CSE 332
Data Structures & Parallelism

Shortest Paths

Melissa Winstanley
Spring 2024

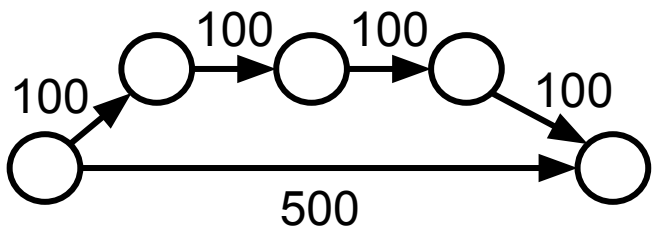
Shortest Path Applications

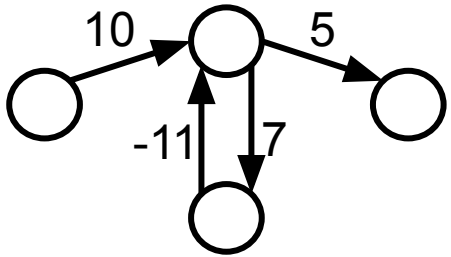
- Network routing
- Driving directions
- Cheap flight tickets
- Critical paths in project management (see textbook)
- …

BFS is great if all we care about is path length

- But what if we care about path **cost** (ie a weighted graph)?

Not as easy





Why BFS won't work: Shortest path may not have the fewest edges

- Annoying when this happens with costs of flights

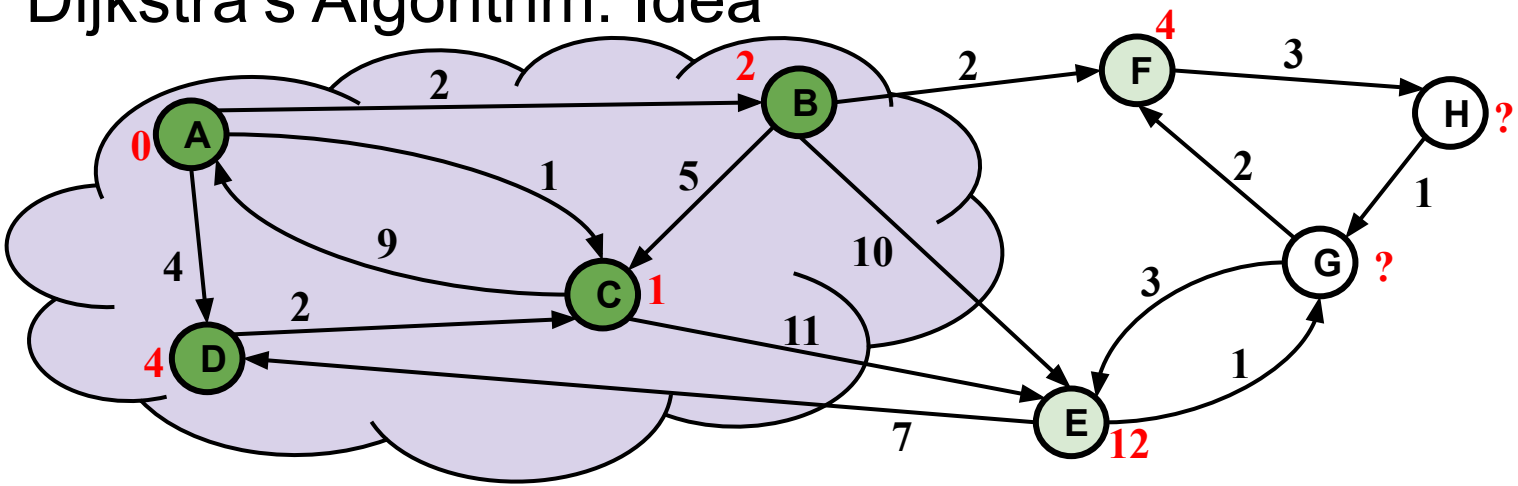
We will assume there are no negative weights

