Computing A Near-Maximum Independent Set In Dynamic Graphs

Weiguo Zheng[1,2],   Chengzhi Piao[1],   Hong Cheng[1],   Jeffrey Xu Yu[1]

[1]The Chinese University of Hong Kong, HK SAR, China

[2]Fudan University, China

ICDE 2019, Macau

Outline

- Background
  - Graph
  - Independent Set
  - Dynamic Graph
  - Problem Definition
  - Evaluation

- Existing Algorithms
  - Reduction Rule
  - Greedy Algorithm

- Dependency Graph based Framework
  - Brute-Force Search
  - Dependency Graph
    - Construction
    - Dealing with Updates
    - Maintenance
    - DGOracle
  - Batch Update

- Experiment

Background

Graph

A wide variety of real-world data is represented as graph structure.

- drug molecule graphs
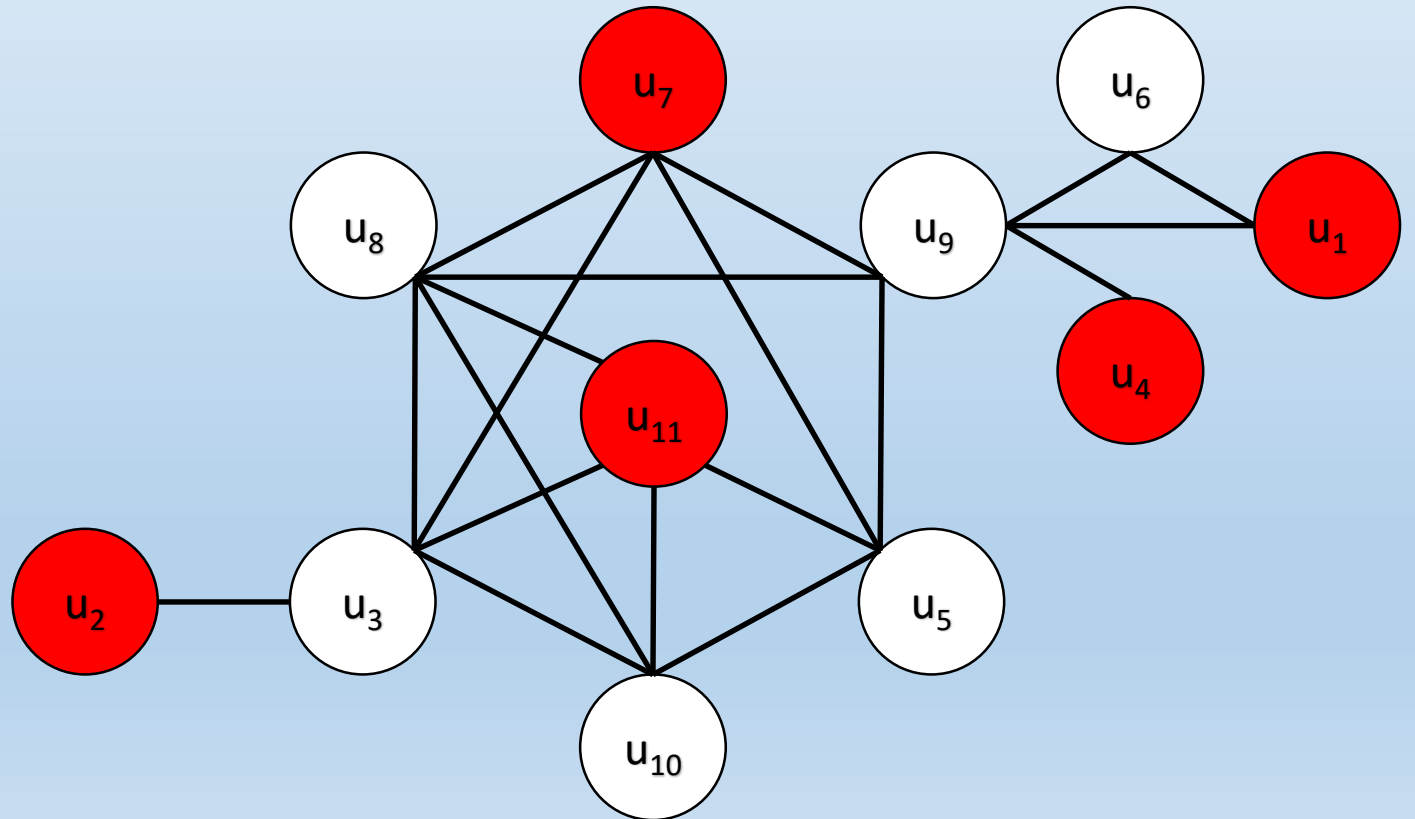- road networks
- knowledge graphs
- …
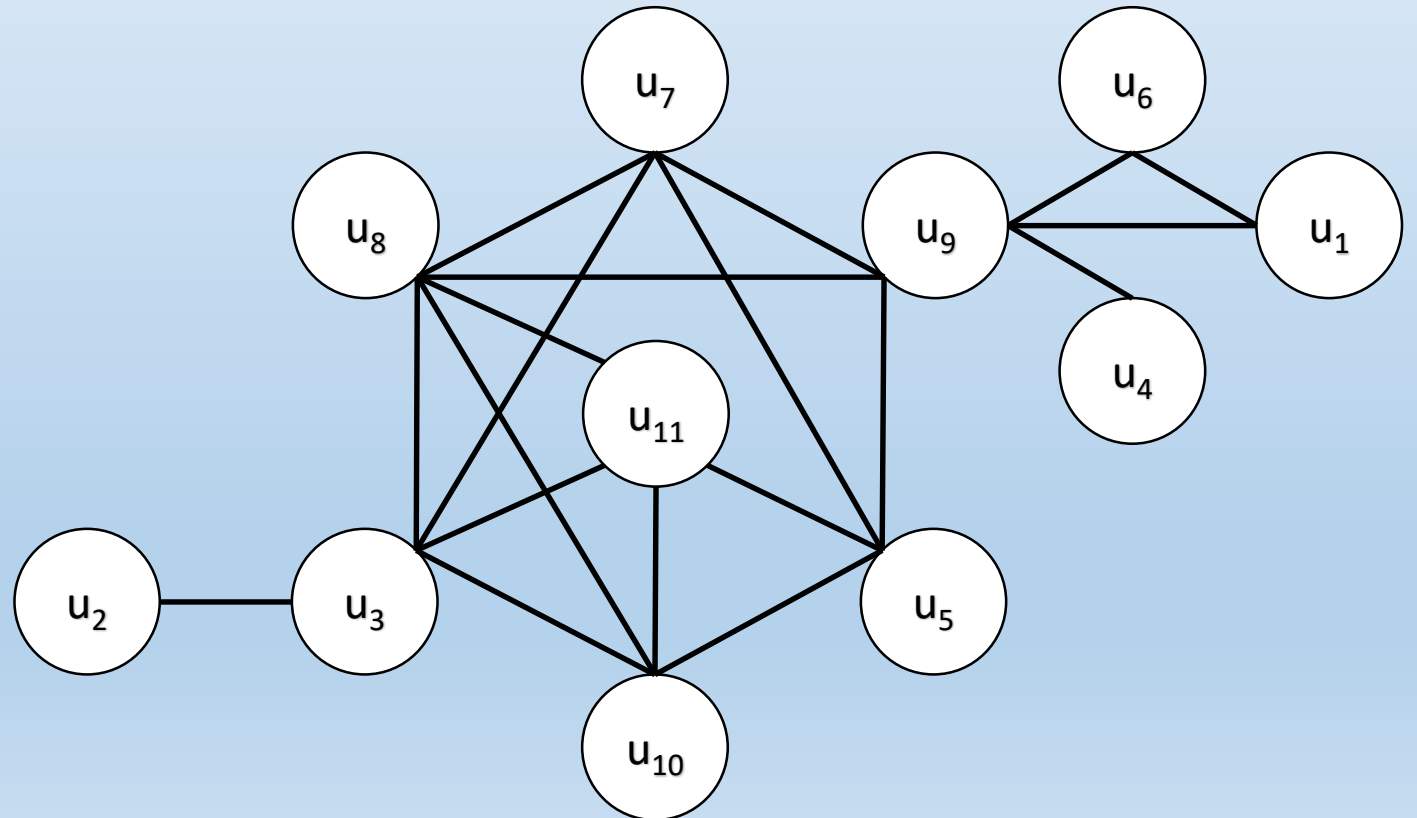
Graph

Graph problems have attracted much attention these years.

- Shortest path query
- Reachability computation
- Subgraph matching
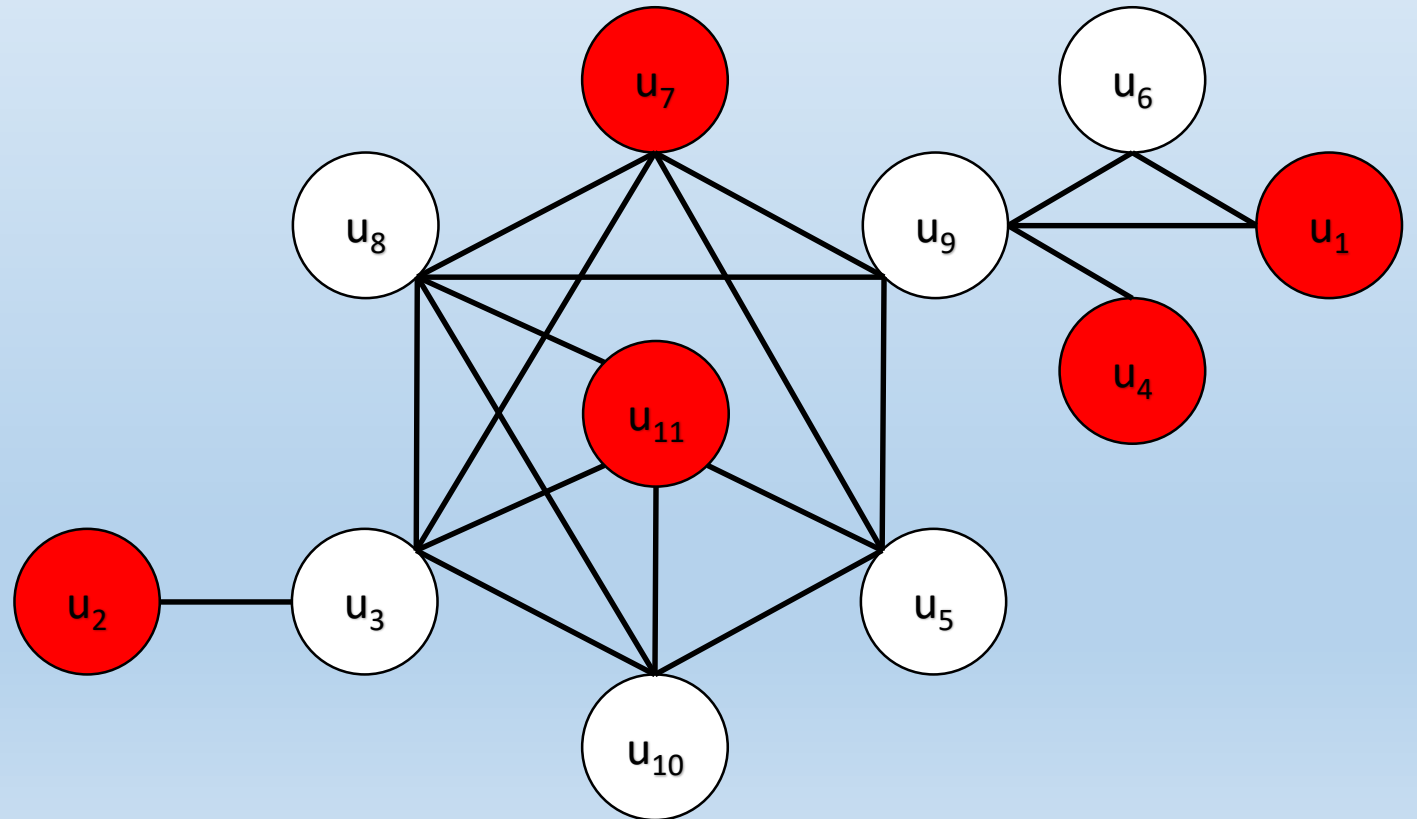- Maximum clique
- …

Independent Set

Given a set of vertices in graph G, we call it an independent set of G if and only if there are **no neighborhoods** among them.
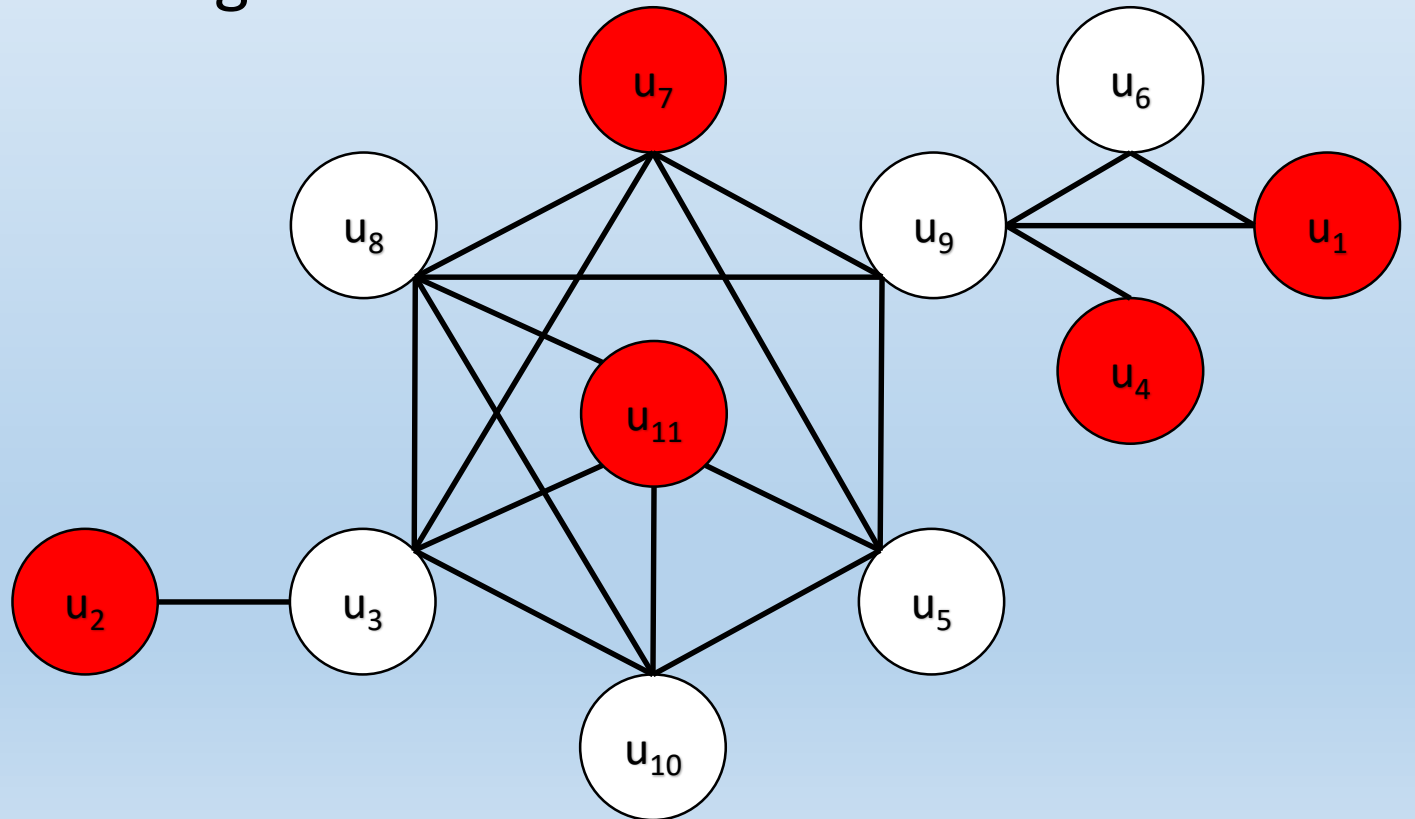
Independent Set

Maximal independent set

- Can not add more vertices into it.

Independent Set

Maximum independent set(MIS)

- A MIS has the largest size among all independent sets.
- Maybe not unique.

Independent Set

Finding a MIS is a fundamental NP-hard problem and has many applications.

- association rule mining
- automated map labeling
- road network routing

Independent Set

However, it is impossible(too time-consuming) to mine a MIS in large networks. Efficiently mining a near-maximum one is also quite reasonable.

Near-Maximum Independent Set(Near-MIS)

- |Near-MIS| is close to |MIS|
- the larger the better

Dynamic Graph

In real-world, the graph structure is changing all the time.

For example, in a social network:

- register/cancel an account   ->   add/delete a node
- like/dislike another user   ->   add/delete an edge

Online replying to the frequent changing operations needs much efficiency.

