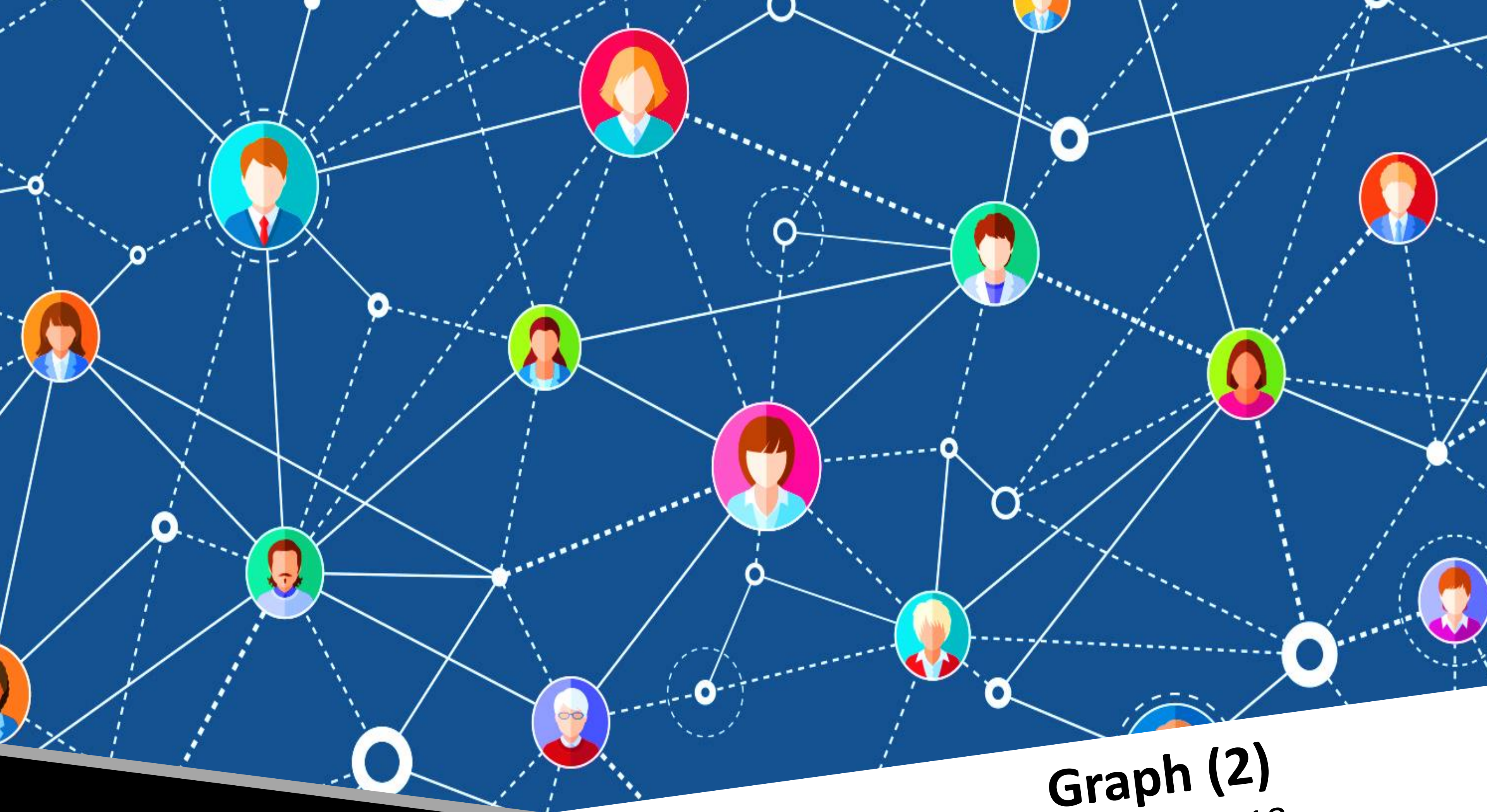


Graph (2)

Nov 29th, 2018

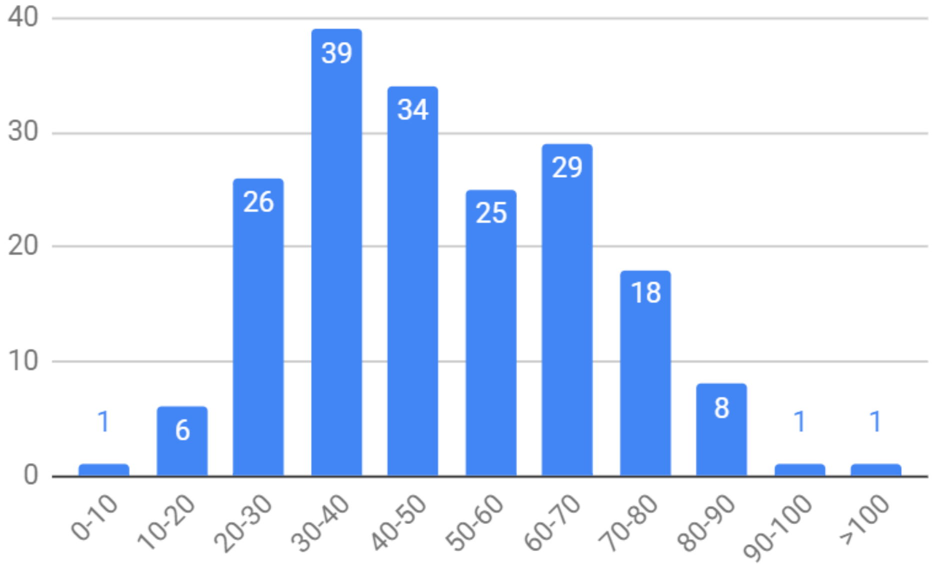
Algorithm Design and Analysis

YUN-NUNG (VIVIAN) CHEN HTTP://ADA.MIULAB.TW

國立臺灣大學
National Taiwan University

Slides credited from Hsueh-I Lu & Hsu-Chun Hsiao

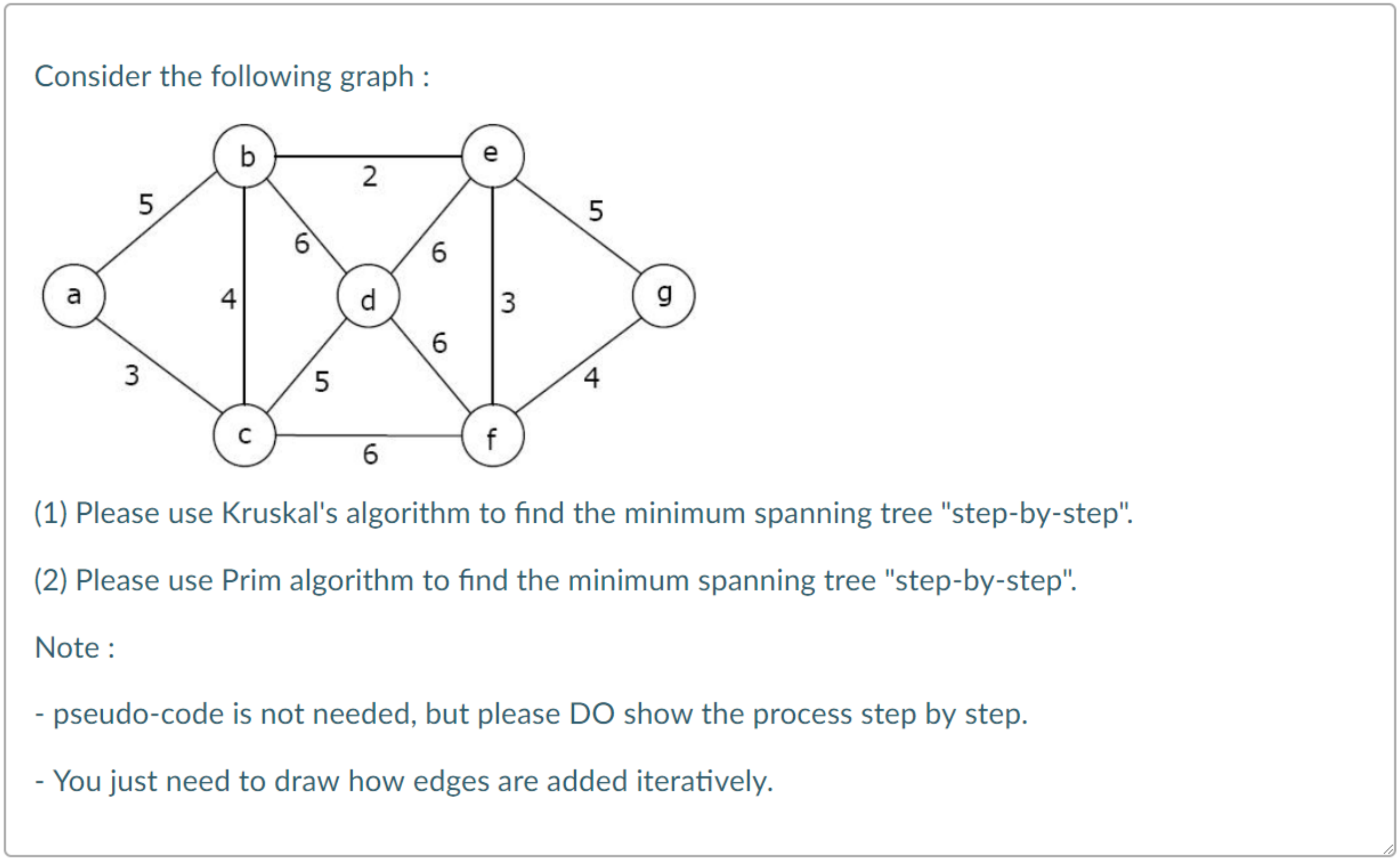
Announcement



- Midterm announced
  - Check the scores / answers
  - Find TAs (office hour / email) if you have questions by 12/06 (Thur)
- Homework 3 released
  - Due on 12/13 (Thur) 14:20 (two weeks)
- Mini-HW 8 released
  - Due on 12/06 (Thur) 14:20

Frequently check the website for the updated information!

Mini-HW 8



Consider the following graph :

(1) Please use Kruskal's algorithm to find the minimum spanning tree "step-by-step".

(2) Please use Prim algorithm to find the minimum spanning tree "step-by-step".

Note :

- pseudo-code is not needed, but please DO show the process step by step.
- You just need to draw how edges are added iteratively.