- *Problem* is *ill-defined* if there are negative-cost *cycles*
- Today's algorithm is *wrong* if edges can be negative

Dijkstra’s Algorithm

- Named after its inventor Edsger Dijkstra (1930-2002)
  - Truly one of the “founders” of computer science; 1972 Turing Award; this is just one of his many contributions
  - Sample quotation: “computer science is no more about computers than astronomy is about telescopes”
- The idea: reminiscent of BFS, but adapted to handle weights
  - Grow the set of nodes whose shortest distance has been computed
  - Nodes not in the set will have a “best distance so far”
  - A priority queue will turn out to be useful for efficiency

Dijkstra's Algorithm: Idea

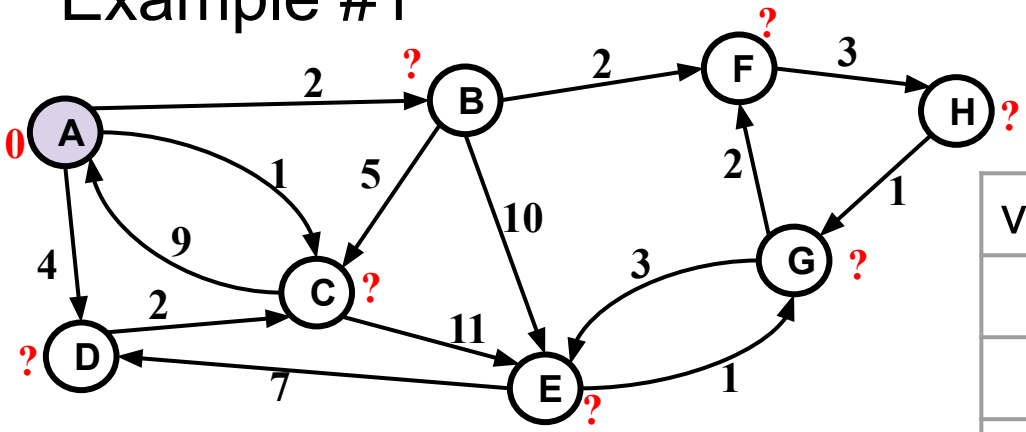


- Initially, start node has cost 0 and all other nodes have cost
- At each step:
  - Pick closest unknown vertex **v**
  - Add it to the "cloud" of known vertices
  - Update distances for nodes with edges from **v**
- That's it! (Have to prove it produces correct answers)

The Algorithm

```
dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  while(not all nodes are known) {
    b = find unknown node with smallest cost
    b.known = true
    for each edge (b,a) in G
      if(!a.known)
        if(b.cost + weight((b,a)) < a.cost){
          a.cost = b.cost + weight((b,a))
          a.pred = b
        }
  }
}
```

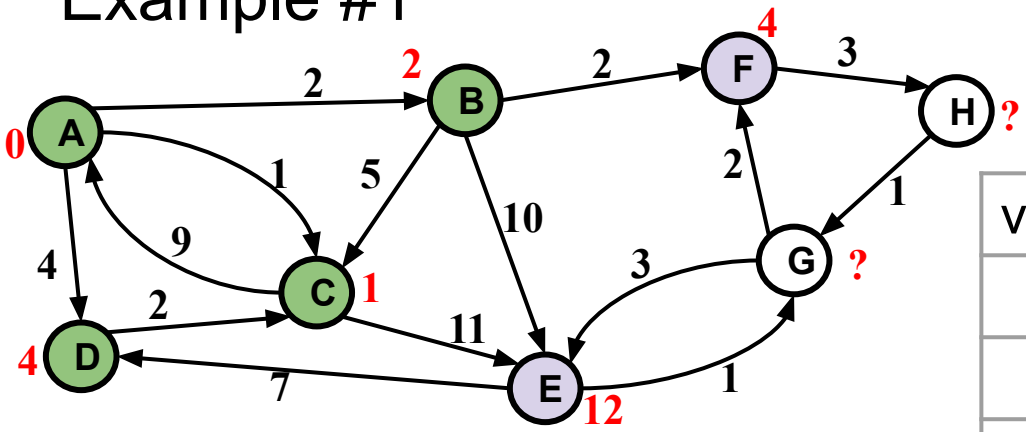
Example #1



Order added to known set:

vertex	known?	cost	pred
A			
B			
C			
D			
E			
F			
G			
H			

Example #1

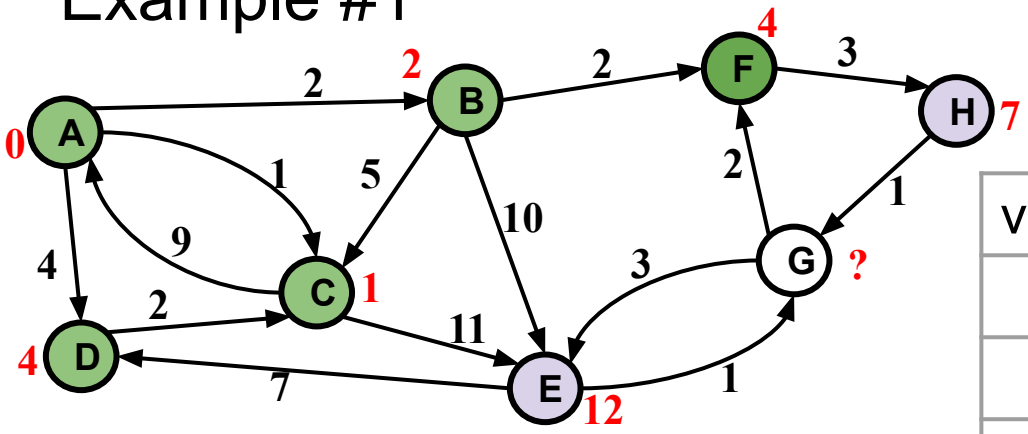


vertex	known?	cost	pred
A	T	0	
B	T	2	A
C	T	1	A
D	T	4	A
E		≤ 12	C
F		≤ 4	B
G		∞	
H		∞	

Order added to known set:

A, C, B, D

Example #1

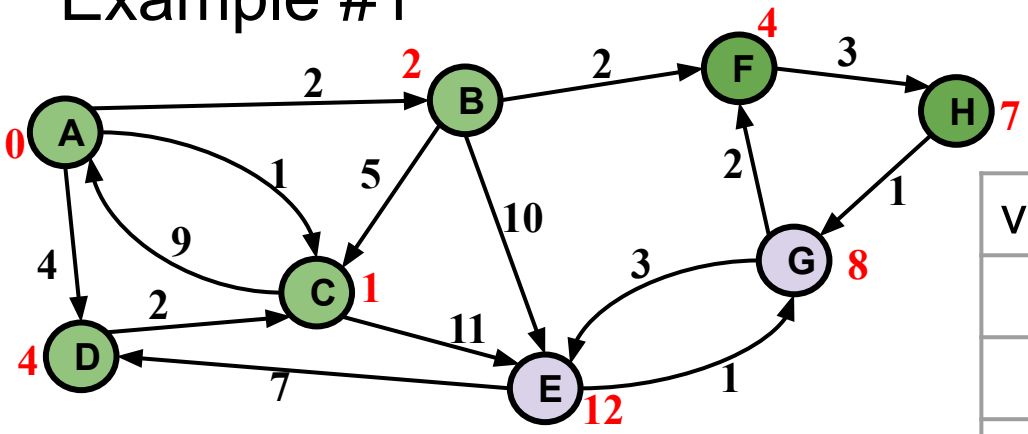


vertex	known?	cost	pred
A	T	0	
B	T	2	A
C	T	1	A
D	T	4	A
E		≤ 12	C
F	T	4	B
G		∞	
H		~~∞~~ ≤ 4+3 = 7	F

Order added to known set:

A, C, B, D, **F**

Example #1

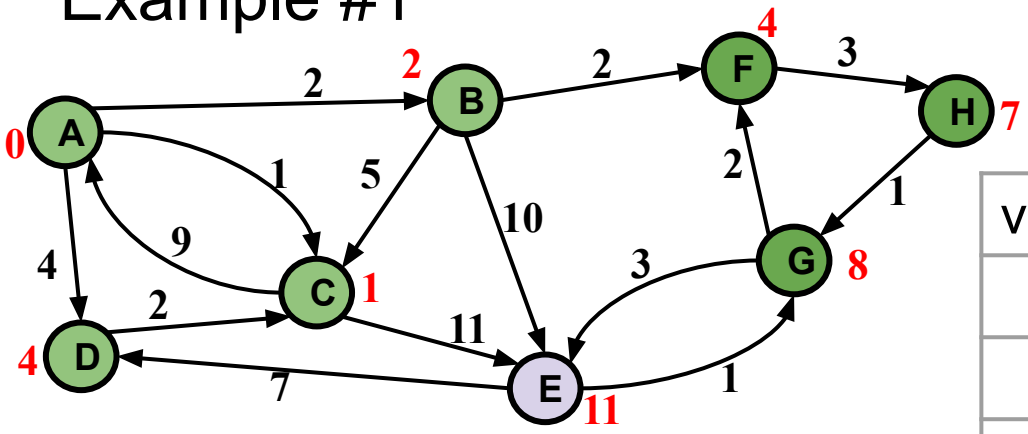


vertex	known?	cost	pred
A	T	0	
B	T	2	A
C	T	1	A
D	T	4	A
E		≤ 12	C
F	T	4	B
G		~~∞~~ ≤ 7+1 = 8	H
H	T	7	F

Order added to known set:

A, C, B, D, F, **H**

Example #1

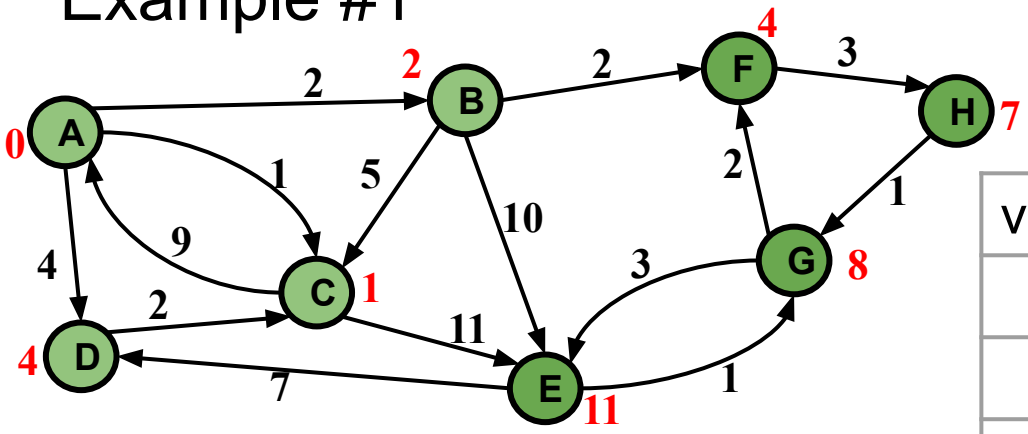


Order added to known set:

A, C, B, D, F, H, **G**

vertex	known?	cost	pred
A	T	0	
B	T	2	A
C	T	1	A
D	T	4	A
E		~~12~~ ≤ 8+3=11	G
F	T	4	B
G	T	8	H
H	T	7	F