Problem Definition

Target: maintain a Near-MIS in dynamic graphs

Input

- An undirected graph G_0 and the initial Near-MIS$_0$
- A sequence of updates on the graph: { update$_i$ }

Output

- online maintain the Near-MIS after each update: { Near-MIS$_i$ }

Evaluation

$(G_i, \text{Near-MIS}_i) \rightarrow \text{update}_i \rightarrow (G_{i+1}, \text{Near-MIS}_{i+1})$

Efficiency

- initializing time cost(e.g. building an index)
- time cost on dealing with each update

Evaluation

$(G_i, \text{Near-MIS}_i) \rightarrow \text{update}_i \rightarrow (G_{i+1}, \text{Near-MIS}_{i+1})$
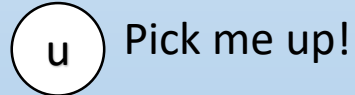
Effectiveness

- $\text{Gap}_i = |\text{MIS}_i| - |\text{Near-MIS}_i|$
  - $\text{Accuracy}_i = |\text{Near-MIS}_i| / |\text{MIS}_i| <= 1$
  - Gap_0 depends on the input(G_0 and Near-MIS_0)
- Increasing Rate of Gap
  - $IR(i) = (\text{Gap}_i - \text{Gap}_0) / i$
  - The smaller the better

Existing Algorithms

Reduction Rule

Degree-Zero reduction: degree(u) = 0, no neighborhood

- The node u must be in the MIS.
- Directly put u into the solution and delete it from the graph.

(u) Pick me up!

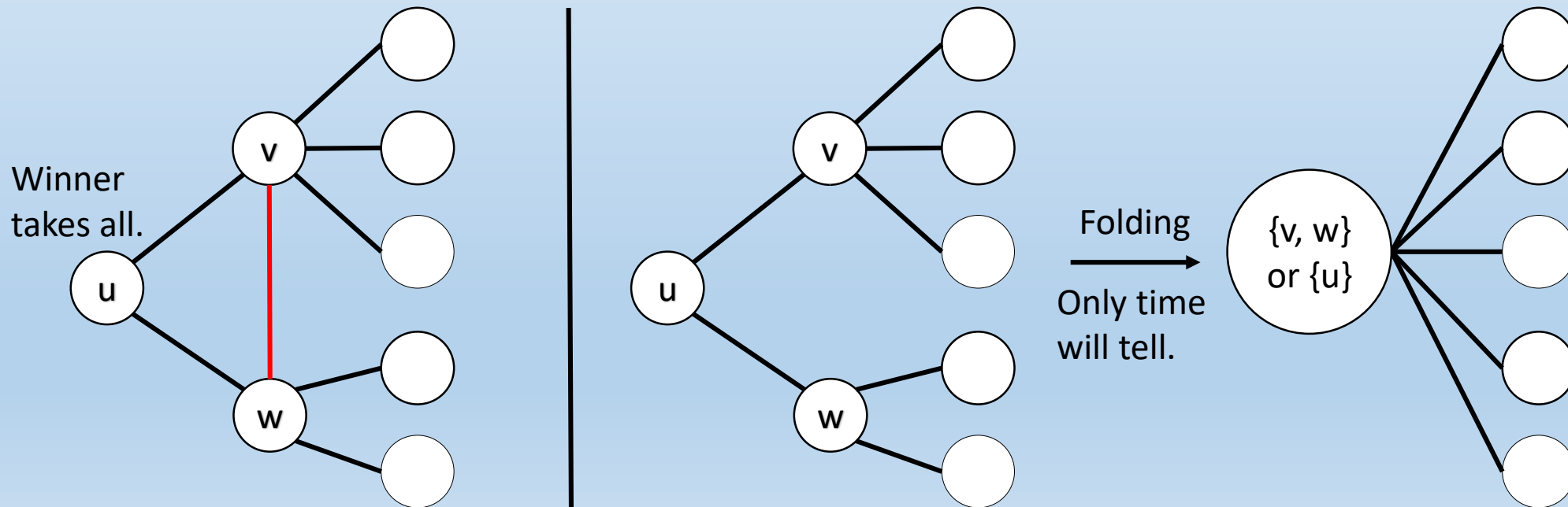# Reduction Rule

Degree-One reduction: degree(u) = 1, (u, v)

- There must be a MIS containing u.

- Of course there may exist a MIS containing v but not u. However, u is obviously a better choice as it does not affect other nodes.

- So we can add u into the solution, and then delete u and v from the graph.

u or v?
That is not a question.
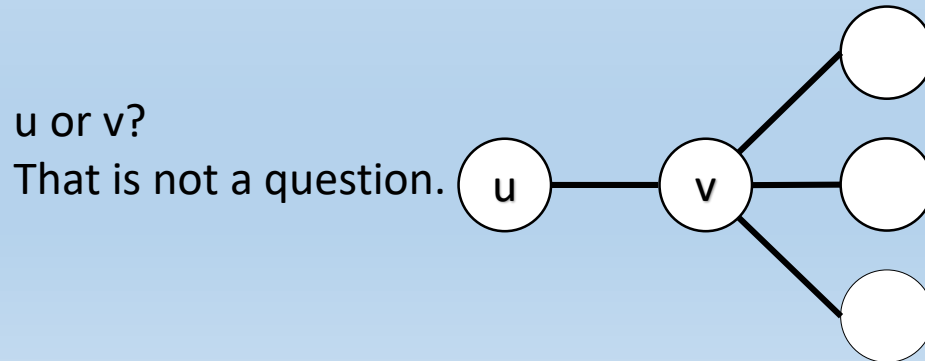
Reduction Rule

Degree two reduction: degree(u) = 2, (u, v) and (u, w)

- Isolation: (v, w) exists -> choose u and delete {u, v, w}

- Folding: (v, w) does not exist -> merge {u, v, w}

Greedy Algorithm

Reducing-Peeling Framework
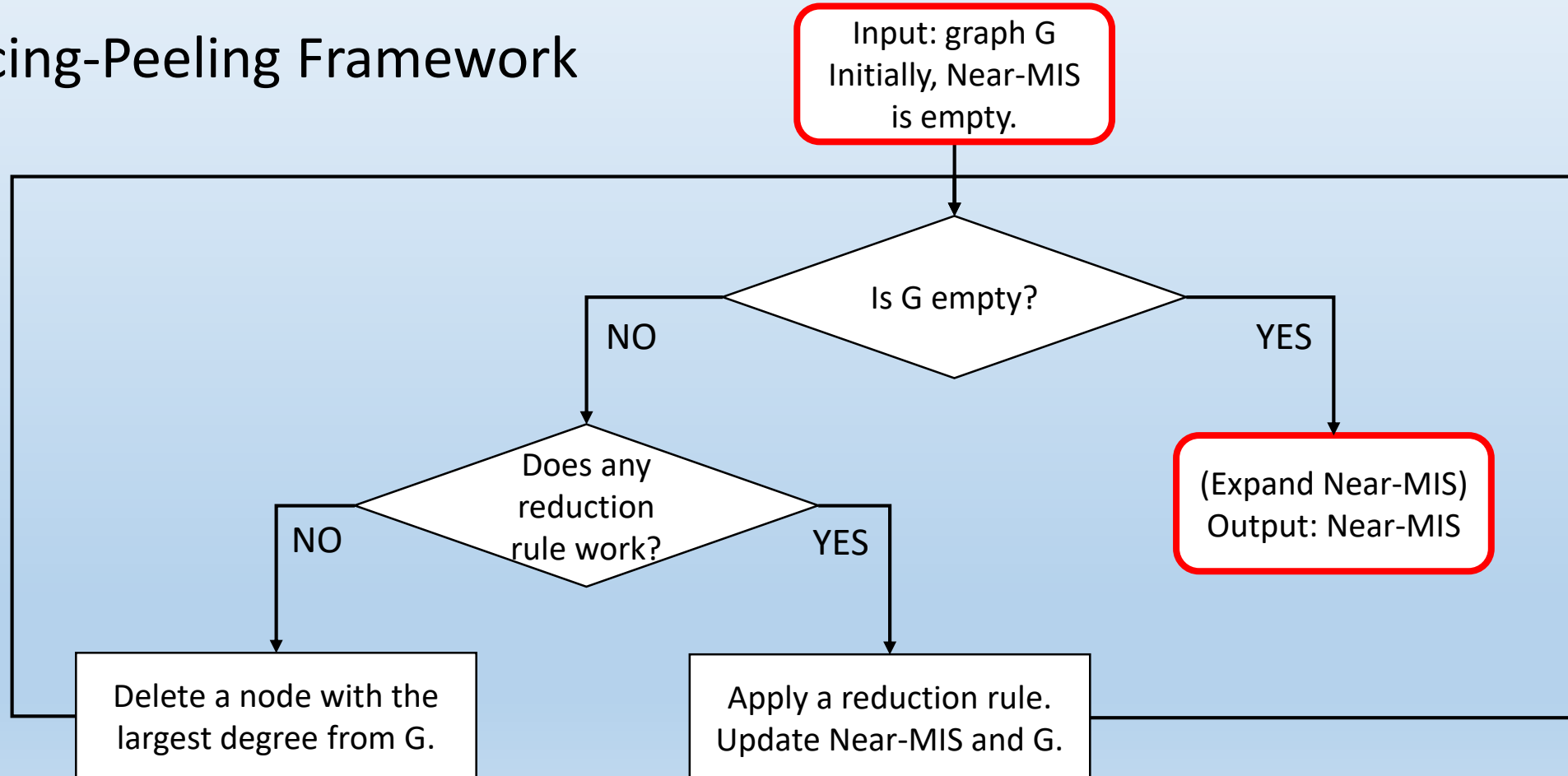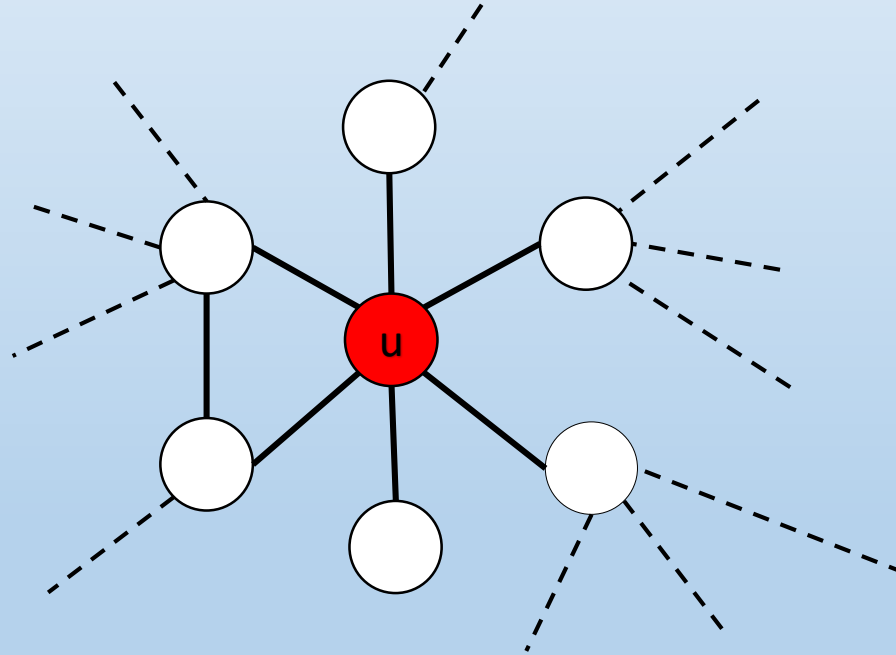
Greedy Algorithm

Reducing-Peeling Framework

Delete a node with the largest degree from G.

- This step does an inexact reduction.

- Greedy Idea: The largest degree node is unlikely to be in a MIS.

- Gap = |MIS| - |Near-MIS| <= times of inexact reductions

Dependency Graph based Framework

A Fundamental Operation

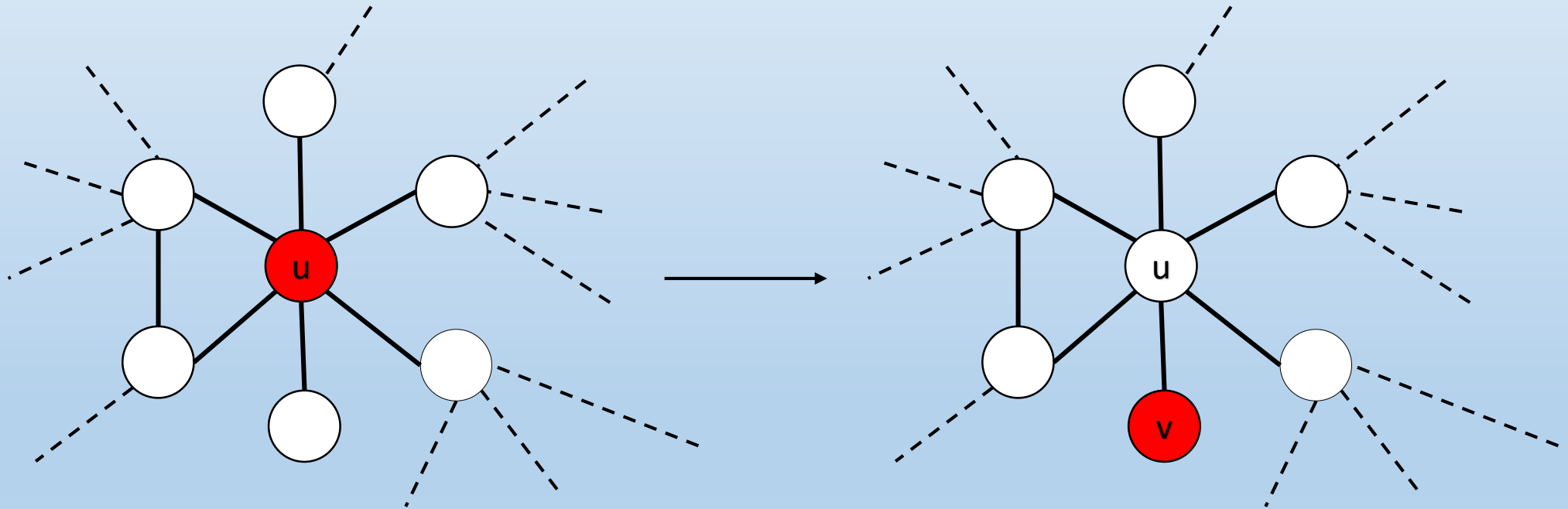Given a MIS of graph G, what if we want to move out a node u from this MIS.

Move Out u from the MIS
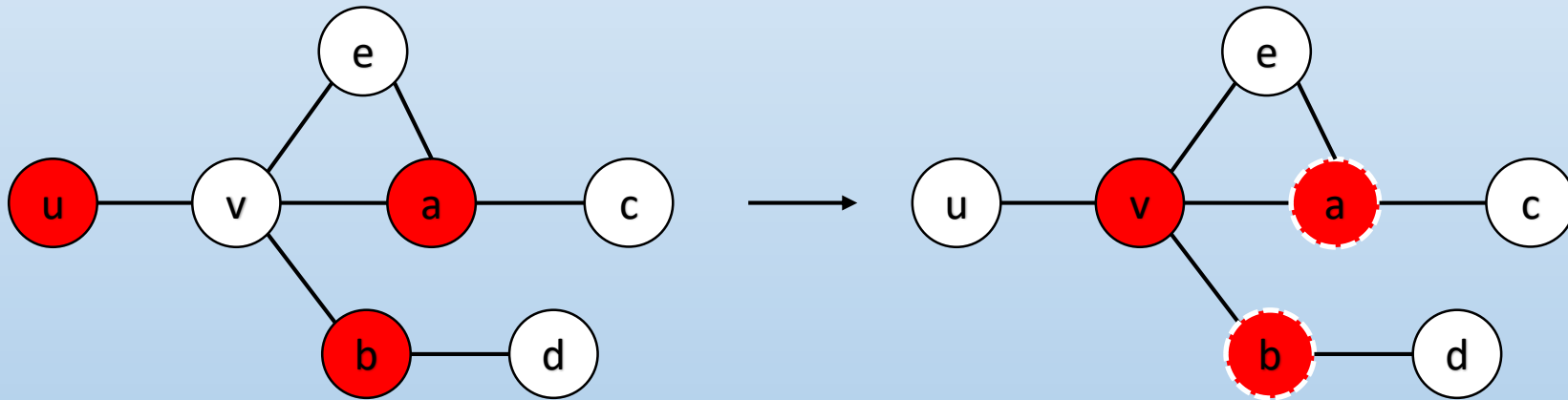
Do nothing.

- Size decreases 1.
- The MIS becomes a Near-MIS.
- Things can not be worse.

