Data Structures

Single Source Shortest Path

CS 225
Brad Solomon

November 5, 2025

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN
Department of Computer Science


YOU ARE INVITED TO
TRICK OR RESEARCH
RESEARCH FAIR, POSTER SESSION, PARTICIPATING LABS, AND CANDY!
WEDNESDAY, NOVEMBER 5, 2025
4:00 P.M. TO 5:30 P.M.
Siebel School of Computing and Data Science
Be there and be scared!
CS STARS
STUDENT AMBASSADORS/RESEARCH SCHOLARS

Exam 4 (11/12 — 11/14)

Autograded MC and one coding question

Manually graded short answer prompt

Practice exam will be on PL

Topics covered can be found on website

Registration is open!

https://courses.engr.illinois.edu/cs225/fa2025/exams/

Learning Objectives

Finish discussion about Prim's runtime

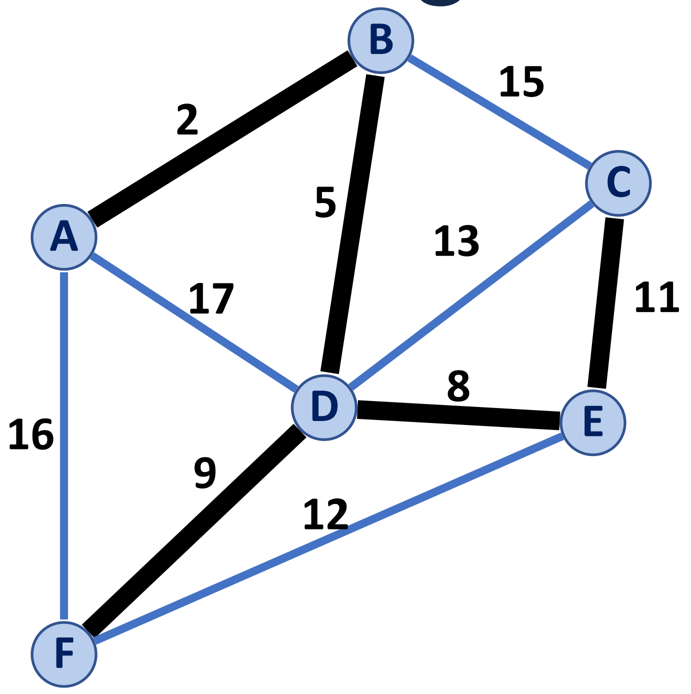
Compare Kruskal and Prim MST Algorithms

Introduce Single-Source Shortest Path Problem

Discuss Dijkstra's Algorithm

Extend to All-Paths Shortest Path (if time)

Prim's Algorithm



A	B	C	D	E	F
0, —	2, A	11, E	5, B	8, D	9, D

```
PrimMST(G, s):
  Input: G, Graph;
         s, vertex in G, starting vertex
  Output: T, a minimum spanning tree (MST) of G

  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q   // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T           // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m

  return T
```

$|V| = n, |E| = m$

Prim's Big O

7 — 9: O(n)

12—14:

MinHeap: O(n)

Unsorted Array: O(1)

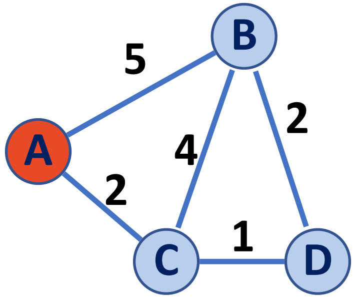
16—22: Complicated!

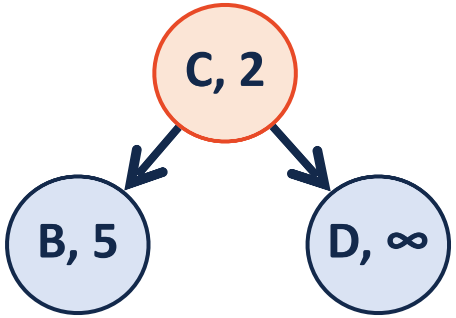
```
6  PrimMST(G, s):
7    foreach (Vertex v : G.vertices()):
8      d[v] = +inf
9      p[v] = NULL
10   d[s] = 0
11
12   PriorityQueue Q // min distance, defined by d[v]
13   Q.buildHeap(G.vertices())
14   Graph T          // "labeled set"
15
16   repeat n times:
17     Vertex m = Q.removeMin()
18     T.add(m)
19     foreach (Vertex v : neighbors of m not in T):
20       if cost(v, m) < d[v]:
21         d[v] = cost(v, m)
22         p[v] = m
23
```

Depends on choice of **PriorityQueue** (MinHeap vs Unsorted Array)

Depends on choice of **Graph** (Adjacency Matrix vs Adjacency List)