Example #1

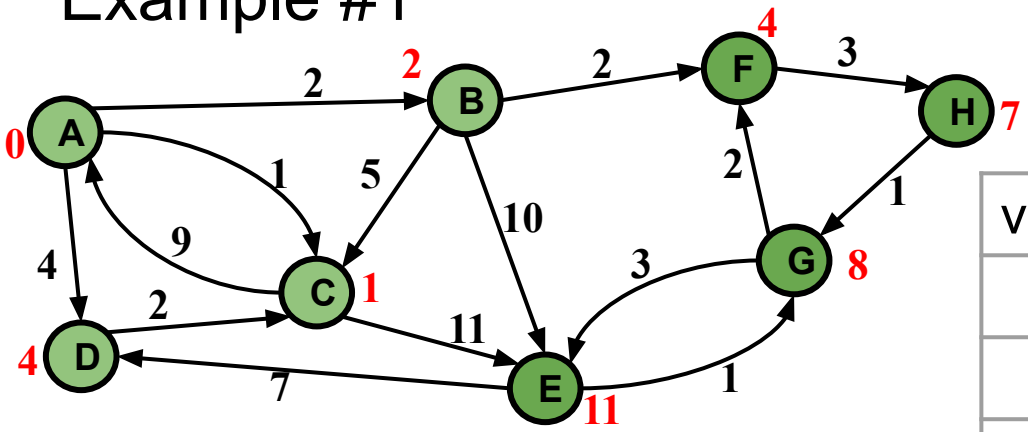


Order added to known set:

A, C, B, D, F, H, G, **E**

vertex	known?	cost	pred
A	T	0	
B	T	2	A
C	T	1	A
D	T	4	A
E	T	11	G
F	T	4	B
G	T	8	H
H	T	7	F

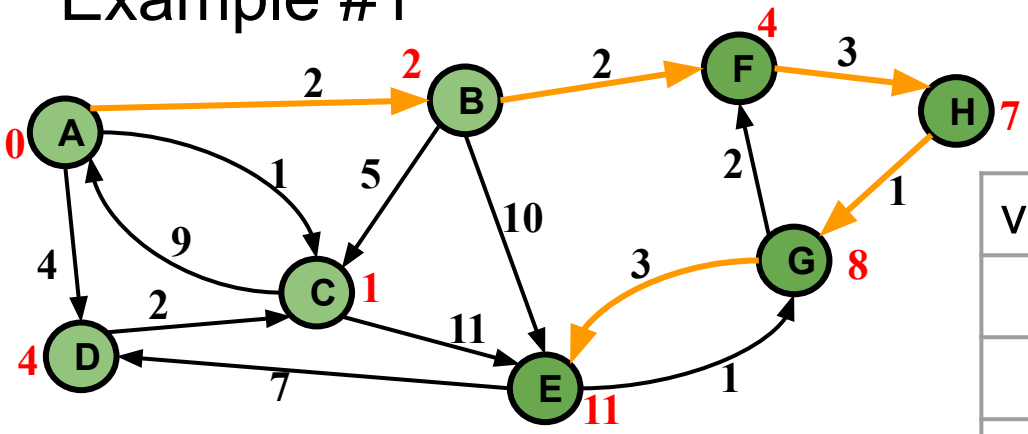
Example #1



Path from A to E:

vertex	known?	cost	pred
A	T	0	
B	T	2	A
C	T	1	A
D	T	4	A
E	T	11	G
F	T	4	B
G	T	8	H
H	T	7	F