Move Out u from the MIS

To maintain a MIS, we must choose a neighbor to replace u.
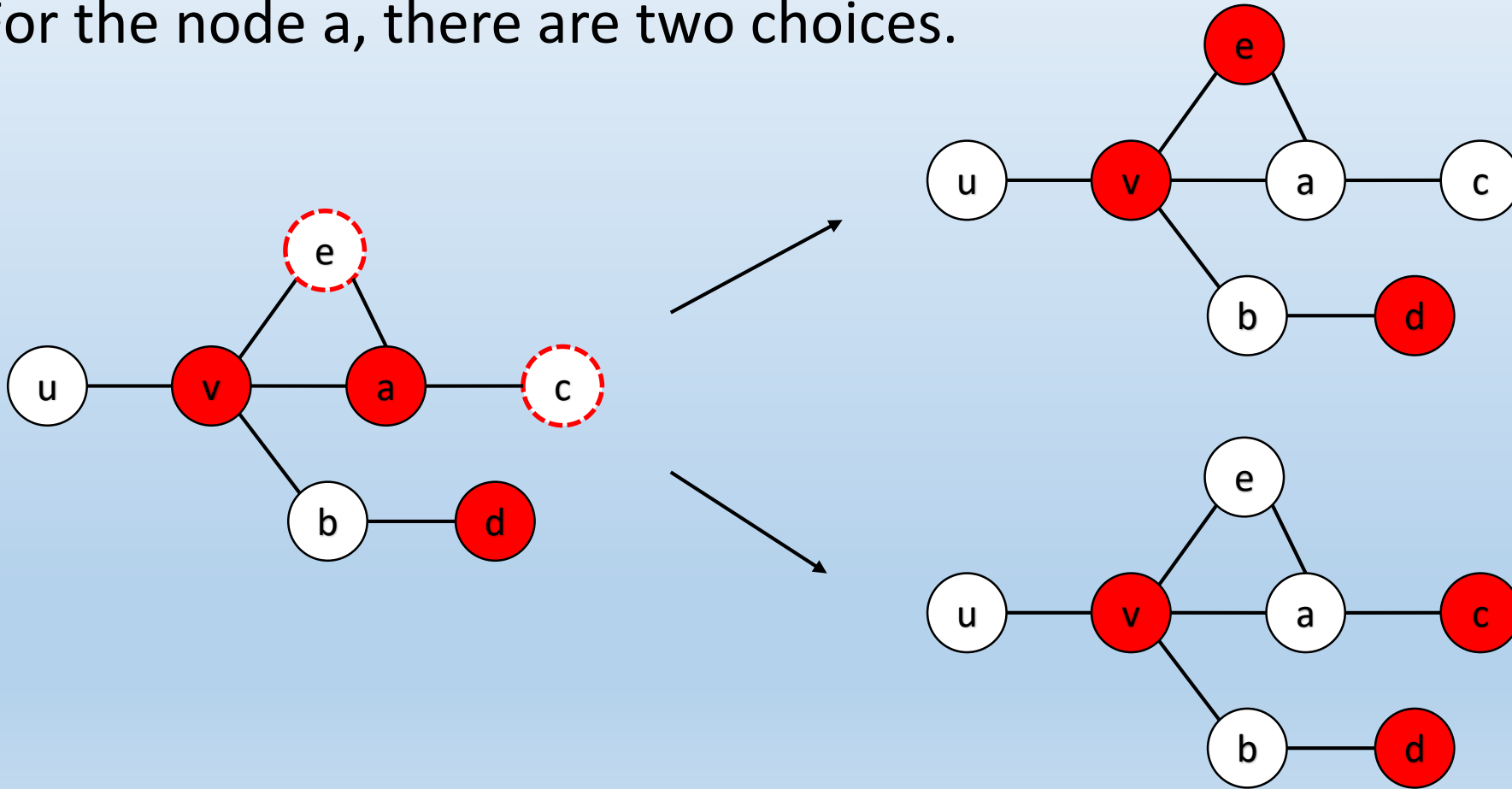
Move Out u from the MIS

However, we usually need to do this recursively.

Recursively move both a and b out.
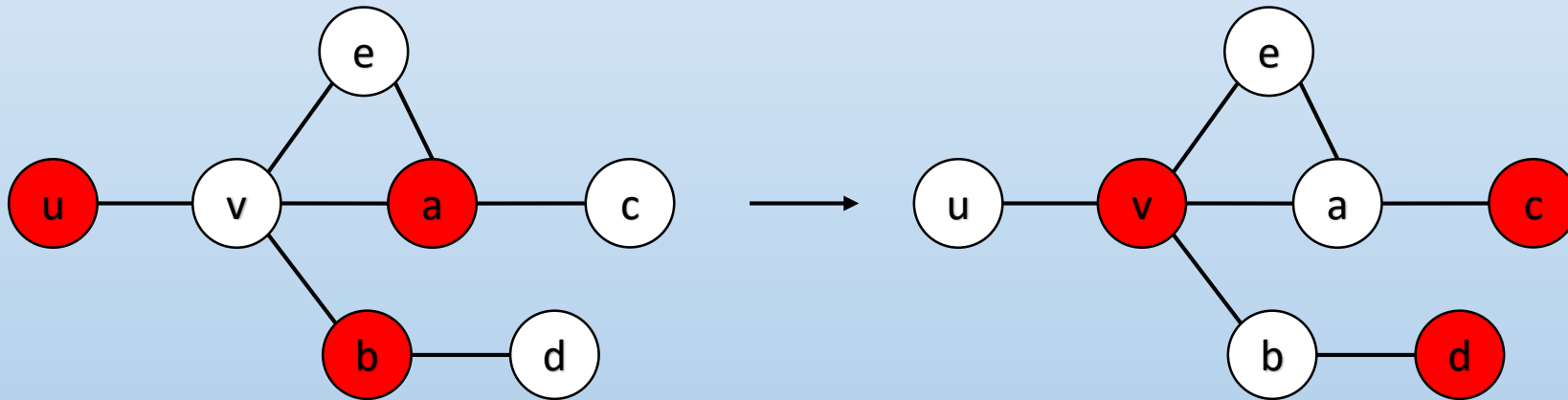
Move Out u from the MIS

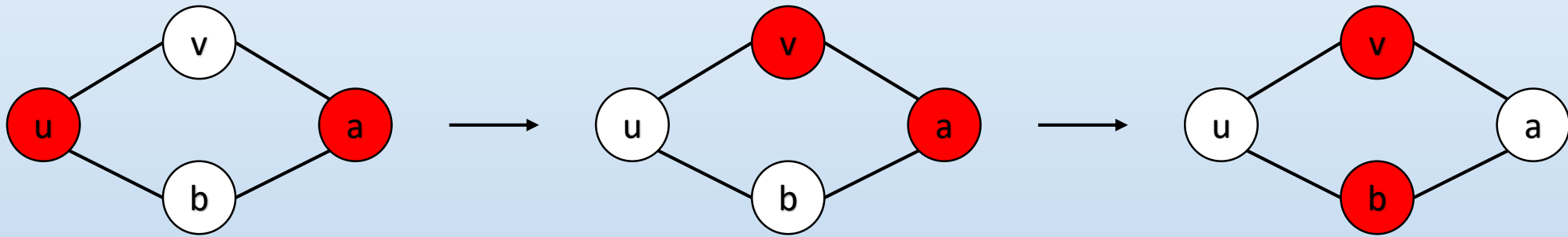For the node a, there are two choices.

Valid Swap

Move out: {u, a, b}

Move in: {v, c, d}
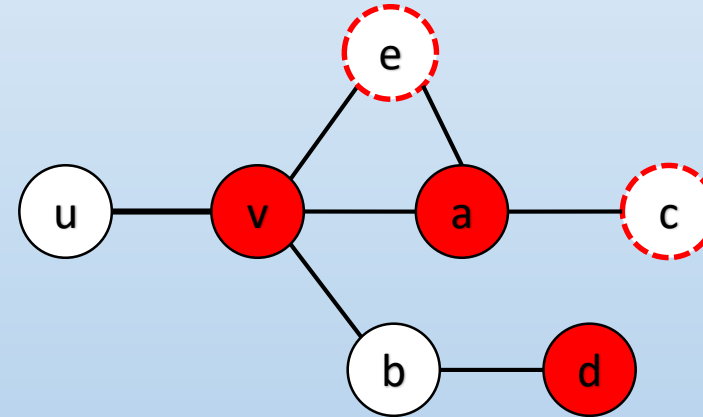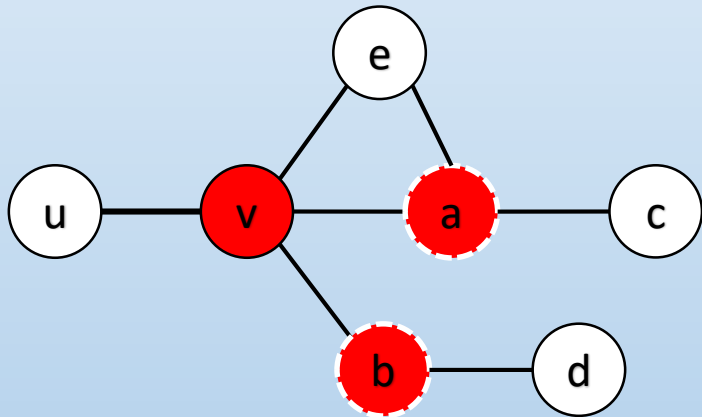

Happy ending!

Another Example

Although we just choose a neighbor to replace u, there may be more than one neighbors moved in finally.

Brute-Force Search
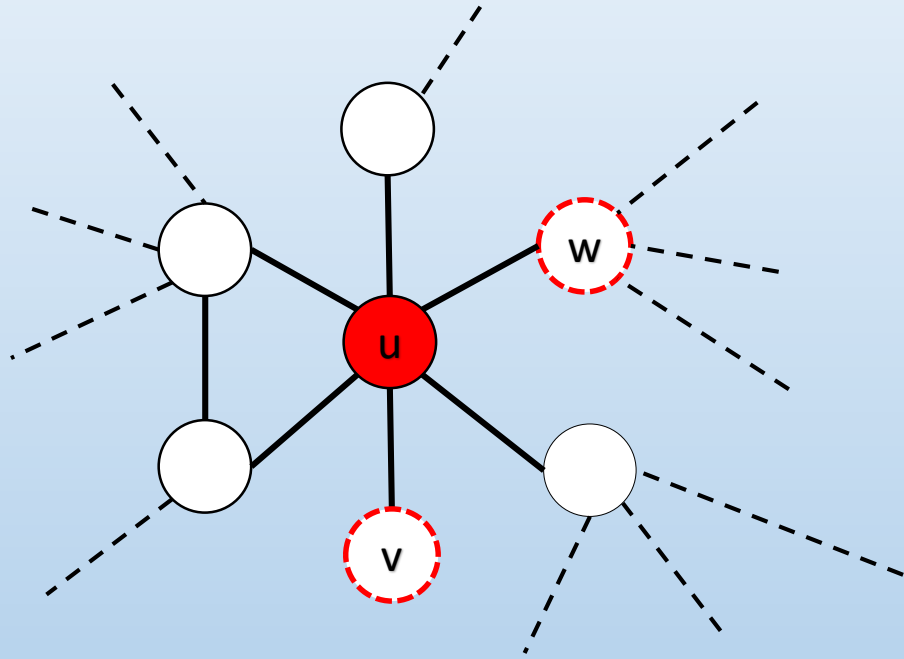
Move in a white node.

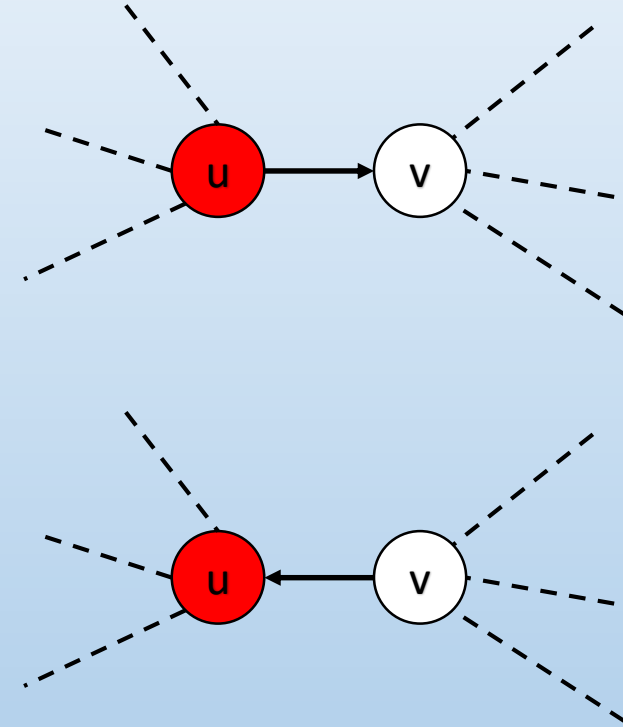-> Move out all its red neighbors.

Move out a red node.

-> Move in one of its white neighbors.

Two Ways of Pruning

To move out a red node,
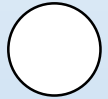search at most two given neighbors.

Only search in one direction.
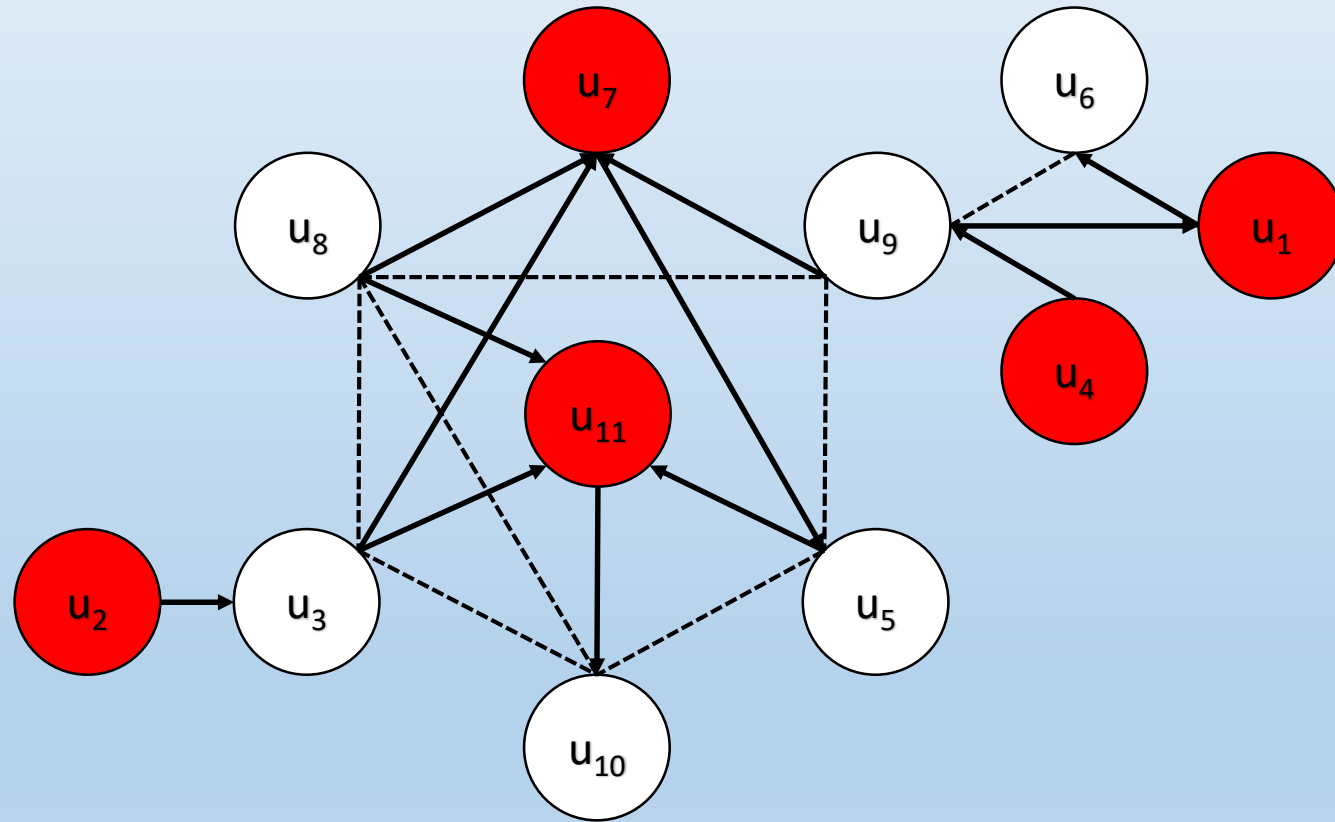
Dependency Graph

● We call the nodes in the independent set **reducing nodes**.

