Fundamental Graph Algorithms
Part Four
Announcements

• Problem Set One due right now.
  • Due Friday at 2:15PM using one late period.
• Problem Set Two out, due next Friday, July 12 at 2:15PM.
  • Play around with graphs and graph algorithms!
Outline for Today

- **Kosaraju's Algorithm, Part II**
  - Completing our algorithm for finding SCCs.
- **Applying Graph Algorithms**
  - How to put these algorithms into practice.
Recap from Last Time
Strongly Connected Components

• Let $G = (V, E)$ be a directed graph.
• Two nodes $u, v \in V$ are called strongly connected iff $v$ is reachable from $u$ and $u$ is reachable from $v$.
• A strongly connected component (or SCC) of $G$ is a set $C \subseteq V$ such that
  • $C$ is not empty.
  • For any $u, v \in C$: $u$ and $v$ are strongly connected.
  • For any $u \in C$ and $v \in V - C$: $u$ and $v$ are not strongly connected.
Condensation Graphs

- The **condensation** of a directed graph $G$ is the directed graph $G^{SCC}$ whose nodes are the SCCs of $G$ and whose edges are defined as follows:

  $$(C_1, C_2) \text{ is an edge in } G^{SCC} \text{ iff } \exists u \in C_1, v \in C_2. (u, v) \text{ is an edge in } G.$$

- In other words, if there is an edge in $G$ from *any* node in $C_1$ to *any* node in $C_2$, there is an edge in $G^{SCC}$ from $C_1$ to $C_2$.

- **Theorem:** $G^{SCC}$ is a DAG for any graph $G$. 
How do we find all the SCCs of a graph?
Topological Sort(ish)

- If we look purely at the *last* node from each SCC to turn green, we get a topological sort of $G^{SCC}$ in reverse.
  - Here, each SCC is represented by a single node.
  - We proved this result last time.
- There's still a problem – we still don't have a way of identifying the last node of each SCC!
- We do have one foothold, though...
- **Onward to new content!**
Making Progress!

• The last node colored green by DFS must be the last node colored green in some SCC.

• This gives a rough idea for an algorithm:
  • Take the last node in the ordering that hasn't already been put into an SCC.
  • Find all nodes in the same SCC as that node.
  • Repeat.
Claim 1: This node must belong to a source SCC.
Claim 2: The SCCs of this reversed graph are the same as the SCCs of the original graph.
Claim 3: Since $E$ is in a source SCC in the original graph, $E$ is in a sink SCC in this graph.
Claim 4: The only nodes reachable from E are the nodes in the same SCC as E.
Claim 5: The only unvisited nodes reachable from \( H \) are the nodes in the same SCC as \( H \).
procedure kosarajuSCC(graph G):
    for each node v in G:
        color v gray.

    let L be an empty list.
    for each node v in G:
        if v is gray:
            run DFS starting at v, appending each
            node to list L when it is colored green.

    construct $G^R$ from G.
    for each node v in $G^R$:
        color v gray.

    let scc be a new array of length n
    let index = 0
    for each node v in L, in reverse order:
        if v is gray:
            run DFS on v in $G^R$, setting scc[u] = index
            for each node u colored green this way.
        index = index + 1
    return scc
Proving Correctness

- Here's a quick sketch of the correctness proof of Kosaraju's algorithm:
  - As proven earlier, the last nodes in each SCC will be returned in reverse topological order.
  - Each time we do a DFS in the reverse graph starting from some node, we only reach nodes in the same SCC or in ancestor SCCs.
  - Since we process the SCCs in topological order, at each point the only unvisited nodes reachable are nodes in the same SCC.
Kosaraju's Algorithm Runtime

- What is the runtime of the Kosaraju's algorithm?
  - Runtime for running DFS starting from each node in the graph: $\Theta(m + n)$.
  - Runtime for reversing the graph and coloring all nodes gray: $\Theta(m + n)$.
  - Runtime for running DFS in the reversed graph: $\Theta(m + n)$.
  - Total runtime: $\Theta(m + n)$.
- This is a **linear-time algorithm**!
Why All This Matters

- Depth-first search is an important building block for many other algorithms, including topological sorting, finding connected components, and Kosaraju's algorithm.

