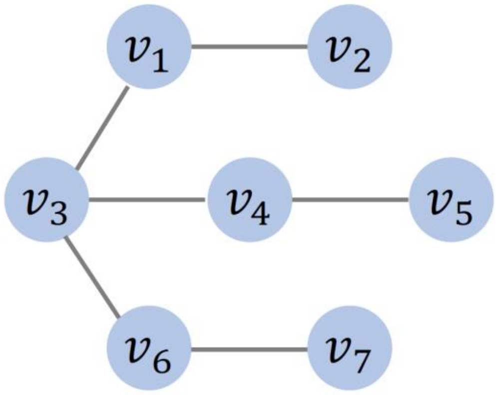
Graph Algorithms 2

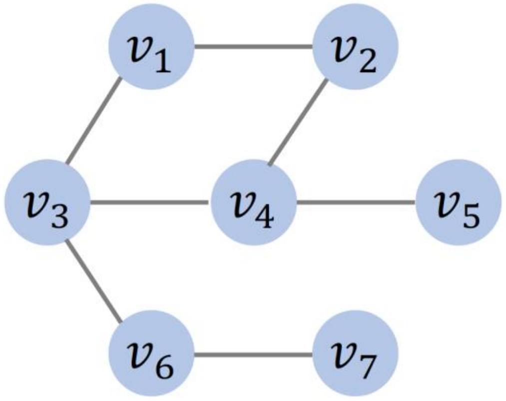
Outline

- Prim’s algorithm
- Kruskal’s algorithm
- A* Search algorithm
- Max Flow Problem
- Bipartite graph

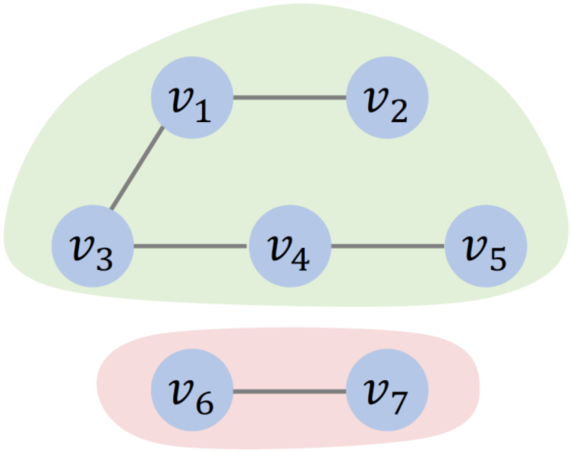
Tree vs Graph

- Trees are undirected graphs (not all undirected graphs are trees)
- Trees do not have cycles
- Trees are connected graphs, connected acyclic undirected graphs
- If a tree has n vertices, then it has $n - 1$ edges
  - Less than $n - 1$ edges -> Disconnected
  - More than $n - 1$ edges -> There is a cycle





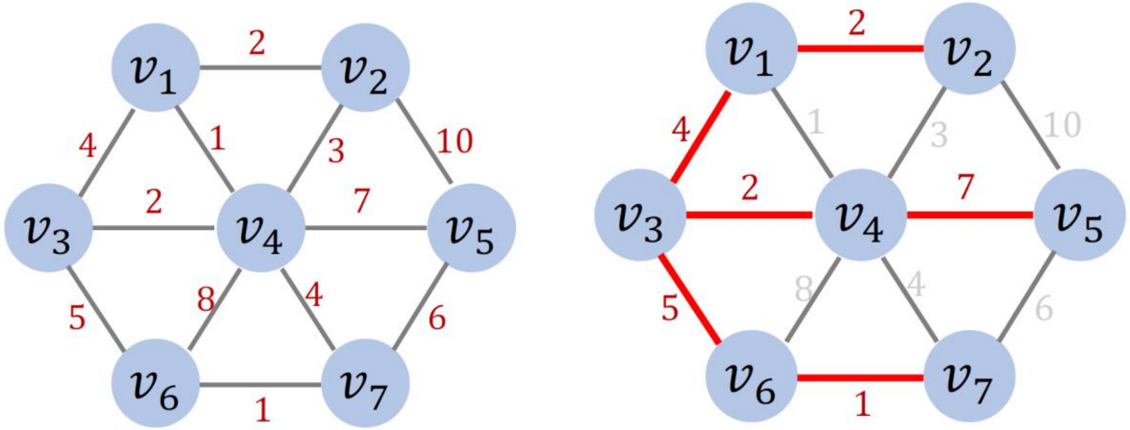
Not a tree

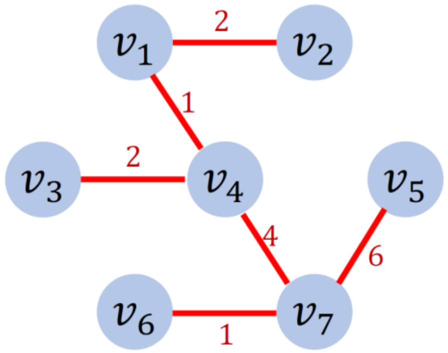


Not a tree

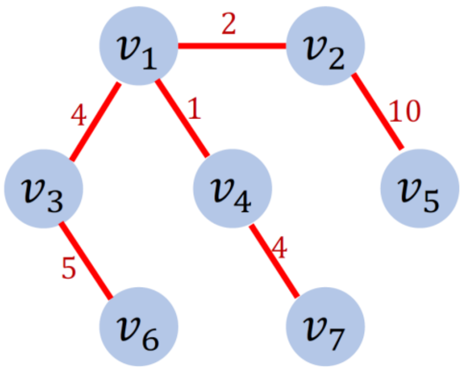
Spanning Trees

- Input: a **connected** undirected graph G with n vertices
- Find such a subgraph:
  - Keep all the n vertices
  - Keep n – 1 edges
  - The subgraph is connected
- The subgraph is a spanning tree
  - Not unique
  - For G with positive edge weights: **Minimum spanning tree (MST)** is a spanning tree that minimizes the sum of weights

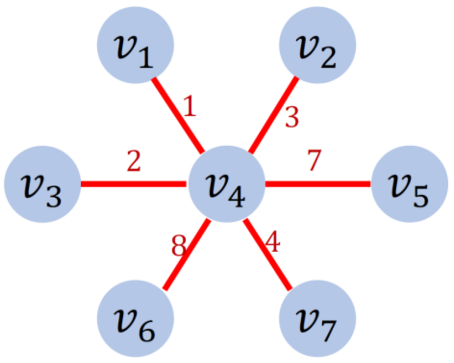




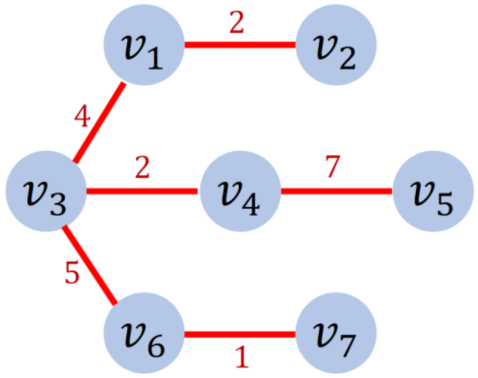
Sum of weights:16



Sum of weights:26



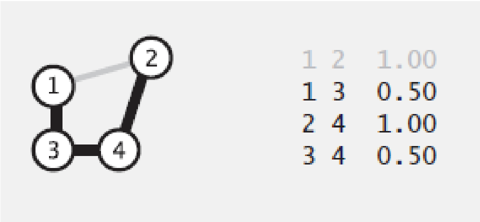
Sum of weights:25

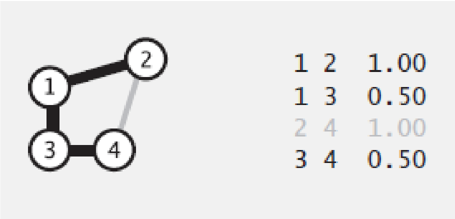


Sum of weights:21

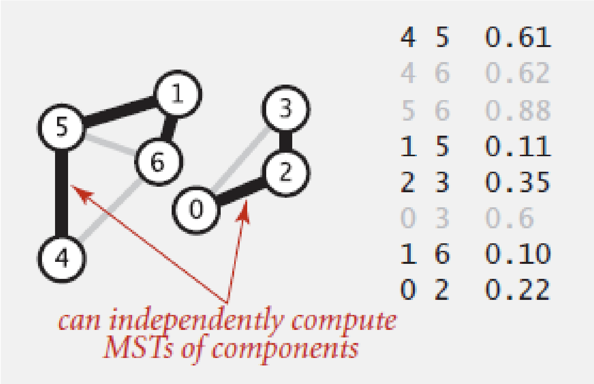
Minimum spanning tree (MST)

- What if edge weights are not all distinct?
  - Greedy MST algorithms still correct if equal weights are present!





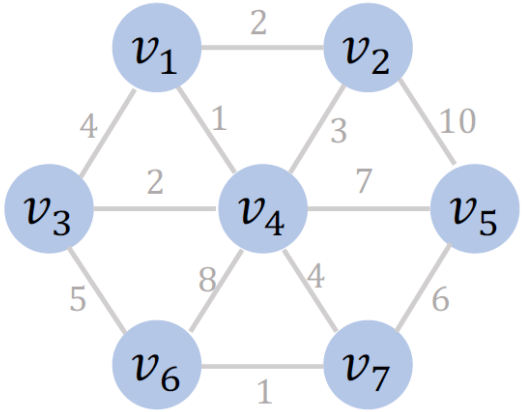
- What if graph is not connected?
  - No MST, but there is a minimum spanning forest = MST of each component



- How to represent the MST
  - A list of edges (with weights)

Prim's Algorithm

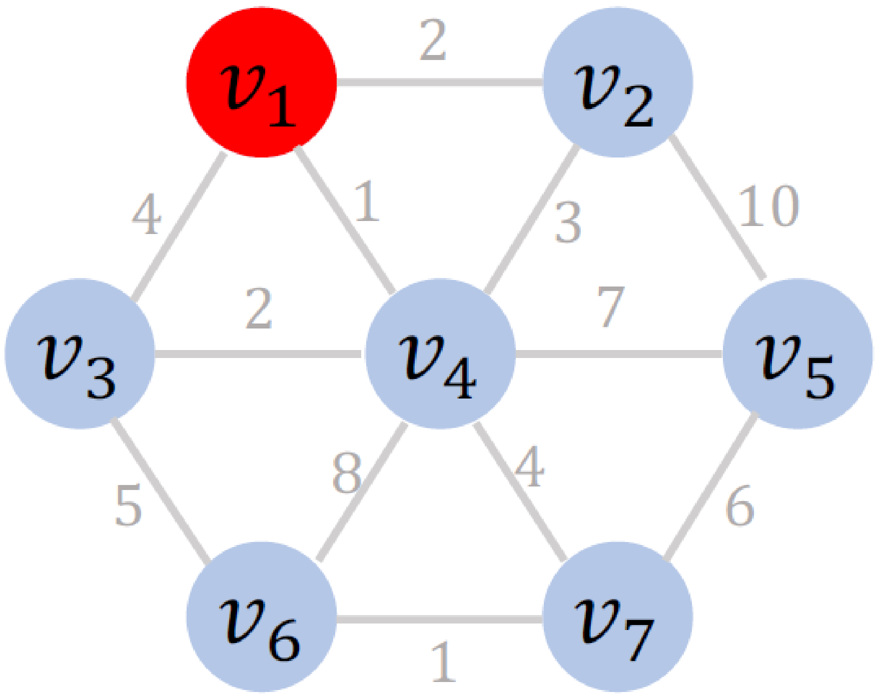
- Basic idea: Grow the tree in successive stages
- Initially, the tree has one vertex and no edge
- In each iteration, add one vertex and one edge to the tree
- Throughout, maintain the properties of trees:
  - Connectivity
  - No cycle: disregard if the vertices are visited
- The algorithm runs in n iterations (n is the number of vertices)



$\mathcal{U}$: vertices of spanning tree

Iteration 1

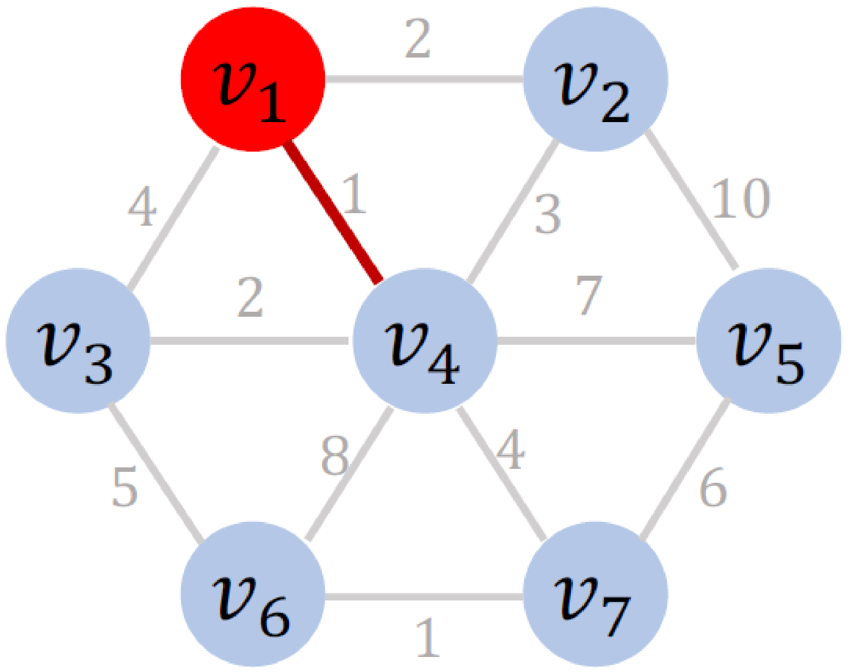
- Pick any vertex in the graph
- Maybe pick v_1
- Add v_1 to $\mathcal{U}$



$\mathcal{U} = \{v_1\}$

Iteration 2

- The edges connecting $\mathcal{U}$ to $\mathcal{V}\backslash\mathcal{U}$:
  $$e_{1,2}, e_{1,3}, e_{1,4}$$
- Among them, $e_{1,4}$ has the smallest weight, record it
- Add v_4 to $\mathcal{U}$



$$\mathcal{U} = \{v_1\}$$

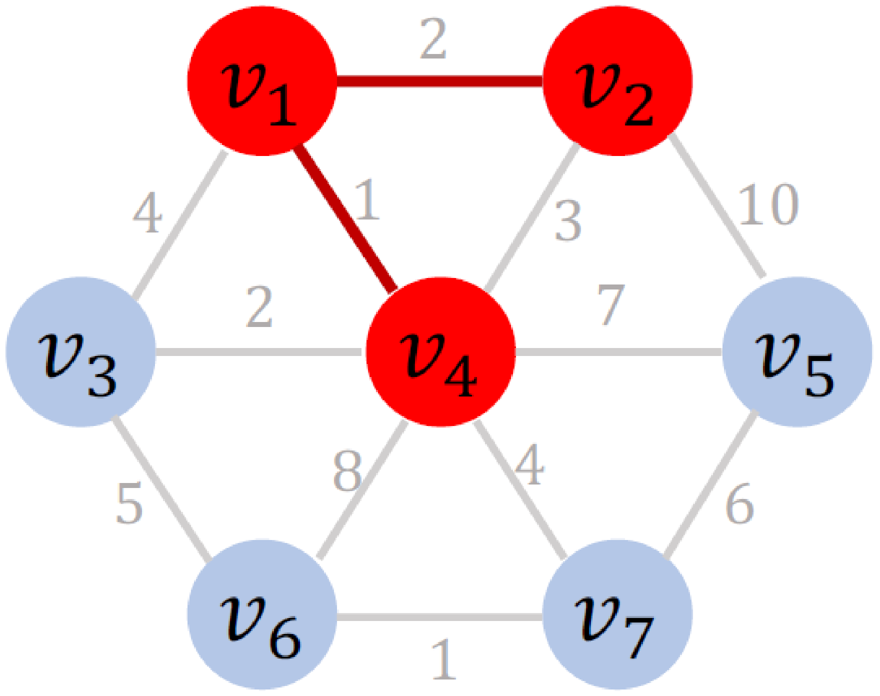
Iteration 3

- The edges connecting $\mathcal{U}$ to $\mathcal{V} \backslash \mathcal{U}$:

$$e_{1,2}, e_{1,3}$$

$$e_{4,2}, e_{4,3}, e_{4,5}, e_{4,6}, e_{4,7}$$

- Among them, $e_{1,2}$ has the smallest weight, record it
- Add v_2 to $\mathcal{U}$



$$\mathcal{U} = \{v_1, v_4, v_2\}$$

Iteration 4

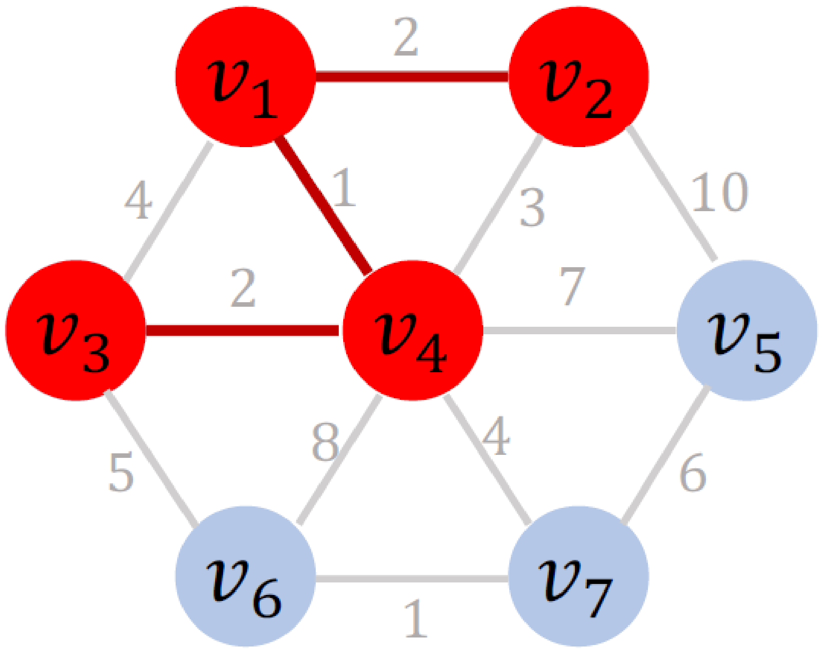
- The edges connecting $\mathcal{U}$ to $\mathcal{V} \backslash \mathcal{U}$:

$$e_{1,3}$$

$$e_{4,3}, e_{4,5}, e_{4,6}, e_{4,7}$$

$$e_{2,5}$$

- Among them, $e_{4,3}$ has the smallest weight, record it
- Add v_3 to $\mathcal{U}$.



