Graph Algorithms

Fall 2020

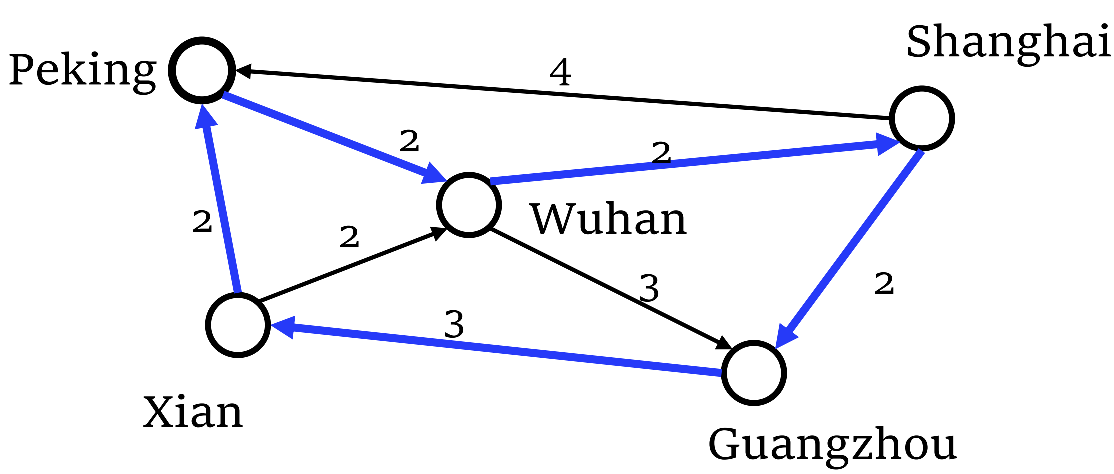
School of Software Engineering
South China University of Technology

Shortest-Path Algorithms

Section 9.3

Recall Path cost ,Path length

- Path cost: the sum of the costs of each edge
- Path length: the number of edges in the path
  - Path length is the unweighted path cost



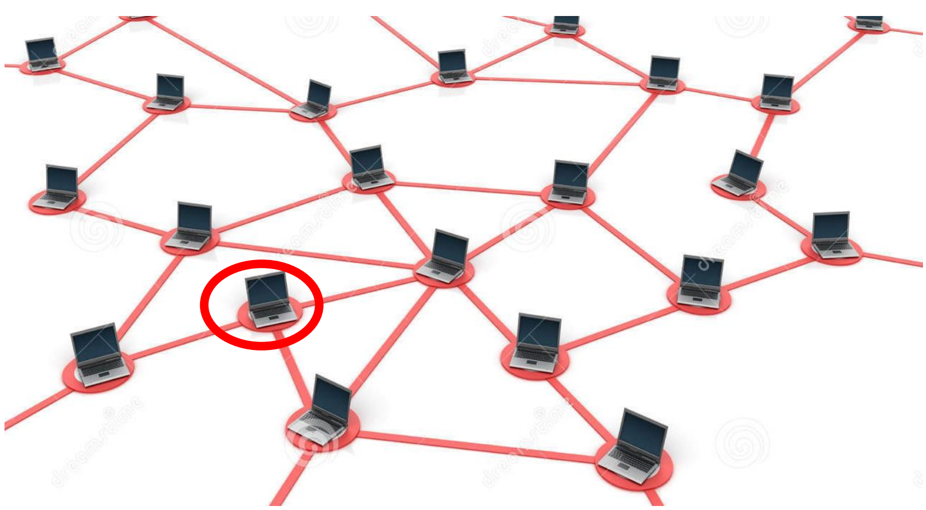
length(p) = 5

cost(p) = 11

Shortest Path Problems

- Given a graph G = (V, E) and a "source" vertex s in V, find the minimum cost paths from s to every vertex in V
  - Single-Source Shortest Paths problem

- Many variations:
  - unweighted vs. weighted
  - cyclic vs. acyclic
  - pos. weights only vs. pos. and neg. weights
  - etc



Why study shortest path problems?

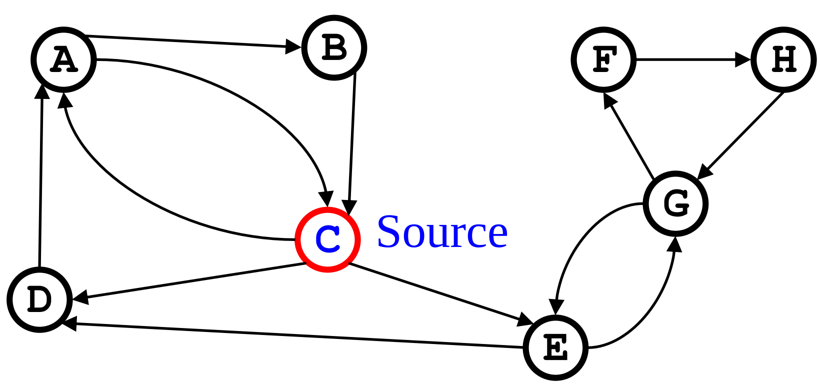
- Traveling on a budget: What is the cheapest airline schedule from Guangzhou to city X?
- Optimizing routing of packets on the internet:
  - Vertices are routers and edges are network links with different delays. What is the routing path with smallest total delay?
- Shipping: Find which highways and roads to take to minimize total delay due to traffic
- etc.

