CSE 373: Data Structures and Algorithms

Lecture 16: Dijkstra's Algorithm (Graphs)

Instructor: Lilian de Greef
Quarter: Summer 2017

Today

- Announcements
- Graph Traversals Continued
  - Remarks on DFS & BFS
  - Shortest paths for weighted graphs:
    Dijkstra's Algorithm!

Announcements:

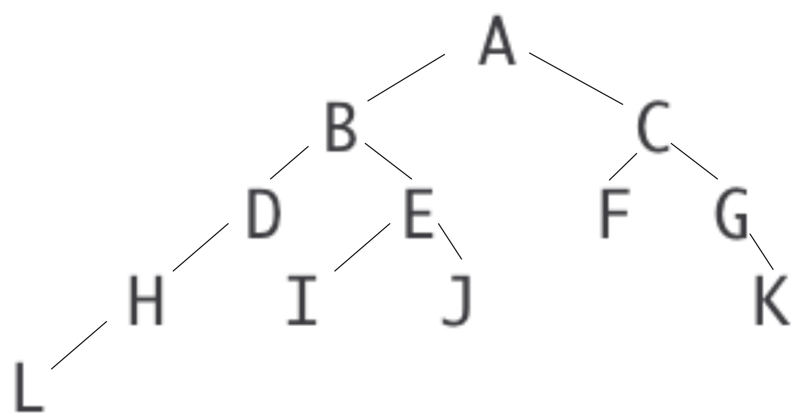
Homework 4 is out!

- Due next Friday (August 4th) at 5:00pm
- May choose to pair-program if you like!
  - Same cautions as last time apply: choose partners and when to start working wisely!
- Can almost entirely complete using material by end of this lecture
- Will discuss some software-design concepts next week to help you prevent some (potentially non-obvious) bugs

Another midterm correction… (😨 & 😫)

1. **True or False:** (6 points)
   Circle whether the statement is either true or false.

   f. **(true / false):** In an AVL tree, the longest and shortest paths (i.e. number of edges) from the root to a leaf do not differ by more than one.



Bring your midterm to *any* office hours to get your point back.

I will have the final exam *quadruple-checked* to avoid these situations!
(I am so sorry)

Graphs: Traversals Continued

And introducing Dijkstra's Algorithm for shortest paths!

Graph Traversals: Recap & Running Time

Marking a vertex adds it to the "set of visited vertices"

- Traversals: General Idea
  - Starting from one vertex, repeatedly explore adjacent vertices
  - Mark each vertex we visit, so we don't process each more than once (cycles!)

- Important Graph Traversal Algorithms:

	Depth First Search (DFS)	**Breadth First Search (BFS)**
Explore…	as far as possible before backtracking	all neighbors first before next level of neighbors
Choose next vertex using…	recursion or a stack	a queue

- Assuming "choose next vertex" is $O(1)$, entire traversal is
  - Use graph represented with adjacency

Comparison (useful for Design Decisions!)

- Which one finds **shortest** paths?
  - i.e. which is better for "what is the shortest path from **x** to **y**" when there's more than one possible path?

- Which one can use less space in finding a path?

- A third approach:
  - *Iterative deepening (IDFS)*:
    - Try DFS but disallow recursion more than **`K`** levels deep
    - If that fails, increment **`K`** and start the entire search over
  - Like BFS, finds shortest paths. Like DFS, less space.

Graph Traversal Uses

In addition to finding paths, we can use graph traversals to answer:

- What are all the vertices *reachable* from a starting vertex?
- Is an undirected graph connected?
- Is a directed graph strongly connected?

- But what if we want to actually output the path?