A	B	C	D
0	5	2	∞

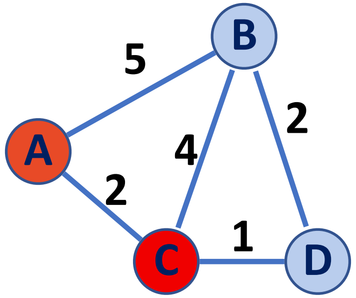


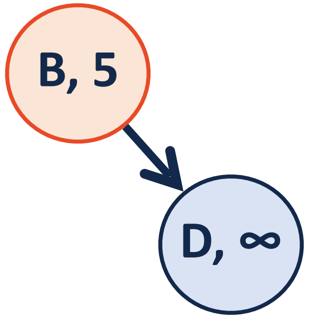



```
PrimMST(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m

```

	Adj. Matrix	Adj. List
Heap	O(n) + ___________ + O(n^2) + _______________	O(n) + ___________ + O(m) + _______________

A	B	C	D
0	5	2, A	∞

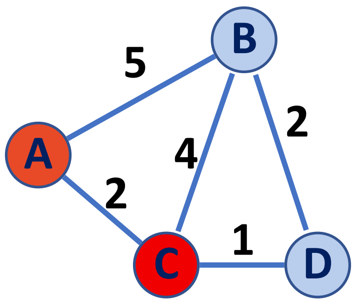


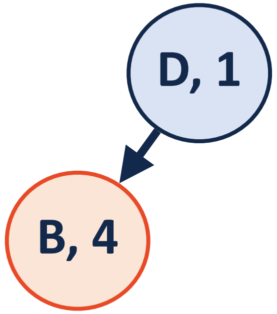



```
PrimMST(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m

```

	Adj. Matrix	Adj. List
Heap	O(n) + O(n log n) + O(n^2) + ________________	O(n) + O(n log n) + O(m) + ________________

A	B	C	D
0	4	2, A	1





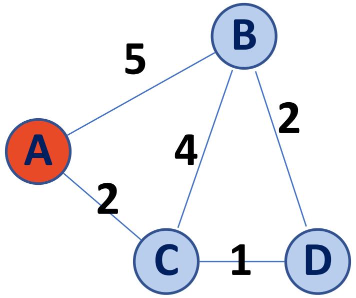

```
PrimMST(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m
```

1) Change minheap value

2) HeapifyUp()

	Adj. Matrix	Adj. List
Heap	O(n) + O(n log n) + O(n^2) + O(m log n)	O(n) + O(n log n) + O(m) + O(m log n)

(A, 0)
(D, ∞)
(C, 2)
(B, 5)




```
PrimMST(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m

```

	Adj. Matrix	Adj. List
Heap	$O(n^2 + m \lg(n))$	$O(n \lg(n) + m \lg(n))$
Unsorted Array		

Prim's Algorithm

Sparse Graph: $m \sim n$

Adj List Heap best

Dense Graph: $m \sim n^2$

Unsorted Array best

```
PrimMST(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m
```

	Adj. Matrix	Adj. List
Heap	$O(n^2 + m\lg(n))$	$O(n\lg(n) + m\lg(n))$
Unsorted Array	$O(n^2)$	$O(n^2)$

MST Algorithm Runtime:

Kruskal's Algorithm:

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

Prim's Algorithm:

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

Sparse Graph: $m \sim n$

Dense Graph: $m \sim n^2$

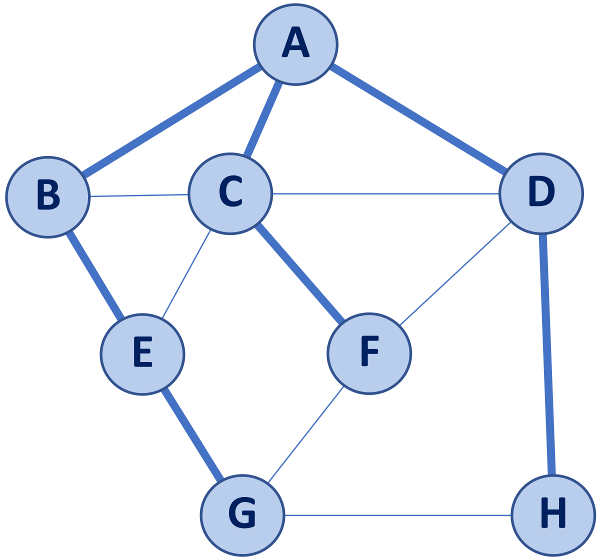
Suppose I have a new heap:

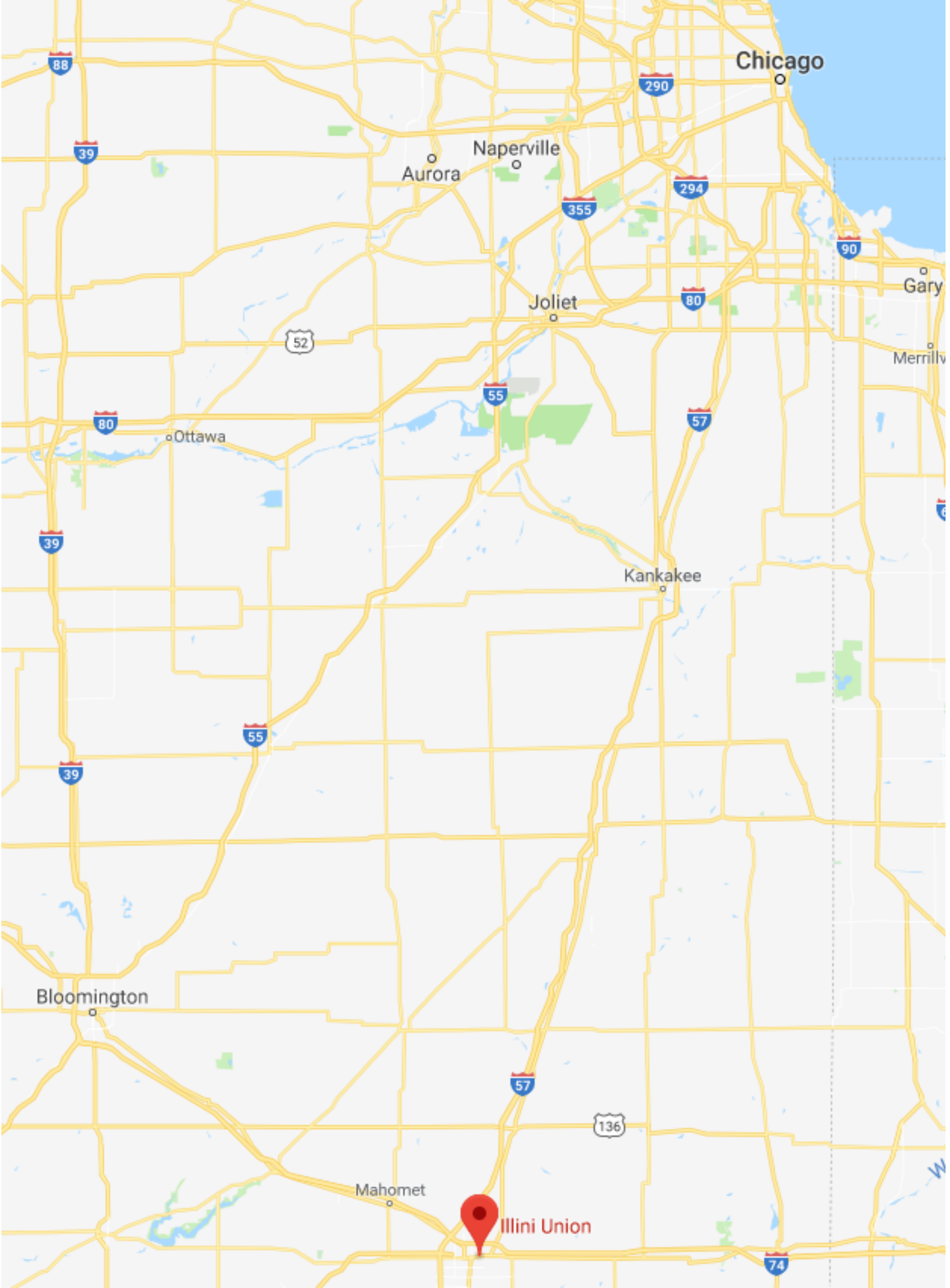
	Binary Heap	Fibonacci Heap
Remove Min	O(lg(n))	O(lg(n))
Decrease Key	O(lg(n))	O(1)*

What's Prim's updated running time?

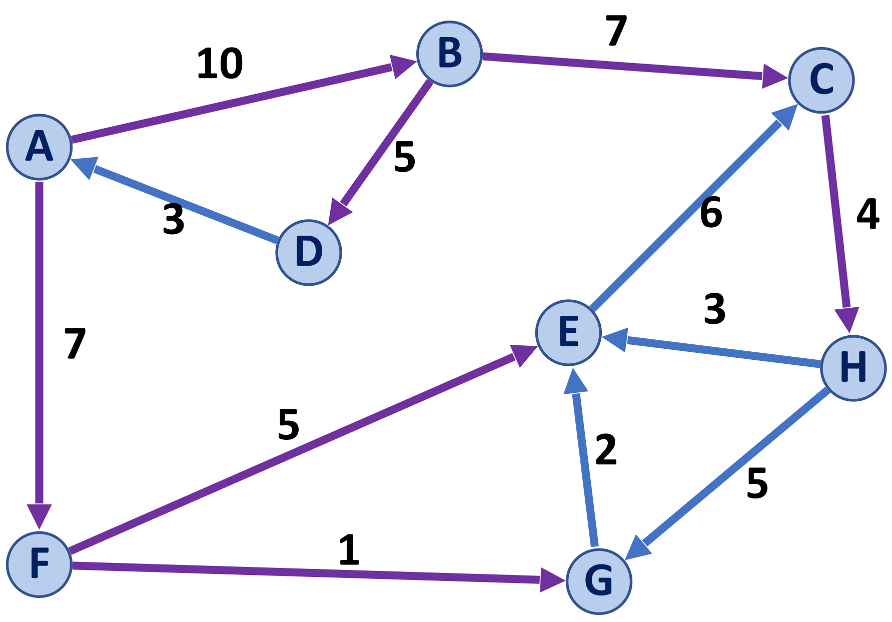
```
PrimMST(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m
```