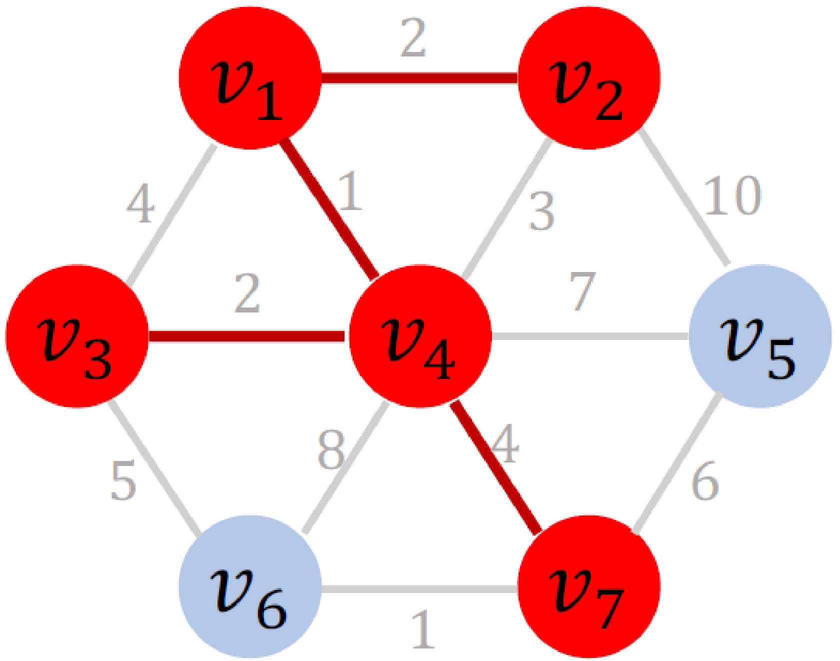
$\mathcal{U} = \{v_1, v_4, v_2, v_3\}$

Iteration 5

- The edges connecting $\mathcal{U}$ to $\mathcal{V}\backslash\mathcal{U}$:

$$e_{4,5}, e_{4,6}, e_{4,7}$$
$$e_{2,5}$$
$$e_{3,6}$$

- Among them, $e_{4,7}$ has the smallest weight, record it
- Add v_7 to $\mathcal{U}$



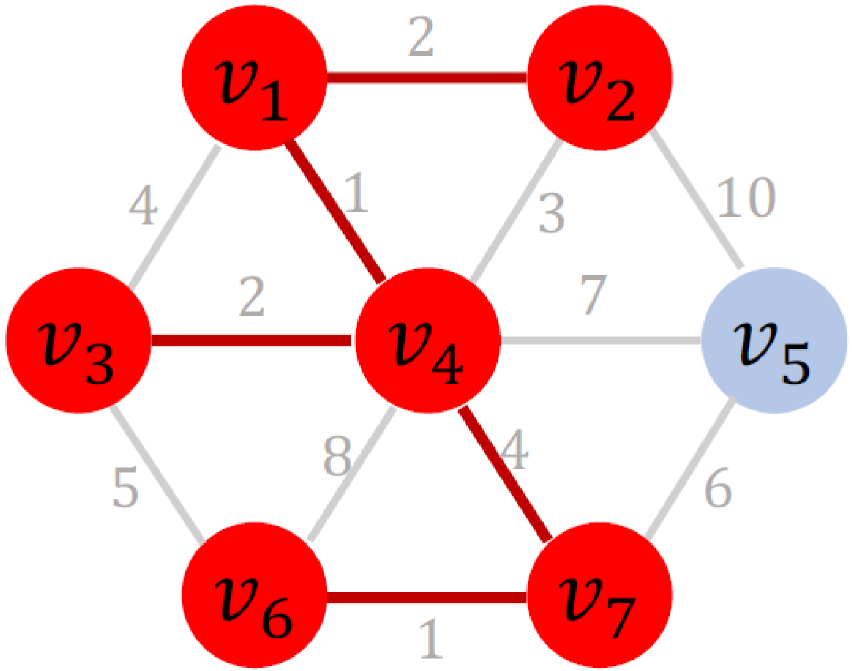
$\mathcal{U} = \{v_1, v_4, v_2, v_3, v_7\}$

Iteration 6

- The edges connecting $\mathcal{U}$ to $\mathcal{V} \backslash \mathcal{U}$:

$$e_{4,5}, e_{4,6}$$
$$e_{2,5}$$
$$e_{3,6}$$
$$e_{7,5}, e_{7,6}$$

- Among them, $e_{7,6}$ has the smallest weight, record it
- Add v_6 to $\mathcal{U}$



$$\mathcal{U} = \{v_1, v_4, v_2, v_3, v_7, v_6\}$$

Iteration 7

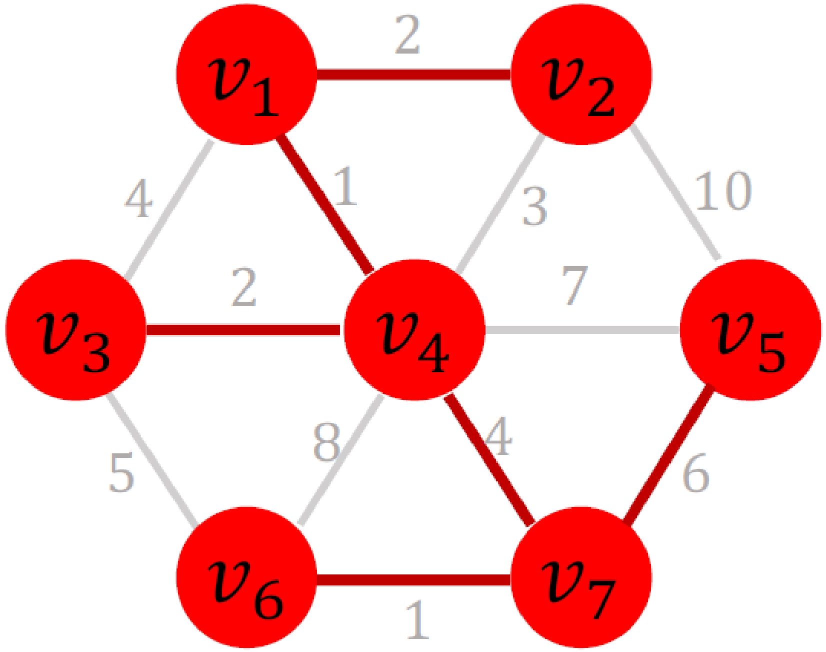
- The edges connecting $\mathcal{U}$ to $\mathcal{V}\backslash\mathcal{U}$:

$$e_{4,5}$$
$$e_{2,5}$$
$$e_{7,5}$$

- Among them, $e_{7,5}$ has the smallest weight, record it
- Add v_5 to $\mathcal{U}$

- After this
  - Now $\mathcal{U}$=$\mathcal{V}$. (All the vertices have been added to $\mathcal{U}$.)
  - Return the tree:

$\{e_{1,4}, e_{1,2}, e_{4,3}, e_{4,7}, e_{7,6}, e_{7,5}\}$



$\mathcal{U} = \{v_1, v_4, v_2, v_3, v_7, v_6, v_5\}$

Prim's Algorithm

1. Initialize
   - Let $\mathcal{T}$ (the set of edges in the MST) be an empty set
   - Create an array minWeight of size n (number of nodes) to store the minimum weight of an edge that connects each node to the MST
   - Initialize minWeight[start] = 0 (starting node with 0 cost to itself) and all other values in minWeight to infinity
   - Create an array inMST of size n to keep track of whether each node is in the MST, initialize all values in visited to false
2. While $\mathcal{T}$ has fewer than n–1 edges
   - Find the unvisited node u with the minimum value in minWeight
   - Mark u as visited and add it to the MST
   - For each neighbor v of u:

     If v is unvisited and the edge weight (u,v) is less than minWeight[v]

       update minWeight[v] to the weight of (u,v)
3. Return $\mathcal{T}$

```cpp
// Prim's algorithm without priority queue
void prim(int start, vector<vector<int>>& graph) {
    int n = graph.size();
    vector<bool> inMST(n, false);     // Track nodes in MST
    vector<pair<int, int>> mstEdges; // Store edges in MST
    int totalCost = 0;
    inMST[start] = true;  // Add starting node to MST
    for (int count = 1; count < n; ++count) {  // Repeat until all nodes are in MST
        int minWeight = INT_MAX;
        int u = -1, v = -1;
        // Find the minimum weight edge (u, v) with u in MST and v not in MST
        for (int i = 0; i < n; ++i) {
            if (inMST[i]) {
                for (int j = 0; j < n; ++j) {
                    if (!inMST[j] && graph[i][j] < minWeight) {
                        minWeight = graph[i][j];
                        u = i;
                        v = j;
                    }
                }
            }
        }
        // Add edge (u, v) to MST
        if (u != -1 && v != -1) {
            inMST[v] = true;
            totalCost += minWeight;
            mstEdges.push_back({u, v});
        }
    }
}
```

Improving using priority queue

- Challenge: Find the min weight edge with exactly one endpoint in $\mathcal{U}$
- Solution: Maintain a PQ of edges with (at least) one endpoint in $\mathcal{U}$
  - Key = edge; priority = weight of edge
  - Delete-min to determine next edge $e_{u,v}$ to add to $\mathcal{U}$
  - Disregard if both endpoints u and v are unvisited (both in $\mathcal{U}$)
  - Otherwise, let v be the unvisited vertex (not in $\mathcal{U}$):
    - add to PQ any edge incident to v (assuming other endpoint not in $\mathcal{U}$)
    - add e to $\mathcal{U}$ and mark v as visited

Minimum edge weight data structure	Time complexity
adjacency matrix, searching	O(n^2)
binary heap, priority queue	O(mlog(n))

n: # of vertices, m: # of edges

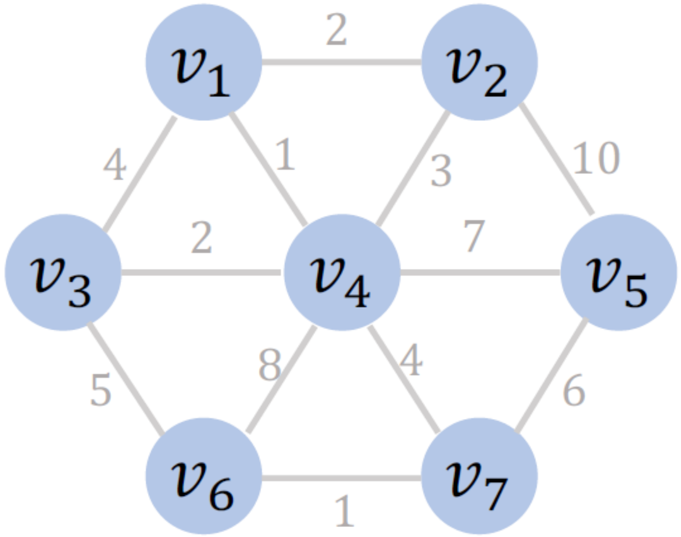
```
MST-Prim(G, V, E){
  for(each x∈V){
    cost(x)=∞;
    parent(x)=Null;
  }
  Choose a node u to be the source or starting point;
  cost(u)=0;
  Insert all vertices to a priority queue PQ;
  while (PQ≠∅){
    x = PQ.Extract_Min();
    for (each neighbor, y of x){
         if (y!∈U && w(x,y)<cost(y)){
            cost(y)=w(x,y);
            parent(y)=x;
            PQ.Decrease_Key(y, cost(y));
         }
    }
  }}
```

Kruskal’s algorithm

- Prim’s Algorithm: vertex-wise; Kruskal’s algorithm: edge-wise
- Basic idea: Maintain a forest, i.e., a collection of trees
- Initially, there are n trees; every vertex is a tree
- Each iteration examines one edge; the edge may be chosen so that two trees are merged
- Stop when there is only one tree
- The algorithm runs in at most m iterations (Because there are m edges)

Preparation

- Build a queue of edges
- Sorted so that the weights are in ascending order



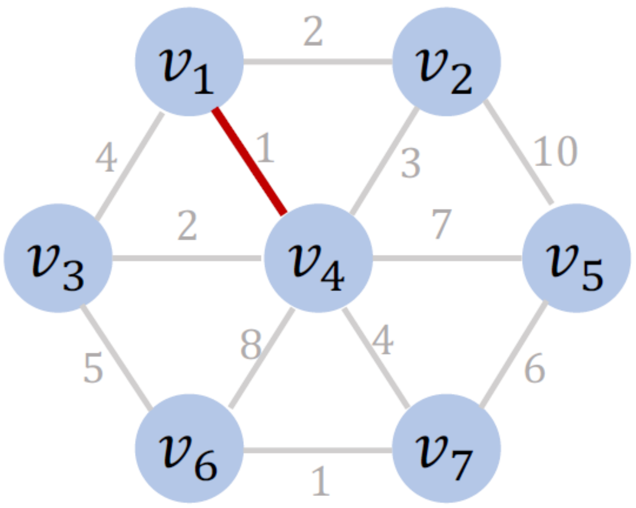
$\mathcal{T} = \emptyset$. (Record the selected edges.)

Edge	Weight
$(1,4)$	1
$(6,7)$	1
$(1,2)$	2
$(3,4)$	2
$(2,4)$	3
$(1,3)$	4
$(4,7)$	4
$(3,6)$	5
$(5,7)$	6
$(4,5)$	7
$(4,6)$	8
$(2,5)$	10

SINGAPORE INSTITUTE OF TECHNOLOGY

Iteration 1

- Dequeue and get the edge (1, 4)
- v_1 and v_4 are not in the same tree.
- Thus **accept** edge (1, 4)
- Append (1, 4) to $\mathcal{T}$

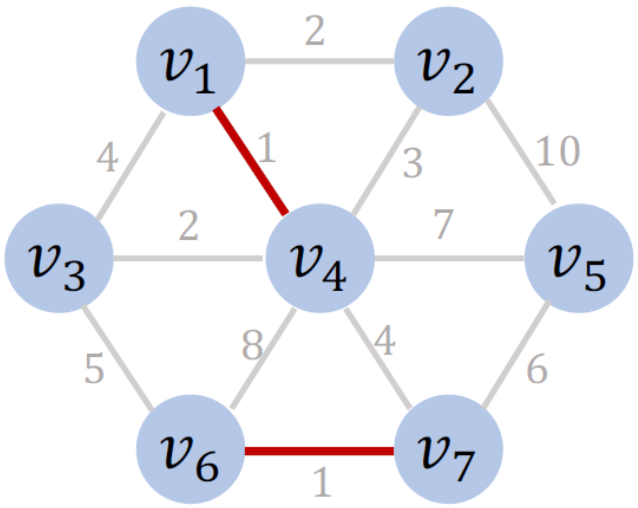


$\mathcal{T} = \{e_{1,4}\}$

Edge	Weight
$(1,4)$	1
$(6,7)$	1
$(1,2)$	2
$(3,4)$	2
$(2,4)$	3
$(1,3)$	4
$(4,7)$	4
$(3,6)$	5
$(5,7)$	6
$(4,5)$	7
$(4,6)$	8
$(2,5)$	10

Iteration 2

- Dequeue and get the edge (6, 7)
- v_6 and v_7 are not in the same tree
- Thus **accept** edge (6, 7)
- Append (6, 7) to $\mathcal{T}$

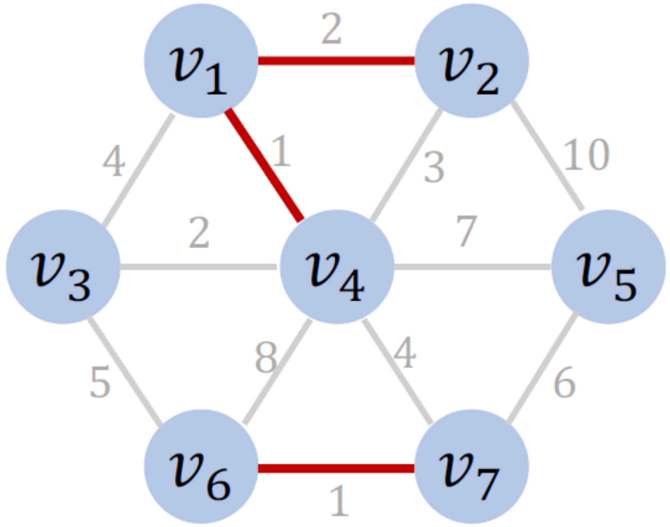


$\mathcal{T} = \{e_{1,4}, e_{6,7}\}$

Edge	Weight
$(6,7)$	1
$(1,2)$	2
$(3,4)$	2
$(2,4)$	3
$(1,3)$	4
$(4,7)$	4
$(3,6)$	5
$(5,7)$	6
$(4,5)$	7
$(4,6)$	8
$(2,5)$	10

Iteration 3

- Dequeue and get the edge (1, 2)
- v_1 and v_2 are not in the same tree
- Thus **accept** edge (1, 2)
- Append (1, 2) to $\mathcal{T}$

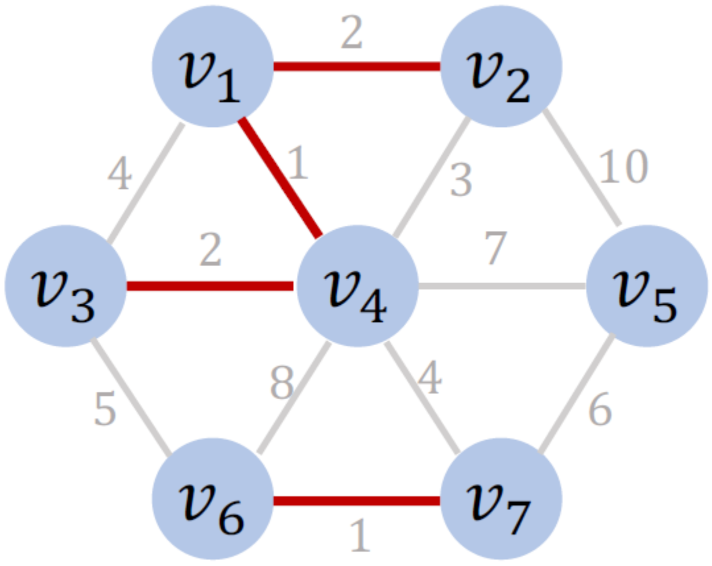


$$\mathcal{T} = \{e_{1,4}, e_{6,7}, e_{1,2}\}$$

Edge	Weight
$(1,2)$	2
$(3,4)$	2
$(2,4)$	3
$(1,3)$	4
$(4,7)$	4
$(3,6)$	5
$(5,7)$	6
$(4,5)$	7
$(4,6)$	8
$(2,5)$	10

Iteration 4

- Dequeue and get the edge (3, 4)
- v_3 and v_4 are not in the same tree
- Thus **accept** edge (3, 4)
- Append (3, 4) to $\mathcal{T}$

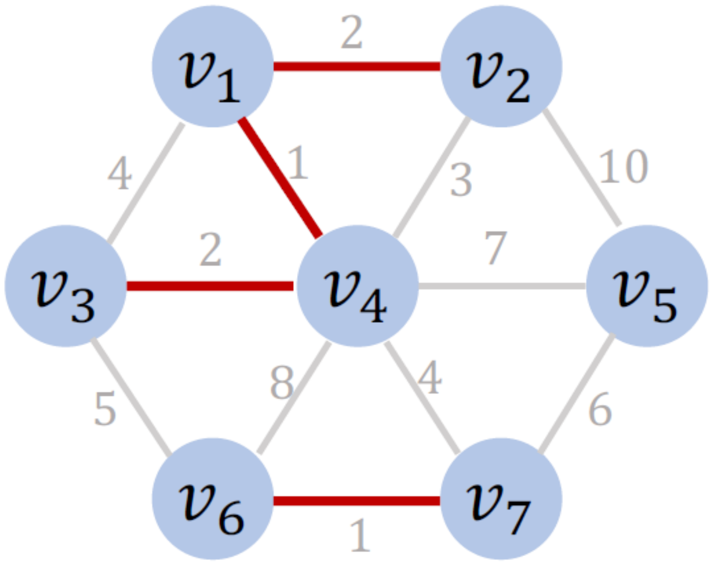


$$\mathcal{T} = \{e_{1,4}, e_{6,7}, e_{1,2}, e_{3,4}\}$$

Edge	Weight
$(3,4)$	2
$(2,4)$	3
$(1,3)$	4
$(4,7)$	4
$(3,6)$	5
$(5,7)$	6
$(4,5)$	7
$(4,6)$	8
$(2,5)$	10

Iteration 5

- Dequeue and get the edge (2, 4)
- v_2 and v_4 are in the same tree
- Thus **reject** edge (2, 4)

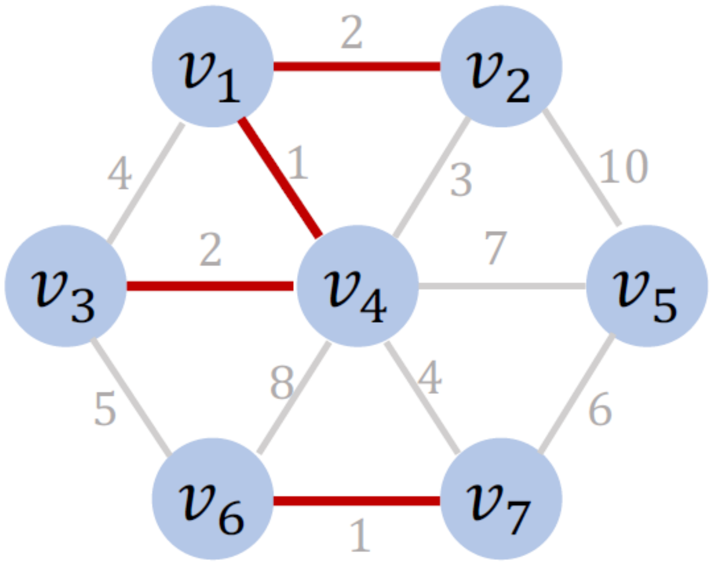


$\mathcal{T} = \{e_{1,4}, e_{6,7}, e_{1,2}, e_{3,4}\}$

Edge	Weight
$(2,4)$	3
$(1,3)$	4
$(4,7)$	4
$(3,6)$	5
$(5,7)$	6
$(4,5)$	7
$(4,6)$	8
$(2,5)$	10

