Graph Traversals and Dijkstra's Algorithm

CSE 332 Spring 2021

Instructor: Hannah C. Tang

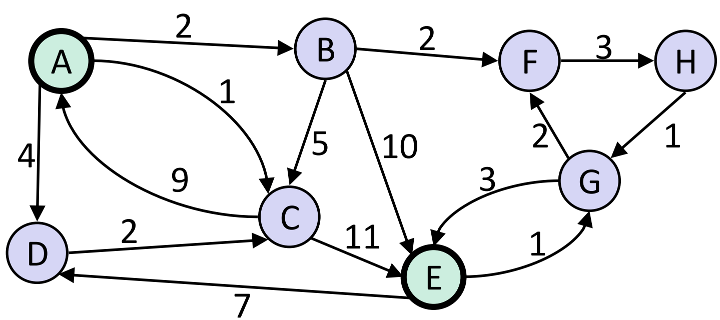
Teaching Assistants:

Aayushi Modi	Khushi Chaudhari	Patrick Murphy
Aashna Sheth	Kris Wong	Richard Jiang
Frederick Huyan	Logan Milandin	Winston Jodjana
Hamsa Shankar	Nachiket Karmarkar	

gradescope

gradescope.com/courses/256241

- Find the shortest path from A to E …
  - … assuming this graph is unweighted
  - … assuming this graph is weighted
  - (don't worry about finding a general algorithm; just find the path manually)



Lecture Outline

- **Topological Sort (cont.)**

- Traversals
  - Introduction
  - Trees and Graphs: Level-order / Breadth-first
  - Trees: Three Flavors of Depth-first
  - Graphs: Depth-first
  - Conclusion

- Shortest Paths!

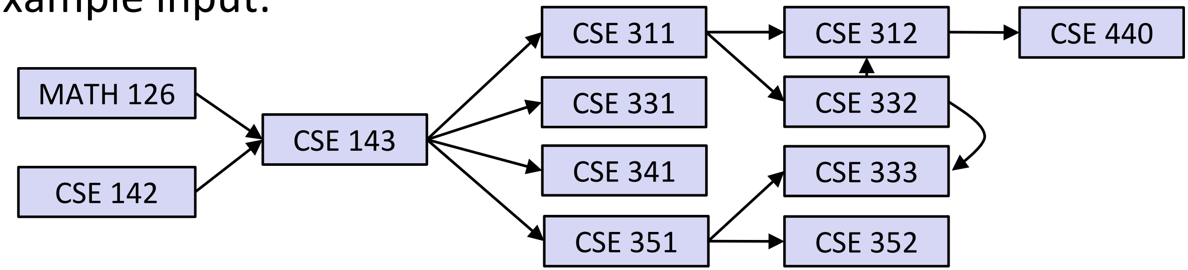
- Dijkstra's Algorithm

Topological Sort

Disclaimer: Do not use for official advising purposes!
Falsely implies CSE 332 is a prereq for CSE 312, etc.

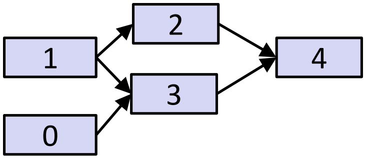
- Output all the vertices of a DAG in an order such that no vertex appears before any other vertex that has a path to it
  - A DAG represents a ***partial order***, and a topological sort produces a ***total order*** that is consistent with it

- Example input:

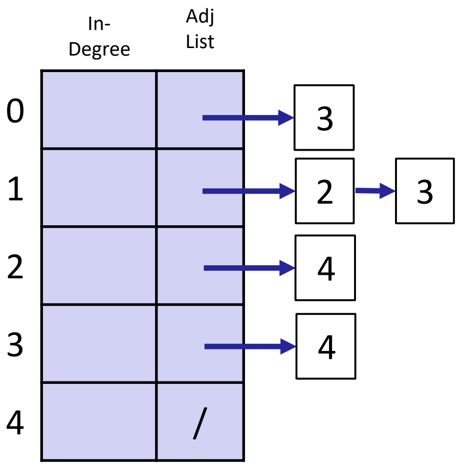


- Example output:
  - 126, 142, 143, 311, 331, 332, 312, 341, 351, 333, 352, 440

TopoSort: A Naïve Algorithm



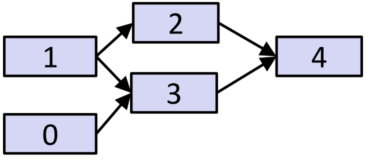
1. Label ("mark") each vertex with its in-degree
   - Could write directly into a vertex's field or a parallel data structure (e.g., array)
2. While there are vertices not yet output:
   - Choose a vertex v with labeled with in-degree of 0
   - Output v and conceptually remove it from the graph
   - Foreach vertex w adjacent to v:
     - Decrement the in-degree of w



TopoSort: Notes

- Needed a vertex with in-degree of 0 to start
  - Remember: graph must be acyclic!
- If >1 vertex with in-degree=0, can break ties arbitrarily
  - Potentially many different correct orders!