Shortest Path





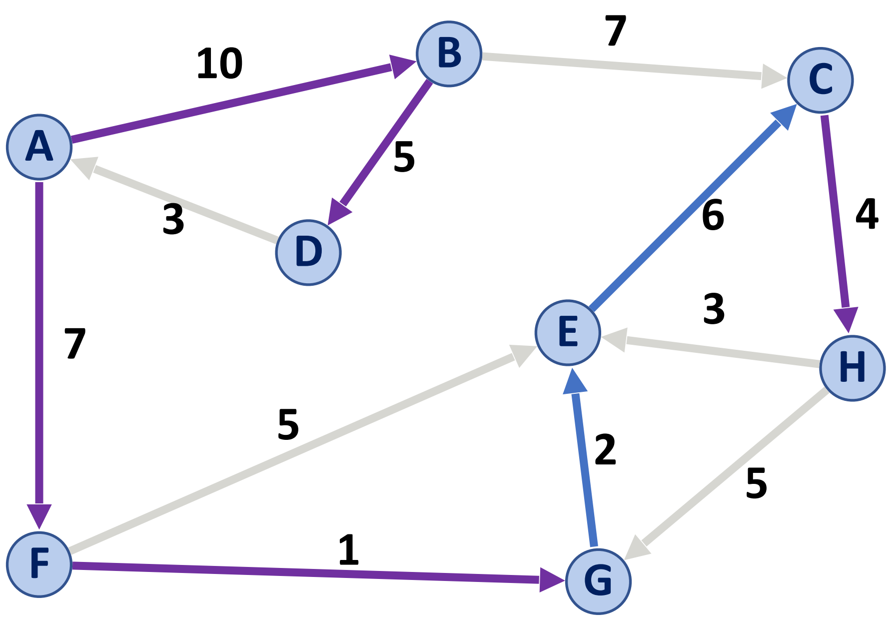
Dijkstra's Algorithm (SSSP)




```
DijkstraSSSP(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if ________________ < d[v]:
        d[v] = ________________
        p[v] = u
```

A	B	C	D	E	F	G	H
--							
0							

Dijkstra's Algorithm (SSSP)




```
DijkstraSSSP(G, s):
  foreach (Vertex v : G.vertices()):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if cost(u, v) + d[u] < d[v]:
        d[v] = cost(u, v) + d[u]
        p[v] = u
```

A	B	C	D	E	F	G	H
--	A	E	B	G	A	F	C
0	10	16	15	10	7	8	20

Dijkstra's Algorithm (SSSP)

What is the running time of Dijkstra's Algorithm?

```
DijkstraSSSP(G, s):
  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if cost(u, v) + d[u] < d[v]:
        d[v] = cost(u, v) + d[u]
        p[v] = m

  return T
```

Dijkstra's Algorithm (SSSP)

$O(m + n \log n)$

What is the running time of Dijkstra's Algorithm?

The same as Prim's!

6-9: O(n)

11-12: O(n)

15: repeat below n x

16-22: O(log n)

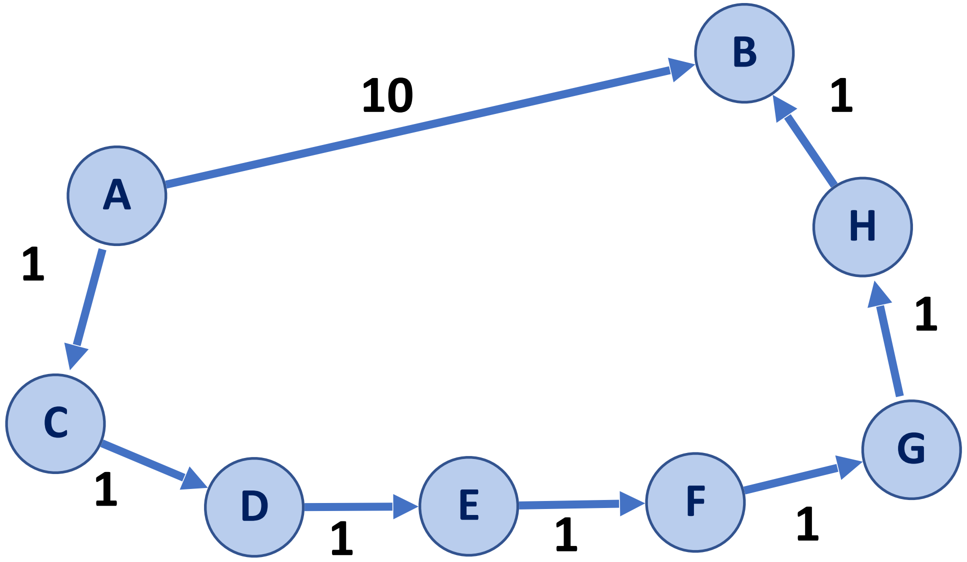
[w/ Fib Heap O(1) updates]

```
   DijkstraSSSP(G, s):
6    foreach (Vertex v : G):
7      d[v] = +inf
8      p[v] = NULL
9    d[s] = 0
10
11   PriorityQueue Q // min distance, defined by d[v]
12   Q.buildHeap(G.vertices())
13   Graph T          // "labeled set"
14
15   repeat n times:
16     Vertex u = Q.removeMin()
17     T.add(u)
18     foreach (Vertex v : neighbors of u not in T):
19       if cost(u, v) + d[u] < d[v]:
20         d[v] = cost(u, v) + d[u]
21         p[v] = m
22
23   return T
```

