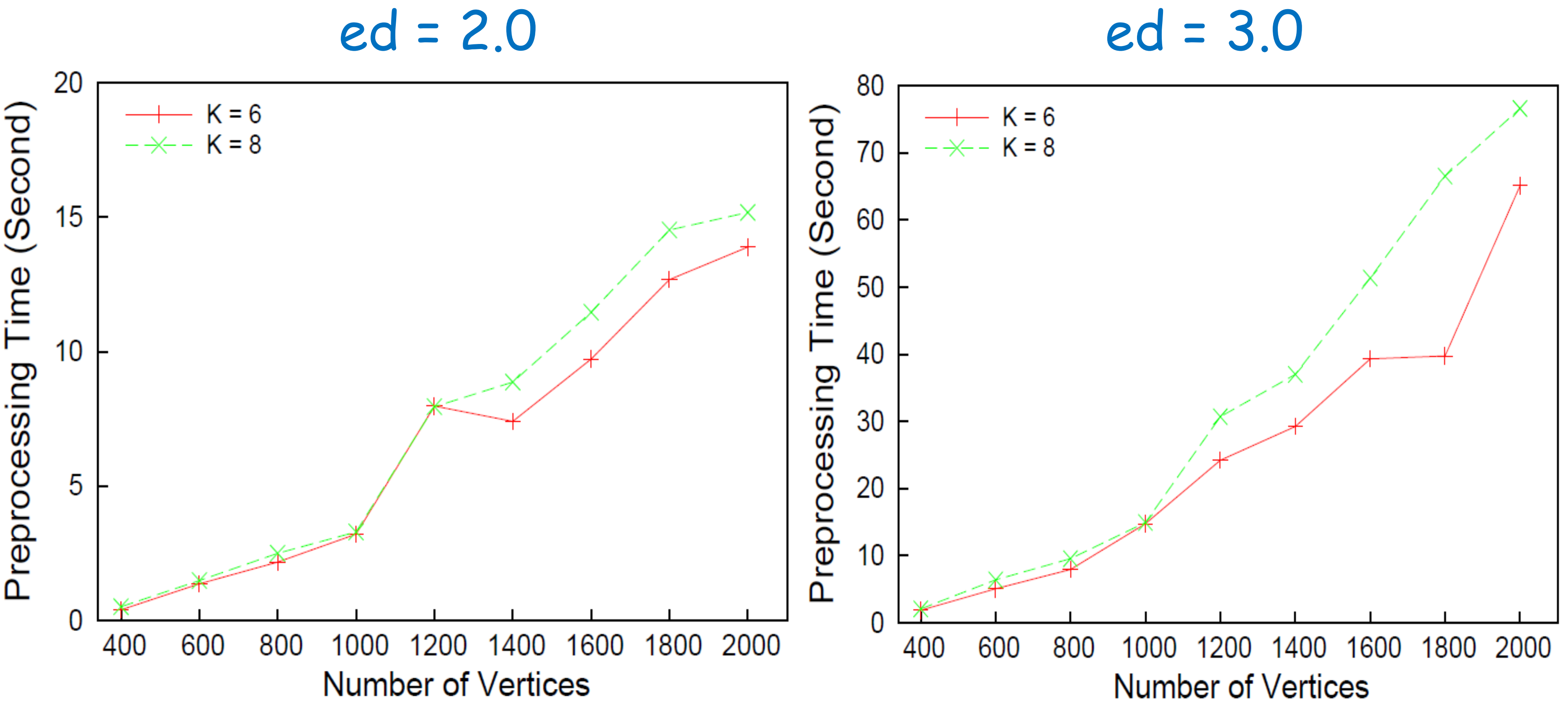



ed = 3.0




Many flowers found !

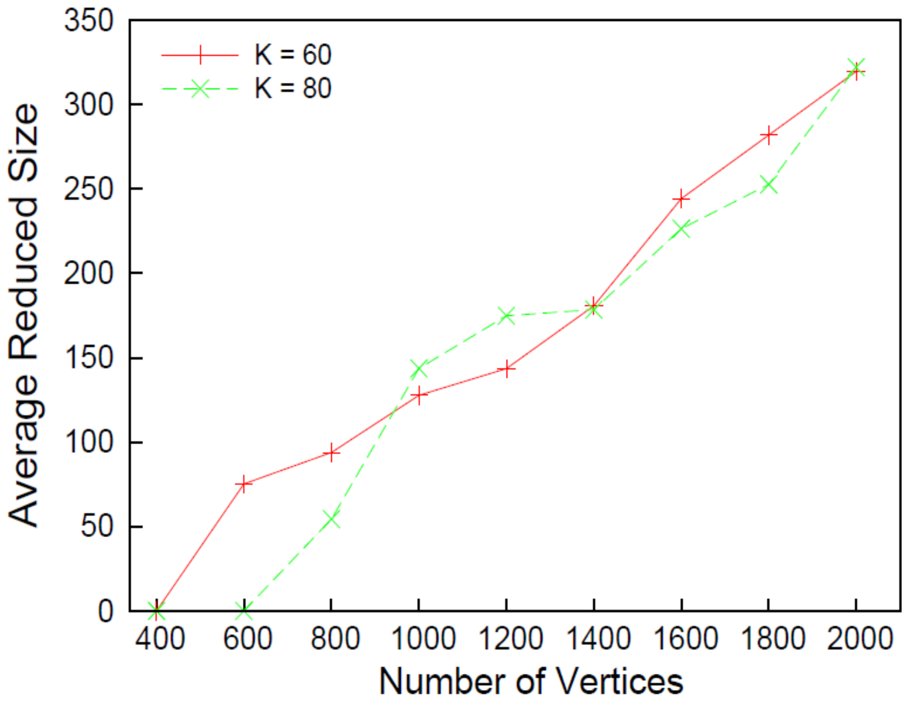
# Preprocessing Time
# (n large, k small)



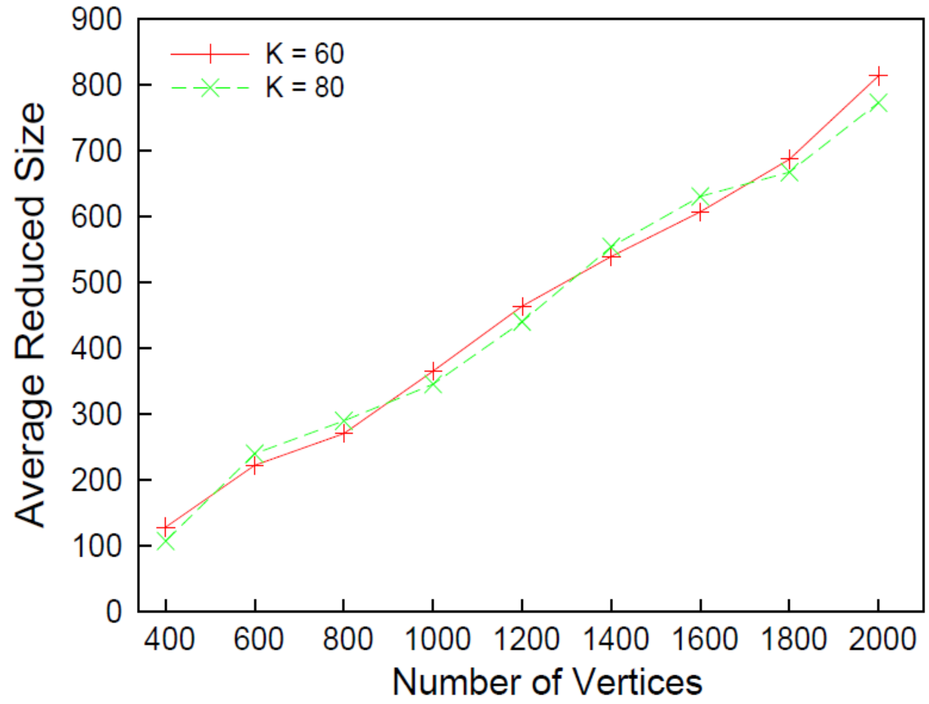

Scales linearly with n and ed

# Reduced Size
# (n large, k large)

ed = 2.0




ed = 3.0




Now flower reduction does not work well

# Which Rules are Powerful?

- n = 1000 and ed = 3.0

| *Rules* | $k = 4$ | $k = 40$ |
|---|---|---|
| *Chain* | 607 | 377 |
| *Dummy* | 785 | 679 |
| *Flower* | 996 | 1000 |
| *Chain+Dummy* | 522 | 377 |
| *Chain+Flower* | 366 | 377 |
| *Dummy+Flower* | 151 | 377 |
| *Dummy+Dummy+Flower* | 0 | 377 |

No flowers found…

# Start Heuristics

- Three heuristics:
  - Big-Degree

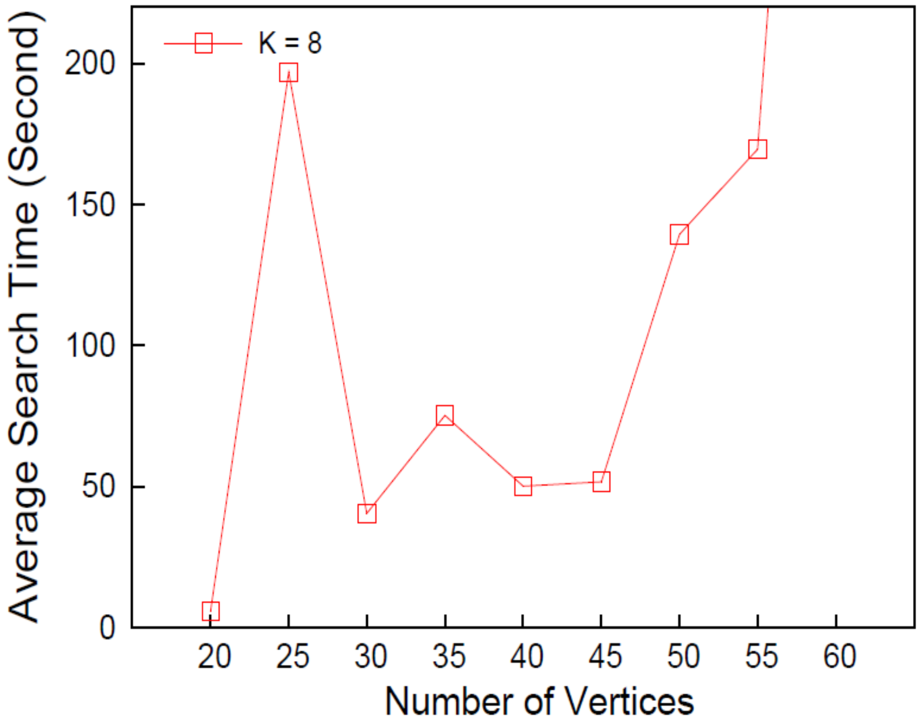
    pick biggest total degree vertices until acyclic
  - Even's Fractional Approximation

