Graphs

17/4/19

- Defined by $g = (V, E)$, i.e a set of vertices and edges
- If directed, edges are $V_1 \rightarrow V_2$
- If undirected, edges are $V_1 \longleftrightarrow V_2$
- A path is a sequence of vertices connected by edges: $V_1 \rightarrow V_2 \rightarrow \dots \rightarrow V_K$ or $V_1 \leftrightarrow V_2 \leftrightarrow \dots \leftrightarrow V_K$
- An undirected graph is connected if there is a path between any two vertices
  - ↳ a forest is an undirected acyclic graph
  - ↳ a tree is a connected forest.
- Graphs can be represented as:
  (1) adjacency lists, i.e an array of {linked lists containing neighbours}
  (2) adjacency matrix. Requires $O(V^2)$ space.
  ↳ choose depending on sparsity. If E/V^2 is large, use matrix.

Depth-first search

- Visits all vertices reachable from a starting vertex, by recursively searching neighbours (but marking visited to prevent loops).
- Can be implemented with a stack

```
def dfs(g, s):
    for v in g.vertices:          } O(V)
        v.visited = False
    toexplore = Stack(s)
    s.visited = True
    while not toexplore.empty():  <- no vertex can be pushed more than once ∴ O(V)
        v = toexplore.pop()
        for w in v.neighbours:                    }
            if not w visited:                     }
                toexplore.push(w)                 } run once per edge ∴ O(E).
                w.visited = True                  }
```

- Thus DFS is $O(V+E)$, using aggregate analysis.

- The code for a breadth-first search is identical, except a queue is used rather than a stack.

Dijkstra's Algorithm

- For a graph whose edges have positive weights, Dijkstra's algo allows us to find the shortest ~~min weight~~ path.
- Similar to BFS, except we use a priority queue to store frontier vertices, and greedily choose the nearest one.
  - ↳ if we see a vertex that has already been visited, we update its distance and its position in the PQ.
- Once an item is popped, its distance is the true minimum distance and it never gets added back to the PQ
- So each vertex called popmin(), and we had to push/decreasekey for each edge.
  - ↳ using Fibonacci heap: popmin() is $O(\lg n)$, push/decreasekey $O(1)$

$$\therefore \text{Dijkstra runtime is } \boxed{O(E + V\log V)}$$

Proof of correctness (by contradiction):

- Let v be the first vertex for which after popmin(), v.distance is not the true shortest distance.
- Let a shortest path from s to v be $s = v_1 \to \dots \to v_k = v$
- Let u_i be the first vertex that has not been popped.

$$\begin{aligned}
\text{distance}(s \text{ to } v) &< v.\text{distance} \\
&\leq v_i.\text{distance} \\
&\leq v_{i-1}.\text{distance} + \text{cost}(v_{i-1} \to v_i) \\
&= \text{distance}(s \text{ to } v_{i-1}) + \text{cost}(v_{i-1} \to v_i) \\
&\leq \text{distance}(s \text{ to } v).
\end{aligned}$$

Contradiction $\therefore$ v.distance is the true shortest distance.

- Thus it cannot be pushed back into the PQ
- And thus the algo must terminate

The Bellman equation

- Let vertices represent states and edges represent actions. The goal is to find the best sequence of actions.
- Uses different terminology to Dijkstra (more general)
  - cost → weight
  - ditance → minweight
  - shortest path → minimal weight path.
- e.g finding the best FX rate, with weight = − ln (exchange rate).
- Same as dynamic programming:

$$W_{ij} = \begin{cases} 0, & i = j \\ \text{weight}(i \to j), & \text{if there is an edge} \\ \infty & \text{otherwise.} \end{cases}$$

← minweight action to go from state i to j

$M_{ij}^{(l)}$ is the minimal weight path from i to j in l steps.

$$M_{ij}^{(1)} = W_{ij} \qquad M_{ij}^{(l)} = \min_k \left\{ W_{ik} + M_{kj}^{(l-1)} \right\}$$

NB: Assumes no negative weight cycles.

i.e make an intermediate jump then choose best path.

- Can be reformulated as 'matrix multiplication':

$$M_{ij}^{(l)} = (W_{ij} + M_{1j}^{(l-1)}) \wedge (W_{i2} + M_{2j}^{(l-1)}) \wedge \dots \wedge (W_{in} + M_{nj}^{(l-1)})$$

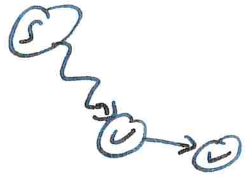
where $x \wedge y \equiv \min(x,y)$, $n \equiv |V|$.

↳ the minimal weight path must have $< n$ edges because we assumed no negative weight cycles

↳ requires $\log V$ matrix multiplications, ∴ $O(V^3 \log V)$

- This is a brute-force algo and often cannot be used.