Dijkstra’s Algorithm (SSSP)

Claim: Dijkstras will always visit a node through its optimal shortest path.

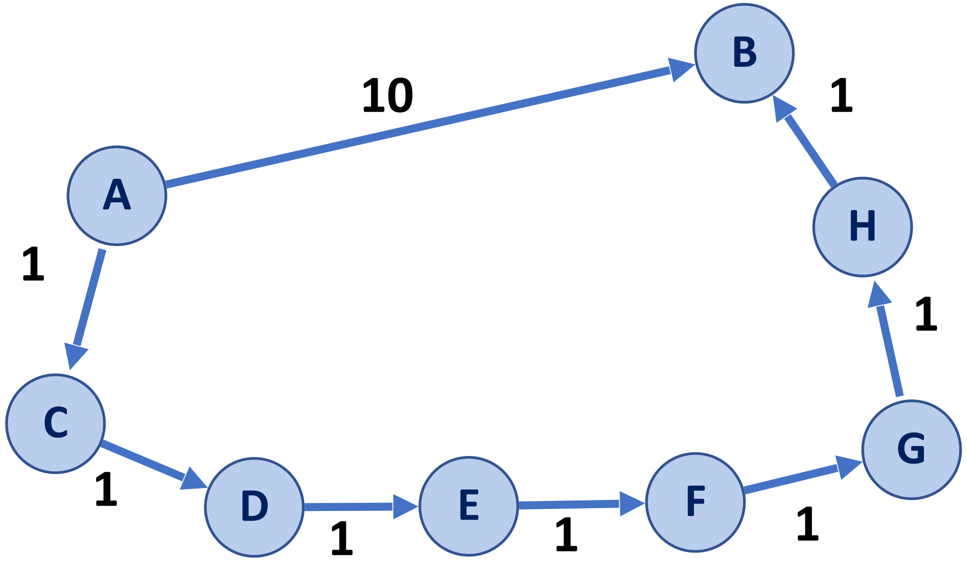
When we will visit B in the following graph?



Dijkstra's Algorithm (SSSP)

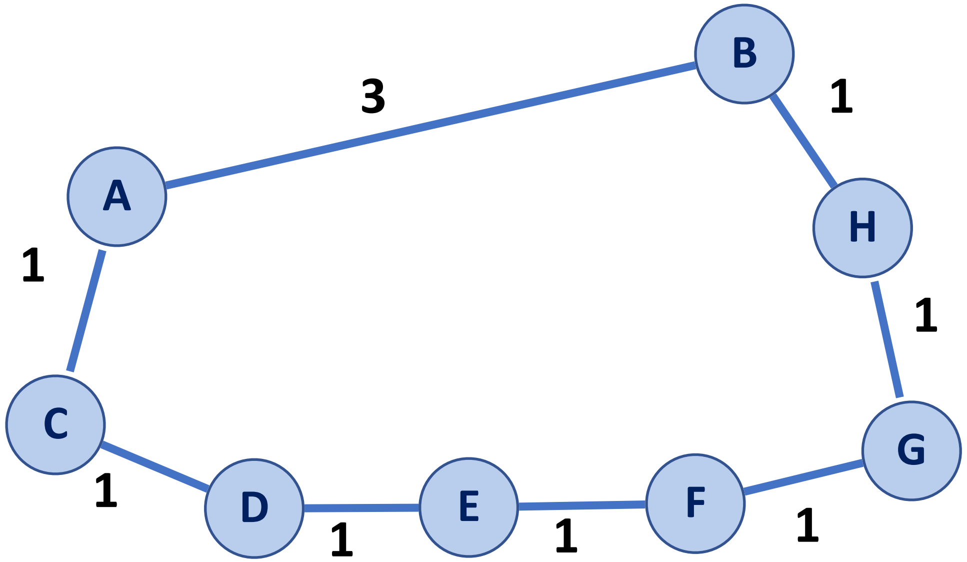
Claim: Dijkstras will always visit a node through its optimal shortest path.