Example #1



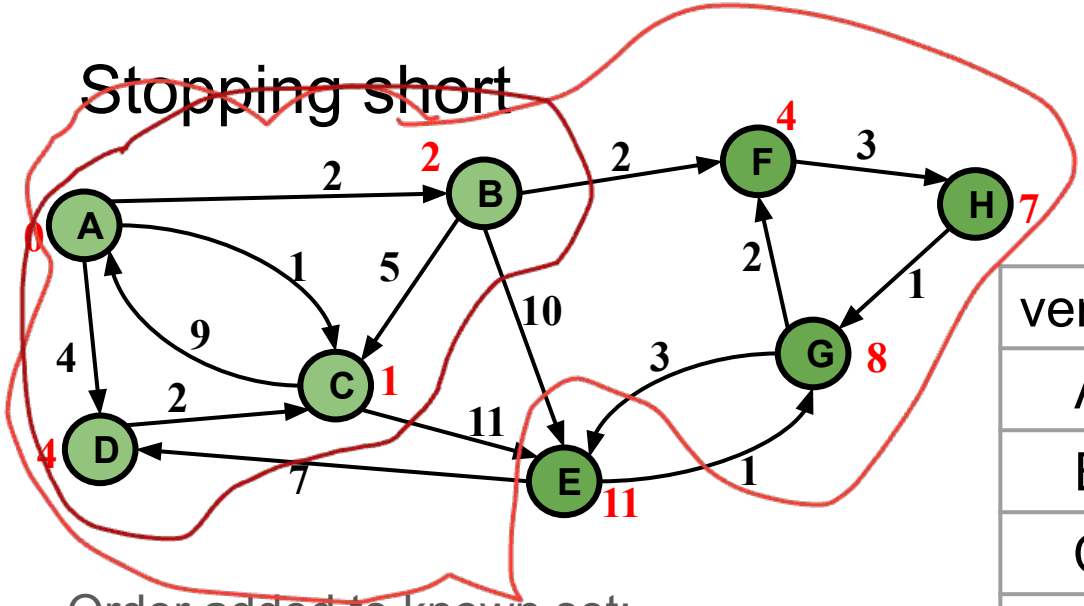
Path from A to E:

vertex	known?	cost	pred
A	T	0	
B	T	2	A
C	T	1	A
D	T	4	A
E	T	11	G
F	T	4	B
G	T	8	H
H	T	7	F

Stopping short

- How would this have worked differently if we were only interested in:
  - The path from A to G?
  - The path from A to D?

Stopping short



Order added to known set:

A, C, B, D, F, H, G, **E**

How would this have worked differently if we were only interested in:

- The path from A to G?
- The path from A to D?

vertex	known?	cost	pred
A	T	0	
B	T	2	A
C	T	1	A
D	T	4	A
E	T	11	G
F	T	4	B
G	T	8	H
H	T	7	F

A Greedy Algorithm

- Dijkstra's algorithm
  - For single-source shortest paths in a weighted graph (directed or undirected) with no negative-weight edges

- An example of a **greedy algorithm**:
  - At each step, irrevocably does what seems best at that step
    - A locally optimal step, not necessarily globally optimal
  - Once a vertex is known, it is not revisited
    - Turns out to be globally optimal

Greedy failure example

Making change

Use smallest # of coins to make 15 cents

Options: 25, 10, 5, 1

10, 5

Use smallest # of coins to make 15 cents

Options: 25, 13, 10, 5, 1

7 1

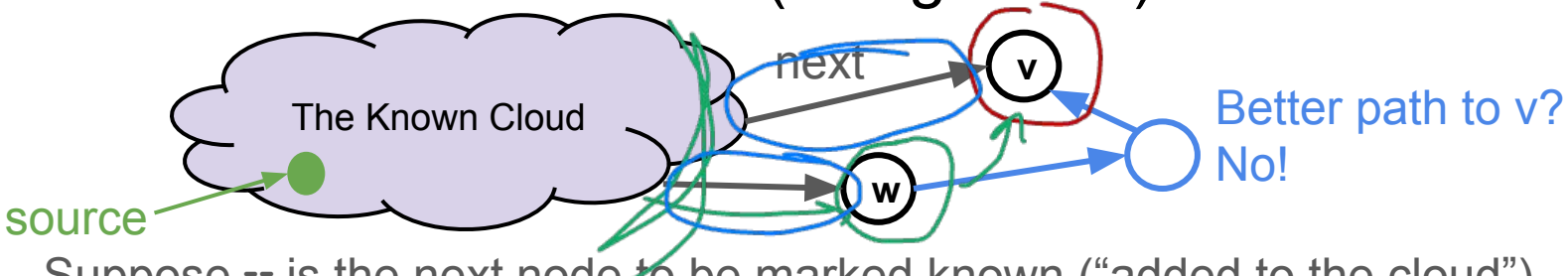
13, 1, 1

Where are we?

