NOI.PH Training: Graphs 2

Standard Algorithms on Graphs

CJ Quines, Kevin Atienza

Contents

1 Shortest paths

This will be a pretty short document. I'll expand it later if I find the time. If you want more details on something in particular, let me know and I'll prioritize it! (i.e., I'll try to write it during the training week.)

You can use the following slides as a more complete tutorial: `https://drive.google.com/file/d/1-RcKKIwel2zpEhX6_d64hJ5mkkFyl0Sb/view` . The following parts will just be extra notes.

Check out VisuAlgo too: `https://visualgo.net/en/sssp?slide=1`

1.1 In DAGs

In directed acyclic graphs, you don't need to use any fancy algorithm; you can just run a DFS from the source.

This includes finding shortest paths on *weighted* DAGs, and even *longest paths*! (The longest path problem is hard[1] on general graphs—no one has found any polynomial-time solution yet.)

1.2 Dijkstra's algorithm

You know that `if (vis[next]) continue;` line in the implementation in the slides? Never forget that line. Never, ever, ever forget that line. (Exercise: what happens without that line?)

Dijkstra's essentially BFS, but using a priority queue instead of a queue. I say "essentially" because there may be other technical differences, e.g., (important!) only marking a node as visited when you actually pop it from the queue, rather than when you push it. Also, in my usual implementation, I commonly insert a node in a priority queue multiple times (once every time I encounter it), which will never happen in BFS. I do this for convenience: it makes the implementation much simpler. The effect on the running time is that we get a $\log e$ factor instead of $\log n$, which in most practical scenarios is just a constant factor away ($e = \mathcal{O}(n^2)$ in *simple* graphs), and it's very unlikely to be a bottleneck.

1.3 Bellman–Ford algorithm

Bellman–Ford is the only one among these "basic" shortest path algorithms that works when there are negative-cost edges. It detects negative cycles (which means that the idea of a "shortest walk" doesn't really make much sense since you can go around the negative cycle as many times as you want), and if there are none (at least none reachable from the source), it finds the shortest path.

The main issue is that its worst-case running time is a bit worse: $\mathcal{O}(ne)$ for a *single-source* shortest paths, compared to $\mathcal{O}((n+e)\log e)$ for Dijkstra's or $\mathcal{O}(n^3)$ for Floyd's *all-pairs* shortest paths.

Much much later on, you may hear about something called "minimum-cost flows". One algorithm on minimum-cost flows involves invoking a shortest-paths algorithm multiple times on graphs that may have negative costs, so Bellman–Ford is often used there.

[1]More precisely, NP-hard.

1.4 Floyd–Warshall algorithm

Floyd–Warshall is also an easy-to-implement way to compute transitive closure in directed graphs. For example, if you had a set of airports, and you knew which direct flights there were from one airport to another, you can use Floyd–Warshall to compute, for each pair of airports, whether there is a series of flights that goes from the first airport to the second airport.

1.5 Implementation tricks

If you have multiple sources, you can add a new vertex, and then add several weight-zero edges from that vertex to the source vertices. Same thing with multiple targets.

Also, when you're given a graph, don't necessarily think you have to work with that graph. You can change a vertex to be multiple vertices to keep track of "states". You can add more edges to represent boosts or shortcuts.

1.6 Shortest paths are a state of mind

I think I mentioned this in Backtracking 1, but you know, backtracking is pretty much just DFS. And things like A*, the minimax algorithm, or alpha–beta pruning, are also just search algorithms. The underlying logic is the same; the difference is the order we search.

To emphasize this, consider this code, which is a DFS:

```cpp
bool visited[N];
for (int i = 1; i <= n; i++) visited[i] = false;
stack<int> st;

st.push(i); // source
while (!st.empty()) {
    v = st.top(); st.pop();
    if (visited[v]) continue;
    visited[v] = true;
    for (auto u : g[v]) {
        if (!visited[u]) st.push(u);
    }
}
```

If you replaced `stack<int> st` with `queue<int> qu`, then it's a BFS. If you replaced it, instead, with a priority queue, where you explore vertices in increasing distance, then you have Dijkstra's. You'd have to add additional information to make sure the priority queue explores vertices in increasing distances, so the insertion step would include updating the distances. But that's it. That's the only difference between BFS, DFS, and Dijkstra's.

Another thing. Shortest paths algorithms aren't restricted to finite graphs, or even graphs at all. One useful perspective is to think of it as just looking for the fastest way to get from any set of source states, to any set of target states. AI researchers call this uniform–cost search instead of Dijkstra's, because that's fancier; see https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#Practical_optimizations_and_infinite_graphs .

