



CSCI2100B CSCI2100S DATA STRUCTURES

Spring 2011

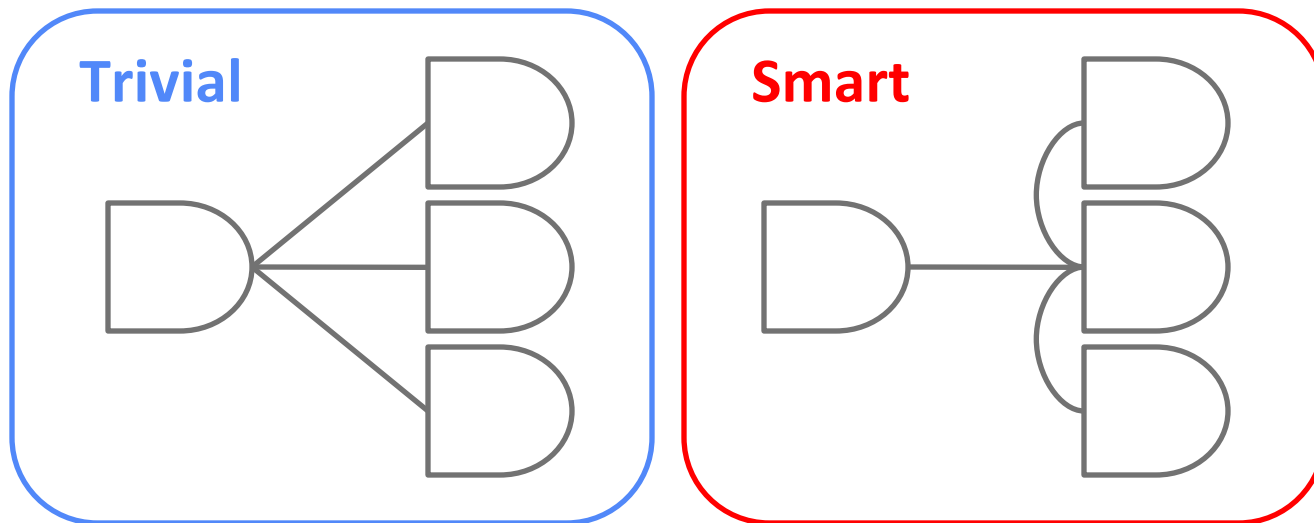
More Graph Algorithms:
Minimum Spanning Tree & Shortest Path

MINIMUM SPANNING TREE



The Wiring Problem

- In circuit design, we often have to **connect** the pins of several components with the same signal.
- A straight-forward way is to use $N - 1$ wires connecting all N components.
- Clearly, we can make a better arrangement is to connect components closer to each other first and then use a long wire to connect to **faraway** component.



The Wiring Problem: Modeling

- We can model this wiring problem with a connected, undirected graph $G = (V, E)$ where V is the set of pins, E is the set of possible **interconnects** between pairs of pins.
- Each edge (u, v) , we have a weight $w(u, v)$ specifying the **cost** (the amount/length of wire) to connect u and v .
- Then we wish to find an acyclic subset $T \subseteq E$ that connects all the vertices and whose total weight

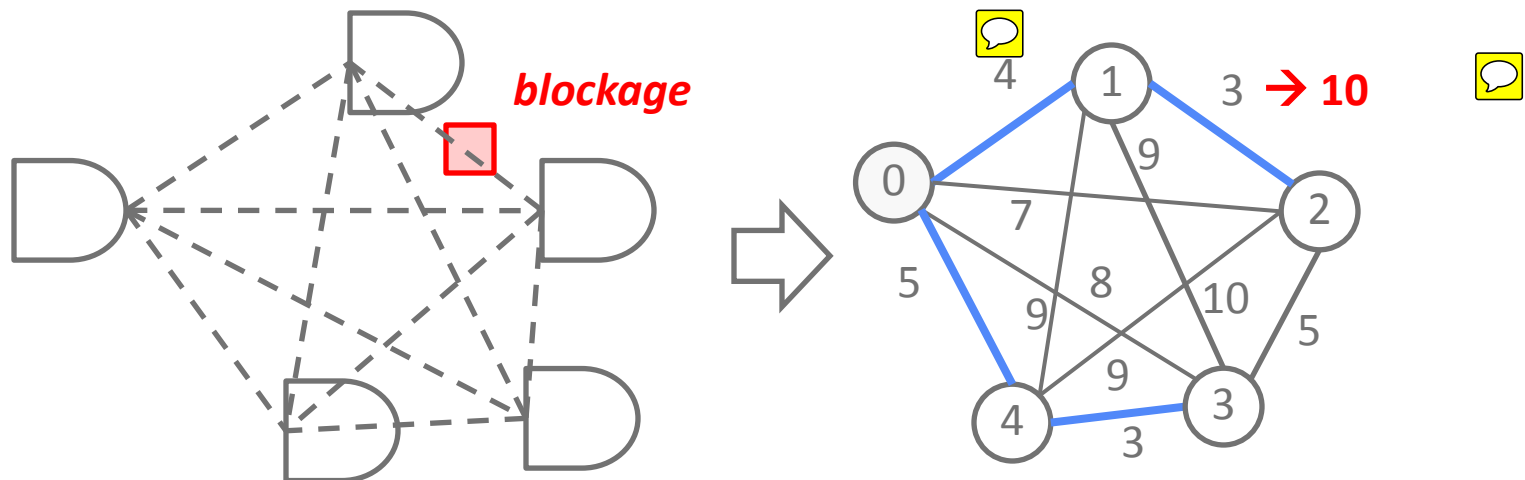
$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

is **minimized**.

- This is what we call the **minimum spanning tree** problem, which can be solved in $O(E \lg V)$ time using **Prim's algorithm** with a heap.

