Max Bandwidth Path and Depth-First Search

CSE 101: Design and Analysis of Algorithms

Lecture 2

CSE 101: Design and analysis of algorithms

- Max bandwidth path and depth-first search
  – Reading: Sections 3.1 and 3.2
- Homework 1 will be assigned today
  – Due Oct 9, 11:59 PM

How to approach problems

- When can we use an algorithm developed for one problem to solve another?
- Modifying algorithms vs. using algorithms in reductions
- Defining problems precisely

Example

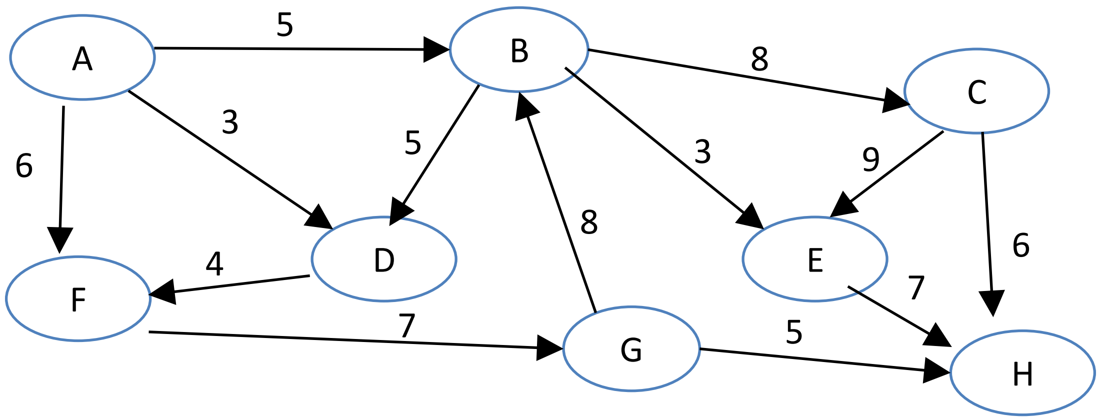
- We'll start with a familiar algorithm (graph search) and try to re-use it for a new problem (max bandwidth path)

Defining problems precisely

- Instance: What is the input?
- Solution type: What form is your output (path, quantity, boolean, etc.)?
- Restrictions: What solution types are allowed?
- Objective: How do you compare which solutions are better than others?

Max bandwidth path

- Graph represents network, with edges representing communication links. Edge weights are bandwidth of link.



What is the largest bandwidth of a path from A to H?

Path

- Definition: A path is a sequence of vertices and edges
$$v_1, e_1, v_2, e_2 \dots v_{n-1}, e_{n-1}, v_n$$
such that $e_i = (v_i, v_{i+1})$
- The length of a path is the number of edges
- When we say **path** in this class, we are talking about **simple paths**, which means no two edges are the same
- Note that a single vertex v_1 is a trivial path from the vertex to itself

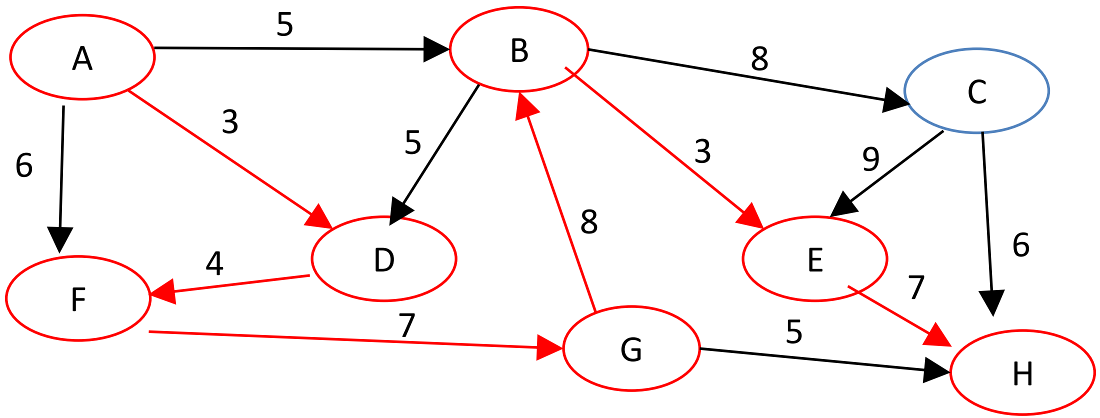
Problem statement

- Instance: Directed graph G= (V, E) with positive edge weights w(e), two vertices s, t $\in V$
- Solution type: A path p in G
- Restriction: The path must go from s to t
- Bandwidth of a path

$$\mathrm{BW}(p) = \min_{e \in p} w(e)$$

- Objective: Over all possible paths p between s and t, find the maximum $\mathrm{BW}(p)$

Bandwidth of a path



What is the bandwidth of the path in red from A to H?