Naïve TopoSort: Running Time?




```
labelEachVertexWithItsInDegree();
for (i=0; i < numVertices; i++){
  v = findNewVertexOfDegreeZero();
  put v in output
  foreach w adjacent to v
    w.indegree--;
}
```

do this block |V| times

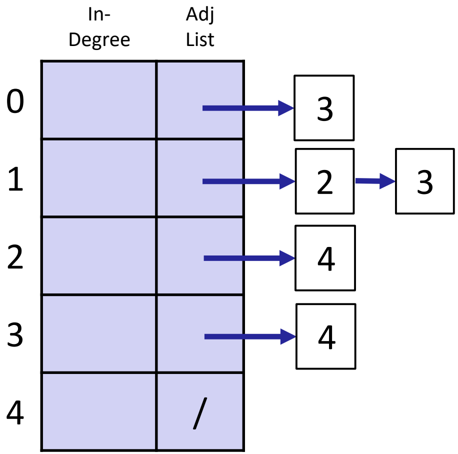
① $|V|+|E|$

② $|V|^2$ ← |V| work, |V| times

③ $|V|$ ← $\Theta(1)$ work, |V| times

④ $|E|$ ← $O(|V|^2)$, but $\Theta(E)$ is tighter (since once per edge)

$O(|V|^2+|E|)$

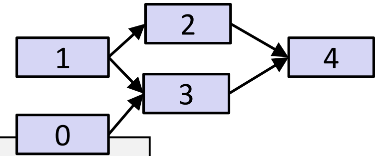


TopoSort's Runtime: Doing Better

- Avoid searching for a zero-degree node every time!
  - Keep the "pending" 0-degree nodes in a list, stack, queue, table, etc
  - The order we process them affects output, but not correctness or efficiency (*as long as add/remove are both O(1)*)

- Using a queue:
  - Label each vertex with its in-degree, enqueuing 0-degree nodes
  - While "pending" queue is not empty:
    - v = dequeue()
    - Output v and remove it from the graph
    - For each vertex w adjacent to v (i.e. w such that (v,w) in E):
      - decrement the in-degree of w
      - if new degree is 0, enqueue it

Better TopoSort: Running Time?




```
① pending = labelAllAndReturnZeros();
  while ( !pending.empty() ){
②   v = pending.dequeue();
③   put v in output
④   foreach w adjacent to v
      w.indegree--;
      if (w.indegree == 0)
        pending.enqueue(w);
  }
```

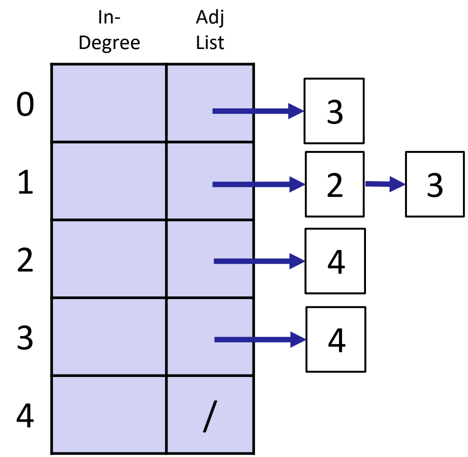
① |V| + |E|

② |V| ← Θ(1) work, |V| time

③ |V| ← same as above

④ |E| ← same as befor

O(|V| + |E|)



Lecture Outline

- Topological Sort (cont.)
- Traversals
  - **Introduction**
  - Trees and Graphs: Level-order / Breadth-first
  - Trees: Three Flavors of Depth-first
  - Graphs: Depth-first
  - Conclusion
- Shortest Paths!
- Dijkstra's Algorithm

Tree and Graph Reachability

- Find all vertices ***reachable*** from a starting vertex v
  - ie, there exists a path
  - Might "do something" at each visited vertex (an iterator!)
    - "Do something" is called ***visiting*** or ***processing*** a vertex
      - eg, print to output, set some field, etc.
    - ***Traversing*** a vertex or ***iterating*** over a vertex is different!
      - Just fetch adjacent/child vertices

- Related Questions:
  - Is an undirected graph connected?
  - Is a directed graph weakly / strongly connected?
    - For strongly, need a cycle back to starting vertex *for each vertex in the graph*

Tree and Graph Traversals

- Can answer reachability with a tree or graph ***traversal***
  - Iterates over every vertex in a graph in some defined ordering
  - "Processes" or "visits" its contents

- There are several types of tree traversals
  - Level Order Traversal aka Breadth-First Traversal
  - Depth-First Traversal
    - Pre-order Traversal
    - In-order Traversal
    - Post-order Traversal

Tree/Graph Traversals Follow a Pattern

1. Initialization:
   - Create an empty data structure to track "remaining work"
   - Mark start as visited
2. While we still have work, follow the vertices:
   3. Get a vertex
   4. Visit/process that vertex
   5. Update its neighbors (eg, add to "remaining work" if it's not already there)

order depends on algo

```
traverseGraph(Vertex start) {
  pending = {start}
  mark start as visited

  while (!pending.empty()) {
    next = pending.remove()
    process(next)
    foreach u adjacent to next
      if (!u.marked)
        mark u
        pending.add(u)
}
```

Tree/Graph Traversal: Running Time

- Assuming add() and remove() are $O(1)$, traversal is $O(|E|)$
  - Remember: we default to using an adjacency list

Tree/Graph Traversal: Order

- The order we process() depends *entirely* on how pending.add() and pending.remove() are implemented
  - Queue:
    - Tree: Level-order
    - Graph: Breadth-first search (BFS)
  - Stack:
    - Tree: Depth-first (3 flavors!)
    - Graph: Depth-first search (DFS)
  - … and more?

- DFS and BFS are "big ideas" in computer science
  - Depth: explore one part before exploring other unexplored parts
  - Breadth: explore parts closer to the start before exploring farther parts

Lecture Outline

- Topological Sort (cont.)