- How to do it:
  - Instead of just "marking" a node, store the previous node along the path
  - When you reach the goal, follow `path` fields back to where you started (and then reverse the answer)
  - If just wanted path *length*, could put the integer distance at each node instead once

Single source shortest paths

- Done: BFS to find the minimum path length from **v** to **u** in $O(|E|+|V|)$
- Actually, can find the minimum path length from **v** to *every node*
  - Still $O(|E|+|V|)$
  - No faster way for a "distinguished" destination in the worst-case
- Now: Weighted graphs

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

- As before, asymptotically no harder than for one destination

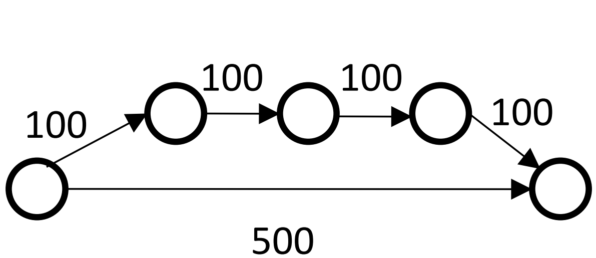
A Few Applications of Shortest Weighted Path

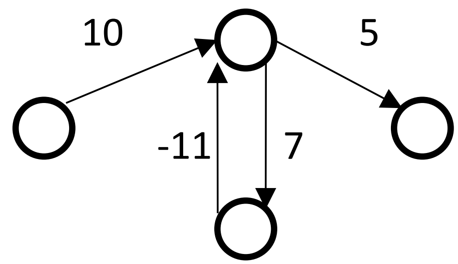
- Driving directions
- Cheap flight itineraries
- Network routing
- Critical paths in project management

Not as easy as BFS

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

- Annoying when this happens with costs of flights



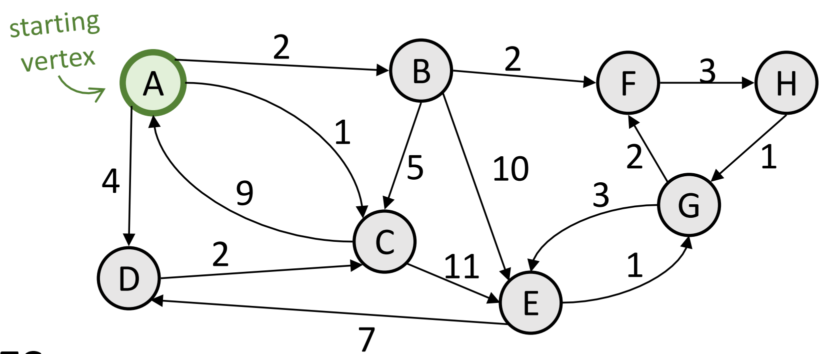


We will assume there are no negative weights

- *Problem* is *ill-defined* if there are negative-cost *cycles*
- *Today's algorithm* is *wrong* if *edges* can be negative
  - There are other, slower (but not terrible) algorithms

Algorithm: General Idea

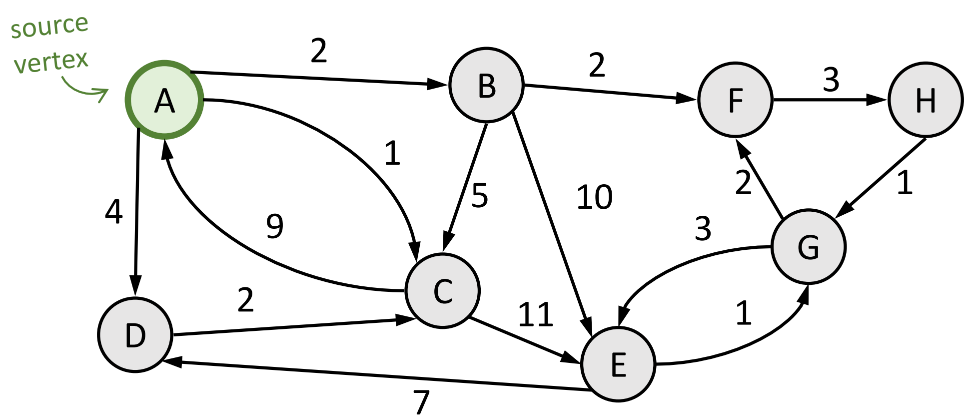
Goal: From one starting vertex, what are the shortest paths to each of the other vertices (for a weighted graph)?