- We can find CCs and SCCs in (asymptotically) the same amount of time.

- Further reading: look up Tarjan's SCC algorithm for a way to find SCCs with a single DFS!
Applied Graph Algorithms
The Story So Far

- We have now seen many algorithms that operate on graphs:
  - BFS
  - DFS
  - Dijkstra's algorithm
  - Topological sort (x2)
  - Finding CCs
  - Kosaraju's algorithm

- How do we apply these in practice?
Reusing Algorithms

- Developing new graph algorithms is **hard!**
- Often, it is easier to solve a problem on graphs by reusing existing graph algorithms.

**Key idea:** Use an existing graph algorithm as a “black box” with known properties and a known runtime.

- Makes algorithm easier to write: can just use an off-the-shelf implementation.
- Makes correctness proof easier: can “piggyback” on top of the existing correctness proof.
- Makes algorithm easier to analyze: runtime of key subroutine is known.
Sample Problem: **Minimizing Turns**
Minimizing Turns

- You are given a (possibly directed) graph $G = (V, E)$ where each edge goes either north, south, east, or west.
- You begin driving in some direction $d$.
- **Goal**: Find the path from $s \in V$ to $t \in V$ that minimizes the total number of turns made.
What This Looks Like

- This problem doesn't exactly match any of the algorithms we've seen so far.
- Similar to a shortest path problem, but we're charged whenever we make a turn, rather than whenever we follow an edge.
- Could we relate this back to BFS or Dijkstra's algorithm?
Shortest Paths as a Black Box

• Here's what we have now:

• Here are two options for solving our problem:
  • Open up the black box and try to change how it finds shortest paths. (Harder)
  • Change which input we put into the black box to trick it into solving our problem. (Easier)
Reductions

- Goal: Take our given graph \( G = (V, E) \), starting node \( s \), and starting direction \( d \), then build a new graph \( G' = (V', E') \) such that the following holds:

  \( \textbf{Shortest paths in } G' \textbf{ correspond to minimum-turn paths in } G. \)

- If we can build this graph \( G' \), our algorithm will be the following:
  - Build the graph \( G' \) out of \( G \), \( s \), and \( d \).
  - Use an existing algorithm for finding shortest paths to find shortest paths in \( G' \).
  - Using the shortest paths found in \( G' \), determine the minimum-turn path from \( s \) to \( t \).
A Major Observation

- When computing shortest paths in a graph, each node represents a possible “position” we can be in.

- In our problem, though, “position” also includes the direction you are currently facing.

- **Useful technique:** What if we create one node in the graph for each combination of a position in the original graph and a current direction?
The Construction

- For each $v \in V$, construct four nodes: $v_N$, $v_S$, $v_E$, $v_W$
- For each edge $(u, v) \in E$ that goes in direction $d$, construct four edges: $(u_N, v_d)$, $(u_S, v_d)$, $(u_E, v_d)$, $(u_W, v_d)$
- Assign costs as follows:
  - $l(u_{d_1}, v_{d_2}) = 0$ if $d_1 = d_2$
  - $l(u_{d_1}, v_{d_2}) = 1$ if $d_1 \neq d_2$
- New graph has $4n$ nodes and $4m$ edges.
procedure minTurnPath(graph G, node s, node t, direction d):
    construct G' from G as described earlier.
    run Dijkstra's algorithm to find shortest paths from \( s_d \) to each other node in G'.
    return the shortest of the following paths:
    the shortest path from \( s_d \) to \( t_N \)
    the shortest path from \( s_d \) to \( t_S \)
    the shortest path from \( s_d \) to \( t_E \)
    the shortest path from \( s_d \) to \( t_W \)
Correctness Proof Sketch