Outline

- DFS Applications
  - Strongly Connected Components
  - Topological Sorting
- Minimal Spanning Trees (MST)
  - Boruvka's Algorithm
  - Kruskal's Algorithm
  - Prim's Algorithm

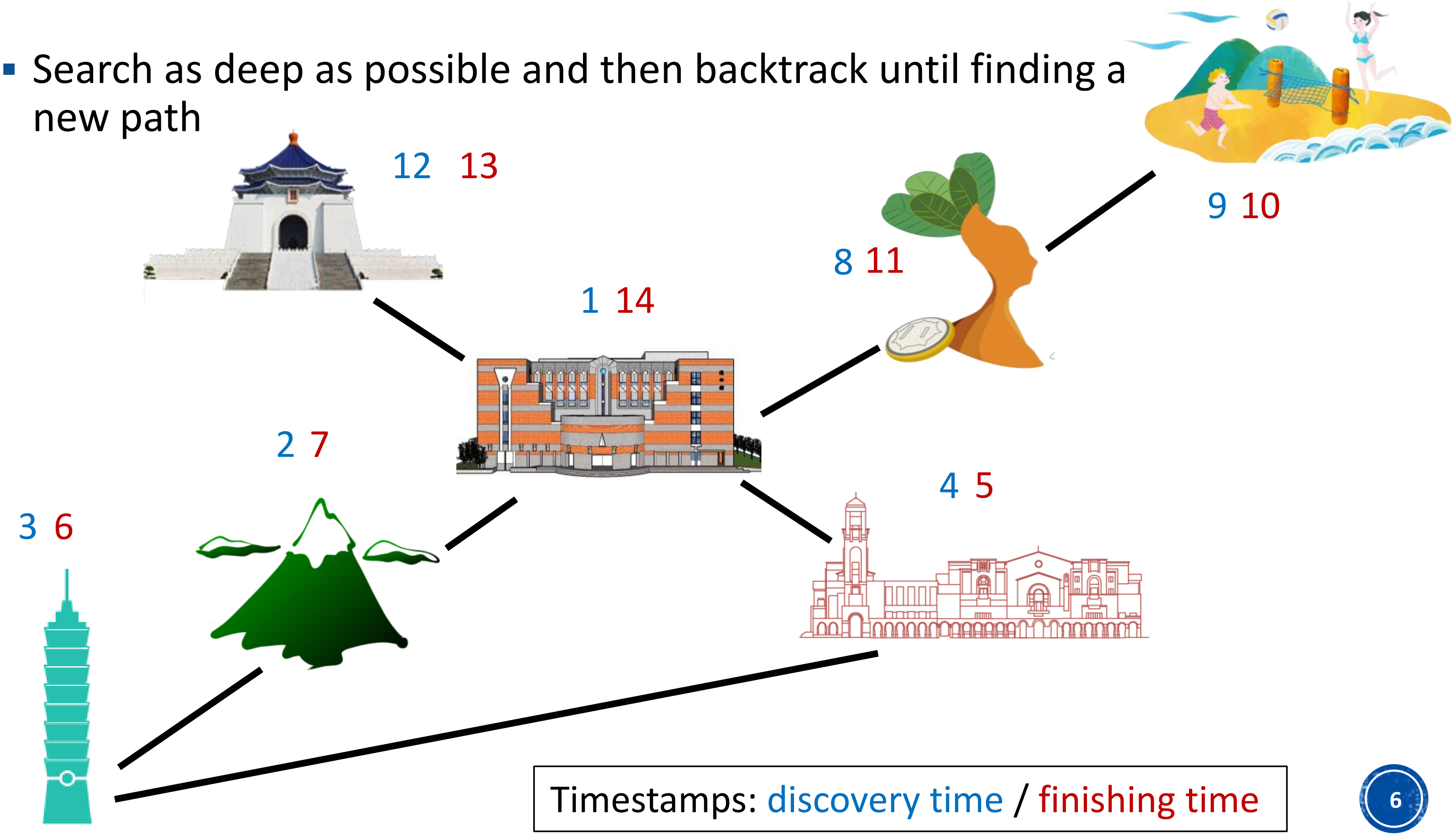
5

Depth-First Search

Textbook Chapter 22.3 – Depth-first search

Depth-First Search (DFS)

- Search as deep as possible and then backtrack until finding a new path



Timestamps: discovery time / finishing time

DFS Algorithm

```
// Explore full graph and builds up
a collection of DFS trees
DFS(G)
  for each vertex u in G.V
    u.color = WHITE
    u.pi = NIL
  time = 0 // global timestamp
  for each vertex u in G.V
    if u.color == WHITE
      DFS-VISIT(G, u)
```

```
DFS-Visit(G, u)
  time = time + 1
  u.d = time  // discover time
  u.color = GRAY
  for each v in G.Adj[u]
    if v.color == WHITE
      v.pi = u
      DFS-VISIT(G, v)
  u.color = BLACK
  time = time + 1
  u.f = time // finish time
```

- Implemented via recursion (stack)
- Color the vertices to keep track of progress:
  - GRAY: discovered (first time encountered)
  - BLACK: finished (all adjacent vertices discovered)
  - WHITE: undiscovered

DFS Properties

- Parenthesis Theorem
  - Parenthesis structure: represent the discovery of vertex u with a left parenthesis "(u" and represent its finishing by a right parenthesis "u)". In DFS, the parentheses are properly nested.
- White Path Theorem
  - In a DFS forest of a directed or undirected graph $G = (V, E)$,
    - vertex v is a descendant of vertex u in the forest ⇔ at the time $u.d$ that the search discovers u, there is a path from u to v in G consisting entirely of WHITE vertices
- Classification of Edges in G
  - Tree Edge
  - Back Edge
  - Forward Edge
  - Cross Edge

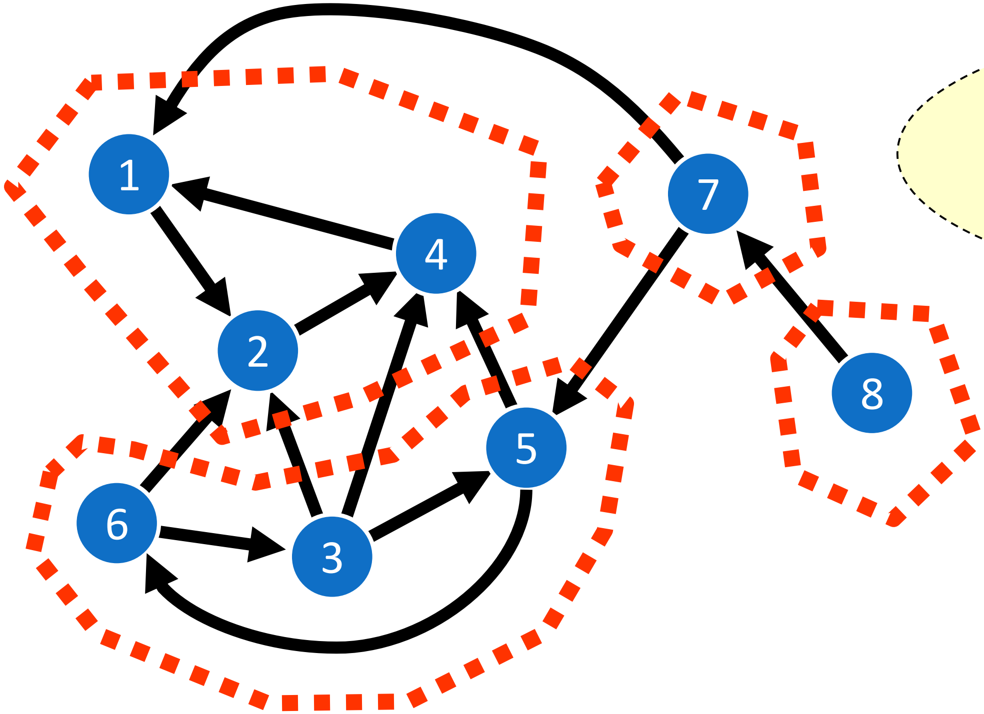
9

Strongly Connected Components

Textbook Chapter 22.5 – Strongly connected components

Strongly Connected Components

- Input: a directed graph $G = (V, E)$
- Output: a connected component of G
  - a **maximal** subset U of V s.t. any two nodes in U are reachable in G

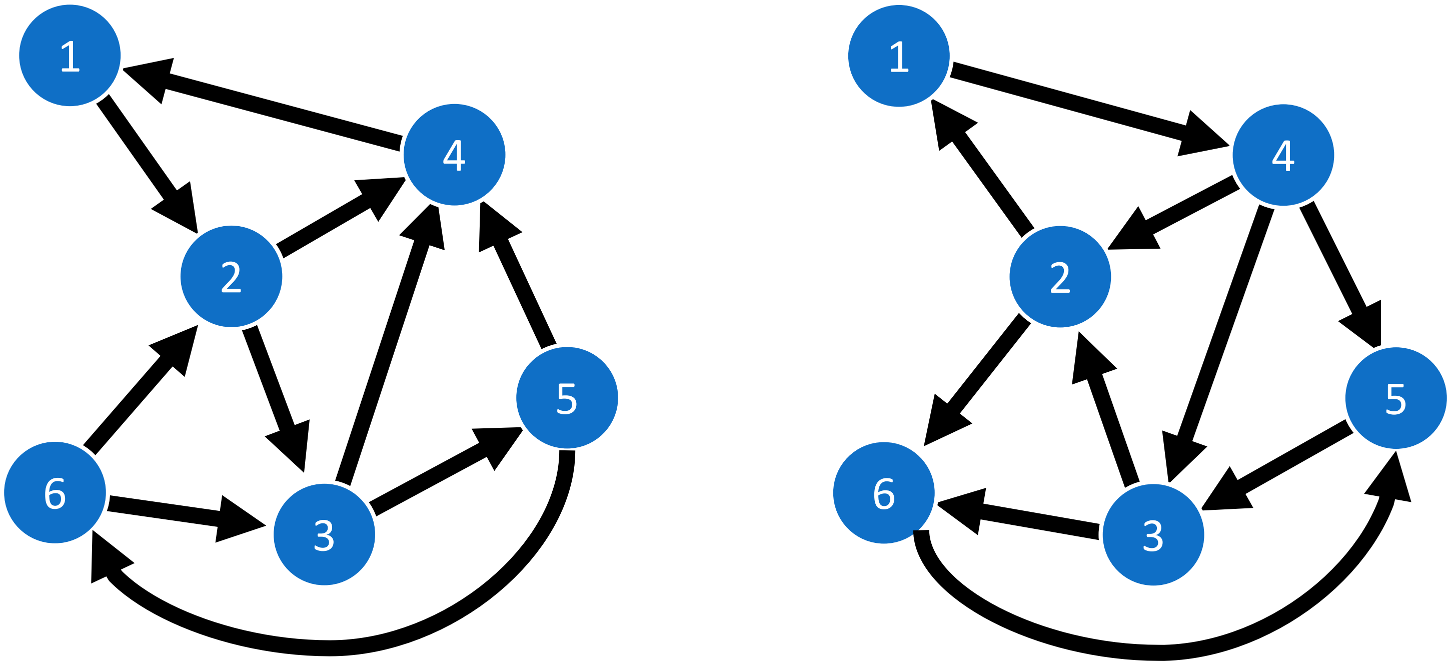


Why must the strongly connected components of a graph be disjoint?

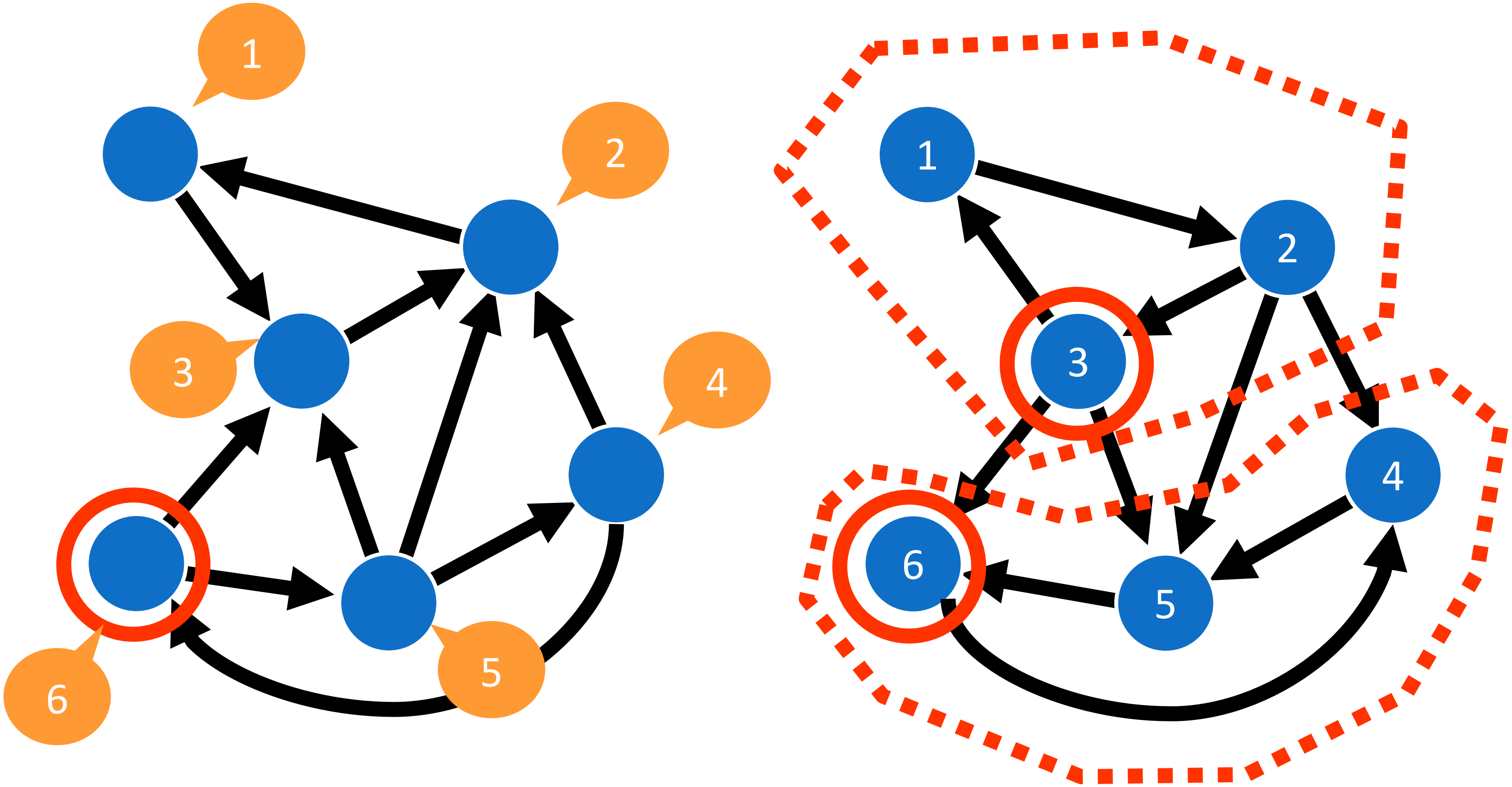
Algorithm

- Step 1: Run DFS on G to obtain the finish time $v.f$ for $v \in V$.
- Step 2: Run DFS on the transpose of G where the vertices V are processed in the decreasing order of their finish time.
- Step 3: output the vertex partition by the second DFS