The Wiring Problem: Example

- The shaded edges form the minimum spanning tree and implies the **best** wiring scheme for connecting the 5 pins.
- Imagine what would happen some obstacle is placed between vertices 1 and 2 and make the weight of edge (u, v) becomes 10?



Minimum Spanning Tree (MST)

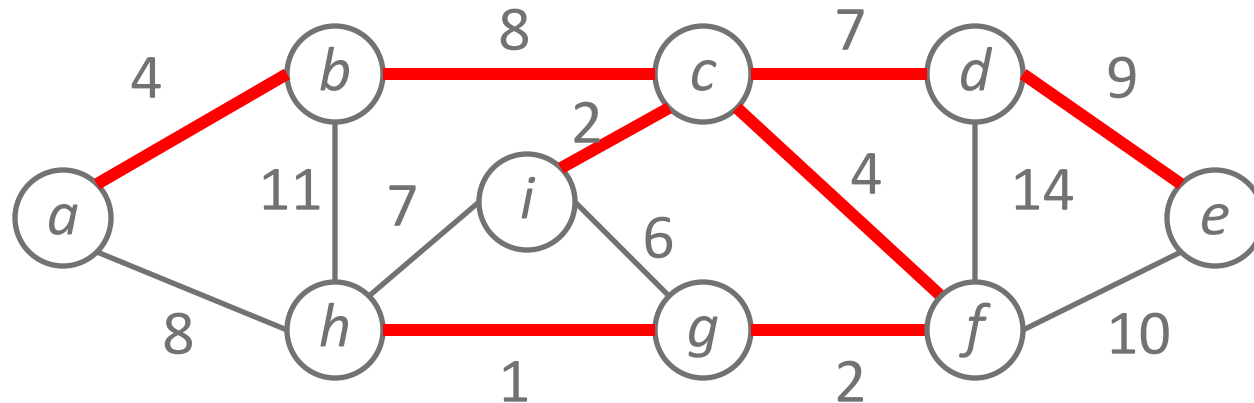
- Assume that we have a connected, undirected graph $G = (V, E)$ with a **weight** function $w: E \rightarrow R$.
- We wish to find an acyclic subset $T \subseteq E$ that connects all of the vertices and whose **total weight**

$$w(T) = \sum_{(u,v) \in T} w(u, v)$$

is **minimized**.

- Since T is acyclic and connects all of the vertices, it must form **tree** - **spanning tree** since it spans the graph G .

MST: Example



- The edges in the MST is highlighted.
- The weight of the tree is 37.

Question:

Is there any different tree having total weight of 37? 

Question:

In general, is MST unique? 

Generic MST

- The common MST algorithms (like Kruskal's and Prim's) try to grow the minimum spanning tree **one** edge at a time.
- The algorithm manages a set of edges A , **maintaining** the following loop invariant:
 A is a subset of some minimum spanning tree
- At each step, we determine an edge (u, v) (a **safe edge**) that can be added to A without violating this invariant.
- Thus **$A \cup (u, v)$** is also a subset of MST.

Generic MST: Algorithm

GENERIC-MST(G, w)

```
1:  $A \leftarrow \phi$ 
2: while  $A$  does not form a spanning tree do
3:     find an edge  $(u, v)$  that is safe for  $A$ 
4:      $A \leftarrow A \cup \{(u, v)\}$ 
5: end while
6: return  $A$ 
```

- **Initialization:** after line 1 - the set A trivially satisfies the loop invariant.
- **Maintenance:** the loop in lines 2 – 4 maintains the invariant by adding only safe edges.
- **Termination:** all edges added to A are in a MST, so A returned in line 5 must be a MST.

Prim's Algorithm

- **The Big Q: How to find a safe edge to be added to A ?**
- Prim's algorithm has the property that **the edges in A always form a single tree**.
- The tree starts from an arbitrary root vertex r and grows until the tree span all vertices in V .
- At each step, a **light** edge is added to A that connects A to an isolated vertex of $G_A = (V, A)$.
- This rule adds only safe edges!

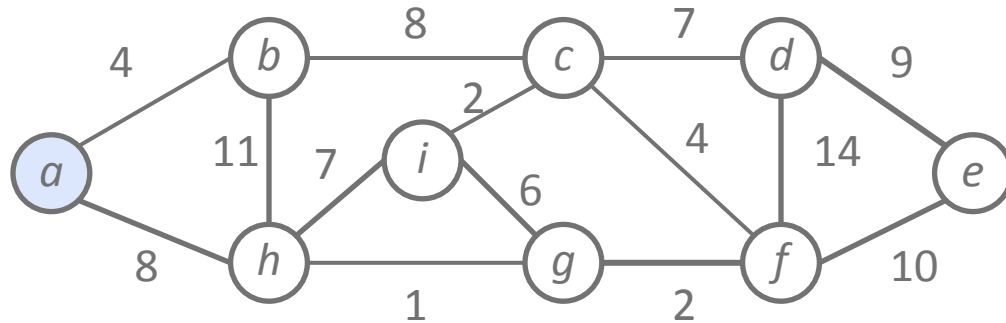
Prim's algorithm is a classic example of greedy algorithm since at each step an edge that contribute the minimum amount possible to the tree's weight is picked.