1.7 Transitive Closure

There's a thing called "transitive closure". It's a mathematical concept, but we can translate it into graph terms which may feel more familiar to you.

Essentially, the problem of finding the transitive closure is: For every pair of nodes i and j, compute whether j is reachable from i. There are $\mathcal{O}(n^2)$ values, so we can put them into a Boolean/binary matrix: $r_{i,j} = 1$ if you can go from i to j and 0 otherwise. Note that $r_{i,i} = 1$ always.

Well, this is mostly a special case of shortest paths! So, just run your favorite shortest path algorithm. Then if the shortest path from i to j is ∞, then there's no path at all, and $r_{i,j} = 0$. Otherwise, $r_{i,j} = 1$.

With Floyd's algorithm for example, this runs in $\mathcal{O}(n^3)$. You can also directly choose to work with Boolean values instead of using *min*—it's mostly the same DP.

2 Minimum spanning trees

Again, see `https://drive.google.com/file/d/1-RcKKIwel2zpEhX6_d64hJ5mkkFylOSb/view` for a treatment. Check out VisuAlgo too: `https://visualgo.net/en/mst?slide=1` .

2.1 Kruskal's algorithm

We've talked about implementing UFDS in Graphs 1.

2.2 Prim's algorithm

I've never directly used Prim's, although I have used it as a basis for other algorithms.

2.3 Greedy algorithms

What you'll notice with Kruskal's algorithm and Prim's algorithm is that they're both *greedy* algorithms, which means they need to be proven correct. Greedy algorithms are bad unless they're proven correct. The proofs are pretty simple though. Please read them.

In fact, there's a somewhat deep connection between MSTs and greedy algorithms. The keyword is "matroids"—you don't have to learn this yet! Especially if you're overwhelmed.

3 Eulerian paths

For now, I will refer you to Wikipedia: `https://en.wikipedia.org/wiki/Eulerian_path`

An **Eulerian trail** (also commonly called "Eulerian path") on an undirected graph is a trail that goes through each edge exactly once. An **Eulerian circuit** (also commonly called "Eulerian cycle") also returns to the starting node.

The main theorems are pretty natural:

Theorem 3.1. We know precisely when there's an Eulerian trail and/or circuit in a *connected* undirected graph.

- If there are no nodes of odd degree, then there is an Eulerian circuit, and *every* Eulerian trail is an Eulerian circuit.
- If there are two nodes of odd degree, then there is an Eulerian trail, and every Eulerian trail goes from a node of odd degree to the other. In particular, there is no Eulerian circuit.
- If there are more than two nodes of odd degree, then there is no Eulerian trail.

Recall that there are always an even number of odd-degree nodes, so exactly one odd-degree node is impossible.

The main idea is pretty simple: *every time you enter a node, you have to leave.* This is true for every intermediate node, that's why they must all have even degrees. The only exception is the start and end nodes. It think this idea pretty much proves the theorem. You just need to fill in the details.

Also don't forget that the graph must be connected! It's pretty obvious why. Though actually, it still works for some disconnected graphs if only one connected component has edges at all, i.e., the other components are just *singleton* nodes.

The above is for undirected graphs. A similar thing holds for directed graphs. The key condition is: *the indegree must equal the outdegree* for intermediate nodes, because of the same principle *every time you enter, you have to leave.* I encourage you to state the general analog of Theorem 3.1 for directed graphs.

3.1 Hierholzer's algorithm

The Wikipedia page has a good description of Hierholzer's. In practice, I never implement it with a doubly linked list, and instead use a `set`. I've never used the queue implementation before. (There's one implementation below.)

The gist is:

- First find a path from one odd-degree node to the other. (Or if there are none, just find any cycle.) You can do this without doing anything fancy, not even DFS/BFS, since you can just *keep walking*. You're guaranteed not to get stuck because the degrees are even—every time you enter, you can always leave.
- While there are still unvisited edges, do the above again on the remaining edges. Again, you won't get stuck.

- Combine these into one big trail/circuit. They're guaranteed to be intertwined because the graph is connected.

You can do this in linear time with the right implementation.