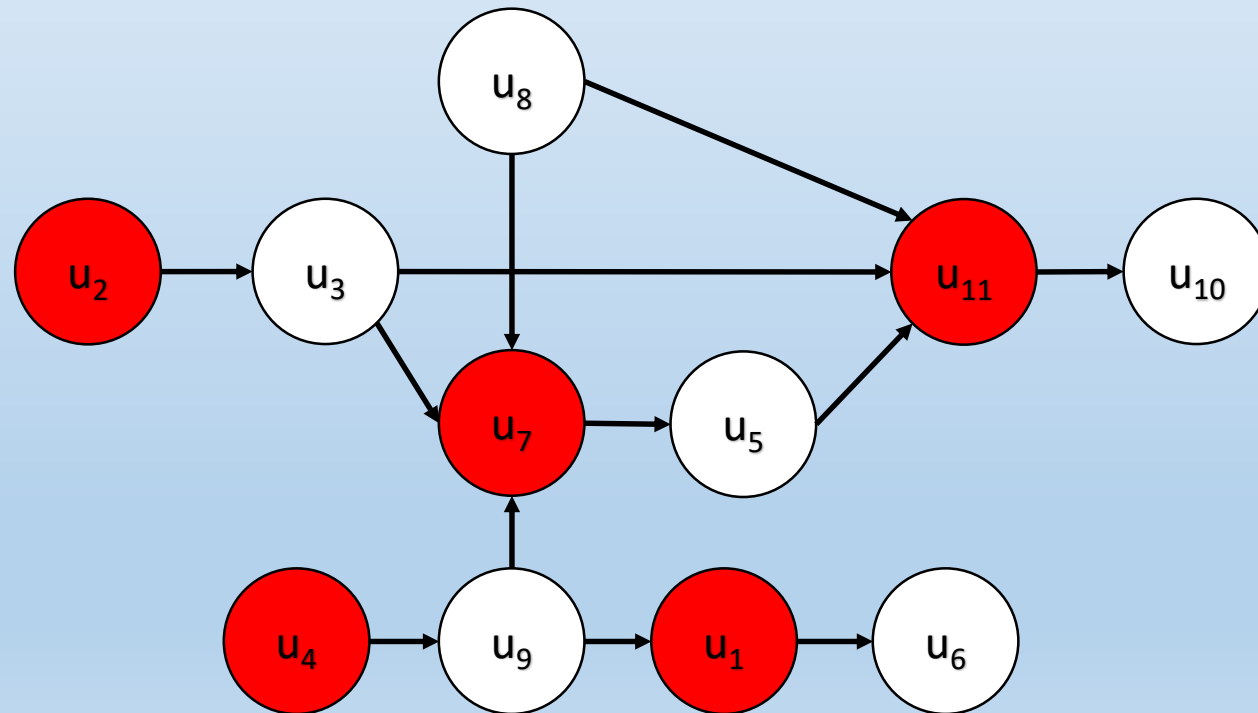○ Others are called **dependent nodes**.

- Remain all edges between reducing and dependent nodes.
- Each edge gets directed.
- Each reducing node has at most two out-neighbors.
- Each dependent node has at most one in-neighbor.

Dependency Graph

Dependency Graph

- The direction implies that whether a node can be moved in/out depends on its out-neighbor(s).

Dependency Graph

Efficiency

- Searching in such a dependency graph is especially efficient.
- In our paper, we prove that the average time complexity is merely O(d).

Effectiveness

- However, dramatically reducing searching space may damage the accuracy.
- It is vital to construct a good dependency graph.
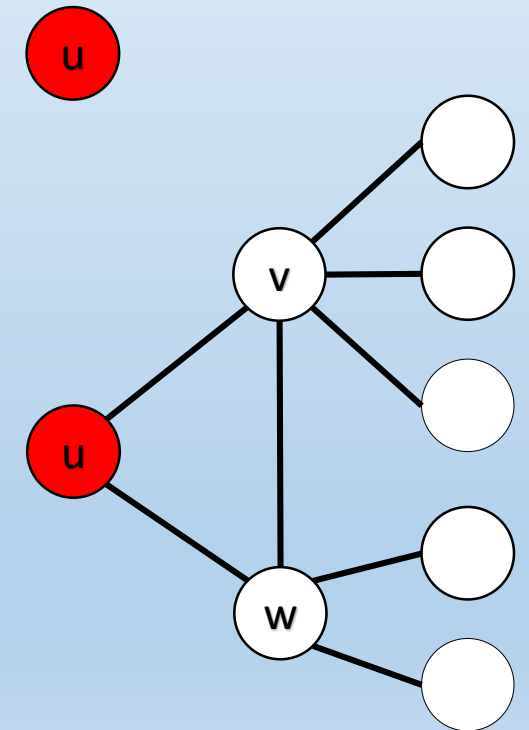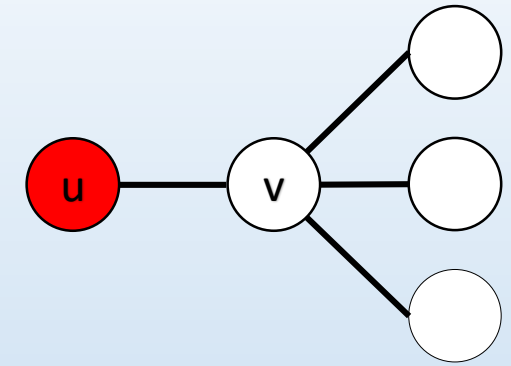
DG: Construction

Let us go back to the reducing-peeling framework.

These simple reduction rules are only applied on a node whose degree is 0, 1, or 2.

So, the output Near-MIS does not contain any high degree node?

Of course, the answer is YES!

Because other neighbors have already been deleted.
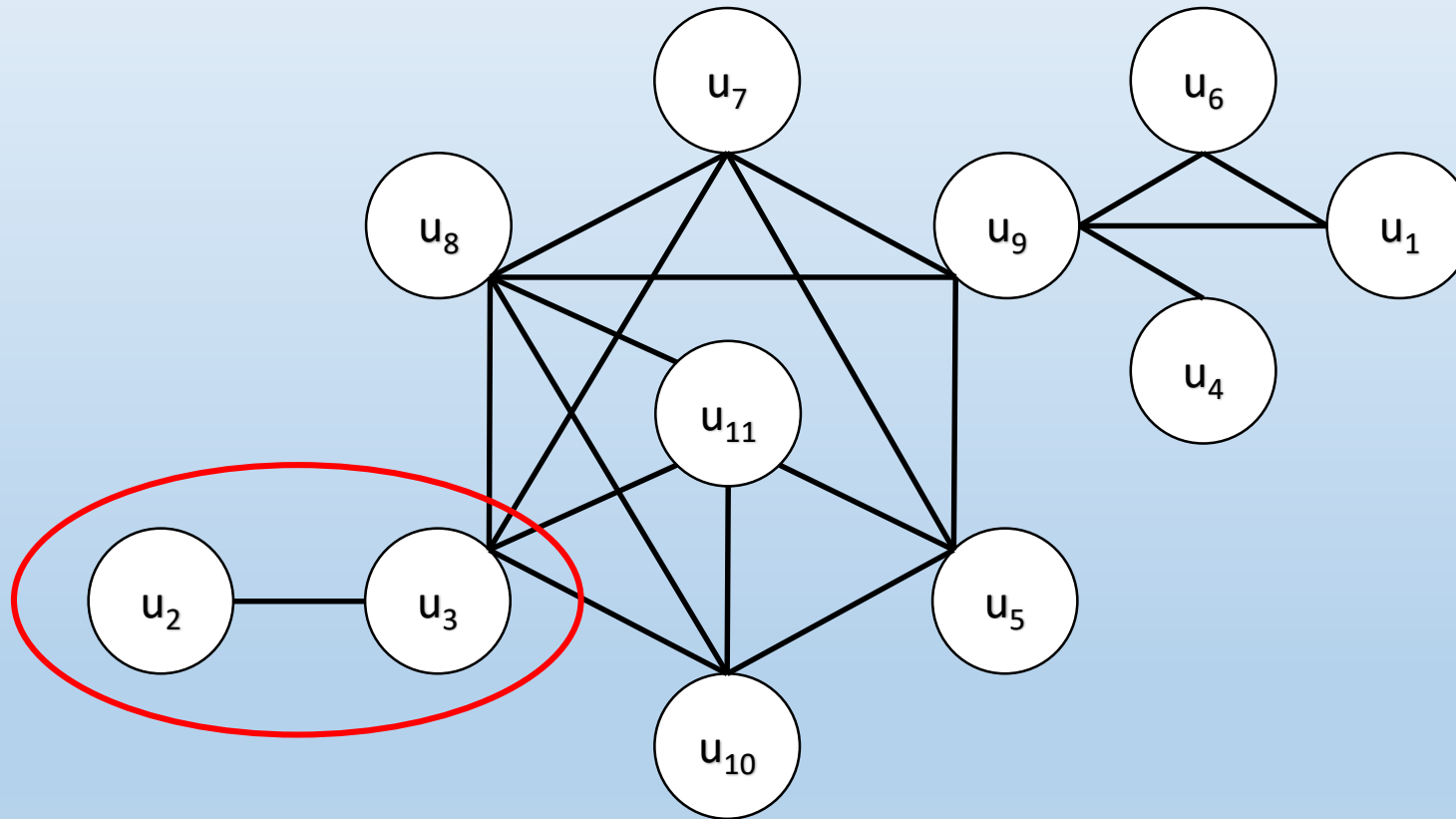
DG: Construction

The reducing-peeling framework does not merely outputs a Near-MIS, but also implies a topological order of the graph.

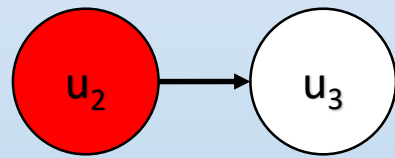Based on this point, we can construct a good dependency graph.

- For each reducing node, its out-neighbors are decided when a reduction rule works.

- Then, the others become its in-neighbors.
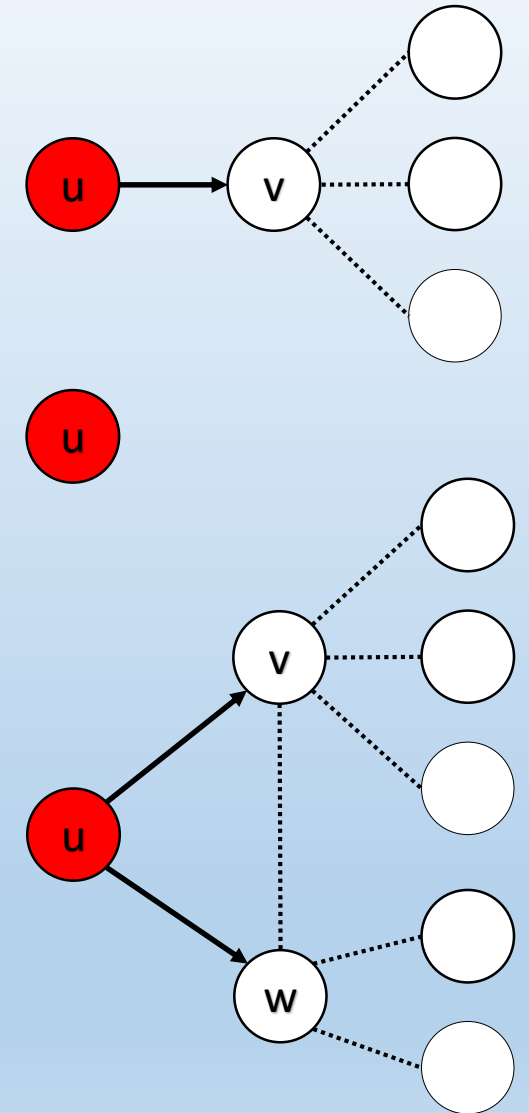
Dependency Graph