    pick the most heavy weight set
  - Even's Full Approximation

    pick the approximation solution

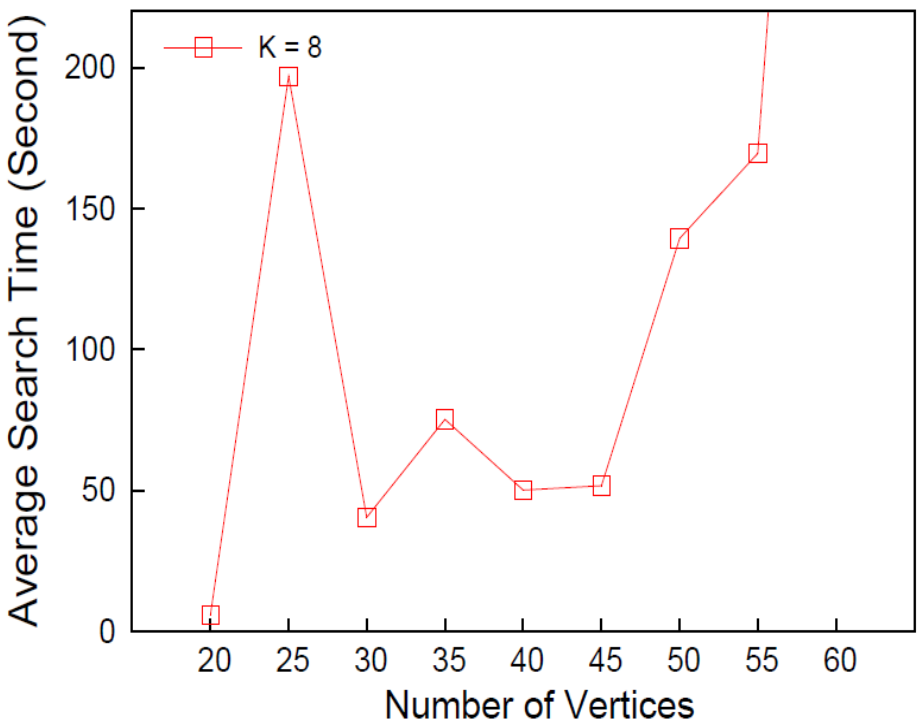
# Evaluating Heuristics

## No heuristic

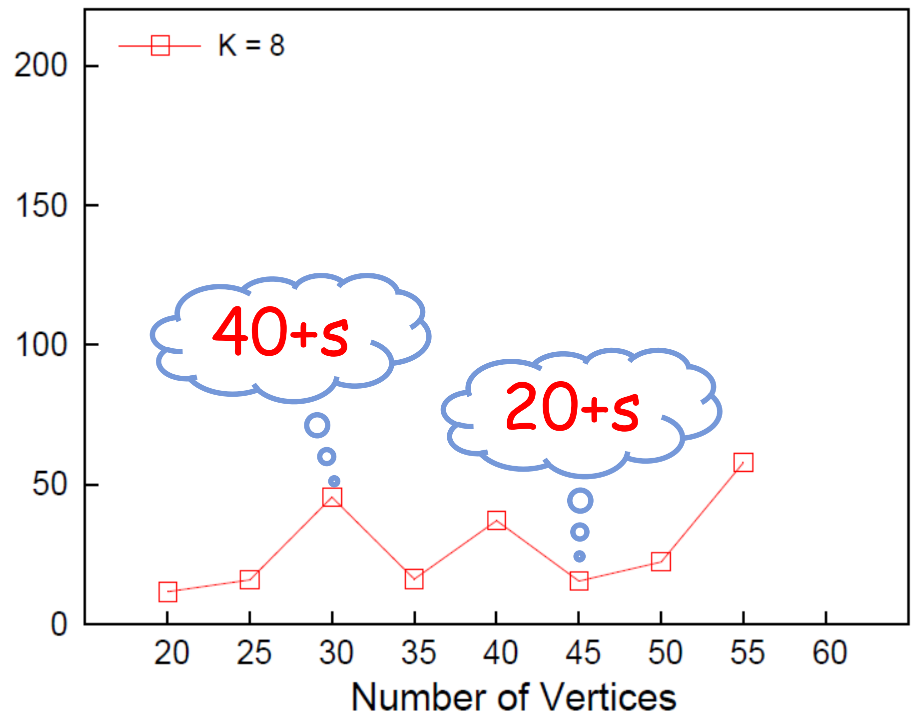


# Evaluating Heuristics

No heuristic

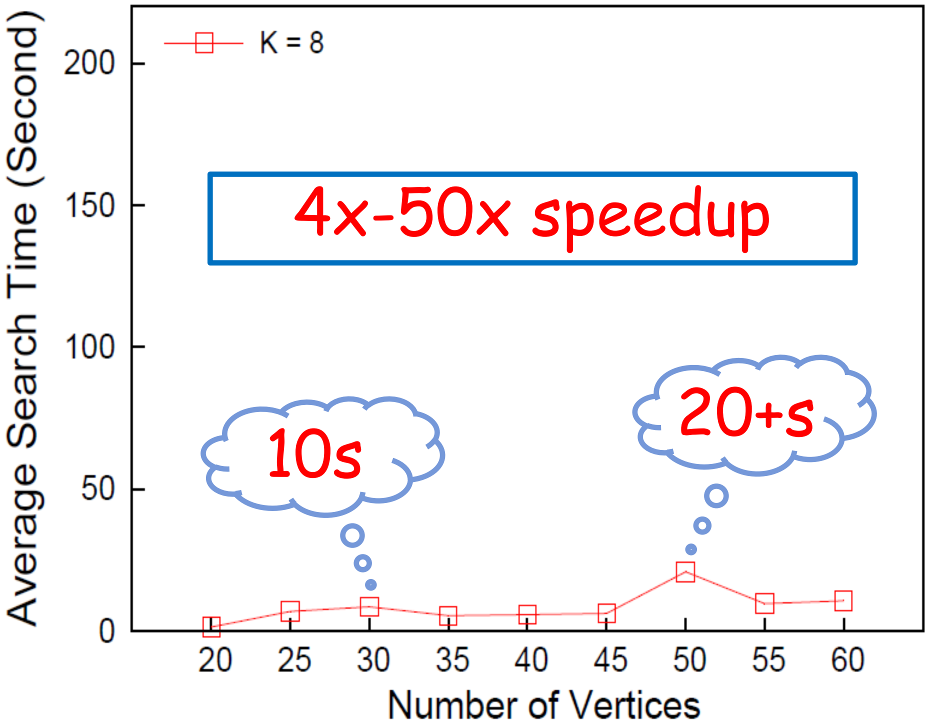



Fractional Approximation

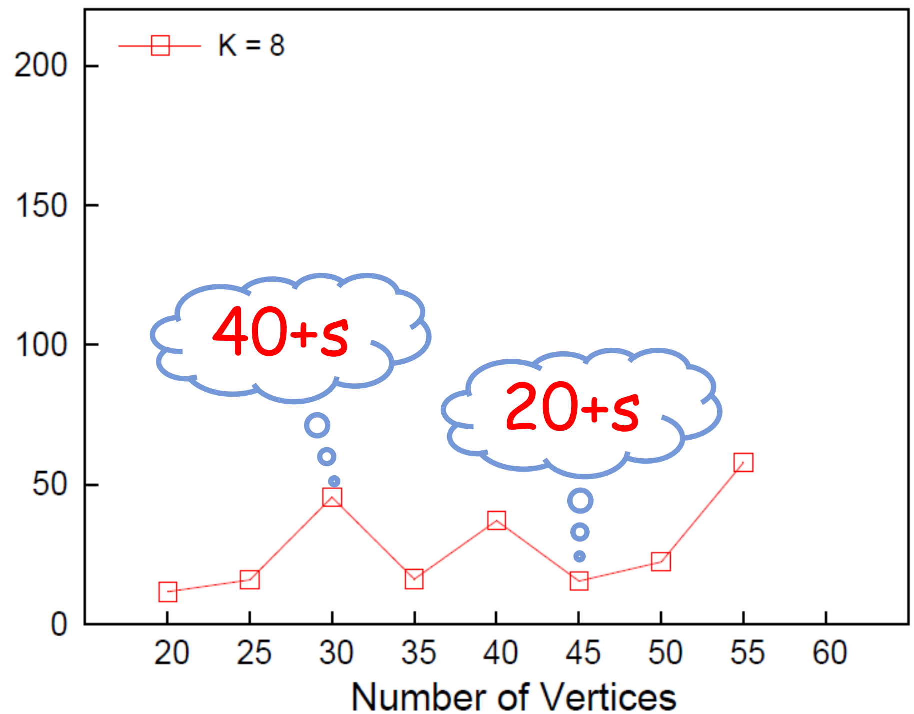


# Evaluating Heuristics

## Big Degree

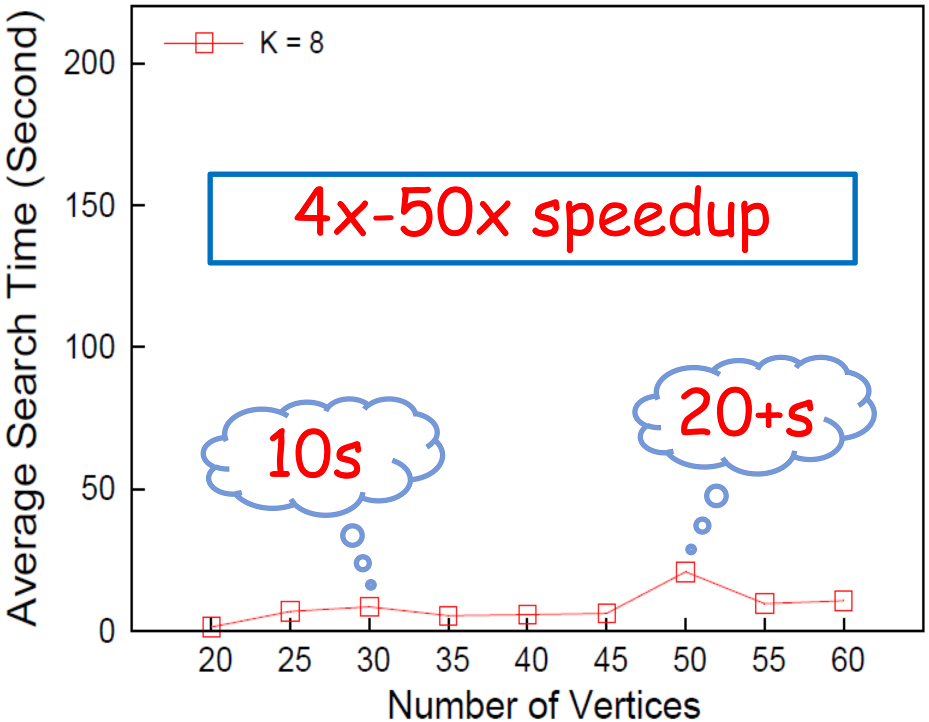



## Fractional Approximation

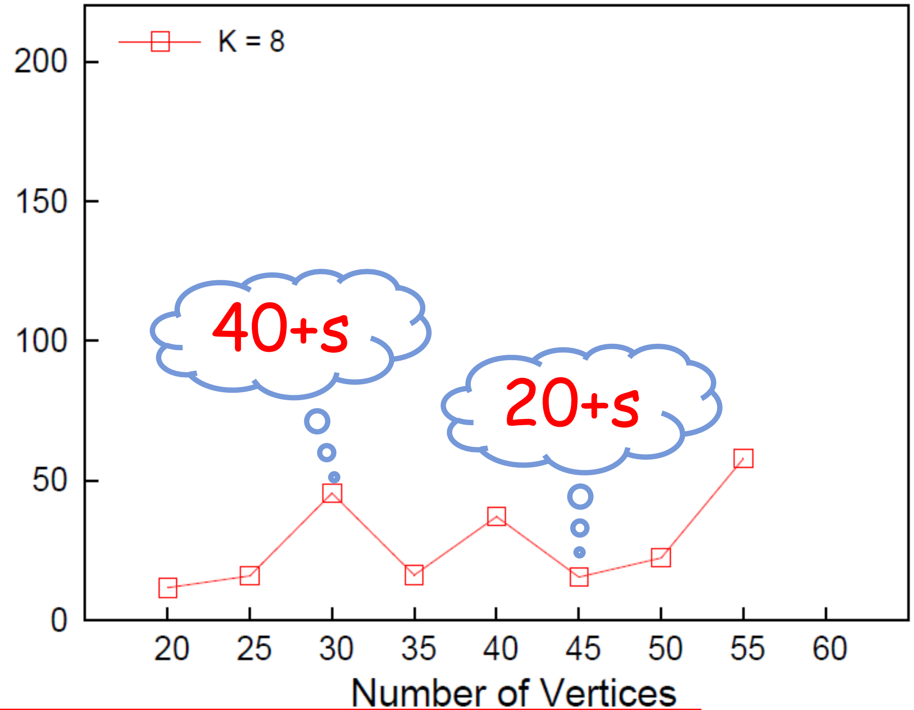


# Evaluating Heuristics

Big Degree



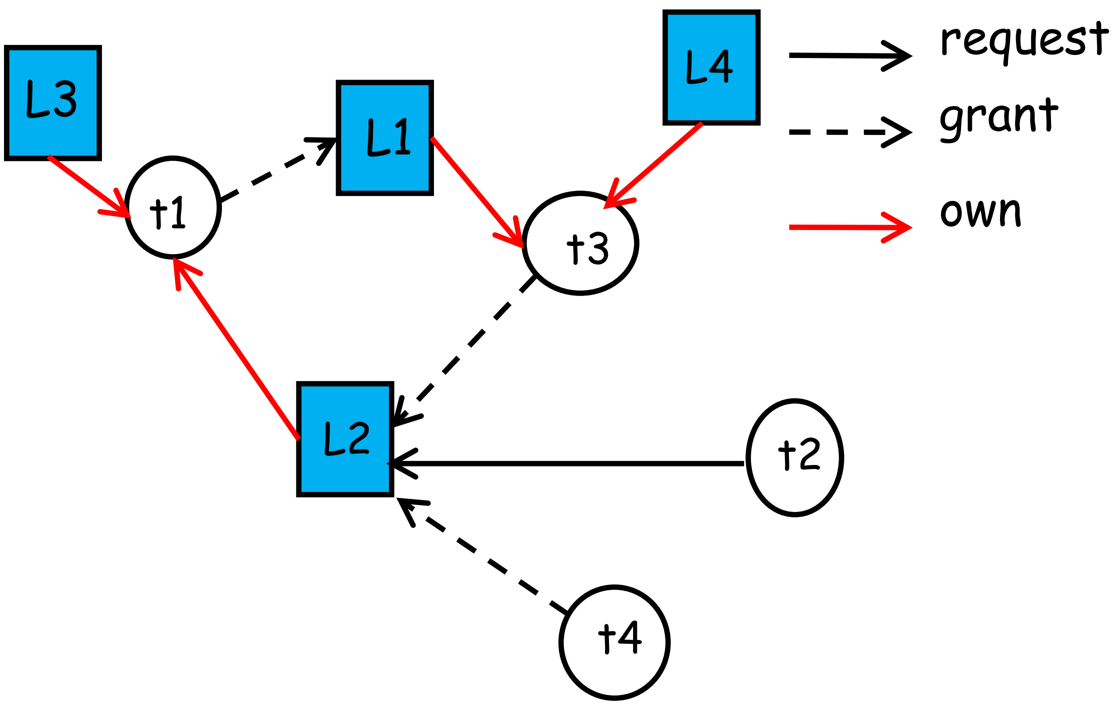

Fractional Approximation



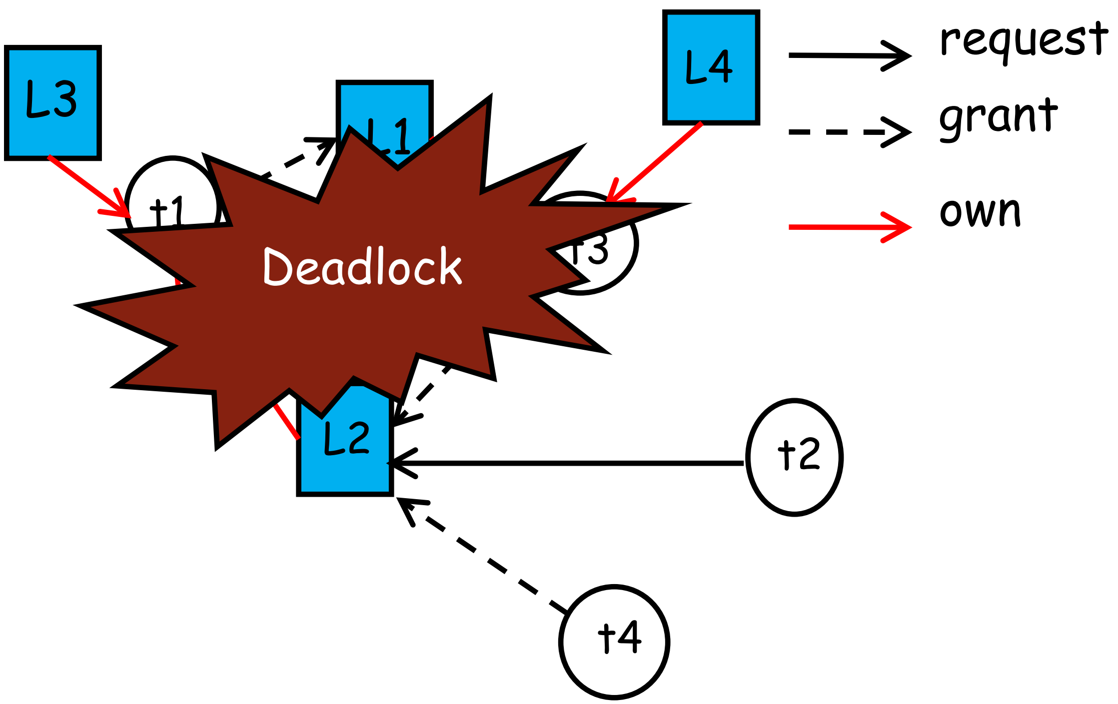

Practical Scheme:

Preprocessing + Big-Degree + Chen's algorithm --> further 2-3x speedup

# Practical Application: Deadlock Recovery

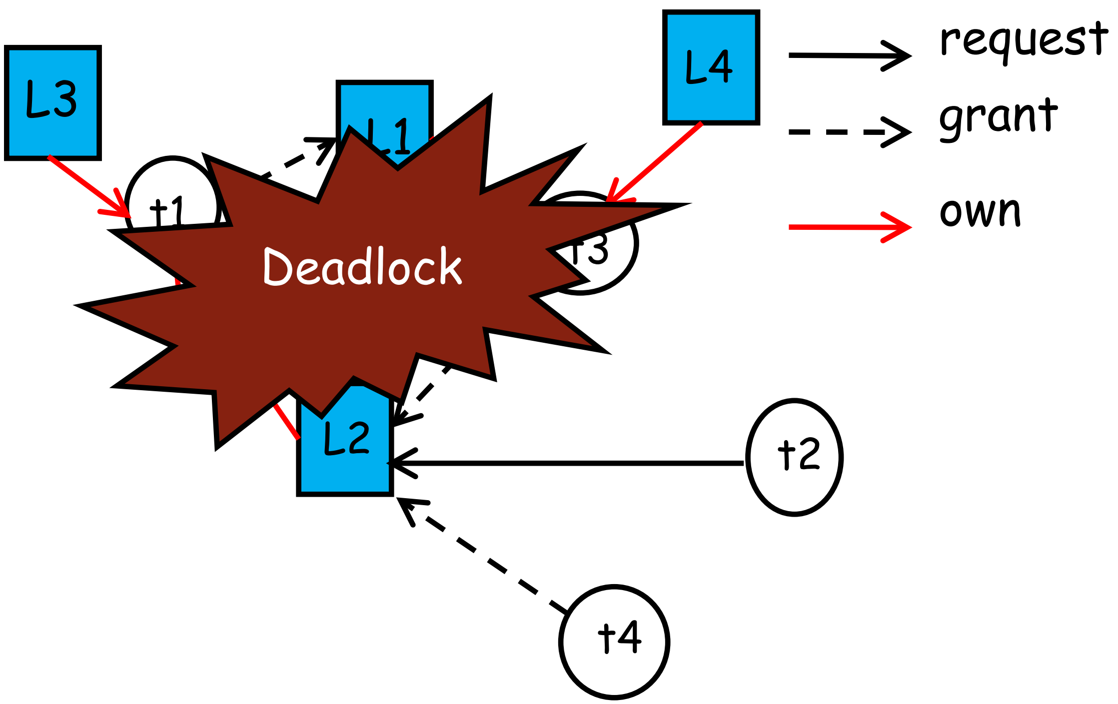


# Practical Application:
# Deadlock Recovery



# Practical Application: Deadlock Recovery

Could DFVS help us for deadlock recovery?



# Practical Application:
# Deadlock Recovery

Could DFVS help us for deadlock recovery?

