**Week 1: Graph Basics and undirected graphs**

Undirected graph: collection of vertices V and edges E which connect vertices

Simple graphs: no loops / multiples edges between vertices

Representing graphs on computers:

* Store edges as a list
* Matrix of edge indices (0/1) – for simple graphs
* List of adjacent vertices, per vertex
* Each of the above has differing runtime complexity for listing edges, vertices, etc – we use adjacency list a lot in this course since many algorithms just need to find vertex neighbors (O(deg) with an adjacency list)

Algorithm runtimes: depend on |V|, |E| in general

* Density of graph matters, i.e. ratio of V to E
* In dense graphs, |E| ~= |V|2
* In sparse graphs, |E| ~= |V|
  + Each vertex has only a few edges – very common in practice

**Exploring graphs:**

Notion of vertices that are reachable from a given vertex

Algorithm: explore every edge leaving every vertex that is reachable from a given vertex

* Boolean variable visited(v)
* Keep list of vertices with edges left to check
* Explore new edges in depth first manner
  + Only backtrack when we hit a dead end
* Exlore(v):
  + Visited(v) = true
  + For (v,w) in E:
    - If not visited(w):
      * Explore(w)
* The above is efficient if we have an adjacency list
* Algorithm will find all nodes and also return a list of edges
* Theorem that it explores only things reachable from v, and explores all such vertices

Algorithm: find all vertices of a graph, not just those reachable from starting vertex

* For all v in V:
  + If not visited(v):
    - Explore(v)
* Similar to above but we check all vertices
* No vertex gets explored more than once
* Have to check all neighbors of a vertex: O(|E|)
* Total work is linear: O(|V|+|E|)

**Connectivity**

Which vertices in G are reachable from given vertices?

Can partition G into connected components (“islands”)

Algorithm to find connected components:

* Explore(v) finds all connected components: slight modification to DFS
* Objective: label connected components
* Add indicator variable – in each outer loop (DFS), increment it by one; in the inner recursive calls (Explore) it stays the same

**Pre-visit and post-visit orderings**

Recall, DFS doesn’t return anything, just marks vertices as visited – need to keep track of other data to be useful

Add some previsit, postvisit functions (envelope the recursive call)

* E.g. record the previsit time and postvisit time to show traversal order

Lemma: for any u,v, the pre/post intervals are either nested or disjoint (\*not\* interleaved i.e. overlapping but not nested)

Proof: If u is visited before v, you either:

* Find v while exploring u (the subroutine case – they are connected)
* Find v after exploring u – have terminated exploring u (they are not connected)

**Week 2: Decomposition of graphs**

**Directed acyclic graphs**

Direction along edge matters, e.g. one-way roads, program build dependencies

Can still run DFS, only change is to only follow directed edges

In some cases, we can write linear ordering of dependencies: this is not possible if the graph has a cycle

A DAG (directed acyclic graph) has no cycles

* DAG is a necessary condition for linear ordering of a graph
* Is it sufficient? TBD…

**Topological sort**

Theorem: any DAG can be linearly ordered (DAG is N&S condition for linear ordering)

Definitions:

* Source: has no incoming edges
* Sink: has no outgoing edges

Basic idea for producing linear ordering:

* Find sink, put at end
* Remove from graph
* Repeat

As nodes are removed, remaining nodes become sinks

We can always find a sink, unless we have a cycle (contradicts DAG)

Runtime complexity of LinearOrder is O(|V|2) since there are |V| paths which can be up to |V| long

* Can improve by only backing up as far as necessary instead of starting from the same vertex each time and following the whole path
* The algorithm itself is a depth first search! With vertices listed by reverse post-order (since post\_order returns a true sink first and then backwards from there)
* Topological sort:
  + DFS(G)
  + Sort vertices by reverse post-order

Theorem: if G is a DAG with an edge u to v, post(u) > post(v) (i.e. u actually comes before v)

Proof:

* Cases:
  + explore v before u
    - Cannot reach u from v, so must finish v before finding u, so post(u) > post(v)
  + Explore v during u
    - V is found during subroutine of explore(u), so post(u) > post(v)
  + Explore v after u (not possible since there exists an edge between them)

**Strongly connected components**

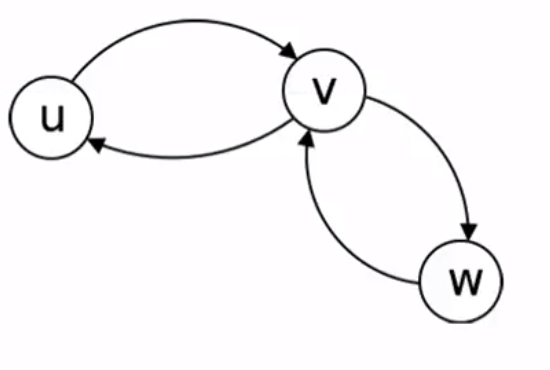
In directed graphs, even if there are no islands, you can get stuck in one area such that some other vertices are now unreachable

Notions of reachability:

* Connected by edges in any direction
* One vertex reachable from another using correct edge directions
* Both vertices reachable from each other using correct edge directions 🡨strongly connected

Theorem: a directed graph can be partitioned into strongly connected components if only if they are connected to each other (i.e. both reachable from each others)

Proof: strong connectivity is an equivalence relation (u<->v and v<->w implies u<->w)



Can draw metagraphs that link strongly connected components

Theorem: the metagraph of any graph G is a DAG

Proof: If not, there must be a cycle and so everything is strongly connected, so there would only be one SCC in the metagraph (which would be a DAG)

**Computing strongly connected components**

For a DAG, find the metagraph of strongly connected components

EasySCC:

* For each v, find all u reachable from v – V^2
* For each v, find all u reachable from v that can also reach v – V\*E

Runtime complexity is O(|V|2+|V||E|)

Can improve this… if v is a sink, explore(v) finds vertices reachable from v, which is exactly the SCC of v – don’t need to call explore twice

Need a way to find sink in SCC

Theorem: if SCCs C, C’; if C🡪C’, post(C’) < post(C)

Proof in slides (fairly simple)

Idea: largest post\_order value is a \*source\* component – and there is a notion of reverse graph; luckily SCCs of graph and reverse graph are the same

Idea: run DFS on the reverse graph 🡪 largest post(v) is a sink in the original graph

Algorithm:

* Run DFS(GR)
* Let v have largest post number
* Explore(v)
* Vertices found are first SCC: remove
* Repeat

Still a little inefficient since we run DFS repeatedly

But, due to above theorem, can just use post-order values from the single DFS

SCCs(G):

* Run DFS(GR)
* For v in reverse post order
  + If not visited(v)
    - Explore v
    - Mark visited vertices as new SCC

Runtime is O(|V|+|E|) since we basically did a DFS on GR then on G

Next unit: shortest paths in graphs

**Module 3: Shortest paths in graphs**

Distance(v,w) = shortest path length between them

If no path between vertices in a directed graph, distance(v,w) = infinity

IN general, find all distances from a node S at once (not just a single pair)

Distance layers: nodes 0, 1, 2, etc nodes away from node S

* Undirected case: No paths that skip distance layers
* Edges can exist within a layer or from one layer to an adjacent layer
* Directed case: there CAN be edges from layers to previous layers, but no layers from one node to a node that is at least 2 layers further from S (no jumping down, but you could jump back due to directed property)

Breadth-first search:

Goal: find distance form origin node to all other vertices in the graph

Idea: DFS is good for exploring connected components (size relations?), BFS is good for understanding distance relations

BFS: explore layer 0, layer 1, layer 2, etc – processing implies exploring all outgoing edges

* If \*undirected\*, there are more edges reprocessed at each stage (O(E^2) vs O(E) maybe?)
* Ordering: for a directed graph, consider a queue of nodes to be processed (FIFO)

BFS(G,S)

For all u in V

Set dist(u) = inf

Dist(S) = 0

Q = {S} (queue)

While Q is not empty

U = Dequeue(Q)