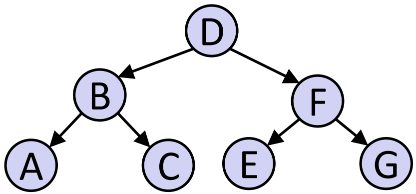
- Traversals
  - Introduction
  - **Trees and Graphs: Level-order / Breadth-first**
  - Trees: Three Flavors of Depth-first
  - Graphs: Depth-first
  - Conclusion

- Shortest Paths!

- Dijkstra's Algorithm

Trees: Level-Order / BFS

- Process top-to-bottom, left-to-right
  - Goes "broad" instead of "deep"
  - Requires a queue to track need-to-explore vertices, which is sometimes called the ***fringe***



- Resembles how we converted our binary heap (ie, a complete tree) to its array representation

```
levelOrderTraverse(Vertex root) {
  q.enqueue(root)
  while (!q.empty())
    next = q.dequeue()
    process(next)
    foreach u in next.children
      q.enqueue(u)
}
```

1. Initialize aux data structure
2. Have vertices in data struct?
   3. Get vertex from data struct
   4. Visit/process vertex
   5. Update vertex's neighbors

Graphs: Breadth-First

- When working with graphs, we refer to level-order traversals as breadth-first traversals
  - We also need to verify if a vertex has been visited – why?

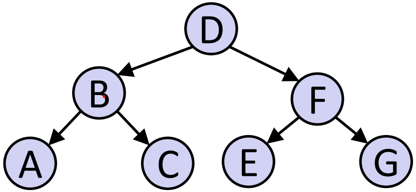
```
breadthFirstTraversal(Vertex start) {
  q.enqueue(start)
  mark start as visited

  while (!q.empty())
    next = q.dequeue()
    process(next)
    foreach u in next.neighbors
      if (!u.marked)
        mark u
        q.enqueue(u)
}
```

1. Initialize aux data structure
2. Have vertices in data struct?
   3. Get vertex from data struct
   4. Visit/process vertex
   5. Update vertex's neighbors

Lecture Outline

- Topological Sort (cont.)
- Traversals
  - Introduction
  - Trees and Graphs: Level-order / Breadth-first
  - **Trees: Three Flavors of Depth-first**
  - Graphs: Depth-first
  - Conclusion
- Shortest Paths!
- Dijkstra's Algorithm

Trees: Depth-First Traversal



- Process deep vertices before shallow ones
  - Eg, visit A before F
  - Succinct implementation if using recursion; otherwise, requires a stack to track need-to-explore vertices

```
traverseIter(Node start) {
  s.push(start)
  while (!s.empty())
    next = s.pop()
    process(next)
    foreach u in next.neighbors
      q.push(u)
}
```

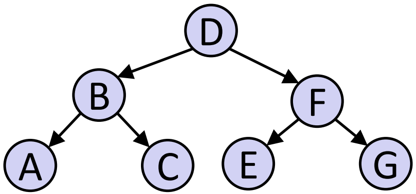
1. Initialize aux data structure
2. Have vertices in data struct?
   3. Get vertex from data struct
   4. Visit/process vertex
   5. Update vertex's neighbors

```
traverseRecur(Node x) {
  if (x == null)
    return;
  process(x.key)
  foreach c in x.children
    traverseRecur(c)
}
```

Trees: Depth-First: Pre-Order

```
preOrder(Node x) {
  if (x == null)
    return;
  process(x.key)
  preOrder(x.left)
  preOrder(x.right)
}
```

- Pre-order "visits" the node before traversing its children
  - DBACFEG

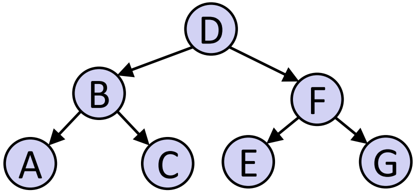


Trees: Depth-First: In-Order

```
preOrder(Node x) {
  if (x == null)
    return;
  process(x.key)
  preOrder(x.left)
  preOrder(x.right)
}
```

```
inOrder(Node x) {
  if (x == null)
    return;
  inOrder(x.left)
  process(x.key)
  inOrder(x.right)
}
```

- Pre-order "visits" the node before traversing its children
  - DBACFEG
- In-order traverses the left child, "visits" the node, then traverses the right child
  - ABCDEF



Trees: Depth-First: Post-Order

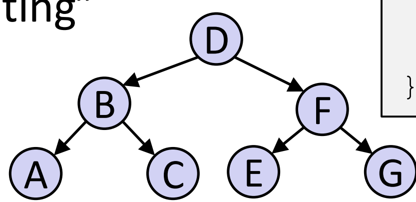
note the root's position!