Iteration 6

- Dequeue and get the edge (1, 3)
- v_1 and v_3 are in the same tree
- Thus **reject** edge (1, 3)

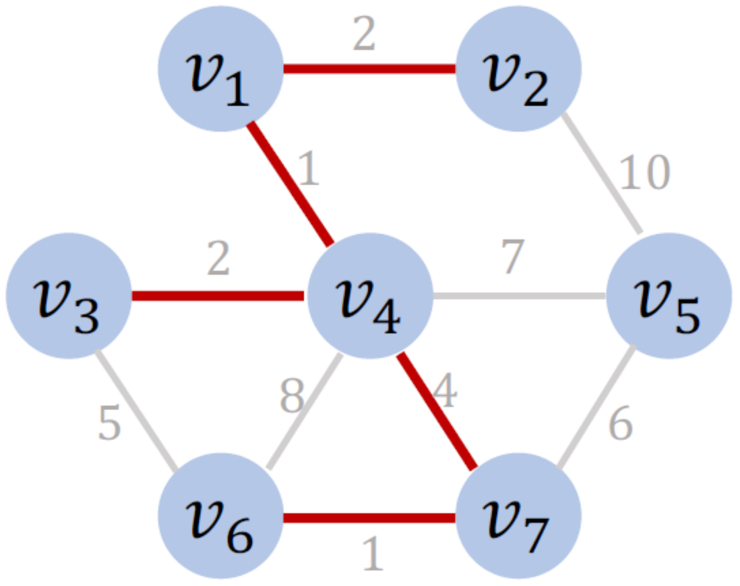


$\mathcal{T} = \{e_{1,4}, e_{6,7}, e_{1,2}, e_{3,4}\}$

Edge	Weight
$(1,3)$	4
$(4,7)$	4
$(3,6)$	5
$(5,7)$	6
$(4,5)$	7
$(4,6)$	8
$(2,5)$	10

Iteration 7

- Dequeue and get the edge (4, 7)
- v_4 and v_7 are not in the same tree
- Thus accept edge (4, 7)
- Append (4, 7) to $\mathcal{T}$

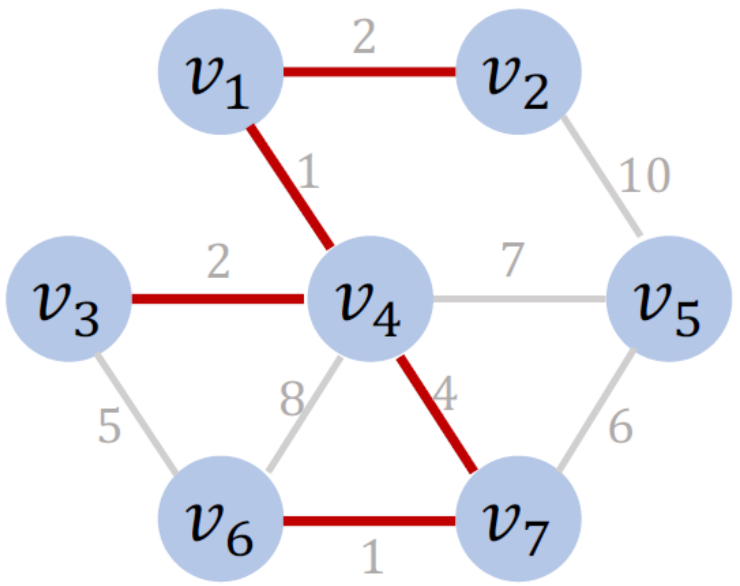


$$\mathcal{T} = \{e_{1,4}, e_{6,7}, e_{1,2}, e_{3,4}, e_{4,7}\}$$

Edge	Weight
$(4,7)$	4
$(3,6)$	5
$(5,7)$	6
$(4,5)$	7
$(4,6)$	8
$(2,5)$	10

Iteration 8

- Dequeue and get the edge (3, 6)
- v_3 and v_6 are in the same tree
- Thus **reject** edge (3, 6)

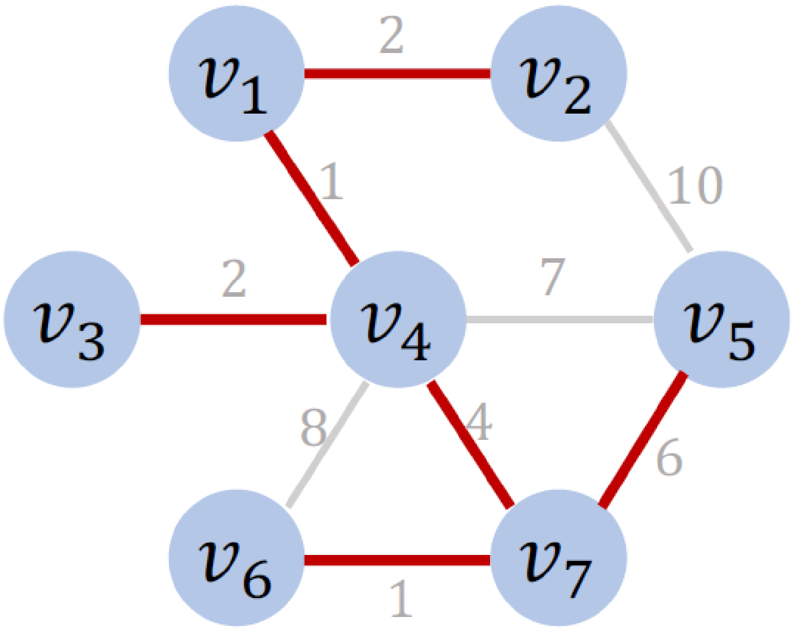


Edge	Weight
$(3,6)$	5
$(5,7)$	6
$(4,5)$	7
$(4,6)$	8
$(2,5)$	10

$$\mathcal{T} = \{e_{1,4}, e_{6,7}, e_{1,2}, e_{3,4}, e_{4,7}\}$$

Iteration 9

- Dequeue and get the edge (5, 7)
- v_5 and v_7 are not in the same tree
- Thus **accept** edge (5, 7)
- Append (5, 7) to $\mathcal{T}$



Edge	Weight
$(5, 7)$	6
$(4, 5)$	7
$(4, 6)$	8
$(2, 5)$	10

- After this
  - All the vertices are connected
  - Return the edges $\mathcal{T}$

$$\mathcal{T} = \{e_{1,4}, e_{6,7}, e_{1,2}, e_{3,4}, e_{4,7}, e_{5,7}\}$$

Kruskal's Algorithm

1. Put all the edges of the input graph into a queue
2. Sort the queue so that the weights are in ascending order
3. Let set $\mathcal{T}$ (which stores the selected edges) be the empty set
4. While $\mathcal{T}$ has fewer than n–1 edges

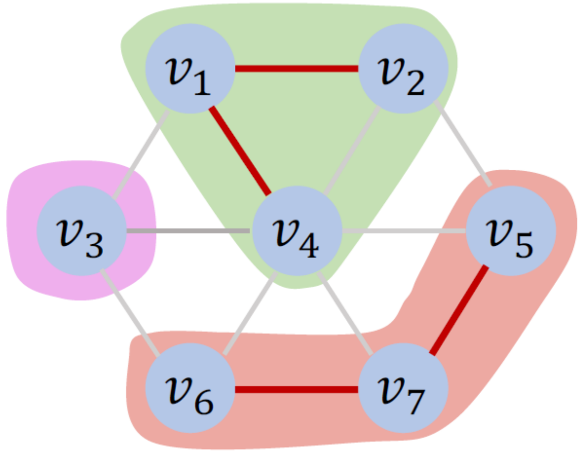
   Get an edge: $e_{u,v}$ ← dequeue( )

   If u and v are in different trees, then add $e_{u,v}$ to $\mathcal{T}$ and merge the two trees.
5. Return $\mathcal{T}$

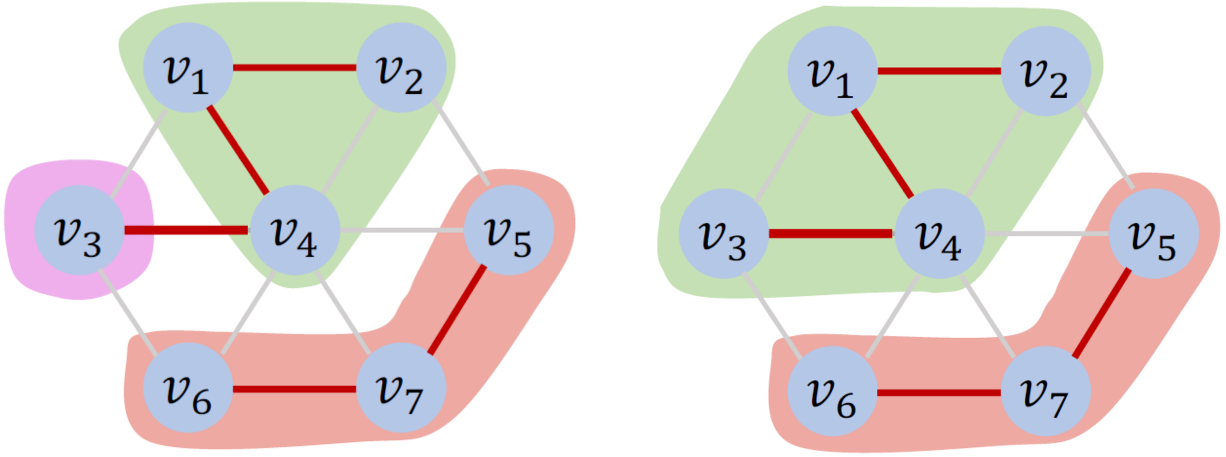
Using Disjoint Sets Data Structure

- How to decide whether two vertices are in the same tree?
- Solution: Using disjoint sets data structure. Put vertices of a tree in the same set. Deciding whether two vertices belong to the same set costs near $O(1)$ time. **Find()**
- How to merge two trees? **Union()**

Find()

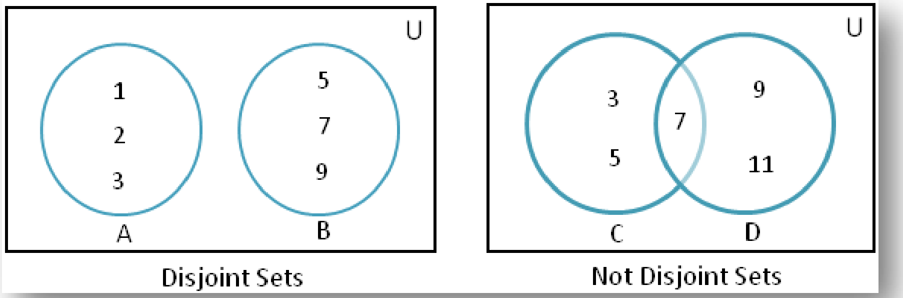


Union()



Disjoint Subsets

- Definition: A∩B = ∅ -> A and B are disjoint subsets.
- Example: A = {1,2,3} and B = {4,5}; share no same item -> disjoint.
- Object set: 0, 1, 2, 3, 4, 5
- Disjoint subset: {0} {1} { 2 3 4 } { 5 6 }
  - Set A: { 0 }
  - Set B: { 1 }
  - Set C: { 2, 3, 4 }
  - Set D: { 5, 6 }



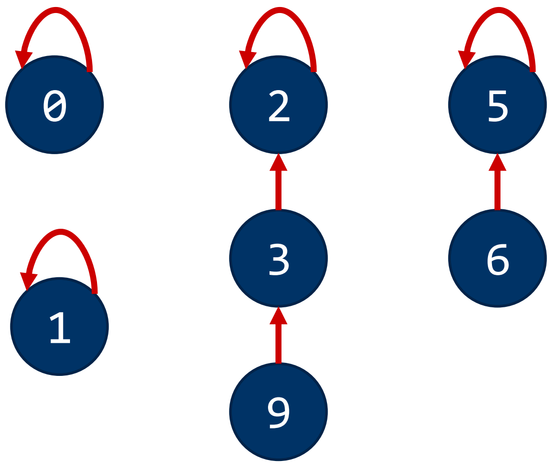
- Common operations:
  - Find: find(item) -> Set ID, i.e., find(3) -> C.
  - Union: union(A, B) -> new subset A∪B = { 0, 1 }.
- How to support fast find and union?
- Idea: use a tree to represent each subset.

https://www.math-only-math.com/images/xdisjoint-of-sets-using-Venn-diagram.png.pagespeed.ic.vAR3eqLUT-.png

Disjoint Subsets – Trees

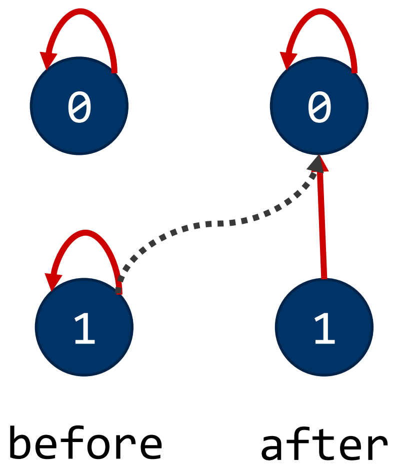
SINGAPORE INSTITUTE OF TECHNOLOGY

- Disjoint subset: {0} {1} { 2 3 9 } { 5 6 }
- Common operations:
  - Find: find(item) -> Set ID, i.e., find(3) -> C.
  - Union: union(A, B) -> new subset A∪B = { 0, 1 }.
- Tree id: the root item.
  - Each subset has one item as id. Unique?
- Data struct for each item: [Item id, Parent item id]
  - Root item: parent item is itself.
- Set 0: [0, 0] -> only one item in the subset.
- Set 1: [1, 1] -> only one item in the subset.
- Set 2: a) [9, 3]; b) [3, 2]; c) [2, 2]. Who is the root?
- Set 3: a) [6, 5]; b) [5, 5].



Disjoint Subsets – Find and Union

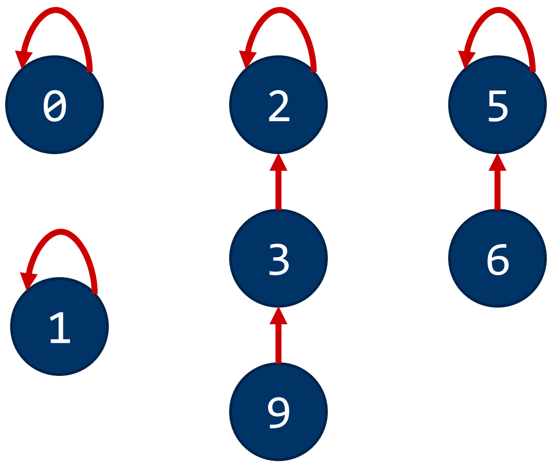
- Many ways for the find and union operations.
- Let us start with some intuitive ones.
- With the trees, what is the time complexity of find(·)?
  - Worst case, O(V) items in a tree with one node per level.
  - Finding the leaf item, so take O(V) checks to the root.
- What is the time complexity of union(·,·).
  - Relink both trees. O(1).
  - i.e., union(Set 0, Set 1).
  - Item 1's pointer to item 0.
- Fine for union(), but not for find().
- How to make find() faster?
  - A lot options. We cover a few here.



https://www.math-only-math.com/images/xdisjoint-of-sets-using-Venn-diagram.png.pagespeed.ic.vAR3eqLUT-.png

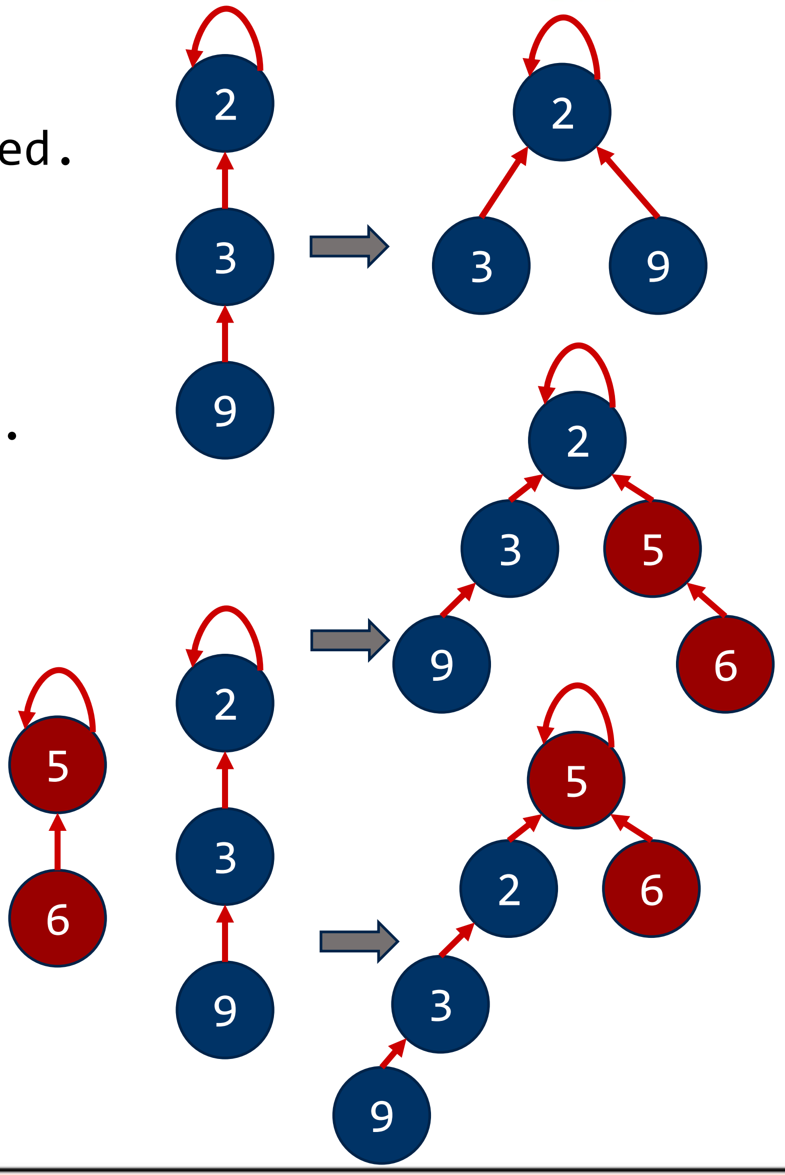
Disjoint Subsets – Faster Find

- Data struct for each item: [Item id, Parent item id, **root id**].
  - Allocate some space to record the root id for each item in the tree.
  - Object set: 0, 1, 2, 3, 5, 6, 9
  - Root id: 0, 1, 2, 2, 5, 5, 2
- Here, find() can be in O(1) time.
- But how about union? Still O(1)?
  - Merge two trees together with relinking.
  - Update all root id. O(V) operation for large trees.
- Now we presented two options:
  - Quick union O(1) with slow find O(V).
  - Quick find O(1) with slow union O(V).
- Question: why do we have a tree looks like a list?
  - For list of length O(V), O(V) time traverse.
  - Binary tree, can be O(logV) levels, much faster. But how?

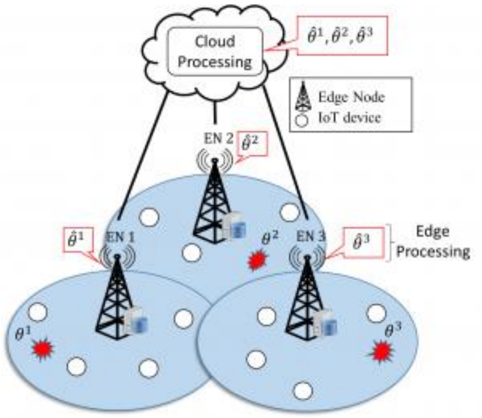


Disjoint Subsets – Maintain a Balanced Tree

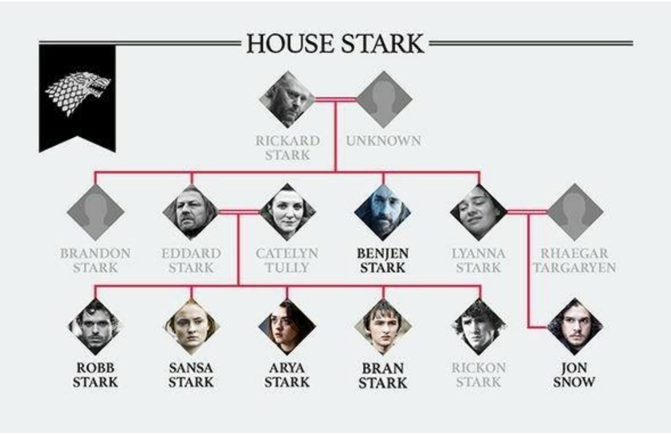
- Initially, **build a flat tree**.
- Throughout the further unions, **keep flat**, or balanced.
- Check our example to merge {5, 6} and {2, 3, 9}.
  - One option gives us flat tree.
  - The other option makes the tree very imbalanced.
- Intuitively, root shall come from the larger subset.
  - i.e., 2 shall be the root compared to 5.
- For implementation, how to do this?
  - How can we say which tree is larger or smaller?
- Choice 1: record tree size in root.
  - Tree size: the number of items in the tree.
  - Root from the tree with more items.
  - Smaller tree becomes a subtree.
  - Update tree size, i.e., int A + int B.
  - Time for find() being pushed to O(log|V|).