When we will visit H in the following graph?



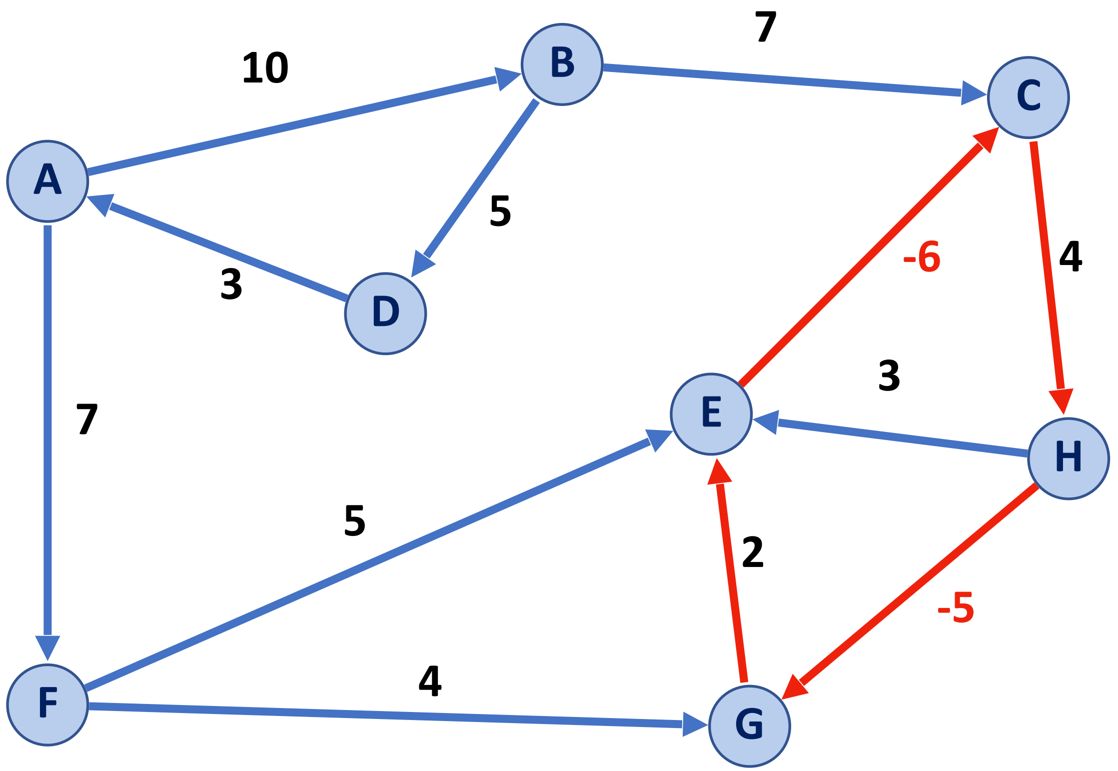
Dijkstra’s Algorithm (SSSP)

How does Dijkstra’s algorithm handle undirected graphs?



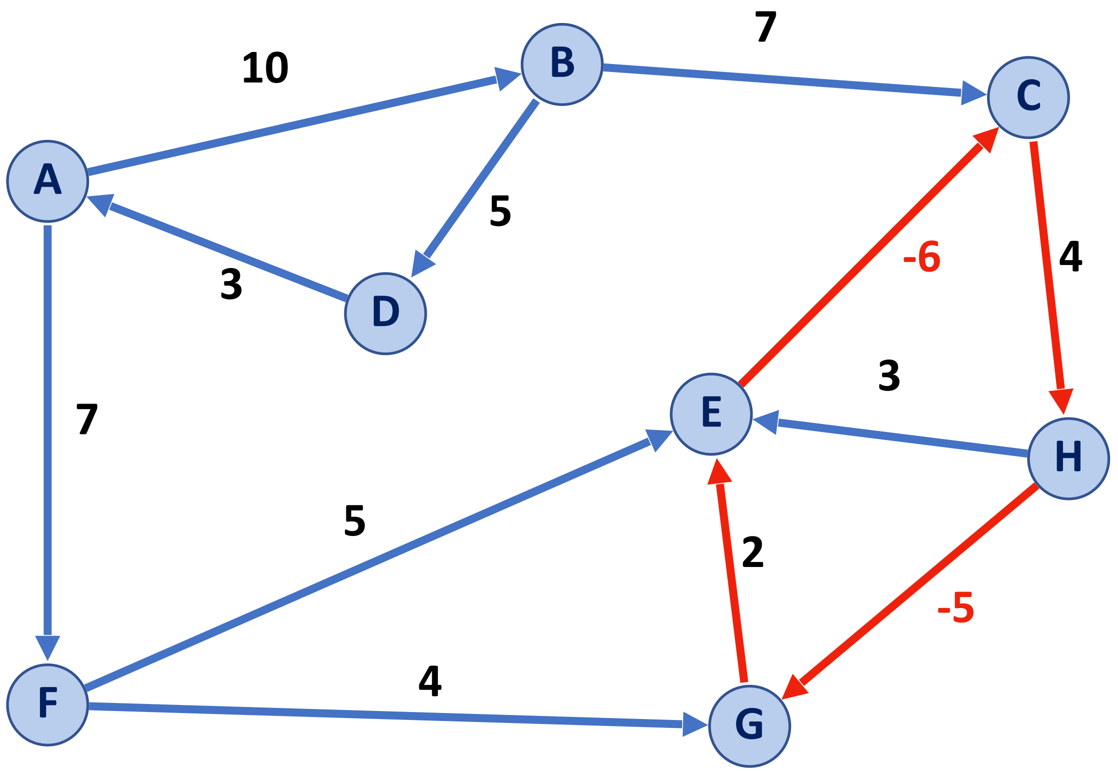
Dijkstra’s Algorithm (SSSP)

