COL702: Advanced Data Structures and Algorithms

Thanks to Miles Jones, Russell Impagliazzo, and Sanjoy Dasgupta at UCSD for these slides.

ALGORITHM MINING TECHNIQUES

Deeper Analysis: What else does the algorithm already give us?

Augmentation: What additional information could we glean just by keeping track of the progress of the algorithm?

Modification: How can we use the same idea to solve new problems in a similar way?

Reduction: how can we use the algorithm as a black box to solve new problems?

GRAPH REACHABILITY AND DFS

Graph reachability: Given a directed graph G, and a starting vertex v, return an array that specifies for each vertex u whether u is reachable from v

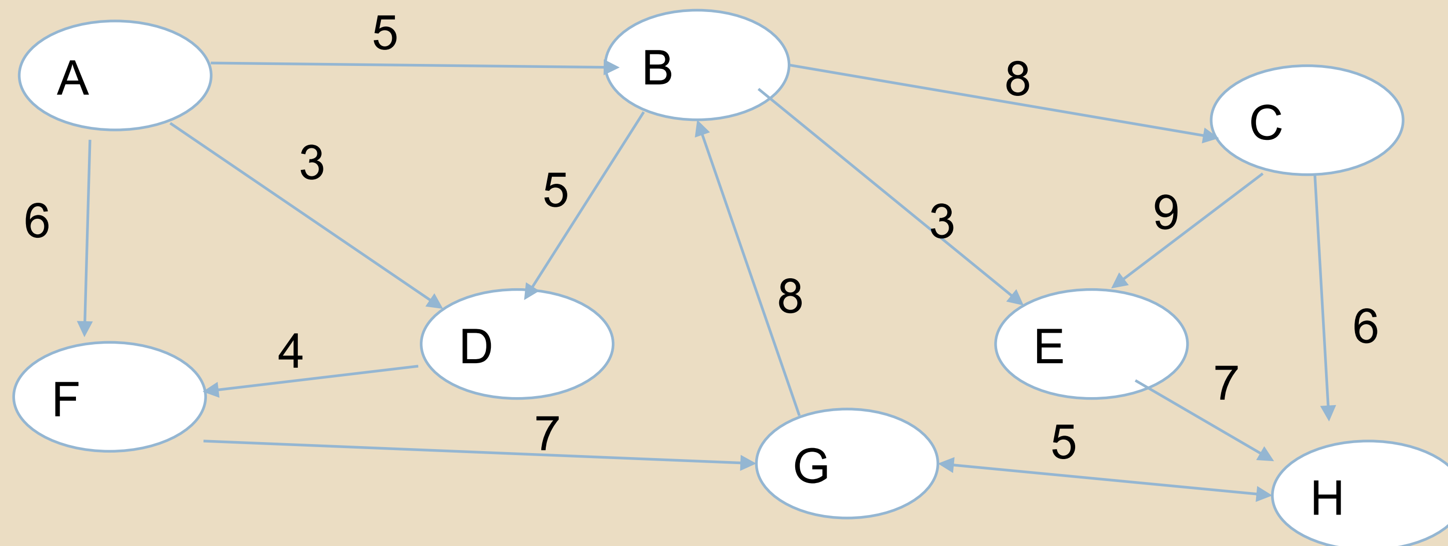
Depth-First Search (DFS): An efficient algorithm for Graph reachability

Breadth-First Search (BFS): Another efficient algorithm for Graph reachability.

MAX BANDWIDTH PATH

Graph represents network, with edges representing communication links.

Edge weights are bandwidth of link, how much can be sent



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

PROBLEM STATEMENT

Instance: Directed graph $G = (V, E)$ with positive edge weights, $w(e)$, two vertices $s, \mathrm{t} \in V$

Solution type: a path p from s to t in G.

Bandwidth of a path:

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

Objective: Over all possible paths p between s and t, find one that maximizes $\mathrm{BW}(p)$.