Idea: Similar to BFS

- Repeatedly increase a “set of vertices with known shortest distances”
- Any vertex not in this set will have a “best distance so far”
- Each vertex has a “cost” to represent these shortest/best distances
- Update costs (i.e. “best distances so far”) as we add vertices to set

Shortest Path Example #1



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

Known Set (in order added):

(extra space in case you want/need it)

This is called… Dijkstra's Algorithm

Named after its inventor Edsger Dijkstra (1930-2002)

Truly one of the "founders" of computer science;
this is just one of his many contributions

"Computer science is no more about computers than astronomy is about telescopes."

- Edsger Dijkstra

Dijkstra's Algorithm (Pseudocode)

Dijkstra's Algorithm – the following algorithm for finding single-source shortest paths in a weighted graph (directed or undirected) with no negative-weight edges:

1. For each node `v`, set `v.cost = ∞ and v.known = false`
2. Set `source.cost = 0`
3. While there are unknown nodes in the graph
   a) Select the unknown node `v` with lowest cost
   b) Mark `v` as known
   c) For each edge `(v,u)` with weight `w`,

```
c1 = v.cost + w   // cost of best path through v to u
c2 = u.cost       // cost of best path to u previously known
if(c1 < c2){      // if the path through v is better
  u.cost = c1
  u.path = v      // for computing actual paths
}
```

Dijkstra’s Algorithm: Features

- When a vertex is marked known, the cost of the shortest path to that node is known
  - The path is also known by following back-pointers

- While a vertex is still not known, another shorter path to it *might* still be found

Note: The “Order Added to Known Set” is not important

- A detail about how the algorithm works (client doesn’t care)
- Not used by the algorithm (implementation doesn’t care)
- It is sorted by path-cost, resolving ties in some way
  - Helps give intuition of why the algorithm works

Dijkstra's Algorithm: Commentary

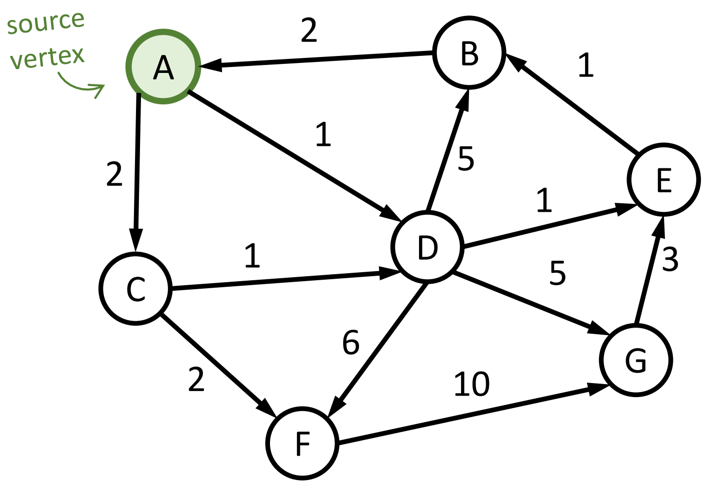
Dijkstra's Algorithm is one example of...

- A **greedy algorithm**:
  - Make a locally optimal choice at each stage to (hopefully) find a global optimum
  - i.e. Settle on the best looking option at each repeated step
  - Note: for some problems, greedy algorithms cannot find best answer!

vertex	known?	cost	path
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

- **Dynamic programming**:
  - Solve a complex problem by breaking it down into a collection of simpler subproblems, solve each of those subproblems just once, and store their solutions.
  - i.e. Save partial solutions, and use it to solve further parts to avoid repeating work

Dijkstra's Algorithm: Practice Time!