Disjoint Subsets – Applications



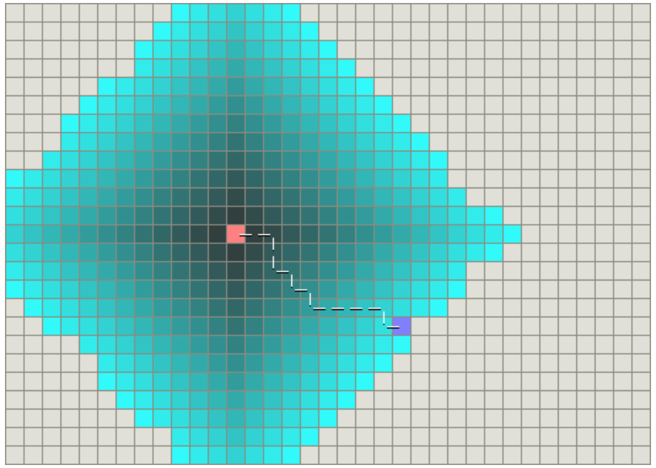
- Choice 2: record tree depth, or rank, in root.
  - Shorter tree becomes a subtree.
  - Key idea is still to balance the tree.
  - Time for find() being pushed to O(log|V|).
- More advanced options available.
  - i.e., path compression, path halving, path splitting.
  - Different options, or the combination of options, give different complexity.
  - Read references if interested.
- Applications:
  - Network connectivity.
    - Local networks -> global ne
  - Image segmentation.
    - From one pixel to one objec
  - Least common ancestor.



https://blogs.kcl.ac.uk/kclip/files/2020/06/drawing-300x258.png https://i.pinimg.com/originals/38/46/f2/3846f296fe3c2a37d77a8f33eedace34.jpg
https://www.ias.informatik.tu-darmstadt.de/uploads/Theses/OpenTopics/segmentation.jpg

A* Search algorithm

- DFS, BFS, Dijkstra…
  - **Expand “blindly”** by exploring nodes without any “intelligence” guidance towards the goal
  - **Consider all directions as equally likely** to lead to the goal or follow a **fixed order** of exploration.
  - Do not have a "sense" of direction toward the goal, meaning they operate without any preference for paths that might be closer to the target
  - Time consuming if the graph is big even with Bidirectional or Iterative Deepening improvements



- Other Improvements
  - Heuristic search: rank the directions/options based on importance (priority)
  - Only expand limited options, consider more options if cannot find – Iterative Widening
  - Pruning, stop expanding for options that unlikely/unfeasible to reach the goal

Heuristic Search

- Key: Heuristic Function
- The heuristic function assigns a priority to each direction or option based on the current state, guiding the search towards the goal.
- A more accurate heuristic function leads to faster convergence on the optimal solution by reducing unnecessary exploration.
- No one-size-fits-all heuristic; need to tailored to the specific problem.
- The heuristic should be computationally efficient, as a costly heuristic can negatively impact search performance, e.g. Manhattan distance,

$$h(s) = |x_s - x_g| + |y_s - y_g|$$

```
bfs(s) {
  q = new queue()
  q.push(s), visited[s] = true
  while (!q.empty()) {
    u = q.pop() - check if meet the goal
    for each edge(u, v) {
      if (!visited[v]) {
        q.push(v)
        visited[v] = true
      }
    }
  }
}
```

```
greedyBestFirstSearch(s) {
  q = new priority_queue()
  q.push(s, h(s)), visited[s] = true
  while (!q.empty()) {
    u = q.pop() - check if meet the goal
    for each edge(u, v) {
      if (!visited[v]) {
        q.push(v, h(s))
        visited[v] = true
      }
    }
  }
}
```

A* Search algorithm

- Cost Function:

  $$f(x) = g(x) + h(x)$$

  - g(x): actual moving cost from the start node to the current node x
  - h(x): heuristic function, representing the estimated cost from the current node x to the goal
  - The **priority queue** returns the node x with the minimum f(x), prioritizing nodes that appear closer to the goal based on f(x)

```
function AStar(Graph, start, goal):
    create vertex priority queue Q
    distTo[start] ← 0                              // g(s) = 0 for the start node
    Q.add_with_priority(start, h(start))            // priority = h(start)
    for each vertex v in Graph.Vertices:
        if v ≠ start
            prev[v] ← UNDEFINED
            distTo[v] ← INFINITY                    // Set g(v) = ∞ initially
            Q.add_with_priority(v, INFINITY)

    while Q is not empty:
        u ← Q.extract_min()                         // Node with lowest f(u) = g(u) + h(u)
        if u == goal:                               // Goal reached
            return distTo, edgeTo                   // Shortest path found

        for each neighbor v of u:
            tentative_gScore ← distTo[u] + Graph.Edges(u, v)
            if tentative_gScore < distTo[v]:
                edgeTo[v] ← u                       // Track the path
                distTo[v] ← tentative_gScore        // Update g(v) = g(u) + cost(u, v)
                fScore ← distTo[v] + h(v)           // Calculate f(v) = g(v) + h(v)
                Q.decrease_priority(v, fScore)      // Update priority in Q with f(v)

    return 0                                        // If Q is empty and goal not found
```

IDA* (Iterative Deepening A*):

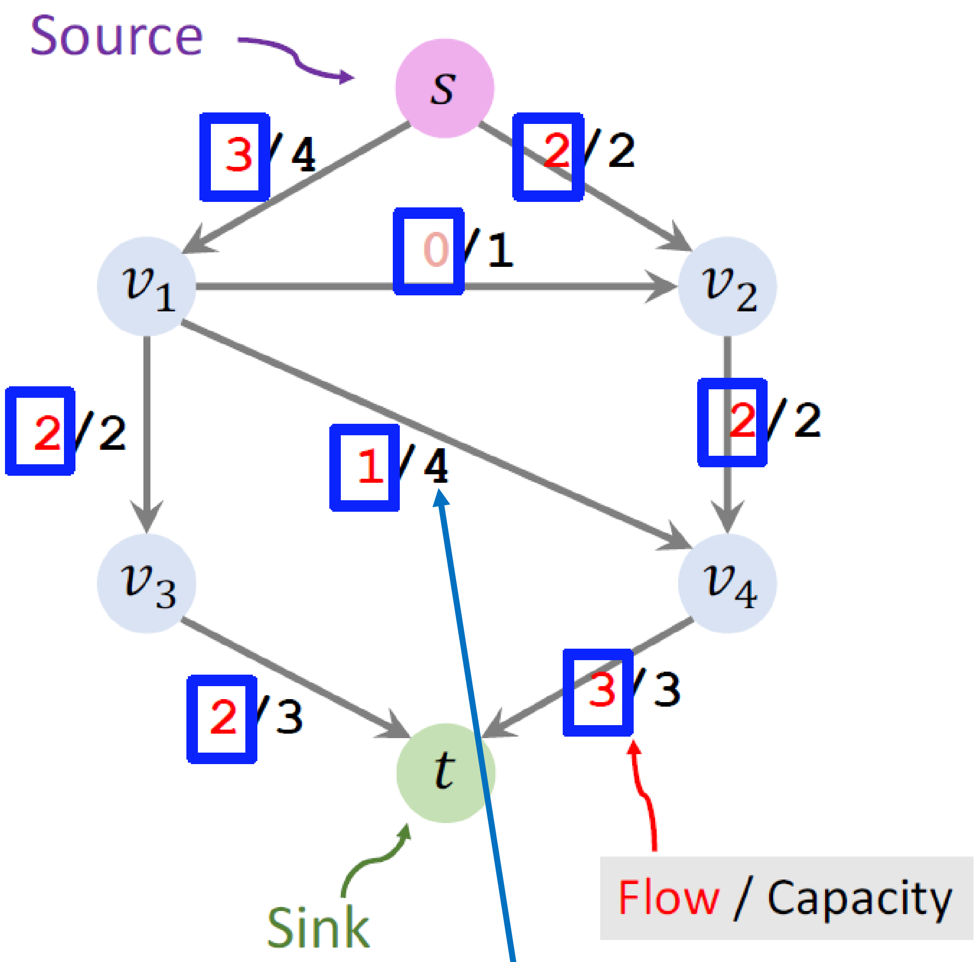
- **Challenge with A***: While effective, A* can consume large amounts of memory, storing all expanded nodes in the priority queue and open/closed lists
- **Solution with IDA***: Similar to **Iterative Deepening Depth-First Search (IDDFS)**, IDA* applies a **cost limit instead of a depth limit.** Each iteration explores paths within a given f(x) threshold (cost limit), gradually increasing the limit in subsequent iterations
- **Advantage:** IDA* significantly reduces memory usage compared to A*, as it doesn't need to store the entire search tree, making it suitable for memory-constrained environments

Max Flow Problem

- Send water from the source s to the sink t
- The edges are pipes which have certain capacities, e.g., $4m^3$/s
- How much water can flow from source s to the sink t at most?

- Inputs: A weighted directed graph, the source s, and the sink t
- Goal: Send as much water as possible from s to t
- Constraints:
  - Each edge has a weight (i.e., the capacity of the pipe)
  - The flow must not exceed the capacity

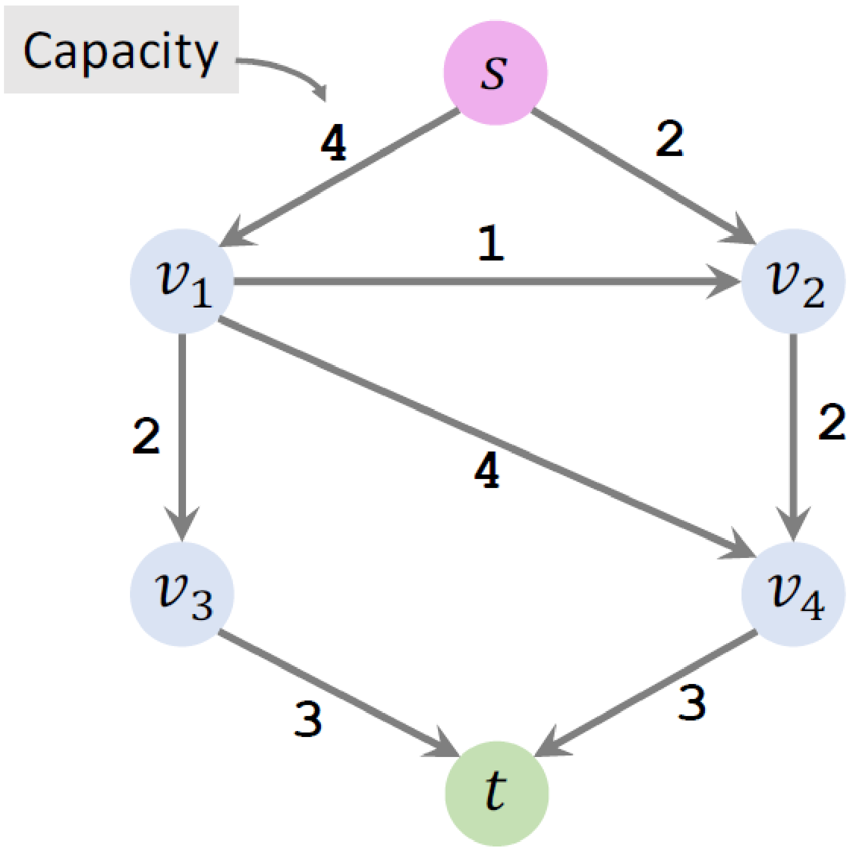
Max (Net) Flow = 5



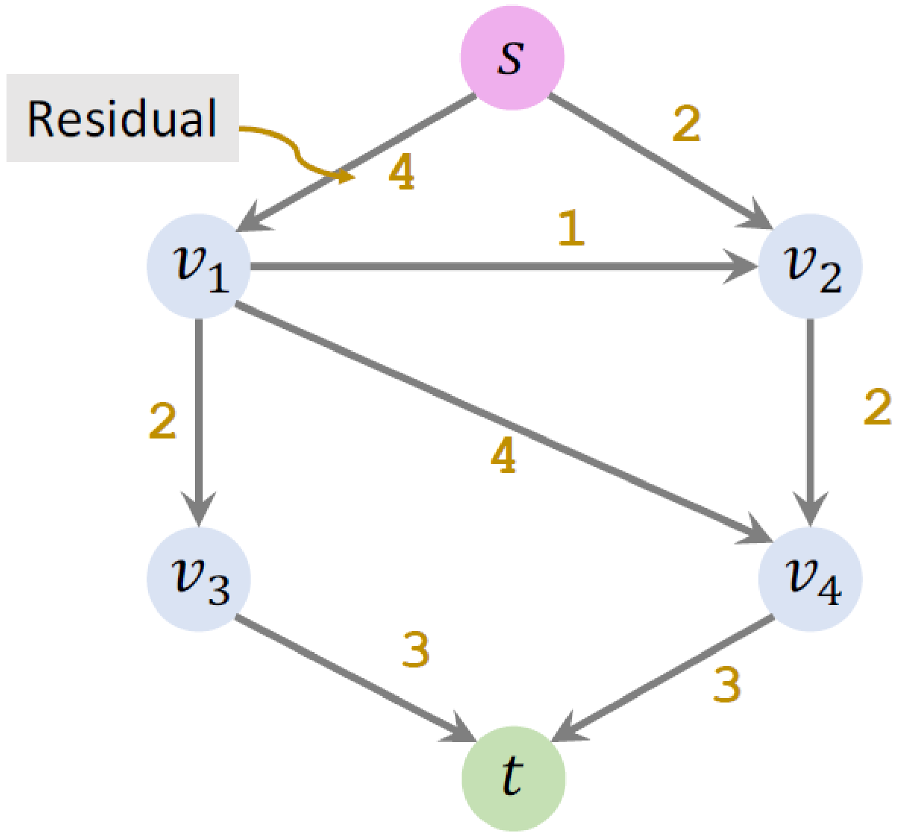
Capacity of the pipe is 4 m^3/s
A flow of 1 m^3/s goes through the pipe
It has a residual of 3 m^3/s

Initialization

- Augmenting path: a path from s to t that does not contain cycles



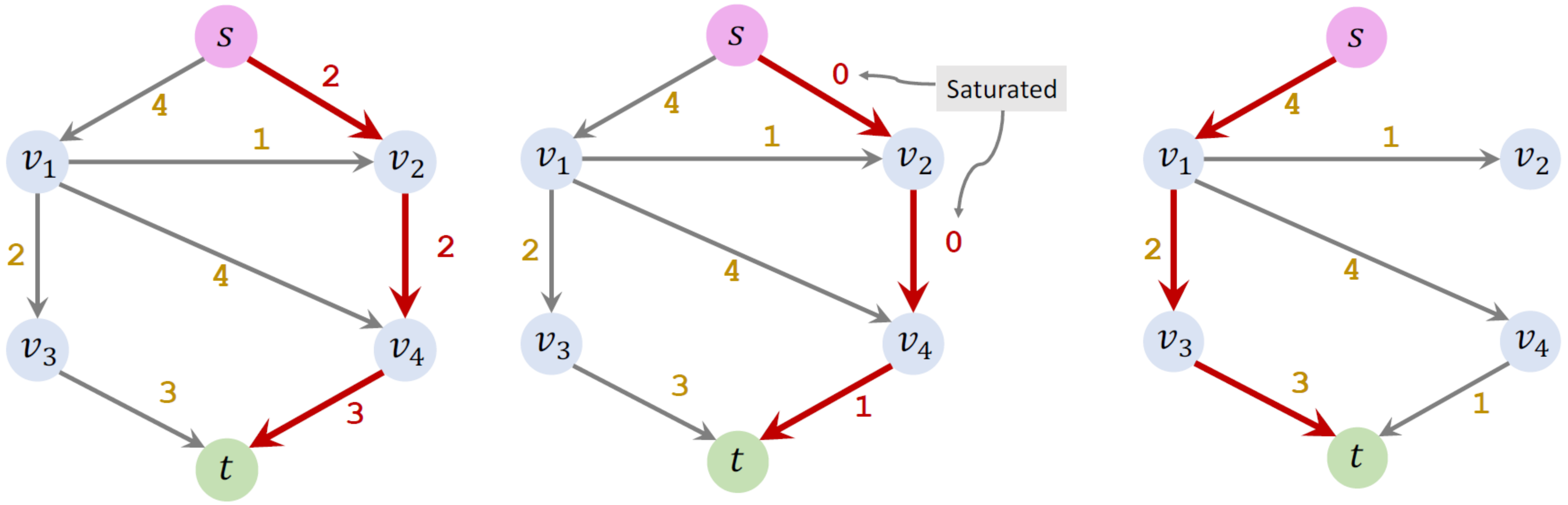
Original Graph



Residual Graph

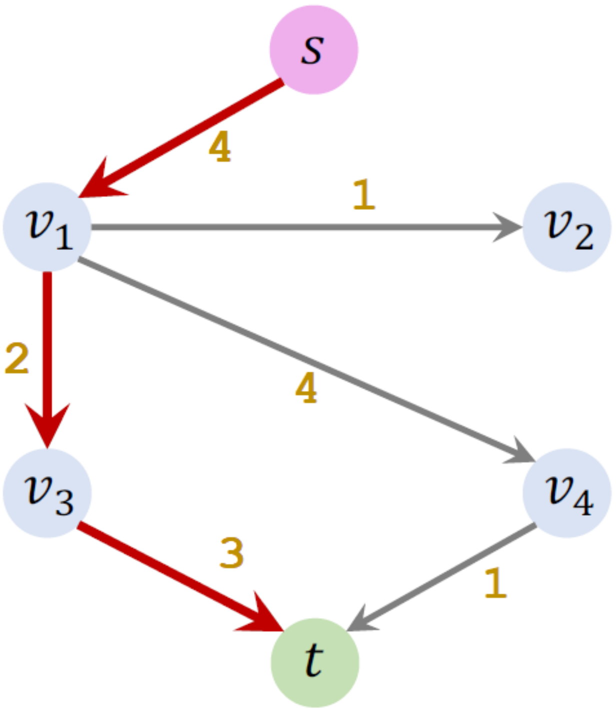
Iteration 1: find an augmenting path and update residuals

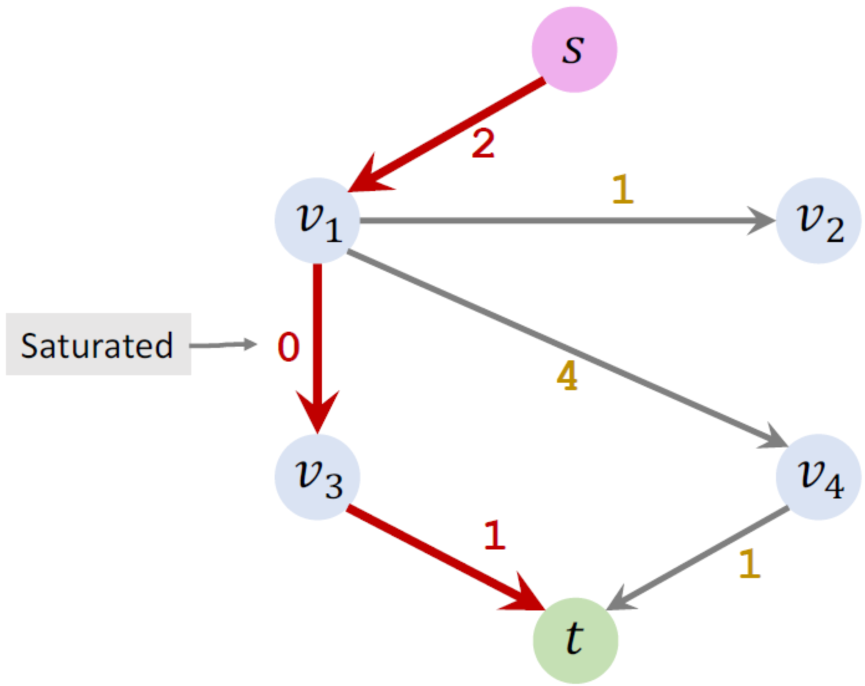
- Working on the **residual graph**
- Augmenting path: a path from s to t that does not contain cycles
- Found path $s \to v_2 \to v_4 \to t$ (Bottleneck capacity = **2**)
- Update residuals: -2 for the edges, remove saturated edges



