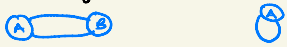
LECTURE 19: Graphs

3/26/24

- graphs w/o parallel edges and w/o self-loops are simple graphs



- simple path: path b/w 2 vertices w/ no repeating vertices
- connected graph: ∃ simple path b/w any pair of vertices
- cycle: simple path w/ same first and last nodes
- complete graph: all edges

$$|E| = \frac{|V| * (|V|-1)}{2} \approx |V|^2$$

- dense graph: many edges

  $|E| \approx |V|^2$

  represent as adjacency matrix

- sparse graph: few edges

  $|E| << |V|^2$ or $|E| \approx |V|$

  represent as adjacency list

- all trees are sparse b/c they are barely connected → better for dense
- adjacency matrix is matrix w/ 1s (there is an edge) and 0s (there isn't)
  - for directed set rows/column as to/from
- distance matrix stores the weights where the 1s are
  - to note where there aren't edges, use ∞
  - `#include <limits>`
    `numeric_limits<double>::infinity()`
- adjacency list stores which nodes you can get to from each node. → better for sparse
  - can also store distances
  - $\sim \frac{E}{V}$ edges in each vertex list ($\frac{2E}{V}$ for undirected)
  - access vertex list: $O(1)$
  - find edge: $O(\frac{E}{V})$
  - average cost for one vertex is $O(1+\frac{E}{V})$
  - cost for all vertices is $V * O(1+\frac{E}{V}) = O(E+V)$
  - in undirected, each edge is in adj. list twice

e.g. Time Complexities to find if an edge is there:

adjmat: worst, best, avg all $O(1)$

adjlist: worst: $O(V)$
best: $O(1)$
avg: $O(1+\frac{E}{V})$

e.g. Time Complexities to find closest vertex to a given vertex in weighted graph:

adjmat: worst, best, avg. all $O(V)$

adjlist: worst: $O(V)$ best: $O(1)$ avg: $O(1+\frac{E}{V})$

e.g. determine if there is an edge from a vertex X

adjmat: worst: $O(V)$
best: $O(1)$
avg: $O(V)$

adjlist: all $O(1)$

- to determine most efficient (lowest cost) route from X to Y, first do SSSP (need to know all possibilities before deciding)
- single-source shortest path: find shortest path to get to any vertex from a given starting vertex
  - DFS: only works on trees; may give wrong answer due to multiple paths in a graph
  - BFS: works for unweighted edges or where all weights are same
  - Djikstra's Alg: works for weighted edges
- DFS:
  - uses stack
  - works on graphs & digraphs
  - find if a path exists from s to g (may not be shortest)

```
Algorithm GraphDFS
  Mark source as visited
  Push source to Stack
  While Stack is not empty
     Get/Pop candidate from top of Stack
     For each child of candidate //adjacent
        If child is unvisited
           Mark child visited
           Push child to top of Stack
           If child is goal
              Return success
  Return failure
```

  - to backtrack, need to store how you got there.
- DFS analysis of adjlist:
  - every vertex at most once - $O(V)$
  - look through adjlist for every vertex - $O(1+\frac{E}{V})$ /vertex
  - total: $O(E+V)$
    sparse: $O(V+V) = O(V)$
    dense: $O(V^2+V) = O(V^2)$
- DFS analysis of adj.mat:
  - every vertex at most once - $O(V)$
  - look through adjmat row for every vertex - $O(V)$
  - total: $O(V^2)$ (always)
- BFS:
  - given unweighted graph, discovers shortest path from s to g
  - queue
  - works on graphs & digraphs
  - same code as DFS, but with queue stuff
    ↳ complexity also the same

LECTURE 20: MST

3/29/24

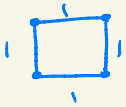
- MST Problem: given weighted, undirected, connected graph G, find subgraph T s.t. sum of weights is minimal and all vertices are connected

e.g. Prove that a unique shortest edge must be in every MST:

Suppose it's not. And then add it. This creates a cycle. Remove biggest weight (not 1) to get an MST w/ 1. Makes a smaller sum of weights ↯

- smallest 2 weights need to be in MST. 3rd smallest is graph has >3 nodes

e.g. Graph w/ >1 MST AND graph where MST doesn't have shortest edge:



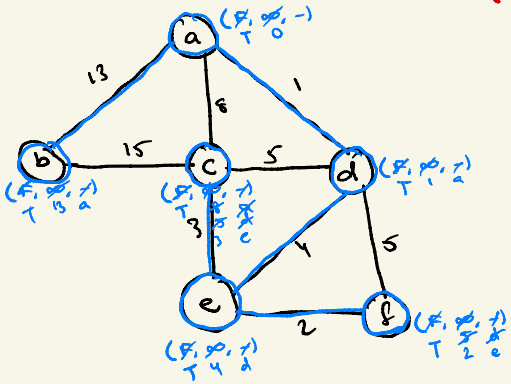
- Prim's Algorithm:
  - greedily select edges one-by-one and add to a growing sub-graph to get a tree
  1) initialize tree w/ one arbitrary vertex
  2) find min-weight edge of all edges that connect tree to other vertices and add that vertex
  3) repeat

- have two groups (visited, not visited)
- choose not visited w/ smallest distance from any visited
- needs to remember ① k_v: has v been visited (initially false) ② d_v: minimal edge weight to v (initially ∞)
  ③ p_v: what vertex precedes v (initially unknown)

1) loop over all vertices: find smallest false k_v
2) mark k_v as true
3) loop over all vertices: update false neighbors of v

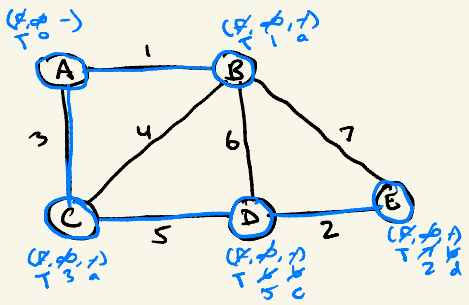
(k_v, d_v, p_v)

e.g.