Prim's Algo: Example (1)

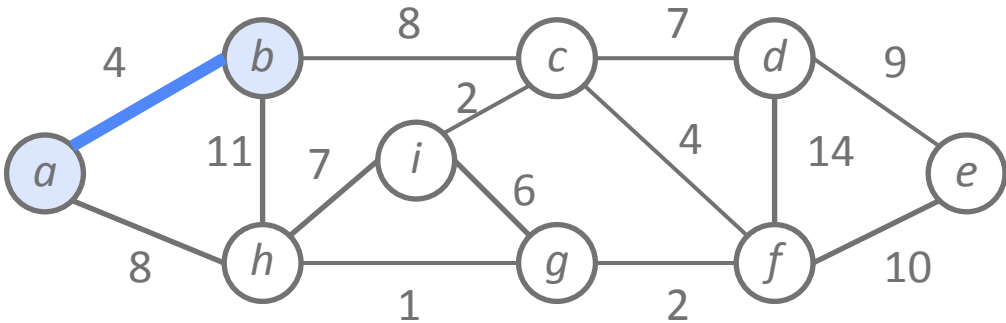


Priority Queue

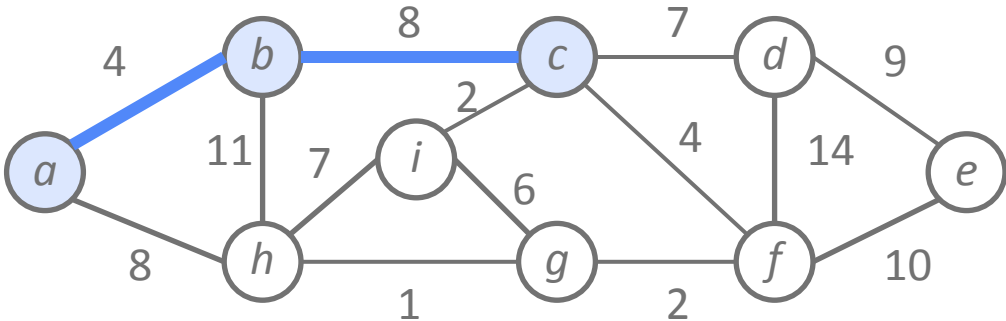
<i>b</i>	<i>h</i>		
4	8		



<i>c</i>	<i>h</i>		
8	8		



<i>i</i>	<i>f</i>	<i>d</i>	<i>h</i>
2	4	7	8

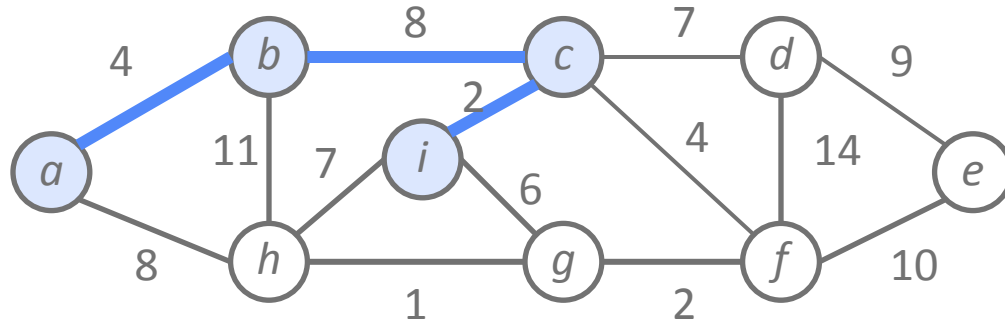


Prim's Algo: Example (2)

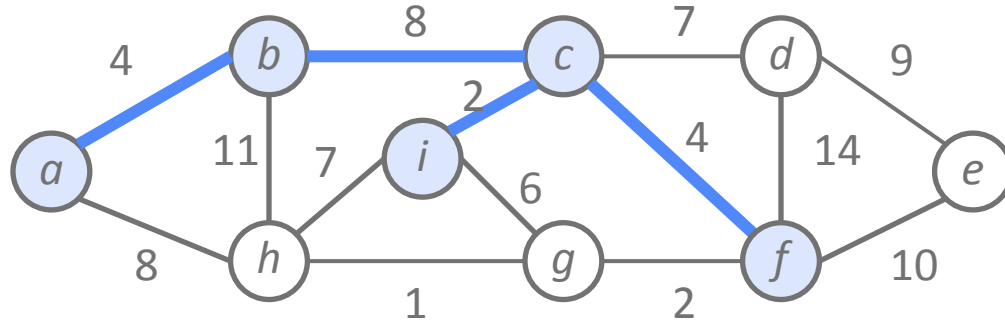


Priority Queue

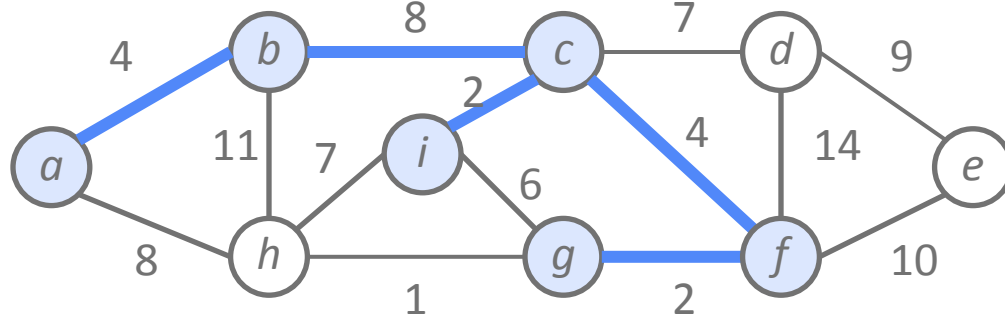
<i>f</i>	<i>g</i>	<i>d</i>	<i>h</i>
4	6	7	7



<i>g</i>	<i>h</i>	<i>d</i>	<i>e</i>
2	7	7	10



<i>h</i>	<i>e</i>	<i>d</i>	
1	10	7	

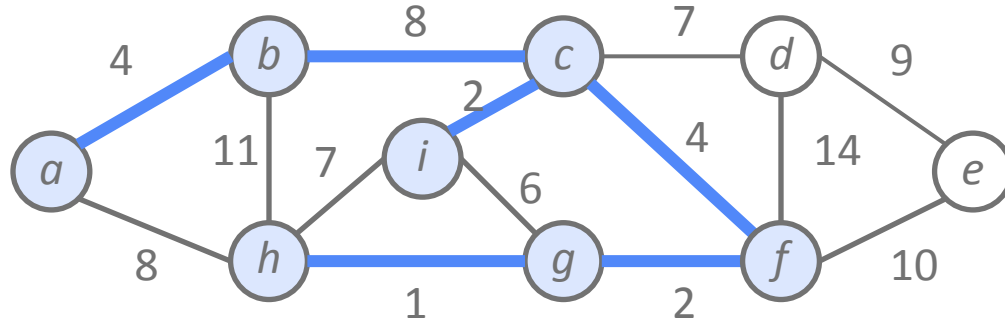


Prim's Algo: Example (3)

