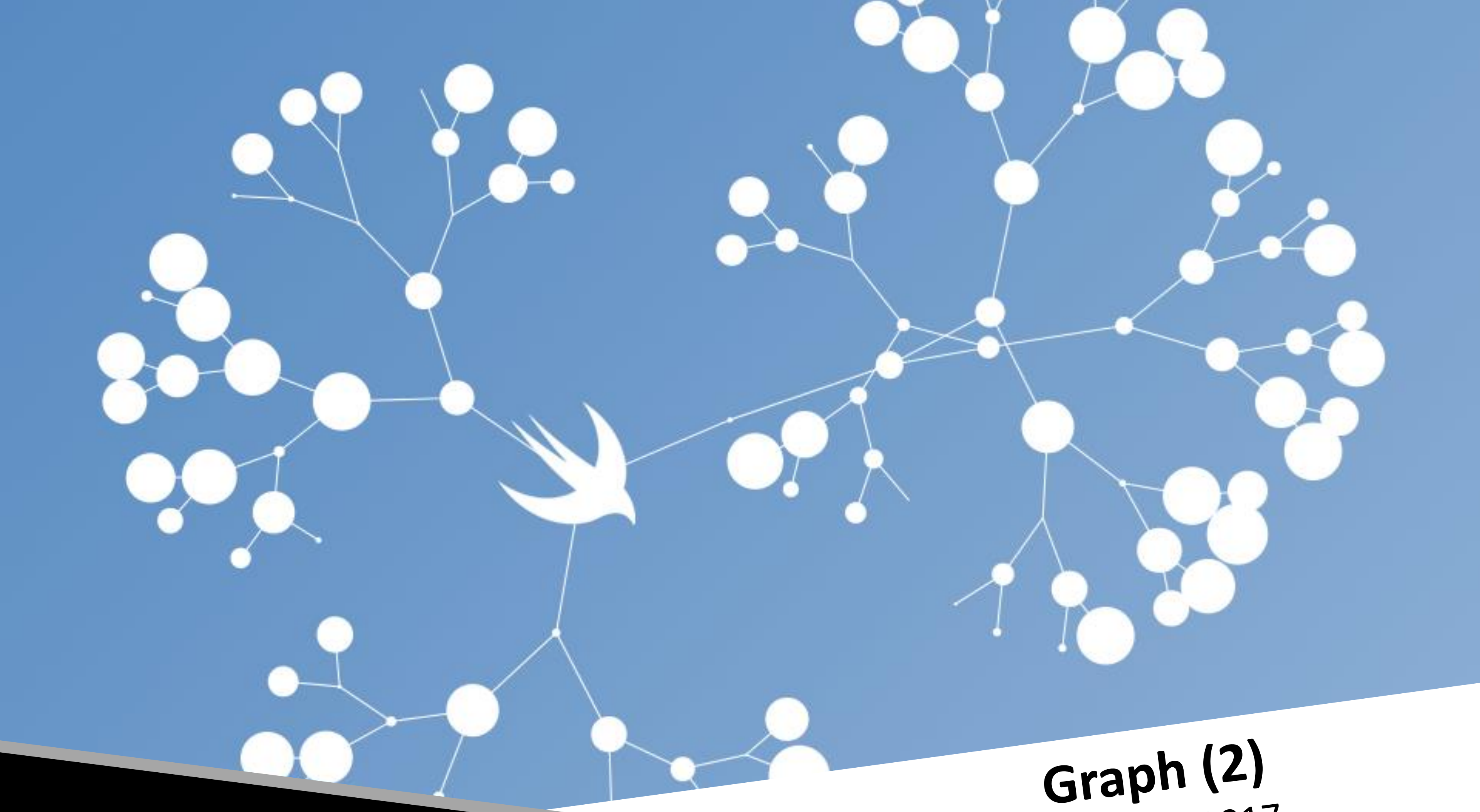


Algorithm Design and Analysis

Graph (2)

Nov 30th, 2017

YUN-NUNG (VIVIAN) CHEN HTTP://ADA17.CSIE.ORG

國立臺灣大學
National Taiwan University

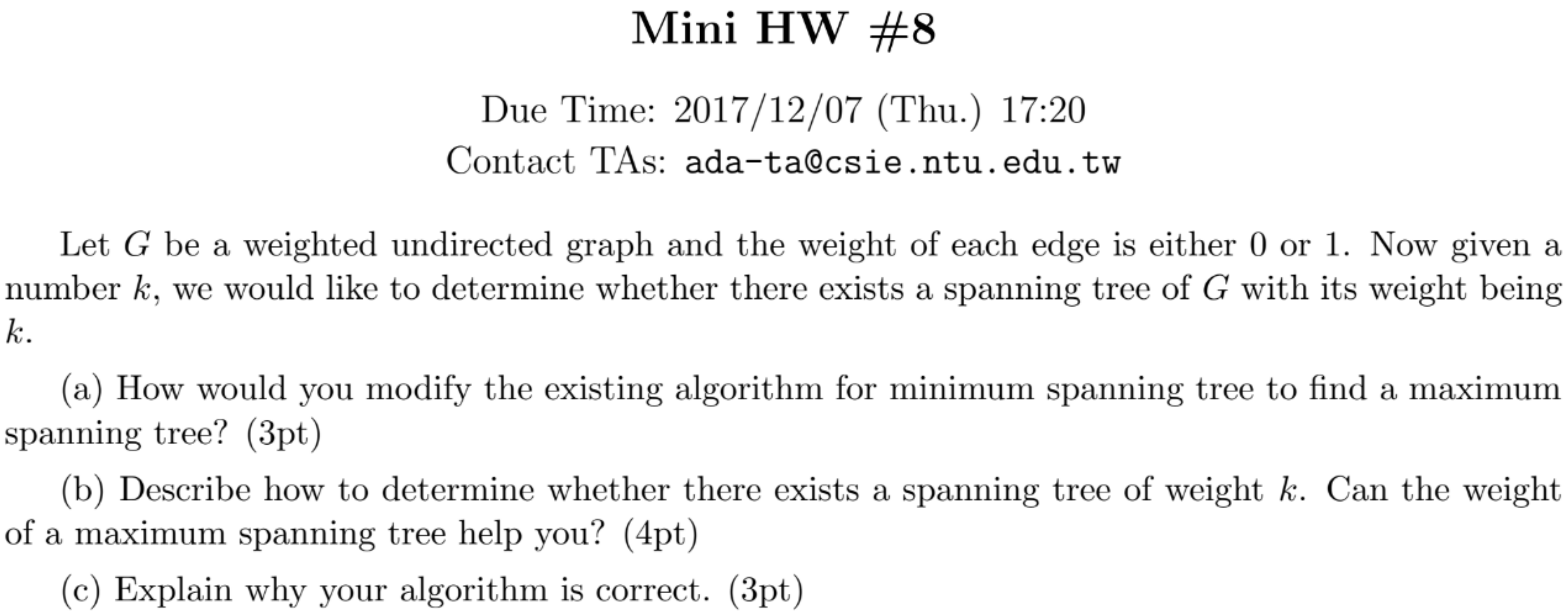
Slides credited from Hsueh-I Lu & Hsu-Chun Hsiao

Announcement

- Mini-HW 8 released
  - Due on 12/07 (Thur) 17:20
- Homework 3 released
  - Due on 12/14 (Thur) 17:20 (two weeks)
- Next-week room changed!!
  - 12/07 (Thur) to forever
  - Location: R103
- Midterm discussion
  - Today 16:30-17:20
  - Location: R103

Frequently check the website for the updated information!

Mini-HW 8



Mini HW #8

Due Time: 2017/12/07 (Thu.) 17:20
Contact TAs: `ada-ta@csie.ntu.edu.tw`

Let G be a weighted undirected graph and the weight of each edge is either 0 or 1. Now given a number k, we would like to determine whether there exists a spanning tree of G with its weight being k.

(a) How would you modify the existing algorithm for minimum spanning tree to find a maximum spanning tree? (3pt)

(b) Describe how to determine whether there exists a spanning tree of weight k. Can the weight of a maximum spanning tree help you? (4pt)

(c) Explain why your algorithm is correct. (3pt)

Outline

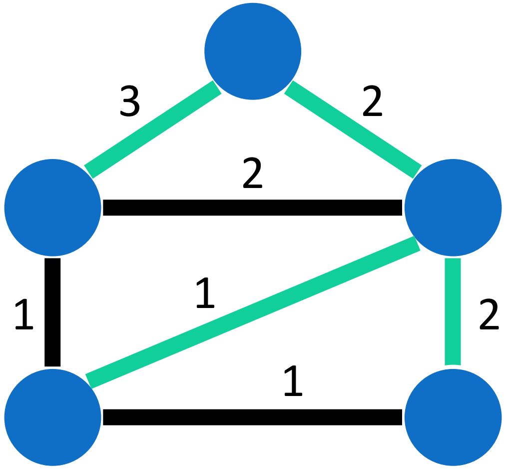
- Minimal Spanning Trees (MST)
  - Boruvka's Algorithm
  - Kruskal's Algorithm
  - Prim's Algorithm
- Single-Source Shortest Paths
  - Bellman-Ford Algorithm
  - Lawler Algorithm (SSSP in DAG)
  - Dijkstra Algorithm

5

Minimal Spanning Tree (MST)