Iteration 2: find an augmenting path and update residuals

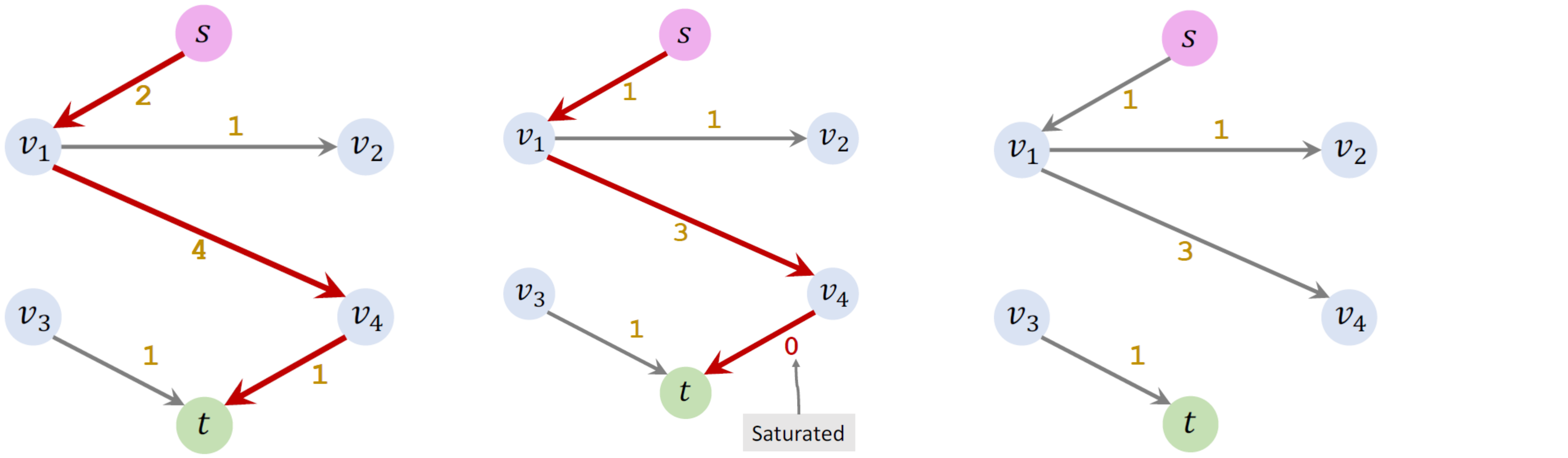
- Working on the **residual graph**
- Augmenting path: a path from s to t that does not contain cycles
- Found path $s \rightarrow v_1 \rightarrow v_3 \rightarrow t$ (Bottleneck capacity = **2**)
- Update residuals: -2 for the edges, remove saturated edges





Iteration 2: find an augmenting path and update residuals

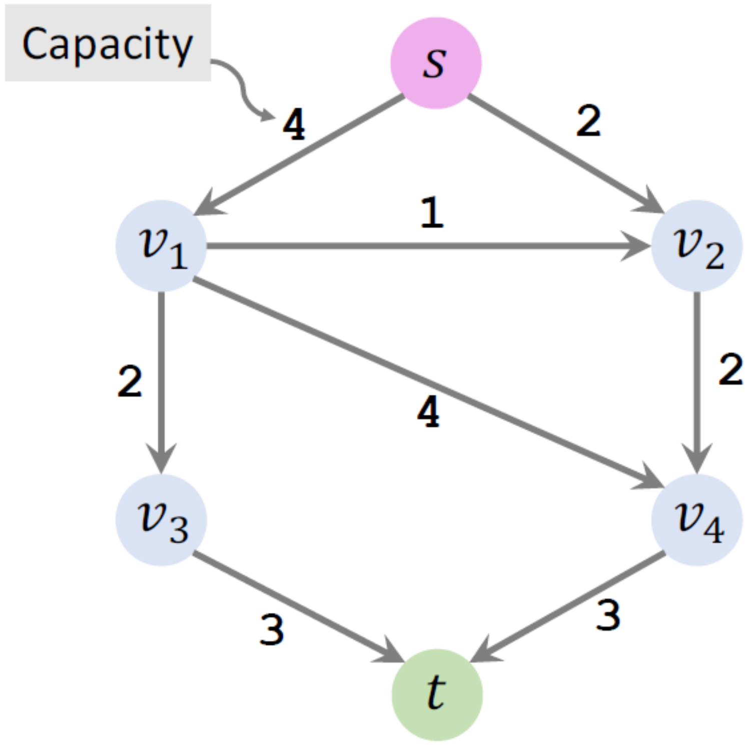
- Working on the **residual graph**
- Augmenting path: a path from s to t that does not contain cycles
- Found path $s \rightarrow v_1 \rightarrow v_4 \rightarrow t$)Bottleneck capacity = 1)
- Update residuals: -1 for the edges, remove saturated edges

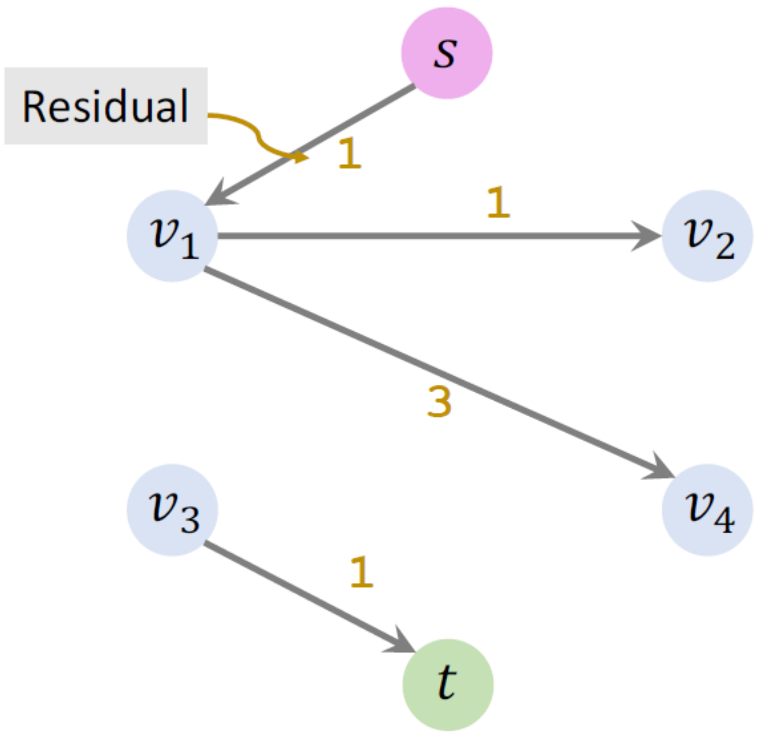


Cannot find any path from source to sink

Flow = Capacity - Residual

- Amount of flow: 5, outward flow of s == inward flow of t == 5





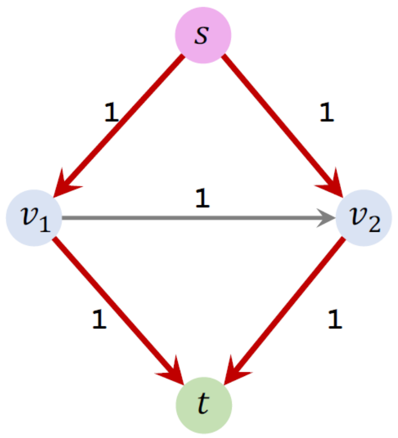
$$\text{Flow} = \text{Capacity} - \text{Residual}.$$

Simple solution

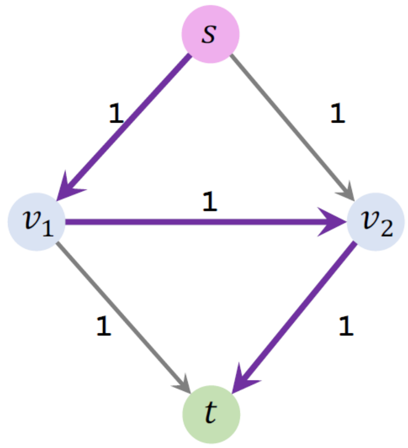
1. Build a residual graph; initialize the residuals to the capacity
2. While augmenting path can be found:
    1. Find an augmenting path (on the residual graph)
    2. Find the bottleneck capacity x in the augmenting path
    3. Update the residuals (residual ← residual – x)
3. **Flow = Capacity - Residual**

This simple solution may fail

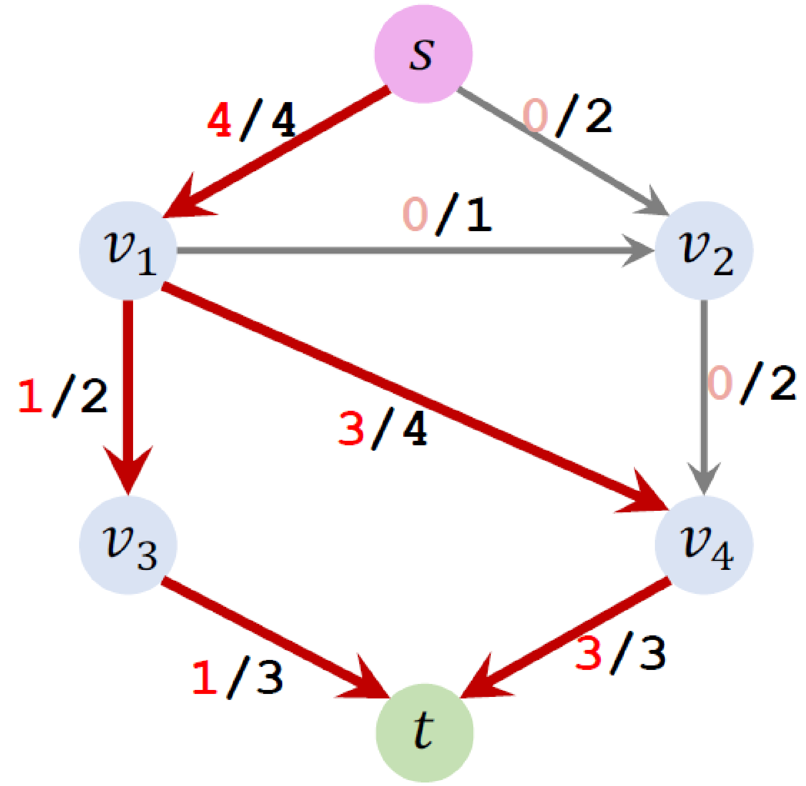
- Always finds the blocking flow, may not be the maximum flow
- A flow is blocking flow if no more flow from source to sink can be found
- The “pipes” are blocked
- Maximum flow is also blocking flow
- Once a bad path is selected, the simple solution cannot make corrections



Maximum Flow



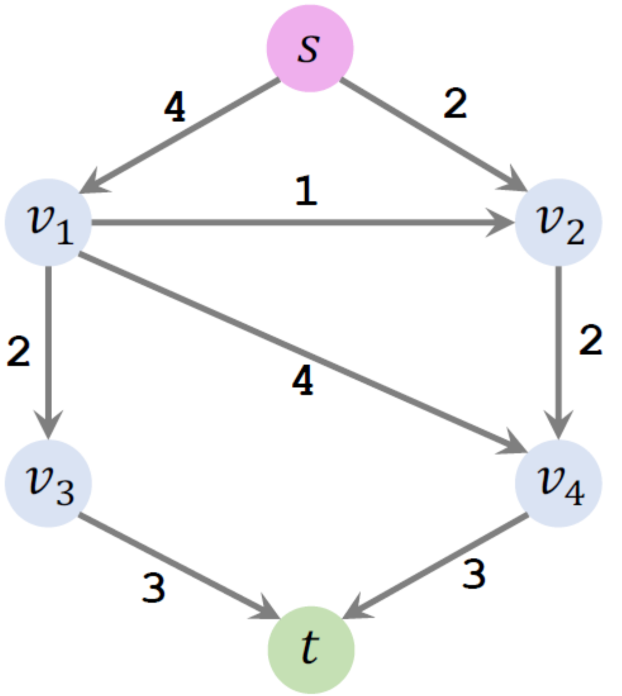
Not Maximum Flow



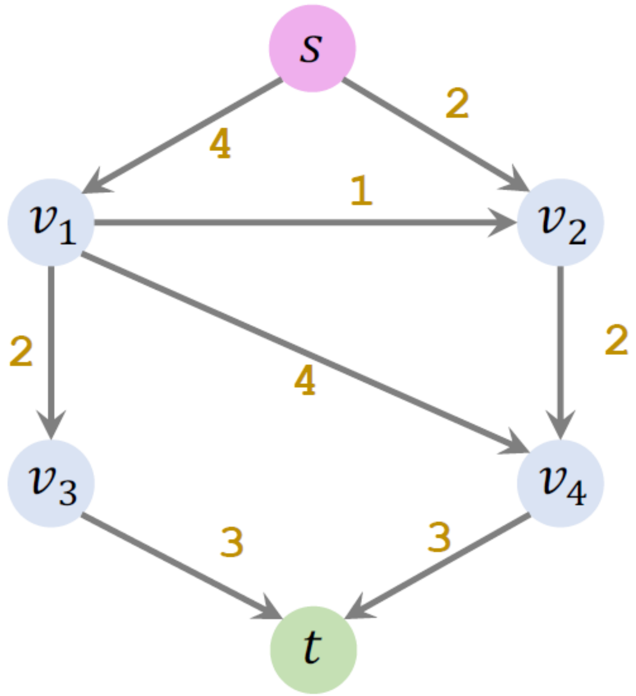
Amount of flow: 4

Ford-Fulkerson Algorithm

- Key idea: allow correction by adding backward path



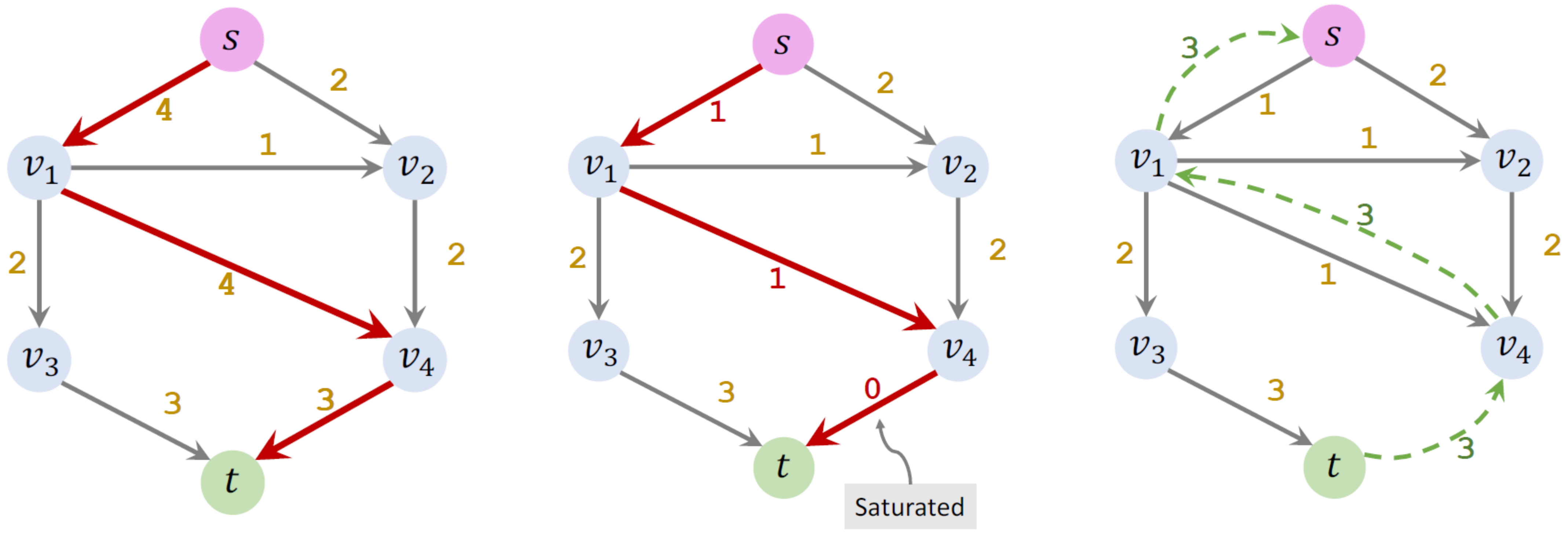
Original Graph



Residual Graph

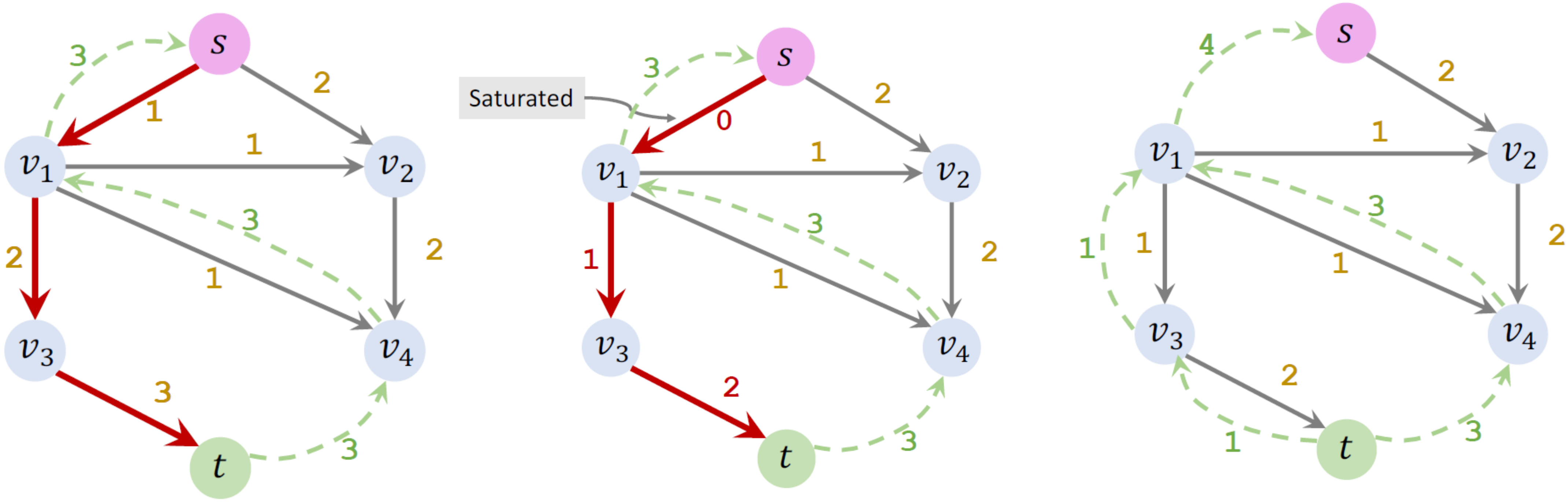
Iteration 1

- Working on the **residual graph**
- Found augmenting path $s \rightarrow v_1 \rightarrow v_4 \rightarrow t$ (Bottleneck capacity = 3)
- Update residuals: -3 for the edges, remove saturated edges
- **Add a backward path** $t \rightarrow v_4 \rightarrow v_1 \rightarrow s$ with weight= 3