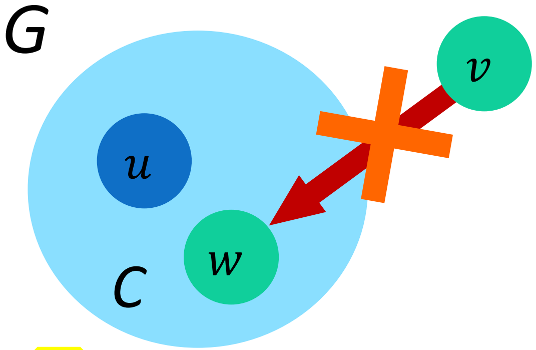
Transpose of A Graph



Example Illustration



Algorithm Correctness



Lemma

Let C be the strongly connected component of G (and G^T) that contains the node u with the largest finish time $u.f$. Then C cannot have any incoming edge from any node of G not in C.

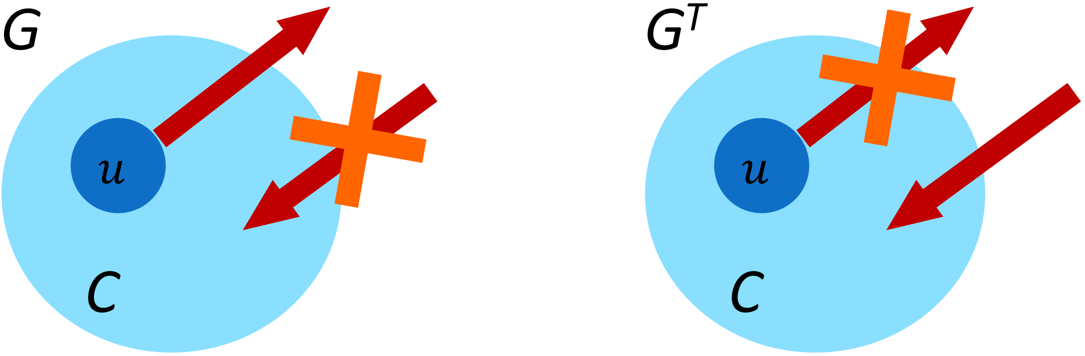
- Proof by contradiction
  - Assume that (v, w) is an incoming edge to C.
  - Since C is a strongly connected component of G, there cannot be any path from any node of C to v in G.
  - Therefore, the finish time of v has to be larger than any node in C, including u. → $v.f > u.f$, contradiction

Algorithm Correctness

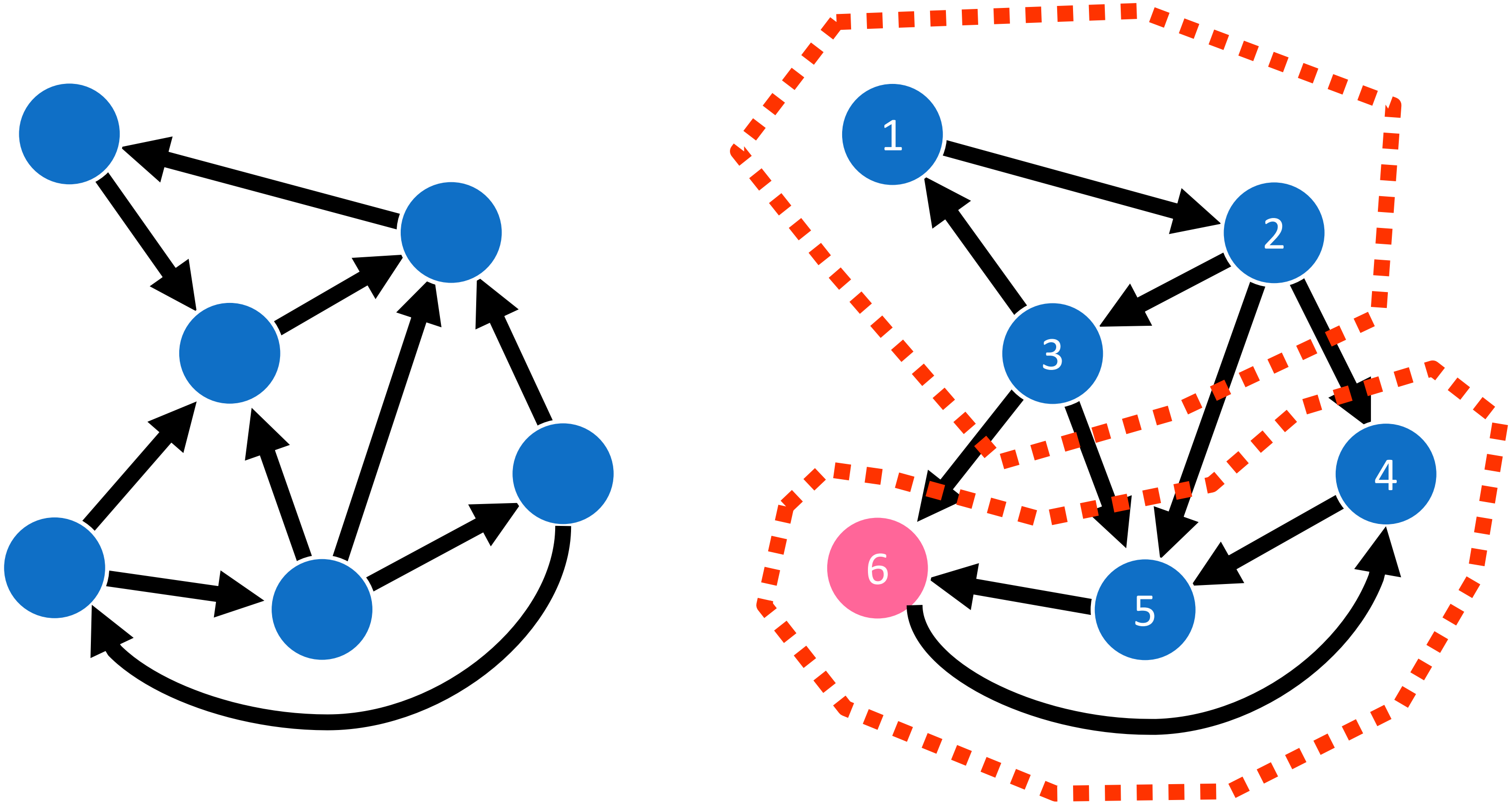
Theorem

By continuing the process from the vertex u^* whose finish time $u^*.f$ is the largest excluding those in C, the algorithm returns the strongly connected components.

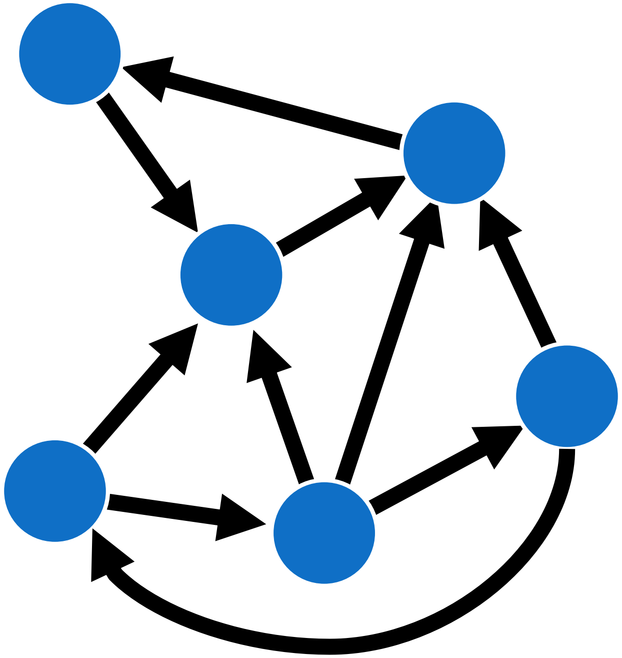
- Practice to prove using induction

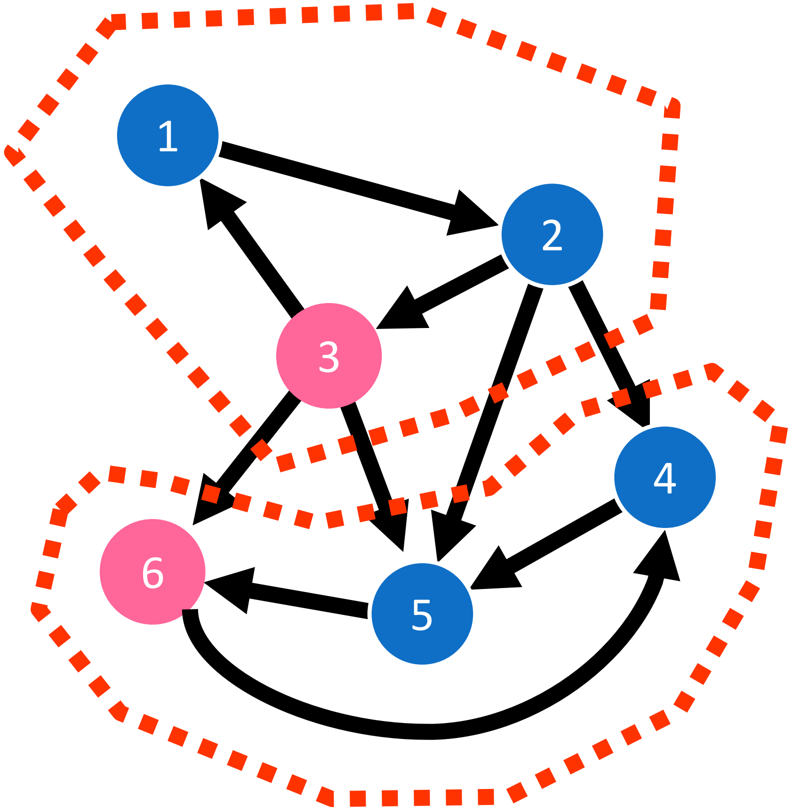


Example



Example



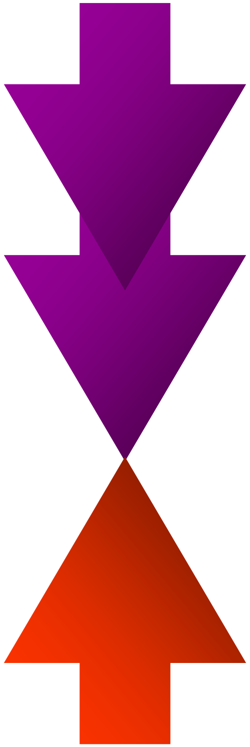


Time Complexity

- Step 1: Run DFS on G to obtain the finish time $v.f$ for $v \in V$.
- Step 2: Run DFS on the transpose of G where the vertices V are processed in the decreasing order of their finish time.
- Step 3: output the vertex partition by the second DFS