Textbook Chapter 23 – Minimal Spanning Trees

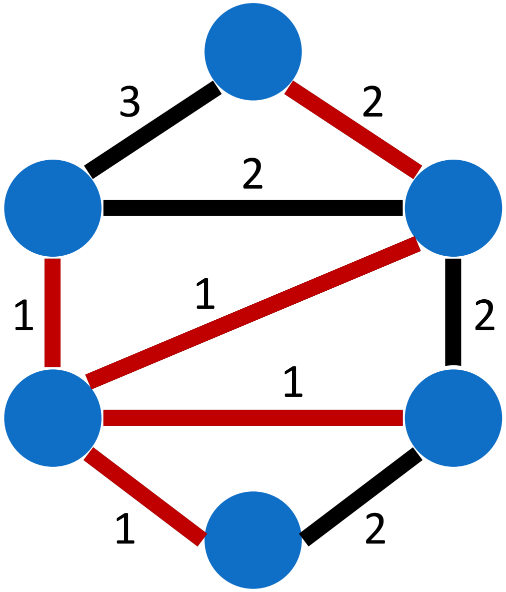
Spanning Tree



- 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

10

Borůvka’s Algorithm

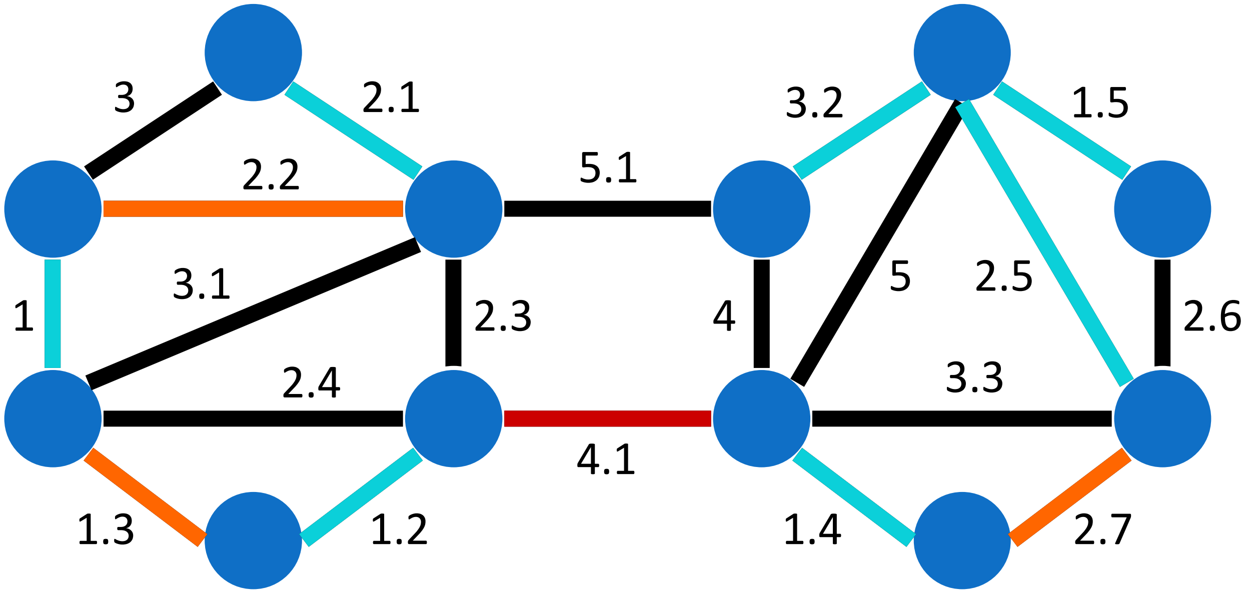
Inventor of MST

- 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

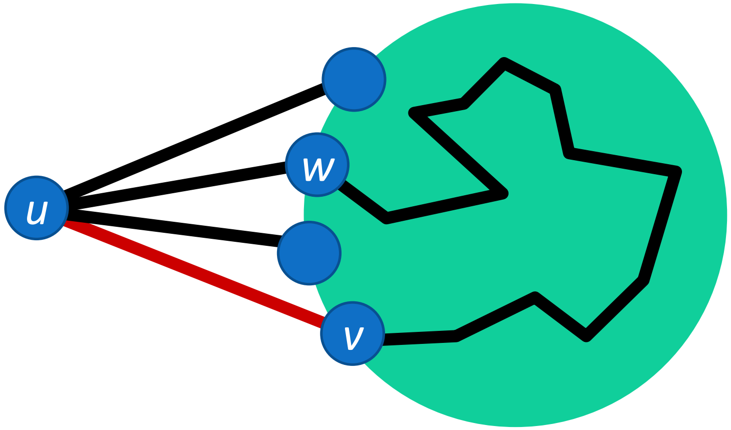
Borůvka’s Algorithm Illustration



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!

18

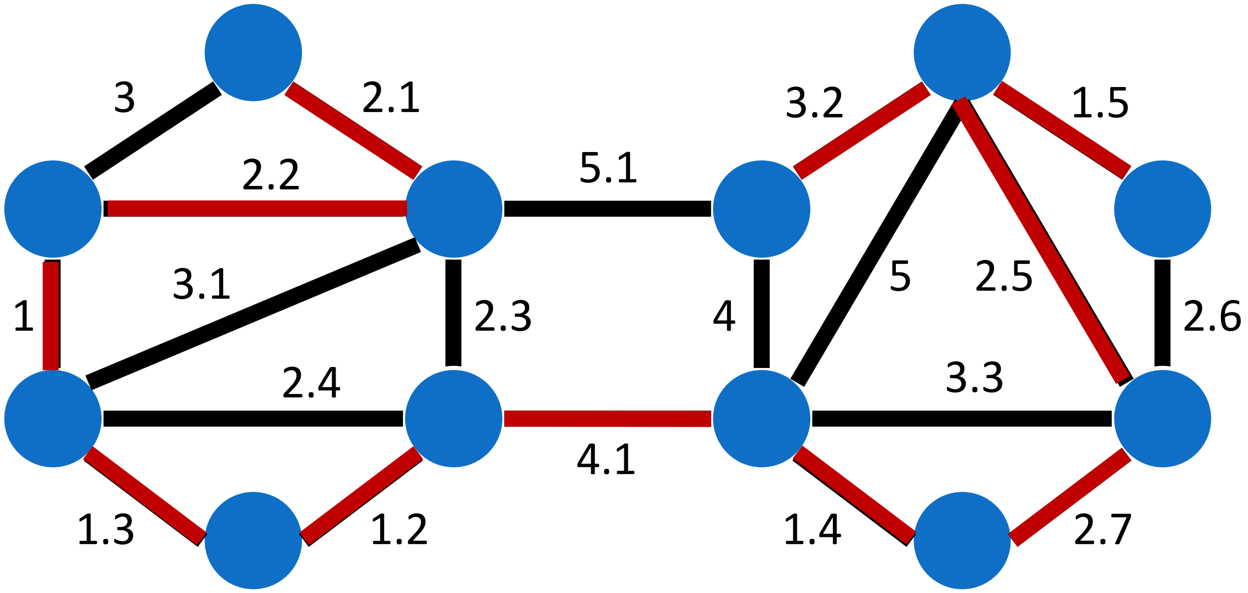
Kruskal's Algorithm

Textbook Chapter 23.2 – The algorithms of Kruskal and Prim

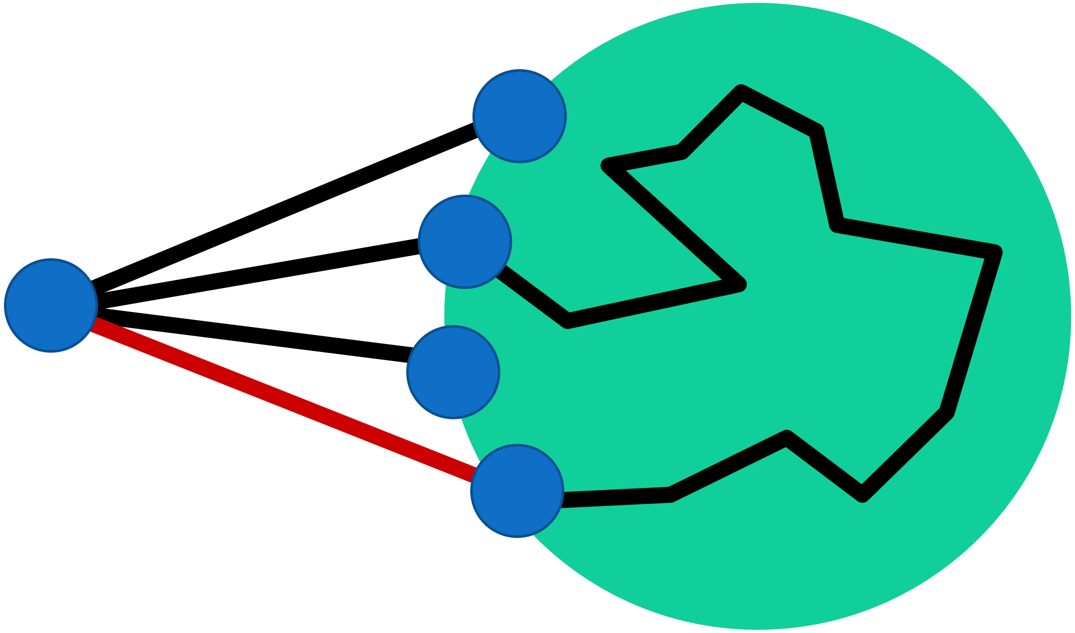
Kruskal’s Algorithm

- 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: $O(m \log m) = O(m \log n)$