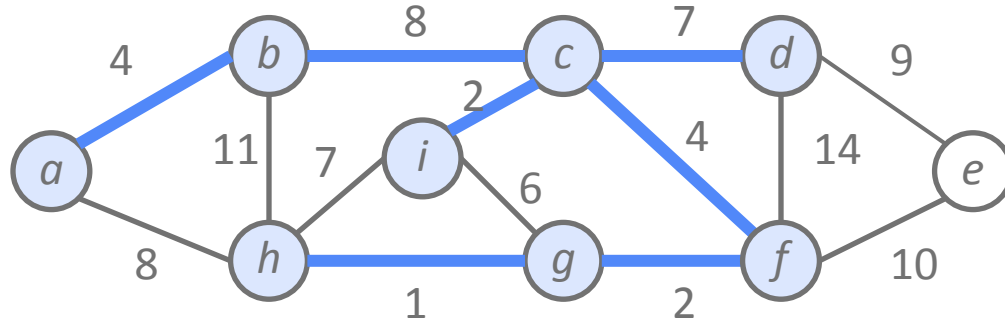


Priority Queue

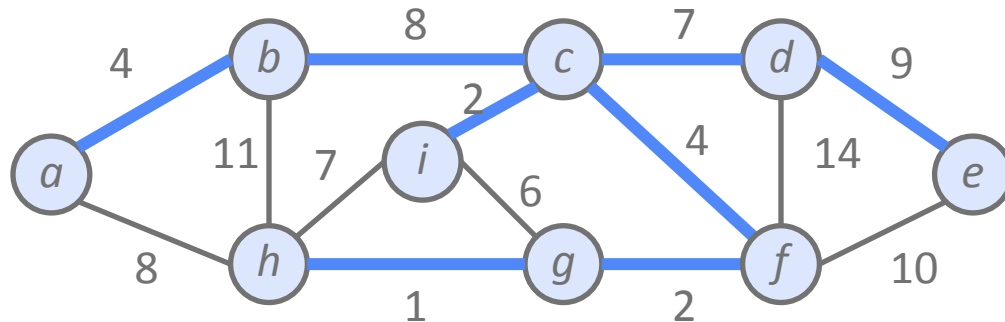
<i>d</i>	<i>e</i>		
7	10		



<i>e</i>			
9			



empty. done.






Prim's Algo: Implementation

- The key to implementing Prim's algorithm is to make it easy to **select a new edge** to be added.
- During execution of the algorithm, all vertices that are not in the tree reside in a priority queue Q (min-heap) based on a key value.
- $\text{key}[v]$ is the **minimum weight** of any edge connecting v to a vertex in the tree.
- The algorithm keeps A implicitly as
$$A = \{(v, \pi[v])\} : v \in V - \{r\} - Q\}$$
- When the algorithm terminates, Q is empty, the MST A for G is thus

$$A = \{(v, \pi[v])\} : v \in V - \{r\}\}$$

Prim's Algorithm

MST-PRIM(G, w, r)

```
1: for each  $u \in V[G]$  do   
2:    $key[u] \leftarrow \infty$   
3:    $\pi[u] \leftarrow \text{NULL}$   
4: end for  
5:  $key[r] \leftarrow 0$   
6:  $Q \leftarrow V[G]$    
7: while  $Q \neq \phi$  do   
8:    $u \leftarrow \text{EXTRACT-MIN}(Q)$   
9:   for each  $v \in \text{Adj}[u]$  do  
10:    if  $v \in Q$  and  $w(u, v) < key[u]$  then  
11:       $\pi[u] \leftarrow v$   
12:       $key[u] \leftarrow w(u, v)$   
13:    end if  
14:  end for  
15: end while
```

Prim's Algorithm: Analysis

- If Q is implemented as a min-heap, we can use **build_heap** to perform initialization from line 1 - 6 in $O(V)$ time.
- The body of the while loop executes V times, and since each **delmin** takes $O(\lg V)$ time, the total time is $O(V \lg V)$.
- The for-loop in lines 9 - 14 executed $O(E)$ times altogether. (Why?)
- Line 10 can be done in **constant** time by storing a flag.
- Line 12 involves an implicit **decrease** operation, which takes $O(\lg V)$ time.
- Total time for Prim's algorithm
 $= O(V \lg V + E \lg V) = O(E \lg V)$.

Prim's Algo: Code (1)

The weights of the edge can be stored at the nodes of the adj. lists.
Alternatively you can build a direct-address table for the weight function.

```
void prim(graph G, int r){
    int i, u, v, cnt = 0, n = G->V;
    int *p = (int *)malloc(n * sizeof(int));
    int *key = (int *)malloc(n * sizeof(int));
    char *pre = (char *)malloc(n * sizeof(char));
    pq_t *pq;
    node_t *t;

    for (i = 0; i < n; i++){
        key[i] = PRIM_INFITY;
        p[i] = PRIM_NULL;
        e[i] = i;
        pre[i] = -1;
    }
    key[r] = 0;
    pq = pq_build_key(n, e, key);
```

Prim's Algo: Code (2)