What should we do after learning an algorithm?

- Prove it is correct
  - Not obvious!
  - We will sketch the key ideas
- Analyze its efficiency
  - Will do better by using a data structure we learned earlier!

Correctness: The Cloud (Rough Idea)



Suppose **v** is the next node to be marked known ("added to the cloud")

- The best-known path to **v** must have only nodes "in the cloud"
  - Since we've selected it, and we only know about paths through the cloud to a node right outside the cloud
- Assume the actual shortest path to **v** is different
  - It won't use only cloud nodes, (or we would know about it), so it must use non-cloud nodes
  - Let **w** be the first non-cloud node on this path.
  - The part of the path up to **w** is already known and must be shorter than the best-known path to **v**. So **v** would not have been picked.

Contradiction!

Assume adjacency list

The Algorithm - asymptotic running time

```
dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  while(not all nodes are known) {
    b = find unknown node with smallest cost
    b.known = true
    for each edge (b,a) in G
      if(!a.known)
        if(b.cost + weight((b,a)) < a.cost){
          a.cost = b.cost + weight((b,a))
          a.pred = b
        }
  }
}
```

Assume adjacency list

The Algorithm - asymptotic running time

```
dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  while(not all nodes are known) {
    b = find unknown node with smallest cost
    b.known = true
    for each edge (b,a) in G
      if(!a.known)
        if(b.cost + weight((b,a)) < a.cost){
          a.cost = b.cost + weight((b,a))
          a.pred = b
        }
  }
}
```

O(V)

V times

O(V)

d times

Constant stuff

Assume adjacency list

The Algorithm - asymptotic running time

```
dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  while(not all nodes are known) {
    b = find unknown node with smallest cost
    b.known = true
    for each edge (b,a) in G
      if(!a.known)
        if(b.cost + weight((b,a)) < a.cost){
          a.cost = b.cost + weight((b,a))
          a.pred = b
        }
  }
}
```

O(V) plus constant

V times

O(V)

d times

Constant stuff

$V + c_1 + V*(V + c_2 + d*c_3) \Rightarrow V + V^2 + V + V*d \Rightarrow O(V^2 + E)$

Improving asymptotic running time

- So far: $O(|V|^2 + |E|)$

- Due to each iteration looking for the node to process next
  - We solved it with a queue of zero-degree nodes
  - But here we need the lowest-cost node and costs can change as we process edges

- Solution?

Assume adjacency list

Efficiency, second approach

```
dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  build-heap with all nodes
  while(heap is not empty) {
    b = deleteMin()
    b.known = true
    for each edge (b,a) in G
      if(!a.known)
        if(b.cost + weight((b,a)) < a.cost){
          decreaseKey(a, "new cost - old cost")
          a.pred = b
        }
  }
}
```

O(V)

log(V)

logV

Assume adjacency list

Efficiency, second approach

```
dijkstra(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  build-heap with all nodes
  while(heap is not empty) {
    b = deleteMin()
    b.known = true
    for each edge (b,a) in G
      if(!a.known)
        if(b.cost + weight((b,a)) < a.cost){
          decreaseKey(a, "new cost - old cost")
          a.pred = b
        }
  }
}
```

$V + c_1 + V + V * (\log V + c_2 + d*(c_3 + \log V)) \Rightarrow V + V\log V + V + E(\log V)$

$\Rightarrow O(V \log V + E \log V)$

Dense vs sparse again

- First approach: $O(|V|^2 + |E|)$ or: $O(|V|^2)$
- Second approach: $O(|V|\log|V| + |E|\log|V|)$