Dependency Graph

Dependency Graph

Dependency Graph

Dependency Graph
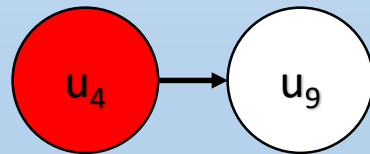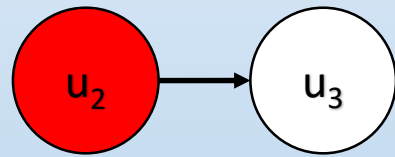
Dependency Graph
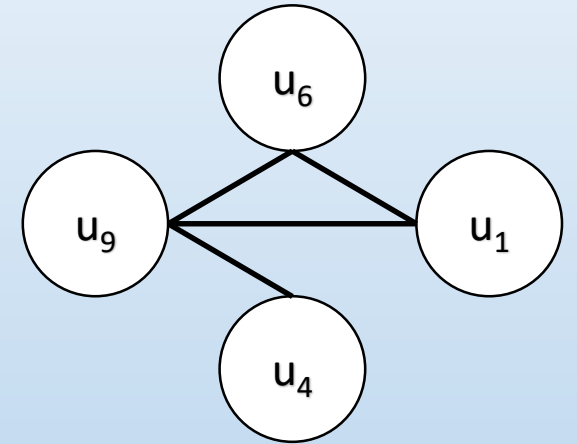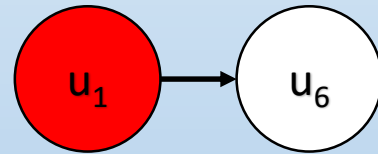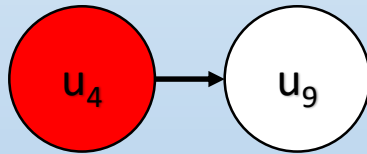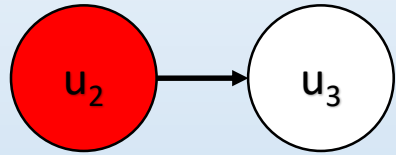
Dependency Graph
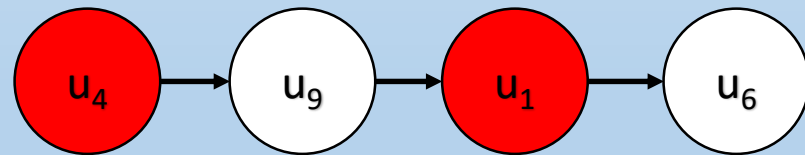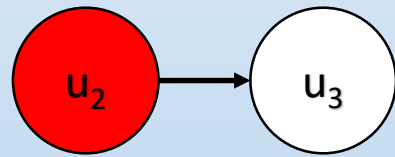
Dependency Graph
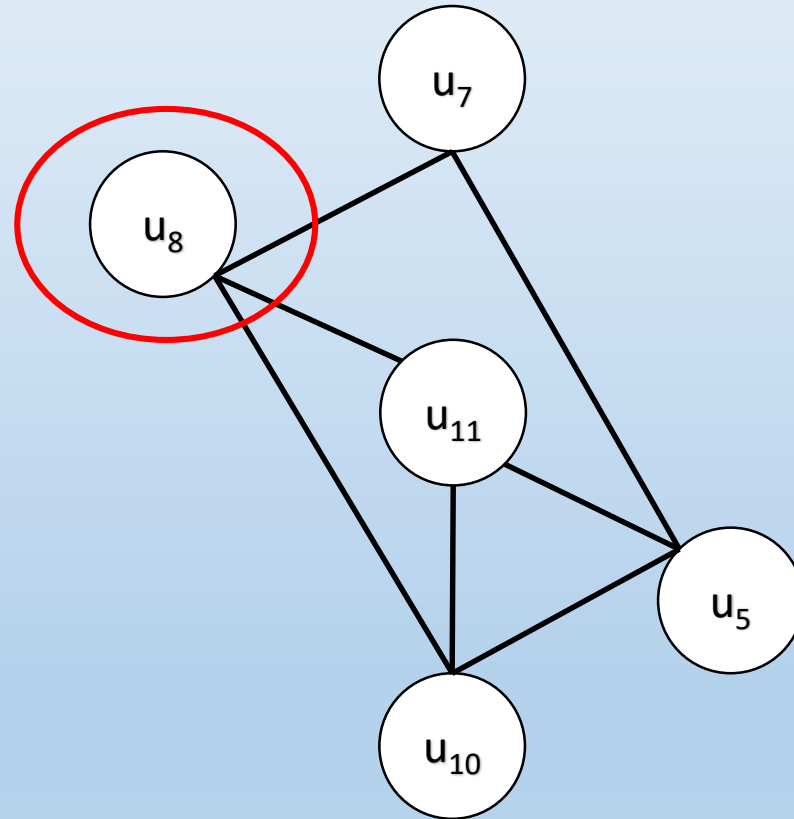
Dependency Graph
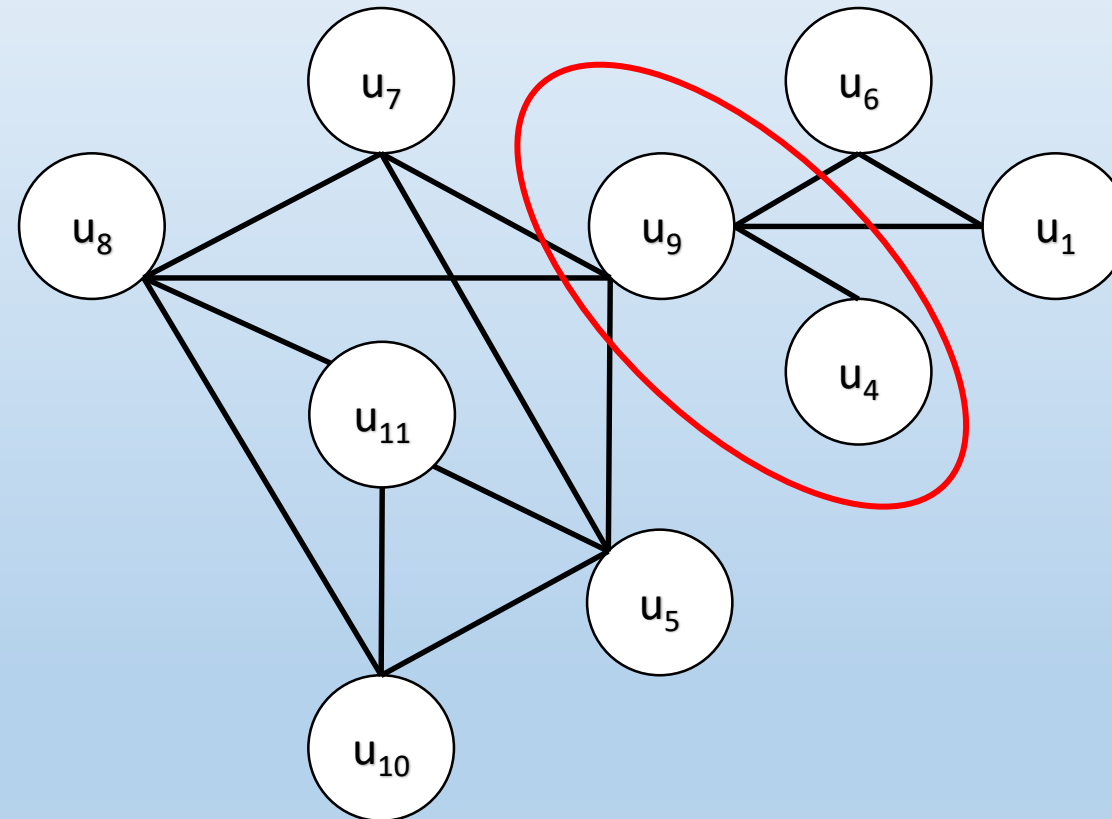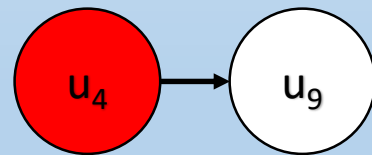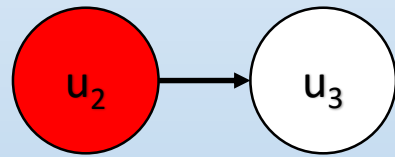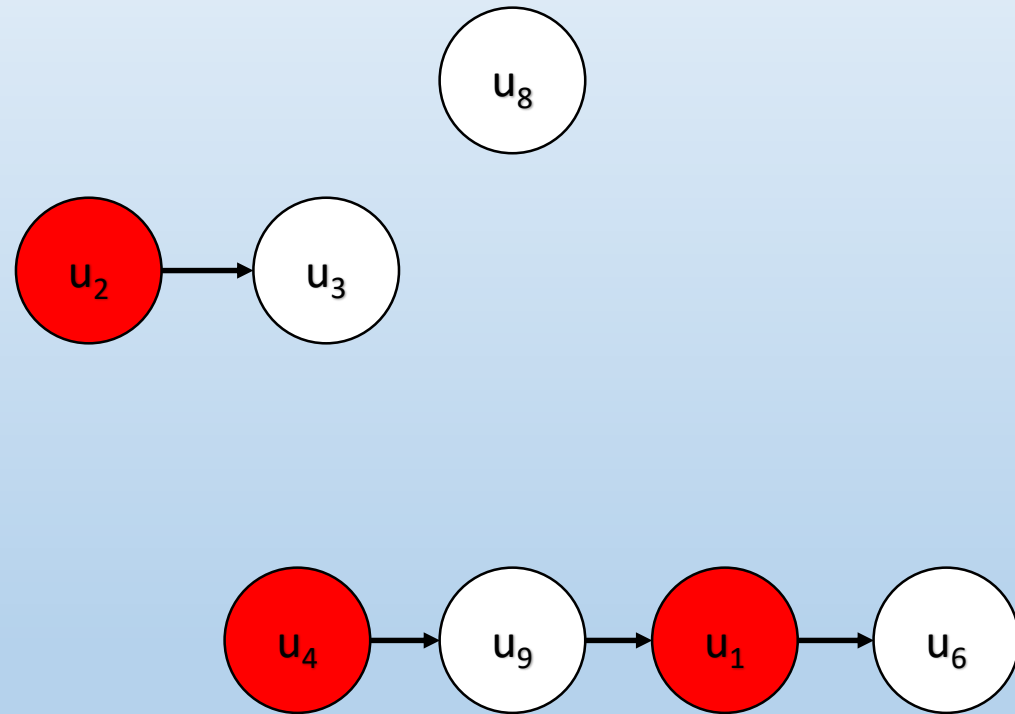
Dependency Graph
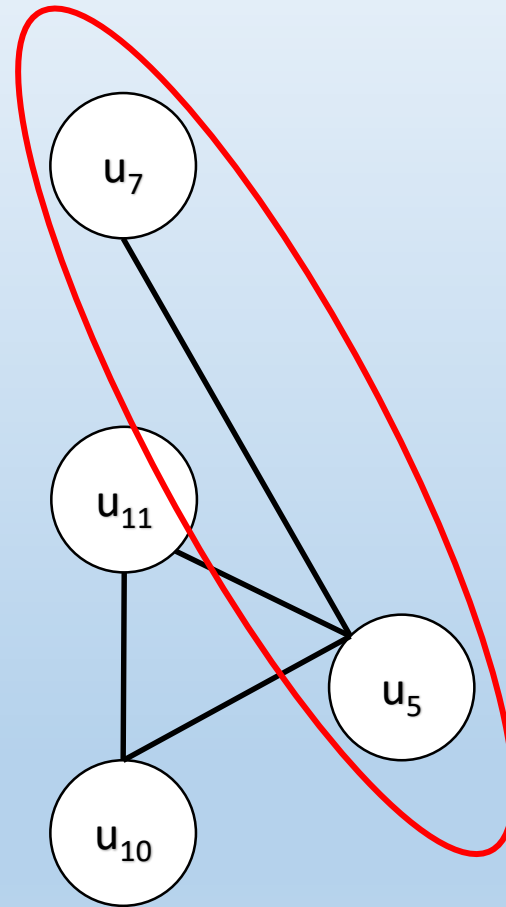
Dependency Graph
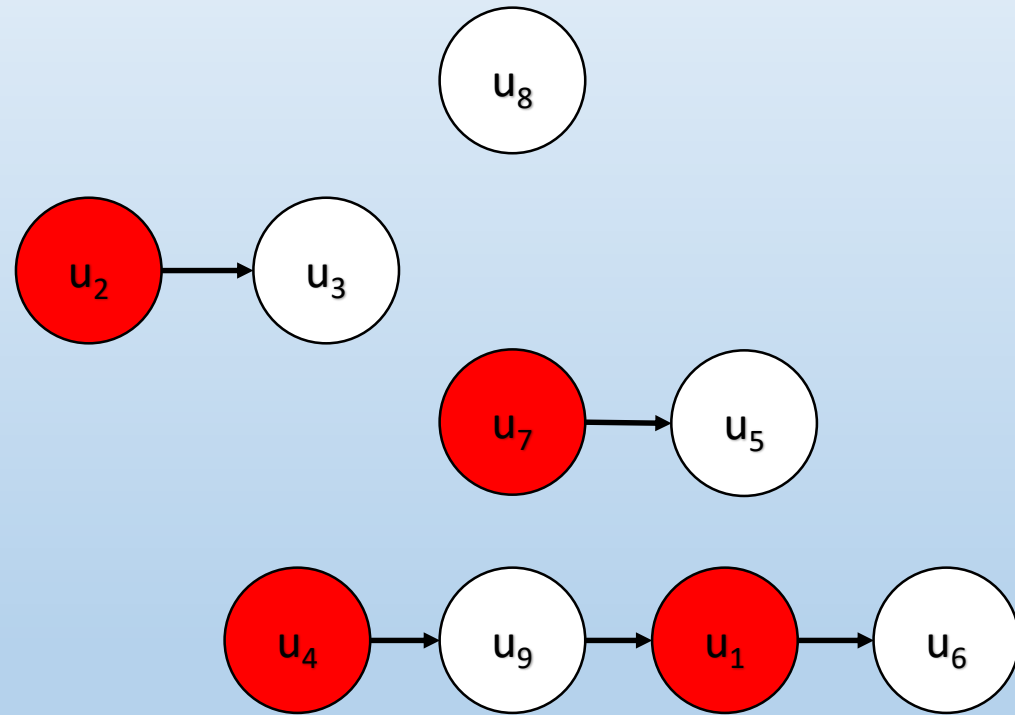
Dependency Graph
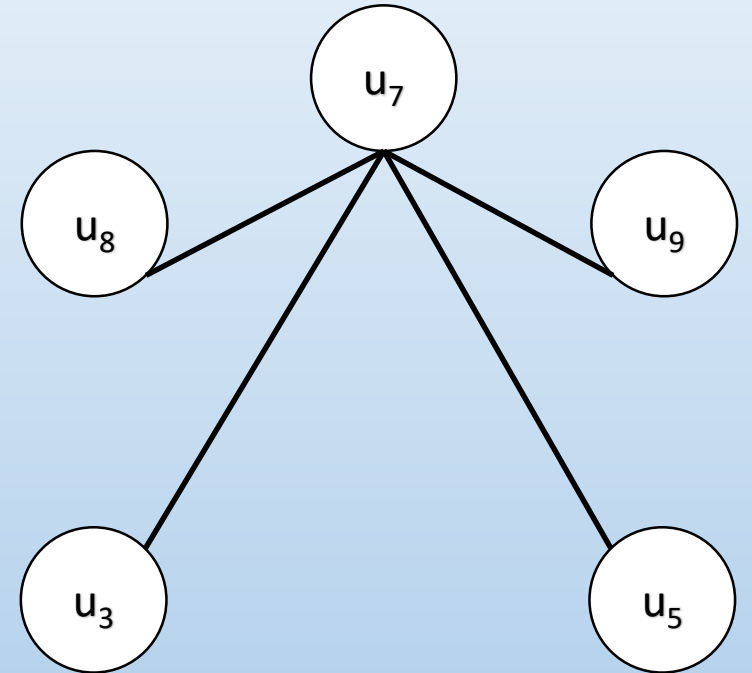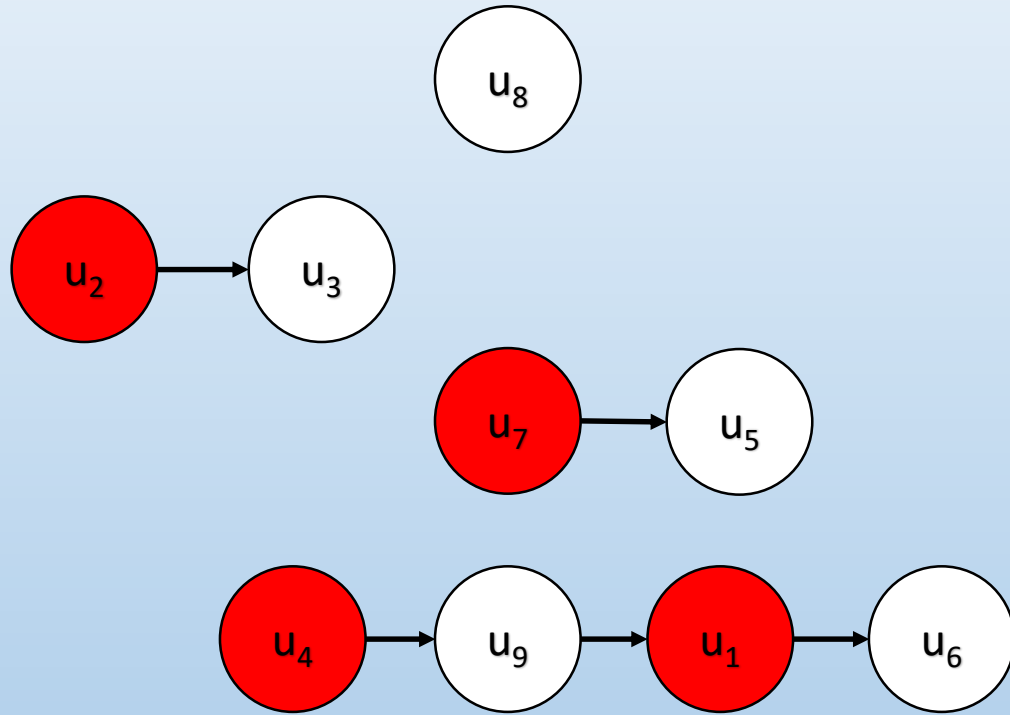
Dependency Graph
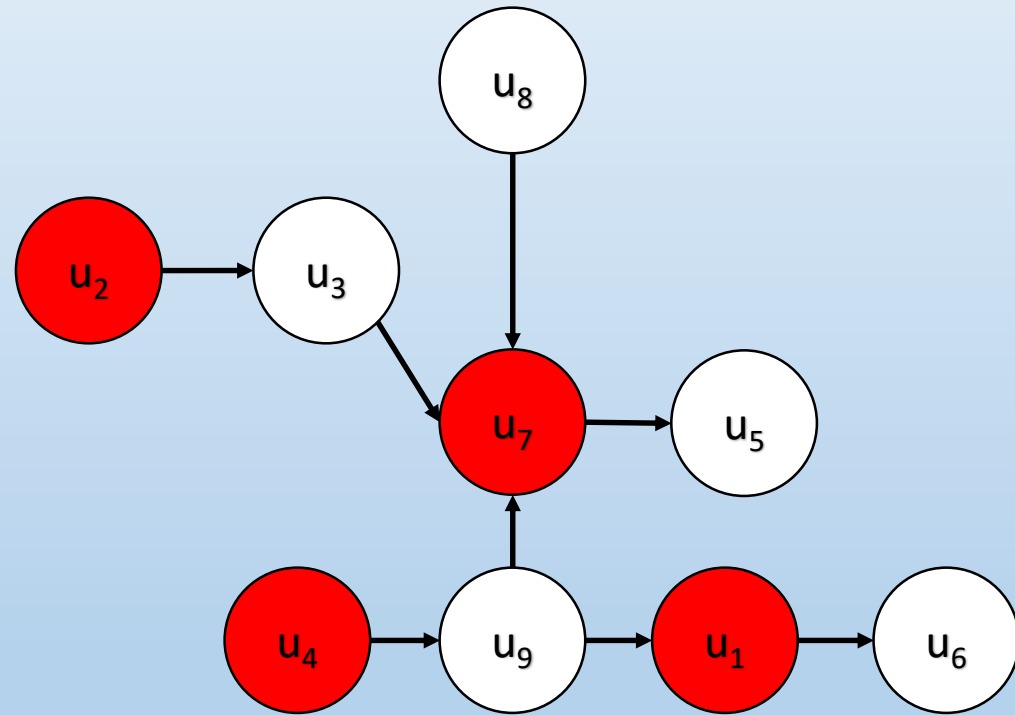
Dependency Graph
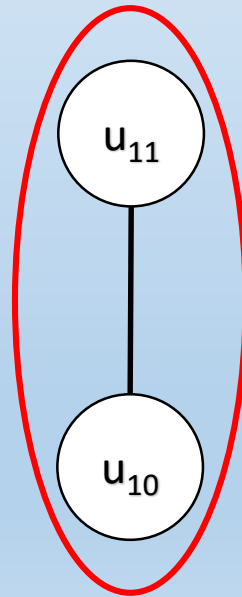
Dependency Graph
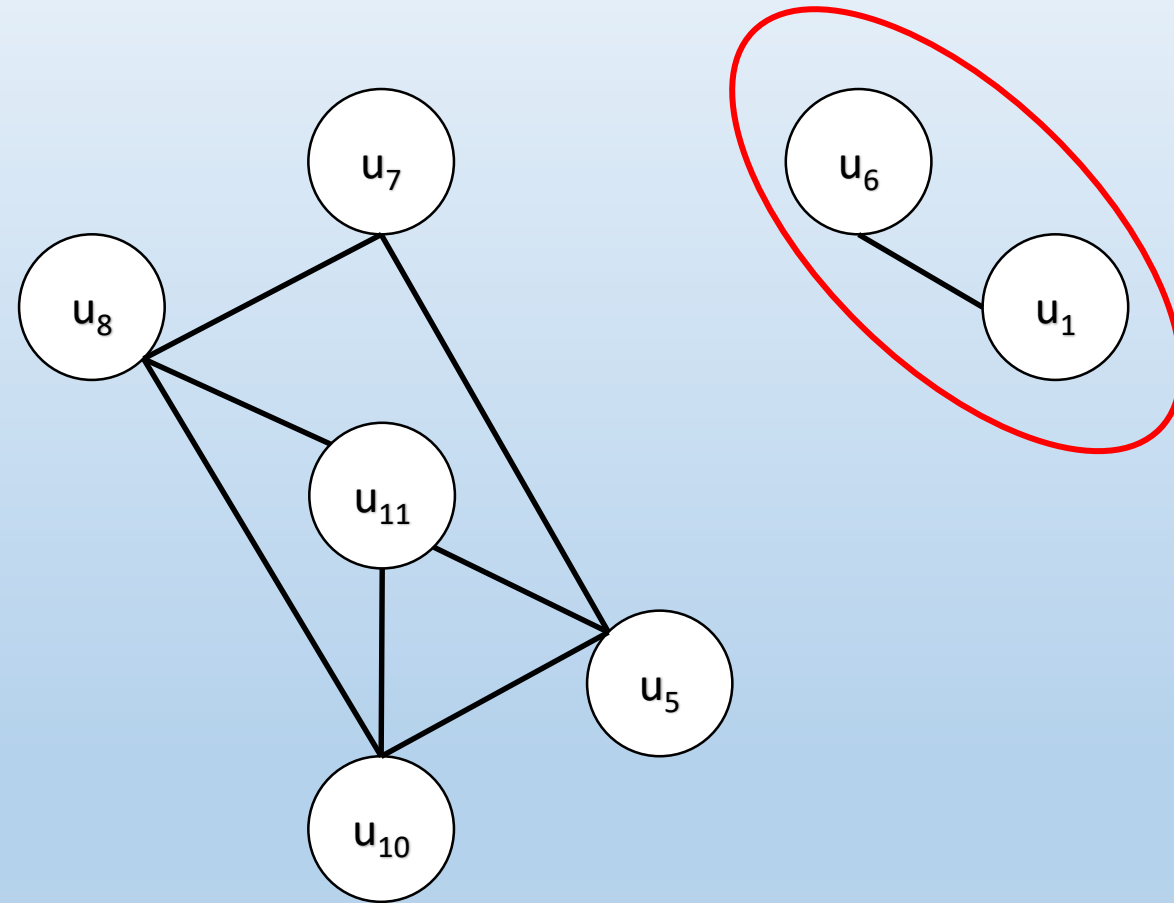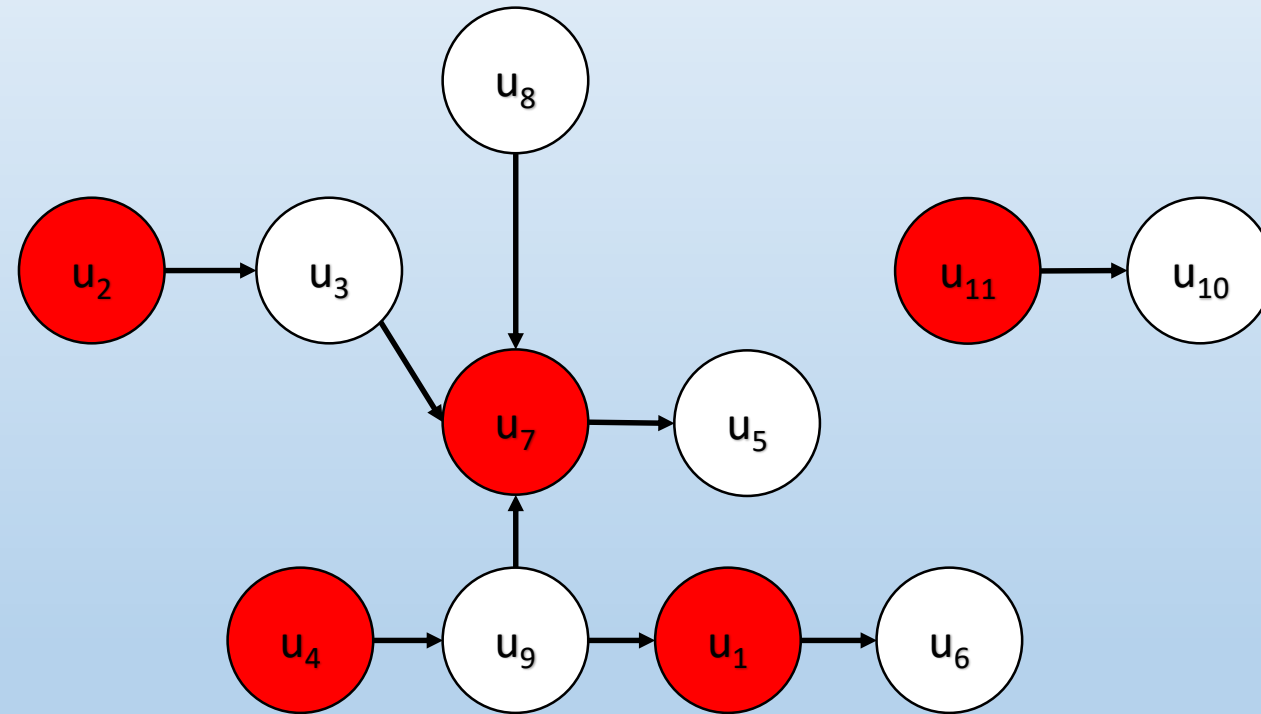
Dependency Graph
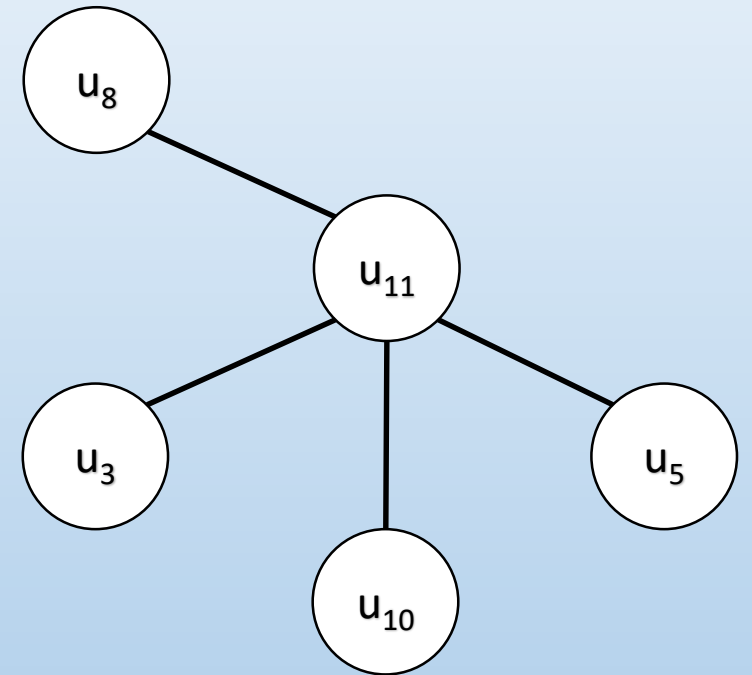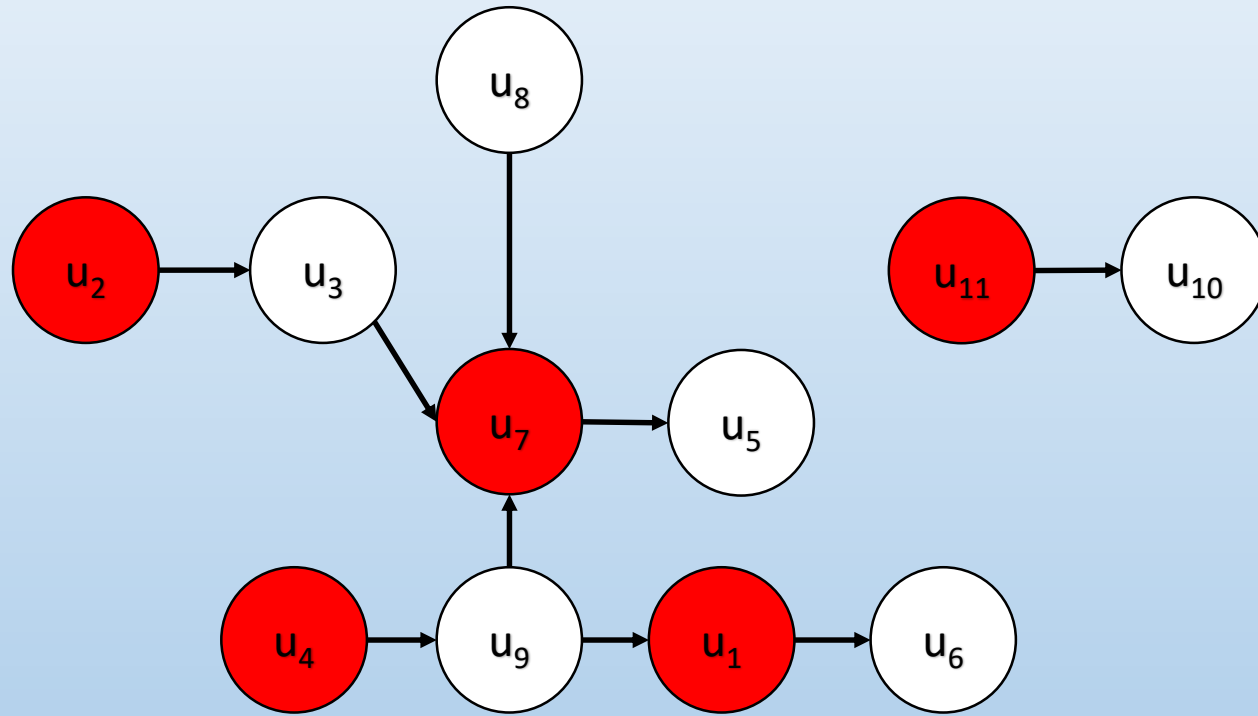
Dependency Graph