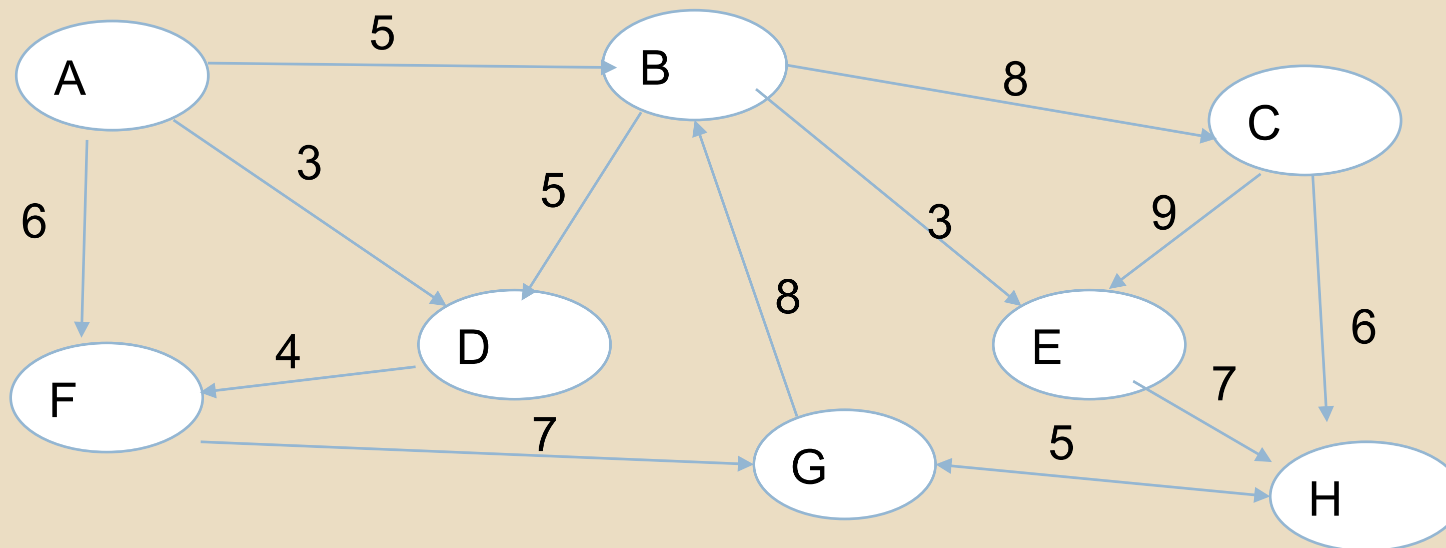
BRAINSTORMING RESULTS

Two kinds of ideas:

- Modify an existing algorithm (DFS, BFS, Dijkstra's algorithm)
- Use an existing algorithm (DFS) as a sub-routine (possibly modifying the input when you run the algorithm

RELATED APPROACH

One approach: "*Add edges from highest weight to lowest, stopping when there is a path from s to t*"



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

REDUCING TO GRAPH SEARCH

These approaches use reductions

We are using a known algorithm for a related problem to create a new algorithm for a new problem

Here the known problem is : Graph search or Graph reachability

The known algorithms for this problem include Depth-first search and Breadth-first search

In a reduction, we map instances of one problem to instances of another. We can then use any known algorithm for that second problem as a sub-routine to create an algorithm for the first.

Graph reachability:

Given a directed graph
G and a start vertex s,

produce the set $X \subseteq V$ of all vertices v reachable from s by a directed path in G.