For all (u,v) in E

If dist(v) = inf # dist=inf implies hasn’t been discovered yet (could also use #.number of nodes/edges + 1)

Enqueue(Q,v)

Dist(v) = dist(u) + 1

* Note: nodes never get added to queue more than once, so at most V elements in queue
* Algorithm terminates and in at most V outer loops
* Running time: O(|E|+|V|)
  + Each vertex enqueued at most once
  + Each ends examined either once or twice, directed or undirected

Proof of correctness:

* Reachable nodes have finite distance from S and are discovered
  + Contradiction: if some reachable node is not discovered in BFS, take shortest path from S to u; if vk is before u, u will be discovered when processing vk – so contradiction that it is not discovered
* Unreachable node is not discovered
  + Contradiction – some unreachable node is discovered, means it was discovered while processing vk, which is reachable, so there is a path from S to vk, and also a path from S to u
* Order Lemma: by the time a node at distance d is dequeued, all nodes of distance at most d have already been \*enqueued\*
  + Contradiction: if u at dist=d begins processing while v at dist=d’ (d’ <= d) has not been discovered
    - Then, u was discovered while processing u’ at dist >= (d-1)
    - Also, node v has edge from node v’ with dist >= (d’-1)
    - So d’ – 1 <= d-1
    - So v’ was discovered before u’ was dequeued, since was assume order lemma was correct up until the u and v contradiction
    - So v’ was enqueued before u was enqueued
    - So v’ was dequeued before u was dequeued, so v was discovered by the time u is dequeued
    - = contradiction
* Correct distances: proof by induction
  + When S is discovered, d(S,S) = 0; trivially correct
  + Induction on d(S,u): suppose correct for all nodes <= k from origin
  + Take v at dist = k+1, where all nodes at k are found correctly
  + V was discovered while processing u; d(S,v) <= d(S,u) + 1 🡪d(s,u) >= k
  + Since v is discovered after u is dequeued, and with order lemma, d(S,u) < d(S,v) = k+1;
  + Because we know d(S,v) = k+1 per our assumption, and d(S,u) is >= k but strictly less than d(S,v)
  + Therefore, d(S,u) = k; and the dist assignment = k+1 is correct
  + And then induction on k
* Queue property:
  + AT a given time if the first node in queue is dist=d, then all nodes in queue have either d or d+1, and all d are before all d+1
  + Proof:
    - We know all nodes of d are enqueued before any d is dequeued, so all go before any d+1 since those come from exploring/dequeing the d’s

Shortest path tree

* Make a tree that depicts how a node was discovered from another in the BFS
* Lemma: shortest path tree is a tree (does not contain cycles )
  + Proof by contradiction: suppose there is a cycle
  + There is at most one outgoing edge from each node since we could only have discovered it in one way; as we go along the path, distance to S decreases by 1 since the nodes point to the node \*from which they were discovered\*
  + Cycle 🡪 d <= d – k which cannot happen for k > 1: contradiction
* BFS(G,S): as before but add a line after dist assignment that stores prev[v] = u (previous node from which v was discovered)
* We now know all nodes’ parents in shortest path tree

Reconstructing the shortest path from the tree, (from S🡪u):

* Just retrace via prev:
* While u != S
  + Result.append(u)
  + U = prev(u)
* Return reverse(Result)

Complexity is just O(dist(S,u)) << O(V) usually

Notes from assignment:

Best way to implement BFS is using deque (double ended queue) from collections – can append and pop/left and right

* Deque is implemented as a doubly linked list: good performance for operating on either end of the queue but poor (O(N)) for accessing a random element in the list
* Can of course be used as either a queue or a stack since either end works equally well

Bipartite graphs: can divide into two classes of nodes such that no node connects to a node of the same class

**Module 4: Paths in Graphs 2**

Fastest route problem: graphs with weighted edges

Naïve algorithm background:

* Any subpath of an optimal path is also optimal; otherwise if the subpath was not optimal there would be a better optimal path
* Edge relaxation: dist[v] is an upper bound on the actual distance from S to v \*during the algorithm\* (i.e. dist gets updated)
* Idea: during algorithm, we check whether going through some node u will improve dist; and check all such edges along nodes in adjacency matrix starting from S

Naïve Algorithm:

For all u in V:

Dist[u] = inf

Prev[u] = 0

dist[S] = 0

Do:

Relax all edges

While at least one dist changes

Proof of correctness

Contradiction: no edge can be relaxed and some v exists such that dist[v] > d(S,v);

* There must be at least two previous nodes on this path, call them u and p, with d(S,p) = dist[p], and d(S,u) = d(S,p) + w(p,u) { p is not broken }
* Say u is also broken, dist[u] > d(S,u) = dist[p] + w(p,u)
* But the rhs above is the inequality checked to see if edge can be relaxed, so there is a contradiction to “no edge can be relaxed”
* Qed

Instead, use Dijkstra’s algorithm… intuition:

* Assume no negative edge weights
* Start at S and relax all edges
* When one dist is fixed, relax all its outgoing edges and update dist appropriately
* Keep track of nodes whose optimal dist is already known for sure (all other undiscovered possible paths are already longer than current dist)

Dijsktras(G,S):

Initalized all dist[v] = inf, prev[u] = null

Dist[S] = 0

H = Queue(V) :: dist values as keys

While H is not empty:

u = pop\_min(H)

For all (u,v) in E

If dist[v] > dist[u] + w(u,v)

Dist[v] = dist[u] + w(u,v)

Prev[v] = u

ChangePriority(H, v, dist[v])

The last step says, since we improved the dist value of node v, we need to update its priority in our priority queue

Proof of correctness

When a node u is selected via ExtractMin (pop\_min), dist[u] = d(S,u)

Contradiction: Say there is node of unknown distance which is selected via ExtractMin, and assume its dist value is wrong;

* recall dist value is upper bound on correct d value
* This implies its dist value is strictly greater than correct distance
* The path to u must go from known region to unknown region via some nonnegative path; if it goes through some other node first, since ExtractMin picked this node, its length must be strictly less than the current dist estimate
* So the assumption that all edges from the known region were relaxed must have been violated to arrive at this situation
* Basically, the optimal subpath idea, in a way

Complexity of Dijkstra’s

T(make queue) + |V|\*T(ExtractMin) + |E|\*T(Change priority)

T() implies time is dependent on what data structure you use

Priority queue implementations:

* Arrays:
  + One array to store dist values, one array for Boolean flag to say if it’s still in queue
  + ExtractMin: O(|V|^2): first check if it’s in the data structure, then find distance, V times
  + ChangePriority: constant time
  + Overall O(|V|^2)
* Binary heap
  + Build is V
  + ExtractMin = |V|(log|V|)
  + Change priority: cheat and insert a new element to the heap whenever we want to change the dist value, instead of updating the element
    - Makes the heap bigger, but at most the total number of elements is number of edges since this is the max number of ChangePriority operations
    - And E <= V^2
  + Overall: O( (|V|+|E|)\* log(|V|))
  + Much better than V^2 when E << V^2
* Recall from data structures course:
  + Binary heaps: Parent greater than children
  + want to keep tree shallow (height = log(n))
  + All operations in log(n) time
  + Can store as array where parent/children are stored in positions relative to an element
* Note from hw: binary heap in python: heapq
  + Heapq is a min heap
  + Entry key is the minimized value, so any other identifier needs to be later in the tuple

Graphs with negative weights – currency arbitrage example

* In general want to calculated maximum product over some path length
* To convert products to sums, take log(weights)
  + Max\_pdt(r\_j) <-> max\_sum(log(r\_j))
  + Requires positive weights in the original graph
  + Or minimize sum of -log
    - Thus we have a shortest path problem
* Cannot just apply Dijkstra’s as is, since there are negative weights and we could improve by going along these (but algorithm won’t find it due to assumption that if we are exploring a node, it’s distance is correct)
* Any graphs with negative weight cycles have (some) dist values of -infinity for those nodes that can access the cycle