24

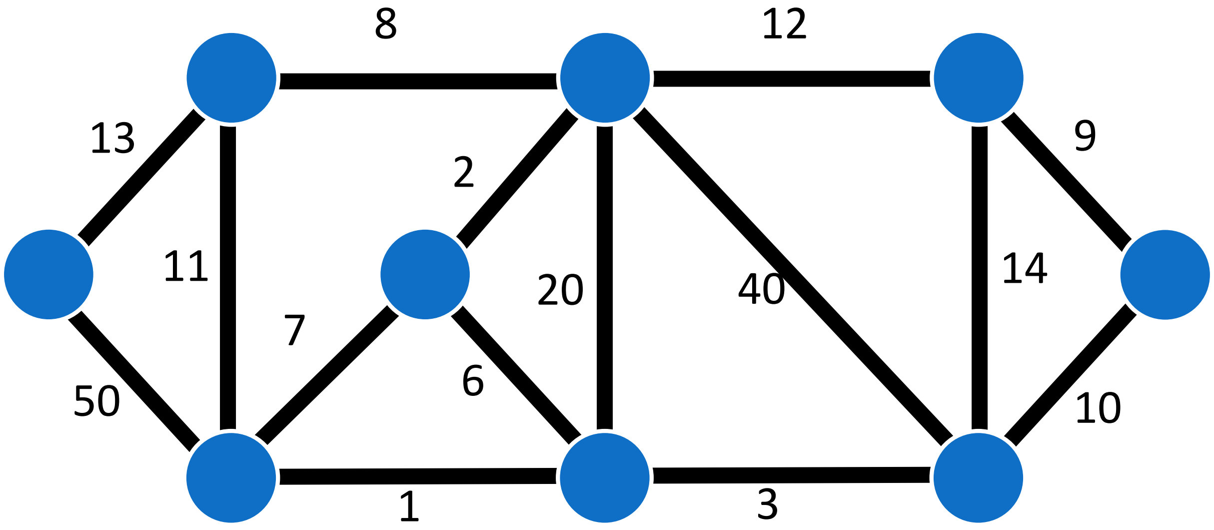
Prim’s Algorithm

Textbook Chapter 23.2 – The algorithms of Kruskal and Prim

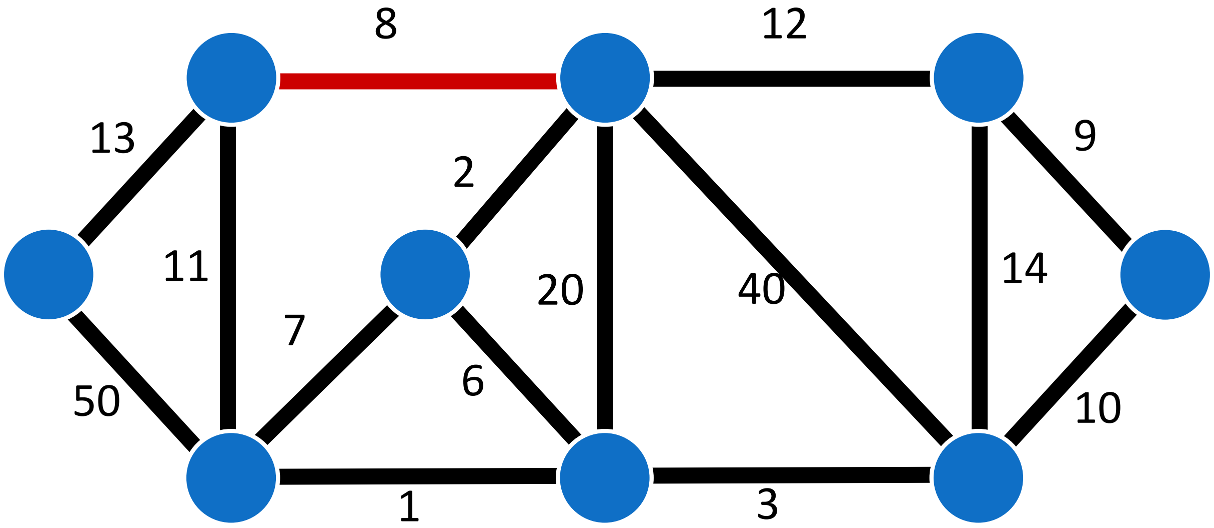
Prim's Algorithm

- 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

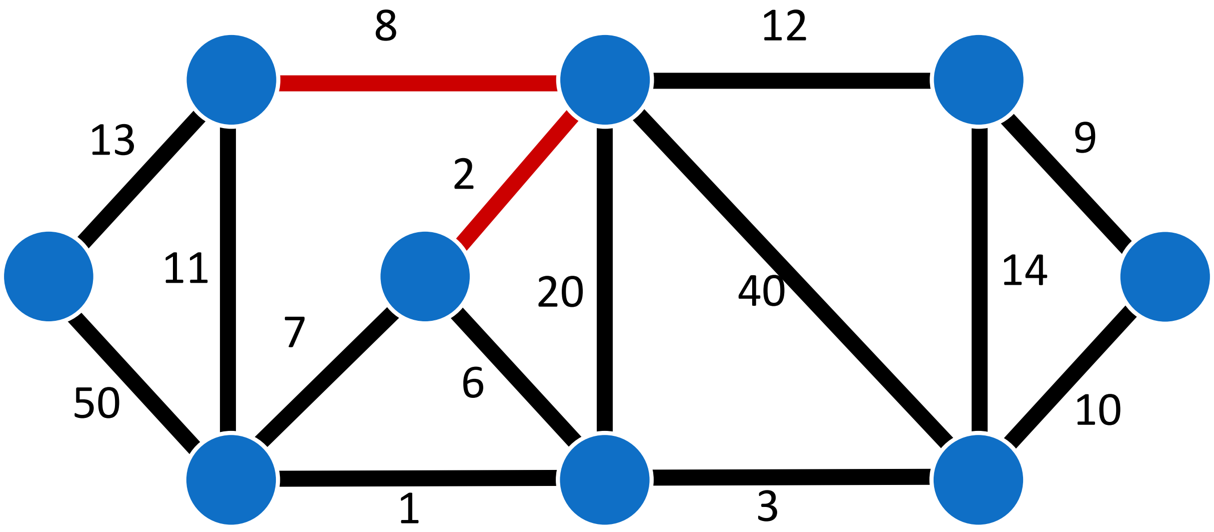
Prim’s Algorithm Illustration



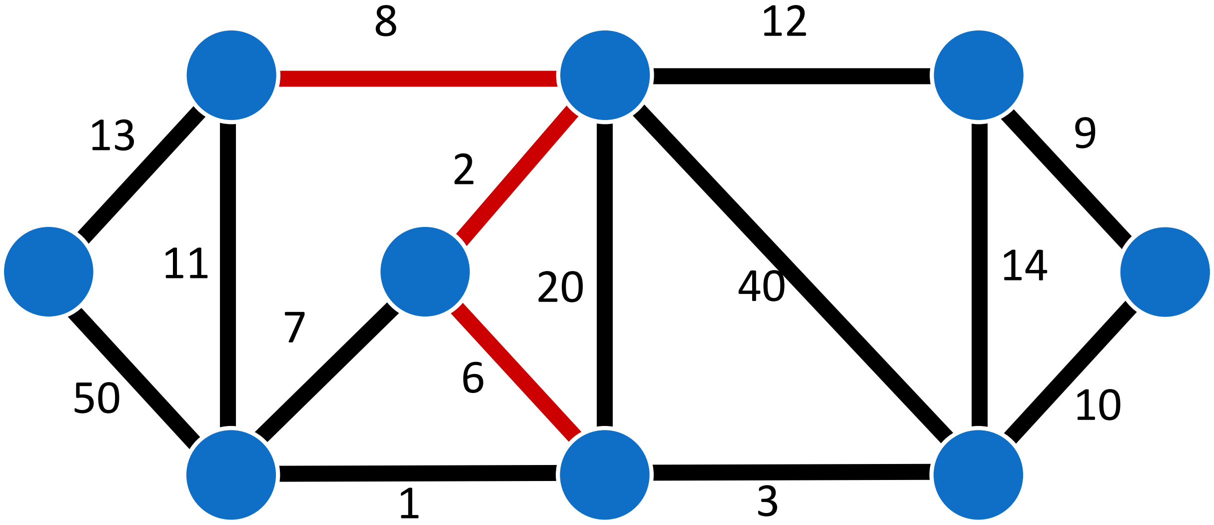
Prim’s Algorithm Illustration



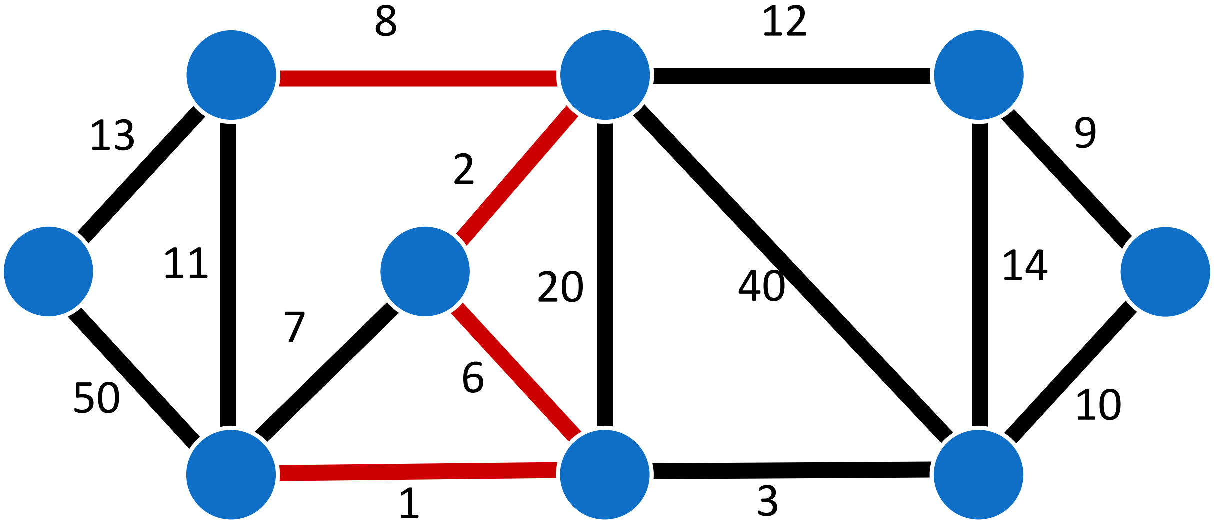
Prim’s Algorithm Illustration



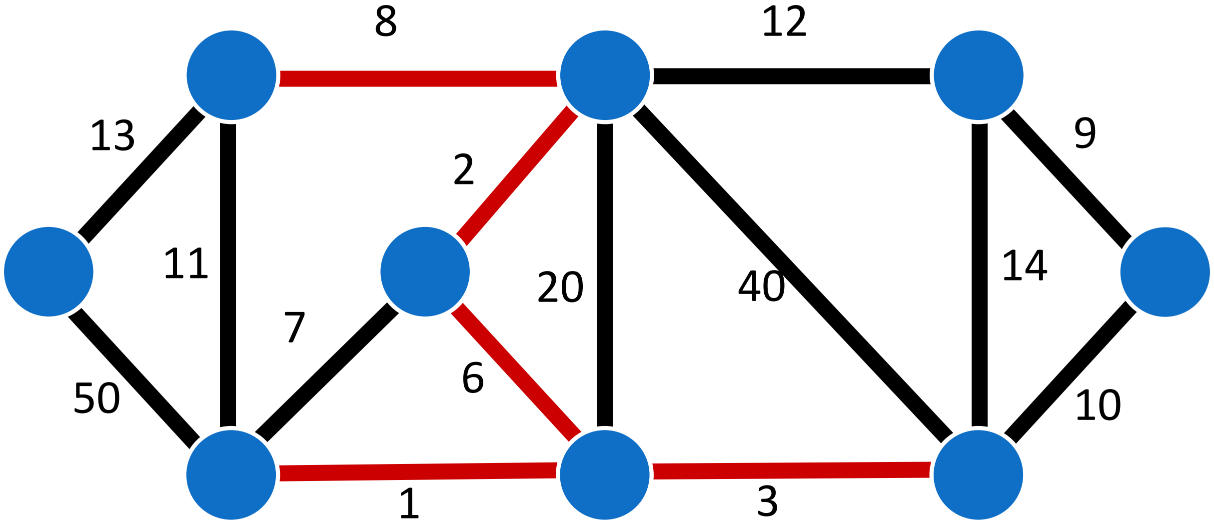
Prim’s Algorithm Illustration



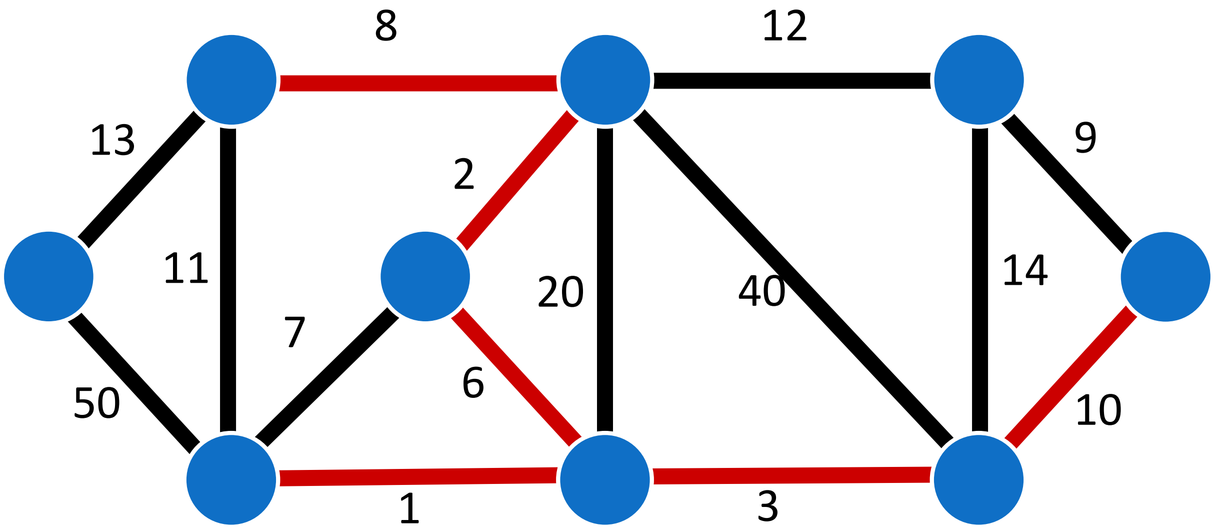
Prim’s Algorithm Illustration



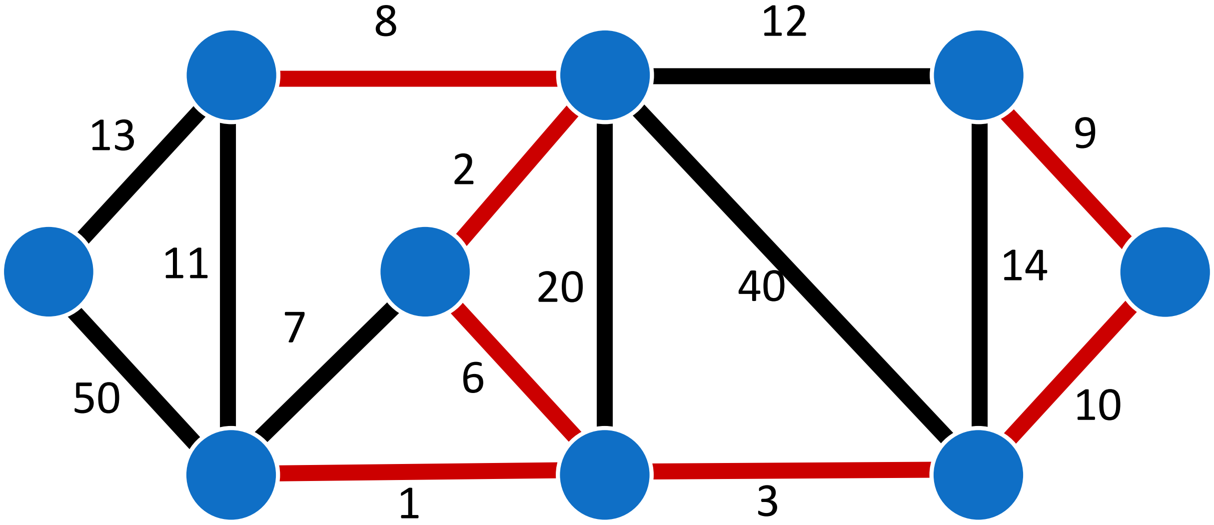
Prim’s Algorithm Illustration



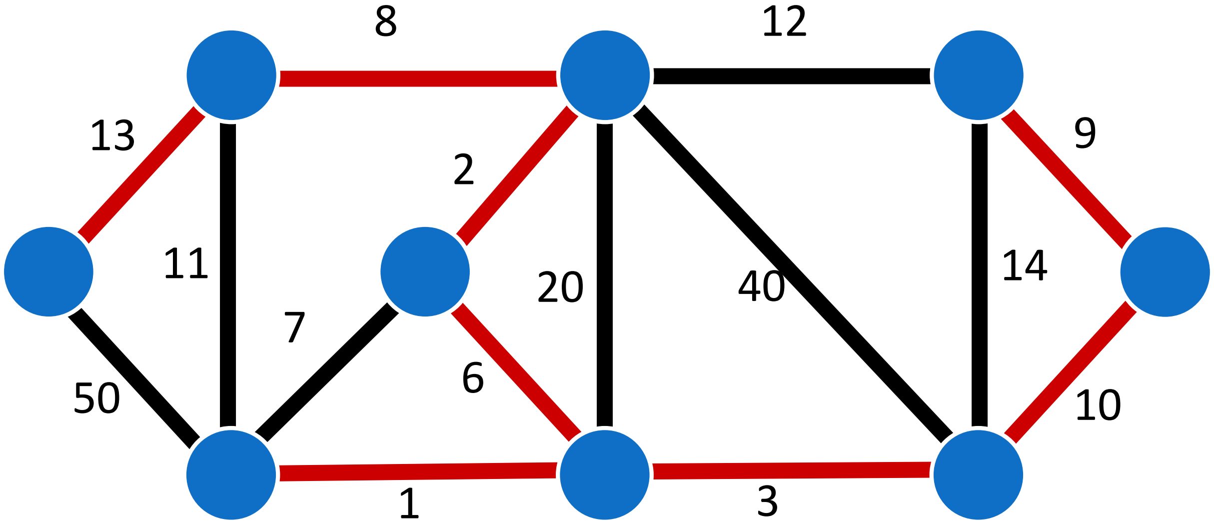
Prim’s Algorithm Illustration



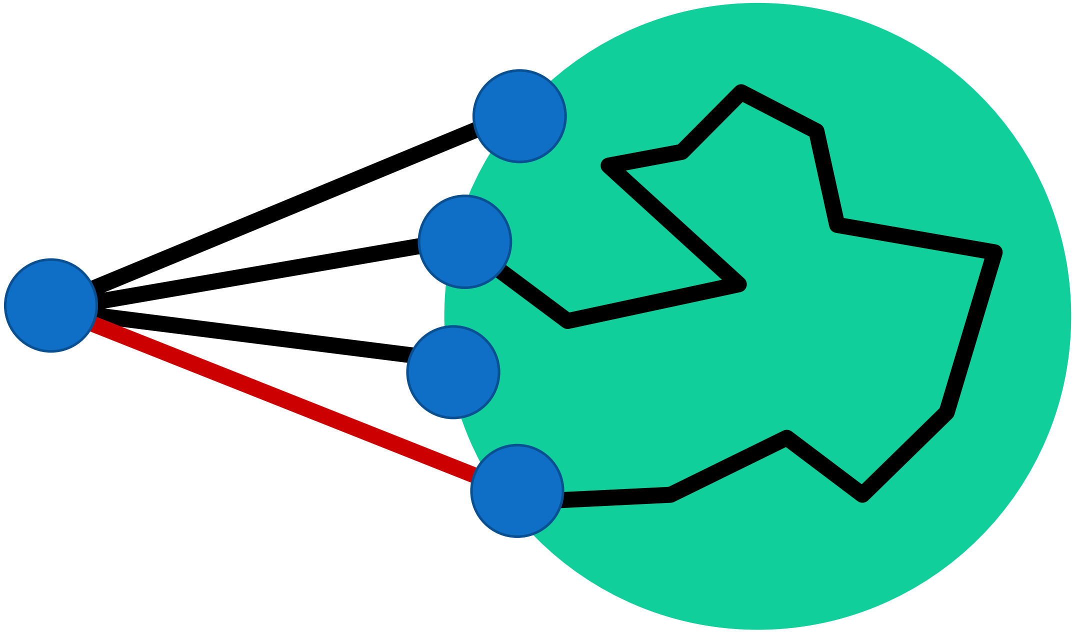
Prim’s Algorithm Illustration



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: $O(m + n \log n)$

Single-Source Shortest Paths

Textbook Chapter 24 – Single-Source Shortest Paths

Shortest Path Problem

- Input: a weighted, directed graph $G = (V, E)$
  - Weights can be arbitrary numbers, not necessarily distance
  - Weight function needs not satisfy triangle inequality
- Output: a minimal-cost path from s to t s.t. $\delta(s, t)$ is the minimum weight from s to t

- Problem Variants
  - Single-source shortest-path problem
  - Single-destination shortest-path problem
  - Single-pair shortest-path problem
  - All-pair shortest path problem

Cycles in Graph

- Can a shortest path contain a negative-weight edge?

  Yes.

- Can a shortest path contain a negative-weight cycle?

  Doesn’t make sense.

- Can a shortest path contain a cycle?

  No.