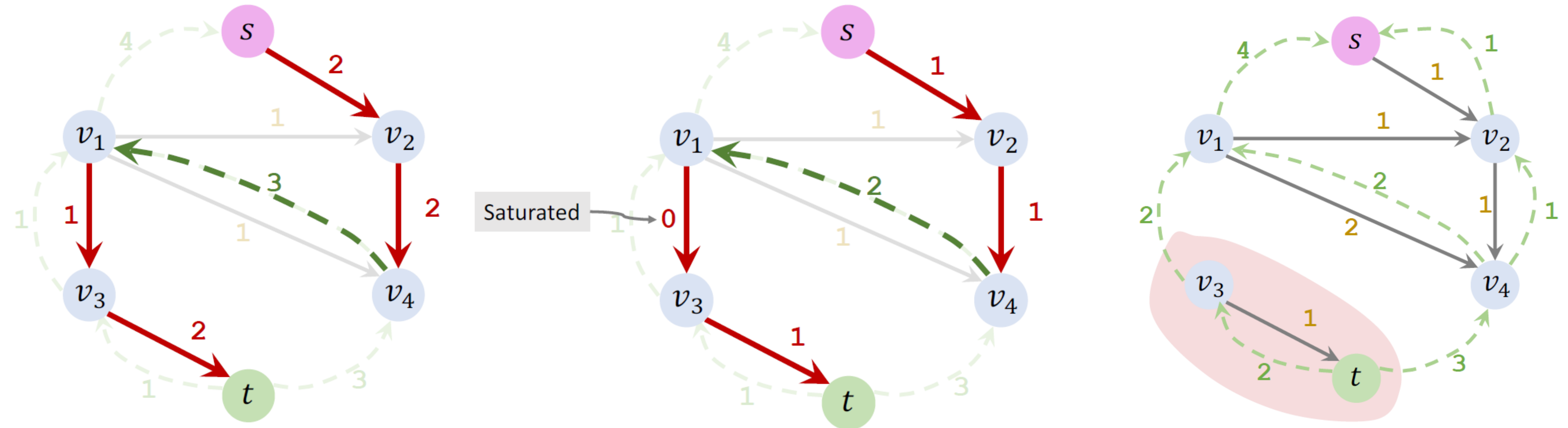
Iteration 2

- Working on the **residual graph**
- Found augmenting path $s \rightarrow v_1 \rightarrow v_3 \rightarrow t$ (Bottleneck capacity = 1)
- Update residuals: -1 for the edges, remove saturated edges
- **Add a backward path** $t \rightarrow v_3 \rightarrow v_1 \rightarrow s$ with weight= 1



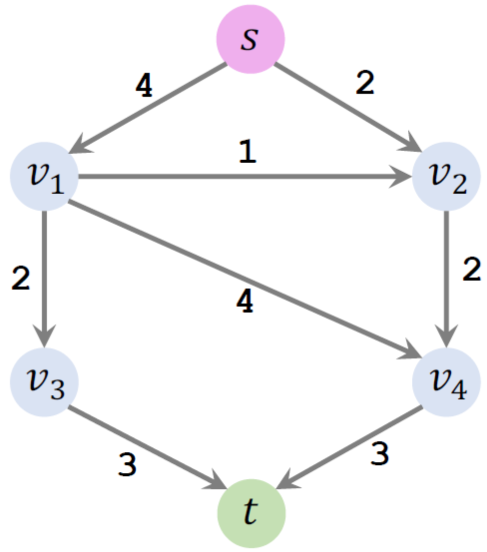
Iteration 3

- Working on the **residual graph**
- Found augmenting path $s \rightarrow v_2 \rightarrow v_4 \rightarrow v_1 \rightarrow v_3 \rightarrow t$ (Bottleneck capacity = 1)
- Update residuals: -1 for the edges, remove saturated edges
- **Add a backward path** $t \rightarrow v_3 \rightarrow v_1 \rightarrow v_4 \rightarrow v_2 \rightarrow s$ with weight= 1
- Cannot find any path anymore

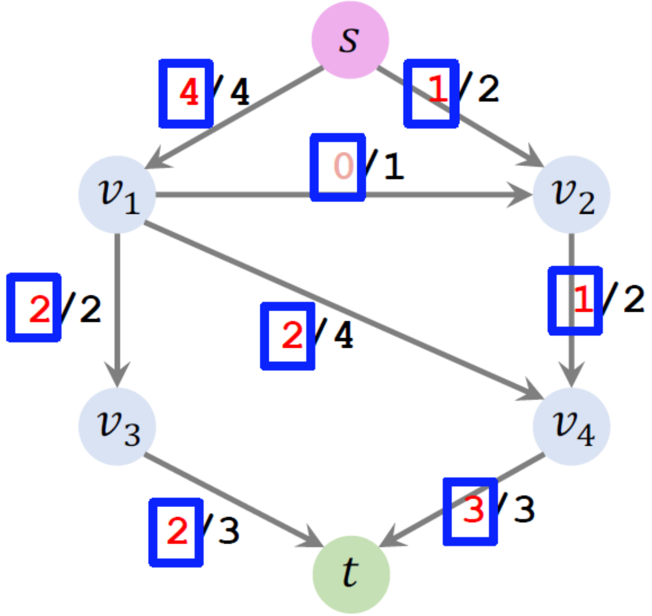


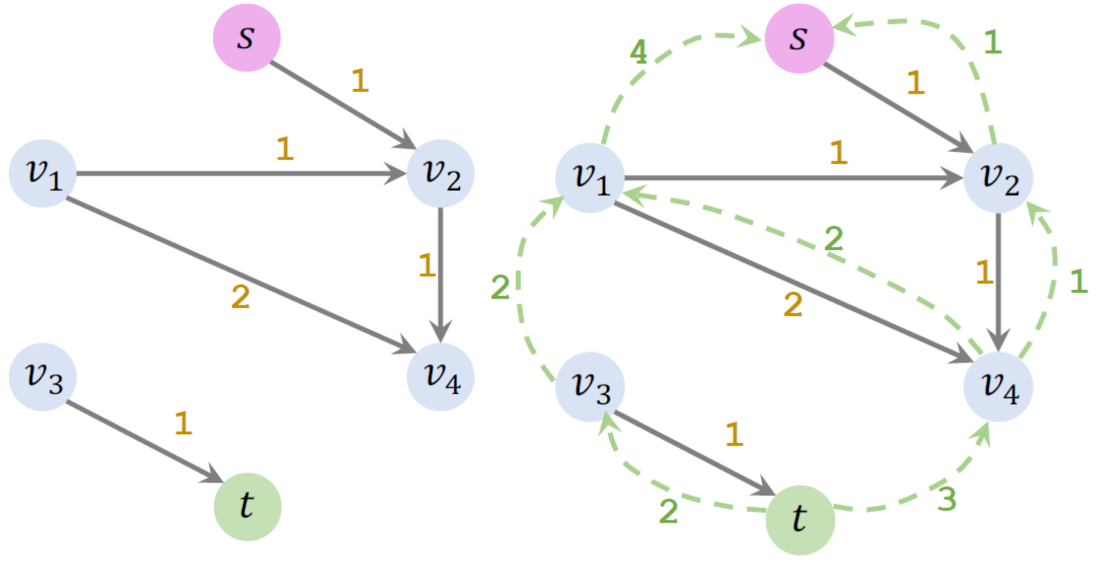
Flow = Capacity – Residual

- Max flow = 5



Original Graph

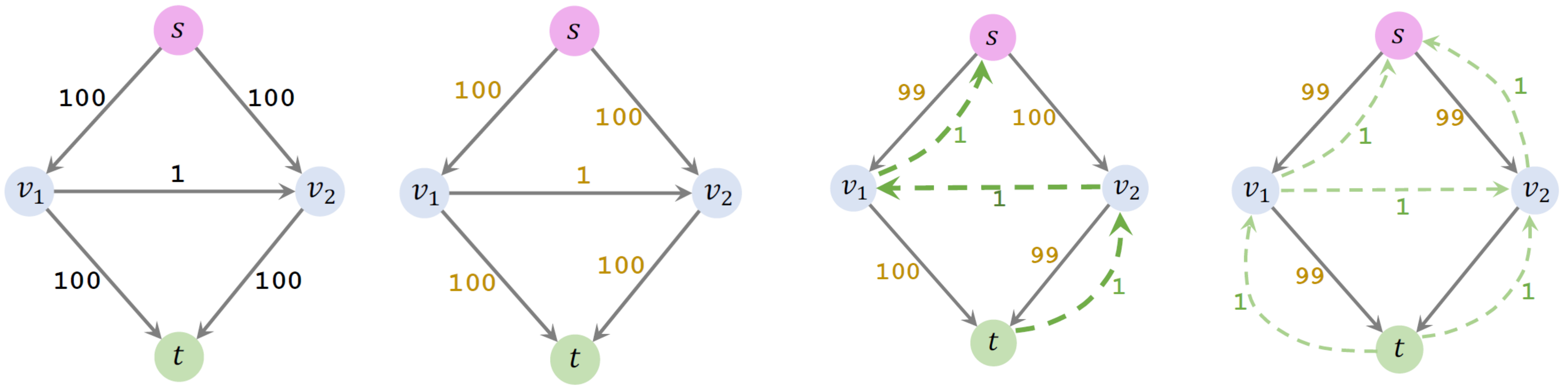




Residual Graph

Ford-Fulkerson bad case

- The amount of the max flow is 200
- Ford-Fulkerson algorithm may take 200 iterations to find the max flow
- In every iteration, the amount of flow increases by 1



Original Graph

Residual Graph

Iteration 1

Iteration 2

Time Complexity

- Each iteration increases the amount of flow by at least 1
- Thus, # Iterations ≤ Amount of Max Flow
- It takes O(m) time to find a path in unweighted graph (Ignore the weights in the residual graph) (m: # of edges, f: amount of max flow)
- Thus, the per-iteration time complexity is O(m)
- The worst-case time complexity is $O(fm)$

Edmonds–Karp Algorithm

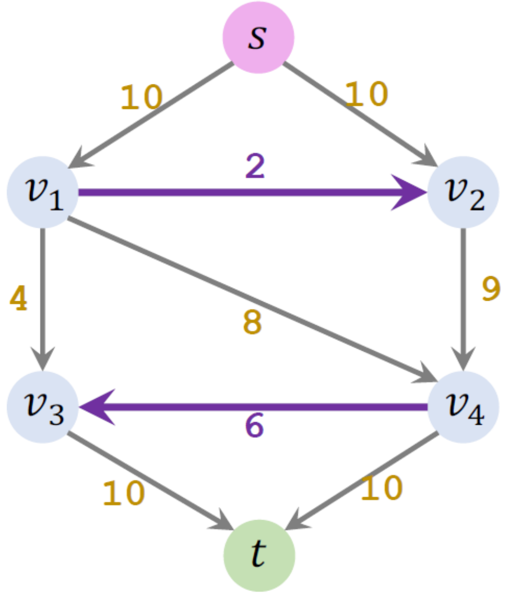
- Ford-Fulkerson:

1. Build a residual graph; initialize the residuals to the capacities
2. While augmenting path can be found:
    1. Find an augmenting path (on the residual graph)
       Improvement: Find the shortest augmenting path (on the residual graph)
    2. Find the bottleneck capacity x on the augmenting path
    3. Update the residuals (residual ← residual – x)
    4. Add a backward path (Along the path, all edges have weights of x)

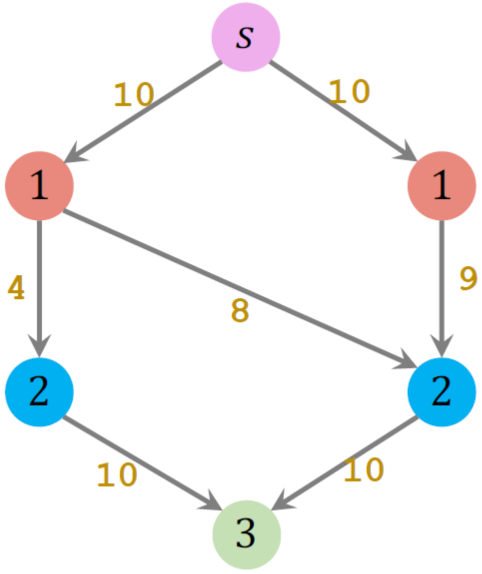
- Improvement is Edmonds–Karp
    - When finding path, regard the residual graph as unweighted
    - This can be found by a BFS, where we apply a weight of 1 to each edge
    - Time complexity: $O(m^2n)$ (m is #edges; n is #vertices)

Dinic’s Algorithm

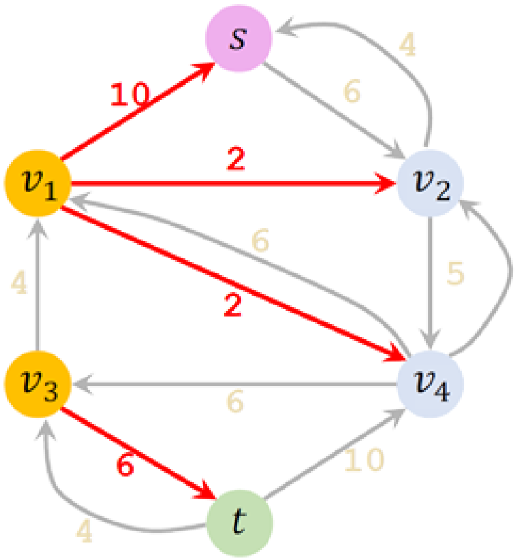
- Based on **level graph**: only edges connect to the next BFS level. Assign levels to all nodes, level of a node is shortest distance (in terms of number of edges) of the node from source.



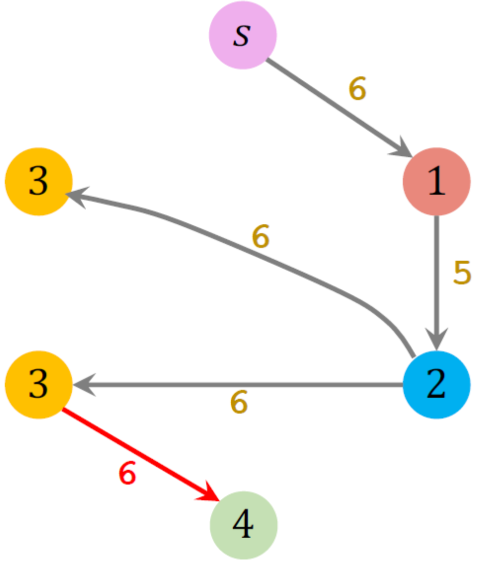
Original Graph



Level Graph



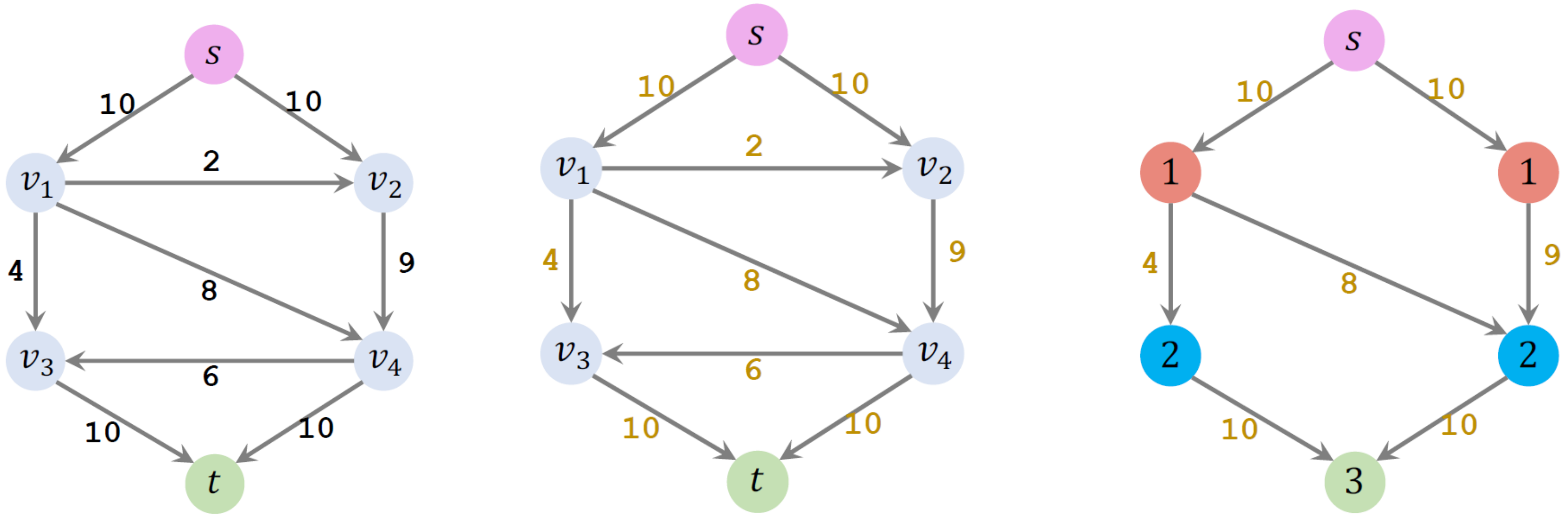
Original Graph



Level Graph

Dinic's Algorithm

- Construct level graph
- Find blocking flow in level graph
  - Blocking flow: if no more flow from source to sink can be found
  - Blocking flow can be found using the simple solution



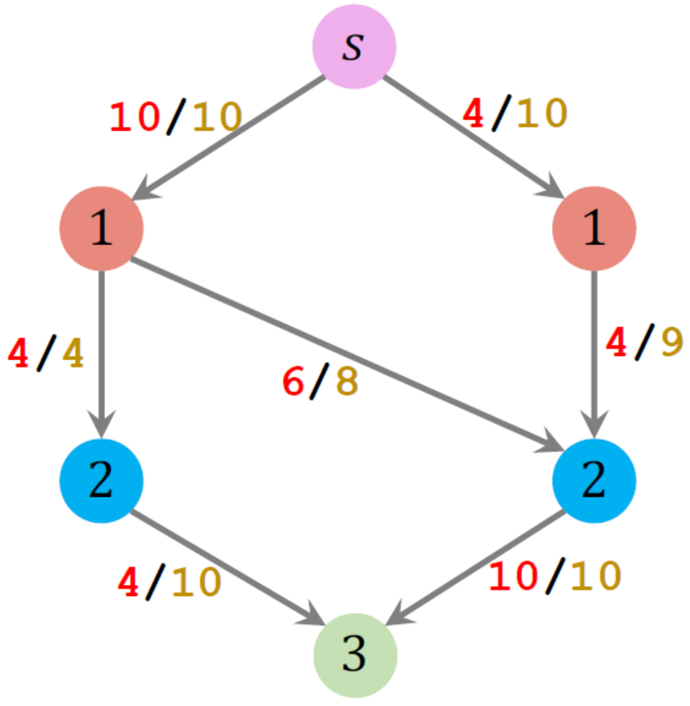
Original Graph

Residual Graph

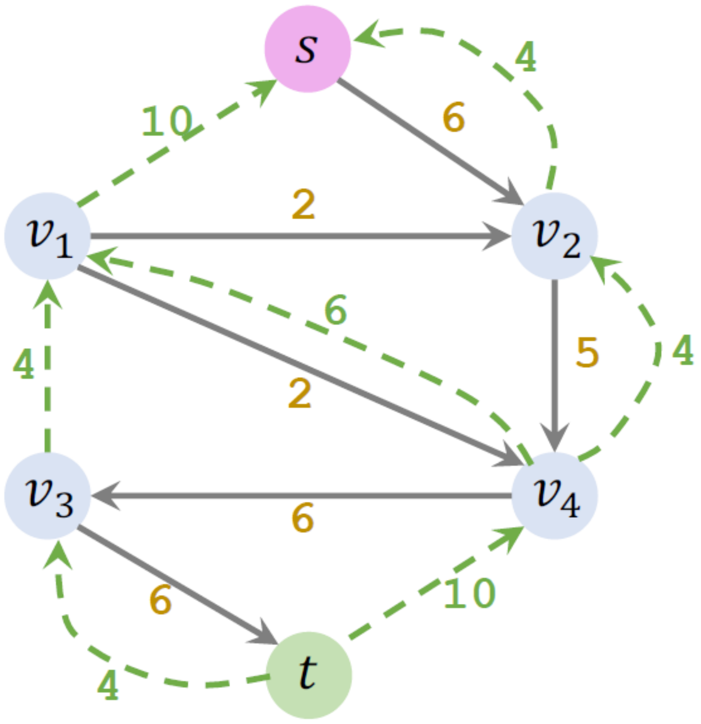
Level Graph

Iteration 1

- Construct level graph
- Find blocking flow in level graph
- Update the residual graph, remove saturated edges
- Add flows to the residual graph as backward paths