Actually, it's kinda fun to implement yourself. You can also check my implementation below, but **it's more educational to try implementing it yourself** before looking at it:

```cpp
// zero-indexed
struct EulerianTrailer {
    int n;
    vector<vector<pair<int, int>>> adj;
    vector<bool> edge_available;
    EulerianTrailer(int n): n(n), adj(n) {}

    // add a new edge. call this many times before finding Eulerian path
    void add_edge(int i, int j) {
        // check that zero-indexed
        assert(0 <= i && i < n);
        assert(0 <= j && j < n);
        // add edge
        int idx = edge_available.size();
        adj[i].emplace_back(j, idx);
        adj[j].emplace_back(i, idx);
        edge_available.push_back(true);
    }

    // pop an edge incident with i.
    int _pop_neighbor(int i) {
        while (!adj[i].empty()) {
            auto [j, idx] = adj[i].back(); adj[i].pop_back();
            if (edge_available[idx]) {
                edge_available[idx] = false;
                return j;
            }
        }
        return -1;
    }

    // walk from t to s. push everything onto stack st
    void _walk(int s, int t, stack<int>& st) {
        st.push(t);
        while (s != st.top())
            st.push(_pop_neighbor(st.top()));
    }

    // return an Eulerian trail (as a sequence of nodes) from s to t
    vector<int> find_eulerian_trail(int s, int t) {

        // verification (remove for speed)
        // verify zero-indexed
        assert(0 <= s && s < n);
        assert(0 <= t && t < n);
        // verify even degrees (except s and t)
        for (int i = 0; i < adj.size(); i++) {
            assert((adj[i].size() + (i == s) + (i == t)) % 2 == 0);
        }

        // find path
        vector<int> past;
        stack<int> future;
        _walk(s, t, future);
        while (!future.empty()) {
            int curr = future.top(), neigh; future.pop();
            past.push_back(curr);
            // walk from curr to curr until
            // all edges incident to curr are consumed
            while ((neigh = _pop_neighbor(curr)) != -1)
                _walk(neigh, curr, future);
        }

        return past;
    }
};
```

3.2 Euler tours of trees

You've probably heard of an "Eulerian tour"—or more accurately, "Eulerian trail"—of a graph, which is a tour that passes through each edge exactly once. The term "Euler tour" on trees is somewhat different though: it's a tour that passes through each edge exactly *twice*.

You can think of this better as "walking along the boundary" of the tree, such as the following (Figure 1):

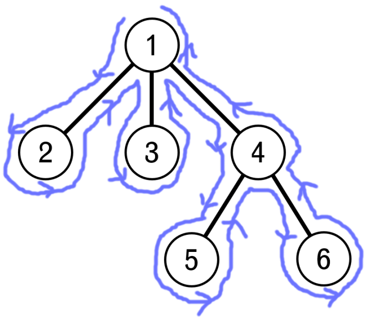

Figure 1: The Euler tour of a tree.

Note that unlike the Eulerian trails on graphs which only exist if most nodes have even degrees, Eulerian tours *always* exist, precisely because we're passing through each edge exactly twice, so each node effectively has double its original degree. (So you can kinda think of the Eulerian tour on a tree as the Eulerian trail on it with every edge doubled.)

Finding the Euler tour is straightforward: just DFS, and make sure to keep track of each edge you pass through, *including the return to the parent.*

4 Planarity

We've introduced planar graphs in Graphs 1. For now, you could try reading https://drive.google.com/file/d/1-RcKKIwel2zpEhX6_d64hJ5mkkFylOSb/view .

The Wikipedia page may also be helpful: https://en.wikipedia.org/wiki/Planar_graph

Some points:

- 3Blue1Brown has a nice introduction to planar graphs: https://www.youtube.com/watch?v=VvCytJvd4H0&pp=ygUVM2JsdWUxYnJvd24gdXRpbGl0aWVz
- There's a formula $v - e + f = 2$ which holds for connected planar graphs. V is the number of nodes (or "vertices"), E is the number of edges, and F is the number of faces. This is called Euler's formula. (It is traditional to use v instead of n when talking about Euler's formula, and also to use "vertex" instead of "node".)
- The quantity $v - e + f$ is called the **Euler characteristic**. Its value depends only on where you're drawing the graph: On the plane, it's always 2. On other surfaces, this quantity may be different, e.g., on a torus (doughnut surface), it's 0. (On a sphere, it turns out to also be 2.) In other words, the Euler characteristic is actually a number associated with the *surface*.[2]
- The more general Euler's formula for planar graphs applies to even disconnected graphs. If there are c components, then we have $v - e + f - c = 1$. It can be proven by induction, just like the original Euler's formula.
- There are also higher-dimensional analogs of Euler's characteristic, though you probably won't need that much in competitive programming. But still, it's a fun trivia. Read about it if you're interested.
- Because of Euler's formula, we can prove the following bound for connected, *simple* planar graphs: $e \le 3v - 6$ (if $v \ge 3$). In particular, $e = \mathcal{O}(v)$ which means planar graphs are *sparse*. You can trivially prove that K_5 is not planar using this.
- A proof goes as follows: $2e \ge 3f$ because each face is bounded by at least three edges, and each edge is a boundary of two faces. (The quantity $2e$ counts the number of adjacent edge-face pairs, and $3f$ is a lower bound.) Combine it with $v - e + f = 2$ to get the inequality immediately.
- You can also prove that $K_{3,3}$ is not planar using a similar, stronger bound for bipartite planar graphs: $e \le 2v - 4$. (3Blue1Brown covers this, including a proof. Basically the proof is like the above, except you have $2e \ge 4f$.)
- There are high-powered theorems called **Kuratowski's theorem** and **Wagner's theorem** characterizing planar graphs. They give a "simple" criterion to test whether a graph is planar or not. Roughly, they both say that a graph is planar if and only if K_5 and $K_{3,3}$ are "not in it". The "not in it" has a precise version, different for both thoeorems. (See the Wikipedia page.)