Re-using algorithms, modification

- Modification: Take an algorithm for a related problem, and change some of the details to match the new problem
- Complications: Does it actually solve the new problem? Is it as fast as the original algorithm?
- To answer these questions, we need to go back to the proof of correctness and time analysis, to see if the analogous statements are still true

Re-using algorithms, reduction

- Reduction: Use an algorithm for a related problem without changes, as a sub-routine for the new problem
- Complications: Does it actually solve the new problem? How much faster is it compared to the original algorithm?
- Correctness: Show the solution for the created instance of the related problem gives the solution for the actual instance of the new problem
- Time analysis: Calculate the relevant size parameters of the created instance, in terms of the size of the actual instance. Plug that into the time analysis for the original algorithm.

Related problems

- What are some problems that we have already seen in other classes that seem related to the max bandwidth path problem?
  - Graph search
- Can we think of ways to use algorithms for these problems to solve max bandwidth path?

Graph reachability

- Given a directed graph G and a start vertex s, produce a list of all vertices v reachable from s by a directed path in G
- At each point in a graph search algorithm, the vertices are partitioned into
  - X: explored
  - F: frontier (reached but have not yet explored)
  - U: unreached

Graph reachability

```
procedure GraphSearch (G: directed graph, s: vertex)

 Initialize X = empty, F = {s}, U = V – F.
 While F is not empty:
   Pick v in F.
   For each neighbor u of v:
       If u is not in X or F:
           move u from U to F.
   Move v from F to X.

 Return X.
```

X: explored
F: frontier (reached but have not yet explored)
U: unreached

Graph reachability

- Data structures
  - X
  - F
  - U
  - G

What are required capabilities of each?

procedure GraphSearch (G: directed graph, s: vertex)

Initialize X = empty, F = {s}, U = V – F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
    If u is not in X or F:
      move u from U to F.
  Move v from F to X.
Return X.

Graph reachability

- Data structures and required capabilities
  - X is a set
    - Test membership
    - Insert
  - F is a set
    - Find and delete
    - Test membership
    - Insert
  - U is a set
    - Test membership
    - Delete
  - G is a graph
    - For each vertex, loop through its neighbors

Graph reachability

- Data structures and required capabilities
  - X is a set: array of booleans indexed by vertex
    - Test membership: O(1)
    - Insert: O(1)
  - F is a set: stack, or queue and array of booleans
    - Find and delete: pop, dequeue, flip boolean O(1)
    - Test membership: O(1)
    - Insert: push, enqueue, flip boolean O(1)
  - U is a set: array of booleans
    - Test membership O(1)
    - Delete O(1)
  - G is a graph: adjacency list
    - For each vertex, loop through its neighbors O(deg(v)+1)

Graph reachability, time analysis

procedure GraphSearch (G: directed graph, s: vertex)

$\boxed{|V| = n}$

Initialize X = empty, F = {s}, U = V – F. O(n) time to initialize arrays
While F is not empty:
Pick v in F. O(1) time to pop or dequeue
For each neighbor u of v: O(deg(v)) time to list
If u is not in X or F: O(1) time to check and change array value, push/enqueue
move u from U to F. O(1)
Move v from F to X. O(1) time to change array value

Return X.
Total time: $O(\sum \deg(v))$ for all v chosen from F

$\boxed{O(\sum \deg(v)) = O(|E|)}$

Graph reachability, time analysis

- Each v is added to F at most once
- Therefore, each v is deleted from F at most once
- Therefore, $O(\sum \deg(v))$ for all v chosen from F $\leq O(\sum \deg(v))$ for all v in G $= O(m)$
- So, total time is $O(n + m)$

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

Graph reachability, correctness

- Proof of correctness:
  - We must show that at the end of the algorithm:
    - *A:* if $v \in X$ then there is a path from s to v
    - *B:* if $v \notin X$ then there is not a path from s to v

procedure GraphSearch (G: directed graph, s: vertex)

Initialize X = empty, F = {s}, U = V – F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
    If u is not in X or F:
      move u from U to F.
  Move v from F to X.
Return X.