Level Graph

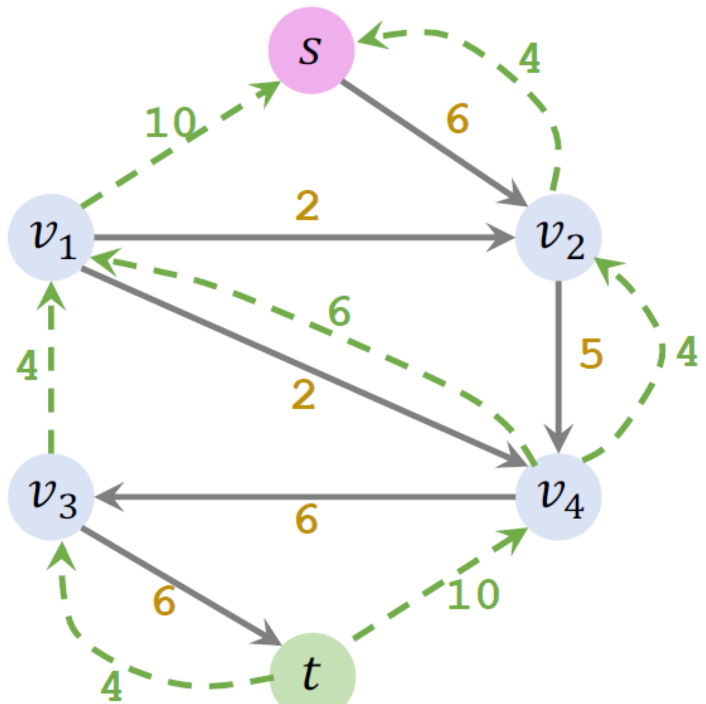


Residual Graph

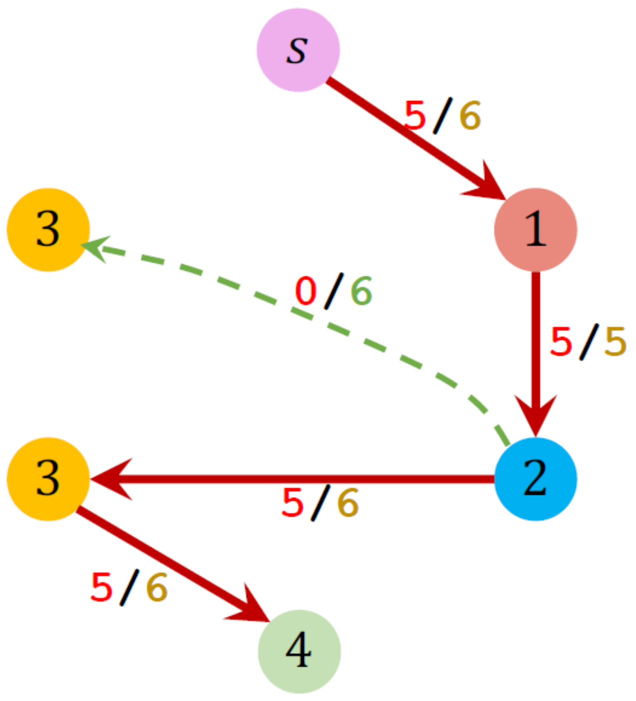
Iteration 2

SINGAPORE INSTITUTE OF TECHNOLOGY

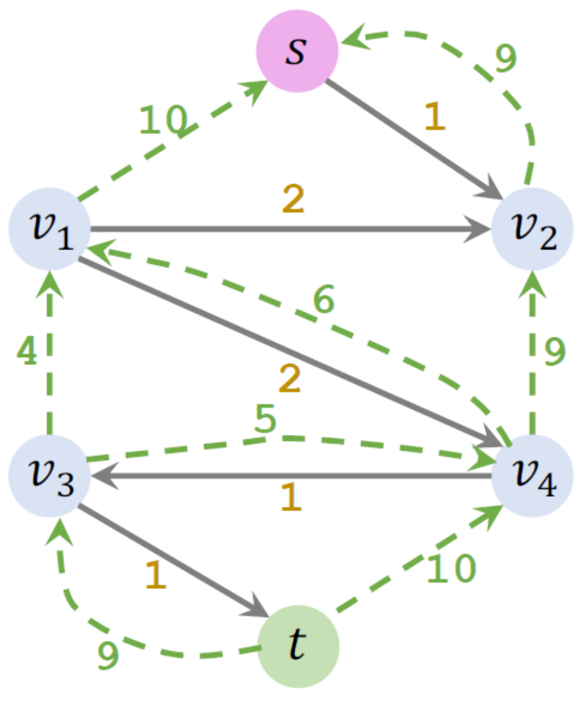
- Construct level graph
- Find blocking flow in level graph
- Update the residual graph, remove saturated edges
- Add flows to the residual graph as backward paths



Old Residual Graph



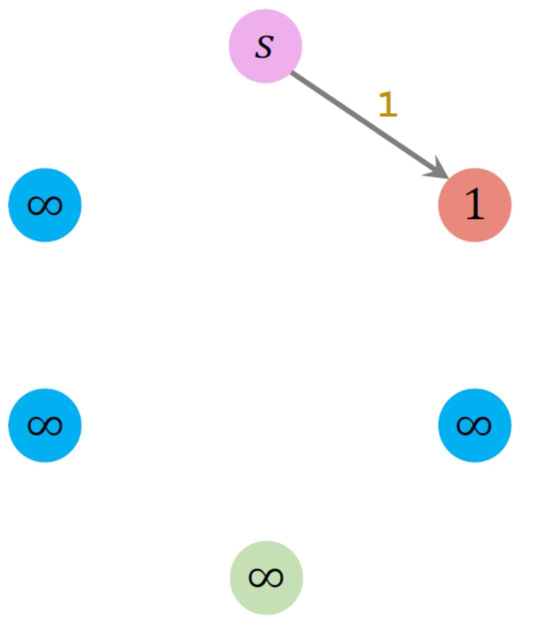
Level Graph



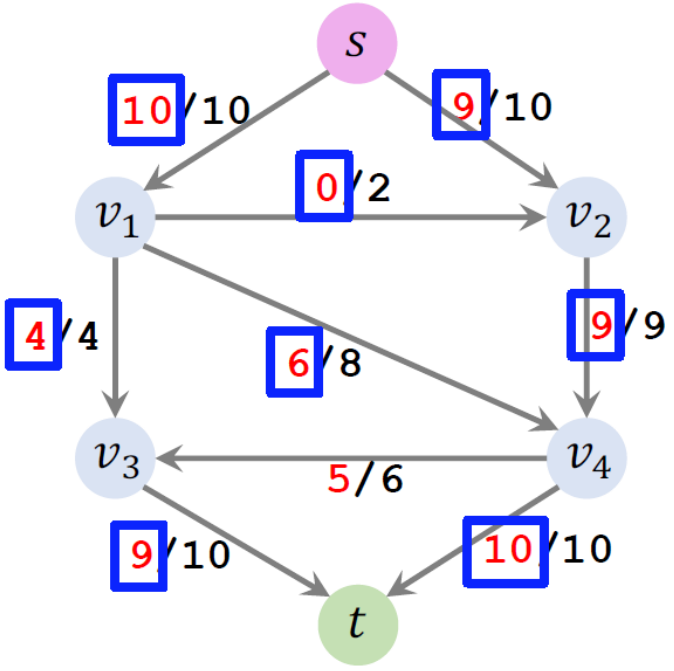
Residual Graph

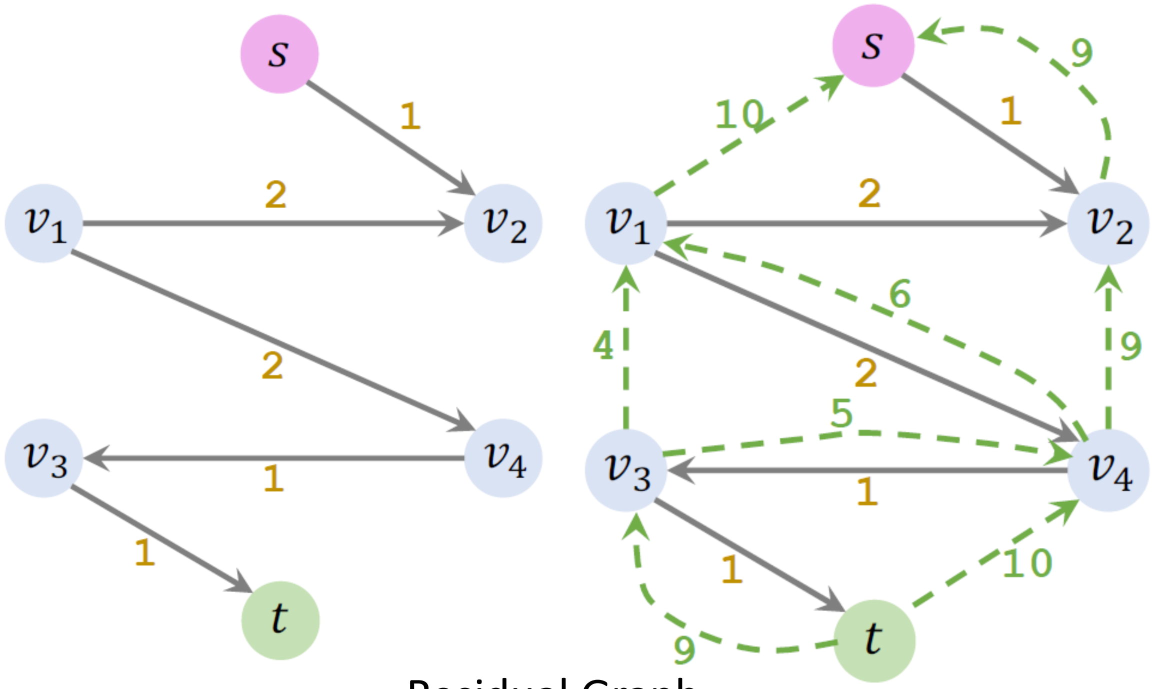
Iteration 3

- Construct level graph
- Find blocking flow in level graph
- On the level graph, no flow can be found
- Flow = Capacity – Residual



Level Graph





Residual Graph

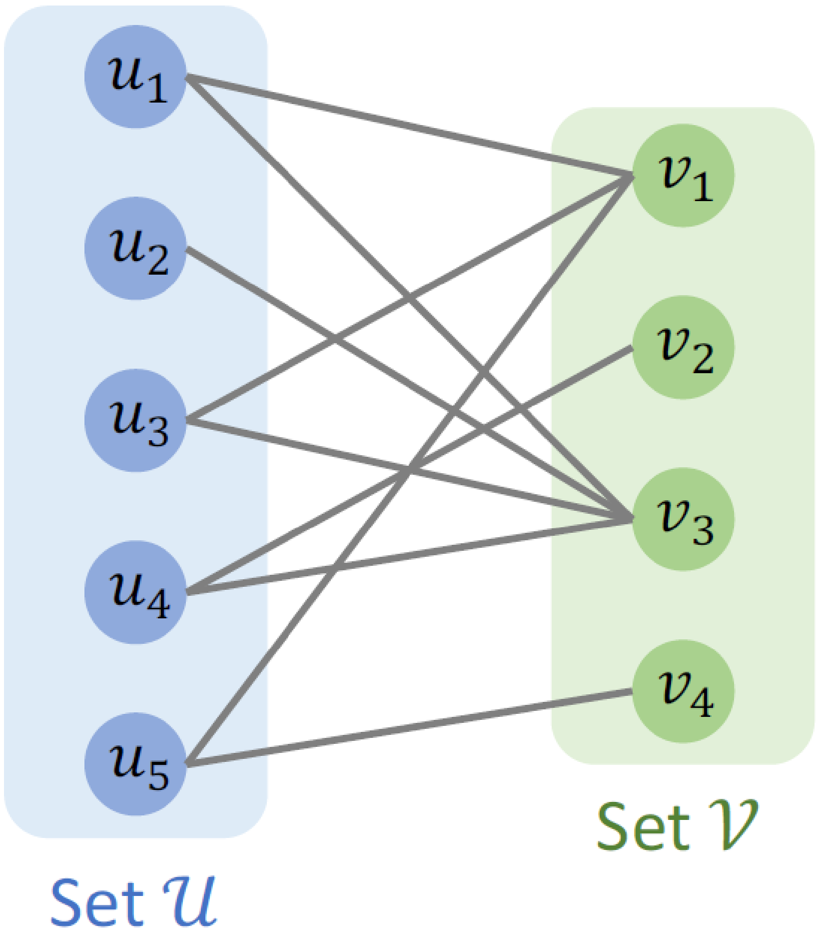
Dinic's algorithm: Time Complexity

1. Initially, the residual graph is a copy of the original graph
2. Repeat:
   1. Construct the level graph of the residual graph
   2. Find a blocking flow on the level graph
   3. Update the residual graph (update the weights, remove saturated edges, and add backward edges)

- Time complexity: $O(mn^2)$, (n is #vertices, m is #edges)
- Dinic's algorithm has at most n – 1 iterations.
- Per-iteration time complexity is $O(mn)$

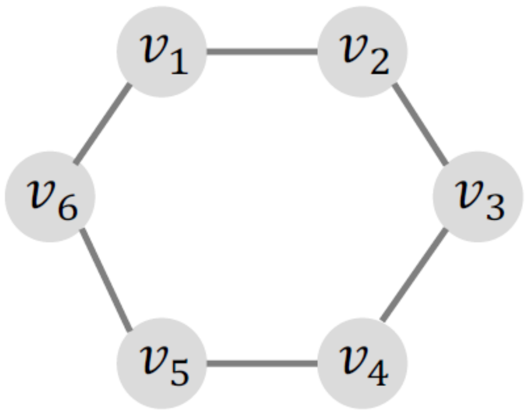
Bipartite Graph

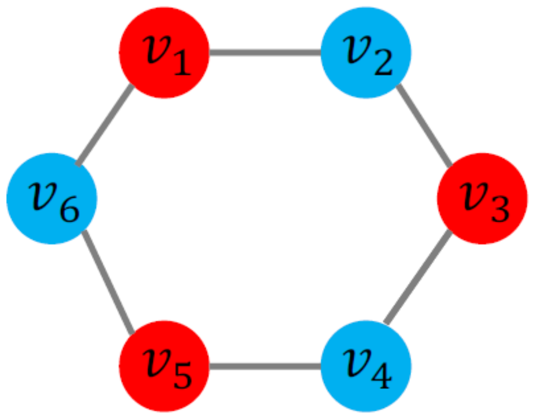
- Bipartite graph: $\mathcal{G} = \mathcal{U}, \mathcal{V}, \mathcal{E}$
- All the edges are between $\mathcal{U}$ and $\mathcal{V}$
- No edge between two vertices in $\mathcal{U}$
- No edge between two vertices in $\mathcal{V}$



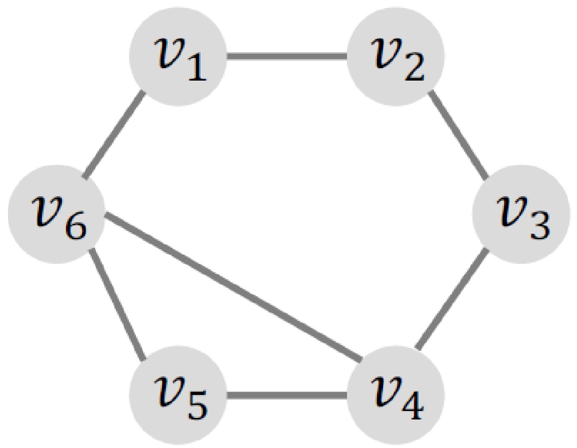
Is the graph bipartite?

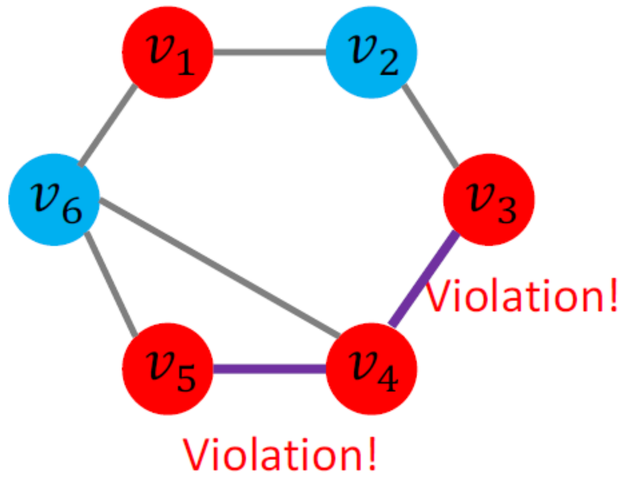
1. Select an arbitrary vertex and assign red color to it.
2. BFS to color neighbors, until all vertices are colored:
   1. Color red vertices' neighbors as blue
   2. Color blue vertices' neighbors as red
   3. During the process, if a vertex has the same color as its neighbor, then output FALSE
3. If no violation is found, return TRUE in the end





Bipartite

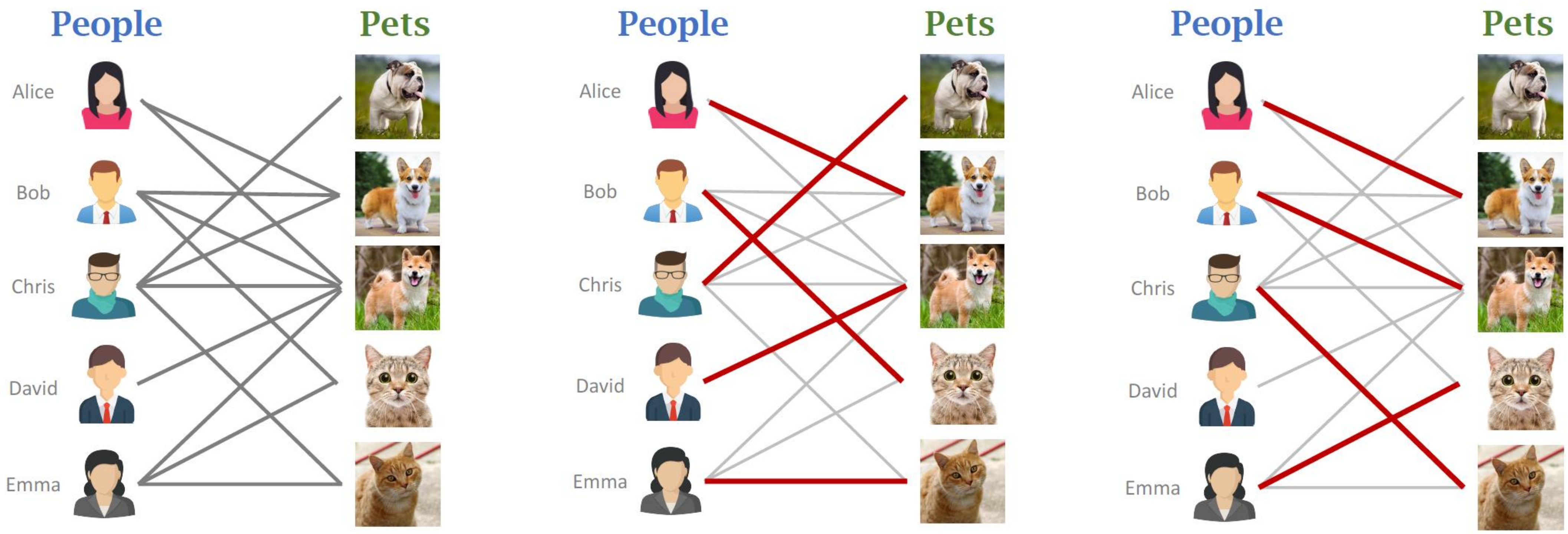




Maximum-Cardinality Bipartite Matching (MCBM)

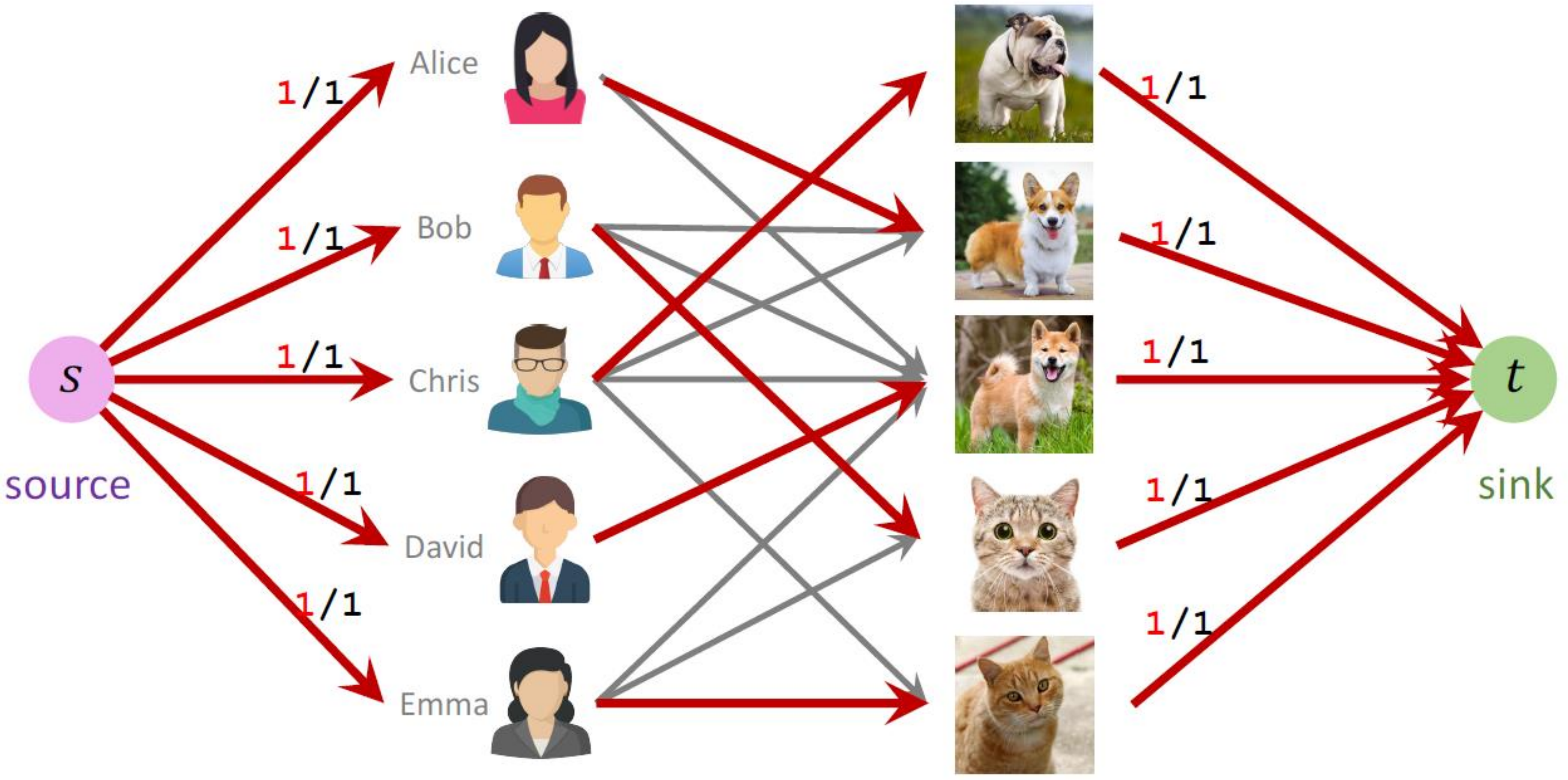
- Bipartite graph: $\mathcal{G} = \mathcal{U}, \mathcal{V}, \mathcal{E}$
- Set $\mathcal{U}$ contains people, Set $\mathcal{V}$ contains pets
- Edges in $\mathcal{E}$ are people's preference
- **Goal:** Maximizing the cardinality of matching, 5

Greedy algorithm can fail!



