Algorithms Notes

Simon Xiang

Lecture notes for the Spring 2022 section of Algorithms and Complexity (CS 331) at UT Austin, taught by Dr. Price. These notes were taken live in class (and so they may contain many errors). Source files: https://git.simonxiang.xyz/math_notes/files.html

Contents

1 Graph searching

Consider a graph $G = (E, V)$, where E is a set of pairs of elements of V. Then $E \leq \binom{V}{2}$ if G is directed, and $E \leq V(V-1)$ if G is directed. Adjacency matrix, adjacency list (how to store). linked list. list comparing complexities. hash table. binary search tree. help i didn't take data structures. i also cannot read the board.

look at the complexity list in the textbook.

Reachability: given $G = (V, E)$ with source s, the goal is to find all the vertices u reachable from s.

Let visited equal {}. Depth first search:

```
def DFS(v)
    if v in visited return
    visited.add(v)
    for w in v.adj:
        DFS(w)
```

Now we do Breadth first search (BFS). It visits vertices in order of distance from s. prinn's minimum spanning tree algorithm.

2 graph reductions

The big idea is to turn things into graphs then run BFS on them.

Example 2.1. snakes and ladder, with max k and n^2 board size.

```
#creating the graph G
for i in [n^2]
    for j in [k]
        if (i+j,y) for j in [k] is in (set of snakes/ladders)
            create edge (i,y)
        else
            create edge (i,i+j)

#run BFS on the graph
BFS(G)
```

Example 2.2. mnongolian puzzle, swapping tokens. Let A be an $n \times n$ matrix. make the state of the graph as a vertex (two points $A_{ij} = n$, $A_{k\ell} = m$). Each vertex is a transformation $((A_{ij}, A_{k\ell}), (A_{(i+m,j)}, A_{(k+n,\ell)}))$ etc depending on which moves are valid (ie $A_{(i+m,j)}, A_{k+n,\ell})$ satisfies $i+m, j, k+n, \ell \in [n]$). Then run BFS on G (graph made according to the above rules) to find the shortest path from (A_{in}, A_{ni}) to (A_{ni}, A_{in}).

3 DFS and topological sort

dags have a topological ordering.

4 A* search

Recall: Dijkstra.

```
Dijkstra(G,S):
    parent, dist = {}, {}
    Q = priority queue([(0,S, None)])
    while Q:
        d, u, p = Q.pop()
        if u in dist:
            continue
        parent(u)=p, dist[u]=d

        for v in u.adj:
            Q.append((dist[u]+w(u -> v),v,u))
        return dist, parent
```

append and pop gives runtime of $O(E \log V)$ or $O(E + V \log V)$. doesn't work for negative edges (simple counterexample). modify the line `if u in dist` to `if u in dist and d = dist[u]`. this is correct, but no longer visits each vertex once, visit them an exponential amount of times. so this becomes a lot worse than bellman ford. but it's correct.

example of exponential time: "spring graph". one half is minimal positive, while the bottom half is big positive then big negative. this takes $n!$ iterations (goes to the end, loops back. then starts from second and goes all the way to the end, loops back). so with k negative edges, this is a 2^k factor, but with only one it's alright. correct and slow is better than fast and wrong.

say we have a destination in mind, `Dijkstra(G,S,t)`. without negative weights, once we pop t off the queue. so after the line `parent[u]=p, dist[u]=d`, add the line `if u==t, break`. basically, stop after visiting t. well what about negatives.

4.1 A* search

dijkstra visits a lot of extra things (eg consider map of the US, get from austin to NY/san fran. visits almost all the continental US in the meantime). how to fix this? have some sort of measure of where the destination is. this leads to $A*$ **search**. we use a **heuristic** or **potential** h. $h(u)$ is some measure of distance u to t. Idea: visit vertices in order of increasing $\text{dist}(s,u) + h(u)$. the heuristic prefers to visit vertices closer to our destination.

alternative view: running Dijkstra shortest paths on a *reweighted graph* G with weight $w_h(u \to v) := w(u \to v) + h(v) - h(u)$. think of it as taking each edge and "adjusting" the weight. let $\text{length}_h(u_1 \to u_2 \to \cdots \to u_n) = \text{length}(u_1 \to u_2 \to \cdots \to u_n) + h(u_k) - h(u_1)$ (since the other terms will all cancel). this length doesn't depend on path, only depends on start and end. therefore shortest path in w_h is precisely the shortest path in w. so $\text{dist}_h(s,u) = \text{dist}(s,u) + h(u) - h(s)$. running Dijkstra on this reweighted graph sorts by distance under reweighted graph, sorting by $\text{dist}(s,u)+h(u)-h(s)$. but $h(s)$ is the same. therefore Dijkstra on $w)h$ visits u in order of increasing $\text{dist}(s,u) + h(u)$. so these two ideas are the same.