Dependency Graph
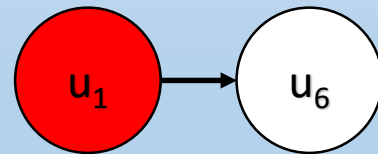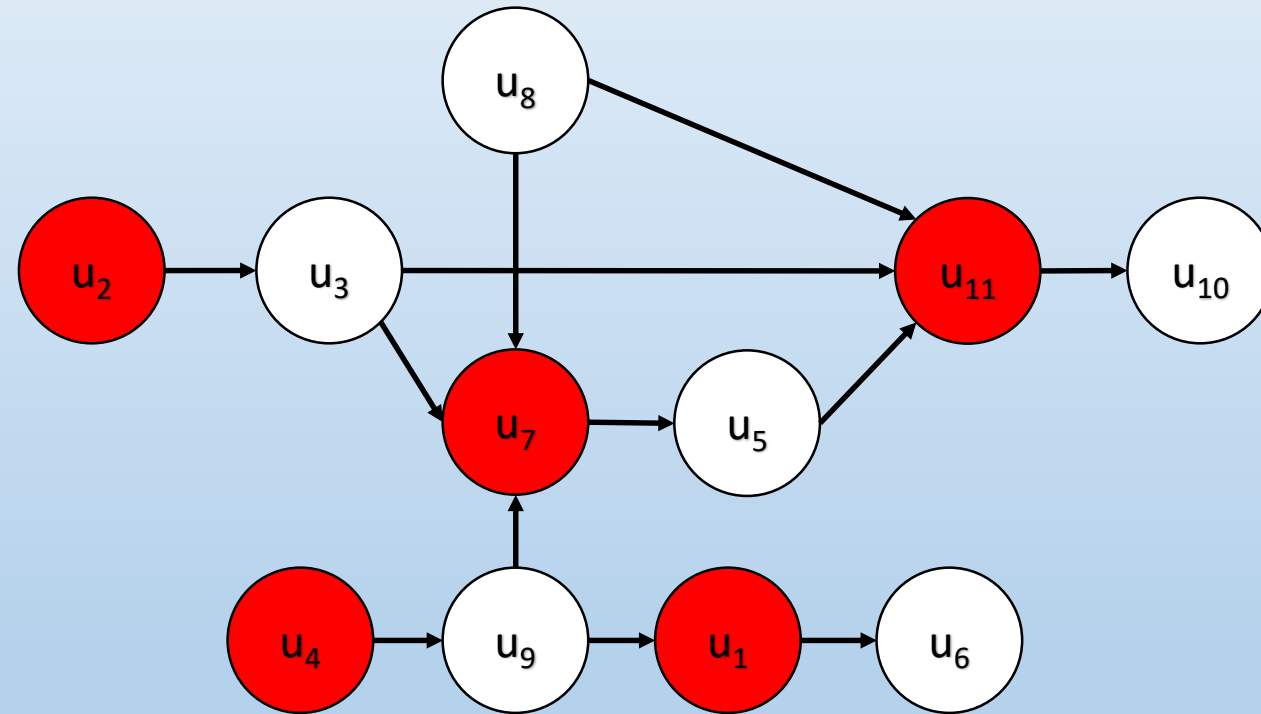
DG: Dealing with Updates

We mainly focus on the following three operations.

- Delete a node u
  - u is a reducing node.
  - Try to move out u.
- Add an edge (u, v)
  - Both u and v are reducing nodes.
  - Try to move out u or v. Simply abandon either if it fails.
- Delete an edge (u, v)
  - Try to move in the dependent node.

DG: Maintenance
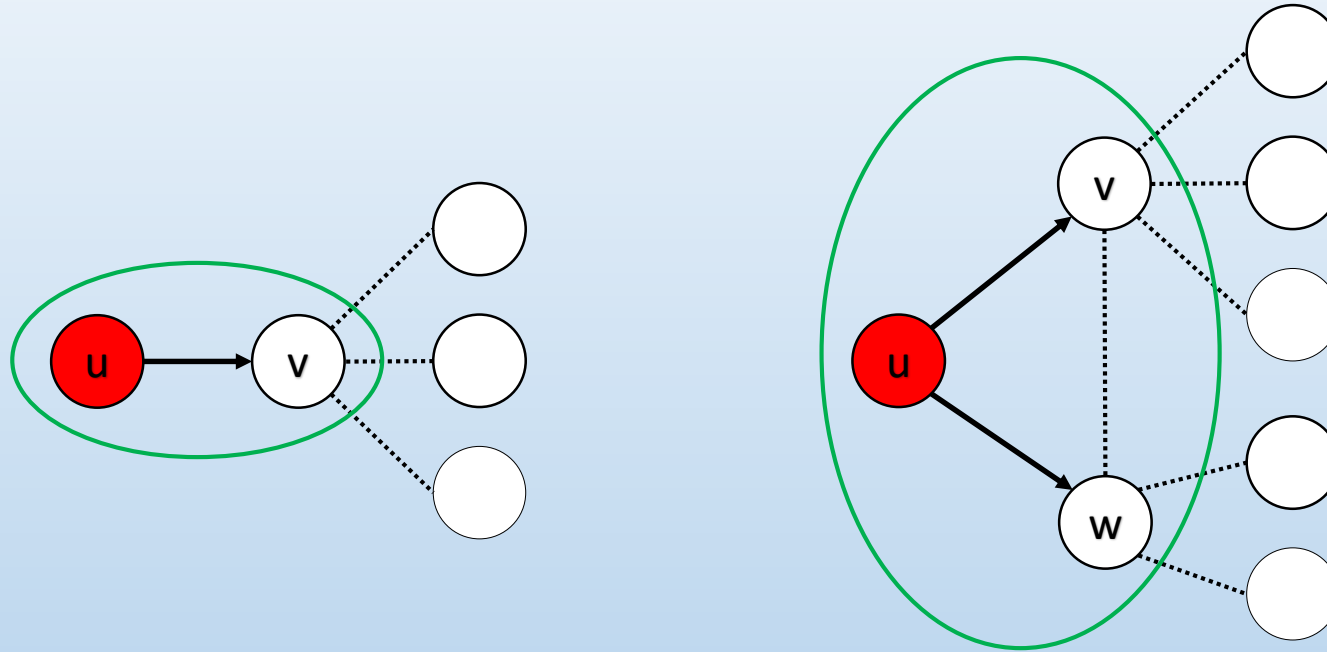
in a Valid Swap

• Each reducing node forms a group with its out-neighbor(s).

• The component of each group does not change. (e.g. {u, v}, {u, v, w})

• The direction of an affected edge changes accordingly.