[2]Caveat: $v - e + f$ is only guaranteed to equal to the Euler characteristic if all faces can be deformed to a disc.[3] On a sphere, this is automatically true (if the graph is connected). For example, if you draw a singleton graph on a donut, then the "face" can't really be deformed to a disc. If you draw a self-loop around the donut, then the face is still connected, but looks like a cyclinder so it still can't be deformed to a disc. But if you draw a second self-loop around the other direction fo the donut, then the face becomes deformable to a disc, so $v - e + f = 0$ now holds.

- There are algorithms to test for planar graphs. There are linear-time ones too. They rarely come up in contests though, but you're welcome to study them if you like. The keyword is "planarity testing".
- Planar graphs have a *dual* graph: Consider the faces as vertices, and consider two faces to be adjacent if they share a nontrivial boundary. See Figure 2.

  You should be able to see that the dual graph is a planar graph too. Funnily, the "edges" are the same in both graphs; they're just interpreted somewhat differently in both cases.

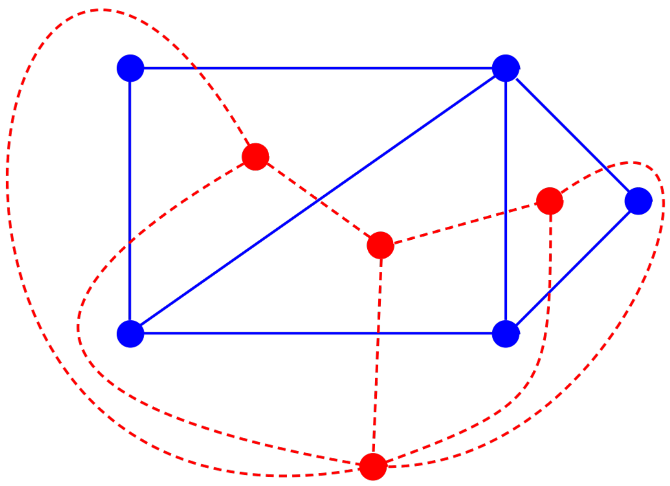

Figure 2: A planar graph and its dual. Image taken from Wikipedia.

5 Problems

5.1 Non-coding problems

N1 [2★] Show that Dijkstra sometimes fails when negative edges are allowed.

N2 [2★] Show that Floyd–Warshall sometimes fails when negative edges are allowed.

N3 Prove or disprove:

(a) [2★] One can solve the maximum spanning tree problem by negating the weights and taking the minimum spanning tree.

(b) [2★] If a connected undirected simple graph has an Eulerian Cycle, then it has a Hamiltonian Path.

(c) [2★] If a connected undirected simple graph has a Hamiltonian Cycle, then it has an Eulerian Path.

(d) [2★] Consider an algorithm that picks a vertex s of a graph, computes the shortest path distances to all other vertices, and returns the sum of these distances. This sum is at most twice the weight of the minimum spanning tree.

(e) [2★] A simple, connected graph has at least two edges, and all the weights of the edges are different. The edge with the second smallest weight is in the minimum spanning tree.

(f) [3★] Let T be a minimum spanning tree. If we increase the weight of some edge e, then there exists a minimum spanning tree T' in the new graph, such that either $T' = T$, or $T' = T \setminus \{e\} \cup \{e'\}$ for some edge $e' \neq e$.

(g) [3★] Consider the following attempt at solving the "longest simple path problem". First, negate the edges. Then determine if there is a negative cycle using Bellman–Ford. If there is, we output nothing. Otherwise, output all pairs of shortest paths (using Bellman–Ford once for each vertex), and print the shortest one. Finally, negate all edges again. This algorithm is correct if it outputs something.