Correctness, *A*

- A: if $\mathrm{v} \in X$ then there is a path from s to v
- Proof of correctness: (loop invariant)
  - After the $t\text{-}th$ iteration of the while loop, every element of X or F is reachable from s in G
- Base case: before going through the loop, X is empty and F is {s}

procedure GraphSearch (G: directed graph, s: vertex)

Initialize X = empty, F = {s}, U = V – F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
    If u is not in X or F:
      move u from U to F.
  Move v from F to X.
Return X.

Correctness, *A*

- *A*: if $v \in X$ then there is a path from s to v
- Proof of correctness: (loop invariant)
  - After the $t\text{-}th$ iteration of the while loop, every element of X or F is reachable from s in G
- Base case: before going through the while loop, X is empty and F is {s}
- Suppose the loop invariant is true after t iterations. What happens in the next iteration?

procedure GraphSearch (G: directed graph, s: vertex)

Initialize X = empty, F = {s}, U = V – F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
    If u is not in X or F:
      move u from U to F.
  Move v from F to X.
Return X.

Correctness, *A*

- A: if $\mathrm{v} \in X$ then there is a path from s to v
- You pick a vertex v in F. (Which vertex depends on the data structure. For the sake of this proof, we can pick any of the vertices in F next.)
- We move all neighbors of v into F if they are in U

procedure GraphSearch (G: directed graph, s: vertex)

Initialize X = empty, F = {s}, U = V – F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
    If u is not in X or F:
      move u from U to F.
  Move v from F to X.
Return X.

Correctness, *A*

- *A*: if $v \in X$ then there is a path from s to v
- You pick a vertex v in F. (Which vertex depends on the data structure. For the sake of this proof, we can pick any of the vertices in F next.)
- We move all neighbors of v into F if they are in U
  - If there is a path from s to v and an edge (v,u) then there is a path from s to u

procedure GraphSearch (G: directed graph, s: vertex)

Initialize X = empty, F = {s}, U = V – F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
    If u is not in X or F:
      move u from U to F.
  Move v from F to X.
Return X.

Correctness, *A*

- A: if $\mathrm{v} \in X$ then there is a path from s to v
- You pick a vertex v in F. (Which vertex depends on the data structure. For the sake of this proof, we can pick any of the vertices in F next.)
- We move all neighbors of v into F if they are in U
- We move v from F to X

procedure GraphSearch (G: directed graph, s: vertex)

Initialize X = empty, F = {s}, U = V – F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
    If u is not in X or F:
      move u from U to F.
  Move v from F to X.
Return X.

Correctness, *A*

- *A*: if $v \in X$ then there is a path from s to v
- You pick a vertex v in F. (Which vertex depends on the data structure. For the sake of this proof, we can pick any of the vertices in F next.)
- We move all neighbors of v into F if they are in U
- We move v from F to X
  - By the induction hypothesis, we know there is a path from s to v

procedure GraphSearch (G: directed graph, s: vertex)

Initialize X = empty, F = {s}, U = V – F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
    If u is not in X or F:
      move u from U to F.
  Move v from F to X.
Return X.

Correctness, *A*

- *A*: if $\mathrm{v} \in X$ then there is a path from s to v
- You pick a vertex v in F. (Which vertex depends on the data structure. For the sake of this proof, we can pick any of the vertices in F next.)
- We move all neighbors of v into F if they are in U
- We move v from F to X
- Thus, it remains true that all elements of F and X are reachable from s

procedure GraphSearch (G: directed graph, s: vertex)

```
Initialize X = empty, F = {s}, U = V – F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
     If u is not in X or F:
        move u from U to F.
  Move v from F to X.
Return X.
```

Correctness, *B*

- By the end of the algorithm, we are guaranteed that F is empty and all elements of X are reachable from s
- *A:* if $v \in X$ then there is a path from s to v
- *B:* if $v \notin X$ then there is not a path from s to v
- Could it be possible that there is some vertex v that is reachable from s but is not in X?

procedure GraphSearch (G: directed graph, s: vertex)

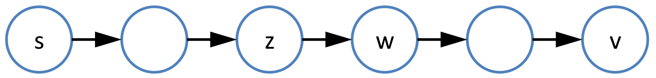
Initialize X = empty, F = {s}, U = V – F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
    If u is not in X or F:
      move u from U to F.
  Move v from F to X.
Return X.

Correctness, *B*

- *B:* if $v \notin X$ then there is not a path from s to v
- Suppose by contradiction that there is a vertex v reachable from s that is not in X. Then there is a path from s to v. Let z be the last vertex in the path that is not in X and w be the first vertex in the path that is not in X.
- Then z must have been in F at some point. And when z was picked from F, w must have been moved from U to F. And down the line, w must have been moved from F to X.

procedure GraphSearch (G: directed graph, s: vertex)

Initialize X = empty, F = {s}, U = V – F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
    If u is not in X or F:
      move u from U to F.
  Move v from F to X.
Return X.