Single-Source Shortest Path Problem

- Input: a weighted, directed graph $G = (V, E)$ and a source vertex s
- Output: a minimal-cost path from s to t, where $t \in V$

Shortest Path Tree

- Let $G = (V, E)$ be a weighted, directed graph with no negative-weight cycles reachable from s
- A shortest path tree $G' = (V', E')$ of s is a subgraph of G s.t.
  - V' is the set of vertices reachable from s in G
  - G' forms a rooted tree with root s
  - For all $v \in V'$, the unique simple path from s to v in G' is a shortest path from s to v in G

Shortest Path Tree Problem

- Input: a weighted, directed graph $G = (V, E)$ and a vertex s
- Output: a tree T rooted at s s.t. the path from s to u of T is a shortest path from s to u in G

Problem Equivalence

- The shortest path tree problem is equivalent to finding the minimal cost $\delta(s, u)$ from s to each node u in G
  - The minimal cost from s to u in G is the length of any shortest path from s to u in G

"equivalence": a solution to either problem can be obtained from a solution to the other problem in linear time

Shortest Path Tree Problem = Single-Source Shortest Path Problem

Bellman-Ford Algorithm

Textbook Chapter 24.1 – The Bellman-Ford algorithm

Bellman and Ford

Richard Bellman, 1920~1984

- Norbert Wiener Prize in Applied Mathematics, 1970
- Dickson Prize, Carnegie-Mellon University, 1970
- John von Neumann Theory Award, 1976.
- IEEE Medal of Honor, 1979,
- Fellow of the American Academy of Arts and Sciences, 1975.
- Membership in the National Academy of Engineering, 1977

Lester R. Ford, Jr. 1927~2017

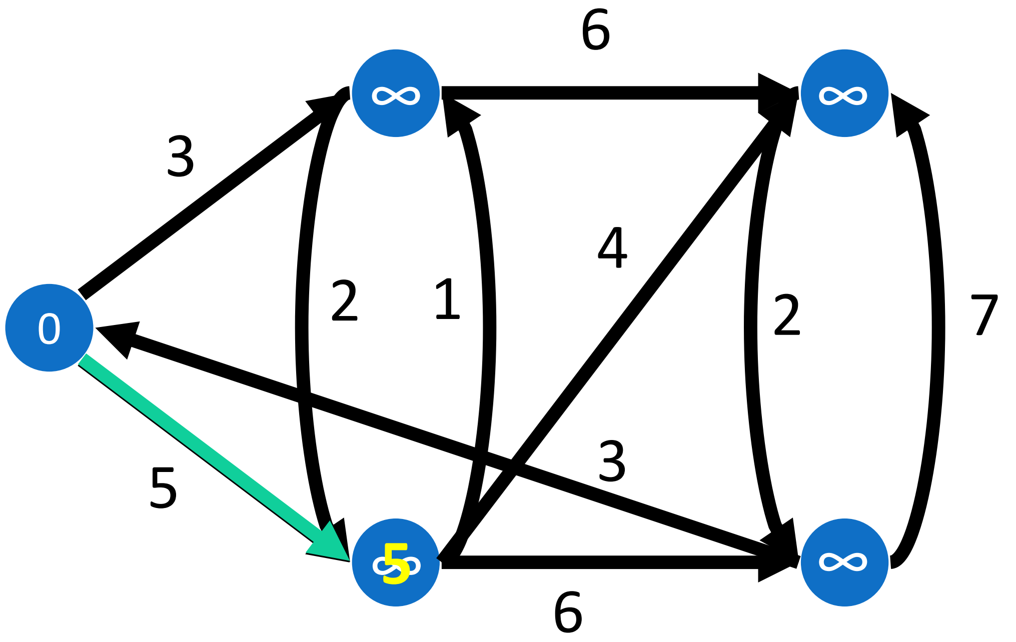
- A important contributor to the theory of network flow.
  - We will learn Ford and Fulkerson’s maximum flow algorithm in a couple of weeks.

Bellman-Ford Algorithm

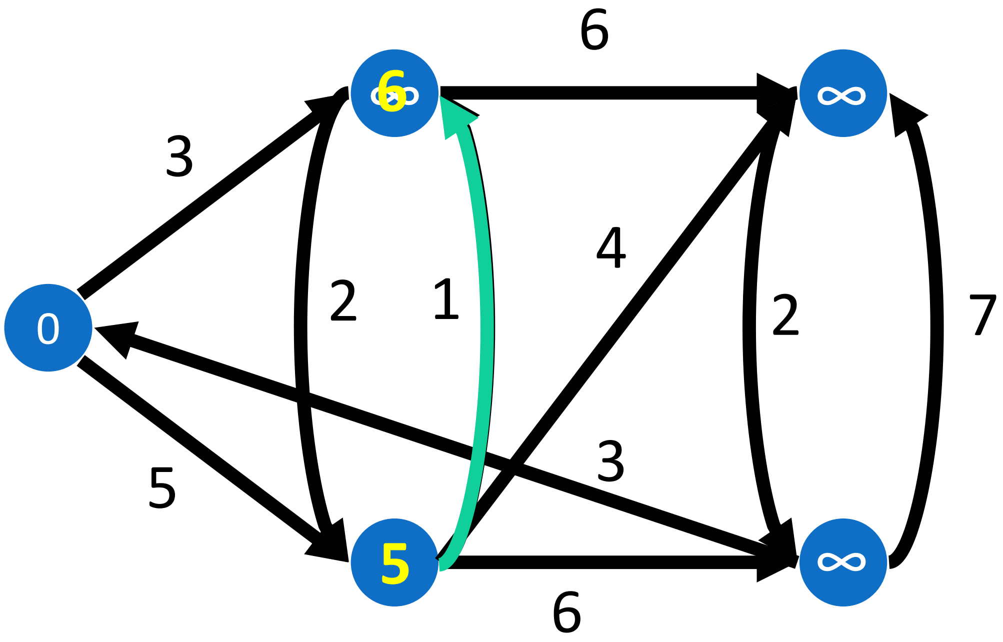
- Idea: estimate the value of $d[u]$ to approximate $\delta(s, u)$
- Initialization
  - Let $d[u] = \infty$ for $u \in G$
  - Let $d[s] = 0$
- Repeat the following step for sufficient number of phases
  - For each edge $(u, v) \in E$, relax edge (u, v)
  - Relaxing: If $d[v] > d[u] + w(u, v)$, let $d[v] = d[u] + w(u, v)$

→ improve the estimation of $d[u]$

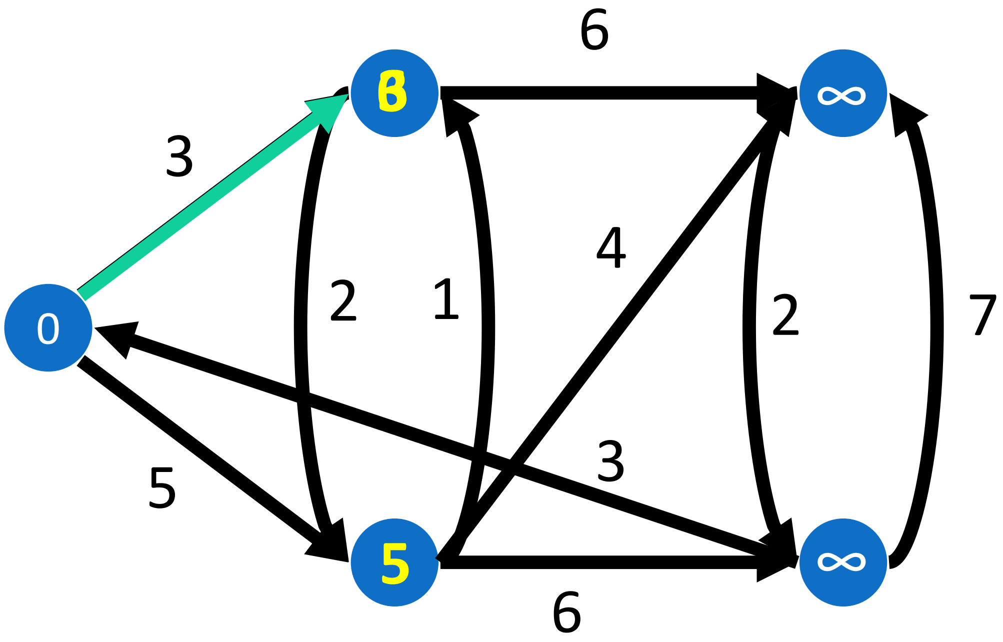
Bellman-Ford Algorithm Illustration



Bellman-Ford Algorithm Illustration

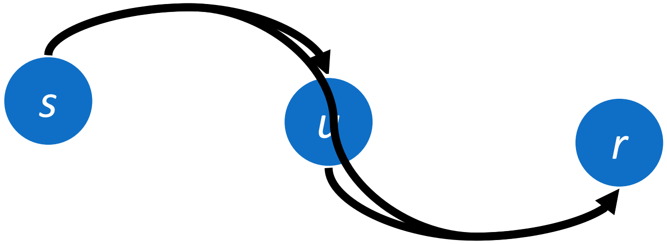


Bellman-Ford Algorithm Illustration



Bellman-Ford Algorithm Correctness

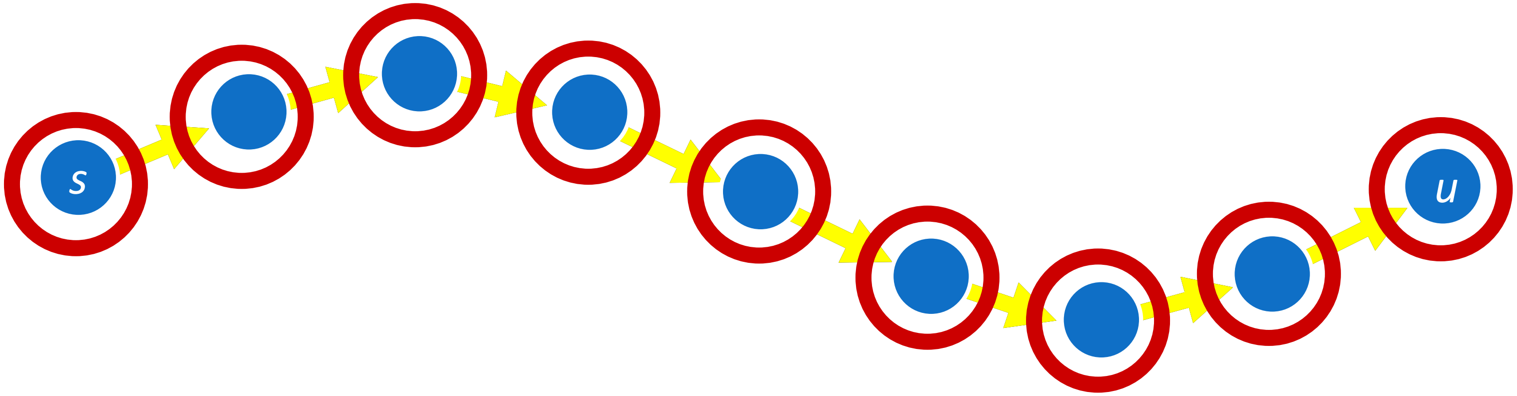
- Observation: let P be a shortest path from s to r
  - For any vertex u in P, the subpath of P from s to u has to be a shortest path from s to u → optimal substructure
  - For any edge (u, v) in P, if $d[u] = \delta(s, u)$, then $d[v] = \delta(s, v)$ also holds after relaxing edge (u, v)