Unweighted Shortest Path

Problem: Given a "source" vertex s in an unweighted directed graph

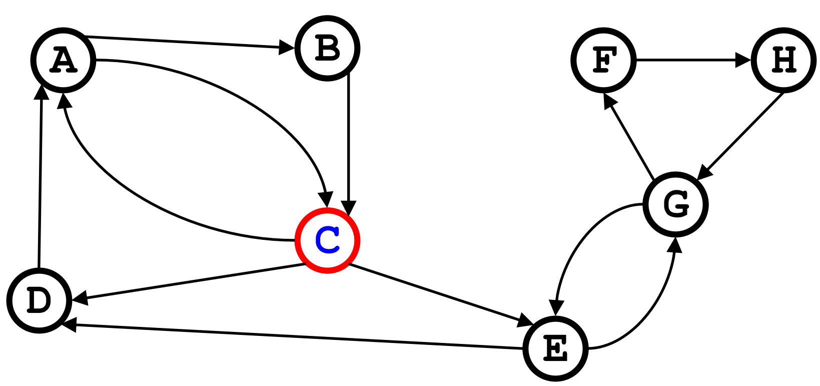
G = (V,E), find the shortest path from s to all vertices in G

Only interested in path lengths



Breadth-First Search Solution

- Basic Idea: Starting at node s, find vertices that can be reached using 0, 1, 2, 3, ..., N-1 edges (works even for cyclic graphs!)

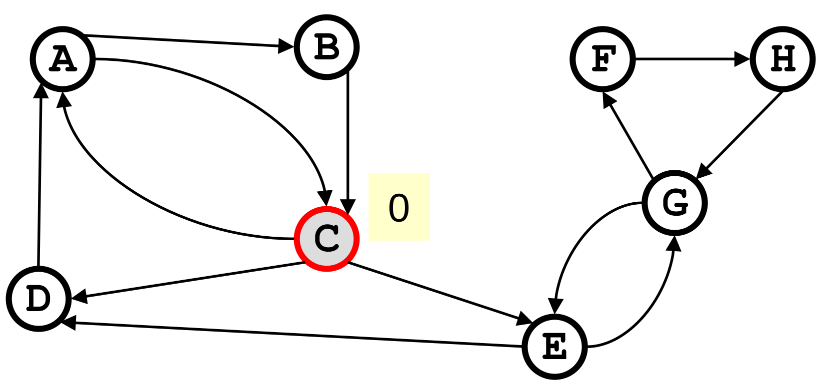


Breadth-First Search Algorithm

- Uses a queue to track vertices that are "nearby"
- source vertex is **s**
- Running time = $O(|V| + |E|)$

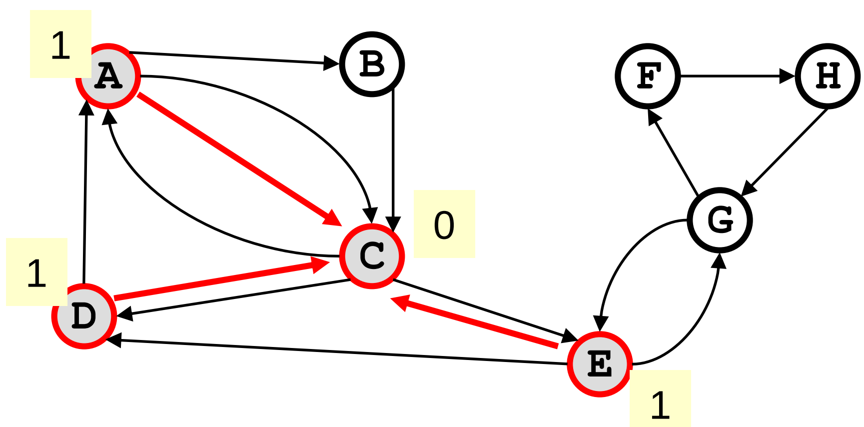
```
Distance[s] := 0
Enqueue(Q,s);
Mark(s); //After a vertex is marked once
         // it won't be enqueued again
while queue is not empty do
   X := Dequeue(Q);
  for each vertex Y adjacent to X do
     if Y is unmarked then
        Distance[Y] := Distance[X] + 1;
        Previous[Y] := X; //if we want to record paths
        Enqueue(Q,Y);
        Mark(Y);
```

Example: Shortest Path length



Queue Q = C

Example (ct'd)

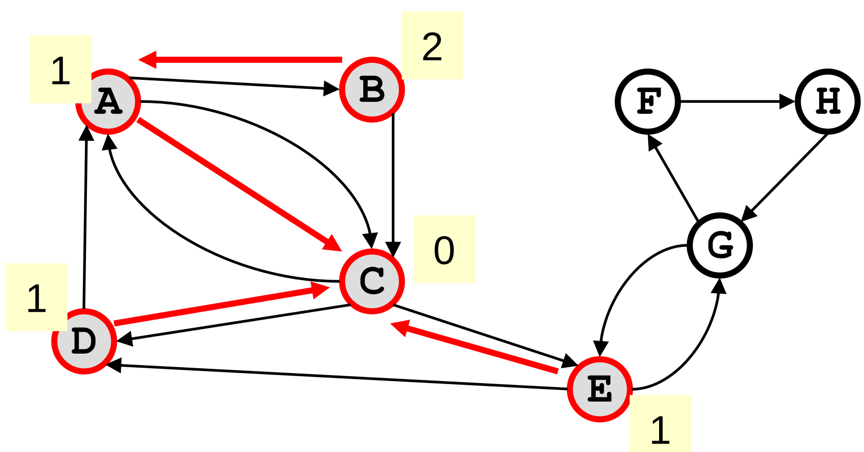


Queue Q = A D E

Indicates the vertex is marked

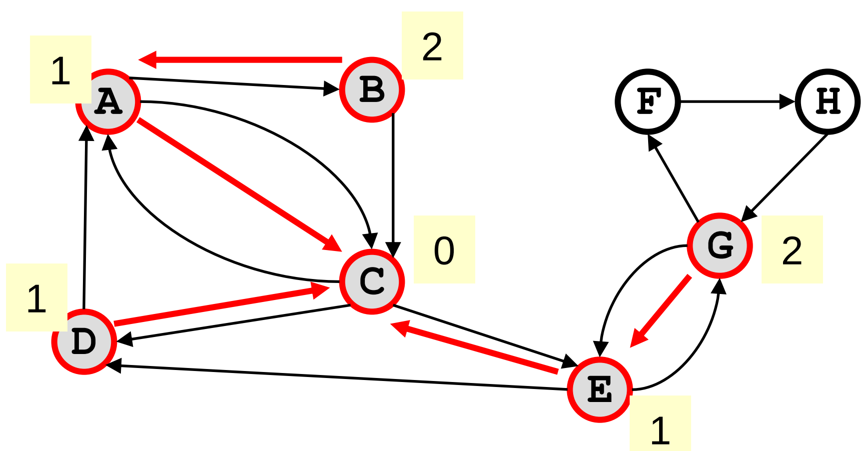
Previous pointer

Example (ct'd)