- Pre-order "visits" the node before traversing its children
  - DBACFEG
- In-order traverses the left child, "visits" the node, then traverses the right child
  - ABCDEF
- Post-order traverses its children before "visiting" the node
  - ACBEGFD

```
preOrder(Node x) {
  if (x == null)
    return;
  process(x.key)
  preOrder(x.left)
  preOrder(x.right)
}
```

```
inOrder(Node x) {
  if (x == null)
    return;
  inOrder(x.left)
  process(x.key)
  inOrder(x.right)
}
```

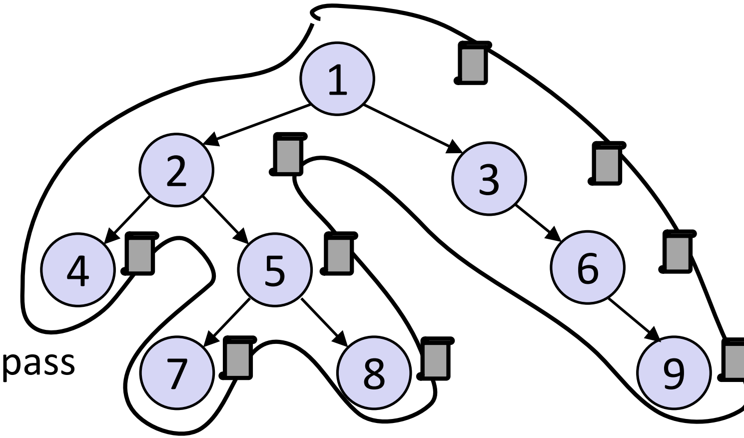
```
postOrder(Node x) {
  if (x == null)
    return;
  postOrder(x.left)
  postOrder(x.right)
  process(x.key)
}
```



Useful Trick for Depth-First Tree Traversals

- *(Useful for humans, not algorithms)*

- Trace a path around the graph, from the top going counter-clockwise
  - *Pre-order*: Process when you pass LEFT side of a node
  - *In-order*: Process when you pass BOTTOM of a node
  - Post-order: Process when you pass the RIGHT side of a node.



Post-order: 4 7 8 5 2 9 6 3 1

Lecture Outline

- Topological Sort (cont.)

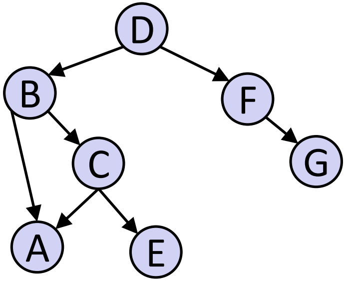
- Traversals
  - Introduction
  - Trees and Graphs: Level-order / Breadth-first
  - Trees: Three Flavors of Depth-first
  - **Graphs: Depth-first**
  - Conclusion

- Shortest Paths!

- Dijkstra's Algorithm

Trees and Graphs: Depth-First

- Still processing "far vertices" before "near" ones
  - Still has recursive and iterative implementations
  - Still must mark previously-visited nodes




```
depthFirstTraversal(Vertex start) {
  s.push(start)
  mark start as visited

  while (!s.empty())
    next = s.pop()
    process(next)
    foreach u in next.neighbors
      if (!u.marked)
        mark u
        s.push(u)
}
```

1. Initialize aux data structure
2. Have vertices in data struct?
   3. Get vertex from data struct
   4. Visit/process vertex
   5. Update vertex's neighbors

Lecture Outline

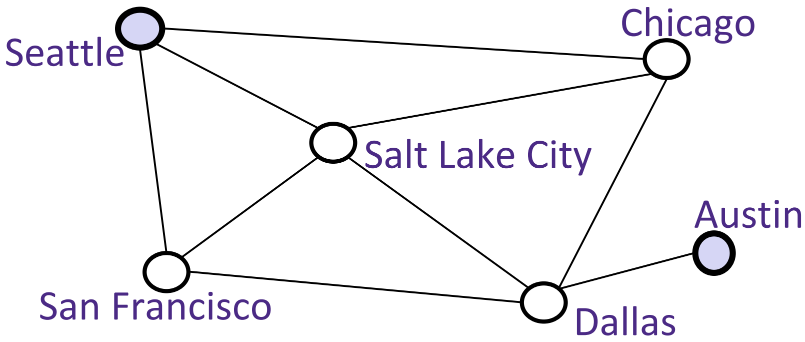
- Topological Sort (cont.)
- Traversals
  - Introduction
  - Trees and Graphs: Level-order / Breadth-first
  - Trees: Three Flavors of Depth-first
  - Graphs: Depth-first
  - **Conclusion**
- Shortest Paths!
- Dijkstra's Algorithm

Saving the Path

- These graph traversals can answer the "reachability question":
  - "*Is there* a path from vertex x to vertex y?"
- But what if we want to *output the actual path* or its length?
  - Eg, getting driving directions vs knowing it's possible to get there
- Modifications:
  - Instead of just "marking" a vertex, store the path's *previous vertex*
    - ie: when processing `u`, set `v.prev` to `u`
  - When you reach the goal, follow `prev` fields backwards to start
    - (don't forget to reverse the answer)
  - Path length:
    - Same idea, but also store integer distance at each vertex

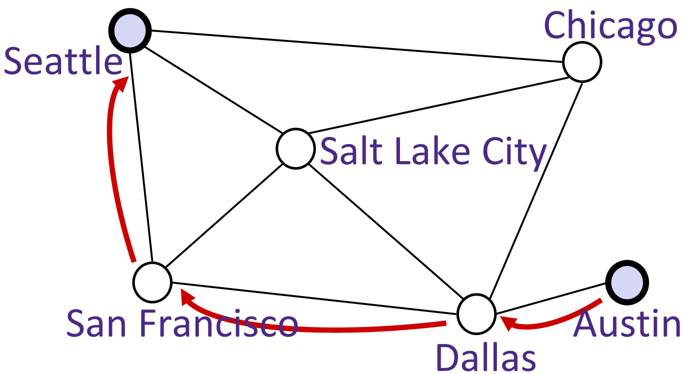
Saving the Path: Example using BFS (1 of 2)