Bellman-Ford

* Shortest paths in graphs with negative weights: Bellman-Ford
  + Almost the same as “naïve algorithm”
  + Relax edges while dist changes
* Algorithm: (assume no negative weight cycles)

For all u in V

Dist[u] = inf

Prev[u] = nil

Dist[S] = 0

Repeat |V|-1 times:

For all (u,v) in adj

Relax(u,v)

* Note: we don’t need to repeat V-1 times, could just do it until no relaxation is possible – this works and is faster with no negative weight cycles
* Pseudocode written as above because easier to prove correctness; but could stop once dist does not change
* Complexity: O(|V||E|) – longer than Dijkstra’s with binary heap implementation
* Often, only one iteration over the edges is needed; on second there is no improvement

Proof of correctness for Bellman-Ford

Lemma: After k iterations of relaxations, for any node u, dist[u] is mallest length of a path from S to u that contain at most k edges

After one iteration, all dist values are the best possible path within 0-1 edges

* Proof by induction with base case 0 iterations
* Induction: before k+1 iteration, it held for the k iteration case; at k+1, the path comes in from an incoming edge which was optimal up to k edge; and the relaxation procedure ensures the next step is also optimal up to k+1

Corollary: if no negative weight cycles, Bellman-Ford correctly finds all distances from S

* Since there are no cycles that would continue to improve path as k increases beyond |V|

Corollary: if no negative weight cycle reachable from S and reachable from u exists, Bellman-Ford correctly finds dist[u] = d(S,u)

Negative weight cycles

Lemma: Negative weight cycle exists **iff** there is a |V|th additional iteration of Bellman-Ford, and some edge is relaxed (Dist is updated)

Proof:

By contradiction: If no negative cycles, then all shortest paths from S contain at most |V|-1 edges; so no dist value can be updated; which contradicts the assumption that an edge can be relaxed

In the other direction; say there is a negative weight cycle but no dist value can be improved on |V|th iteration: look at inequalities that imply no relaxation 🡪 sum of weights >= 0 follows from this, which contradicts the existence of a negative weight cycle

Finding a negative cycle: run |V| iterations and save the node v that was relaxed on the last iteration, since it is definitely reachable from a negative cycle (with at most |V| steps)

* Then start from x 🡨 v and follow the link x = prev[x] for |V| times and we will definitely be on the cycle
* Do this until back to the original node on the cycle

Negative weight cycle doesn’t always imply an infinite arbitrage type situation: need to be able to reach the negative cycle from S and to exit the negative cycle to u

Infinite arbitrage scenarios

Lemma: Infinite arbitrage possible **iff** u is reachable from some node w for which dist[w] decreases on iteration V of Bellman-Ford (basically the above)

Proof:

🡨 : dist[w] decreased on iteration V 🡪 w is reachable from a negative weight cycle, which in turn is reachable from S (since we are doing bellman ford starting at S…); thus u is also reachable from negative cycle and infinite arbitrage is possible

-> Let L be length of shortest path to u with at most V-1 edges; after V-1 edges, dist[u] is = L; infinite arbitrage implies a path shorter than L; this requires dist[u] to be decreased on some iteration k>= V

Note: if at some point, an edge (x,y) was not relaxed and dist[x] did not decrease on iteration i, the edge (x,y) will also not be relaxed on iteration i+1

* Since dist[x] is the same, no benefit in relaxing (x,y)
* Only nodes reachable from those relaxed on previous iterations can be relaxed at the current iteration
* In the 🡪 above, dist[u] being decreased on k <= V 🡪 u is reachable from a node relaxed on the V-th iteration

To **detect** infinite arbitrage:

* Do |V| iterations of Bellman-Ford, save all the nodes relaxed on V-th iterations and call set A
* Put all nodes in A in a queue Q
* Do breadth-first search with Q and find all nodes reachable from A
* All those nodes and only those can have infinite arbitrage

To **reconstruct** infinite arbitrage:

* Remember the parent of a visited node during BFS
* Then we can reconstruct the path to u from some node w
* Go back to w to find the negative cycle
* Use this to achieve infinite arbitrage from S to u

**Module 5: Minimum Spanning Trees**

Creating networks such that all nodes are reachable from each other with minimum total cost (sum of all edges)

Minimum spanning tree (MST): connected, undirected graph with positive edge weights where some subset of edges E’ with minimum total length exists such that G(V,E’) is connected

Properties of trees:

* Connected, acyclic undirected graph
* May be rooted or unrooted (any unrooted can be expressed as rooted tree just by picking root)
* A tree with n vertices has n-1 edges (since adding more edges will create a cycle)
* Any **connected** undirected graph with |V|-1 edges is necessarily a tree
* An undirected graph is a tree iff there is a unique path between any pair of vertices (obviously…)

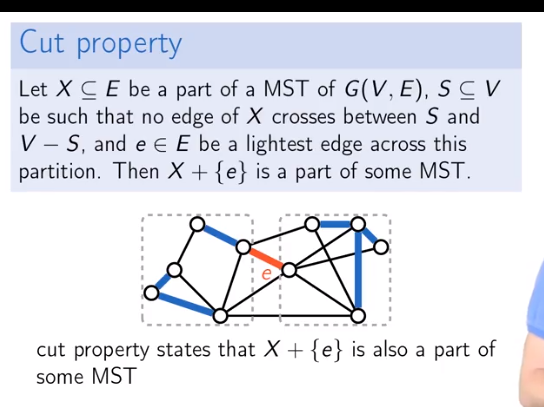
“Any time there are multiples paths between nodes, there is some unnecessary additional edge cost”

Greedy algorithms for MST

Kruskal’s algorithm: repeatedly add next lightest edge if it doesn’t present cycle (ie not necessarily growing a single tree until they are all connected at the end)

Prim’s algorithm: repeatedly add new vertex to current tree by lightest overall edge (ie continuously growing a single tree)

Cut property (used for proof of correctness of above):



Sketch of proof: starting from the above

Note that the MST X is not nec the same of the new MST including e

If same MST, nothing to prove

If new MST is not X, then adding e produces a cycle; adding e and subtracting that other edge e’ will create a MST 🡪 and since e is the lightest weight edge (e <= e’), the resulting tree is surely a MST

Kruskal’s algorithm

Correctness: at any time, we have a forest aka collection of MSTs

Adding an edge e that connects two trees creates a new MST via cut property

Implementation details:

* Use disjoint sets data structure
* Each vertex is initially in a separate set
* To check for cycles, check if in same set and skip if so (would produce cycle)
* If not, need to merge sets once they are joined
* Do for all edges (naïve implementation)

Time complexity:

* Sort edges: O(|E|log|V|)
* Process edges: 2\*|E|\*T(Find) + |V|\*T(Union) = O(|E|log|V|)
  + Since Union with union by rank heuristic: log|V|
  + So Process ub is O(|E|log|V|) as well
  + with path compression as well, process edges ub is O(|E|log\*|V|), ie nearly linear, for second step but does not improve ub on total running time due to edge sort step
  + But this helps a lot if the edges arrive in sorted order!
* Should review all of the above wrt disjoint set data structure …

Prim’s algorithm

X is always a tree which grows by one edge at each iteration – similar to Dijkstra in a way

Uses priority queue instead of disjoint set data structure

Need to update priorities as we go and there become shorter edge lengths that may also connect them to the current tree (if multiple)

Time complexity is similar to dijsktra:

* |V|\*T(ExtractMin) + |E|\*T(ChangePriority)
* O(|V|^2) for array-based implementation: easy to change priority but hard to find the min each time – |V| calls to ExtractMin which has O(|V|) while ChangePriority is O(1)
* With a binary heap, all operations are O(log|V|)
  + Total is O(|E|log|V|)
* Array based better for dense graph, binary heap based better for sparse graph

Also, Kruskal has same time complexity as Prim with binary heap, so could say Kruskal is better for sparse graph while Prim with array-based implementation is better for dense graph