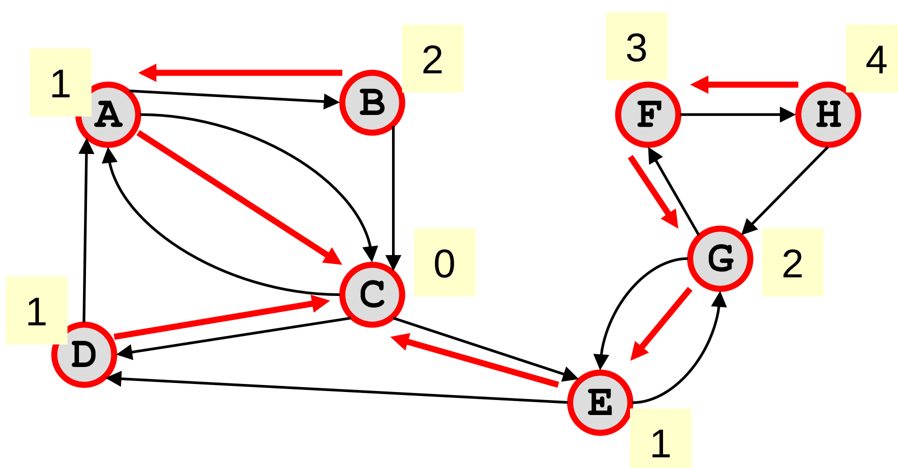
Q = D E B

Example (ct'd)



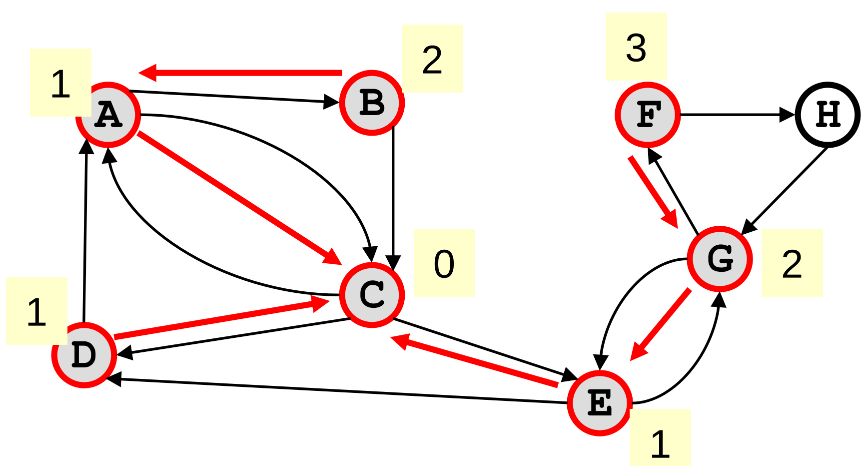
Q = B G

Example (ct'd)



Q = F

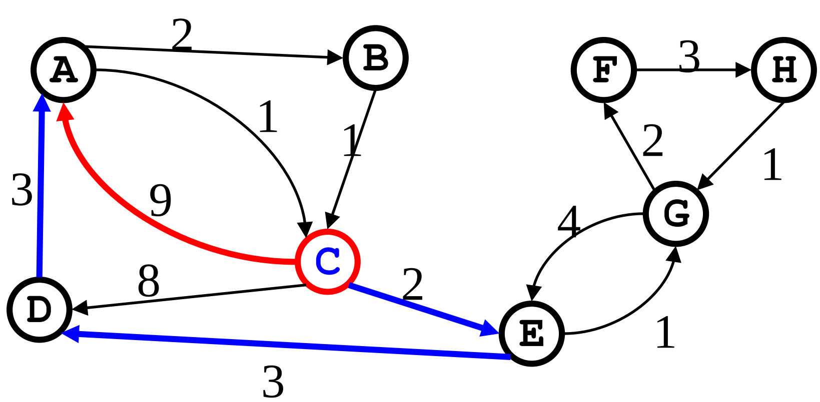
Example (ct'd)



Q = H

Weighted Shortest Path

- What if edges have weights?
- Breadth First Search does not work anymore
  - minimum *cost* path may have more edges than minimum *length* path



from C to A

Shortest path (length) = C→A (cost = 9)

Minimum Cost Path = C→E→D→A (cost = 8)

Dijkstra's Algorithm

- Classic algorithm for solving Single-Source Shortest Paths in weighted graphs (without negative weights)

- A greedy algorithm (irrevocably makes decisions without considering future consequences)