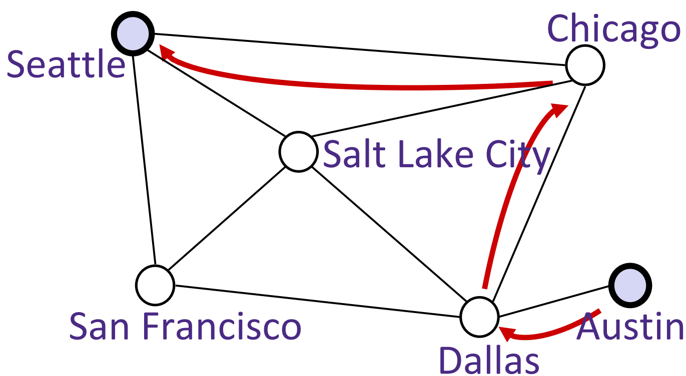
- Find the shortest path from Seattle to Austin
  - Remember marked vertices are not re-enqueued
  - Shortest paths may not be unique



Saving the Path: Example using BFS (2 of 2)

- Find the shortest path from Seattle to Austin
  - Remember marked vertices are not re-enqueued
  - Shortest paths may not be unique





DFS/BFS Comparison

- Breadth-first search:
  - Always finds shortest paths, i.e., finds "optimal solutions"
    - Better for "what is the shortest path from x to y?"
  - But queue may hold up to O(|V|) vertices
    - Eg, at the bottom level of perfect binary tree, queue contains |V|/2 vertices

- Depth-first search:
  - Can use less space when finding a path
    - If longest path in the graph is p and highest out-degree is d then stack never has more than d*p elements

It Doesn't Have to be Either/Or

- A third approach: Iterative deepening (IDDFS):
  - Try DFS, but don't allow recursion more than K levels deep
  - If fails to find a solution, increment K and start the entire search over

- Like BFS, finds shortest paths. Like DFS, less space

Lecture Outline

- Topological Sort
- Traversals
  - Introduction
  - Trees and Graphs: Level-order / Breadth-first
  - Trees: Three Flavors of Depth-first
  - Graphs: Depth-first
  - Conclusion
- **Shortest Paths!**
- Dijkstra's Algorithm

Single-Source Shortest Paths

- We've seen BFS finds the minimum path length from **v** to **u**
  - Runtime: $O(|E|+|V|)$

- Actually, BFS finds the min path length from **v** to *every vertex*
  - Still $O(|E|+|V|)$
  - Worst-case runtime for single-destination is no faster than worst-case runtime for all-destinations

Shortest Path: Applications

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

Wait, these are all weighted graphs!

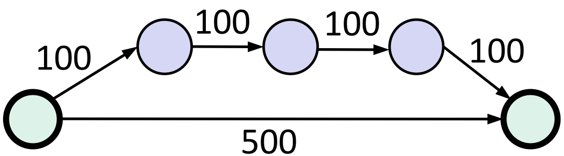
Single-Source Shortest Paths … *for Weighted Graphs*

Given a weighted graph and vertex **v**,
find the minimum-cost path from **v** to every vertex

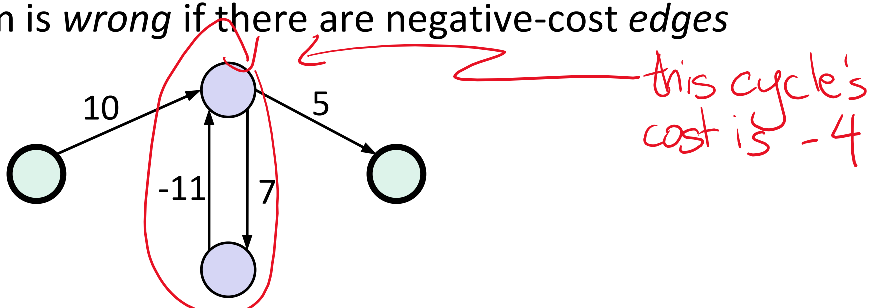
- As before:
  - All-destinations is asymptotically no harder than single-destination
- Unlike before:
  - BFS will not work

BFS for Weighted Graphs

- BFS doesn't work! Shortest path may not have fewest edges
  - Eg: cost of flight. May be cheaper to fly through a hub than fly direct