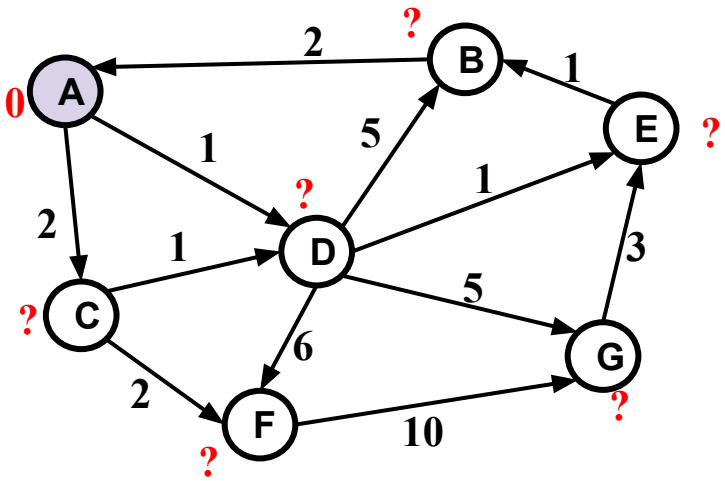
So which is better?

- Sparse: $O(|V|\log|V| + |E|\log|V|)$ (if $|E| > |V|$, then $O(|E|\log|V|)$)
- Dense: $O(|V|^2 + |E|)$, or: $O(|V|^2)$

But, remember these are worst-case and asymptotic

- Priority queue might have slightly worse constant factors
- On the other hand, for “normal graphs”, we might call `decreaseKey` rarely (or not percolate far), making $|E|\log|V|$ more like $|E|$

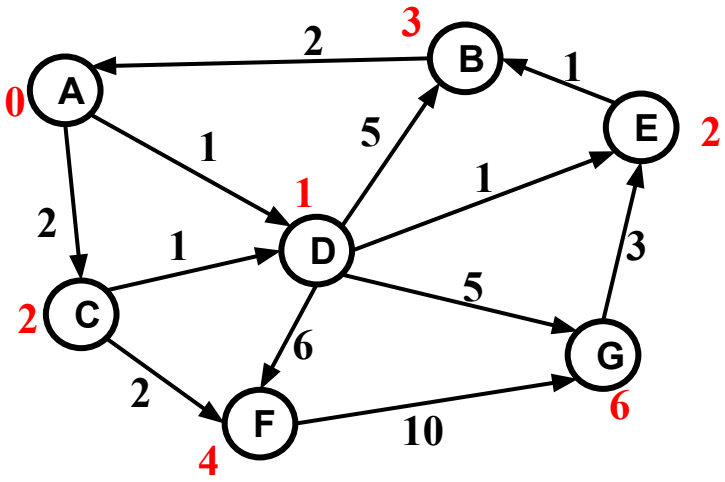
Example #2



Order added to known set:

vertex	known?	cost	pred
A		0	
B			
C			
D			
E			
F			
G			

Example #2



Order added to known set:

A, D, C, E, B, F, G

vertex	known?	cost	pred
A	T	0	
B	T	3	E
C	T	2	A
D	T	1	A
E	T	2	D
F	T	4	C
G	T	6	D

But what about negative weight edges?

- We said that Dijkstra's algorithm doesn't work for **negative-weight edges**
- But what if we want to support that?

- Problem: negative-weight edges ruin our correctness proof
  - The shortest path might involve nodes outside "the cloud"
- Solution: just do our edge calculations for all edges, but |V|-1 times
  - That way we MUST consider all paths that contain all nodes

- Enter Bellman-Ford

Bellman-Ford algorithm

O(VE)
→O(V³)

```
bellmanFord(Graph G, Node start) {
  for each node: x.cost=infinity, x.known=false
  start.cost = 0
  for (i = 0; i < |V| - 1)
    for each edge (b,a) in G
      if(b.cost + weight((b,a)) < a.cost){
        a.cost = b.cost + weight((b,a))
        a.pred = b
      }

  // Relax one more time to find a cycle
  for each edge (b,a) in G
    if(b.cost + weight((b,a)) < a.cost)
      // we found a cycle!
}
```

V
E
relax