Time Complexity: $\Theta(n + m)$

Problem Complexity



$$\text{Upper bound} = O(m+n)$$

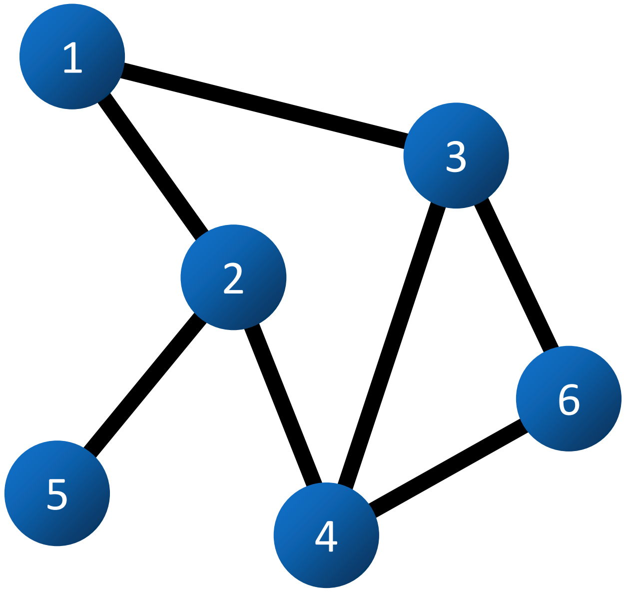
$$\text{Lower bound} = \Omega(m+n)$$

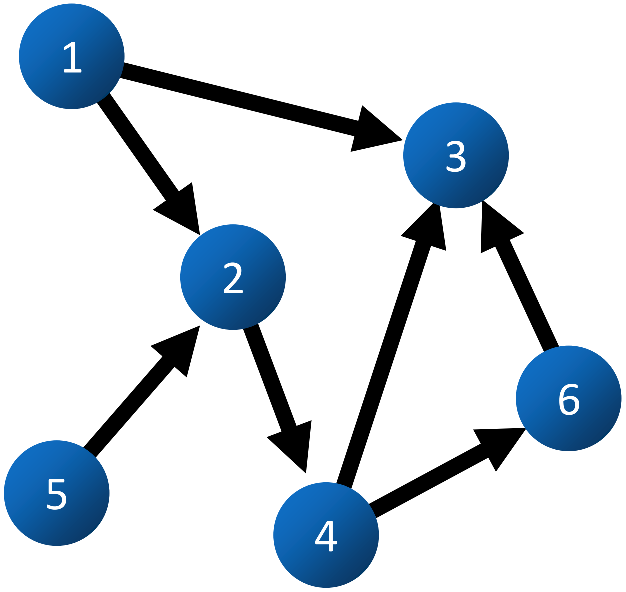
20

Topological Sort

Textbook Chapter 22.4 – Topological sort

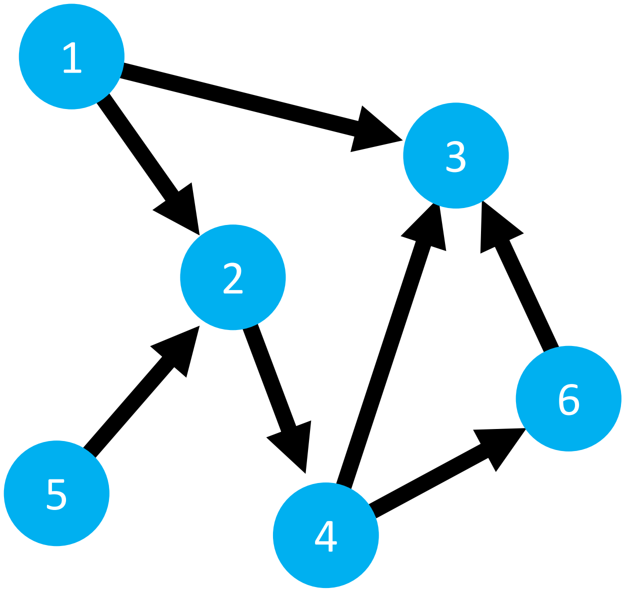
Directed Graph





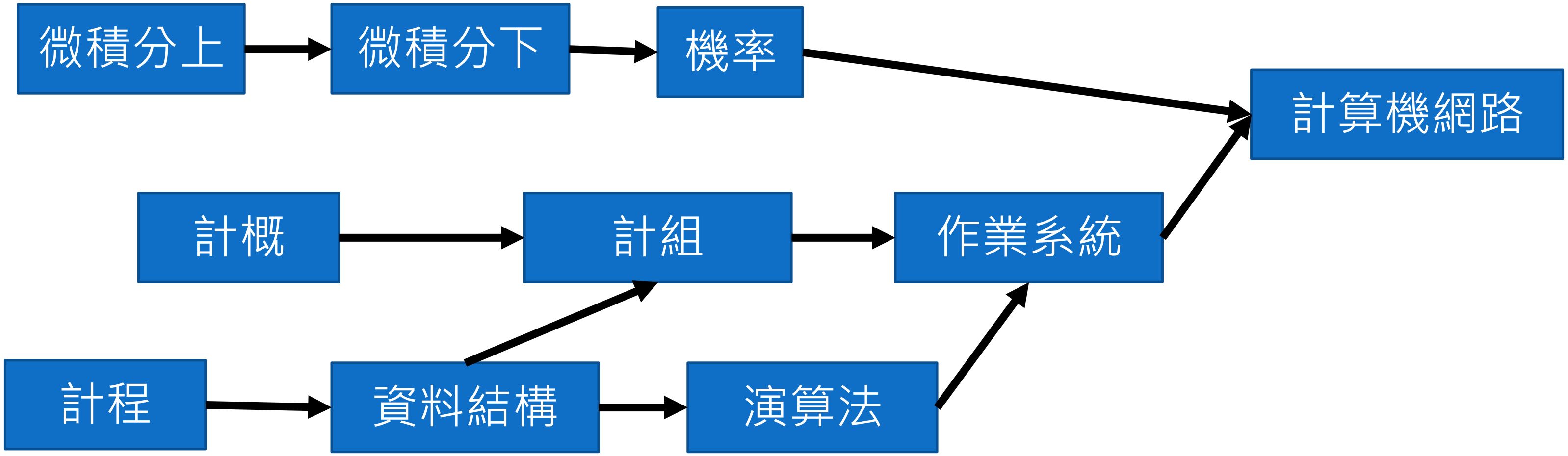
Directed Acyclic Graph (DAG)

- Definition
  - a directed graph without any directed cycle



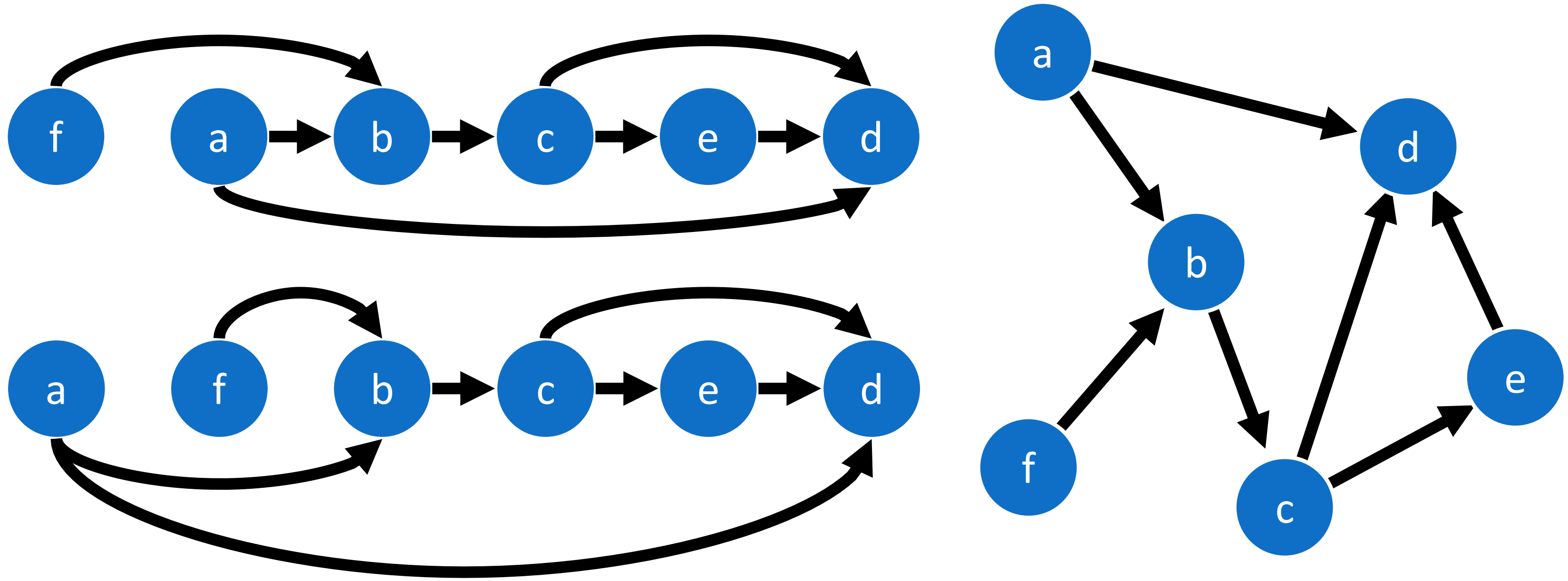
Topological Sort Problem

- Taking courses should follow the specific order
- How to find a course taking order?



Topological Sort Problem

- Input: a directed acyclic graph $G = (V, E)$
- Output: a linear order of V s.t. all edges of G going from lower-indexed nodes to higher-indexed nodes (左→右)



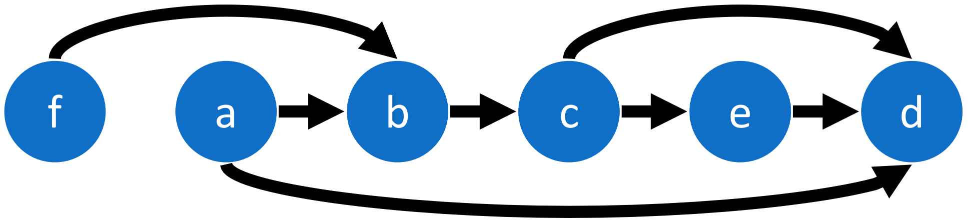
Algorithm

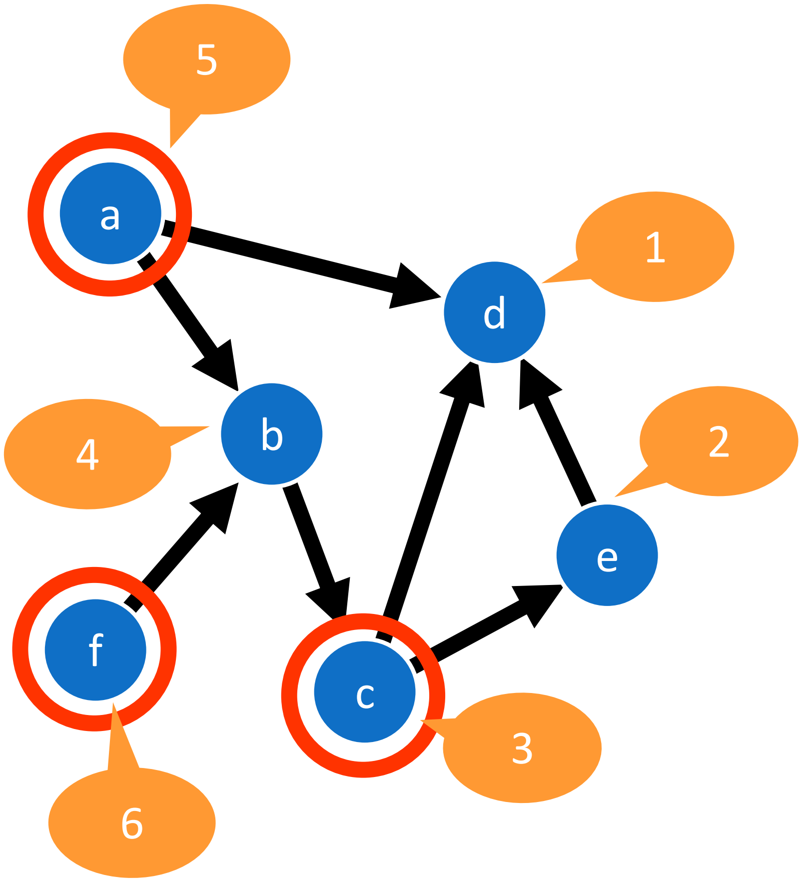
- Run DFS on the input DAG *G*.
- Output the nodes in decreasing order of their finish time.

```
DFS(G)
  for each vertex u in G.V
    u.color = WHITE
    u.pi = NIL
  time = 0
  for each vertex u in G.V
    if u.color == WHITE
      DFS-VISIT(G, u)
```