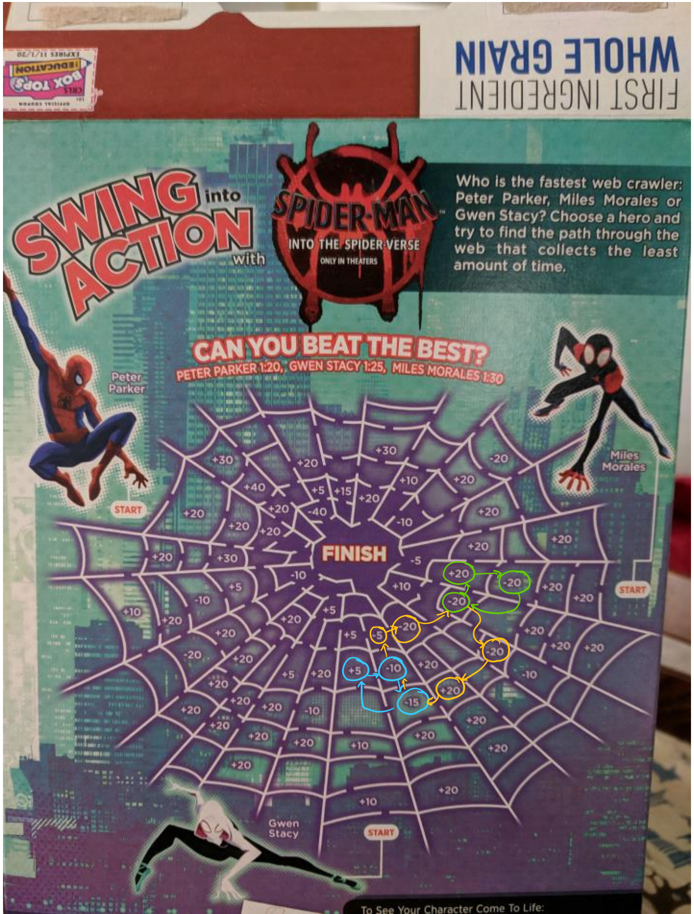
- We will assume there are *no negative edge weights*
  - Entire problem is *ill-defined* if there are negative-cost *cycles*
  - Today's algorithm is *wrong* if there are negative-cost *edges*



Negative Cycles vs Negative Edges

- *Negative cycles*: no algorithm can find a finite optimal path
  - You can always decrease the distance by going through the negative cycle a few more times

- *Negative edges*: Dijkstra's can't guarantee correctness
  - But other algorithms might



Lecture Outline

- Topological Sort
- Traversals
  - Introduction
  - Trees and Graphs: Level-order / Breadth-first
  - Trees: Three Flavors of Depth-first
  - Graphs: Depth-first
  - Conclusion
- Shortest Paths!
- **Dijkstra's Algorithm**

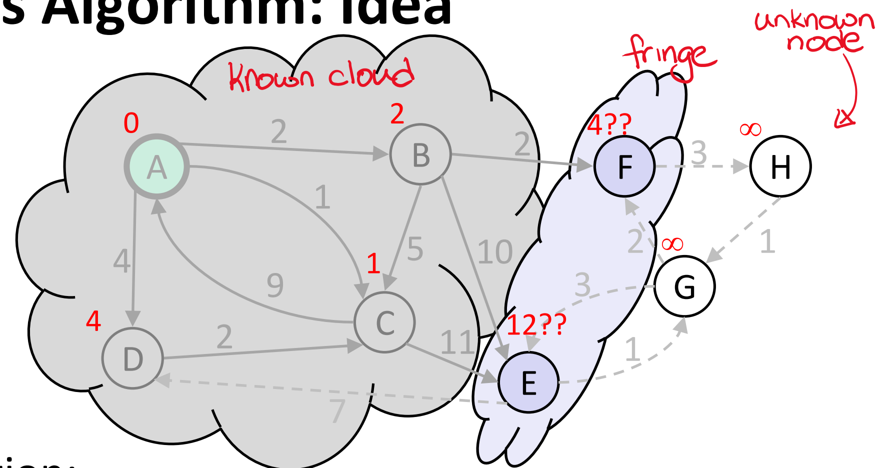
Dijkstra's Algorithm

ijk

- Named after its inventor, Edsger Dijkstra (1930-2002)
  - Truly one of the "founders" of computer science
  - 1972 Turing Award
  - This algorithm is just *one* of his many contributions!
  - Example quote: "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 vertices whose shortest distance has been computed
  - Vertices not in the set will have a "best distance so far"

Dijkstra's Algorithm: Idea



- Initialization:
  - Start vertex has distance **0**; all other vertices have distance ∞
- At each step:
  - Pick closest unknown vertex v
  - Add it to the "cloud" of known vertices
  - Update distances for vertices with edges from v

1. Initialize aux data structure
2. Have vertices in data struct?
   3. Get vertex from data struct
   4. Visit/process vertex
   5. Update vertex's neighbors

Dijkstra's Algorithm: Pseudocode

```
dijkstra(Graph g, Vertex start) {
  foreach vertex v in g:
    v.distance = ∞
    v.known = false
  start.distance = 0

  while there are vertices in g that are not known:
    select vertex v with lowest cost
    v.known = true
    foreach unknown v.neighbor with weight w:
      d1 = v.distance + w  // best path through v to u
      d2 = u.distance      // previous best path to u
      if (d1 < d2):        // if this is a better path to u
        u.distance = d1
        u.previous = v     // backtracking info to
                           // recreate path
}
```

Dijkstra's Algorithm: Important Features

- Once a vertex is marked known, its shortest path is known
  - Can reconstruct path by following back-pointers ("previous" fields)

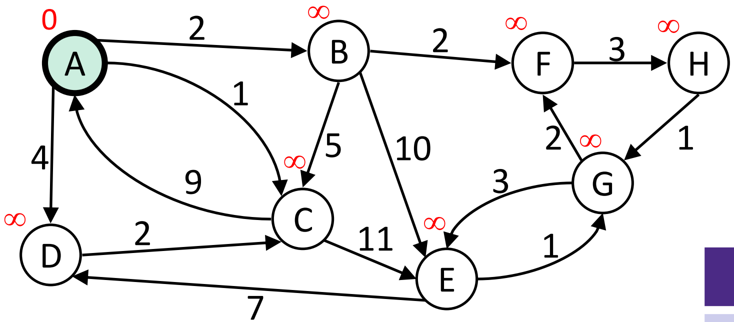
- While a vertex is not known, another shorter path might be found

