Dijkstra’s algorithm, Other graph problems

6.100 LECTURE 14

SPRING 2026

Announcements

- Pset 4 due Friday 3/20
- Pset 4 checkoffs throughout the week after break 3/30–4/3
- Pset 5 released Wednesday after break 4/1

- Start studying for Exam 2
  - covers material through today’s lecture, Friday’s recitation, and Pset 4
- Will release practice exam on Monday 3/30

Shortest paths on weighted graphs

- Task explained in **Pset 4 Section 3**
  - weights add up along paths
  - assume all weights are positive integers (for now)
    - doesn't make sense to have negative or zero road lengths
    - but it might make sense to have negative energy usage
- Pset strategy
  - **expand each edge into a sequence of unit-length edges**
  - **run BFS** to get a shortest path
  - collapse expanded edge sequences into original edges
- Today
  - edge expansion can be wasteful, makes **number of BFS frontiers depend on edge weights**
  - can we simulate the BFS frontiers more efficiently?

Dijkstra’s algorithm

Building on BFS

- **Core BFS idea:** expand from current frontier to next frontier
  - **current frontier** is guaranteed to be at a **certain shortest distance**
  - **next frontier** is hence built to contain all nodes with **next shortest distance**
- This reasoning is known as **induction,** very close link to **recursion**
  - in fact, could build a **recursive BFS**
    - to find frontier n, first find **frontier n-1 and visited set**
    - then expand to frontier n while updating visited set

Skipping frontiers

- In weighted graphs, should be able to “skip” intermediate frontiers
- Suppose we have a **current frontier**
  - when we expand a node on the frontier to unseen neighbors, those edge weights may not all be the same
  - **so not all those neighbors are guaranteed to be on next valid frontier**
  - only those edges with the smallest weight could have possibly reached the next valid frontier

Handling temporary frontiers

- **Situation 1**
  - current frontier 5 with node D
  - expand D to neighbors (F, 2), (G, 4), (H, 2)
- Store newly discovered nodes at temporary frontiers so far
- The smallest next frontier is the only possible valid one, because for nodes on farther frontiers, there may be shorter paths through closer frontiers

Handling temporary frontiers

- **Situation 2**
  - current frontier 5 with nodes D and E
  - expand D, then expand E to neighbor (H, 1)
- If current smallest frontier is not exhausted yet, then further frontiers are not yet valid
- Expanding E causes H's distance to go from 7 to 6, move it to earlier frontier

Handling temporary frontiers

- **Situation 3**
  - current frontier 6 with nodes H and J
  - expand J to neighbor (G, 4)
- Found path to G with weight 10
- But G is already on frontier 9, so keep it there

Handling temporary frontiers

- **Situation 4**
  - current frontier 5 with nodes D and E
  - expand D, then expand E to neighbors (H, 1), (B, 1), (J, 1)
- Expanding E finds path to B with distance 6
- But already processed B with distance 2
  - must be shortest path, because we expanded D with shortest path
  - so B is already finished on previous valid frontier
  - any subsequent edge to it will never improve its distance
- Set of **finished nodes** is akin to visited set in plain BFS

Implementing Dijkstra’s algorithm

- Build on FIFO queue idea from last time
  - but as we add paths to queue, not guaranteed that frontiers are in order
- Instead, **label each node with temporary frontier,** store as **(cost, path)** tuple
  - need a **`remove_min()`** functionality to get node/path off of true frontier
- Also, when first encounter goal, cannot guarantee that it is on best frontier
  - **perform goal check when removing from priority queue instead**
  - **and visited set becomes a finished set**
- Finally, when **exploring neighbors,** construct new path and cost, but **compare with what's in the queue** already, wrap in **`update_queue()`** functionality
  - **case 1:** node not in queue yet, add (cost, path)
  - **case 2:** node in queue with larger cost, update (cost, path)
  - **case 3:** node in queue with smaller or equal cost, do nothing

Dijkstra's with predecessors

- Every time we update the queue due to exploring edge X→Y, we're saying it caused the shortest path to Y known so far to end with X→Y
  - at this point X is already finished
  - but unless it's the start S, it also got put on the queue at some point with edge W→X
  - maybe it got updated a couple more times with edges V→X and U→X, but only that last update matters
  - the shortest path we find for X ends with U→X
- So store a **backward predecessor reference** whenever a node is added or updated on the queue
  - by the time it's removed, the predecessor will be on a correct shortest path
- Simply **trace predecessors back to root** and **reverse** the order