Maximum-Cardinality Bipartite Matching (MCBM)

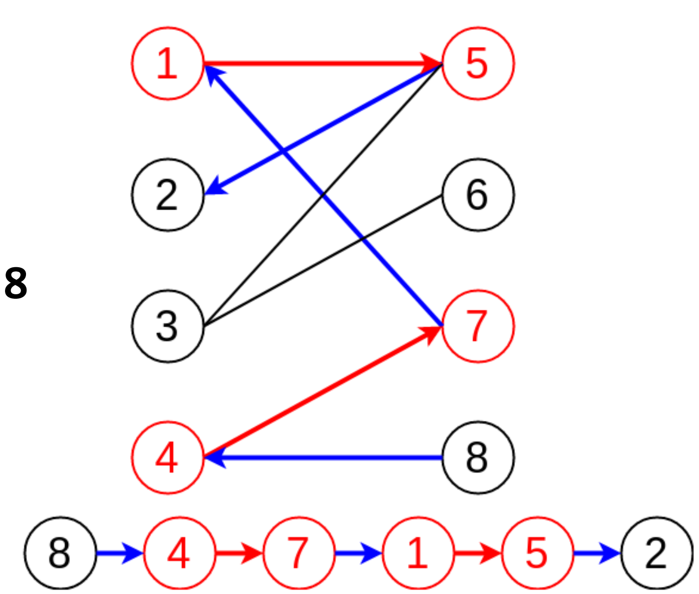
- Capacity of max-flow = Cardinality of max-matching



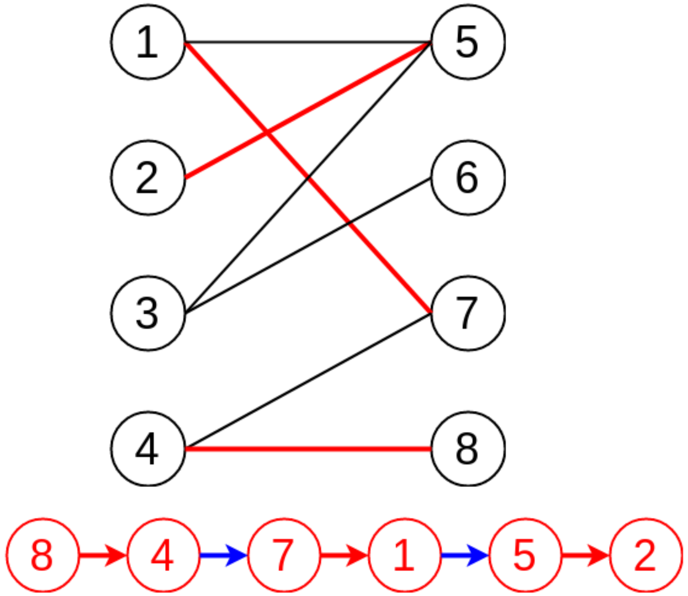
Augmenting path algorithm

- Given a matching
  - An alternating path is a path that begins with an unmatched vertex and whose edges belong alternately to the matching and not to the matching
  - An augmenting path is an alternating path that starts from and ends on (two different) unmatched vertices
  - The path length(total number of edges) of augmenting path must be an odd number
- Total number of unmatched edges is always greater than the number of matched edges by 1
- Switch between unmatched edges and matched edges in the augmenting path

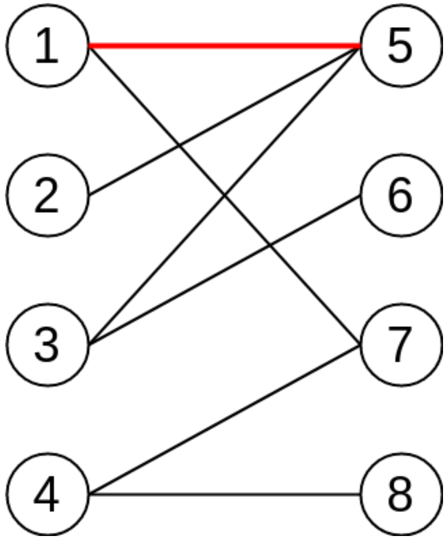
- 2 matchings (1->5, 4->7), so that vertices 1, 4, 5, 7 are matched vertices.
- starting from an unmatched vertex **8**
- unmatched edge -> matched edge -> unmatched edge…

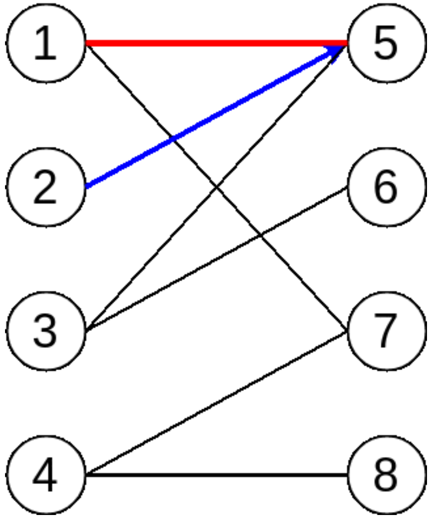


augmenting path:

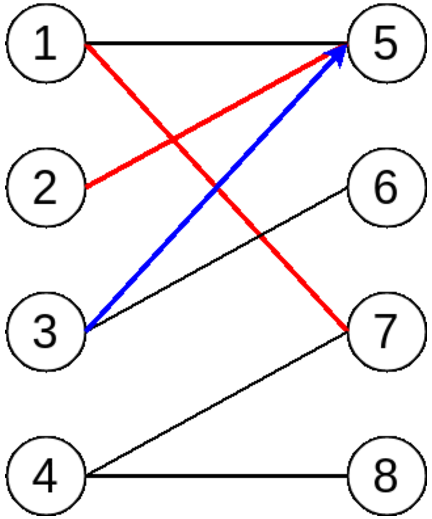


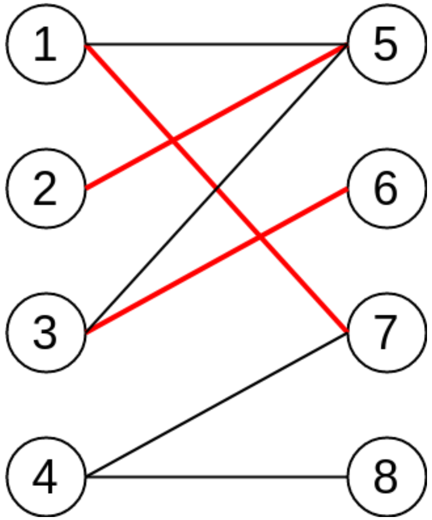
Augmenting path algorithm



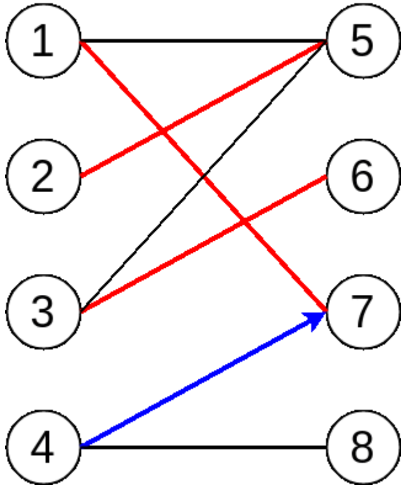


2 -> 5 -> 1 -> 7

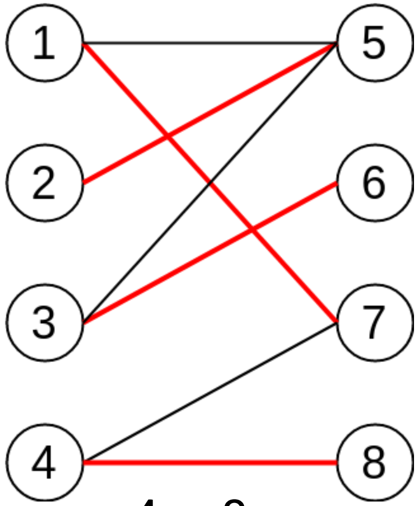




3->5 cannot, thus 3 -> 6



4 -> 7 -> 1 -> 5 -> 2 cannot

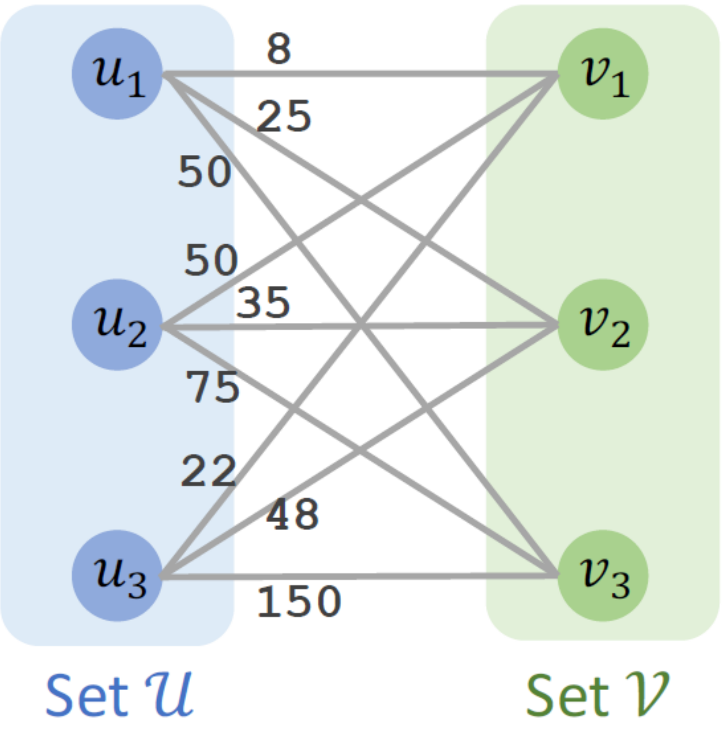


4 -> 8

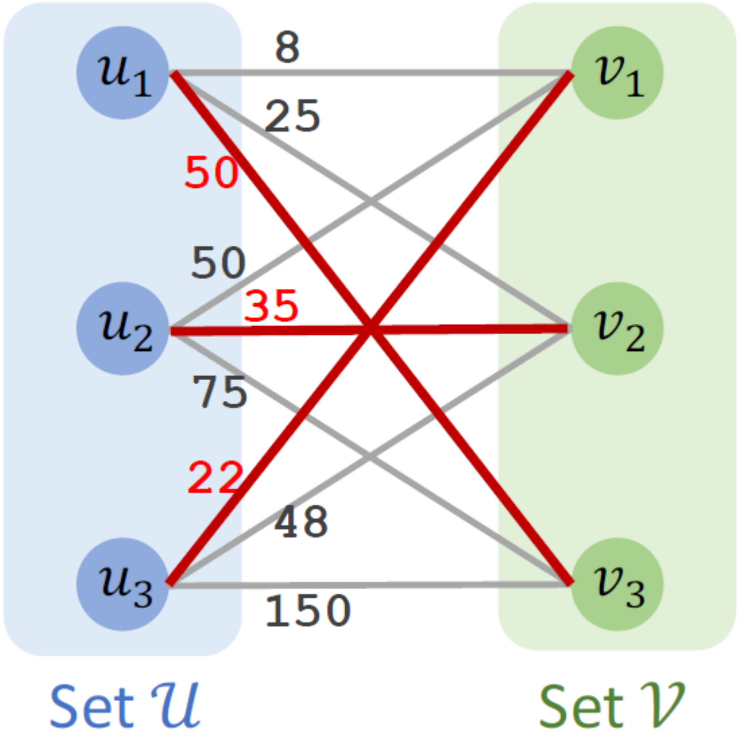
```
Graph g_;
int num_vertices_U_, num_vertices_V_;
int total_num_;
std::vector<bool> visited_;
std::map<int, int> matching_; // init as -1

bool FindMatching(int u) {
    // iterating sets V
    for (unsigned int v = num_vertices_U_; v < total_num_; v++) {
        // if there is a edge between vertices from U and V
        bool is_connected =
            std::find(g_.adj_list_[u].cbegin(), g_.adj_list_[u].cend(), v)
            != g_.adj_list_[u].cend();
        // augmenting path - unmatched -> matched -> ... -> matched -> unmatched
        if (false == visited_[v] && is_connected) {
            visited_[v] = true;
            // if vertex in V is not matched, we will match it
            // if it is matched already, we then go back to set U
            // we will try to figure out if the vertex from U can be
            // matched with another vertex in V
            // remember unmatched -> matched -> unmatched -> .....
            if (-1 == matching_[v] || FindMatching(matching_[v])) {
                matching_[v] = u;
                matching_[u] = v;
                return true;
            }
        }
    }
    return false;
}
```

Minimum-Weight Bipartite Matching: Hungarian Algorithm



	v_1	v_2	v_3
u_1	8	25	50
u_2	50	35	75
u_3	22	48	150



The minimum sum of weight is 50 + 35 + 22 = 107

Subtract Row Minima

	v_1	v_2	v_3
u_1	8	25	50
u_2	50	35	75
u_3	22	48	150

	v_1	v_2	v_3
u_1	8	25	50
u_2	50	35	75
u_3	22	48	150

	v_1	v_2	v_3
u_1	8 -8	25 -8	50 -8
u_2	50 -35	35 -35	75 -35
u_3	22 -22	48 -22	150 -22

	v_1	v_2	v_3
u_1	0	17	42
u_2	15	0	40
u_3	0	26	128

Now, the row minima are zeros

Subtract Column Minima

	v_1	v_2	v_3
u_1	0	17	42
u_2	15	0	40
u_3	0	26	128

	v_1	v_2	v_3
u_1	0	17	42
u_2	15	0	40
u_3	0	26	128

	v_1	v_2	v_3
u_1	0 -0	17 -0	42 -40
u_2	15 -0	0 -0	40 -40
u_3	0 -0	26 -0	128 -40

	v_1	v_2	v_3
u_1	0	17	2
u_2	15	0	0
u_3	0	26	88

Now, the col minima are zeros

Iteration 1

Repeat:

A. Cover all the zeros with a **minimum** number of lines

B. Decide whether to stop

- If n lines are required, the algorithm stops
- If less than n lines are required, then continue with Step C

C. Create additional zeros

- Find the smallest element (denote k) that is not covered by a line (k=2)
- Subtract k from all uncovered elements
- Add k to all the elements that are covered twice

	v_1	v_2	v_3
u_1	0	17	2
u_2	15	0	0
u_3	0	26	88

Not optimal!

	v_1	v_2	v_3
u_1	0	17	2
u_2	15	0	0
u_3	0	26	88

	v_1	v_2	v_3
u_1	0	17 −2	2 −2
u_2	15	0	0
u_3	0	26 −2	88 −2

	v_1	v_2	v_3
u_1	0	15	0
u_2	15 +2	0	0
u_3	0	24	86

	v_1	v_2	v_3
u_1	0	15	0
u_2	17	0	0
u_3	0	24	86

Iteration 2

Repeat:

A. Cover all the zeros with a **minimum** number of lines

B. Decide whether to stop
   - If n lines are required, the algorithm stops
   - If less than n lines are required, then continue with Step C.

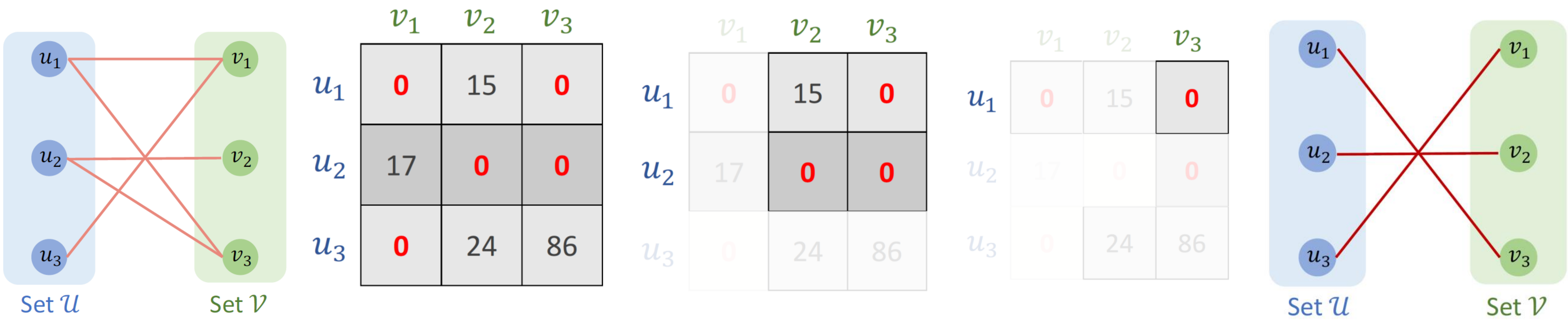
C. Create additional zeros
   - Find the smallest element (denote k) that is not covered by a line
   - Subtract k from all uncovered elements
   - Add k to all the elements that are covered twice

	v_1	v_2	v_3
u_1	0	15	0
u_2	17	0	0
u_3	0	24	86

Minimum number of lines: 3, thus, stop

Output the matching

- Choose a matching among the zeros
- Think of the zeros as edges
- Select zeros if they are the only zeros in row/col



Maximum Matching and Time Complexity

- Hungarian Algorithm for Maximum Matching
  - Idea: Max Matching ➞ Min Matching
  - Negate the signs of all the weights
  - It is equivalent to the minimum matching
  - Run the Hungarian algorithm
- Hungarian algorithm finds a minimum-weight bipartite matching.
  - It requires $\mathcal{U} = |\mathcal{V}| = n$
  - Time complexity: $O(n^3)$