when does this work? we could choose any h and this equivalence is true. what do we need from h? for correctness, when we visit t we should be done. one condition is **admissibility**, that is, h is said to be admissible if $h(t) = 0$ and $h(u) \le \text{dist}(u,t)$ for every u. if this holds, then when we visit t, every unvisited u has $\text{dist}(s,u)+h(u) \ge \text{dist}(s,t) \implies \text{dist}(s,u) + \text{dist}(u,t) \ge \text{dist}(s,t)$. this implies the shortest s leads to a path. so A^* returns the correct shortest path.

well correctness is good, but we want it to be *fast*. how do we make this polynomial time? we don't want it to be worse than regular dijkstra, ie not worse than $E(\log V)$ time. so we require the heuristic to be non-negative, since this is what causes dijkstra to be slow. the name is **consistent**. that is, $w_h(u \to v) \ge 0$ for all u, v. this is equivalent to saying that $w(u \to v) + h(v) - h(u) \geq 0 \iff h(u) - h(v) \leq w(u \to v)$. so if we are consistent and $h(t) = 0$, then we are admissible. why? we know that $\text{dist}_h(u,t) \geq 0$ by consistency. but $\text{dist}_h(u,t) = \text{dist}(u,t) - h(u)$ (removing $h(t)$), which implies $h(u) \leq \text{dist}(u,t)$. So A^* runs in $O(E + V \log V)$ time. this shows that A^* may not necessarily be faster than Dijkstra, but it is not worse.

next pset; we will run this on some examples. popular example: set $h(u) = \|u - t\|$, ie $w(u \to v)\|u - v\|_2$. so $h(u) - h(v) = \|u - t\| - \|v - t\| \leq \|u - v\| = w(u \to v)$ by the triangle inequality. so h is consistent and admissible. the general picture is that the DP may make you consider going through houston to LA from Austin, but not through NY. ie instead of considering most of the continental US in concentric circles, we consider a smaller radius around the "general direction" to LA.

there might be other ways, ie getting off the highway at phoenix and looking around for a direct highway through local roads. the algorithm is conservative (and consistent), always considers going through a small street. with this heuristic it's hard to change the "level of conservativeness" or "aggression" of the algorithm. in the next pset we will look at the ALT heuristic, rather than Euclidian distance it runs a few shortest paths. this never gets you off the highway in some small town.

4.2 All-pairs shortest paths

Last week we were talking about SSSP (single source shortest paths), with Dijkstra and bellman ford. the question now is, what about all pairs shortest paths (APSP)? for every pair of vertices, what's the shortest way to get there. one idea is to run SSSP for all $s \in V$. this gets $O(VE + V^2 \log V)$ if non-negative, and $O(V^2 E)$ if negative.

another cute/short algorithm is the following. this is called **Floyd-Warshall** (APSP algo). we start with a distance matrix

$$\text{dist}(u,v) = \begin{cases} w(u \to v) & \text{if } u \to v \in E \\ \infty & \text{otherwise} \end{cases}$$

Then the algorithm is as follows:

```
for u in V:
    for v in V:
        for w in V:
            dist(v,w)=min(dist(v,w),dist(v,u)+dist(u,w))
```

This has runtime $O(V^3)$ and works for negative edges. so better than running Bellman ford on everything, matches Dijkstra on dense graphs, but worse than Dijkstra on sparse graphs. intuition: it is important that the outer for loop corresponds to the inner vertex.

this is all and good, but wouldn't it be nice if we could get the runtime $O(VE + V^2 \log V)$ even for negative edges. this is called **Johnson's algorithm**, with runtime $O(VE + V^2 \log V)$ with negative edges. the general idea:

(1) find any consistent heuristic h
(2) run Dijkstra on w_h for all sources s

the hard part is finding a consistent heuristic, which will take VE time. what algo takes VE time? that's right, bellman ford. so

(1) run bellman ford from an arbitrary v^* that can reach the entire graph. then set $h(u) = -\text{dist}(v^*, u)$. we have $\text{dist}(v^*, v) - \text{dist}(v^*, u) \leq \text{dist}(v^*, v) \leq \text{dist}(v^*, u) + w(u \to v)$. so $\text{dist}(v^*, u) \leq \text{dist}(v^*, u) + w(u \to v)$, which is precisely $h(u) - h(v)$. now we can run Dijkstra, we converted this tricky graph into a nice graph that has the same set of shortest paths.