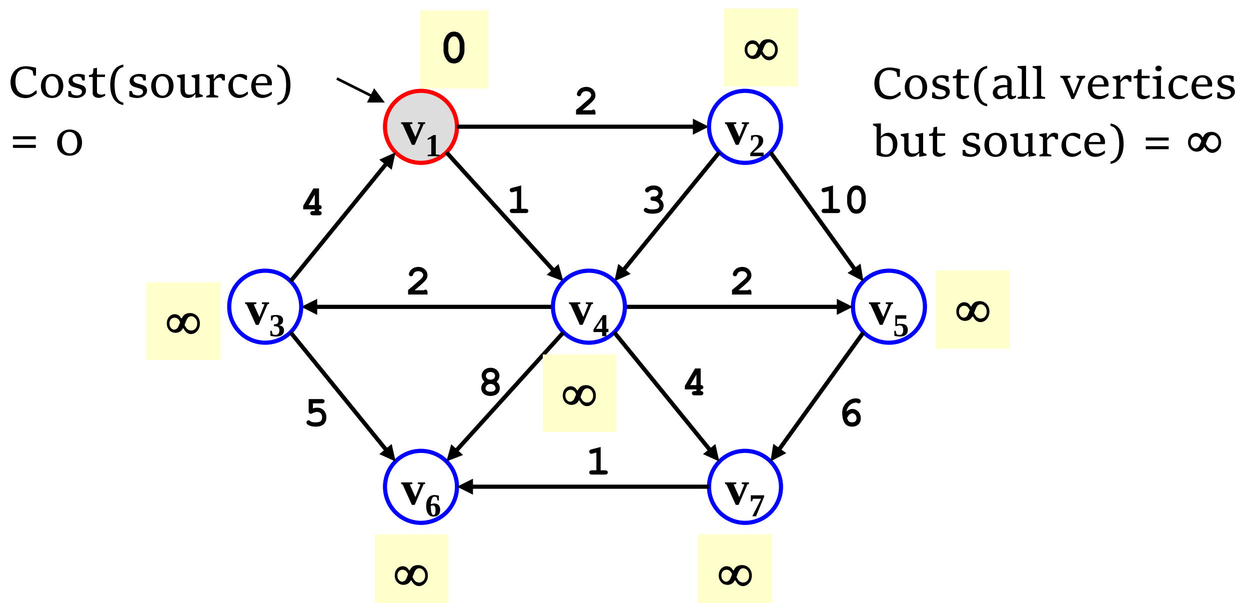
Basic Idea of Dijkstra's Algorithm

- Find the vertex with smallest cost that has not been "marked" yet.
- Mark it and compute the cost of its neighbors.
- Do this until all vertices are marked.
- Note that each step of the algorithm we are marking one vertex and we won't change our decision: hence the term "greedy" algorithm

Dijkstra's Shortest Path Algorithm

- Initialize the cost of s to 0, and all the rest of the nodes to ∞
- Initialize set S to be $\emptyset$
  - S is the set of nodes to which we have a shortest path
- While S is not all vertices
  - Select the node A with the lowest cost that is not in S and identify the node as now being in S
  - for each node B adjacent to A
    - if cost(A)+cost(A,B) < B's currently known cost
      - set cost(B) = cost(A)+cost(A,B)
      - set previous(B) = A so that we can remember the path

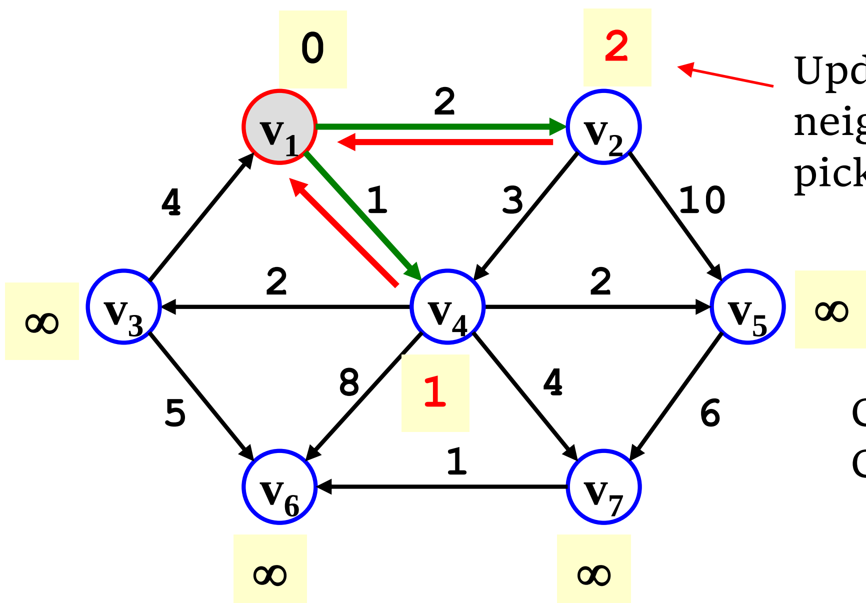
Example: Initialization



Pick vertex not in S with lowest cost in every step.

	S	1	2	3	4	5	6	7
initial	{}	<u>0</u>	∞	∞	∞	∞	∞	∞

Example: Update Cost neighbors

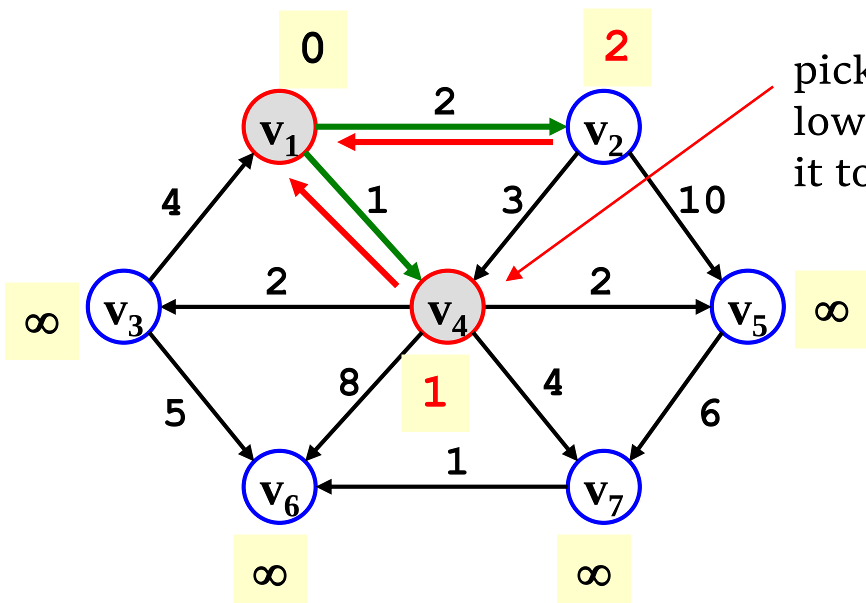


Update Cost neighbors of the picked vertex V1.

