Mining Algorithms for New Applications: Modifying vs. Reductions
Sanjoy Dasgupta
Russell Impagliazzo
Ragesh Jaiswal

Credit: Some of today’s slides are due to Miles Jones

CSE 101, Spring 2020, Week 2
Algorithm Mining

- Algorithms designed for one problem are often usable for a number of other computational tasks, some of which seem unrelated to the original goal.

- Today, we are going to look at how to use the depth-first search algorithm to solve a variety of graph problems.
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.
DFS as recursion

- procedure explore(G,v)
- Input: graph G = (V,E); node v in V output:
  - Output: array visited[u]
  - 1. visited[v] = true
  - 2. for each edge (v,u) in E do:
    - if not visited[u]: explore(G,u)
Key Points of DFS

• No matter how the recursions are nested, for each vertex u, we only run explore(u) ONCE, because after that, it is marked visited. (We need this for termination and efficiency)

• On the other hand, we discover a path to a new destination, we always explore all new vertices reachable (We need this for correctness, to guarantee that we find ALL the reachable vertices)
Bipartite graphs

- Last week, we looked at the graph coloring problem:
- Give the vertices of an undirected graph colors so that neighboring vertices get different colors.
- Use as few as possible distinct colors.
- Special case: 2 colorable graphs = bipartite graphs (bipartite = 2 sides)
When is a graph bipartite?
When is a graph bipartite?
When is a graph bipartite?
When is a graph bipartite?
A criterion for being bipartite

• Theorem: A graph is bipartite if and only if it has no odd cycles.

• Proof: If a graph has an odd cycle, it is NOT bipartite.
A criterion for being bipartite

- Theorem: A graph is bipartite if and only if it has no odd cycles.

- Proof: If a graph has an odd cycle, it is NOT bipartite.
Other direction

- If a graph has no odd cycles, then it is bipartite.
- In each connected component, pick one node $x$. Color $y$ red if it is connected via an even length path to $x$, blue if to an odd length path. There’s always one or the other but not both. An even length path from $x$ to $y$, followed by an odd length path from $y$ to $x$ = odd cycle. Since an even path followed by an edge = odd path, neighbors have different colors.
Odd vs. even paths

• Odd vs. even reachability: which vertices are reachable from $v$ by odd length paths? Even length paths?
• Bipartiteness only makes sense in undirected graphs, but odd vs. even paths makes sense in either, so we’ll also look at this question in directed graphs.
Iterative DFS modified, attempt one

procedure DFS (G: directed graph, v: vertex)

Initialize array visited[u] to False, color[u] to NIL
Initialize stack of vertices F, PUSH v; Visited[v]==True; color[v]==0
While F is not empty:
  v==Pop;
  For each neighbor u of v (in reverse order):
    If not visited[u]:
      Push u; visited[u] == True; color[u] == 1 – color [v]

Return visited
Doesn’t always work

- While this modified DFS works for coloring bipartite graphs, it doesn’t detect odd cycles, and it doesn’t work when there are both even and odd paths to vertices, because it only sets one color. We need to re-explore vertices when we find paths of the other type.
Example

We need to do explore again from B. After we discover the even length path via C, B, D, F, G, have both even and odd length paths.
Iterative DFS modified, attempt two

procedure DFS (G: directed graph, v: vertex)

Initialize arrays visited[u, color] to False (u in V, color =0,1), Initialize stack of vertices F, PUSH (v,0); Visited[v,0]==True

While F is not empty:
(v, color)==Pop;
For each neighbor u of v (in reverse order):
   If not visited[u, 1-color]:
      Push (u, 1-color); visited[u, 1-color] == True;
Return visited
Correctness

- Modify argument from DFS: Loop invariant: every time \([u,\text{color}]\) is marked True, there is a path from \(v\) to \(u\) of parity color.

- Induction along path: There is no first time on a path that the \(J\) th node is not marked visited for color \(J \mod 2\).
Time analysis

• It’s no longer true that each vertex is pushed on the stack at most ONCE

• However,....
Time analysis

• It’s no longer true that each vertex is pushed on the stack at most ONCE.

• However, each vertex is pushed on the stack at most TWICE, once per color. Therefore at most twice the total time of previous version.
As a reduction

- When we modify algorithms, we need to go back and look at not just the claims of correctness, but the proofs of correctness. We also need to reconsider the time analysis from scratch.
- We can rephrase the same algorithm as a reduction, using DFS unmodified, but on a modified input (instance)
Reduction
V’= two copies of each vertex in V, one representing reaching it on an even path, the other on an odd path.
For each edge (u,v) in E, Add two edges: (u0, v1) and (u1, v0) to E'
For each edge \((u,v)\) in \(E\),
Add two edges: \((u0, v1)\) and \((u1, v0)\) to \(E'\)

In \(G'\), Run DFS
From \(A0\).
Correctness

• Claim: \( u_0 \) is reachable in \( G' \) from \( A_0 \) if and only if there is an even length path in \( G \) from \( A \) to \( u \).

• Proof: If \( p \) is an even length path from \( A \) to \( u \) in \( G \), let \( p' \) be the path that follows \( p \), but switches sides every step. Since \( p \) is even , \( p' \) will switch sides an even number of times, and end at \( u_0 \). If \( p' \) is a path from \( A_0 \) to \( u_0 \) in \( G' \), it must switch sides every time. So if we write down the same list of vertices, but ignore sides, we must get an even length path \( p \) from \( A \) to \( u \).
Correctness part 2

- We have already proved DFS is correct.
- So when we run DFS on G’, we will mark u0 visited if and only if it is reachable from A0, if and only if (by the lemma) it is reachable via an even path in G.
Time analysis

• We already know DFS takes time $O(|V|+|E|)$
• So running DFS on $G'$ takes time $O(|V'|+|E'|)$
• $|V'|=2|V|$, $|E'|=2|E|$, so this is also $O(|V|+|E|)$.
• Also time to compute $G'$ is $O(|V|+|E|)$, two steps per Vertex to create new vertices, two steps per edge to Insert edges.
Total time is still $O(|V|+|E|)$. 
Reductions

• Create new instance
• Run existing algorithm on new instance
• Show that old problem on new instance = new problem on original instance.
• Run time: Time to create new instance + time of old algorithm on sizes for new instance
MAX BANDWIDTH PATH

Graph represents network, with edges representing communication links, weights represent max rate for that link.

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 $s, t \in V$

- Solution type: a path $p$ from $s$ to $t$ in $E$.

- Bandwidth of a path:
  
  \[
  BW(p) = \min_{e \in p} w(e)
  \]

- Objective: Over all possible paths $p$ between $s$ and $t$, find one that maximizes $BW(p)$. 
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)
Discuss approaches on piazza

• We’ll use a summary of approaches you came up with and approaches from previous classes in Friday’s lecture.