R1: one lock owned
by only one thread

R2: one thread can
wait on only one lock

# Practical Application: Deadlock Recovery

Could DFVS help us for deadlock recovery?

R1: one lock owned by only one thread

R2: one thread can wait on only one lock

No overlapping cycles

Cycle detection is enough

# A Real System

- The Deadlock Immunity System
  - OSDI '08 (top system conference)
  - Use cycle detection to enable deadlock immunity
  - **10%** overhead on average
    - instrumentations, framework overhead, etc.

# Conclusion

- Quantitative analysis of Chen's FPT algorithm for DFVS
- New reduction rules
  - With significant performance benefits
  - Quantitative analysis

# Open Problems

- Reduction rules when $2 \leq |petal(u)| \leq k$ ?

  $|petal(u)| \geq (k+1)$, flower rule

  $|petal(u)| = 1$, shortcut

- Kernelization for DFVS problem

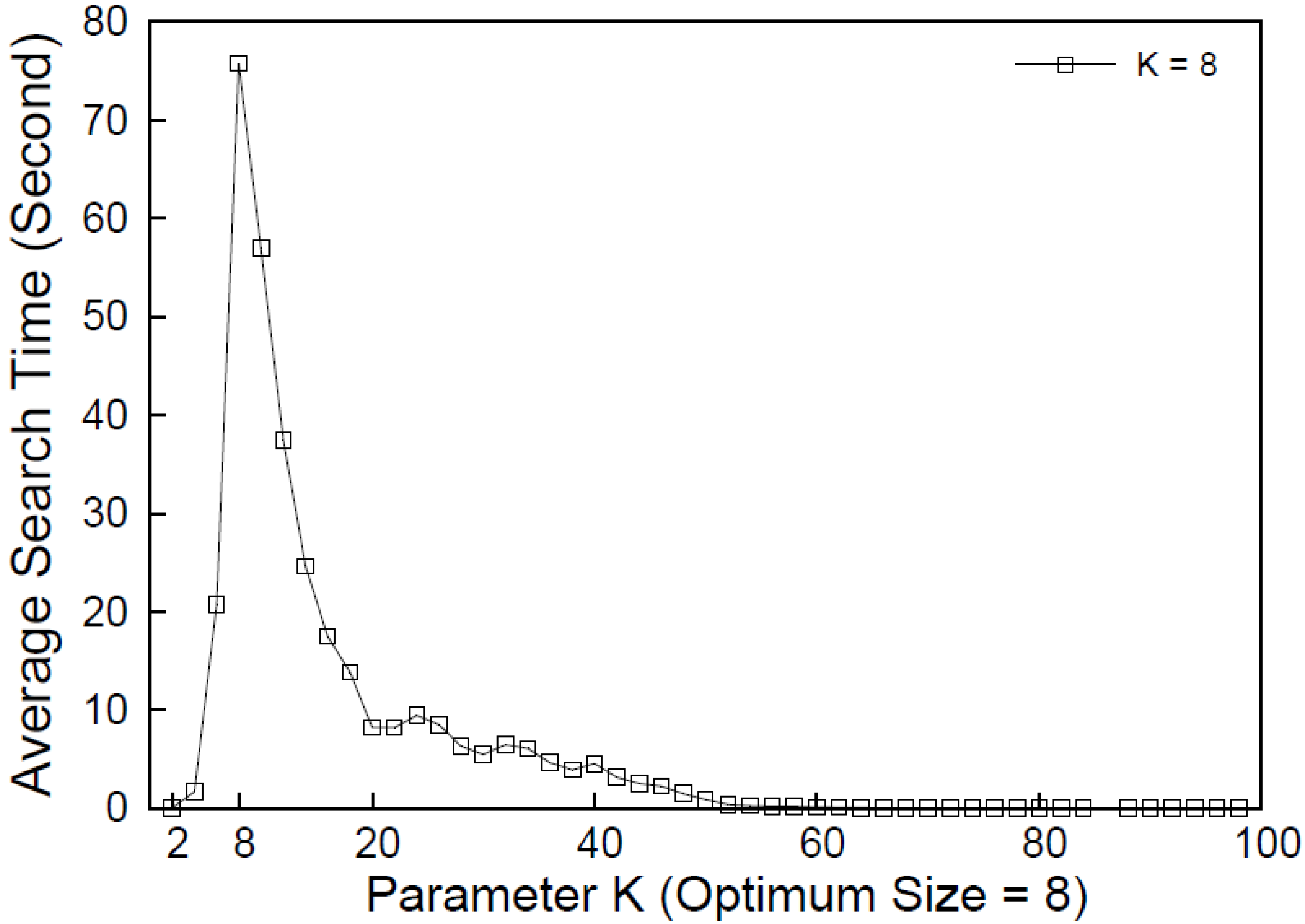
- Better heuristics

  Better approximation algorithm?

# Thanks !

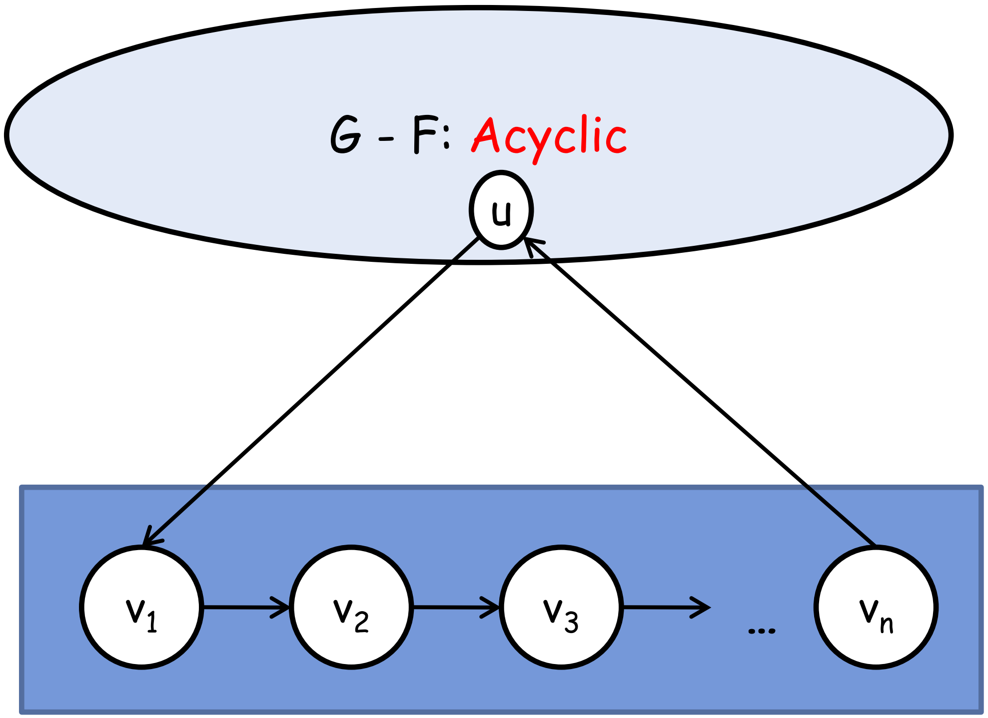
?