Cost(v2) = 2
Cost(v4) = 1

S	1	2	3	4	5	6	7
{}	0	∞	∞	∞	∞	∞	∞
{1}	0	2	∞	1	∞	∞	∞

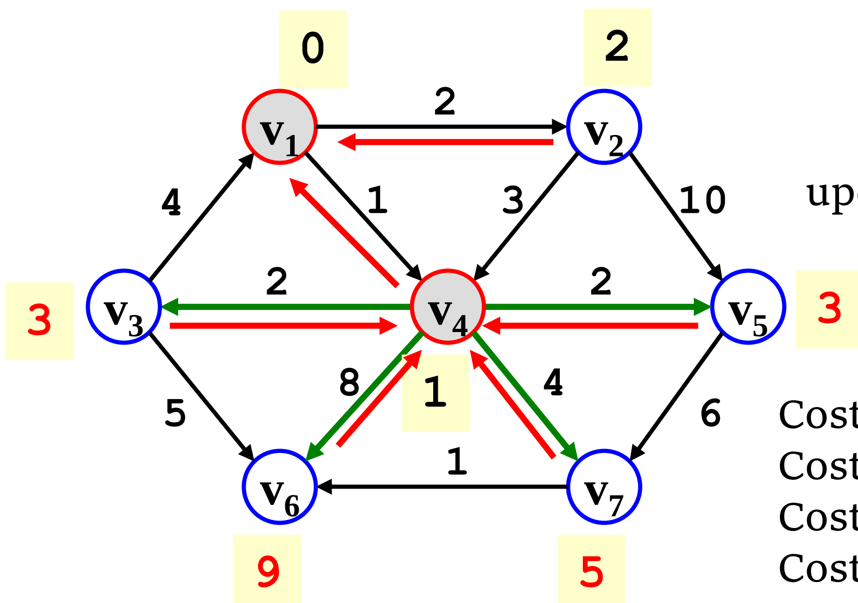
Example



pick vertex with lowest cost and add it to S

S	1	2	3	4	5	6	7
{}	<u>0</u>	∞	∞	∞	∞	∞	∞
{1}	<u>0</u>	2	∞	<u>1</u>	∞	∞	∞

Example

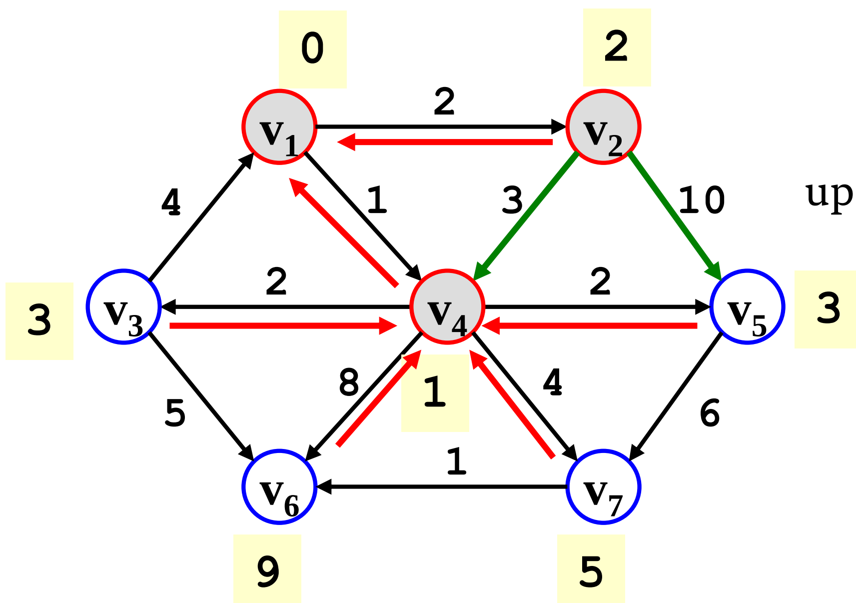


update neighbors

Cost(v3) = 1 + 2 = 3
Cost(v5) = 1 + 2 = 3
Cost(v6) = 1 + 8 = 9
Cost(v7) = 1 + 4 = 5