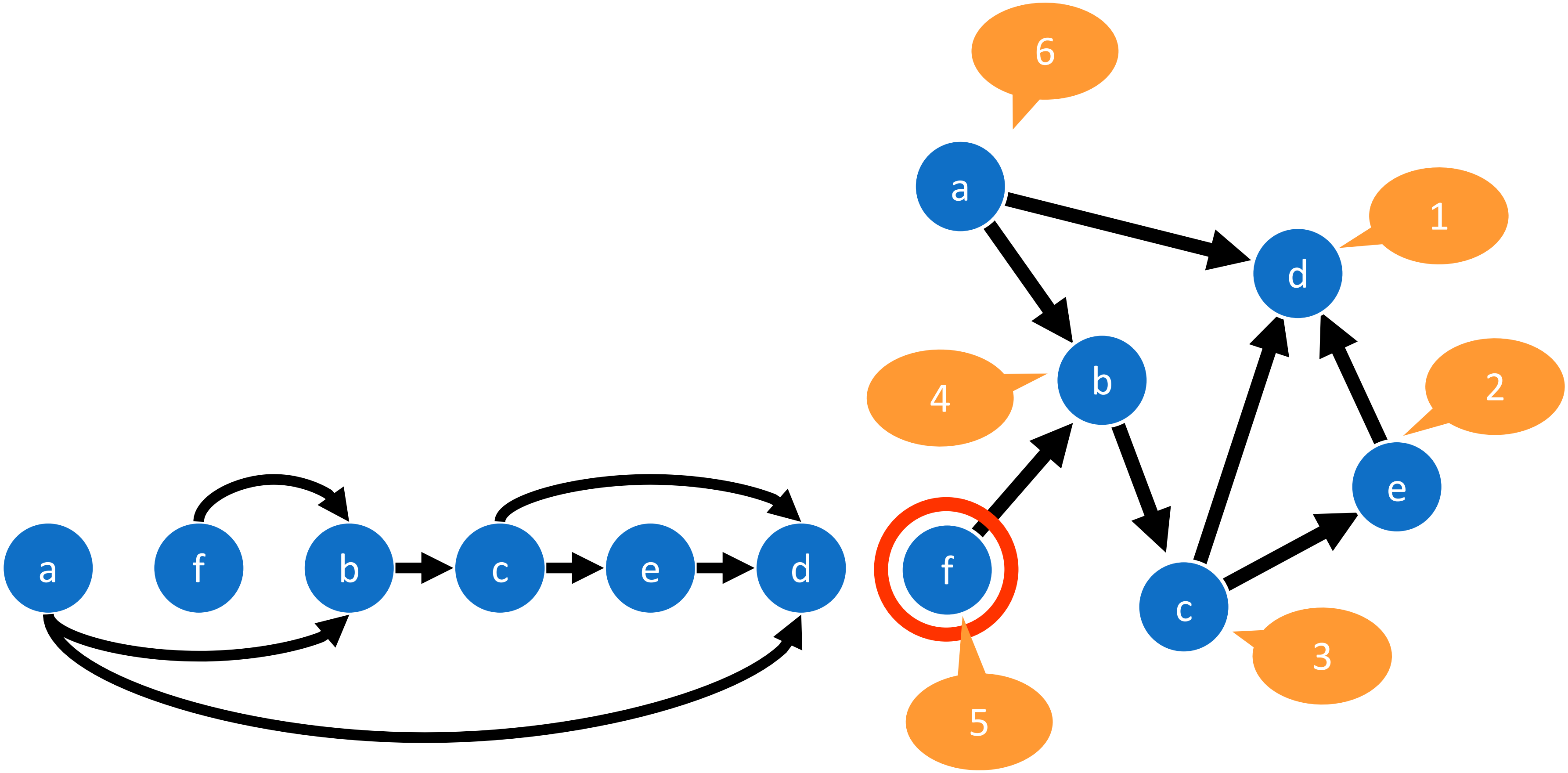
```
DFS-Visit(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY
  for each v in G.Adj[u] (outgoing)
    if v.color == WHITE
      v.pi = u
      DFS-VISIT(G, v)
  u.color = BLACK
  time = time + 1
  u.f = time // finish time
```

Example Illustration





Example Illustration



Time Complexity

- Run DFS on the input DAG *G*. $\Theta(n+m)$
- Output the nodes in decreasing order of their finish time.
  - As each vertex is finished, insert it onto the front of a linked list $\Theta(n)$
  - Return the linked list of vertices

Time Complexity: $\Theta(n+m)$

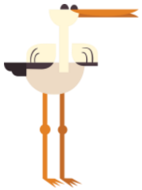
```
DFS(G)
  for each vertex u in G.V
    u.color = WHITE
    u.pi = NIL
  time = 0
  for each vertex u in G.V
    if u.color == WHITE
      DFS-VISIT(G, u)
```

```
DFS-Visit(G, u)
  time = time + 1
  u.d = time
  u.color = GRAY
  for each v in G.Adj[u]
    if v.color == WHITE
      v.pi = u
      DFS-VISIT(G, v)
  u.color = BLACK
  time = time + 1
  u.f = time // finish time
```

Algorithm Correctness

Lemma 22.11
A directed graph is acyclic ⇔ a DFS yields no back edges.

- Proof
  - →: suppose there is a back edge (u, v)
    - v is an ancestor of u in DFS forest
    - There is a path from v to u in G and (u, v) completes the cycle
  - ← : suppose there is a cycle c
    - Let v be the first vertex in c to be discovered and u is a predecessor of v in c
    - Upon discovering v the whole cycle from v to u is WHITE
    - At time $v.d$, the vertices of c form a path of white vertices from v to u
    - By the white-path theorem, vertex u becomes a descendant of v in the DFS forest
    - Therefore, (u, v) is a back edge



White Path Theorem: In a DFS forest of G, v is a descendant of u in the forest ⇔ at the time $u.d$ that the search discovers u, there is a path from u to v in G consisting entirely of WHITE vertices

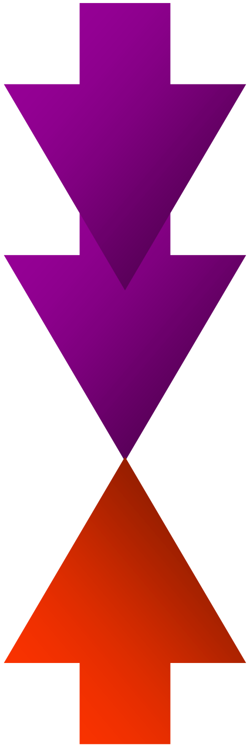
Algorithm Correctness

Theorem 22.12

The algorithm produces a topological sort of the input DAG. That is, if (u, v) is a directed edge (from u to v) of G, then $u.f > v.f$.

- Proof
  - When (u, v) is being explored, u is GRAY and there are three cases for v:
    - Case 1 – GRAY
      - (u, v) is a back edge (contradicting Lemma 22.11), so v cannot be GRAY
    - Case 2 – WHITE
      - v becomes descendant of u
      - v will be finished before u ➡ $v.f < u.f$
    - Case 3 – BLACK
      - v is already finished ➡ $v.f < u.f$

Problem Complexity



$$\text{Upper bound} = O(m+n)$$

$$\text{Lower bound} = \Omega(m+n)$$

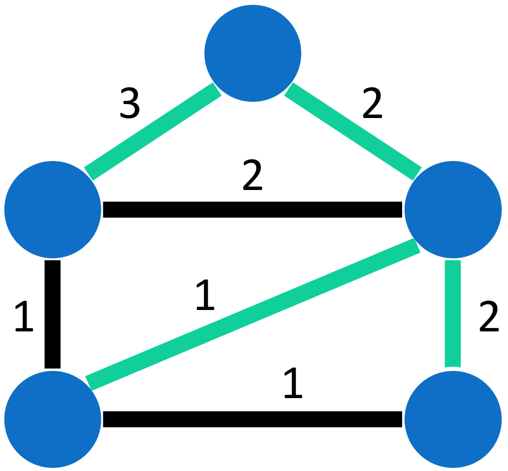
Discussion

- Since cycle detection becomes back edge detection (Lemma 22.11), DFS can be used to test whether a graph is a DAG
- Is there a topological order for cyclic graphs?
- Given a topological order, is there always a DFS traversal that produces such an order?

Minimal Spanning Tree (MST)

Textbook Chapter 23 – Minimal Spanning Trees

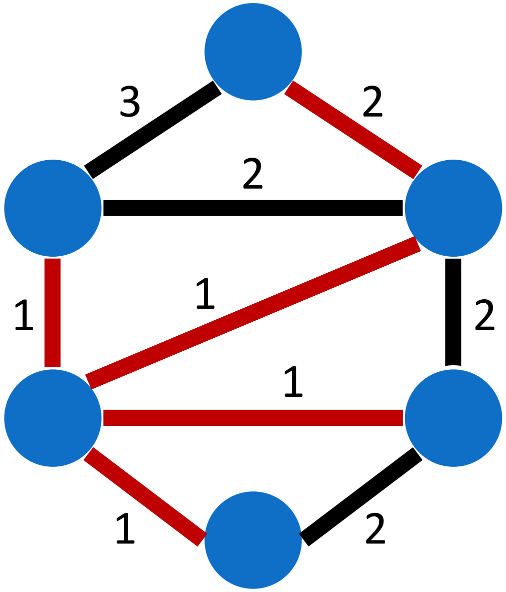
Spanning Tree



- Definition
  - a subgraph that is a tree and connects all vertices
    - Exactly $n - 1$ edges
    - Acyclic
  - There can be many spanning trees of a graph
- BFS and DFS also generate spanning trees
  - BFS tree is typically “short and bushy”
  - DFS tree is typically “long and stringy”

Minimal Spanning Tree Problem

- Input: a connected n-node m-edge graph G with edge weights w
- Output: a spanning tree T of G with minimum $w(T)$



WLOG: we may assume that all edge weights are distinct

Minimal Spanning Tree Problem

- Q: What if the graph is unweighted?

  Trivial

- Q: What if the graph contains edges with negative weights?