- If G contains no negative cycles, then each node u has a shortest path from s to u that has at most *n* – 1 edges
- From observation, after the first i phases of improvement via relaxation, the estimation of $d[u]$ for the first $i + 1$ nodes u in the path is precise ($= \delta(s, u)$)

→ $n - 1$ phases

Bellman-Ford Algorithm Correctness



Bellman-Ford Time Complexity

```
BELLMAN-FORD(G, w, s)
  INITIALIZATION(G, s)
  for i = 1 to |G.V| - 1        $n-1$ times
    for (u, v) in G.E           $O(m)$
      RELAX(u, v, w)
```

```
INITIALIZATION(G, s)
  for v in G.V        ┐
    v.d = ∞           │
    v.π = NIL         ├ $O(n)$
  s.d = 0             ┘
```

```
RELAX(u, v, w)                  $O(1)$
  if v.d > u.d + w(u, v)
    // DECREASE-KEY
    v.d = u.d + w(u, v)
    v.π = u
```

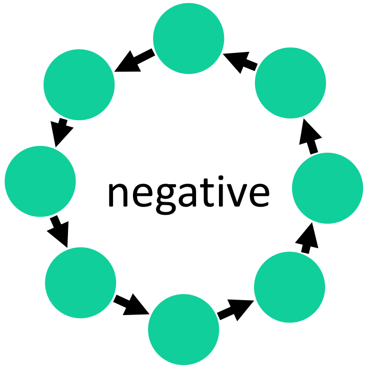
- Time complexity: $O(mn)$

How to do if there is a negative cycle in the graph?

Negative Cycle Detection

- Q: How do we know G has negative cycles?
- A: Using another phase of improvement via relaxation
  - Run another phase of improving the estimation of $d[u]$ for each vertex $u \in V$ via relaxing all edges E
  - If in the n-th phase, there are still some $d[u]$ being modified, we know that G has negative cycles

Negative Cycle Detection



If there exists a negative cycle in G, in the n-th phase, there are still some $d[u]$ being modified.

- Proof by contradiction
    - Let C be a negative cycle of k nodes $v_1, v_2, \dots, v_k$ $(v_{k+1} = v_1)$
    - Assume $d[v_i]$ for all $1 \leq i \leq k$ are not changed in a phase of improvement, then for $1 \leq i \leq k$

$$d[v_{i+1}] \leq d[v_i] + w(v_i, v_{i+1})$$

- Summing all k inequalities, the sum of edge weights of C is nonnegative

$$\sum_{i=1}^{k} d[v_{i+1}] \leq \sum_{i=1}^{k} d[v_i] + \sum_{i=1}^{k} w(v_i, v_{i+1}) \Rightarrow 0 \leq \sum_{i=1}^{k} w(v_i, v_{i+1})$$

Bellman-Ford Algorithm

```
BELLMAN-FORD(G, w, s)
  INITIALIZATION(G, s)
  for i = 1 to |G.V| - 1          $n-1$ times
    for (u, v) in G.E             $O(m)$
      RELAX(u, v, w)
  for (u, v) in G.E
    if v.d > u.d + w(u, v)
      return FALSE
  return TRUE     negative cycle detection
```

```
INITIALIZATION(G, s)
  for v in G.V
    v.d = ∞                 $O(n)$
    v.π = NIL
  s.d = 0
```

```
RELAX(u, v, w)                        $O(1)$
  if v.d > u.d + w(u, v)
    // DECREASE-KEY
    v.d = u.d + w(u, v)
    v.π = u
```

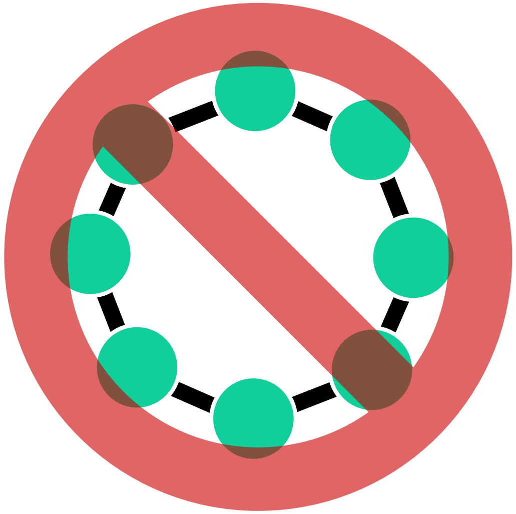
- Time complexity: $O(mn)$
- Finding a shortest-path tree of G: $O(mn) + O(m+n) = O(mn)$

Lawler Algorithm

Textbook Chapter 24.2 – Single-source shortest paths in directed acyclic graphs

Single-Source Shortest Path Problem

- Input: a weighted, directed, and acyclic graph $G = (V, E)$ and a source vertex s
- Output: a shortest-path distance from s to t, where $t \in V$

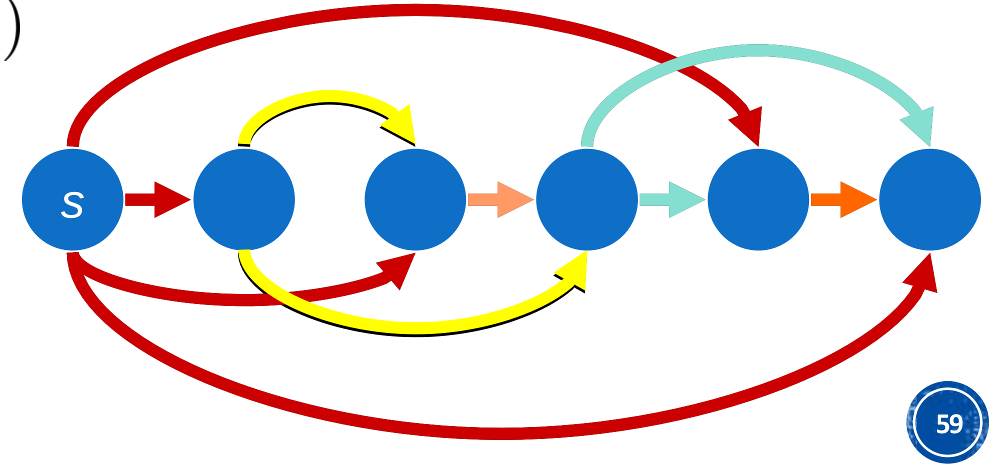


No negative cycle!

Lawler Algorithm

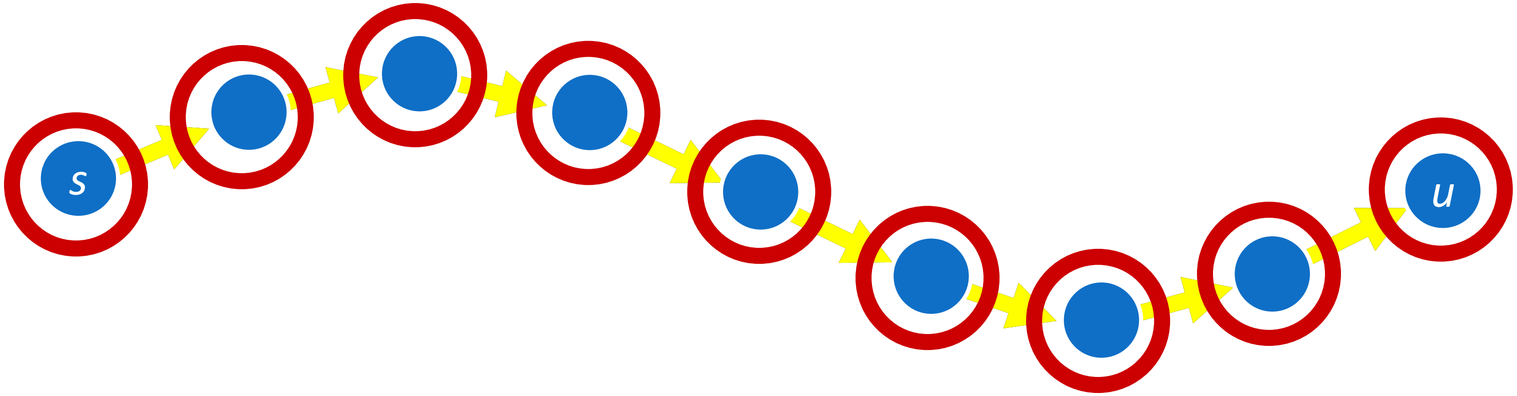
- Idea: one phase relaxation
- Perform a topological sort in linear time on the input DAG
- For $i = 1$ to n
  - Let v_i be the i-th node in the above order
  - Relax each outgoing edge (v_i, u) from v_i

Time complexity: $O(m+n)$



Lawler Algorithm Correctness

- Assume this is a shortest path from s to u
- If we follow the order from topological sort to relax the vertices’ edges, in this shortest path, the left edge must be relaxed before the right edge
- One phase of improvement is enough

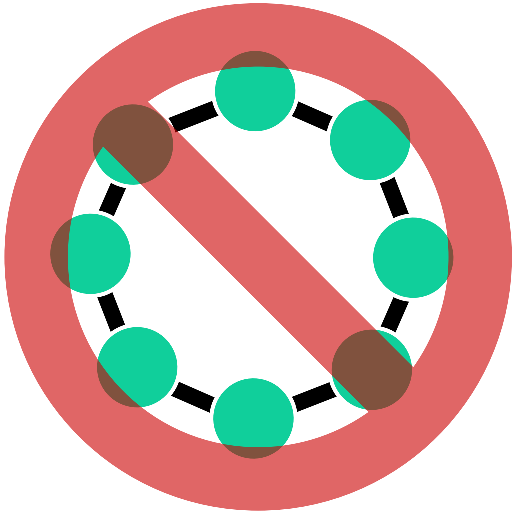


Dijkstra’s Algorithm

Textbook Chapter 24.3 – Dijkstra’s algorithm

Single-Source Shortest Path Problem

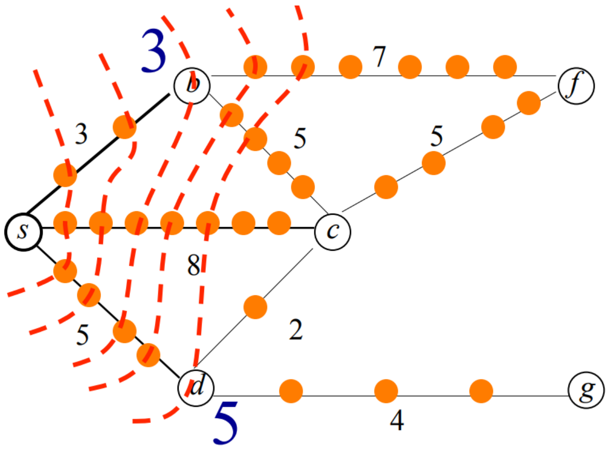
- Input: a non-negative weighted, directed, graph $G = (V, E)$ and a source vertex s
- Output: a shortest-path distance from s to t, where $t \in V$



No negative cycle!