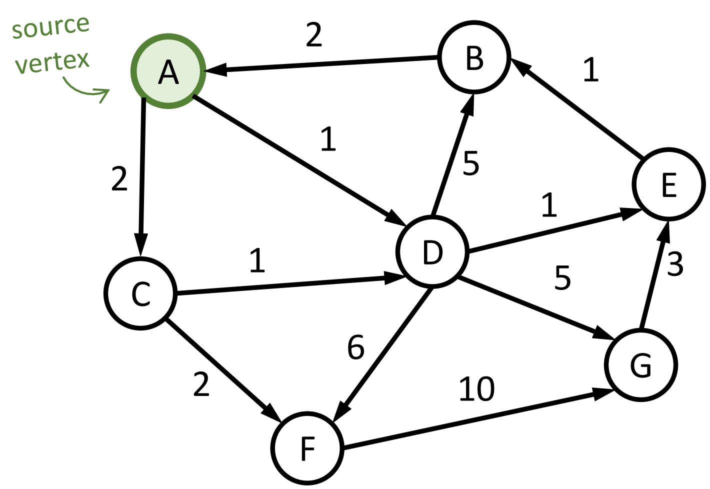
An order of adding vertices to the known set:

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

vertex	known?	cost	path
A			
B			
C			
D			
E			
F			
G			

(space for scratch work)

Dijkstra's Algorithm: Practice Time!

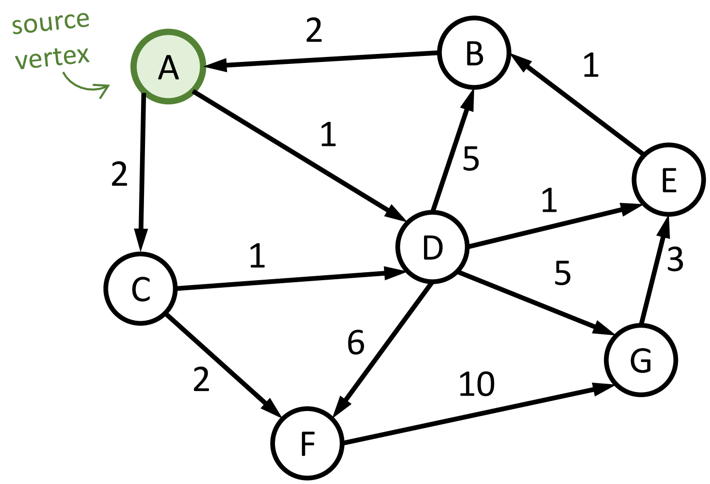


An order of adding vertices to the known set:

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

vertex	known?	cost	path
A			
B			
C			
D			
E			
F			
G			

Dijkstra’s Algorithm: Practice Time!



vertex	known?	cost	path
A	Y	0	
B		≤ 6	D
C		≤ 2	A
D	Y	1	A
E		≤ 2	D
F		≤ 7	D
G		≤ 6	D

An order of adding vertices to the known set:

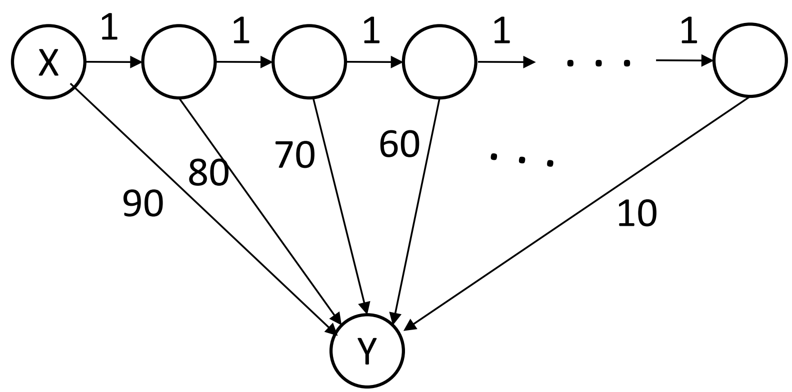
A) A, D, C, E, F, B, G

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

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

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

Example #3



- How will the "best-cost-so-far" for Y proceed?