S	1	2	3	4	5	6	7
{}	0	∞	∞	∞	∞	∞	∞
{1}	0	2	∞	1	∞	∞	∞
{1,4}	0	2	3	1	3	9	5

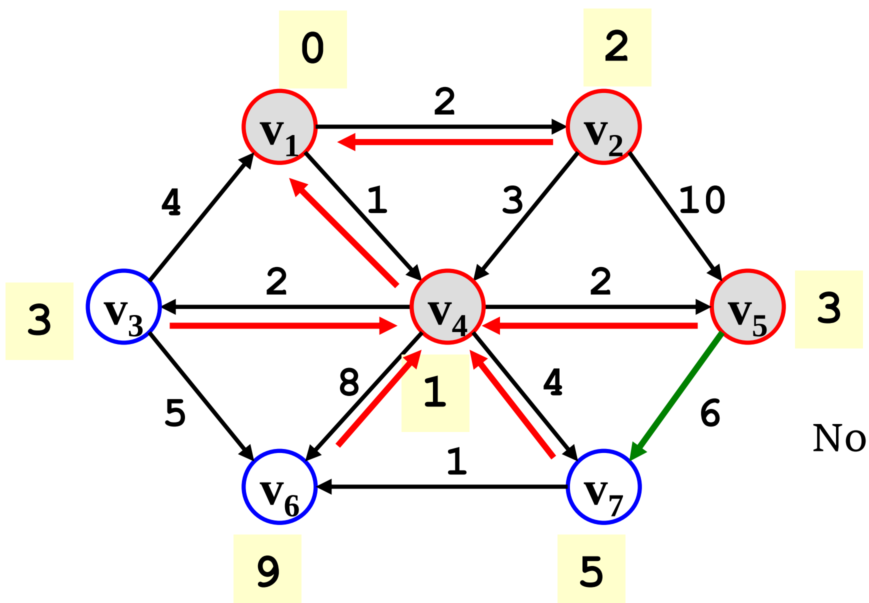
Example



update neighbors

S	1	2	3	4	5	6	7
{}	0	∞	∞	∞	∞	∞	∞
{1}	0	2	∞	1	∞	∞	∞
{1,4}	0	2	3	1	3	9	5
{1,4,2}	0	2	3	1	3	9	5

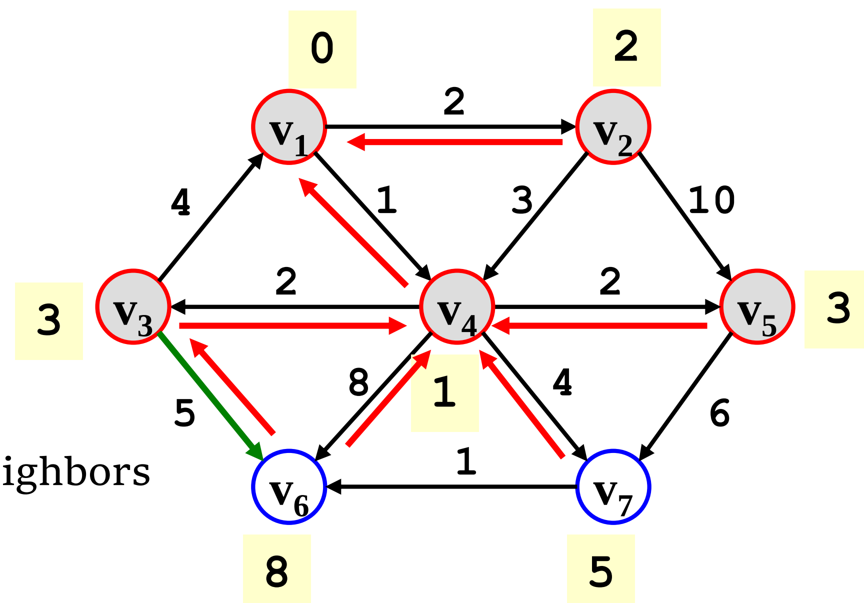
Example



No updating

S	1	2	3	4	5	6	7
{}	0	∞	∞	∞	∞	∞	∞
{1}	0	2	∞	1	∞	∞	∞
{1,4}	0	2	3	1	3	9	5
{1,4,2}	0	2	3	1	3	9	5
{1,4,2,5}	0	2	3	1	3	9	5

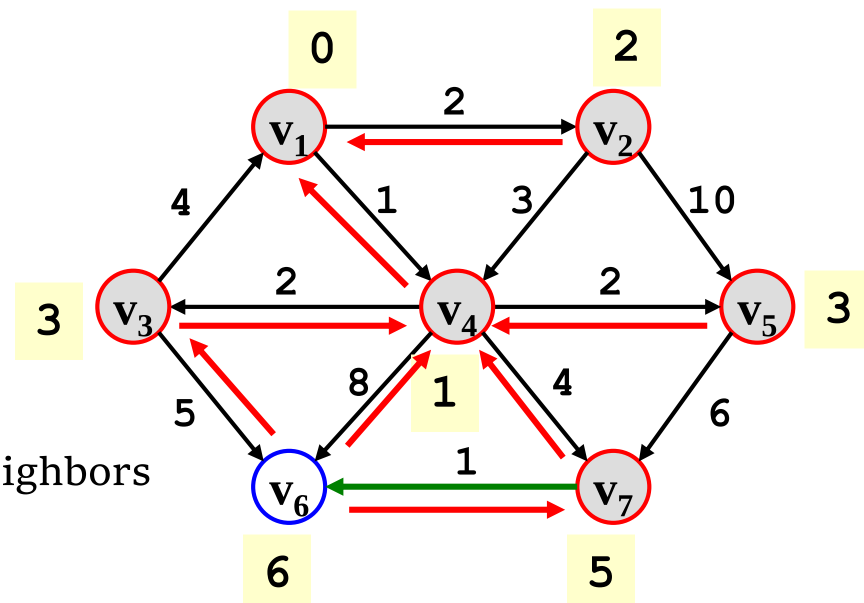
Example



update neighbors

S	1	2	3	4	5	6	7
{}	0	∞	∞	∞	∞	∞	∞
{1}	0	2	∞	1	∞	∞	∞
{1,4}	0	2	3	1	3	9	5
{1,4,2}	0	2	3	1	3	9	5
{1,4,2,5}	0	2	3	1	3	9	5
{1,4,2,5,3}	0	2	3	1	3	8	5