- Main loop: delete min from Q , go to the adj. list and find light edges to be added to A .
- Book-keeping the memberships of Q .
Speed up the algorithm by avoiding searching the heap in $O(V)$ time.

```
while (!pq_is_empty(pq)){
    u = pq_delete_min(pq);
    pre[u] = cnt++;
    for (t = G->adj[u]; t != NULL; t = t->next){
        v = t->v;
        if (pre[v] == -1 && t->w < key[v]){
            p[v] = u;
            key[v] = t->w;
            pq_decrease_key(pq, v, t->w);
        }
    }
}
pq_free(q);
free(p); free(key); free(pre);
}
```

MST: Summary

■ **Minimum Spanning Tree Problem**

use edges with minimum total cost to span all vertices of the graph

- **Application:** wiring problem

- Generic MST algorithm

■ **Prim's Algorithm**

- **Greedy algorithm:** connect the minimum edge connecting the subtree

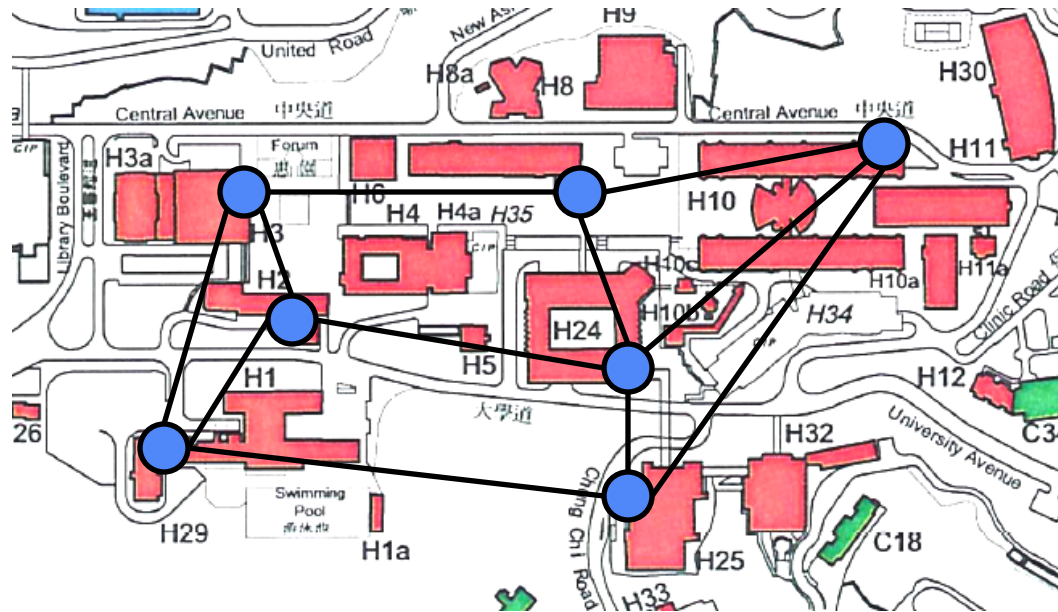
- **Implementation:** requires priority queue (binary heaps) for efficiency. Complexity: $O(E \lg V)$.

SHORTEST PATH



A Lazy CUHK Walker

- We usually walk around to various locations to attend lectures, have meals, have discussions and so on. As our lab is located at SHB, most of the time you start your route at SHB.
- Which is the **shortest** route (quickest route) to arrive all different destinations in CUHK?



Shortest-Paths Problem

- We are given a **weighted, directed** graph $G = (V, E)$, weight function $w : E \rightarrow \mathbb{R}$.
- The **weight** of path $p = \langle v_0, v_1, \dots, v_k \rangle$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^k w(v_{i-1}, v_i)$$

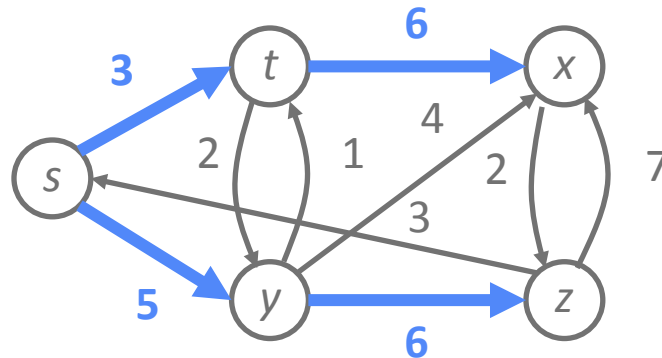
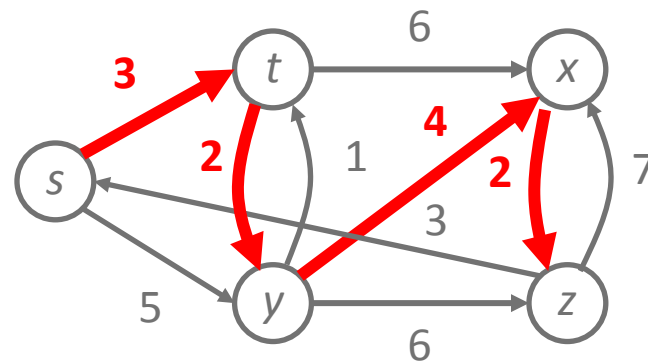
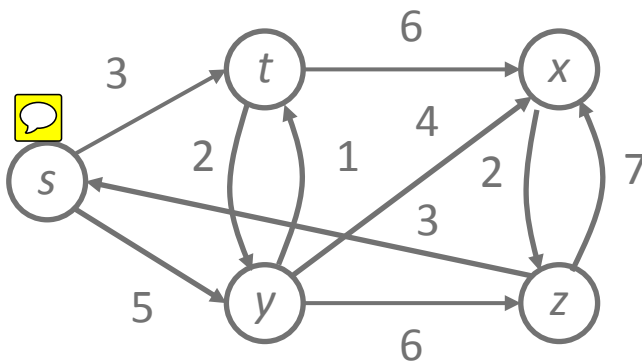
- We define the shortest-path weight from u to v by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \rightsquigarrow v\} & \text{if } \exists \text{ a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

- A **shortest** path from vertex u to vertex v is then defined as any path p with weight $w(p) = \delta(u, v)$

Shortest-Paths: Example

- Shortest paths are **not necessarily unique**.
- The following example shows a weighted, directed graph and two shortest-paths trees with the same root.