How does Dijkstras handle a negative weight cycle?



Dijkstra's Algorithm (SSSP)

How does Dijkstras handle a negative weight cycle?



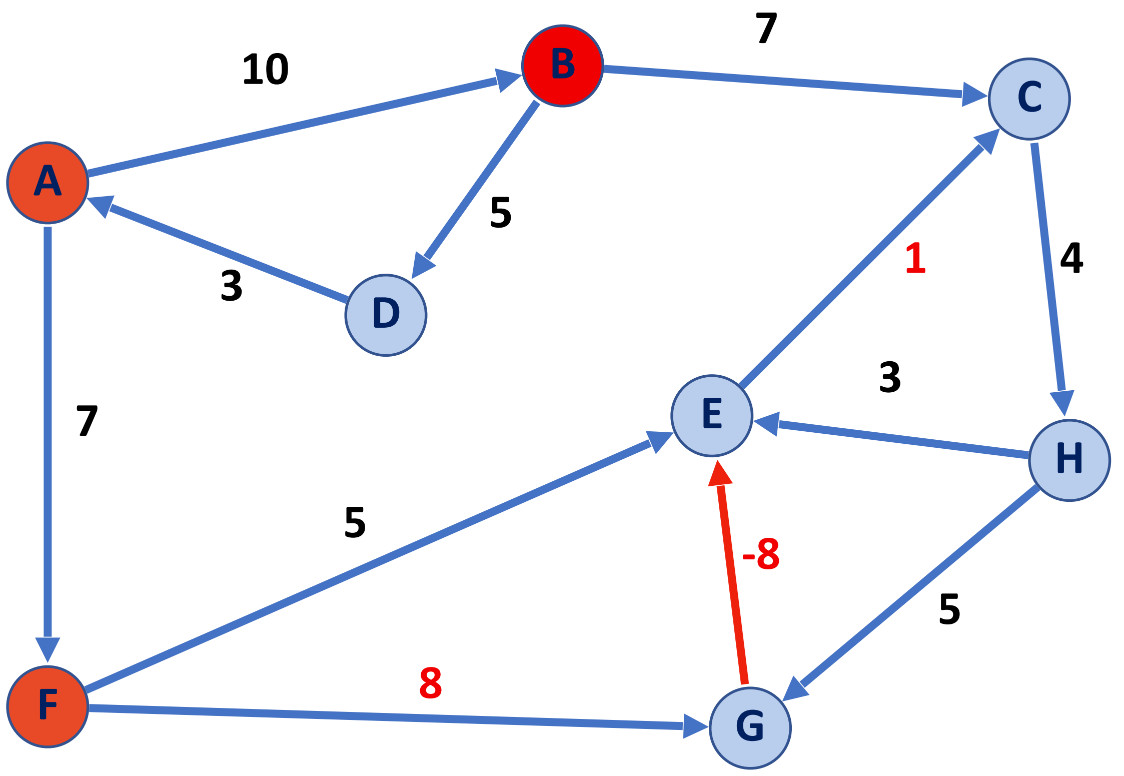
Shortest Path (A → E): A → F → E → (C → H → G → E)*

Length: 12

Length: -5 (repeatable)

Dijkstra’s Algorithm (SSSP)

How does Dijkstras handle a negative weight edge without a cycle?



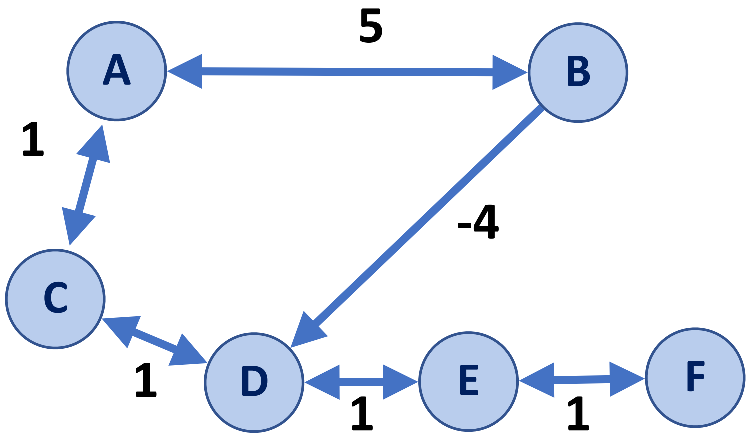
A	B	C	D	E	F	G	H
--	A	B	B	F	A	F	--
0	10	17	15	12	7	15	∞

Dijkstra's Algorithm (SSSP)

We assume that item pulled out of priority queue is **the next smallest item**

Negative weights break this assumption!