# FPT search with different parameter k

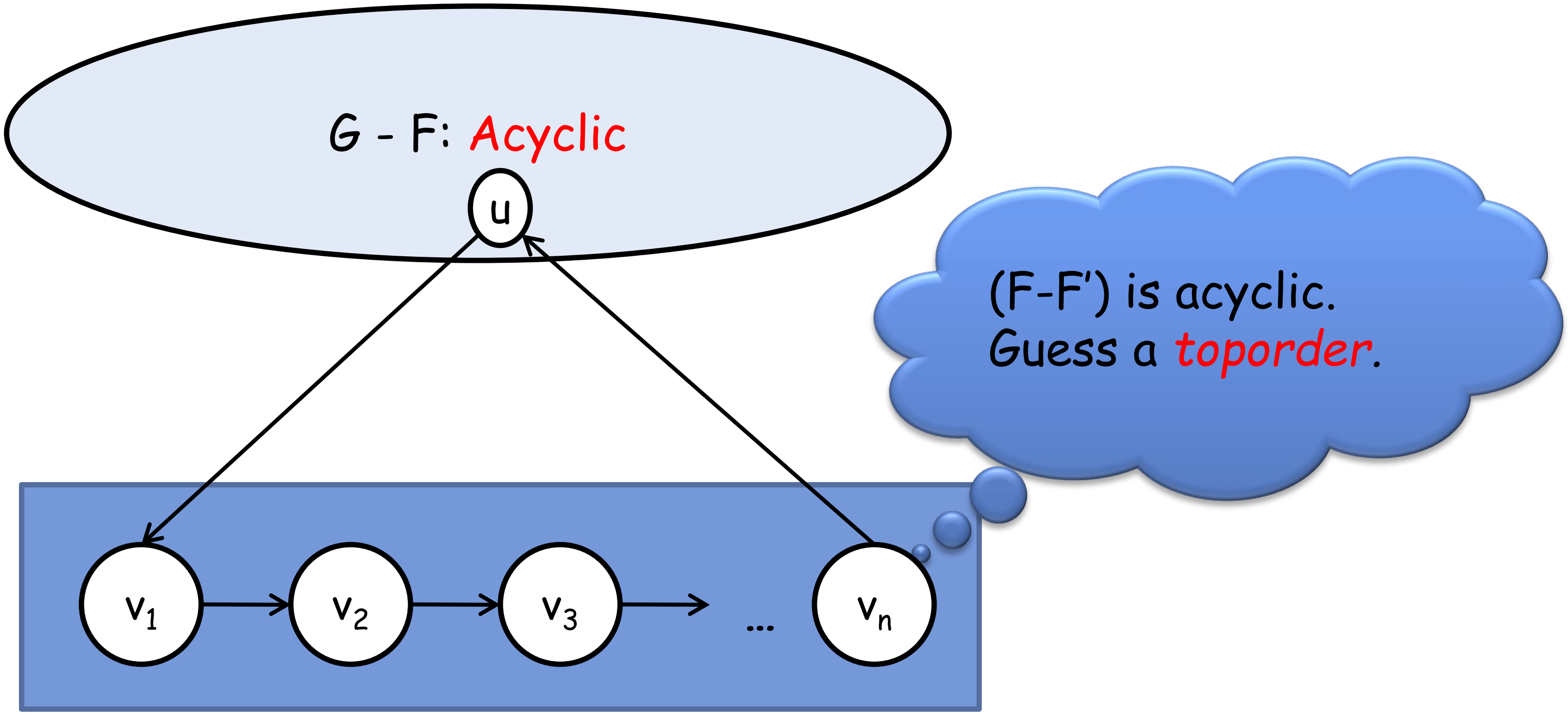


# Backup Slides

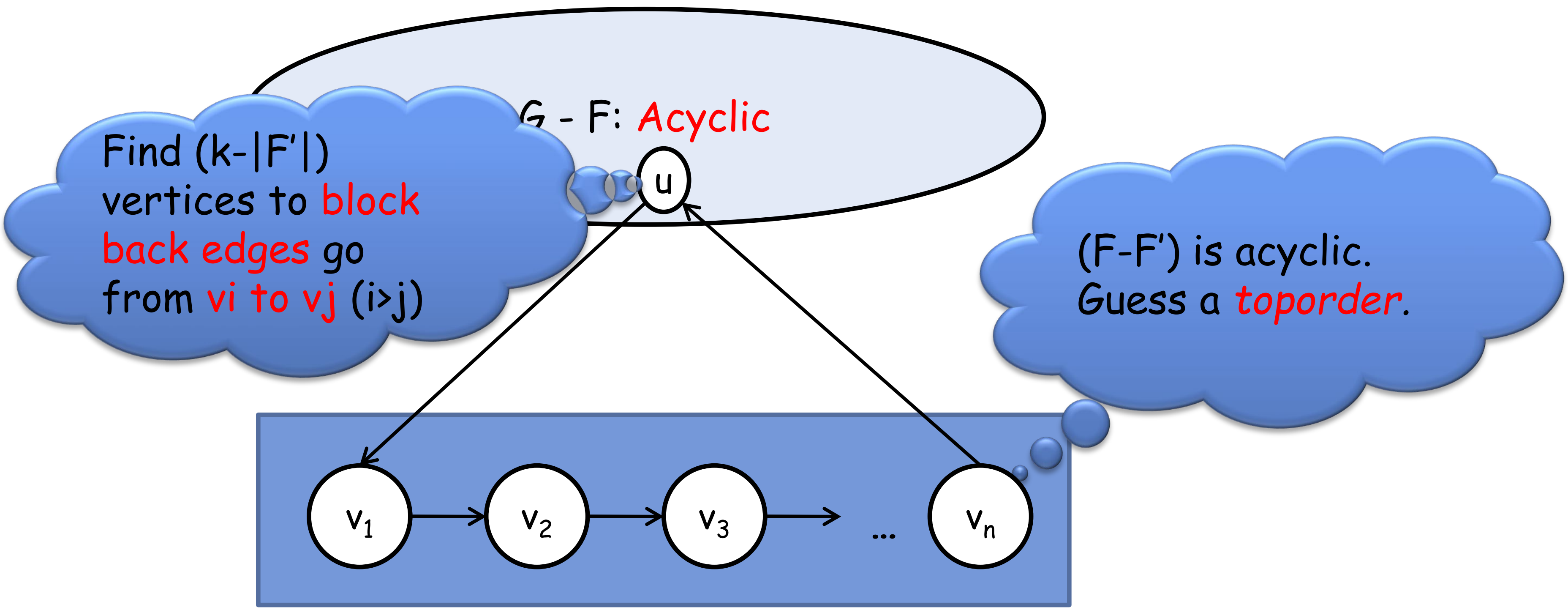
# Skew Separator



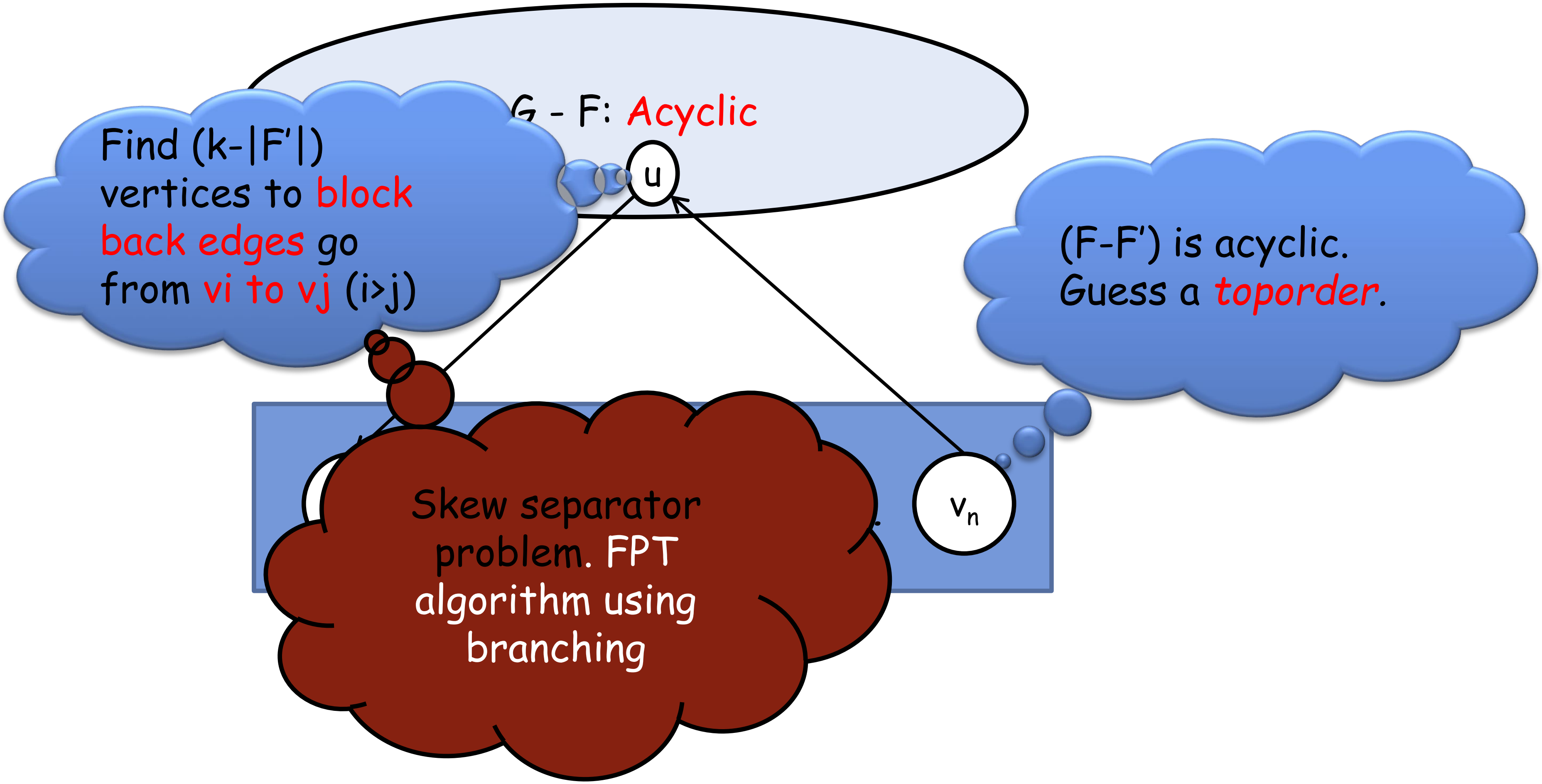
# Skew Separator



# Skew Separator

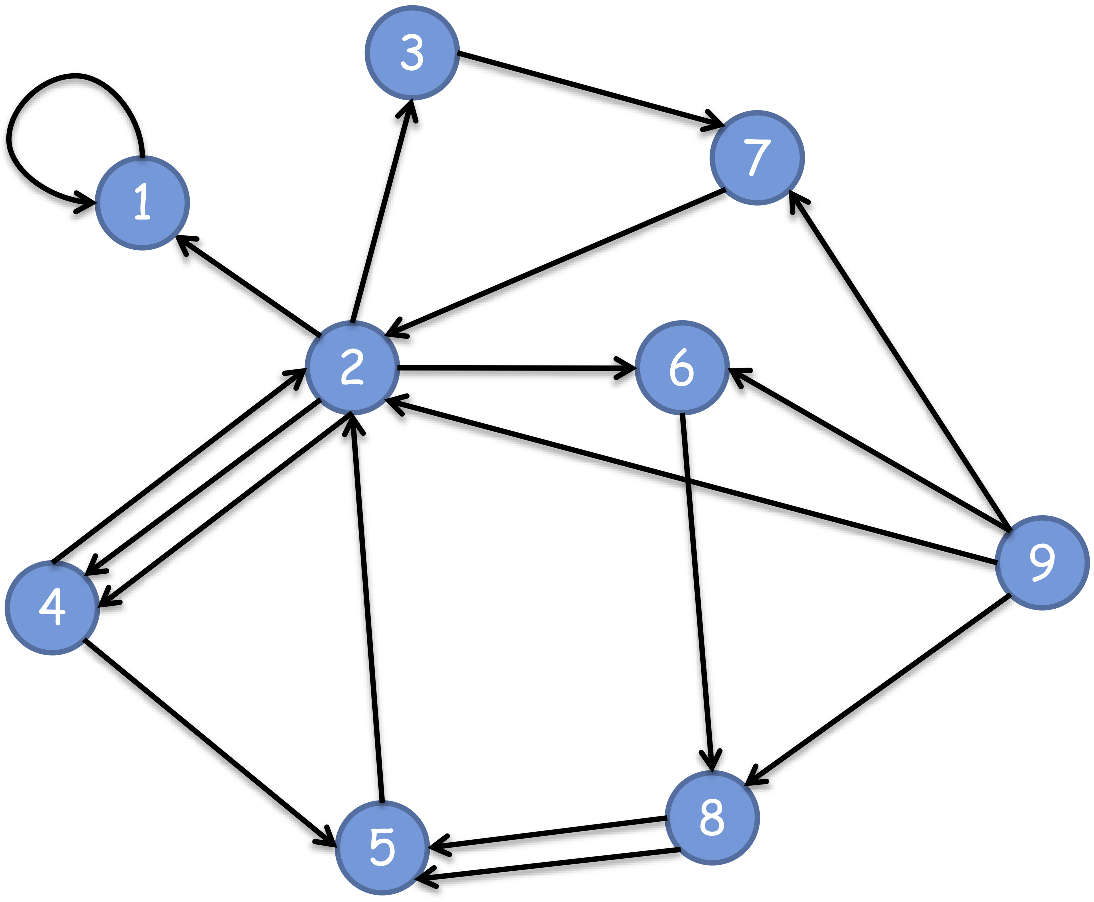


# Skew Separator



# Example of Applying Reductions

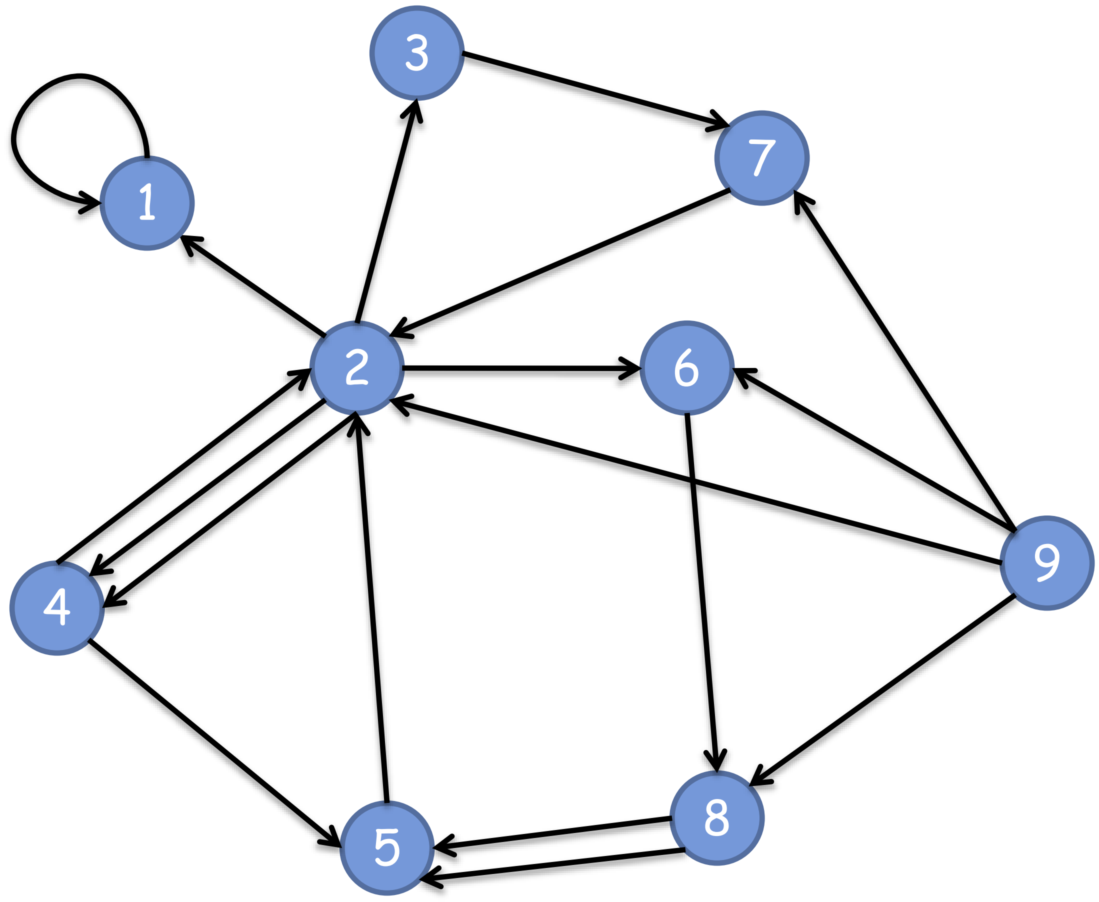