Initialization

- Shortest-path **estimate**: for each vertex v , we maintain an attribute $d[v]$, which is an **upper bound** on the weight of a shortest path from source s to v .
- We initialize the shortest-path estimates and predecessors as a procedure ***initialize_single_source***.

INITIALIZE-SINGLE-SOURCE(G, s)

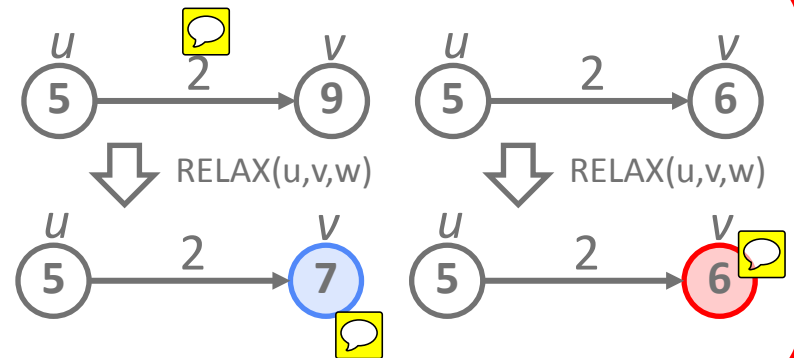
```
1: for each vertex  $v \in V[G]$  do
2:    $d[v] \leftarrow \infty$ 
3:    $\pi[v] \leftarrow NULL$ 
4: end for
5:  $d[s] \leftarrow 0$ 
```


Relaxation



- The process of **relaxing** an edge (u, v) consists of testing whether we can **improve** the shortest path to v found so far by going through u .
- If so, **updates** $d[v]$ and $\pi[v]$.

RELAX(u, v, w)

```
1: if  $d[v] > d[u] + w(u, v)$  then  
2:    $d[v] \leftarrow d[u] + w(u, v)$   
3:    $\pi[v] \leftarrow u$   
4: end if
```



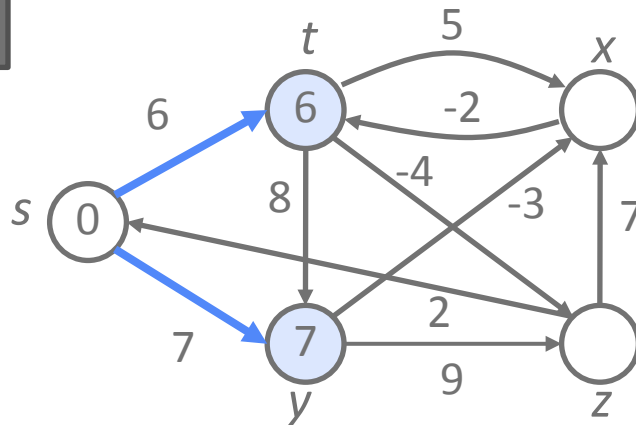
Bellman-Ford Algorithm

- The algorithm solves the shortest-path problem in the general case in which edge weights may be negative 
- It also indicates  whether or not there is a negative-weight cycle that is reachable from the source.
- It uses relaxation to progressively decrease an estimate $d[v]$ until it achieves the actual shortest-path weight $\delta(u, v)$.