DG: Maintenance

in a group
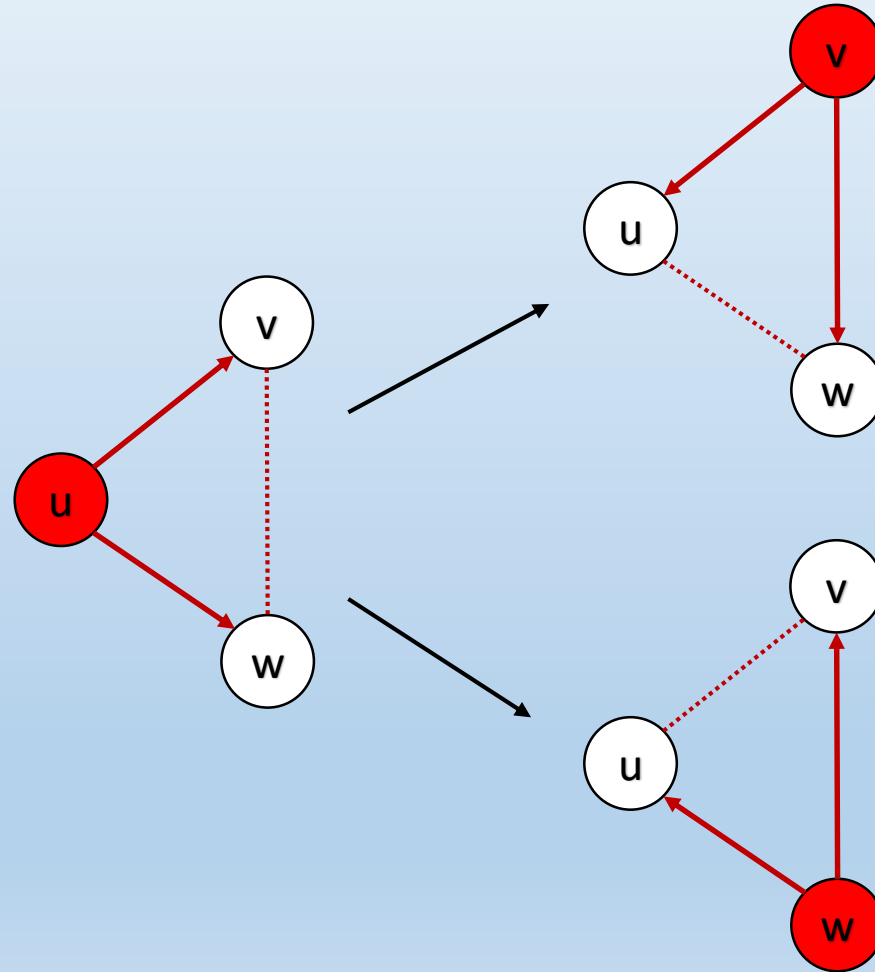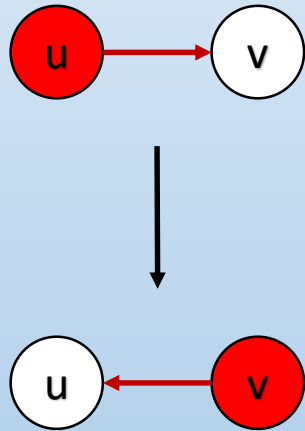
DG: Maintenance

between two groups
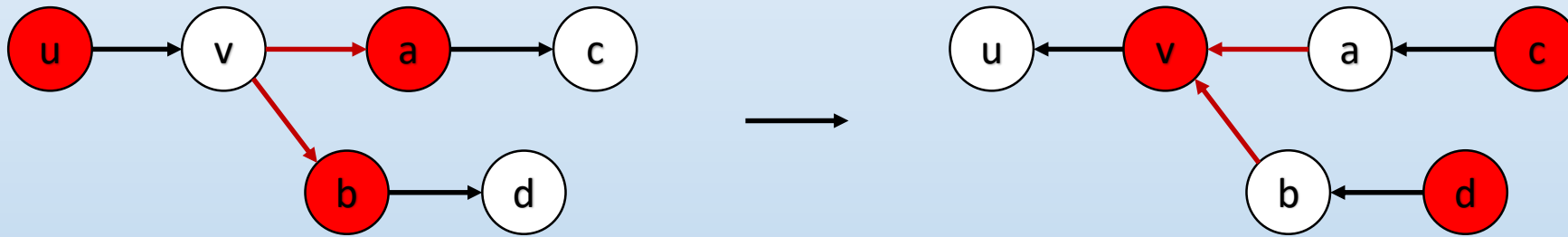
The nodes not in this swap are colored black.
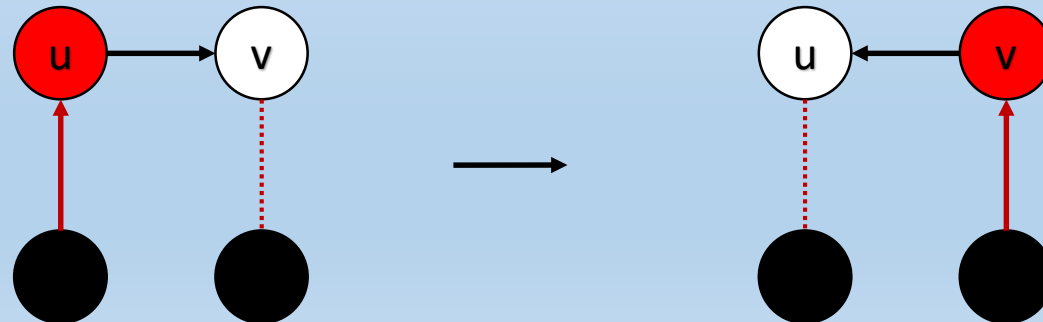
DG: Maintenance

add an edge (u, v): move u out successfully
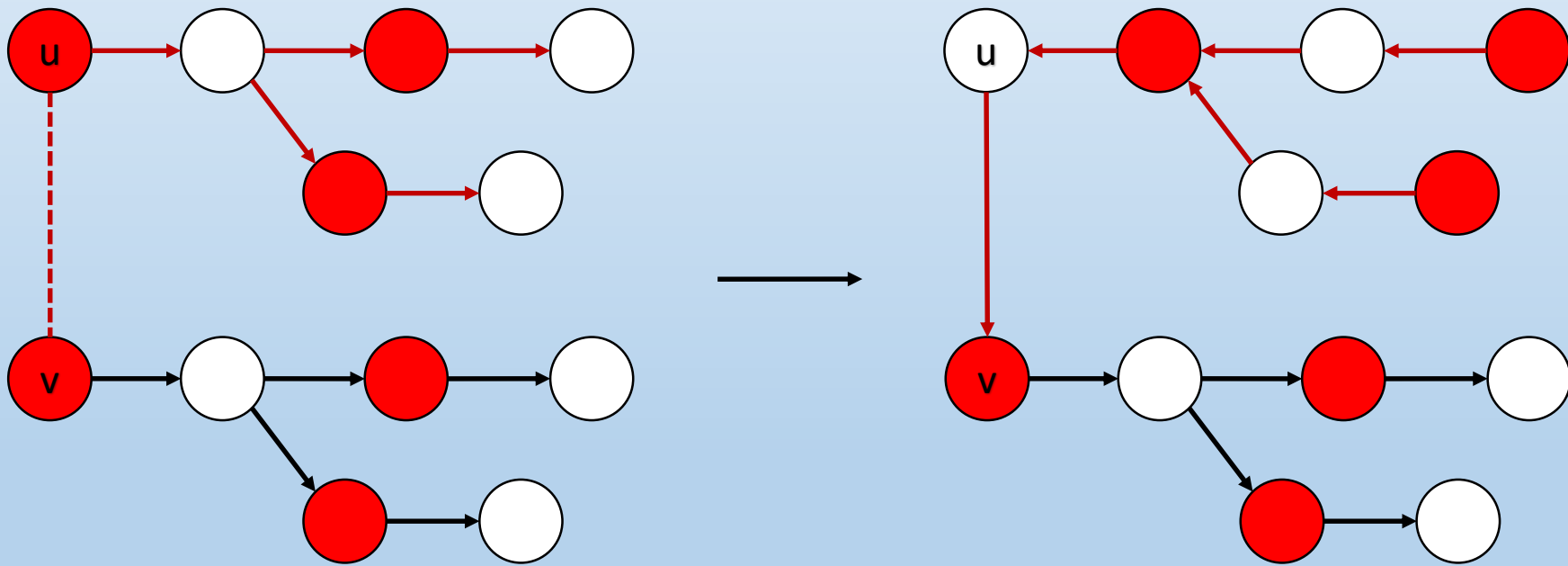
DG: Maintenance

What if we fail to move both u and v out?

- It means that the original out-neighbor on DG is not a good choice.
- The nodes u and v do not even have any out-neighbor.
- In this case, we need to regroup related nodes.
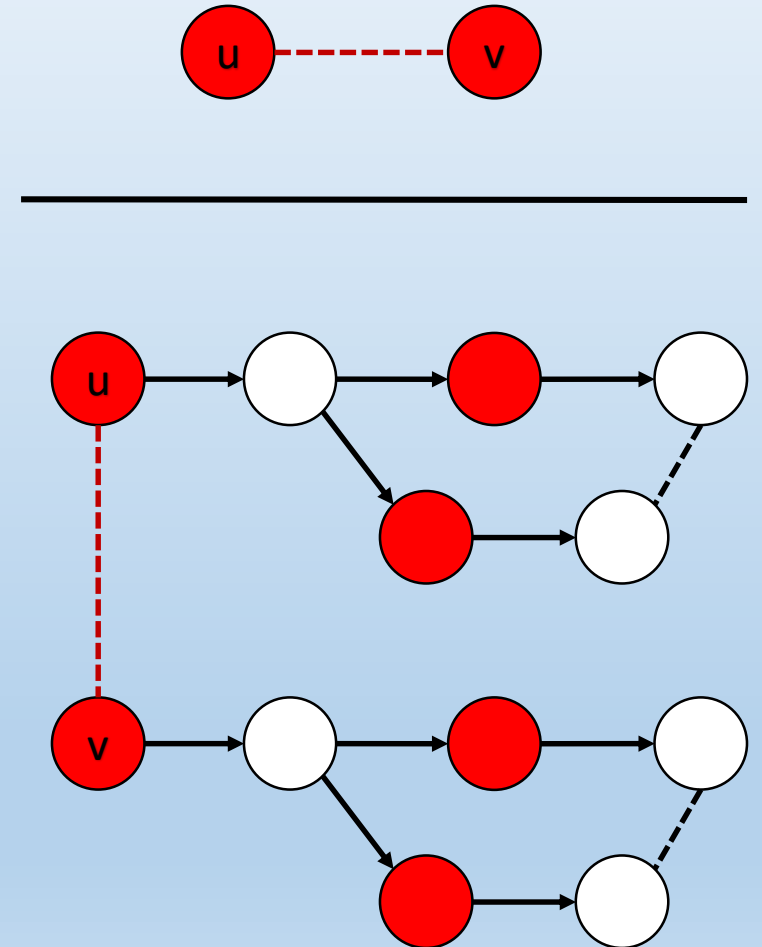
DG: Maintenance

The nodes u and v form a new group.
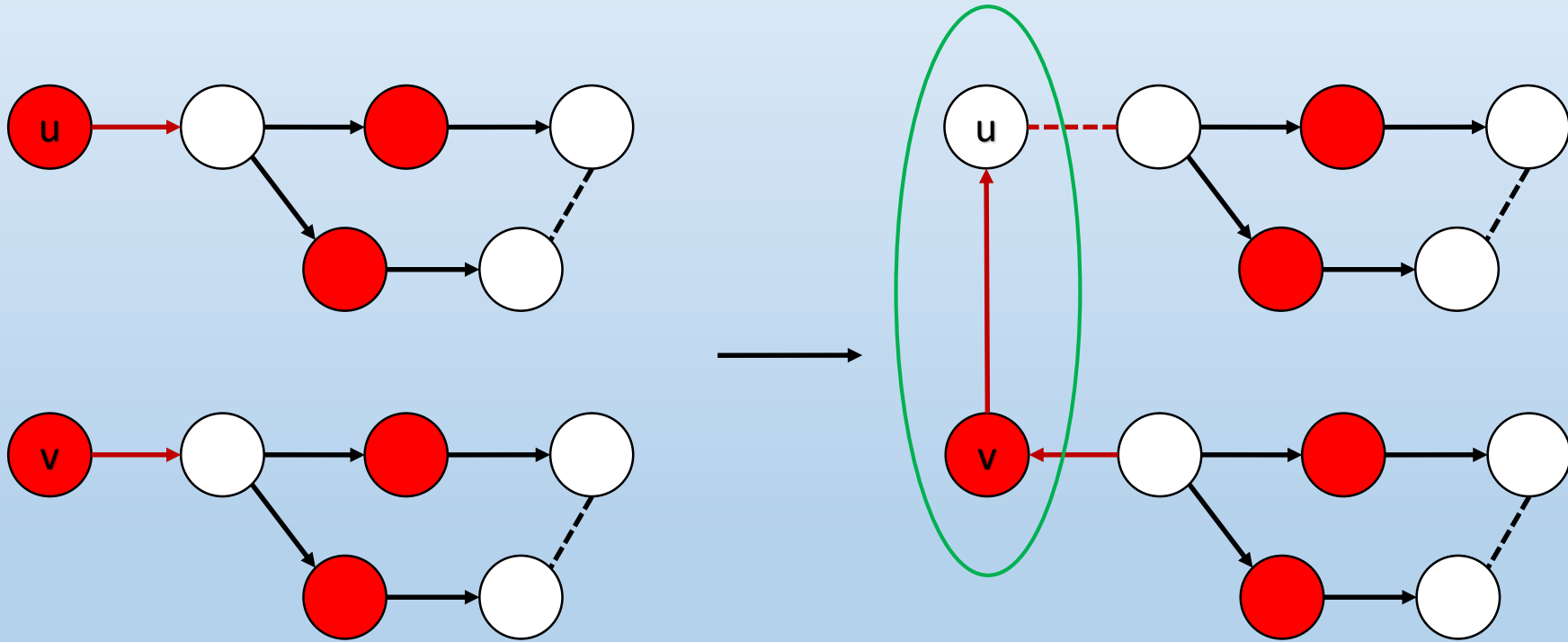
Say goodbye to the bad choice.

DGOracle

Actually, we do not use the given Near-MIS$_0$. (How tricky!!!)

Because the reducing-peeling framework can generate an initial solution Near-MIS$_{RP}$. (Its original job, is not it?)

Can we make use of Near-MIS$_0$ if it is better than NEAR-MIS$_{RP}$?

Specifically, what if Near-MIS$_0$ is exactly a MIS. (Gap$_0$ = 0)

DGOracle

A small modification is enough.
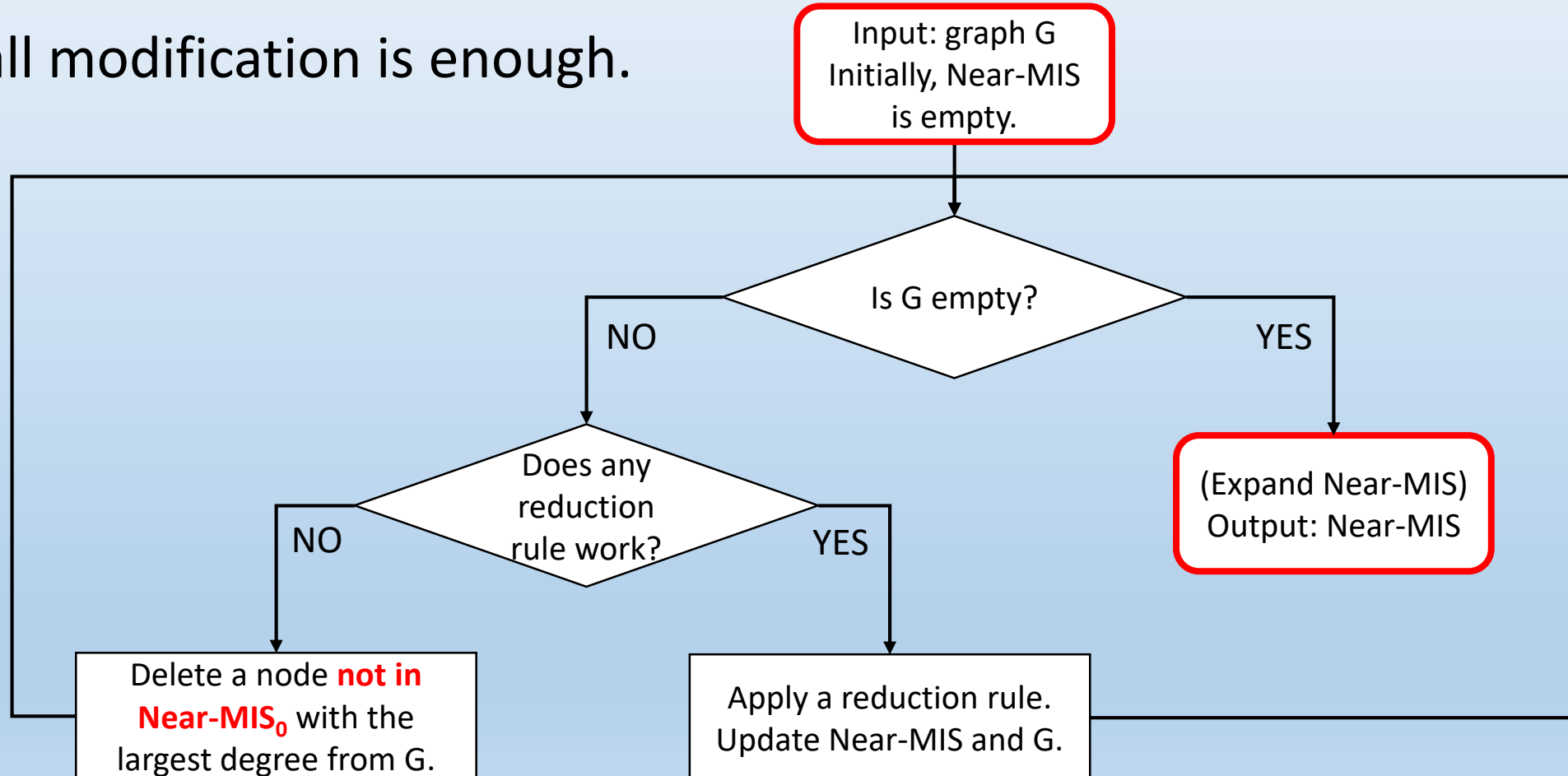
DGOracle

Delete a node **not in** **Near-MIS$_0$** with the largest degree from G.