K = 2. FVS:



# Example of Applying Reductions

K = 2. FVS:

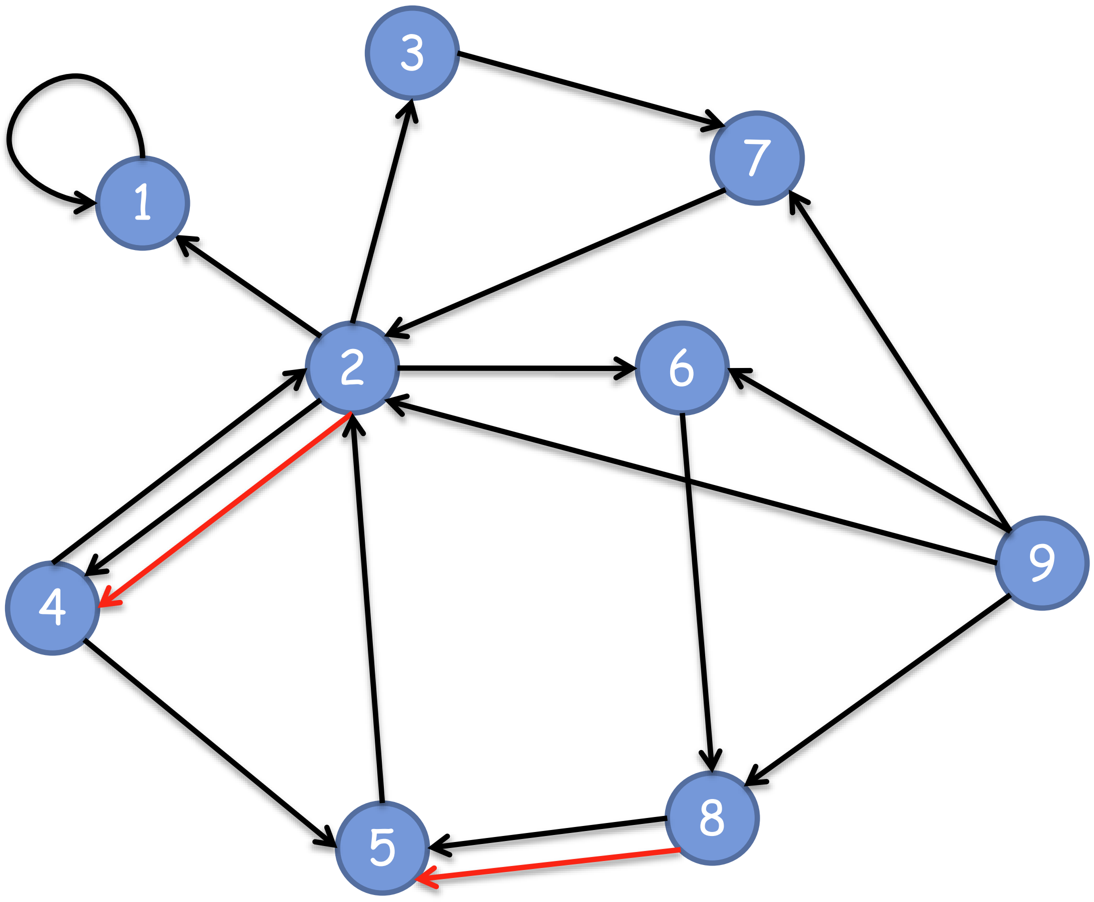
Delete Parallel Edges



# Example of Applying Reductions

K = 2. FVS:

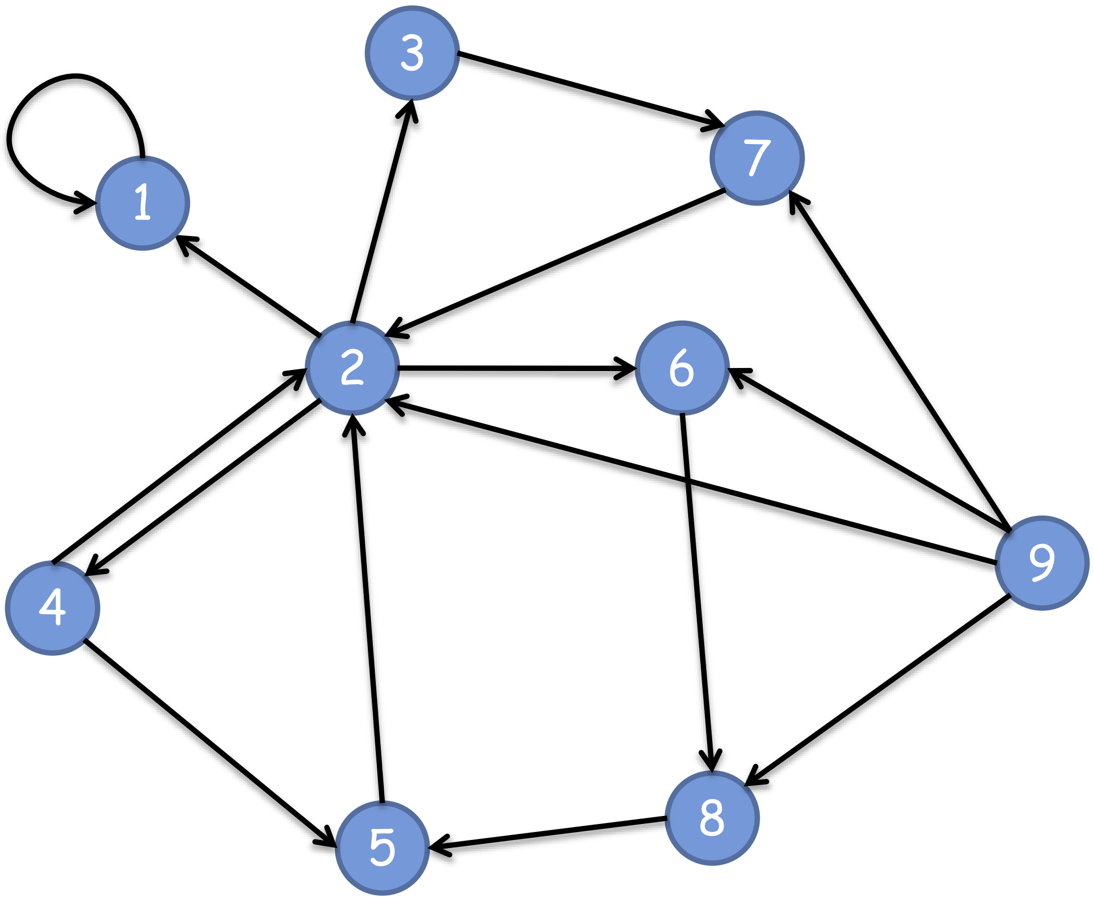
Delete Parallel Edges



# Example of Applying Reductions

K = 2. FVS:

Delete Parallel Edges

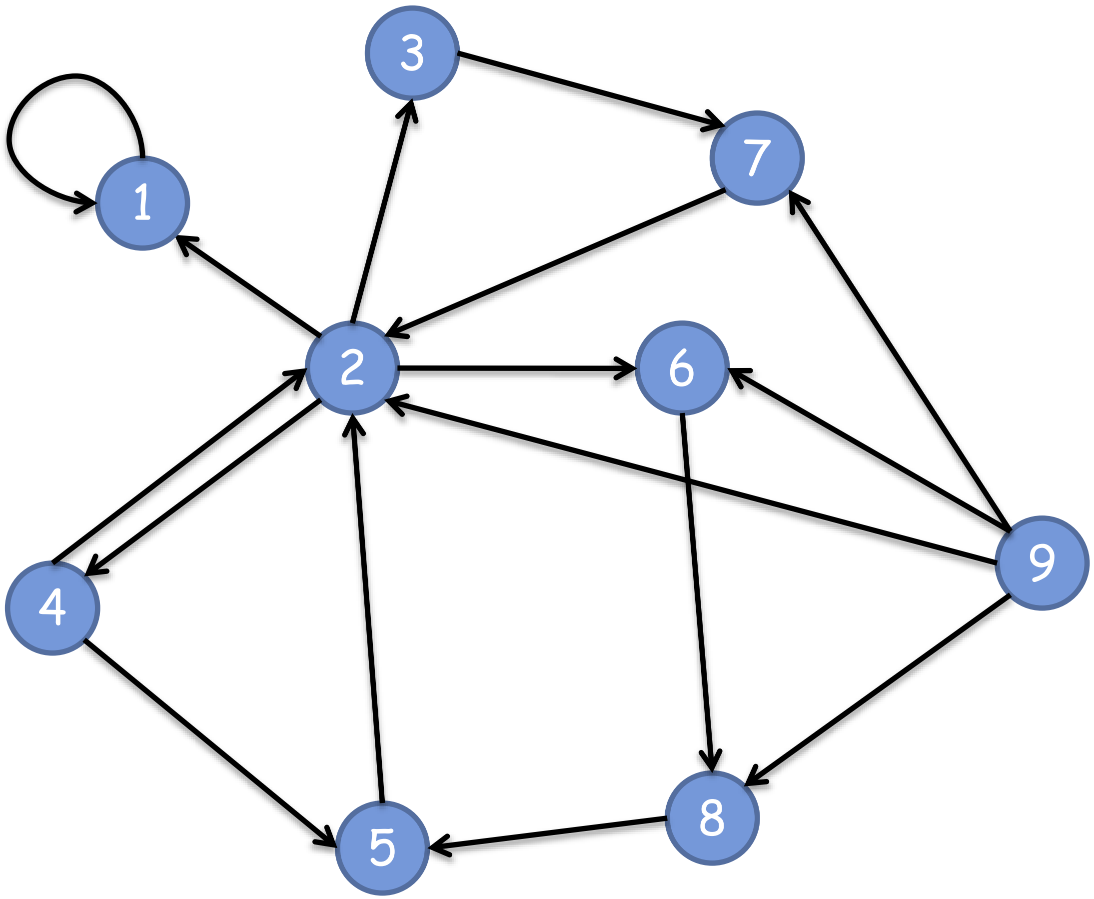


# Example of Applying Reductions

K = 2. FVS:

Delete Parallel Edges

Delete Self Loops

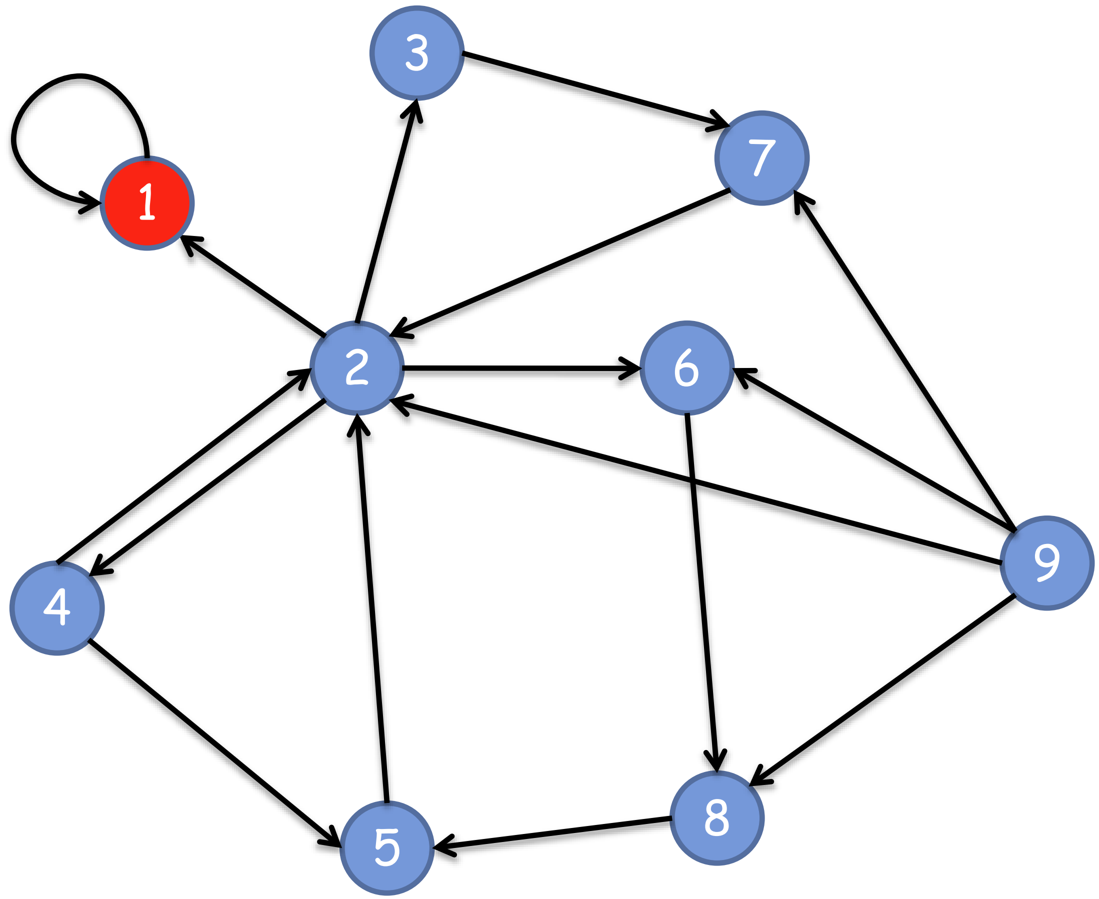


# Example of Applying Reductions

K = 2. FVS:

Delete Parallel Edges

Delete Self Loops

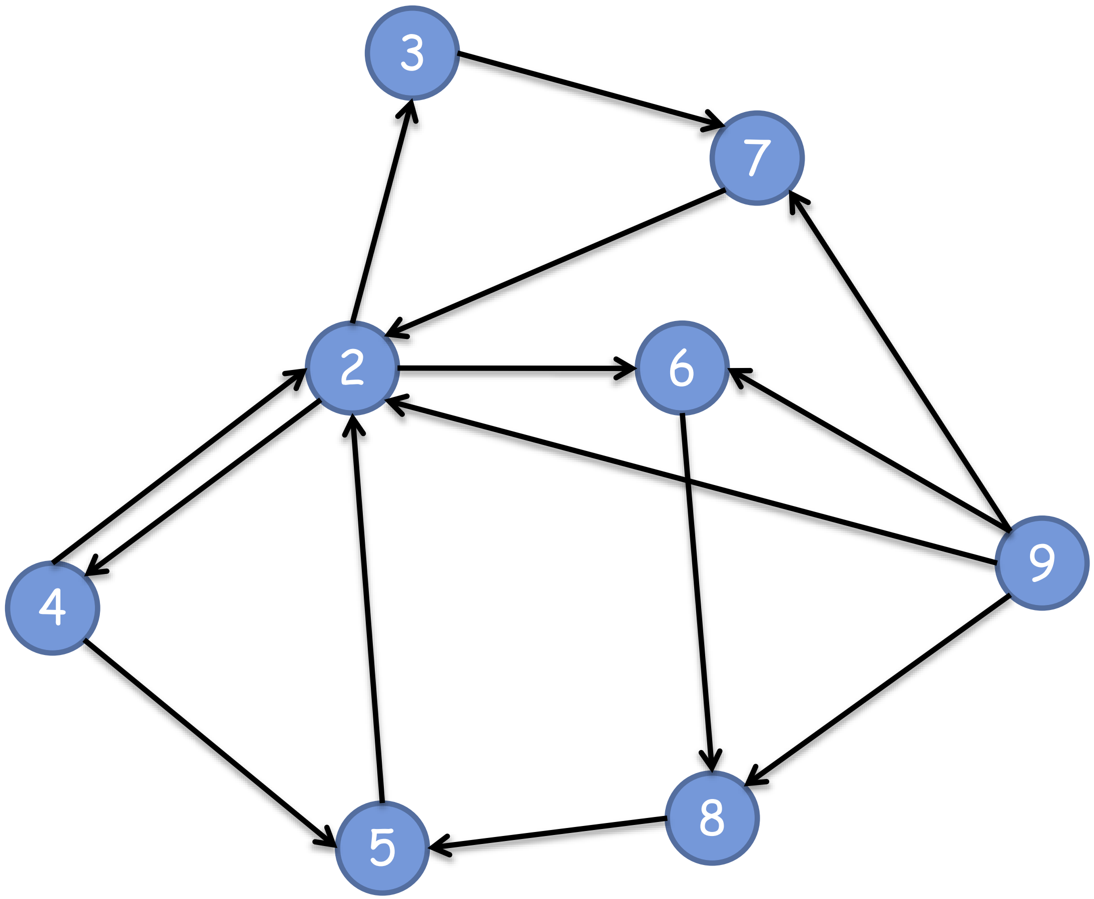


# Example of Applying Reductions

K = 2. FVS:

Delete Parallel Edges

Delete Self Loops



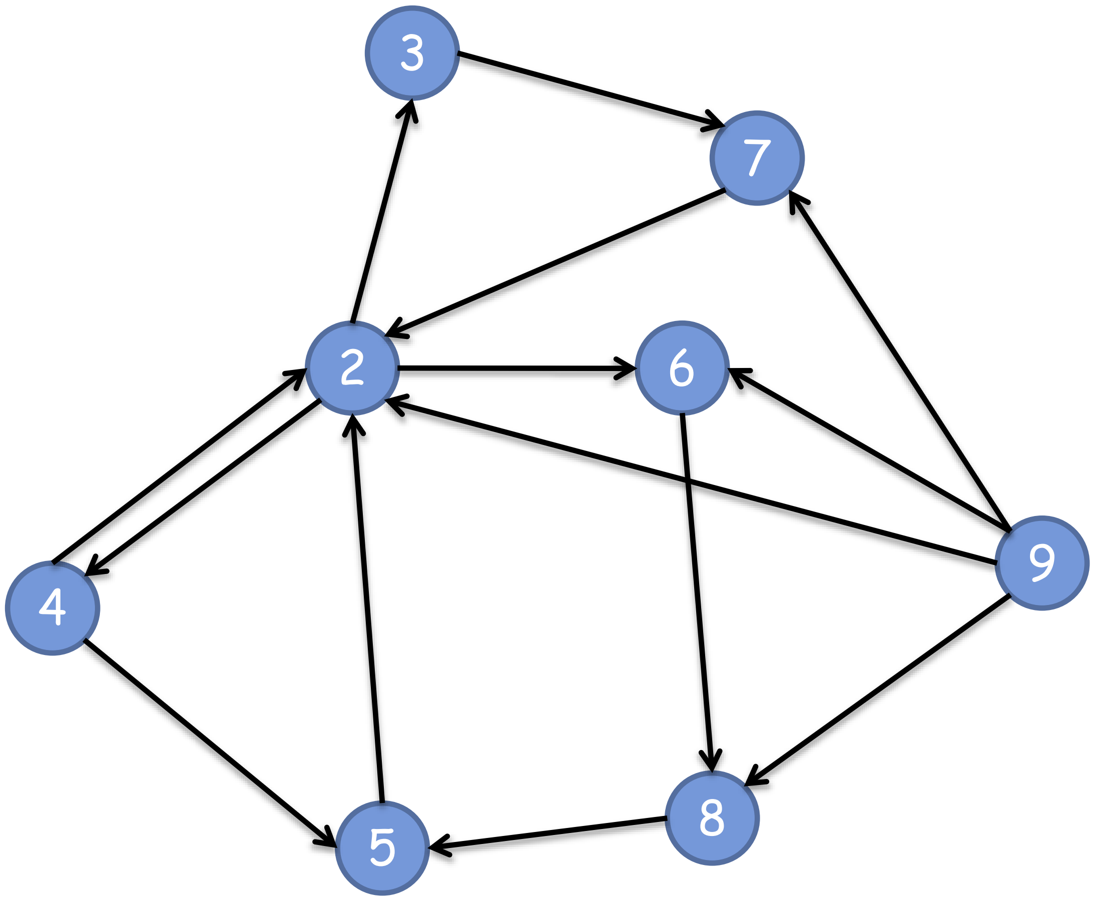
# Example of Applying Reductions

K = 2. FVS:

Delete Parallel Edges

Delete Self Loops

K = 1



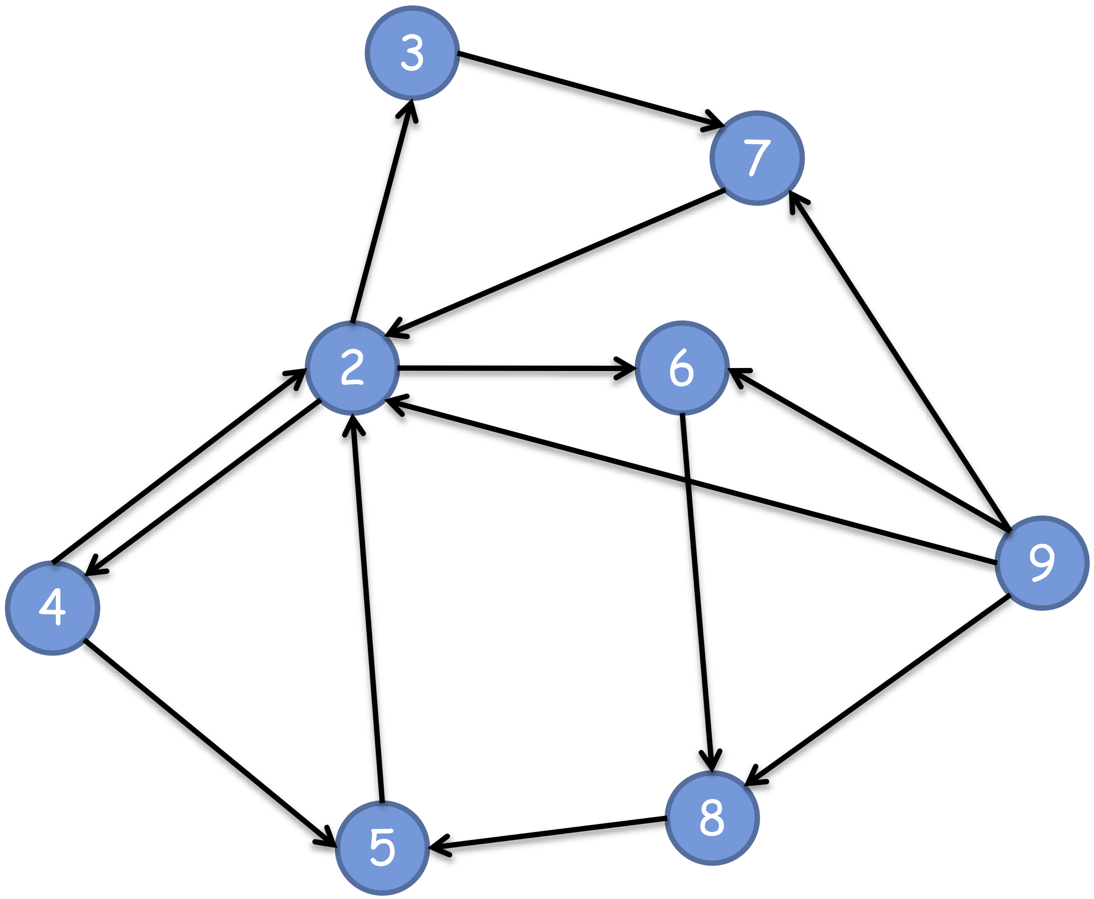
# Example of Applying Reductions

K = 2. FVS: 1

Delete Parallel Edges

Delete Self Loops

K = 1



# Example of Applying Reductions

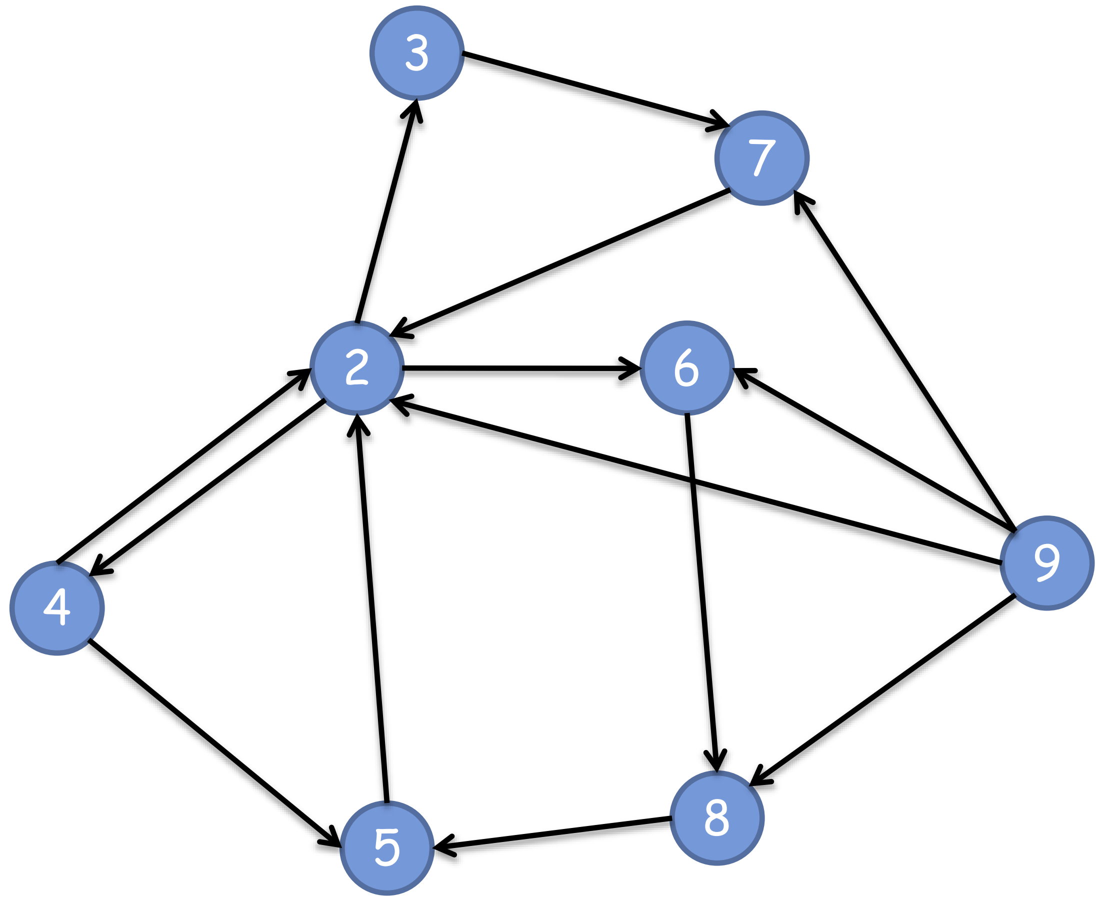
K = 2. FVS:


Delete Parallel Edges

Delete Self Loops

K = 1

Delete Dummy



# Example of Applying Reductions

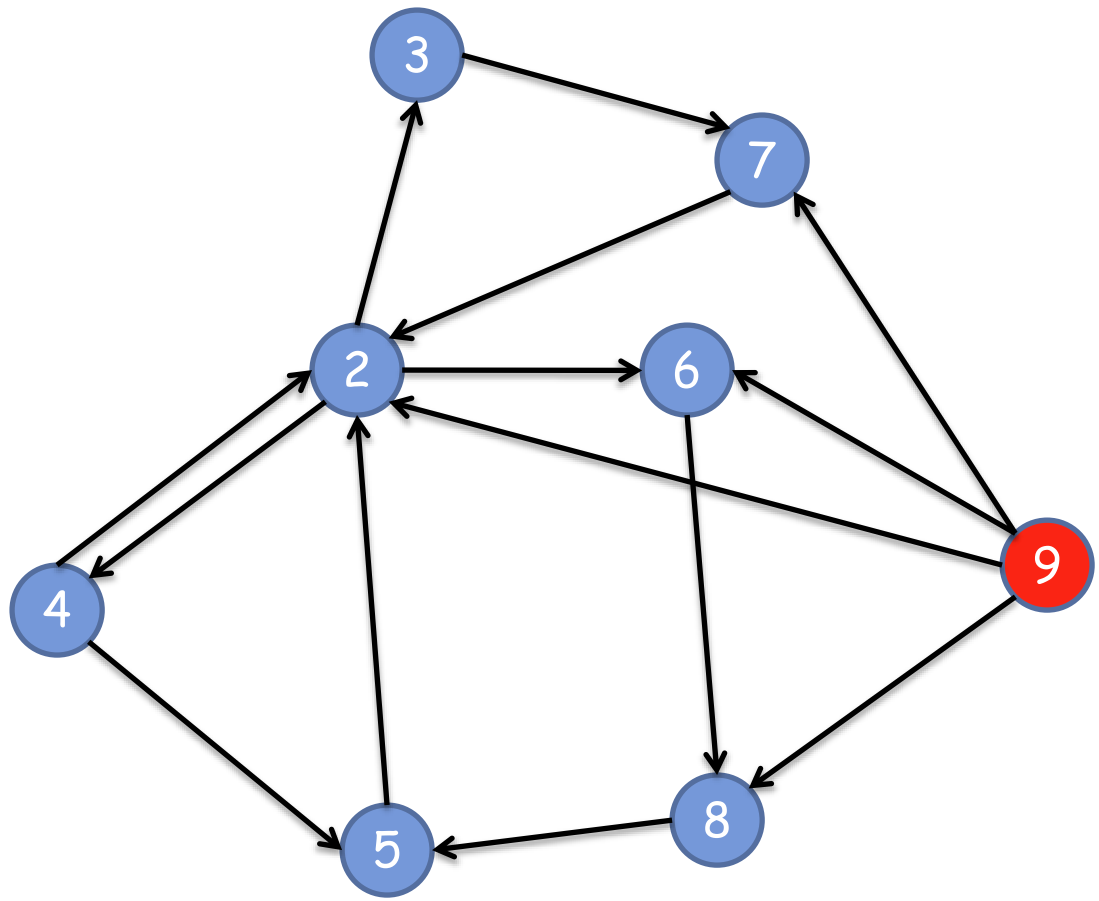
K = 2. FVS: 1

Delete Parallel Edges

Delete Self Loops

K = 1

Delete Dummy



# Example of Applying Reductions

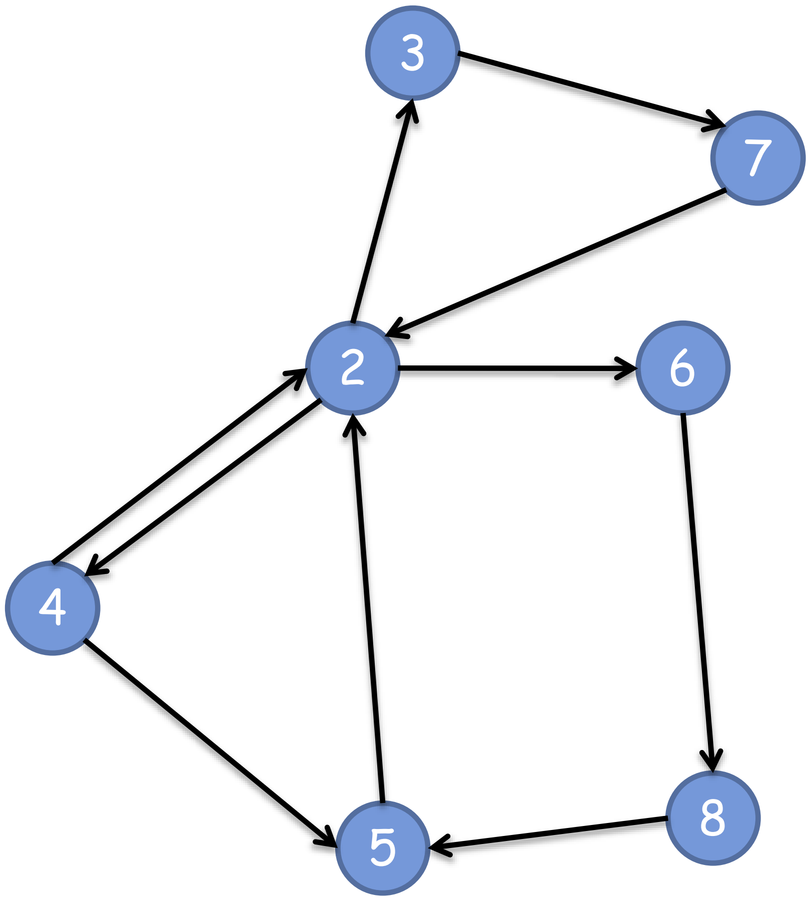
K = 2. FVS: 1

Delete Parallel Edges

Delete Self Loops

K = 1

Delete Dummy



# Example of Applying Reductions

K = 2. FVS: 1

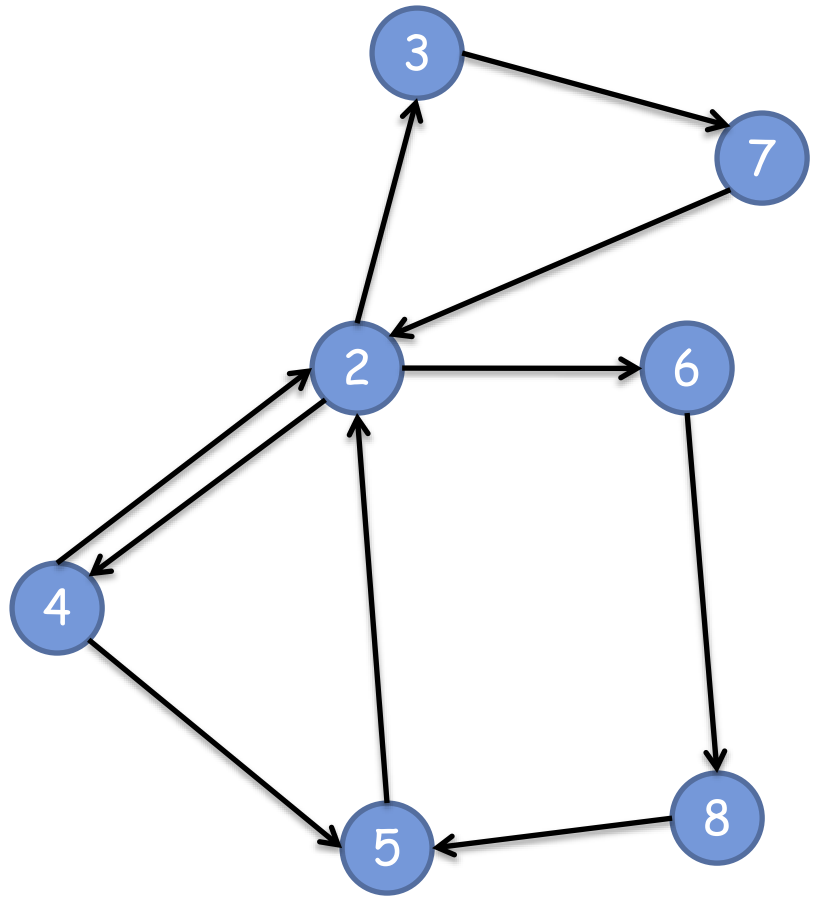
Delete Parallel Edges

Delete Self Loops

K = 1

Delete Dummy

Find a Flower



# Example of Applying Reductions

K = 2. FVS: 1

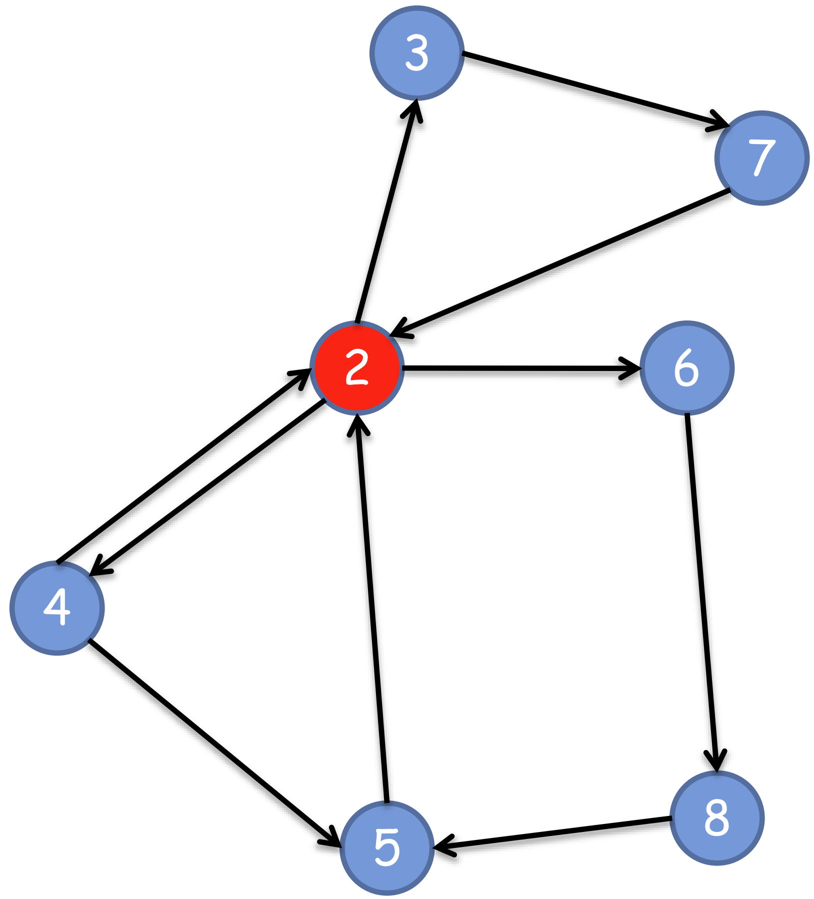
Delete Parallel Edges

Delete Self Loops

K = 1

Delete Dummy

Find a Flower



# Example of Applying Reductions

K = 2. FVS: 

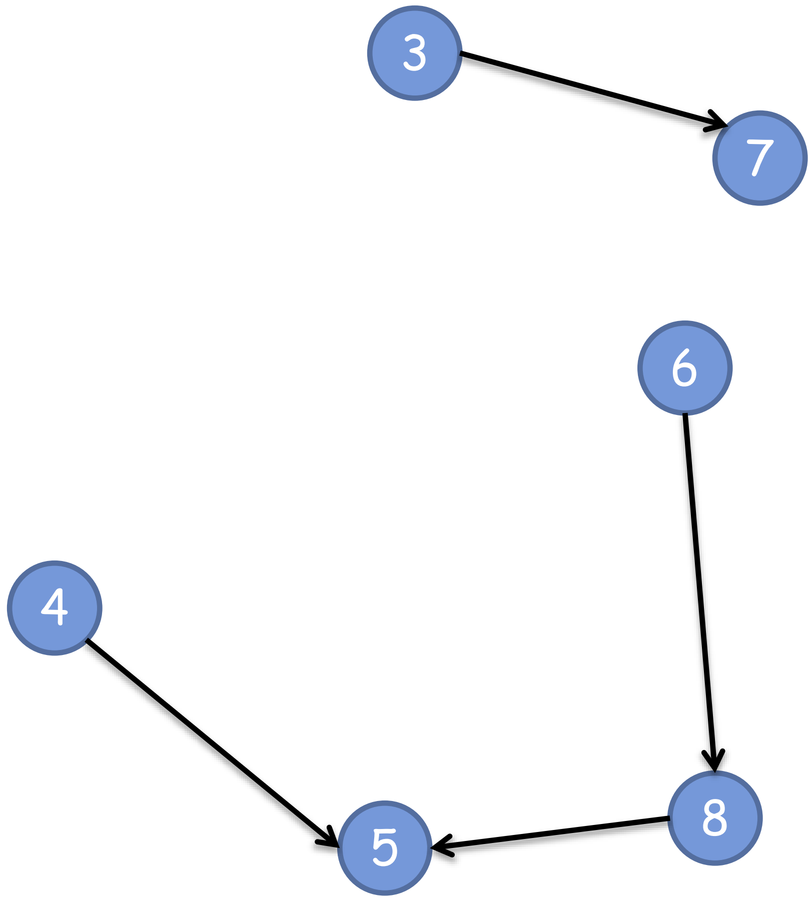

Delete Parallel Edges

Delete Self Loops

K = 1

Delete Dummy

Find a Flower



# Example of Applying Reductions

K = 2. FVS: 1 2

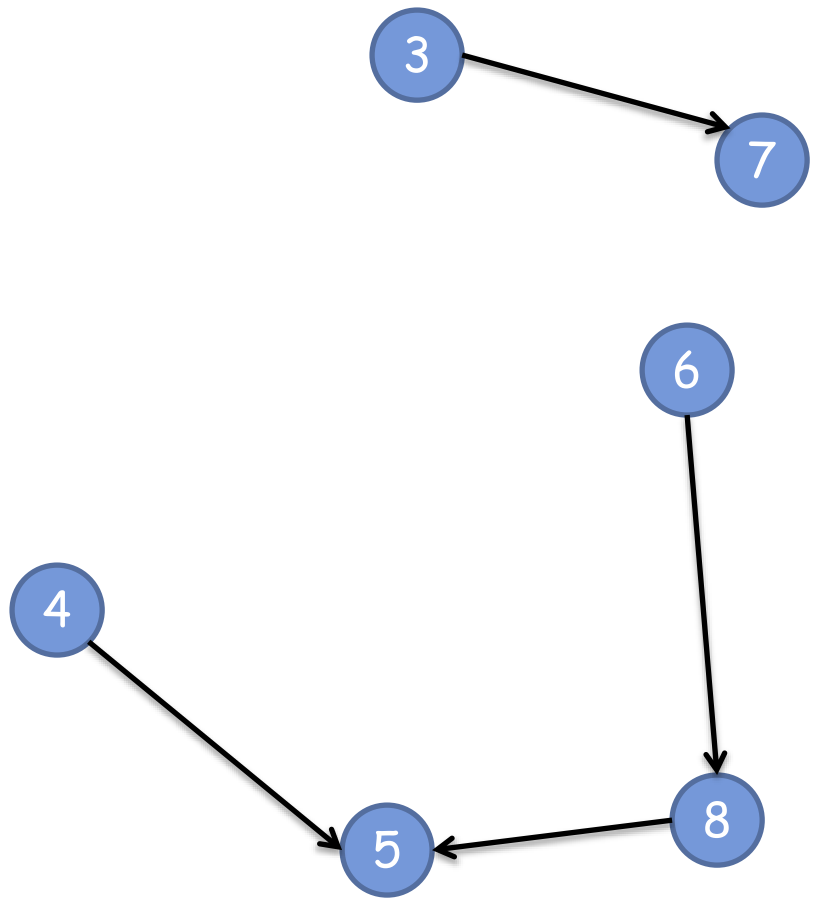
Delete Parallel Edges

Delete Self Loops

K = 1

Delete Dummy

Find a Flower



# Example of Applying Reductions

K = 2. FVS: 1 2

Delete Parallel Edges

Delete Self Loops

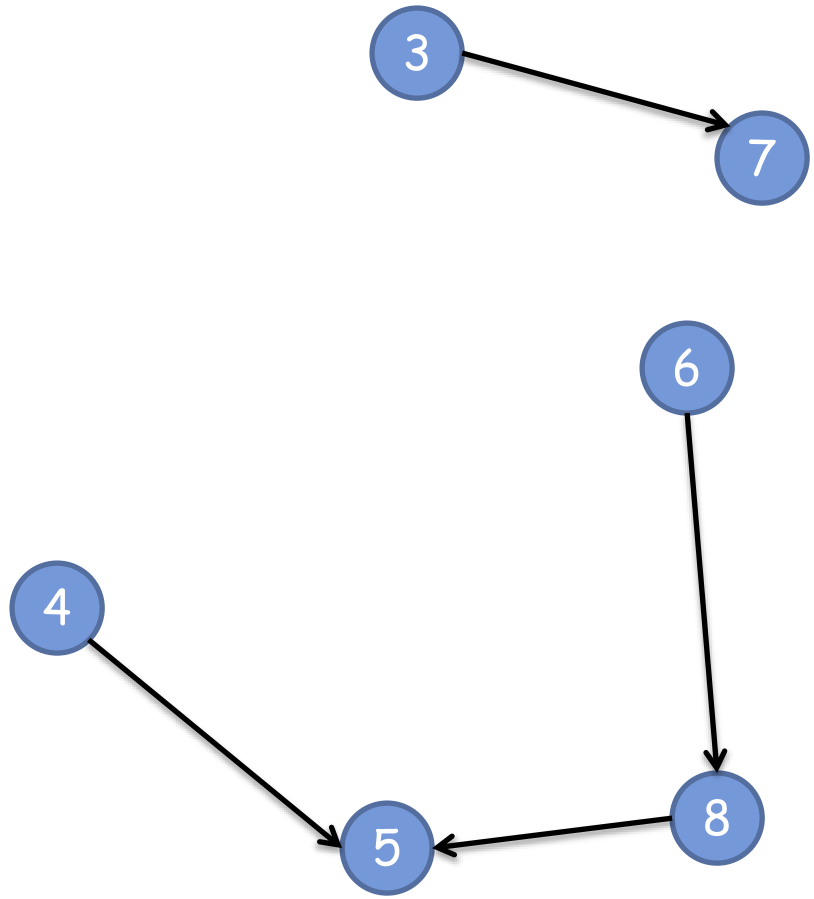
K = 1

Delete Dummy

Find a Flower

K = 0 and Acyclic

Done



# Generation Strategy

- Generation strategy
  - The edges of connected DAG are $\frac{1}{4}$ of the total edge bound
  - Each cycle has at most $\frac{1}{4}$ of nodes
  - Generate cycles until reaching the edge bound