1. choose random point (a)
2. make d=0 and k=T
3. go through other false vertices and change d, p if you can get there in a smaller d from current index
4. find a false vertex w/ smallest d as your current index. make k T
5. repeat 3, 4 until everything T
6. To get edges, take every point and it's p_v to form edges

e.g.



- Prim's Complexity w/ Linear Search: → better for dense

loop v times: O(V)
1. choose v w/ smallest dv and kv is false O(V)
2. Set kv to true O(1)
3. update dv for all P vertices if better O(V)

$O(V^2)$ total

- Prim's Complexity w/ Heaps (PQ):

repeat until PQ is empty: O(E) ← top and pop edge from PQ to get smallest d
1. choose v w/ smallest dv and kv is false O(log E)
2. Set kv to true O(1)
3. update dv for all P vertices if better O(log E)
→ push the edges you updated to PQ

O(E log E) total

sparse: $E \approx V$ → $O(E \log V) = O(V \log V)$

dense: $E \approx V^2$ → $O(E \log V) = O(V^2 \log V)$

- dense: do nested loops
- sparse: use PQ

- Kruskal's Algorithm:

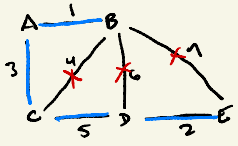
- greedily choose smallest edge weights as long as it doesn't make cycle

1) sort edges: O(E log E) → better for sparse
2) loop through edges in increasing order O(E)
- check for cycle if added O(?) → fast w/ union find

- if j and k already connected, adding edge (j, k) would make a cycle

↳ union-find

e.g.



vertex	A	B	C	D	E
rep	A	~~B~~ A	~~C~~ A	~~D~~ A	~~E~~ A

B, C have same rep so no edge
B, D
B, E

LECTURE 21: Types of Algs

4/2/24

- Brute Force: solves a problem in a direct and obvious way
  - Pro: often easy to implement
  - Cons: does way more work than necessary
  - e.g. Counting Change
    - return a sum in smallest # of coins
    - brute force: check all possible subsets of n coins $O(2^n)$
      Check if sum equals A $O(n)$
      pick subset that minimizes |S| $O(n)$
      best case and worst case: $O(n2^n)$
- Greedy: makes best decisions at each point
  - need to show that locally optimal decisions lead to globally optimal solution
  - pro: much faster solution
  - con: might not give optimal solution
  - e.g. Counting Change:
    - greedy: choose largest possible denomination at each step $O(n)$
      ↳ doesn't work for all denominations

e.g. Sorting: given array, sort it
- brute force: check all permutations $O(n!)$
- greedy algorithm: bubble or selection $O(n^2)$

e.g. Mountain Climbing: find tallest point on mountain
- brute force: look at all possible locations on mountain. Choose highest
- greedy: go until you find local maximum (might not find global)

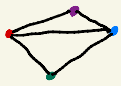
- Divide & Conquer: divide a problem into 2 or more sub-problems (top down)
  - often recursive
  - time complexity often has log n
    ↳ e.g. binary search or quicksort best case
  - pros: efficient, cool recursion
  - cons: recursive calls can be expensive
- Combine & Conquer: start w/ small subproblems and put the small ones together (bottom up)
  - e.g. mergesort
  - subset of divide & conquer
- DP: (overlap)
  - remember smaller subproblems, stores the result, and looks it up
  - pros: can make inefficient (exponential) efficient (poly)
  - e.g. with Fibonacci

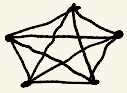
- Types of Problems:
  - Constraint Satisfaction: → Backtracking
    - > just need to satisfy all constraints.
    - > might be >1 solution
    - > only need to find 1 solution or find all solutions
    - e.g. sorting, mazes, spanning tree
  - Optimization Problem: → Branch and Bound
    - > satisfy constraints
    - > min/max a function too
    - > need to develop feasibility set (set of possible solutions)
    - > return best solution
    - e.g. giving change, MST
- Backtracking and Branch and Bound are worst-case algorithms (other than brute-force)
- Backtracking Algorithms:
  - systematically consider all outcomes, but prune searches that don't satisfy constraints
    - ↳ DFS w/ pruning
  - pros: reduces exhaustive search
  - cons: search space still big

e.g. Change Problem

A=12

e.g. 4 Color: Can you color vertices of graph w/ 4 colors s.t. no edge has same colors





not 4-colorable b/c $\exists v \in V$ s.t. $deg(v) \geq 4$

- Planar graph → graph that can be drawn w/ no crossing edges
- can convert map of states to planar graph
- all planar graphs are 4-colorable
- take vertex v_1, consider 4 colors

  take vertex v_2, consider 4 colors

  ⋮

  if one color doesn't work, change it, if all 4 don't work, backtrack to change other vertices until it works.
- keep track of current solution or all solutions

- input: m (# colors), n (# vertices), W (adjmat)

```
Algorithm m_coloring(index i = 0)
   if (i == n)
      print vcolor(0) thru vcolor(n - 1)
      return
   for (color = 0; color < m; color++)
      vcolor[i] = color
      if (promising(i))
         m_coloring(i + 1)
```

Base case (# vertices colored = total # vertices)

color the vertex

T if no constraints violated

F if constraint violated

color next vertex

```
bool promising(index i)
   for (index j = 0; j < i; ++j)
      if (W[i][j] and vcolor[i] == vcolor[j])
         return false

   return true
```

if (edge and color same) ⊏