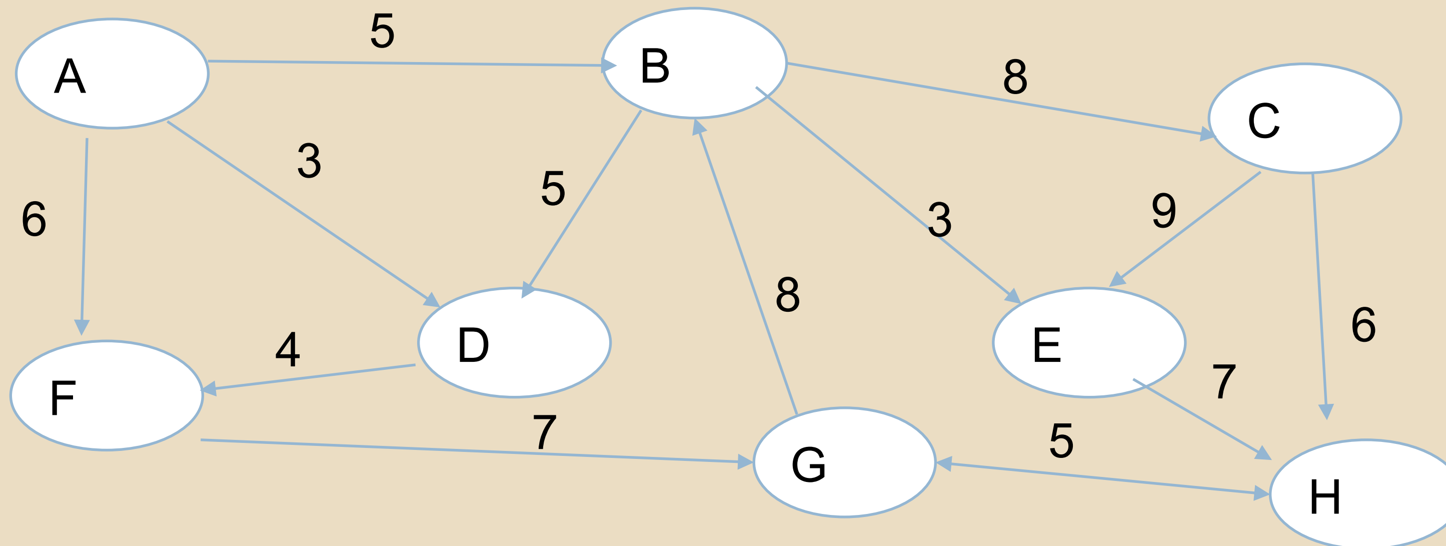
REDUCTION FROM A DECISION VERSION

- Reachability is Boolean (yes, it is reachable or no it is not) whereas MaxBandwidth is optimization (what is the best bandwidth path)
- To show the connection, let's look at a Decision version of Max bandwidth path:
- **Decision Version of MaxBandwidth**

Given G, s, t, B, is there a path of bandwidth B or better from s to t?

MAX BANDWIDTH PATH

Say $B = 7$, and we want to decide whether there is a bandwidth 7 or better path from A to H. Which edges could we use in such a path? Can we use any such edges?



DECISION TO REACHABILITY

Let $E_B = \{ e : w(e) \geq B\}$

Lemma: There is a path from s to t of bandwidth at least B if and only if there is a path from s to t in E_B

DECISION TO REACHABILITY

Let $E_B = \{ e : w(e) \geq B\}$

Lemma: There is a path from s to t of bandwidth at least B if and only if there is a path from s to t in E_B

Proof: If p is a path of bandwidth $BW(p) \geq B$, then every edge in p must have $w(e) \geq B$ and so is in E_B. Conversely, if there is a path from s to t with every edge in E_B, the minimum weight edge e in that path must be in E_B, so $BW(p) = w(e) \geq B$

So to decide the decision problem, we can use reachability: Construct E_B by testing each edge. Then use reachability on s, t, E_B

WHAT THIS ALLOWS US TO DO

Solving one reachability problem, using any known algorithm for reachability, we can answer a ``higher/lower'' question about the max bandwidth:

"Is the max bandwidth of a path at least B?"

REDUCING OPTIMIZATION TO DECISION

Suggested approach

“If we can test whether the best is at least B, we can find the best value by starting at the largest possible one and reducing it until we get a yes answer.”

Here, possible bandwidths = weights of edges

In our example, this is the list: 3, 5, 6, 7, 8, 9

Is there a path of bandwidth 9? If not,

Is there a path of bandwidth 8? If not

Is there a path of bandwidth 7? If not,….

TIME FOR THIS APPROACH

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

From previous classes, we know DFS time $O(n + m)$

When we run it on E_B, no worse than running on E, since

$$|E_B| \leq |E|$$

In the above strategy, how many DFS runs do we make in the worst-case?

What is the total time?

TIME FOR THIS APPROACH

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

From previous classes, we know DFS time $O(n + m)$

When we run it on E_B, no worse than running on E, since

$$|E_B| \leq |E|$$

In the above strategy, how many DFS runs do we make in the worst-case? Each edge might have a different weight, and we might not find a path until we reach the smallest, so we might run DFS m times

What is the total time? Running an $O(n + m)$ algorithm m times means total time $O(m(m + n))$ = $O(m^2)$

IDEAS FOR IMPROVEMENT

Is there a better way we could search for the optimal value?

BINARY SEARCH

Create sorted array of possible edge weights.