- Suppose we start at node $s$ facing direction $d$. Our goal is to get to node $t$ minimizing turns.
- Consider the length, in the new graph, of the shortest path $P$ from $s_d$ to $t_x$ for any direction $x$.
- $l(P)$ is the sum of all the edge costs in path $P$. Edges that continue in the same direction cost 0 and edges that change direction cost 1, so $l(P)$ is the number of turns in $P$.
- Since $P$ is chosen to minimize $l(P)$, $P$ has the fewest number of turns of any path from $s_d$ to $t_x$.
- The minimum-turn path from $s$ to $t$ is then the cheapest of the paths from $s_d$ to $t_N$, $t_S$, $t_E$, $t_W$. 
Formalizing the Proof

• To be more formal, we should prove the following results:

  • **Lemma 1:** There is a path in $G'$ from $s_{d_1}$ to $t_{d_2}$ iff there is a path in $G$ from $s$ to $t$ that starts in direction $d_1$ and ends in direction $d_2$.

  • **Lemma 2:** There is a path in $G'$ from $s_{d_1}$ to $t_{d_2}$ of cost $k$ iff there is a path in $G$ from $s$ to $t$ that starts in direction $d_1$, ends in direction $d_2$, and makes $k$ turns.

• **We will expect this level of detail in the problem sets.**
Analyzing the Runtime

- Time required to construct the new graph: $\Theta(n + m)$, since there are $4n$ nodes and $4m$ edges and each can be built in $\Theta(1)$ time.
- Time required to find the shortest paths in this graph: $O(n^2)$, or better if we use a faster Dijkstra's implementation.
- Overall runtime: $O(n^2)$.
Speeding Things Up

• The algorithm we've described is correct, but it can be made more efficient.
• Observation: Every edge in the graph has cost 0 or 1.
• Our algorithm uses Dijkstra's algorithm in this graph.
• Can we speed up Dijkstra's algorithm if all edges cost 0 or 1?
Some Observations

• Dijkstra's algorithm works by
  • Choosing the lowest-cost node in the fringe.
  • Updating costs to all adjacent nodes.

• **Fact 1:** Once Dijkstra's algorithm dequeues a node at distance \( d \), all further nodes dequeued will be at distance \( \geq d \).

• Can prove this inductively: Initial distance is 0, and all other distances are formed by adding edge costs (which are nonnegative) to the distance of the most recently-dequeued node.
Some Observations

- **Fact 2:** If all edge costs are 0 or 1, every node in the queue will either be at distance $d$ or distance $d + 1$ for some $d$.

- Can prove this by induction:
  - Initially, all nodes in the queue are at distance 0.
  - If all nodes are at distance $d$ or $d + 1$, we dequeue a node at distance $d$. All nodes connected to it will then be reinserted at distance either $d$ or $d + 1$. 
A Better Queue Structure

• Store the queue as a doubly-linked list. Elements at the front are at distance $d$ and elements at the back are at distance $d + 1$.
  
  • Enqueue: Compare distance to distance at front. If equal, put at front. If greater, put at back.
  
  • Dequeue: Remove first element.
  
  • If a distance decreases from $d + 1$ to $d$, move that element to the front.

• All operations can be done in O(1) time.
Theorem: In a graph where all edge costs are 0 or 1, Dijkstra's algorithm runs in time $O(m + n)$.

Proof Sketch: Use this new queue structure to store the nodes. Dijkstra's algorithm takes time $O(m + n)$ plus the time required for $O(m + n)$ queue operations, which with the new structure run in time $O(1)$ each. Thus the runtime is $O(m + n)$.

Corollary: The minimum-turns path problem can be solved in linear time.
Why All This Matters

- Look at the structure of our solution:
  - Show how to solve the new problem (minimizing turns) using a solver for an existing algorithm.
  - Argue correctness using the fact that the existing algorithm is correct.
  - Argue runtime using the runtime of the existing algorithm.
  - *(Optional)* Speed up the algorithm by showing how to faithfully simulate the original algorithm in less time.

- Many problems can be solved this way.
Next Time

- Divide-and-Conquer Algorithms
- Mergesort
- Solving Recurrences