Dijkstra's Algorithm vs BFS

```
dijkstra(Graph g, Vertex start) {

  foreach vertex v in g:
    v.distance = ∞
    v.known = false
  start.distance = 0

  while there are unknown vertices:
    v = lowest cost unknown vertex
    v.known = true
    foreach unknown v.neighbor
    with weight w:
      d1 = v.distance + w
      d2 = u.distance
      if (d1 < d2):
        u.distance = d1
        u.previous = v


}
```

```
breadthFirst(Graph g,
             Vertex start) {
  q.enqueue(start)

  mark start as visited

  while (!q.empty())
    next = q.dequeue()
    process(next)
    foreach u in next.neighbors

      if (!u.marked)
        mark u




        q.enqueue(u)
}
```

Dijkstra's Algorithm: Example #1

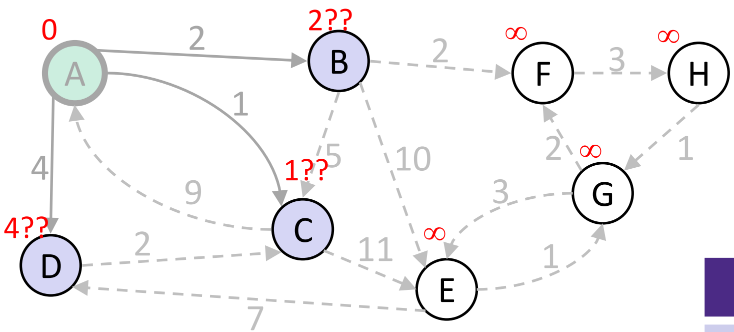


Best distance from A, so far

Vertex	Known?	Distance	Previous
A		∞	
B		∞	
C		∞	
D		∞	
E		∞	
F		∞	
G		∞	
H		∞	

Order Added to Known Set:

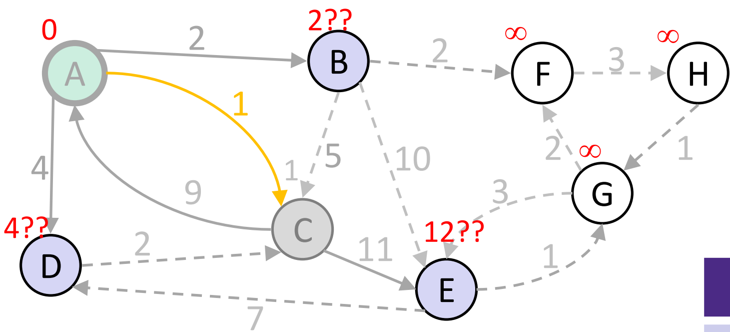
Dijkstra's Algorithm: Example #1



Vertex	Known?	Distance	Previous
A	Y	0	/
B		**≤ 2**	**A**
C		**≤ 1**	**A**
D		**≤ 4**	**A**
E		∞	
F		∞	
G		∞	
H		∞	

Order Added to Known Set:
A

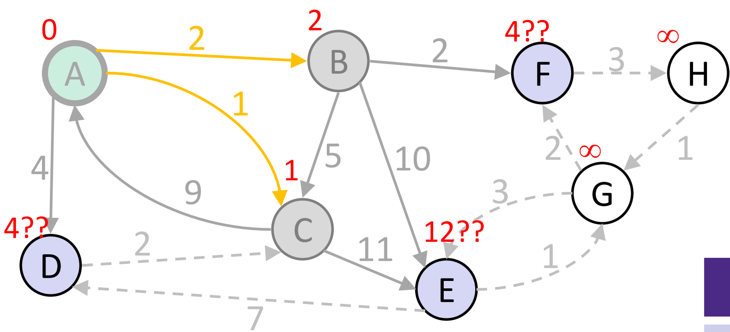
Dijkstra's Algorithm: Example #1



Vertex	Known?	Distance	Previous
A	Y	0	/
B		≤ 2	A
C	Y	1	A
D		≤ 4	A
E		≤ 12	C
F		∞	
G		∞	
H		∞	

Order Added to Known Set:
A, C

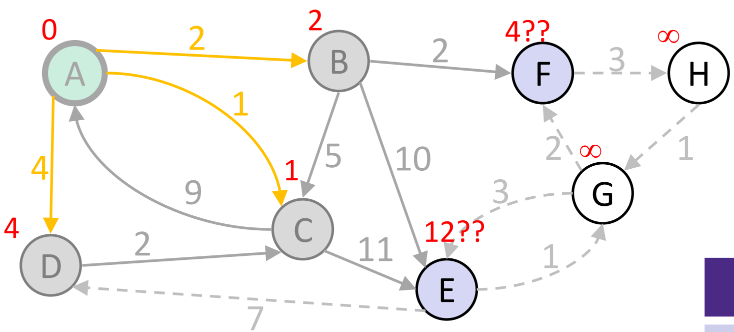
Dijkstra's Algorithm: Example #1



Order Added to Known Set:
A, C, B

Vertex	Known?	Distance	Previous
A	Y	0	/
B	Y	2	A
C	Y	1	A
D		≤ 4	A
E		≤ 12	C
F		**≤ 4**	**B**
G		∞	
H		∞	