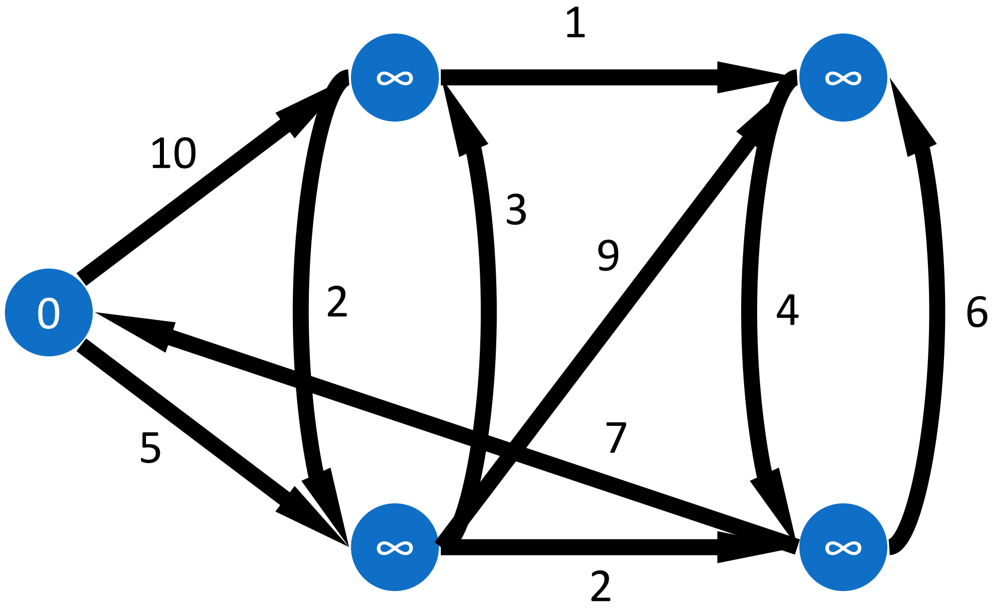
Dijkstra’s Algorithm

- Idea: BFS finds shortest paths on unweighted graph by expanding the search frontier

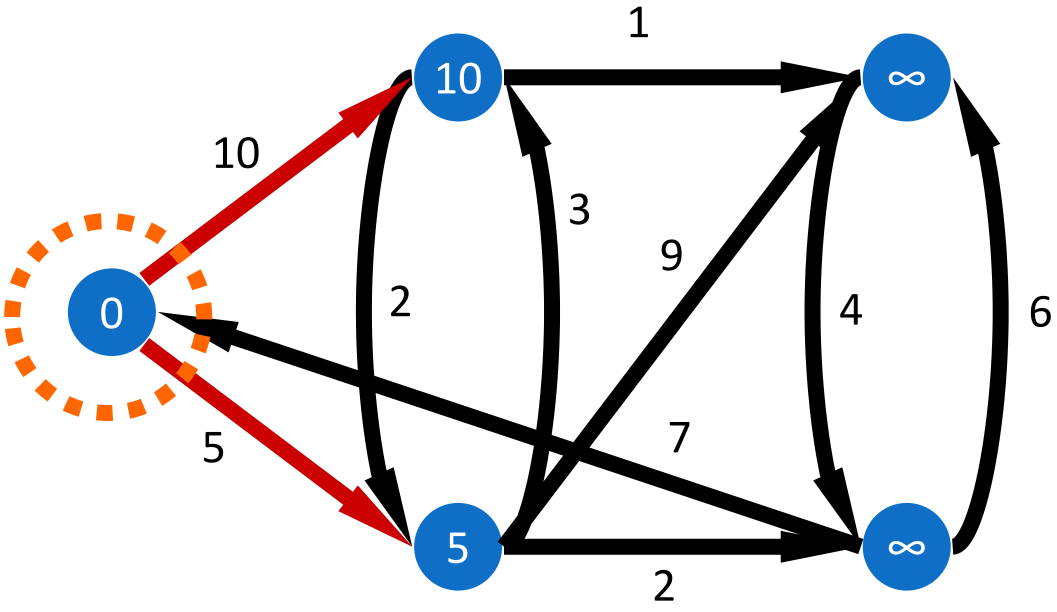


- Initialization
- Loops for n iterations, where each iteration
  - relax outgoing edges of an unprocessed node u with minimal $d[u]$
  - marks u as processed