Bellman-Ford



- If we have a path from s to u and edge $u \to v$, we can update v.minweight as follows:

  if v.minweight > u.minweight + weight($u \to v$):
  v.minweight = u.minweight + weight($u \to v$). } 'relaxing' $u \to v$

- Bellman-Ford uses this to find the minimal weight path, by relaxing each edge of the graph in $V-1$ rounds
  - ↳ advantage over Dijkstra is that it works for graphs with negative weights, and can detect negative cycles.

    negative cycle $\Longleftrightarrow$ graph changes in V^{th} relaxation round.
  - ↳ runtime $O(VE)$

Proof of correctness (induction):

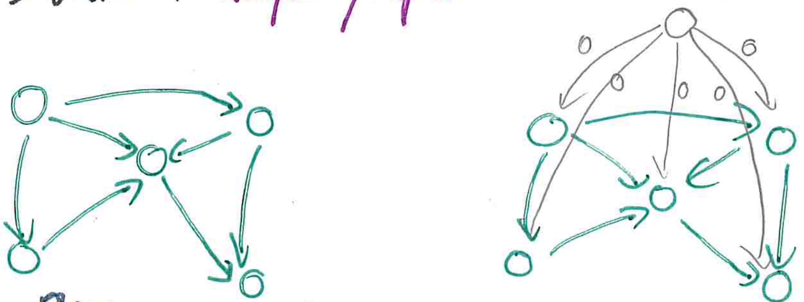
- Consider the minimal-weight path from s to some v:

  $$s = v_0 \to v_1 \to \dots \to v_k = v$$
- Initially, v_0. minweight $= 0$
- After one relaxation round, v_1.minweight is correct because we are on a minimal-weight path. Proceed by induction.
- At most $|V|-1$ edges, hence at most $|V|-1$ iterations.
- If the graph has a negative-weight cycle, an exception will be thrown.

Johnson's algorithm

- Used to find ~~all~~ shortest paths between all pairs of vertices
  - ↳ needed to calculate betweeness centrality - number of ~~pa~~ shortest paths that use a given edge.
- We could run Dijkstra for each vertex, in $O(VE + V^2 \log V)$, but this fails for negative weights.
- Using Bellman-Ford for each vertex would be $O(V^2 E)$.
- Johnson's algo uses BF then Dijkstra
  - ↳ build a helper graph with a new node s, with $d_v \equiv w(s \to v)$

~~tra~~ min weight from $s \to v$



  - ↳ ~~BF~~ run BF on this graph to look for neg. weight cycles.
  - ↳ recreate the original graph with weights: $w'(u \to v) = d_u + w(u \to v) - d_v$
    - ↳ $w'(u \to v) \geq 0$ by relaxation
  - ↳ run Dijkstra on every vertex.
  - ↳ total runtime $O(VE + V^2 \log V)$

Proof of correctness:

- The minweight path using w' instead of w is the same. Consider some path $v_0 \to \dots v_k$

~~original~~ new weights: $w'(v_0 \to v_1) + w'(v_1 \to v_2) + \dots + w'(v_{k-1} \to v_k)$

$$= d_{v_0} + w(v_0 \to v_1) - d_{v_1} + d_{v_1} + w(v_1 \to v_2) - d_{v_2} + \dots + w(v_{k-1} \to v_k) - d_{v_k}$$

i.e. telescoping sum

$\therefore$ weight of path using w' = weight of path using old w $- d_{v_0} + d_{v_k}$

Topological sort

- A directed acyclic graph (DAG) can always be used to represent a total ordering, such that if $v_1 \to v_2$, then v_1 is before v_2 in the total order.
- The algorithm is based on a recursive DFS, adding v to the total order when visit(v) returns.
- We can use a breakpoint proof of correctness:
  - nodes are initially white
  - grey once visited
  - black once it has been added to the total order.

- Consider an edge $v_1 \to v_2$, and we have just entered visit(v_1), i.e v_1 is grey.
  - if v_2 is black, it is already in the list, so prepending v_1 will be correct
  - if v_2 is white, visit(v_2) will be called, so v_2 will be prepended before v_1. Thus v_1 will be before v_2
  - if v_2 is grey, it means we are inside visit(v_2) when we called visit(v_1), implying $v_2 \to v_1$. But we are in a DAG, so this contradicts $v_1 \to v_2$.

$\hookrightarrow$ v_2 cannot be grey.

Minimum Spanning trees

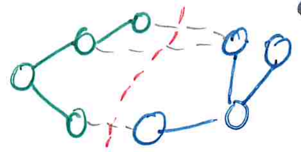
- Given a connected weighted undirected graph, the MST connects all vertices and has minimal weight.

Prim's algorithm

- Greedily builds ~~an~~ MST by choosing the lowest weight connector to a frontier vertex
- ~~W~~ Very similar to Dijkstra, except:
  - need to keep track of the tree
  - we want 'distance from trees' instead of 'distance from start'.
- Runtime is the same as Dijkstra, i.e $O(E + V \log V)$.

Kruskal's algorithm

- Builds an MST by agglomerating smaller subtrees greedily.
- Edges need to be sorted by weight: runtime is $O(E \log E)$
- Despite worse runtime than Prim's, Kruskal has useful intermediate states and can be used to build clusters.
- Can be proved by considering two minimal spanning subtrees



← Kruskal picks the min weight across the cut, so if $t1$ and $t2$ are MSTs within their partition, the result will also be an MST.