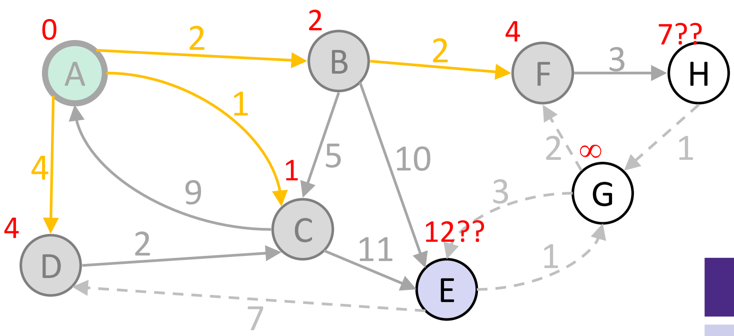
Dijkstra's Algorithm: Example #1



Order Added to Known Set:
A, C, B, D

Vertex	Known?	Distance	Previous
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F		≤ 4	B
G		∞	
H		∞	

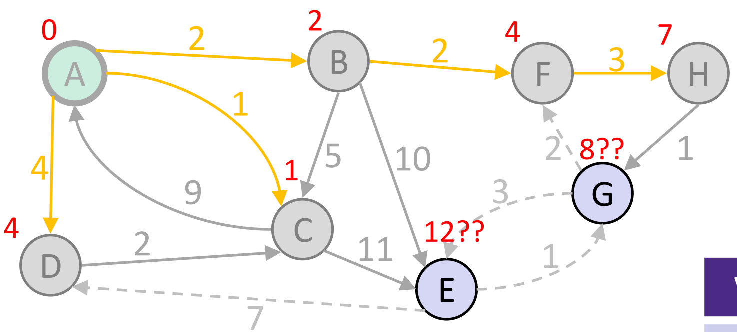
Dijkstra's Algorithm: Example #1



Order Added to Known Set:
A, C, B, D, F

Vertex	Known?	Distance	Previous
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F	Y	4	B
G		∞	
H		**≤ 7**	**F**

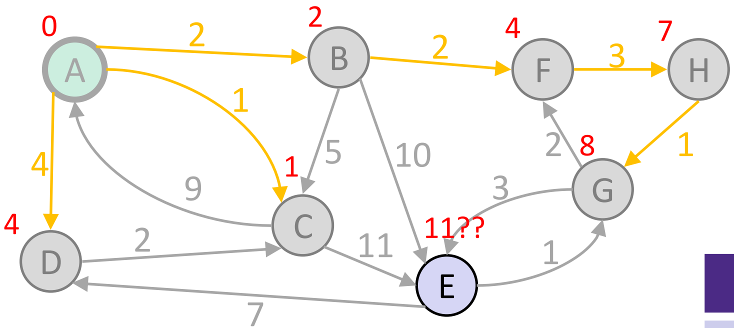
Dijkstra's Algorithm: Example #1



Order Added to Known Set:
A, C, B, D, F, H

Vertex	Known?	Distance	Previous
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		≤ 12	C
F	Y	4	B
G		**≤ 8**	**H**
H	Y	7	F

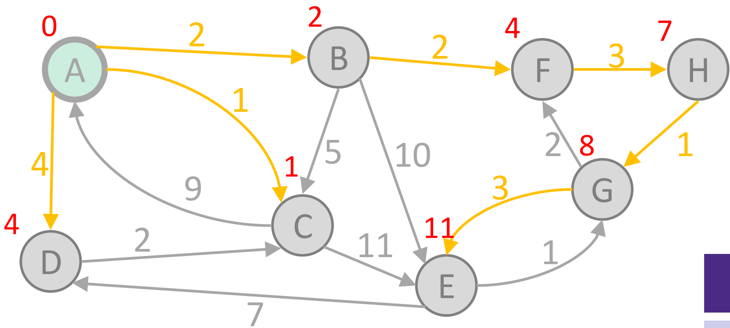
Dijkstra's Algorithm: Example #1



Order Added to Known Set:
A, C, B, D, F, H, G

Vertex	Known?	Distance	Previous
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E		**≤ 11**	**G**
F	Y	4	B
G	Y	8	H
H	Y	7	F

Dijkstra's Algorithm: Example #1

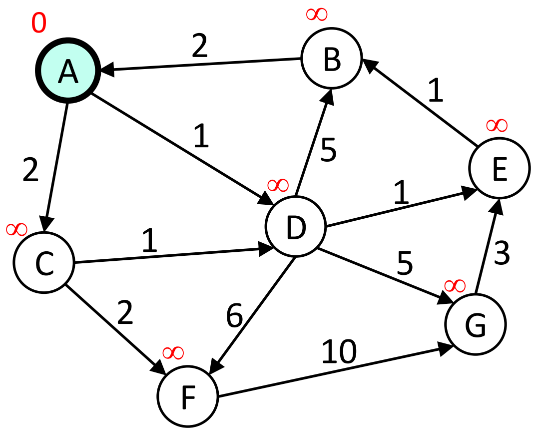


TADA!!!

<u>Order Added to Known Set:</u>
A, C, B, D, F, H, G, E

Vertex	Known?	Distance	Previous
A	Y	0	/
B	Y	2	A
C	Y	1	A
D	Y	4	A
E	Y	11	G
F	Y	4	B
G	Y	8	H
H	Y	7	F

Dijkstra's Algorithm: Example #2



Order Added to Known Set:

Vertex	Known?	Distance	Previous
A		∞	
B		∞	
C		∞	
D		∞	
E		∞	
F		∞	
G		∞	