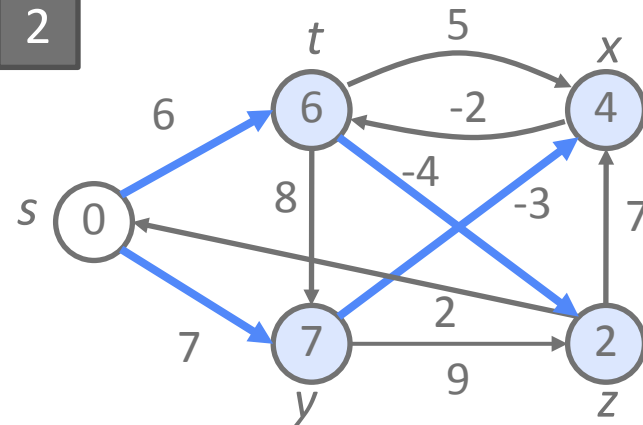


Bellman-Ford Algo.: Example

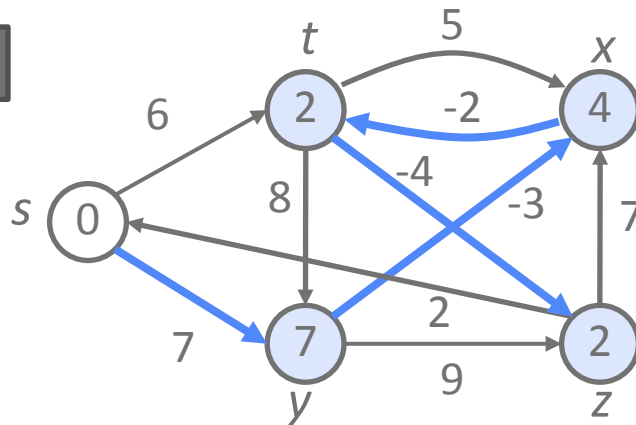
1



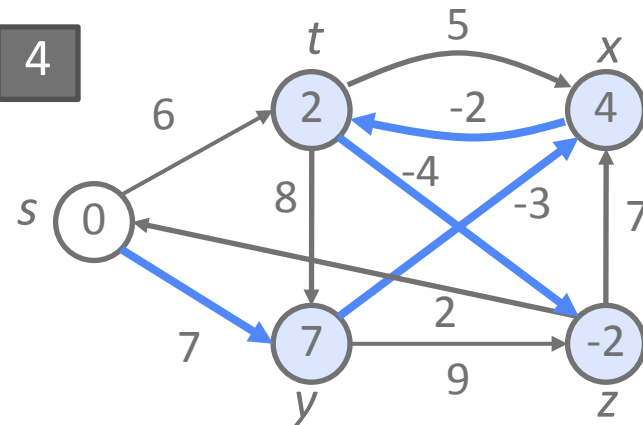
2



3



4



Bellman-Ford Algorithm

```
BELLMAN-FORD( $G, w, s$ )
1: INITIALIZE-SINGLE-SOURCE( $G, s$ )
2: for  $i \leftarrow 1$  to  $|V| - 1$  do
3:     for each edge  $(u, v) \in E$  do
4:         RELAX( $u, v, w$ )
5:     end for
6: end for
7: for each edge  $(u, v) \in E$  do
8:     if  $d[v] > d[u] + w(u, v)$  then
9:         return FALSE
10:    end if
11: end for
12: return TRUE
```

Complexity: $O(VE) = O(V^3)$

Bellman-Ford Algorithm: Code

```
void init(graph G, int s, int *d, int *p){
    int i;
    for (i = 0; i < G->V; i++){
        d[i] = SHPATH_INFTY;
        p[i] = SHPATH_NULL;
    }
    d[s] = 0;
}


int graph_bellmanford(graph G, int s){
    int i, u, v, w, n = G->V;
    int nc = 0; /* nc: negative cycle */
    node *t;
    int *p = malloc(n * sizeof(int));
    int *d = malloc(n * sizeof(int));
    init(G, s, d, p);
    for (i = 0; i < n - 1; i++)
        for (u = 0; u < n; u++)
            for (t = G->adj[u]; t != NULL; t = t->next){
                v = t->v; w = t->w;
                if (d[v] > d[u] + w){ /* relax */
                    d[v] = d[u] + w;
                    p[v] = u;
                }
            }
}
```

Bellman-Ford Algorithm: Code

```
/* detect negative cycle */
for (u = 0; u < n; u++)
    for (t = G->adj[u]; t != NULL; t = t->next){
        v = t->v;
        w = t->w;
        if (d[v] > d[u] + w)
            nc = 1;
    }
printf("Bellman-Ford's Algorithm:\n");
for (i = 0; i < n; i++)
    printf("d[%d] = %d (p=%d)\n", i, d[i], p[i]);

free(p); free(d);
return !nc;
}
```

Dijkstra's Algorithm

- The algorithm only **works** on graphs in which **all** edge weights are **non-negative**. 
- Thus we assume $w(u, v) \geq 0$ for edge in E .
- The algorithm maintains a set S of vertices whose final shortest-path weights from s have already been **determined**.
- Repeatedly selects the vertex $u \in V - S$ with the **minimum** shortest-path estimates.
 - This step requires the use of a **priority queue** so that we can pick u quickly.
- Adds u to S and relax all edges leaving u .

Dijkstra's Algorithm

DIJKSTRA(G, w, s)

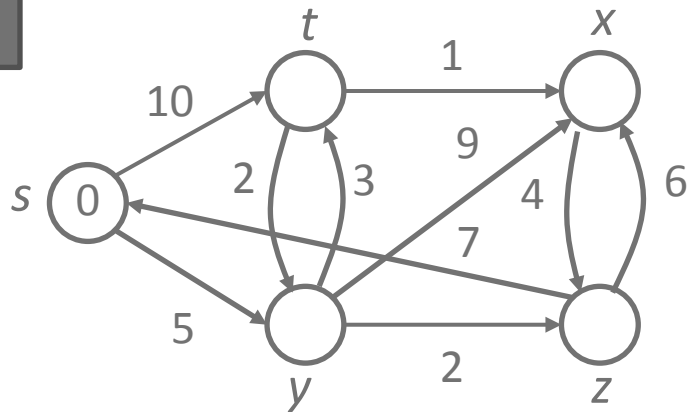
```
1: INITIALIZE-SINGLE-SOURCE( $G, s$ )
2:  $S \leftarrow \phi$ 
3:  $Q \leftarrow V[G]$ 
4: while  $Q \neq \phi$  do
5:      $u \leftarrow \text{EXTRACT-MIN}(Q)$ 
6:      $S \leftarrow S \cup \{u\}$ 
7:     for each vertex  $v \in \text{Adj}[u]$  do
8:         RELAX( $u, v, w$ )
9:     end for
10: end while
```

■ Why is Dijkstra's algorithm correct?