  Add a large constant to every edge; a MST remains the same

Uniqueness of MST

Theorem: MST is unique if all edge weights are distinct

- Proof by contradiction
  - Suppose there are two MSTs A and B
  - Let e be the least-weight edge in $A \cup B$ and e is not in both
  - WLOG, assume e is in A
  - Add e to B; $\{e\} \cup B$ contains a cycle C
  - B includes at least one edge e' that is not in A but on C
  - Replacing e' with e yields a MST with less cost

If edge weights are not all distinct, then the (multi-)set of weights in MST is unique

38 Borůvka’s Algorithm

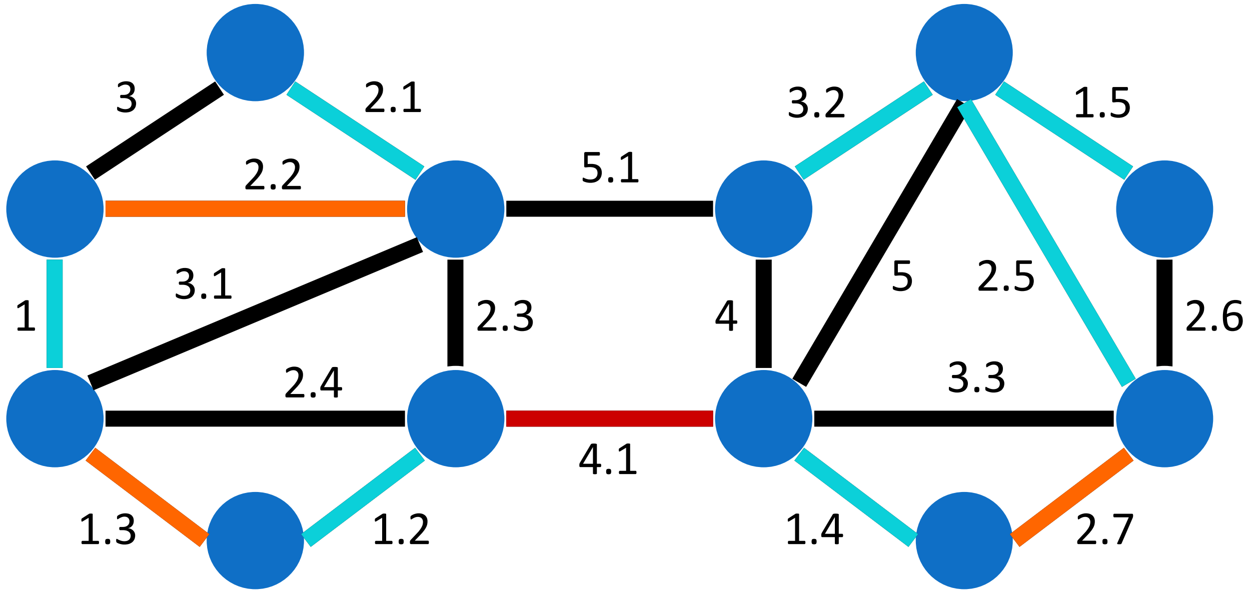
Inventor of MST

- Otakar Borůvka
  - Czech scientist
  - Introduced the problem
  - Gave an $O(m \log n)$ time algorithm
    - The original paper was written in Czech in 1926
    - The purpose was to efficiently provide electric coverage of Bohemia

Borůvka's Algorithm

- Repeat the following procedure until the resulting graph becomes a single node
  - For each node u, mark its lightest incident edge
  - From the marked edges form a forest F, add the edges of F into the set of edges to be reported
  - Contract each maximal subtree of F into a single node

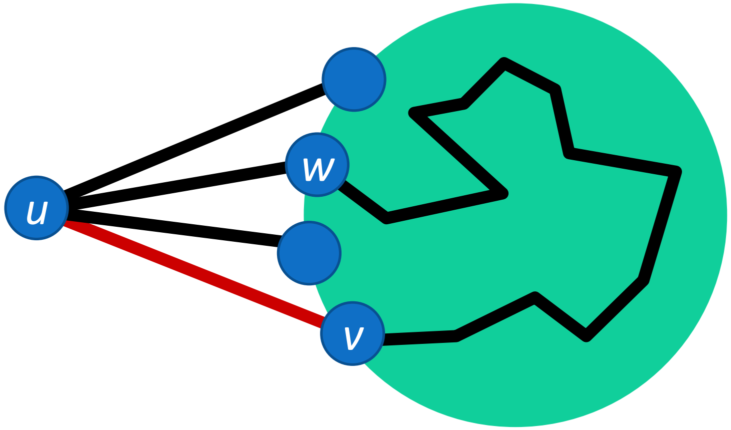
Borůvka's Algorithm Illustration



Algorithm Correctness

Claim: If (u, v) is the lightest edge incident to u in G, (u, v) must belong to any MST of G

- Proof via contradiction
  - An MST T of G that does not contain (u, v)
  - A cycle $C = T \cup (u, v)$ contains an edge (u, w) in C that has larger weight than (u, v)
  - $T' = T \cup (u, v) \backslash (u, w)$ must be a spanning tree of G lighter than T



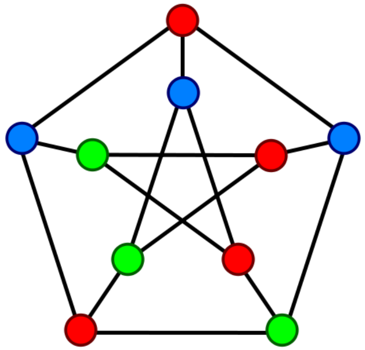
Time Complexity

- The recurrence relation

$$T(m, n) \leq T(m, n/2) + O(m)$$

  - We check all edges in each phase ➡ $O(m)$
  - After each contraction phase, the number of nodes is reduced by at least one half
- Time complexity: $O(m \log n)$

Cycle Property

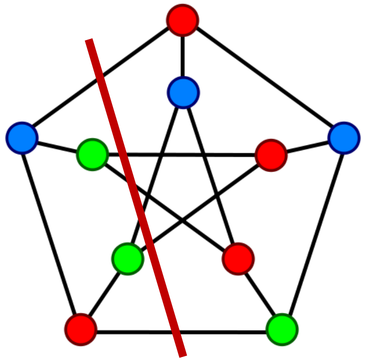


Let C be any cycle in the graph G, and let e be an edge with the maximum weight on C. Then the MST does not contain e.

- For simplicity, assume all edge weights are distinct

- Proof by contradiction
  - Suppose e is in the MST
  - Removing e disconnects the MST into two components T1 and T2
  - There exists another edge e' in C that can reconnect T1 and T2
  - Since $w(e') < w(e)$, the new tree has a lower weight
  - Contradiction!

Cut Property



Let C be a cut in the graph, and let e be the edge with the minimum cost in C. Then the MST contains e.

- Cut = a partition of the vertices
- For simplicity, assume all edge weights are distinct

- Proof by contradiction
  - Suppose e is not in the current MST
  - Adding e creates a cycle in the MST
  - There exists another edge e' in C that can break the cycle
  - Since $w(e') > w(e)$, the new tree has a lower weight
  - Contradiction!

46

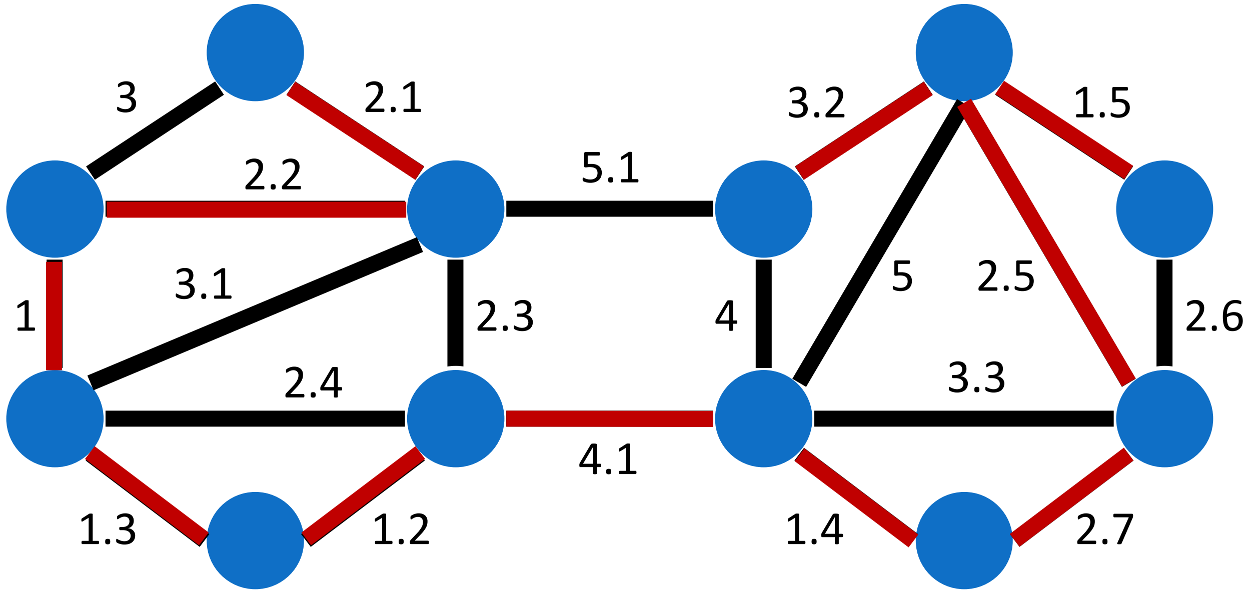
Kruskal's Algorithm

Textbook Chapter 23.2 – The algorithms of Kruskal and Prim

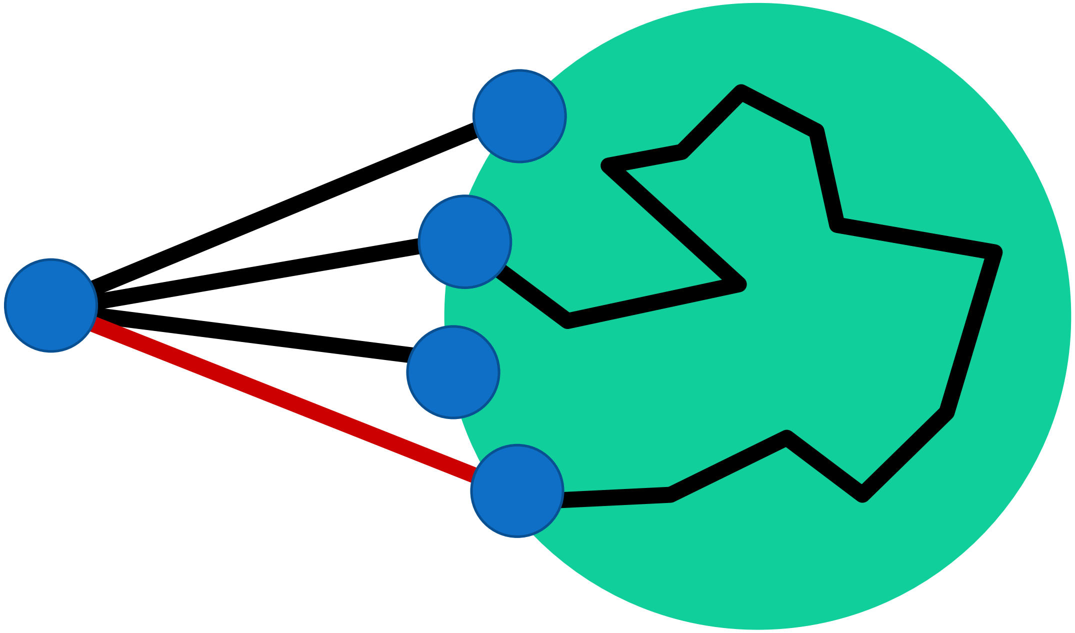
Kruskal’s Algorithm

- For each node u
  - Make-set(u): create a set consisting of u
- For each edge (u, v), taken in non-decreasing order by weights
  - if Find-set(u) ≠Find-set(v) (i.e., u and v are not in the same set) then
    - Output edge (u, v)
    - Union(u, v): union the sets containing u and v into a single set

Kruskal’s Algorithm Illustration



Kruskal’s Algorithm Correctness



The lightest edge incident to a vertex must be in the MST

Kruskal’s Algorithm Correctness

- Consider whether adding e creates a cycle:
  - If adding e to T creates a cycle C
    - Then e is the max weight edge in C
    - The cycle property ensures that e is not in the MST
  - If adding $e = (u, v)$ to T does not create a cycle
    - Before adding e, the current MST can be divided into two trees T1 and T2 such that u in T1 and V in T2
    - e is the minimum-cost edge on the cut of T1 and T2
    - The cut property ensures that e is in the MST

Kruskal’s Time Complexity

```
MST-KRUSKAL(G, w) // w = weights
  A = empty // edge set of MST
  for v in G.V
    MAKE-SET(v)
  sort edges of G.E into non-decreasing order by weight w  $O(m \log m)$
  for (u, v) in G.E, taken in non-decreasing order by weight $m$ times
    if FIND-SET(u) ≠ FIND-SET(v)
      A = A ∪ {u, v}
      UNION(u, v)
  return A
```