N4 [5★] In a connected graph, all the edges have weight either 1 or 2. Prove it is possible to find the minimum spanning tree in $O(|E|)$.

N5 Consider the following attempt at finding an Eulerian cycle for a graph. We start from a given vertex v, and randomly cross an edge that we haven't crossed yet, until we have an Eulerian cycle. We say that an undirected, connected graph G is a *well-designed art gallery from v* if this algorithm always works.

(a) [3★] Prove that G is a well-designed art gallery from v if and only if G has an Eulerian cycle, and v that is contained in every cycle of G.

(b) [5★] Recall that a forest is a simple, undirected graph with no cycles. Suppose we have a forest F, and we add a vertex v, and join v to every vertex of odd degree in F. Prove that the new graph is a well-designed art gallery from v, and that all well-designed art gallerys can be obtained in this way.

(c) [3★] Design an $O(|E|)$ algorithm to determine whether an undirected, connected graph is a well-designed art gallery from a given vertex.

N6 In the complete graph K_n, with an edge between each pair of distinct vertices, the weights of the edges are $1, 2, \ldots, \binom{n}{2}$ in some order.

(a) [3★] Show that it is possible for the weight of the minimum spanning tree to be

$$\sum_{i=1}^{n-1} \left(\binom{i}{2} + 1 \right).$$

Here, $\binom{1}{2} = 0$.

(b) [8★] Prove that this is the maximal bound possible. That is, no matter what assignment of the weights, the weight of the minimum spanning tree is at most the above number.

5.2 Coding problems

It is strongly recommended to solve, think about, or at least look at, **all** of the red problems. They're all great.

The problems are a bit more difficult than a typical problem set would be. Don't worry about solving all of them. Try to solve problems at your level: not too easy, and not too hard.

C1 [5★] **Rainbow Dash:** Problem I in https://cjquines.com/files/icpcmanila2016.pdf

C2 [5★] **Frog Pushers:** Problem F in https://cjquines.com/files/icpcmanila2016.pdf

C3 [8★] **Zadanie Sumy:** https://szkopul.edu.pl/problemset/problem/ROXsaseQWYR11jbNvCgM19Er/site/?key=statement. If you use Google Translate, remember to output `TAK` and `NIE`, not `YES` and `NO`.

S1 [2★] **Anti Brute Force Lock:** UVa 1235

S2 [2★] **Biridian Forest:** https://codeforces.com/problemset/problem/329/B

S3 [3★] **Greg and Graph:** https://codeforces.com/problemset/problem/295/B

S4 [3★] **XYZZY:** UVa 10557

S5 [3★] **The Cheapest Reid:** https://www.hackerrank.com/contests/noi-ph-2017/challenges/the-cheapest-reid

S6 [3★] **Planets:** https://codeforces.com/problemset/problem/229/B

S7 [5★] **Curious Fleas:** UVa 11329

S8 [5★] **Chef and Reversing:** https://www.codechef.com/problems/REVERSE

S9 [5★] **Complete The Graph:** https://codeforces.com/problemset/problem/715/B

S10 [5★] **Legacy:** https://codeforces.com/problemset/problem/786/B

S11 [5★] **Across the River:** https://www.codechef.com/problems/RIVPILE

S12 [8★] **Okabe and City:** https://codeforces.com/problemset/problem/821/D

S13 [8★] **Perishable Roads:** https://codeforces.com/problemset/problem/806/D

S14 [8★] **Year 3017:** https://www.codechef.com/problems/UNIVERSE

S15 [3★] **Airports:** UVa 11733

S16 [3★] **ACM Contest and Blackout:** UVa 10600

S17 [5★] **Design Tutorial: Inverse the Problem:** https://codeforces.com/problemset/problem/472/D

S18 [5★] **Edges in MST:** https://codeforces.com/problemset/problem/160/D

S19 [5★] **Leha and another game about graph:** https://codeforces.com/problemset/problem/840/B

S20 [5★] **Road Maintenance:** https://www.codechef.com/problems/INF16K

S21 [8★] **Sebi and the corrupt government:** https://www.codechef.com/problems/SETELE

S22 [3★] **Spratly Islands Tour:** https://www.hackerrank.com/contests/noi-ph-2019/challenges/spratly-islands-tour

S23 [5★] **City Inspection:** https://www.codechef.com/problems/RAMSINSP

S24 [8★] **Sagheer and Kindergarten:** https://codeforces.com/problemset/problem/812/D

Acknowledgments

Thanks to Kyle and Hadrian for writing the 2017 material for Week 6. Some exercises are taken from the MIT classes 6.006, 6.046, Bollobas's Modern Graph Theory, and Bondy and Murty's Graph Theory.