Here, the inexact reduction is guided by Near-MIS$_0$.

If Near-MIS$_0$ contains node u, we believe that u is a good node even if it has the largest degree.

When the Near-MIS$_0$ is optimal(MIS), this step even becomes exact.

Therefore, we call the dependency graph constructed in this method DGOracle.

Batch Update

How to compute Near-MIS$_k$ based on Near-MIS$_0$ directly?

In other words, we want to solve k updates in a batch.

This problem is very reasonable when many updates are concentrated in a small region.

Batch Update

Affected Subgraph(G_a)

- Add/delete an edge (u, v): u, v and their neighbors are affected.
- Delete a node u: u's neighbors are affected.
- The affected graph G_a is induced by all affected nodes.
- We use G_r to denote the remaining graph.
- An edge between G_a and G_r is called a crossing edge.

Batch Update

Active Node

For each crossing edge (u, v)

- u is in G_a and v is in $G_{r.}$
- v is a reducing node(v is in Near-MIS).
- Then u is called an active node.

Batch Update
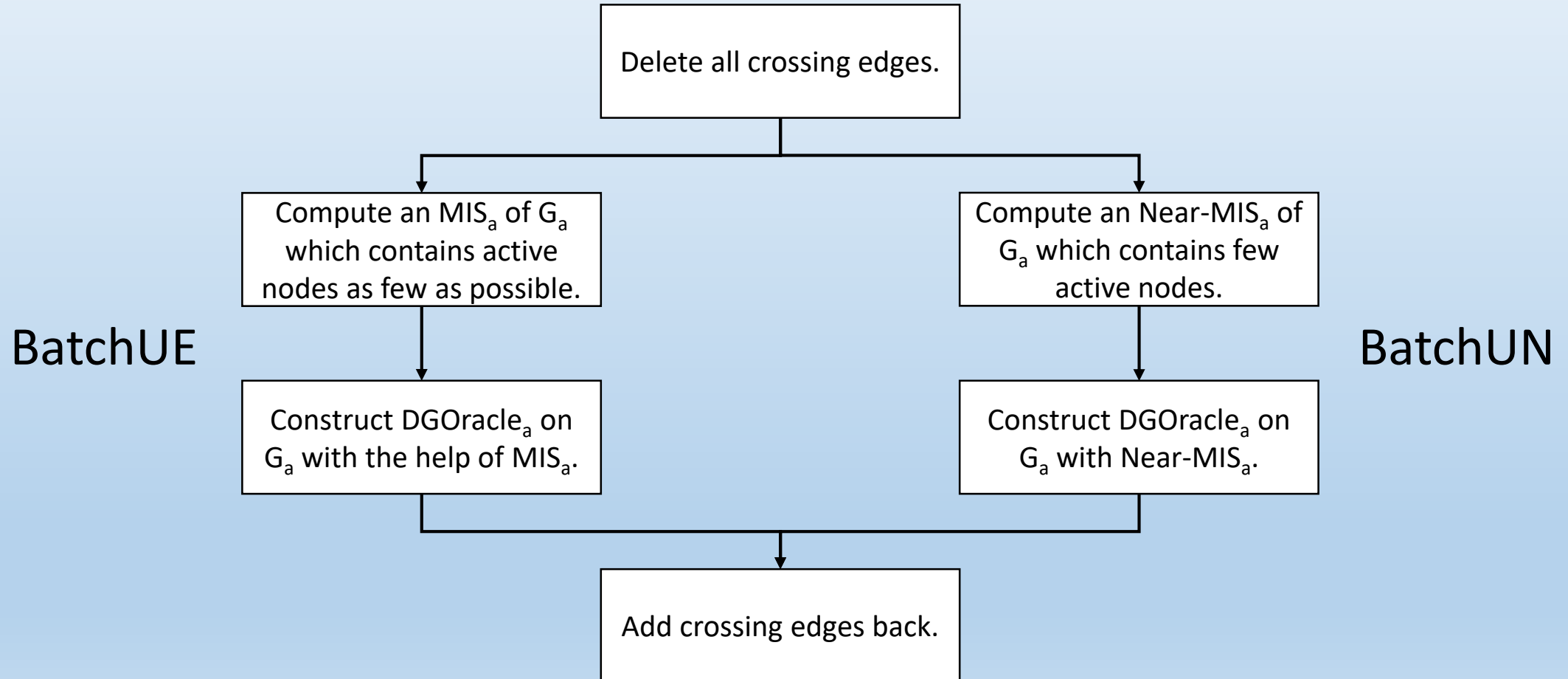
Experiment

Experimental Setting

We use 18 real-world networks and randomly generate 10,000 updates on each graph.

- LSTwo+LazySearch[8]

- DGOne-DIS(without d2 reduction)

- DGTwo-DIS

- DGOracle-DIS

- BatchUE/BatchUN

ID	Graph	|V(G)|	|E(G)|	d
G01	Wiki-Vote	7,115	100,762	28.32
G02	CA-HepPh	12,006	118,489	19.74
G03	AstroPh	18,771	198,050	21.1
G04	Brightkite	58,228	214,078	7.35
G05	Epinions	75,879	405,740	10.69
G06	Slashdot	82,168	504,230	12.27
G07	com-dblp	317,080	1,049,866	6.62
G08	com-amazon	334,863	925,872	5.53
G09	BerkStan	685,230	6,649,470	19.41
G10	web-Google	875,713	4,322,051	9.87
G11	soc-pokec	1,632,803	22,301,964	27.32
G12	as-skitter	1,696,415	11,095,298	13.08
G13	wiki-tomcats	1,791,489	25,444,207	28.41
G14	Wiki-Talk	2,394,385	4,659,565	3.89
G15	com-orkut	3,072,441	117,185,083	76.28
G16	cit-Patents	3,774,768	16,518,947	8.75
G17	com-lj	3,997,962	34,681,189	17.35
G18	LiveJournal	4,846,609	42,851,237	17.68

Gap(Accuracy)

TABLE II

The gaps caused by 1000 updates and the memory usage

Graphs	LSTwo+LazySearch[+]		DGOneDIS		DGTwoDIS		DGOracleDIS		BatchUE		BatchUN		Memory Usage (MB)		
	AGap	RGap	AGap	RGap	AGap	RGap	AGap	RGap	AGap	RGap	AGap	RGap	DGOneDIS	DGTwoDIS	BatchUE
Wiki-Vote	23	23	1	0	1	1	1	1	1	0	1	1	2.48	2.51	2.52
CA-HepPh	62	62	10	9	6	6	6	6	5	2	4	-5	2.99	3.04	3.76
AstroPh	35	35	12	11	5	5	3	3	7	7	5	5	4.98	5.05	5.14
Brightkite	17	17	2	1	1	1	1	1	1	0	1	1	6.29	6.51	6.76
Epinions	19	19	1	1	0	0	0	0	1	0	1	0	11.09	11.39	11.48
Slashdot	26	26	3	1	2	0	0	0	1	1	2	1	13.49	13.81	14.04
com-dblp	72	72	22	10	6	5	5	5	5	4	6	4	31.59	32.81	32.93
com-amazon	89	89	150	-2	21	-2	0	0	3	-5	4	-5	29.18	30.45	30.86
BerkStan	838	838	1074	5	761	4	18	18	15	0	12	3	168.53	171.15	172.28
web-Google	306	306	314	18	173	7	10	10	11	5	16	5	119.8	123.14	124.21
as-skitter	371	371	393	-2	295	0	0	0	7	0	14	-1	294.39	300.86	302.38
Wiki-Talk	74	74	1	0	1	0	0	0	1	0	1	1	163.74	172.87	173.52
LiveJournal	414	414	1458	4	388	-2	2	2	10	-2	18	-2	1096.34	1114.83	1115.12

Efficiency Test

Graphs	LSTwo+LazySearch$^+$	DGOneDIS		DGTwoDIS		DGOracleDIS		BatchUE		BatchUN		DGOneDY		DGTwoDY		DGOracleDY	
	ms	#	ms	#	ms	#	ms	#	ms	#	ms	#	ms	#	ms	#	ms
Wiki-Vote	19.3	1.1	1.3	5.6	7.2	3.6	4.5	1.3	1.5	0.28	0.3	1.3	1.5	5.5	7.1	3.9	4.6
CA-HepPh	36.6	2.7	3.2	3.7	4.6	4.4	5.4	1.7	1.9	0.78	0.9	2.6	3.3	4.2	4.8	4.9	5.7
AstroPh	63.7	1.4	1.5	2.8	3.3	5.8	6.9	0.76	0.9	0.31	0.4	1.3	1.5	3.1	3.7	6.7	7.1
Brightkite	104.6	3.6	4.1	3.9	4.5	7.3	8.3	1.8	2.2	0.91	1.1	4.0	4.6	3.8	4.5	6.9	7.7
Epinions	72.1	0.92	1.2	1.4	1.7	5.2	6.6	0.71	0.85	0.28	0.33	1.1	1.4	1.2	1.8	5.1	6.3
Slashdot	166.8	1.8	2.3	2.3	2.8	7.5	9.3	0.79	0.9	0.17	0.2	1.9	2.1	2.2	2.7	7.4	8.4
com-dblp	88.7	1.1	1.4	1.7	1.9	3.2	4.2	0.56	0.66	0.29	0.41	1.3	1.6	2.1	2.5	3.1	4.2
com-amazon	1209.3	2.1	2.6	2.2	2.7	4.3	5.1	1.2	1.4	0.41	0.52	2.2	2.8	2.3	2.9	4.3	5.6
BerkStan	2723.4	6.4	7.7	7.5	8.6	8.9	10.5	2.8	3.3	1.3	1.5	5.8	7.2	6.1	7.8	8.2	9.6
web-Google	4045.8	4.7	5.4	5.6	6.3	7.7	9.3	2.2	2.6	0.64	0.72	4.3	5.5	5.2	6.5	8.6	9.7
as-skitter	3659.3	7.2	8.6	8.2	9.8	9.6	11.5	2.8	3.4	0.26	0.29	7.1	8.2	7.9	9.1	8.9	10.8
Wiki-Talk	2325.5	0.9	1.1	1.4	1.6	4.7	5.8	0.55	0.62	0.1	0.11	1.2	1.5	1.4	1.8	5.6	6.4
LiveJournal	7521.4	4.2	5.3	12.9	15.2	10.1	12.4	1.1	1.6	0.29	0.39	4.6	5.5	12.9	14.8	9.5	11.2

Scalability

(a) *AGap vs.* $|T|$

(b) Response time *vs.* $|T|$

(c) *DGTwoDIS: AGap*

(d) *DGTwoDIS*: time efficiency
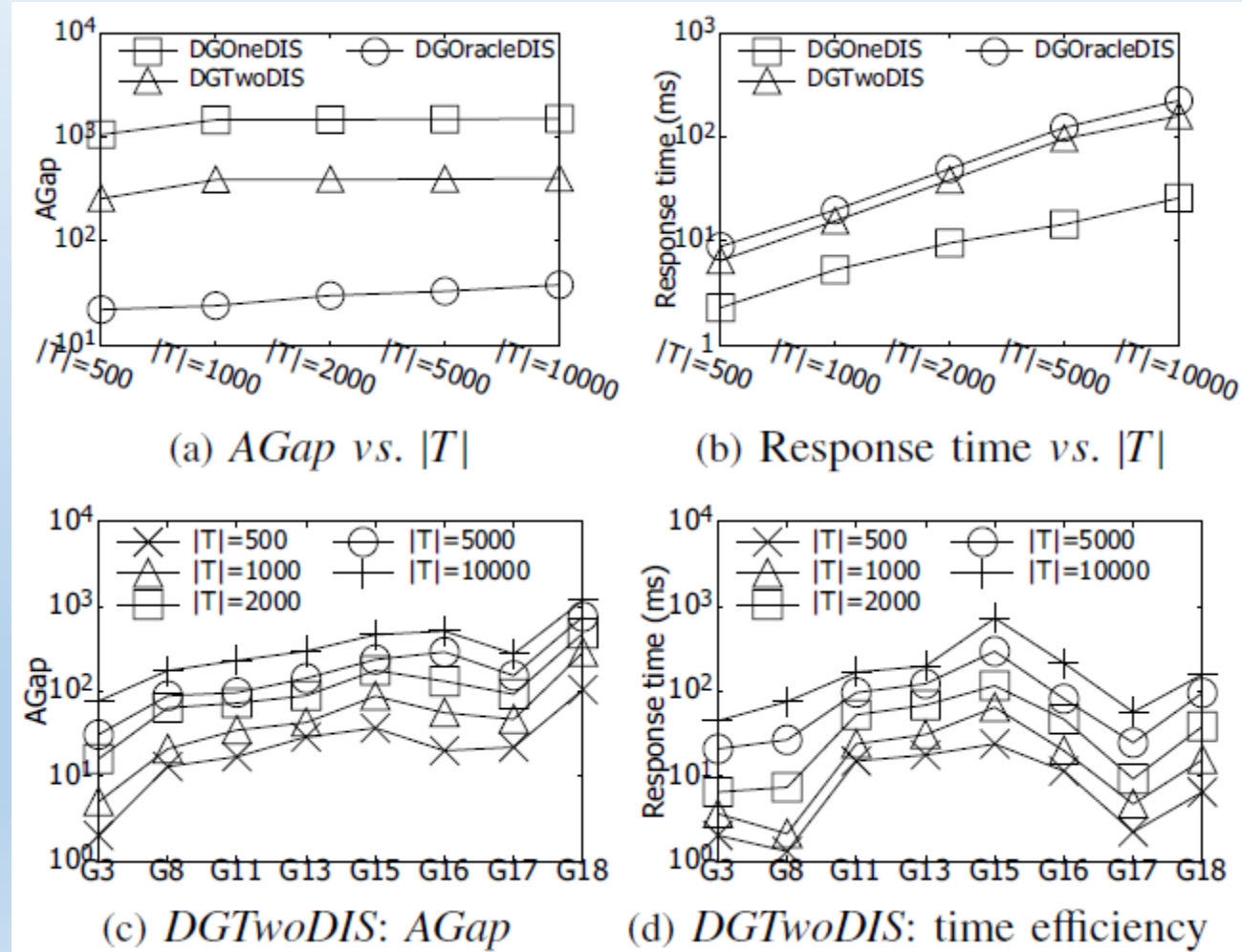
Conclusion

In our paper, we propose a new method to maintain a Near-MIS in dynamic graphs:

Construct a dependency graph to guide the searching procedure based on the traditional reducing-peeling framework.

- Maintain the high quality of a Near-MIS during updates.
- Reduce the time complexity of each update to $O(d)$ in average case.

Related Work

Exact Algorithm

[19] T. Akiba and Y. Iwata

Branch-and-reduce exponential/FPT algorithms in practice: A case study of vertex cover

Theor. Comput. Sci., vol. 609, pp. 211–225, 2016.

https://github.com/wata-orz/vertex_cover

They offer a very good implementation(VCSolver) of finding a minimum vertex cover.

Tip: A minimum vertex cover is the complementary set of a MIS.

Inexact Algorithm

[34] L. Chang, W. Li, and W. Zhang

Computing A near-maximum independent set in linear time by reducing-peeling

Proceedings of the SIGMOD Conference, 2017, pp. 1181–1196.

They propose several powerful reduction rules which significantly increase the accuracy.

Inexact Algorithm in Dynamic Graphs

[8] W. Zheng, Q. Wang, J. X. Yu, H. Cheng, and L. Zou

Efficient computation of a near-maximum independent set over evolving graphs

ICDE, 2018

Target: find the MIS/IS at time t_{i+1} based on the MIS/IS at time t_i

- Develop an efficient constant-time algorithm ***LSTwo*** to return a high quality (large-size) IS.

- Design a **lazy search** algorithm which produces higher-quality ISs.