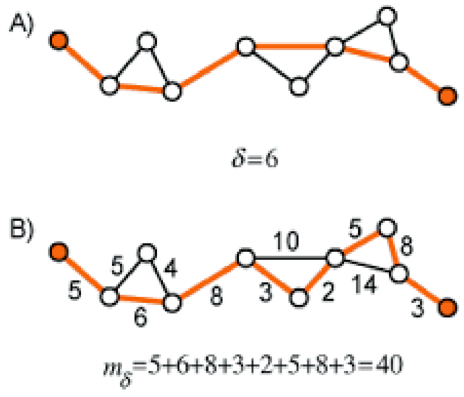
Other graph problems

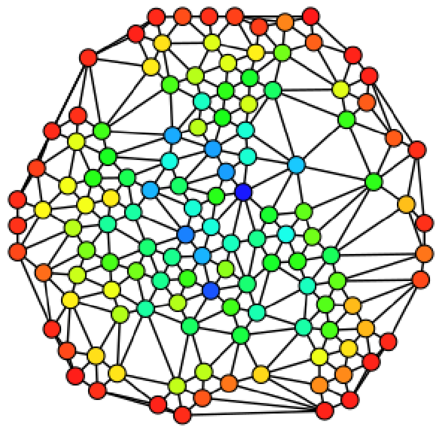
What else can graph search algorithms do?

- BFS/DFS can determine **connected components**
  - image segmentation
  - social network analysis
- DFS can detect **cycles**
  - relevant for **strongly** connected components on directed graphs, where every node can reach every other node
- BFS/Dijkstra’s can find shortest paths from a single node to all other nodes it can reach
  - builds a **shortest-path tree**

All-pairs shortest paths

- Compute shortest paths between **all pairs** of nodes
- **Applications**
  - **longest shortest path (i.e., diameter)**
    - e.g., in a communications network, the longest a message may have to travel, affects worst-case latency
  - **betweenness centrality**
    - the number of shortest paths that pass through a node
    - indicates importance, e.g., reliability/connectivity in physical networks, power/influence in social networks
- **Approaches**
  - could run BFS/Dijkstra's from all nodes
  - but can reused computation, some shortest-path trees may overlap





Minimum spanning tree

- **Scenario:** utilities company needs to provide coverage over geographic network, what is minimum-cost infrastructure?
  - e.g., electrical grid, water pipelines, bike lanes, clearing snow, etc.
- **Problem:** Given a connected undirected graph, find a connected subset of edges that covers/reaches all nodes
  - must be a **tree**
  - with undirected edges, **any node** in a tree can be interpreted as the **"root"**

Network flow problems

- **Scenario:** transport goods from a **source** (e.g., company, factory) to a **target/sink** (e.g., customer, vendor)
  - graph **edges (directed)** each have a **capacity**
  - assign a **flow** to each edge that is within capacity
  - how to handle multiple targets or multiple sources?
    - **hint:** how did you handle goals across multiple layers in Pset 4?
- **Max-flow problem:** transport as much goods as possible
  - **conservation constraint: flow in == flow** out for all nodes except source and target
  - **maximize flow out** of source == **flow into** target
  - relationship to **min-cut**
    - a cut divides the nodes into two groups
    - there exists a cut where the **total flow going across the groups is the bottleneck** in total flow capacity

Network flow problems

- **Scenario:** transport goods from a **source** (e.g., company, factory) to a **target/sink** (e.g., customer, vendor)
  - graph **edges (directed)** each have a **capacity**
  - assign a **flow** to each edge that is within capacity
  - how to handle multiple targets or multiple sources?
    - **hint:** how did you handle goals across multiple layers in Pset 4?
- **Min-cost flow problem:** minimize the cost of a desired flow amount
  - each edge now also has a cost per unit flow
  - more general than **max-flow problem**
  - can also solve the **shortest-path problem**
    - set each edge's unit cost to be the edge weight/distance
    - set total flow to be 1 (or some constant)
    - set all edge capacities to be infinity (or at least $\geq$ total flow)

After break

- Monday 3/30
  - review material
  - introduce new Python convenience features
- **Exam 2 on Monday 4/6**
- Wednesday 4/1
  - start new unit on combinatorial optimization
  - build on graphs
  - algorithmic technique called **dynamic programming**