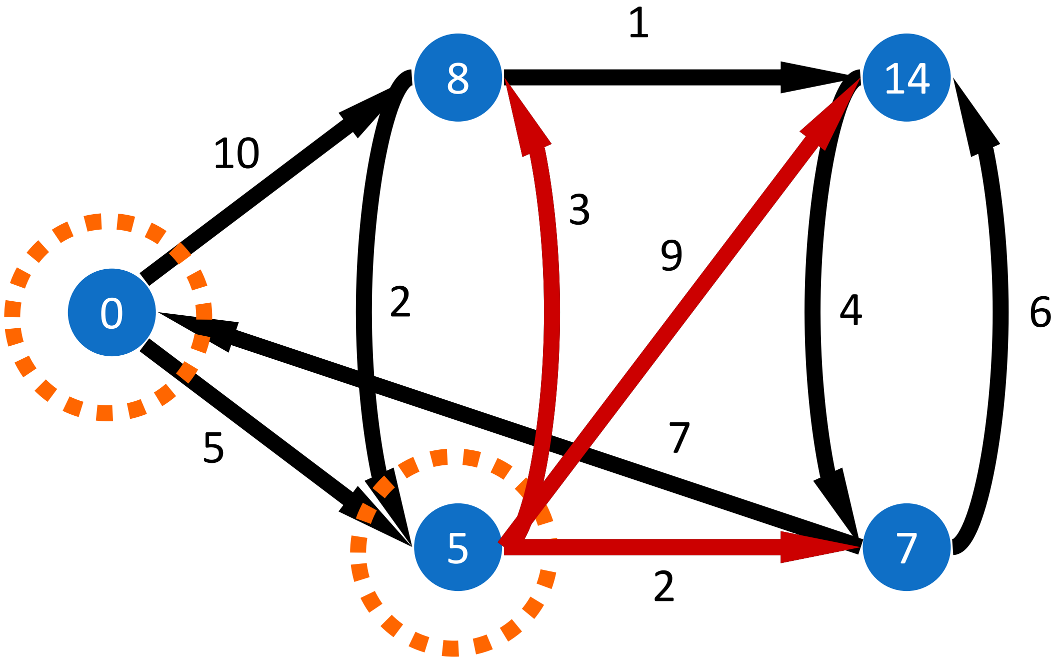
Dijkstra's Algorithm Illustration



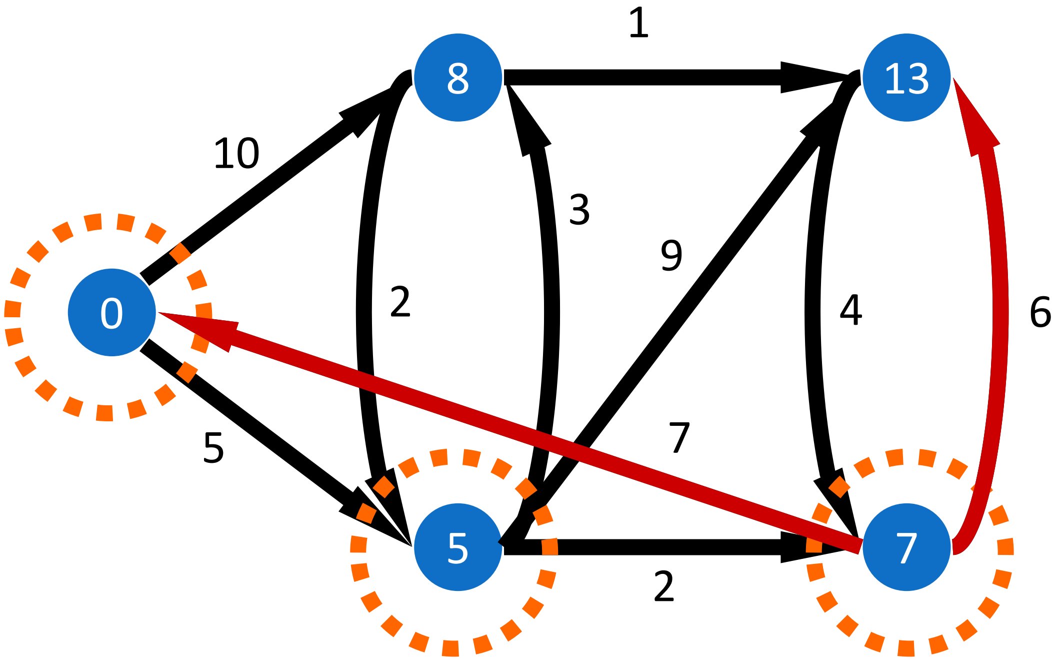
Dijkstra’s Algorithm Illustration



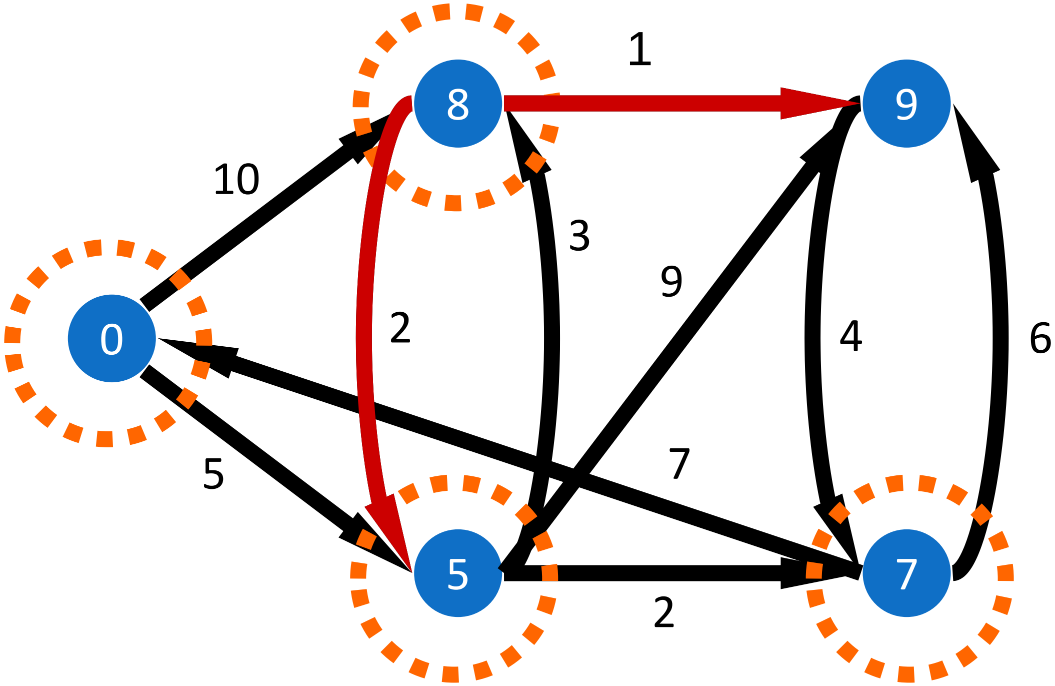
Dijkstra's Algorithm Illustration



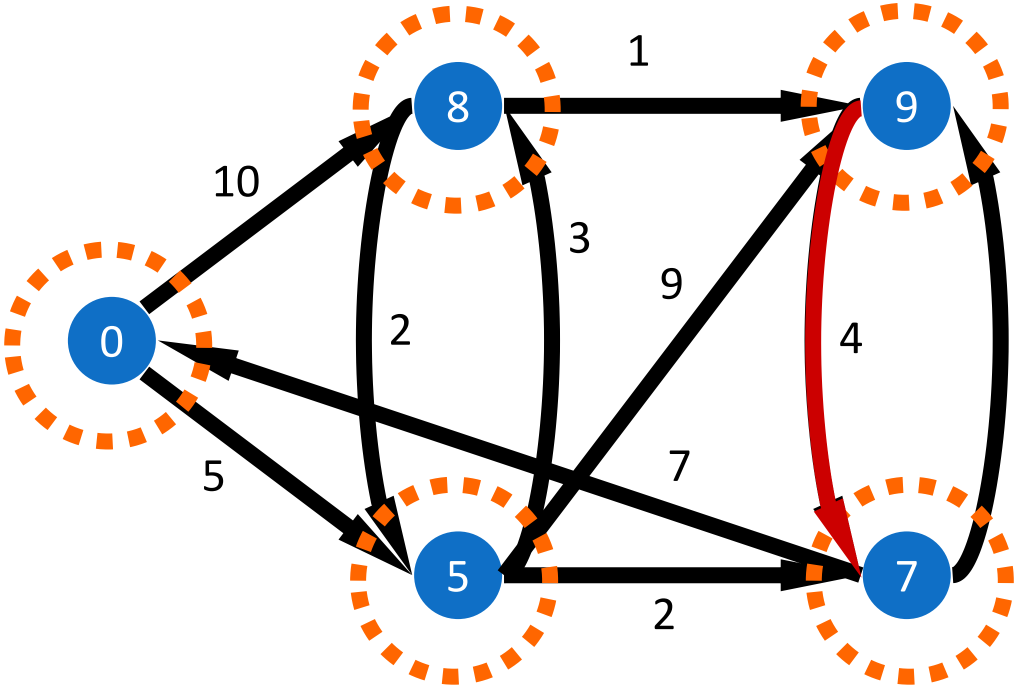
Dijkstra’s Algorithm Illustration



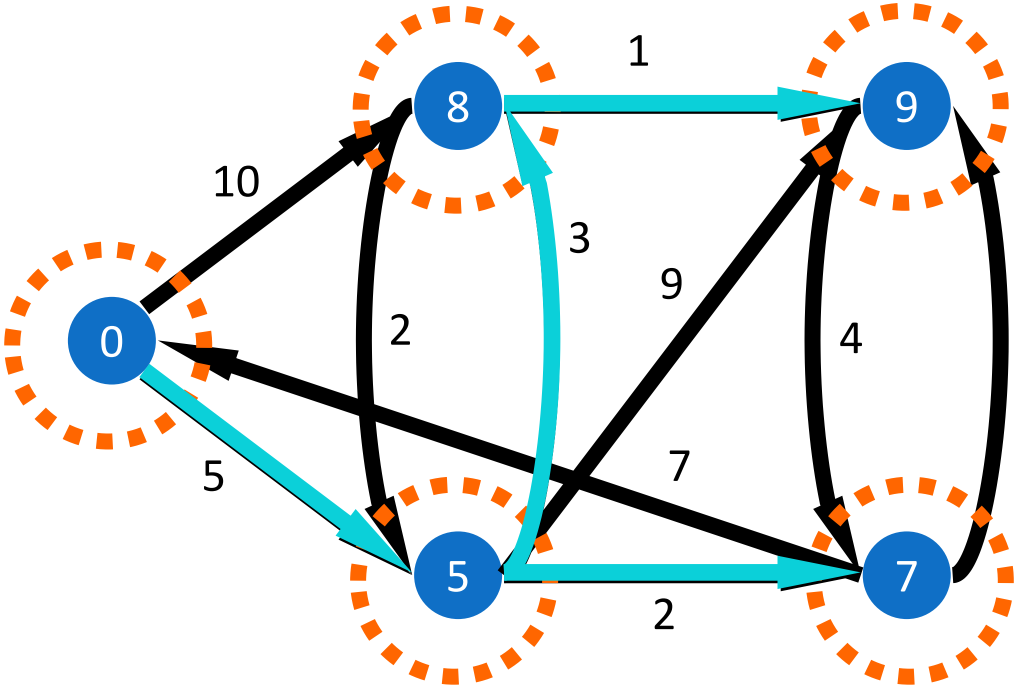
Dijkstra's Algorithm Illustration



Dijkstra’s Algorithm Illustration



Dijkstra’s Algorithm Illustration

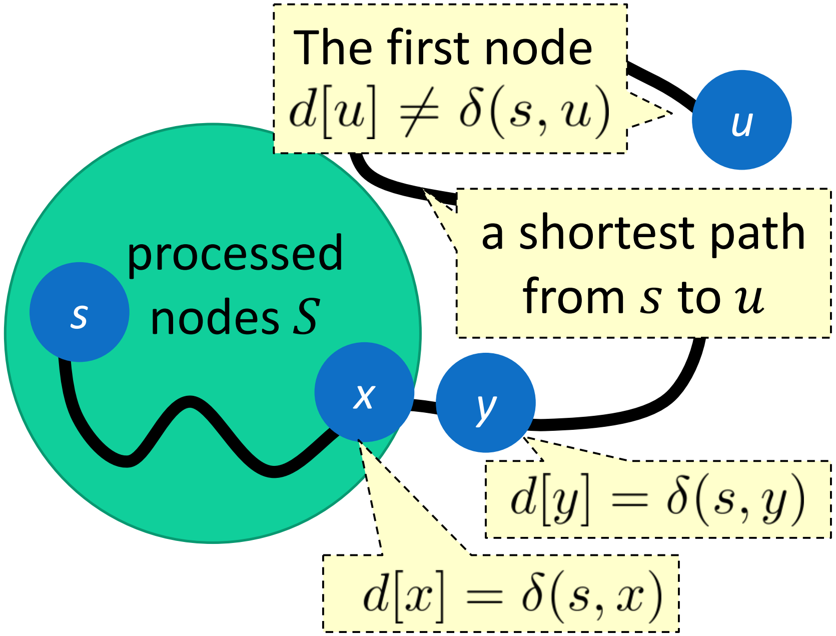


Dijkstra's Algorithm Correctness

The vertex selected by Dijkstra's algorithm into the processed set must precise estimation of its shortest path distance.

- Prove by contradiction
  - Assume u is the first vertex for being processed
  - Let a shortest path P from s to u,
    - x is the last vertex in P from S
    - y is the first vertex in P not from S
  - $d[y] = \delta(s, y)$ because (x, y) is relaxed when putting x into S

$$d[u] > \delta(s,u) \geq \delta(s,y) = d[y]$$



Dijkstra's Time Complexity

```
DIJKSTRA(G, w, s)
  INITIALIZATION(G, s)
  S = empty
  Q = G.v // INSERT             $n$ times
  while Q ≠ empty               $n$ times
    u = EXTRACT-MIN(Q)          $O(n)$
    S = SU{u}
    for v in G.adj[u]           $m$ times
      RELAX(u, v, w)
```

```
INITIALIZATION(G, s)
  for v in G.V     ⎫
    v.d = ∞        ⎬ $O(n)$
    v.π = NIL      ⎪
  s.d = 0          ⎭
```

```
RELAX(u, v, w)                      $O(1)$
  if v.d > u.d + w(u, v)
    // DECREASE-KEY
    v.d = u.d + w(u, v)
    v.π = u
```

- Min-priority queue
  - `INSERT`: $O(1)$
  - `EXTRACT-MIN`: $O(n)$
  - `DECREASE-KEY`: $O(1)$
- Total complexity: $O(n^2 + m)$

Dijkstra’s Time Complexity

```
DIJKSTRA(G, w, s)
  INITIALIZATION(G, s)
  S = empty
  Q = G.v // INSERT                $O(n)$
  while Q ≠ empty                  $n$ times
    u = EXTRACT-MIN(Q)             $O(\log n)$
    S = S∪{u}
    for v in G.adj[u]              $m$ times
      RELAX(u, v, w)
```

```
INITIALIZATION(G, s)
  for v in G.V      ┐
    v.d = ∞         │
    v.π = NIL       ├ $O(n)$
  s.d = 0           ┘
```

```
RELAX(u, v, w)                          $O(1)$
  if v.d > u.d + w(u, v)
    // DECREASE-KEY
    v.d = u.d + w(u, v)
    v.π = u
```

- 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: $O(m + n \log n)$ with Fabonacci heap
- Single-Source Shortest Paths
  - Bellman-Ford Algorithm (general graph and weights)
    - $O(mn)$ and detecting negative cycles
  - Lawler Algorithm (acyclic graph)
    - $O(m + n)$
  - Dijkstra Algorithm (non-negative weights)
    - $O(m + n \log n)$ with Fabonacci heap

Question?

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

Course Website: http://ada17.csie.org

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