Example

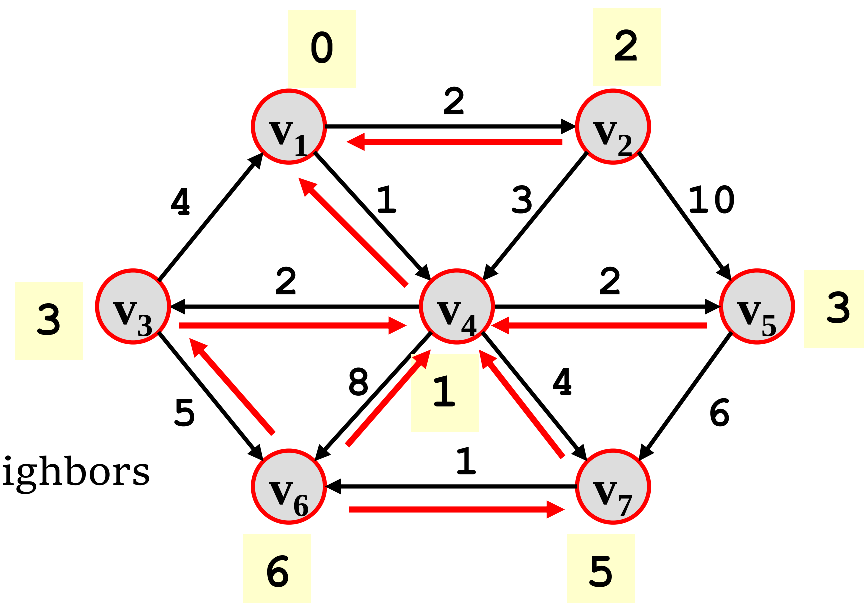


update neighbors

S	1	2	3	4	5	6	7
{}	0	∞	∞	∞	∞	∞	∞
{1}	0	2	∞	1	∞	∞	∞
{1,4}	0	2	3	1	3	9	5
{1,4,2}	0	2	3	1	3	9	5
{1,4,2,5}	0	2	3	1	3	9	5
{1,4,2,5,3}	0	2	3	1	3	8	5
{1,4,2,5,3,7}	0	2	3	1	3	6	5

Example

update neighbors



S	1	2	3	4	5	6	7
{}	0	∞	∞	∞	∞	∞	∞
{1}	0	2	∞	1	∞	∞	∞
{1,4}	0	2	3	1	3	9	5
{1,4,2}	0	2	3	1	3	9	5
{1,4,2,5}	0	2	3	1	3	9	5
{1,4,2,5,3}	0	2	3	1	3	8	5
{1,4,2,5,3,7}	0	2	3	1	3	6	5
{1,4,2,5,3,7, 6}	0	2	3	1	3	6	5

Dijkstra's Algorithm - Implementation

```
//Implementation of Dijkstra's algorithm
//compute shortest path dists from "s"
void Dijkstra(Graph* G, int* D, int s) {
 int i, v, w;
 for (int i=0; i<G->n(); i++) // Initialize
    D[i] = INFINITY;
 D[0] = 0;
 for (i=0; i<G->n(); i++) { //process vertices
  //find the unvisited vertex with min dist
  v = minVertex(G, D);
  if (D[v] == INFINITY) return; //v is unreachable

  G->setMark(v, VISITED);
  //update the distance of v's neighbors
  for (w=G->first(v); w<G->n(); w = G->next(v,w))
   if (D[w] > (D[v] + G->weight(v, w)))
     D[w] = D[v] + G->weight(v, w);
 }
}
```

Dijkstra’s Algorithm - Implementation

- **minVertex** – find the unvisited vertex with minimum distance
- Method 1: scan through the list of |V| vertices searching for the minimum value

```
int minVertex(Graph* G, int* D) {
 int i, v = -1;

 //initialize v to some unvisited vertex
 for (i=0; i<G->n(); i++)
  if (G->getMark(i) == UNVISITED) {
     v = i;
     break; }

 // Now find smallest D value
 for (i++; i<G->n(); i++)
  if ((G->getMark(i) == UNVISITED) && (D[i] < D[v]))
    v = i;
 return v;
}
```

Dijkstra's Algorithm - Implementation

- Complexity analysis of method 1
  - minVertex executes |V| times, and it scans the list of |V| vertices each time.
  - The cost is $\Theta(|V|^2)$

  - The edges are processed $\Theta(|E|)$ times, and each visit to an edge may cause a constant-time update to the array D.
  - The cost is $\Theta(|E|)$

- In total, the cost is $\Theta(|V|^2+|E|)= \Theta(|V|^2)$

Dijkstra's Algorithm – Implementation

- Method 2:
  - Store unprocessed vertices in a priority-queue (implemented using a min-heap) ordered by distance values.
  - The next-closest vertex can be found in the heap in $\Theta(\log|\mathbf{V}|)$ time
- Every time D(***x***) is updated,
  - Reordered ***x*** in the heap by deleting and reinserting it.
  - Or, add the new smaller distance value for ***x*** as a new record in the heap
    - The greater distance values found later will be ignored because the vertex will already be marked as **VISITED**

Dijkstra's Algorithm – Implementation

```
// Implementation using the priority queue
// Class for elements in the heap
Class DijkElem {
Public:
  int vertex, distance;

  DijkElem() {vertex = -1; distance = -1;}

  DijkElem(int v, int d) {vertex = v; distance = d};
};
```

```
//Implementation of Dijkstra's algorithm
void Dijkstra(Graph* G, int* D, int s) {
  int i, v, w;      // v is current vertex
  DijkElem temp;  DijkElem E[G->e()]; // Heap array

  for (int i=0; i<G->n(); i++) // Initialize
    D[i] = INFINITY;
  D[0] = 0;

  // Initialize heap array
  temp.distance = 0; temp.vertex = s;
  E[0] = temp;
  heap<DijkElem, DDComp> H(E, 1, G->e());

  // get an unvisited vertex with smallest distance
  for (i=0; i<G->n(); i++) {
    do {
      if(H.size() == 0) return; // Nothing to remove
      temp = H.removefirst(); //delmin
       v = temp.vertex;
    } while (G->getMark(v) == VISITED);

    G->setMark(v, VISITED);  //mark the vertex

    if (D[v] == INFINITY) return; //unreachable

    for(w=G->first(v); w<G->n(); w=G->next(v,w))
     if (D[w] > (D[v] + G->weight(v, w))) {
        //update D
        D[w] = D[v] + G->weight(v, w);
        temp.distance = D[w]; temp.vertex = w;
        // Insert new distance in heap
        H.insert(temp);
     }
  }
}
```