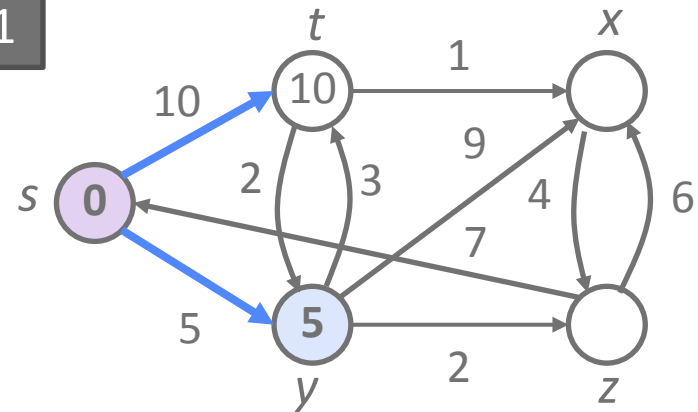


Dijkstra's Algorithm: Example

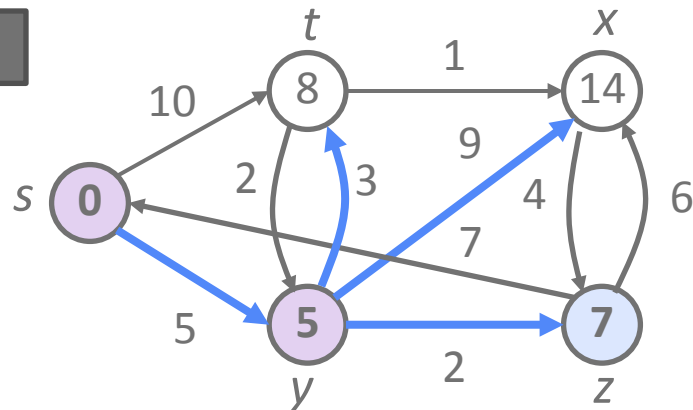
0



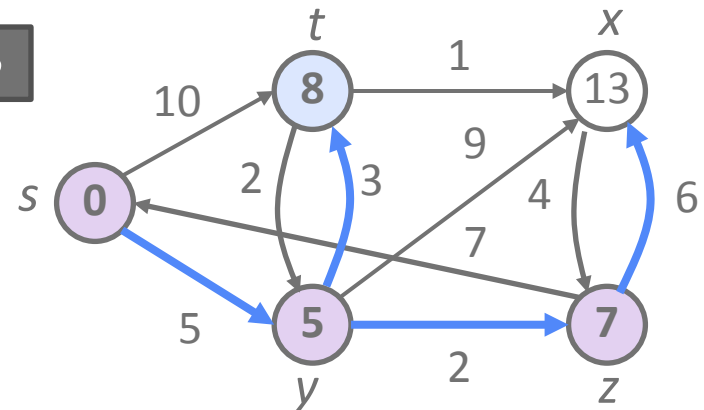
1



2



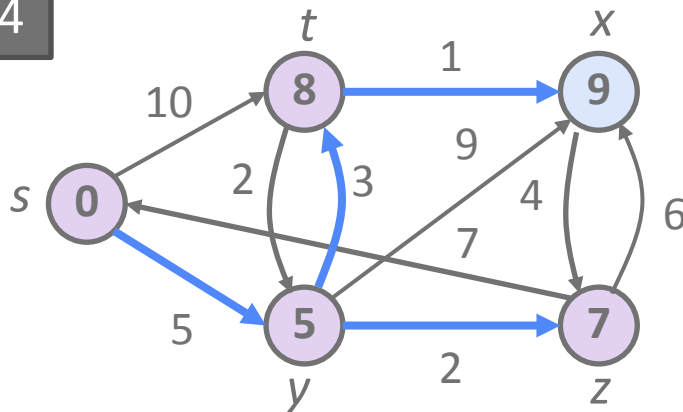
3



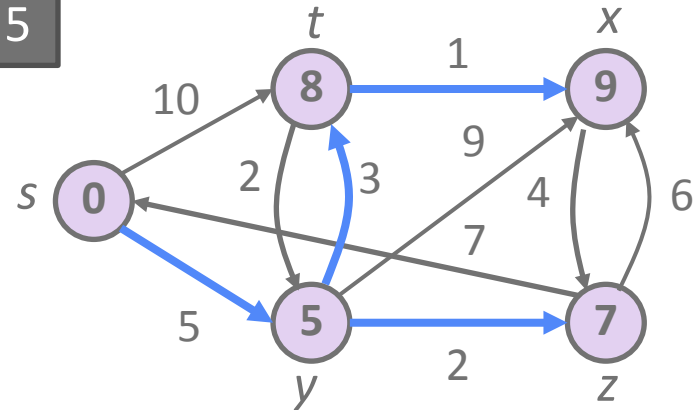


Dijkstra's Algorithm: Example (2)

4



5



Sometimes It is useful to show the values of $d[v]$ in a table.

$d[v]$	s	t	x	y	z
0	0	∞	∞	∞	∞
1: s	0	10	∞	5	∞
2: y	0	8	14	5	7
3: z	0	8	13	5	7
4: t	0	8	9	5	7
5: x	0	8	9	5	7

Dijkstra's Algorithm (Code)

```
void graph_dijkstra(graph G, int s){
    int i, u, v, w, n = G->V;
    int *p = (int *)malloc(n * sizeof(int));
    int *d = (int *)malloc(n * sizeof(int));
    int *e = (int *)malloc(n * sizeof(int));
    pq_t *pq; node *t;

    init(G, s, d, p);
    for (i = 0; i < n; i++)
        e[i] = i;
    pq = pq_build_key(n, e, d);

    while (!pq_is_empty(pq)){
        u = pq_delete_min(pq);
        for (t = G->adj[u]; t != NULL; t = t->next){
            v = t->v; w = t->w;
            if (d[v] > d[u] + w){ /* relax */
                d[v] = d[u] + w;
                p[v] = u;
                pq_decrease_key(pq, v, d[v]);
            }
        }
    }
    free(e); free(p); free(d); pq_free(pq);
}
```

Dijkstra's Algorithm: Analysis

- Initialization: $O(V)$
- While loop on line 4: execute $O(V)$ since no vertex will be inserted to PQ after line 3.
- *delmin* takes $O(\lg V)$ if implemented using **min-heap**.
- Throughout the while loop, for loop on line 7 travels to all adjacency lists: $O(E)$.
 - Be careful that there is an implicit $O(\lg V)$ *decrease* operation
- Time complexity:
 $O(V \lg V + E \lg V) = O(E \lg V)$

Shortest Path: Summary

- **Shortest-paths problem**

Find the shortest weighted paths from a given source vertex s to other vertices.

- **Relaxation**

- **Bellman-Ford Algorithm: $O(VE)$**

a general application of relaxation
determine negative weight cycle

- **Dijkstra's algorithm: $O(E \lg V)$**

greedily pick vertex with the minimum path estimates.

- **Implementation:** requires priority queue