A	B	C	D	E	F
--					
0					



Dijkstra's Algorithm (SSSP)

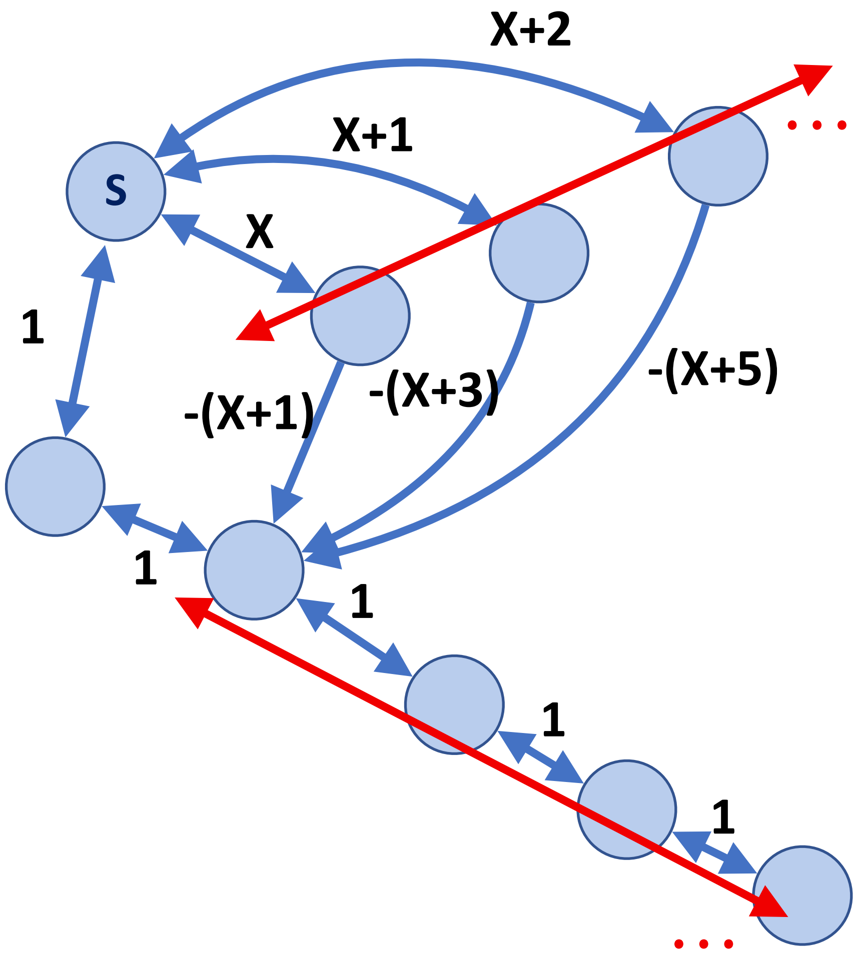
Recalculating all distances is possible, but algorithm runtime is very bad!

```
DijkstraSSSP(G, s):
  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T         // "labeled set"

  repeat until Q.empty():
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if cost(u, v) + d[u] < d[v]:
        d[v] = cost(u, v) + d[u]
        p[v] = m
        if v not in Q:
           Q.push(v)
  return T
```

Dijkstra's Algorithm (SSSP)

Recalculating all distances is possible, but algorithm runtime is very bad!

Worst case: ~n/2 nodes each updating ~n/2 nodes distances




```
DijkstraSSSP(G, s):
  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat until Q.empty():
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if cost(u, v) + d[u] < d[v]:
        d[v] = cost(u, v) + d[u]
        p[v] = m
        if v not in Q:
          Q.push(v)
  return T
```

Dijkstra’s Algorithm (SSSP)

Dijkstras Algorithm works only on non-negative weights

Optimal implementation:

Fibonacci Heap

If dense, unsorted list ties

Optimal runtime:

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

Dense: $O(n^2)$

```
DijkstraSSSP(G, s):
  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T         // "labeled set"

  repeat n times:
    Vertex u = Q.removeMin()
    T.add(u)
    foreach (Vertex v : neighbors of u not in T):
      if cost(u, v) + d[u] < d[v]:
        d[v] = cost(u, v) + d[u]
        p[v] = m

  return T
```

Landmark Path Problem

What if I wanted to get the shortest path from A to G but stopping at L along the way?