- Disjoint-set data structure with union-by-rank (Textbook Ch. 21)
  - `MAKE-SET`: $O(1)$
  - `FIND-SET`: $O(\log n)$
  - `UNION`: $O(\log n)$
  - The amortized cost of m operations on n elements (Exercise 21.4-4): $O(m \log n)$
- Total complexity: $O(m \log m) = O(m \log n)$

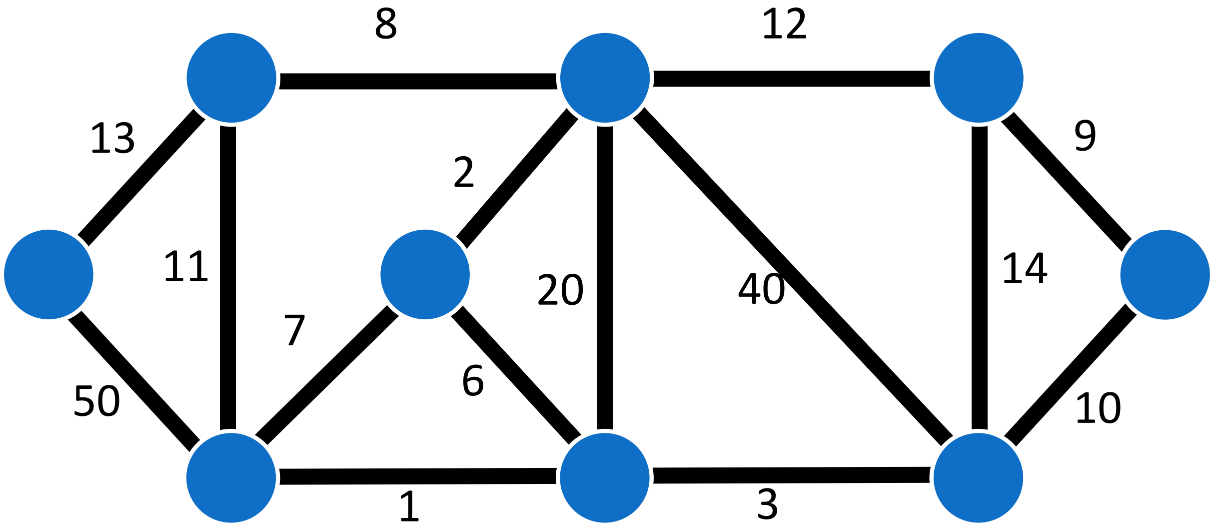
Prim’s Algorithm

Textbook Chapter 23.2 – The algorithms of Kruskal and Prim

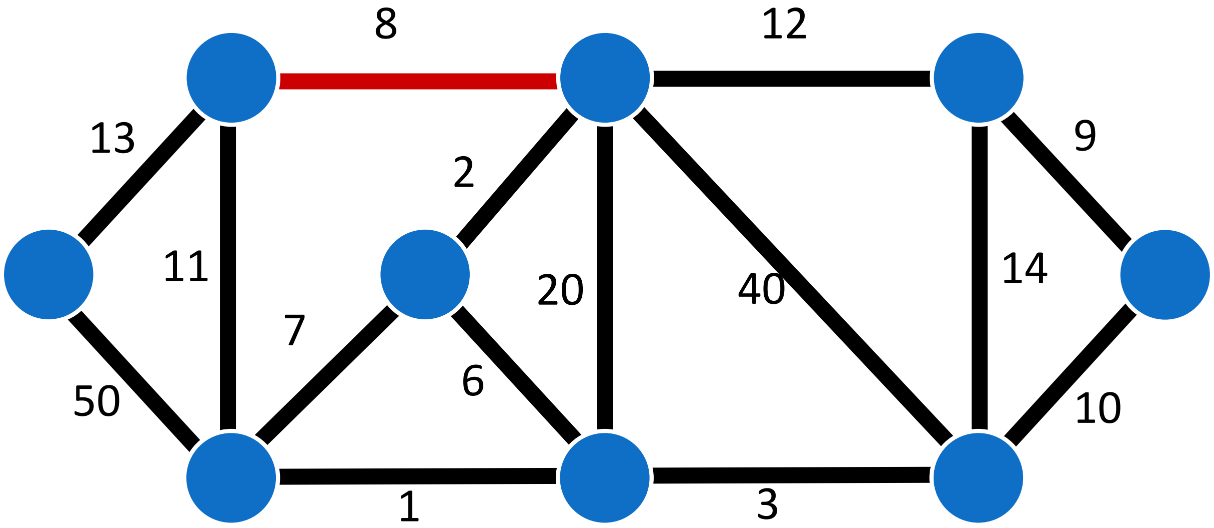
Prim's Algorithm

- Let T consist of an arbitrary node
- For $i = 1$ to $n - 1$
  - add the least-weighted edge incident to the current subtree T that does not incur a cycle