- Is this expensive?

because each *edge* is processed

Where are We?

- Had a problem: Compute shortest paths in a weighted graph with no negative weights
- Learned an algorithm: Dijkstra's algorithm
- 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: Intuition

Rough intuition:

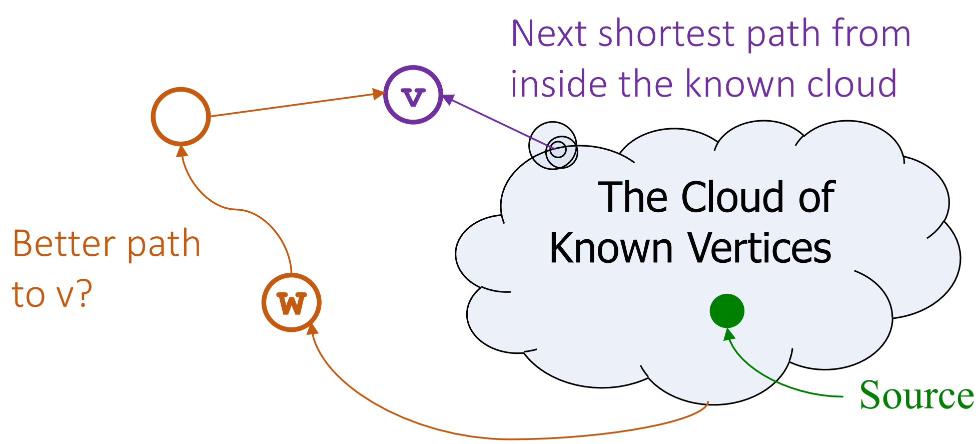
All the “known” vertices have the correct shortest path

- True initially: shortest path to start node has cost 0
- If it stays true every time we mark a node “known”, then by induction this holds and eventually everything is “known”

Key fact we need: When we mark a vertex “known” we won’t discover a shorter path later!

- This holds only because Dijkstra’s algorithm picks the node with the next shortest path-so-far
- The proof is by contradiction…

Correctness: The Cloud (Rough Sketch)



- Suppose **v** is the next node to be marked known (next to add to "the cloud of known vertices")
- The best-known path to **v** must have only nodes "in the cloud"
  - Else we would have picked a node closer to the cloud than v
- Suppose 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!

Efficiency, first approach

Use pseudocode to determine asymptotic run-time

- Notice each edge is processed only once

```
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.path = b
       }
}
```

Improving asymptotic running time

- So far: $O(|V|^2)$
- We had a similar "problem" with topological sort being $O(|V|^2)$ 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?
  - A holding all unknown nodes,
  - But must support operation
    - Must maintain a reference from each node to its current position in the priority queue
    - Conceptually simple, but can be a pain to code up

Efficiency, second approach

Use pseudocode to determine asymptotic run-time

```
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.path = b
      }
}
```

Dense vs. Sparse (again!)

- First approach: $O(|V|^2)$

- Second approach: $O(|V|\log|V|+|E|\log|V|)$

- So which is better?
  - Dense or Sparse? $O(|V|\log|V|+|E|\log|V|)$ (if $|E| > |V|$, then it's $O(|E|\log|V|)$)
  - Dense or Sparse? $O(|V|^2)$

- 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 $|E|$

Practice with Design Decisions

Graphs Edition!

Our three-eye-alien friend uncovered an impressively complete and up-to-date family tree tracing all the way back to the ancient emperor Qin Shi Huang. The alien wants to find a descendant of this emperor who's still alive, and could use your advice!

(According to Wikipedia, Qin Shi Huang had ~50 children, wow!)

What data structure would you recommend?
Why?

What algorithm would you recommend?
Why?

(extra space for notes)