3 5 6 7 8 9

See if there is a path of bandwidth at least the median value

Is there a path of bandwidth 6? Yes

If so, look in the upper part of the values, if not, the lower part, always testing the value in the middle

6 7 8 9 Is there a path of bandwidth 8? No

6 7 Is there one of bandwidth 7? No.

Therefore, best is 6

TOTAL TIME FOR BINARY SEARCH VERSION

How many DFS runs do we need in this version, in the worst case?

What is the total time of the algorithm?

TOTAL TIME FOR BINARY SEARCH VERSION

How many DFS runs do we need in this version, in the worst case?

log m runs total = $O(\log n)$ runs

What is the total time of the algorithm?

Sorting array : $O(m \log n)$ with mergesort

$O(\log n)$ runs of DFS at $O(n+m)$ time per run = $O((n+m)\log n)$ time

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

MODIFYING GRAPH SEARCH

This is pretty good, but maybe we can do even better by looking at how graph search algorithms work, rather than just using them as a "black box"

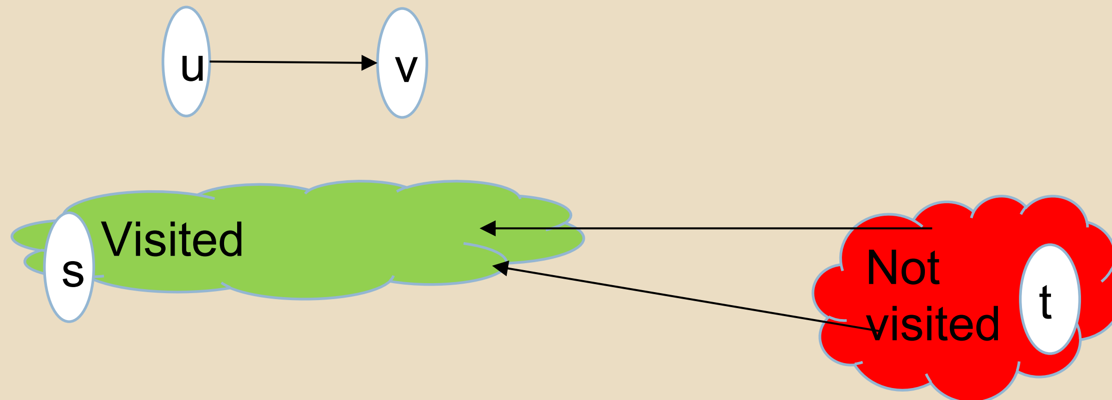
Let's return to a linear search, where we ask "Is there a path of the highest edge weight bandwidth? Second highest?" and so on.

We will use the idea of synergy, that we looked at before. Although each such search takes linear time worst-case, and we have a linear number of them, we'll show how to do ALL of them together in the worst-case time essentially of doing ONE search.

WHAT IS THE DIFFERENCE BETWEEN SEARCHES?

Can think of adding just one edge at a time, from highest weight to lowest weight. So the different searches just differ by a single edge.

What can happen? Before we add in the next edge, say from u to v, some of the nodes were marked visited, others not. s must be marked, but not t



What are the possible cases about u, v?
What happens to reachable set in each case?

UPDATING VISITED: CASE 1

Case 1: u and v were both visited. How does the set of visited vertices change?

UPDATING VISITED: CASE 2

Case 2: u is not reachable (and v can be either reachable or not). How does the set of reachable vertices change ?

UPDATING VISITED: CASE 3

Case 3: u is reachable and v is not reachable. How does the set of reachable vertices change ?

UPDATING VISITED: CASE 3

Case 3: u is reachable and v is not reachable. Anything reachable from v should become reachable, but we don't need to re-explore already discovered parts of the graph.

Run explore(G,v), but don't erase visited before doing it.