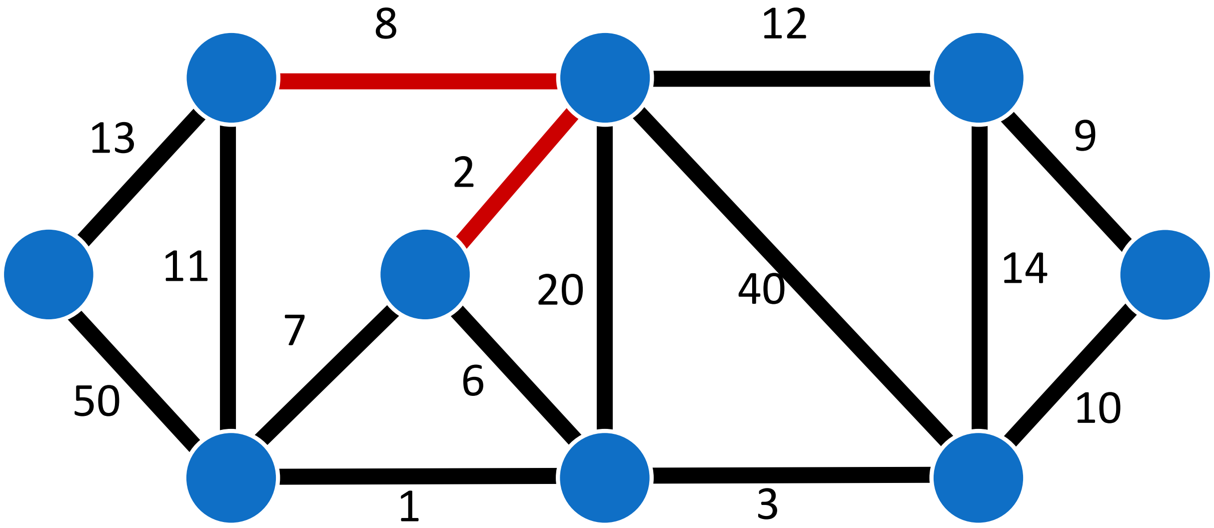
Prim’s Algorithm Illustration



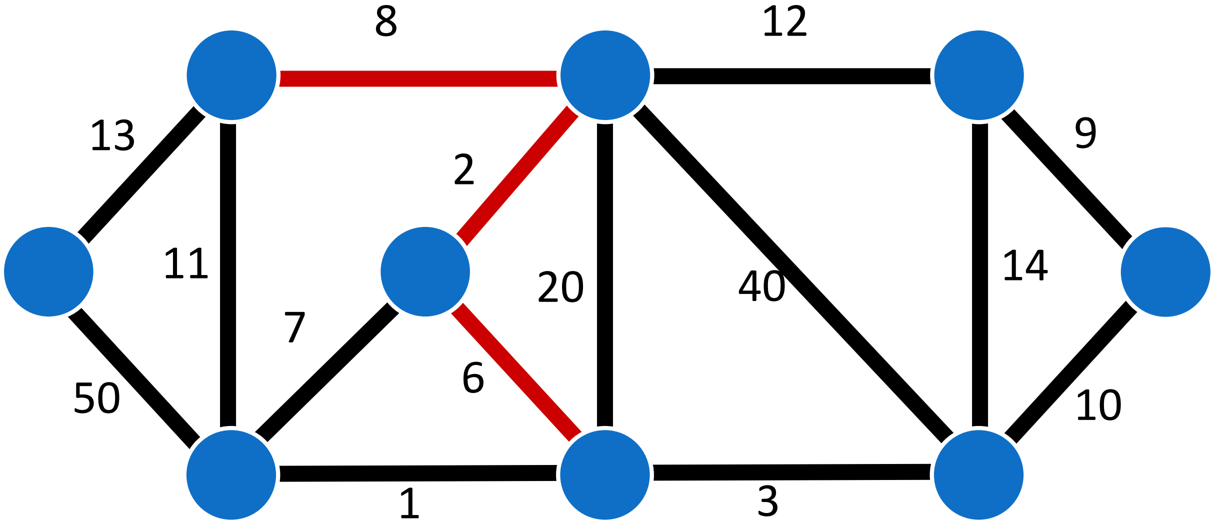
Prim’s Algorithm Illustration



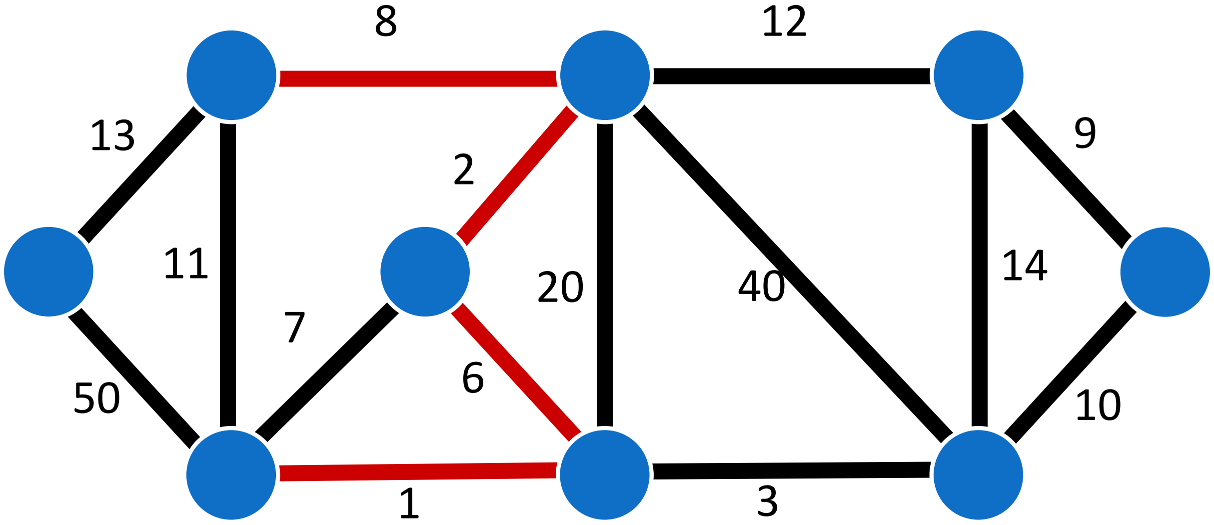
Prim’s Algorithm Illustration



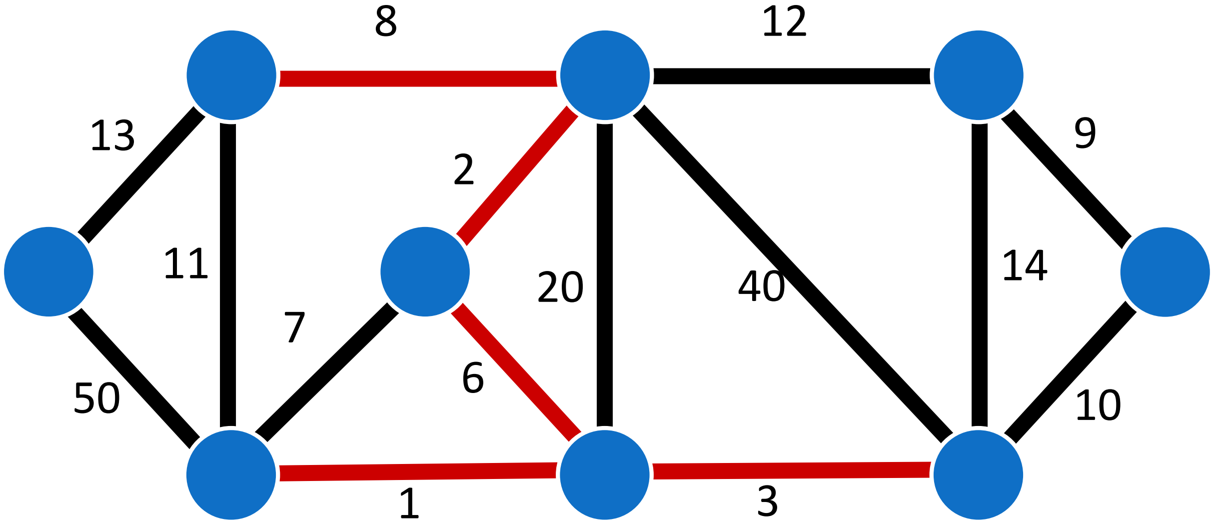
Prim’s Algorithm Illustration



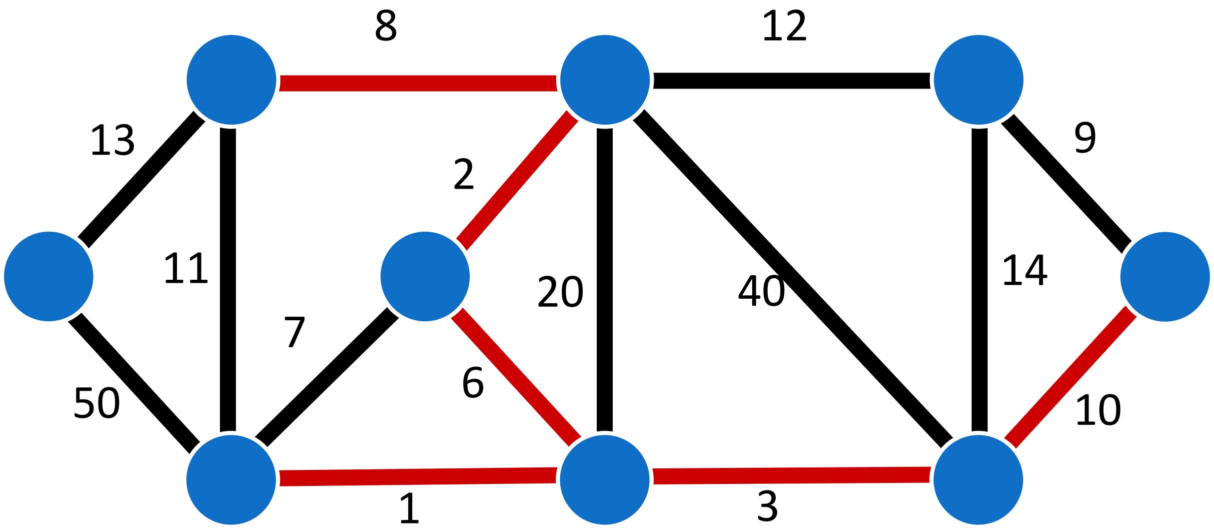
Prim’s Algorithm Illustration



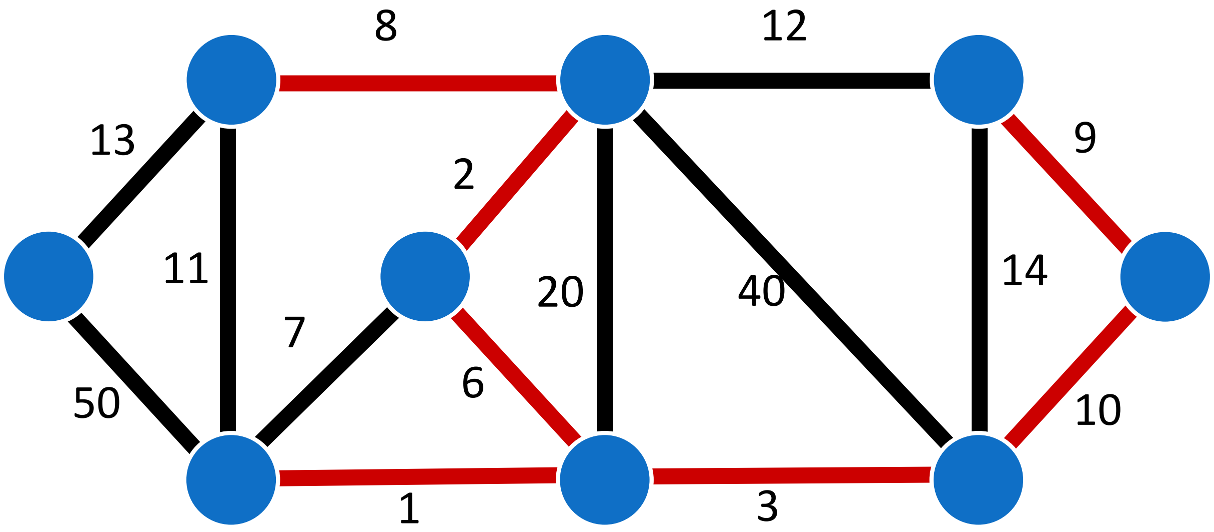
Prim's Algorithm Illustration



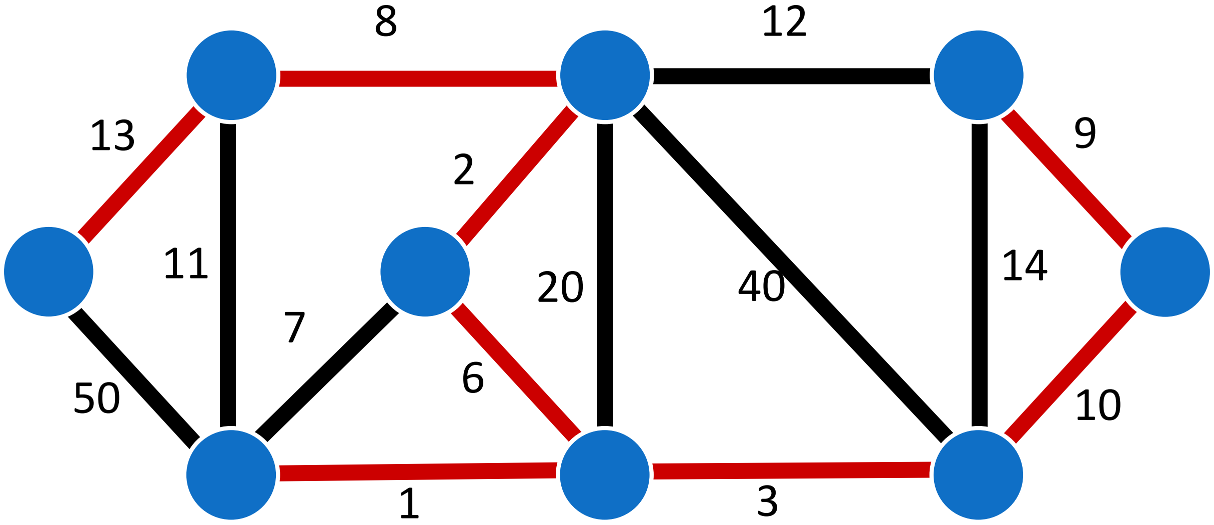
Prim’s Algorithm Illustration



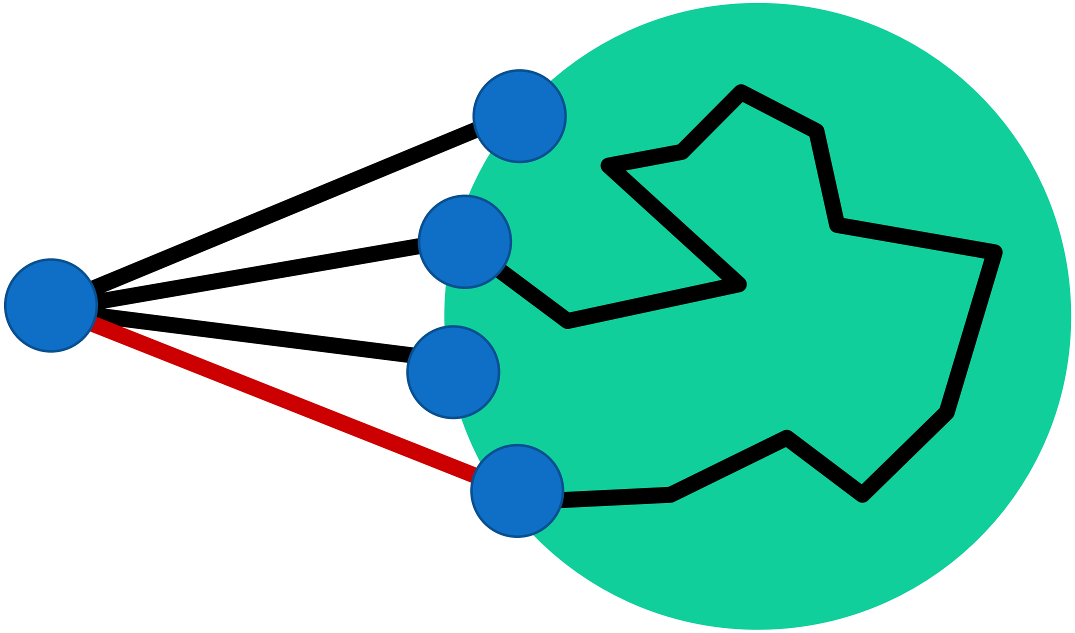
Prim’s Algorithm Illustration



Prim’s Algorithm Illustration



Prim’s Algorithm Correctness



The lightest edge incident to a vertex must be in the MST

Prim’s Time Complexity

```
MST-PRIM(G, w, r) // w = weights, r = root
  for u in G.V                                          ⎫
    u.key = ∞                                           ⎪
    u.π = NIL                                           ⎬ $O(n)$
  r.key = 0                                             ⎪
  Q = G.V                                               ⎭
  while Q ≠ empty                                       $n$ times
    u = EXTRACT-MIN(Q)                                  $O(\log n)$
    for v in G.adj[u]                                   $m$ times
      if v ∈ Q and w(u, v) < v.key
        v.π = u
        v.key = w(u, v) // DECREASE-KEY                 $O(\log n)$
```

- Binary min-heap (Textbook Ch. 6)
  - `BUILD-MIN-HEAP`: $O(n)$
  - `EXTRACT-MIN`: $O(\log n)$
  - `DECREASE-KEY`: $O(\log n)$
- Total complexity: $O(n \log n + m \log n) = O(m \log n)$

Prim’s Time Complexity

```
MST-PRIM(G, w, r) // w = weights, r = root
  for u in G.V
    u.key = ∞
    u.π = NIL                                         $O(n)$
  r.key = 0
  Q = G.V
  while Q ≠ empty                                     $n$ times
    u = EXTRACT-MIN(Q)                                $O(\log n)$
    for v in G.adj[u]                                 $m$ times
      if v ∈ Q and w(u, v) < v.key
        v.π = u
        v.key = w(u, v) // DECREASE-KEY               $O(1)$
```

- Fibonacci heap (Textbook Ch. 19)
  - `BUILD-MIN-HEAP`: $O(n)$
  - `EXTRACT-MIN`: $O(\log n)$ (amortized)
  - `DECREASE-KEY`: $O(1)$ (amortized)
- Total complexity: $O(m + n \log n)$

Concluding Remarks

- Minimal Spanning Trees (MST)
    - Boruvka's Algorithm: $O(m \log n)$
    - Kruskal's Algorithm: $O(m \log n)$
    - Prim's Algorithm: $O(m \log n)$ with binary min-heap
    - Prim's Algorithm: $O(m + n \log n)$ with Fabonacci heap

To Be Continued…

Question?

Important announcement will be sent to @ntu.edu.tw mailbox
& post to the course website

Course Website: http://ada.miulab.tw

Email: ada-ta@csie.ntu.edu.tw