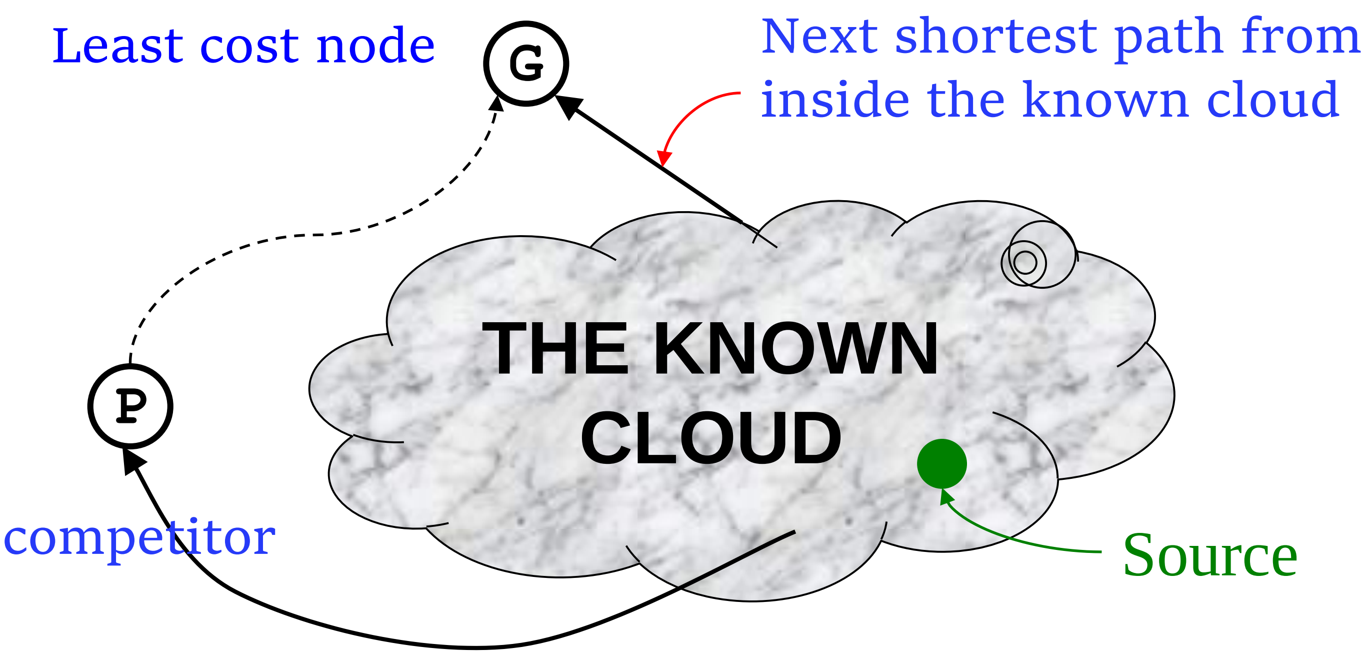
Dijkstra's Algorithm – Implementation

- Complexity analysis of method 2
  - If the vertex with a smaller distance is inserted as a new record, the cost for finding the minimum value using the min-heap becomes $\Theta(\log|\mathbf{E}|)$
  - It executes V times, and the cost is $\Theta(|\mathbf{V}|\log|\mathbf{E}|)$
  - Every visit to an edge cause an update to D and also the min-heap.
  - The cost is $\Theta(|\mathbf{E}|\log|\mathbf{E}|)$
- In total, the cost $\Theta((|\mathbf{V}|+|\mathbf{E}|)\log|\mathbf{E}|)$

Correctness

- Dijkstra’s algorithm is an example of a greedy algorithm
- Greedy algorithms always make choices that currently seem the best
  - Short-sighted – no consideration of long-term or global issues
  - Locally optimal does not always mean globally optimal
- In Dijkstra’s case – choose the least cost node, but what if there is another path through other vertices that is cheaper?

"Cloudy" Proof: The Idea



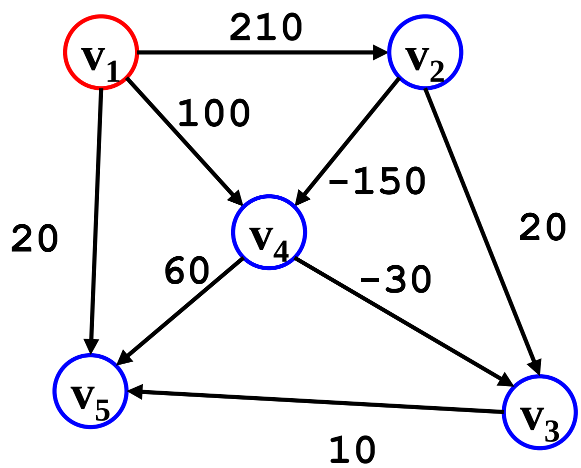
- If the path to G is the next shortest path, the path to P must be at least as long. Therefore, any path through P to G cannot be shorter!

Inside the Cloud (Proof)

- Everything inside the cloud has the correct shortest path
- Proof is by induction on the number of nodes in the cloud:
  - Base case: Initial cloud is just the source s with shortest path 0.
  - Inductive hypothesis: Assume that a cloud of k-1 nodes all have shortest paths.
  - Inductive step: choose the least cost node G → has to be the shortest path to G (previous slide). Add k-th node G to the cloud.

Graphs with Negative Edge Costs

- If the graph has negative edge costs, then Dijkstra's algorithm does not work.



S	1	2	3	4	5
{}	0	∞	∞	∞	∞
{1}	0	210	∞	100	20
{1,5}	0	210	∞	100	20
{1,5,4}	0	210	70	100	20
{1,5,4,3}	0	210	70	100	20
{1,5,4,3,2}	0	210	70	60	20

All Pairs Shortest Path

- Given a edge weighted directed graph G = (V,E) , find for all u,v in V the length of the shortest path from u to v.

- Could run the appropriate single-source algorithm |V| times.
  - On sparse graphs, it is fast to run |V| Dijkstra's algorithms coded with priority queues
  - On dense graphs, the algorithm in textbook section 10.3.4 more faster in practice

Homework 7-2

- Textbook Exercises 9.5