UPDATING VISITED: CASE 3 TIME ANALYSIS

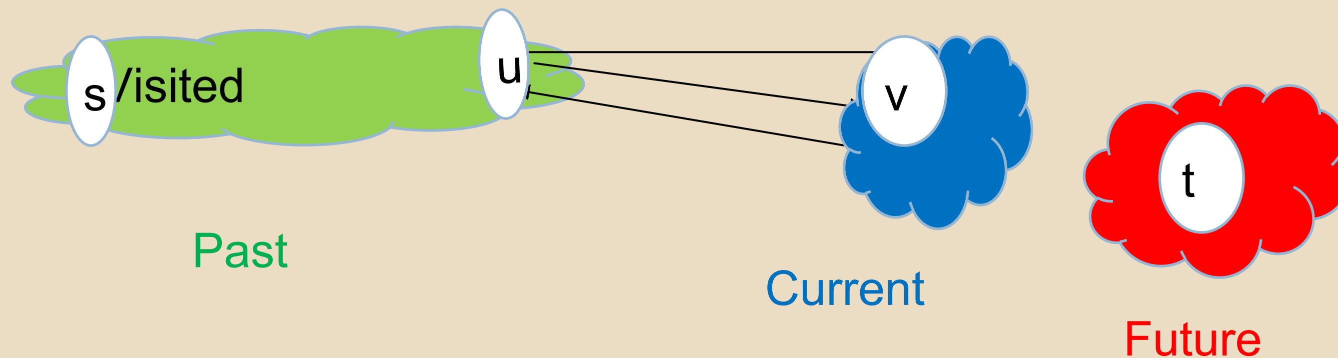
Note: other cases, constant time per edge.

Case 3: u is reachable and v is not reachable. Run explore(G,v), but don't erase visited before doing it. Could be up to linear time BUT:

UPDATING VISITED: CASE 3 TIME ANALYSIS

Note: other cases, constant time per edge.

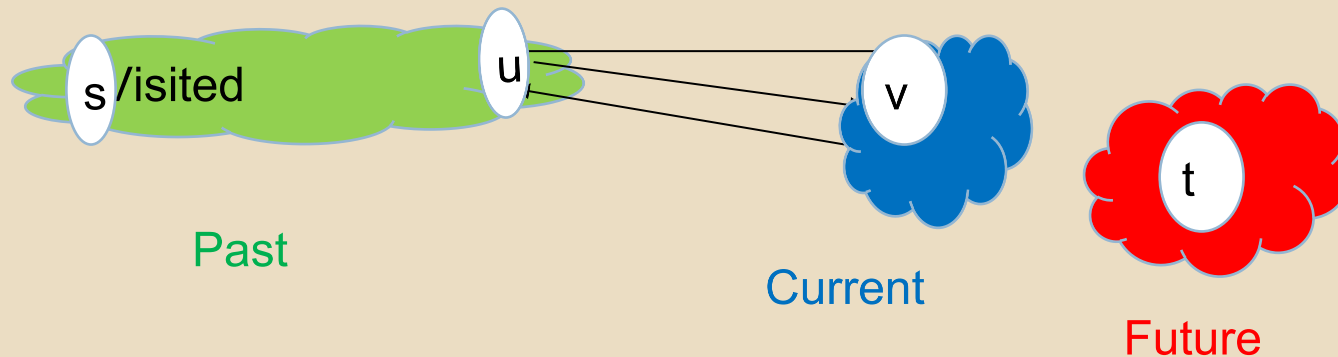
Case 2: u is reachable and v is not reachable. Run explore(G, v), but don't erase visited before doing it. Could be up to linear time BUT time For this search is at most size of region discovered in THIS search, which is disjoint from past and future searches!



UPDATING VISITED: CASE 3 TIME ANALYSIS

Could be up to linear time BUT time

For this search is at most size of region discovered in THIS search, which is disjoint from past and future searches! Therefore, total time for ALL searches is at most sum of sizes of parts discovered in each, at most all the edges.



ANOTHER WAY TO VIEW IT

Although we are performing explores in a different order, we still only explore from each vertex ONCE, overall.

So total time for all explores is still $O(n + m)$ for the same reason as Before.

One cheat: We assumed the edges came sorted. If we need to sort them , this could take $O(m \log m)$ time, for a total of $O(n + m \log m)$, essentially the same as the binary search method. (Under many circumstances, there are faster ways to sort, e.g., counting, radix sort)

REDUCTION VS. MODIFICATION

Reduction is in many ways easier and less confusing, once you get the hang of it. It is more modular, in that you can treat the original algorithm, its correctness and its time analysis all in a “black box” manner.

Modification is sometimes necessary if we can’t come up with a reduction. It can also lead sometimes to better algorithms than reductions. But using modifications successfully require us not just to know WHAT the starting algorithm is, but WHY it works and WHY it is fast.

SYNERGY

Sometimes, solving related problems many times is cheaper in bulk than solving them each individually would be. We should look for places where problems overlap, or where the problems are changing incrementally.