Tension in modifying graph search

- Key point in time analysis: Each vertex v is only explored once (additional times explored give additional factors in time)
- Key point in correctness: Every time a new type or better path to a vertex is found, we need to explore again
- Vanilla graph search: No problem, because there is only one type of path

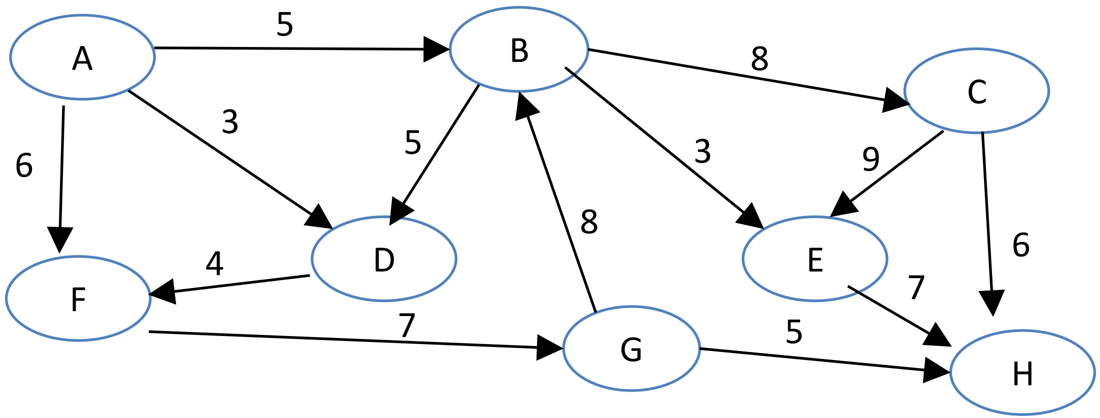
Max bandwidth

- We have an algorithm that takes a graph and starting vertex s and outputs a list of all vertices reachable from s
- How do we use this to solve the max bandwidth problem?
- Break into groups of 4 or 5, discuss approaches and hand in a summary, one per group
  - Don't worry about getting the answer right, just brainstorm ideas

procedure GraphSearch (G: directed graph, s: vertex)

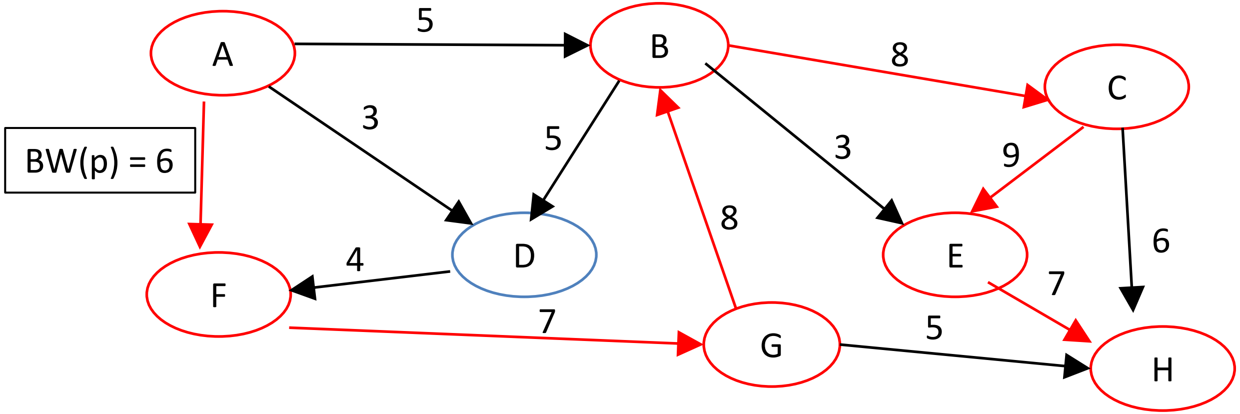
```
Initialize X = empty, F = {s}, U = V – F.
While F is not empty:
  Pick v in F.
  For each neighbor u of v:
      If u is not in X or F:
          move u from U to F.
  Move v from F to X.
Return X.
```

Max bandwidth path



What is the largest bandwidth of a path from A to H?

Max bandwidth path



Max bandwidth path from A to H

Next lecture

- Directed acyclic graphs and strongly connected components
  - Reading: Sections 3.3 and 3.4