Kattis

Weighted Graphs

Arnar Bjarni Arnarson Atli FF

6. nóvember 2023

School of Computer Science
ReykjavÃk University

Today we're going to cover

- Topological sort
- Strongly connected components
- Minimum spanning tree
- Shortest paths

Topological sort

- We have n tasks
- Each task i has a list of tasks that must be completed before we can start task i
- Find an order in which we can process the tasks
- Can be represented as a directed graph
  - Each task is a vertex in the graph
  - If task j should be finished before task i, then we add a directed edge from vertex i to vertex j
- Notice that this can't be solved if the graph contains a cycle
- A modified depth-first search can be used to find an ordering in $O(n + m)$ time, or determine that one does not exist

Topological sort

```
vector<int> adj[1000];
vector<bool> visited(1000, false);
vector<int> order;

void topsort(int u) {
    if (visited[u]) {
        return;
    }

    visited[u] = true;
    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        topsort(v);
    }

    order.push_back(u);
}

for (int u = 0; u < n; u++) {
    topsort(u);
}
```

Strongly connected components

- We know how to find connected components in undirected graphs
- But what about directed graphs?

- Such components behave a bit differently in directed graphs, especially since if v is reachable from u, it doesn't mean that u is reachable from v

- The definition remains the same, though
- A strongly connected component is a maximal subset of the vertices such that each pair of vertices is reachable from each other

Strongly connected components

- The connected components algorithm won't work here
- Instead we can use the depth-first search tree of the graph to find these components
- *see example*

Strongly connected components

```
vector<int> adj[100];
vector<int> low(100), num(100, -1);
vector<bool> incomp(100, false);
int curnum = 0;

stack<int> comp;

void scc(int u) {

    // scc code...
}

for (int i = 0; i < n; i++) {
    if (num[i] == -1) {
        scc(i);
    }
}
```

Strongly connected components

```
void scc(int u) {
    comp.push(u);
    incomp[u] = true;

    low[u] = num[u] = curnum++;
    for (int i = 0; i < adj[u].size(); i++) {
        int v = adj[u][i];
        if (num[v] == -1) {
            scc(v);
            low[u] = min(low[u], low[v]);
        } else if (incomp[v]) {
            low[u] = min(low[u], num[v]);
        }
    }

    if (num[u] == low[u]) {
        printf("comp: ");
        while (true) {
            int cur = comp.top();
            comp.pop();
            incomp[cur] = false;
            printf("%d, ", cur);
            if (cur == u) {
                break;
            }
        }

        printf("\n");
    }
}
```

Strongly connected components

- Time complexity?
- Basically just the DFS analyze function (which was $O(n+m)$), with one additional loop to construct the component
- But each vertex is only in one component...
- Time complexity still just $O(n+m)$

Example problem: Tourist safety

Tourists sometimes like to explore the cities they're in. They start at their hotel, and then drive randomly around the city. This can be dangerous, however, as when the tourist becomes tired and wants to go home, it may not be possible!

Given a directed road network, determine which road intersections are tourist-safe. A road intersection u is tourist-safe if for each road intersection v reachable from u, u is reachable from v.

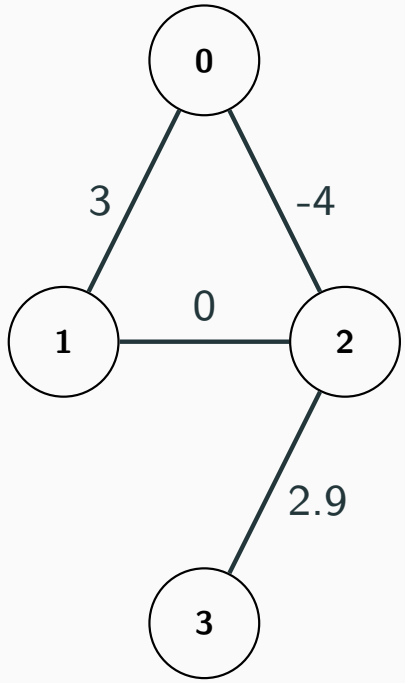
Weighted graphs

- Now the edges in our graphs may have weights, which could represent
  - the distance of the road represented by the edge
  - the cost of going over the edge
  - some capacity of the edge
- We can use a modified adjacency list to represent weighted graphs

Weighted graphs

```
struct edge {
    int u, v;
    int weight;

    edge(int _u, int _v, int _w) {
        u = _u;
        v = _v;
        weight = _w;
    }
};
```



Weighted graphs

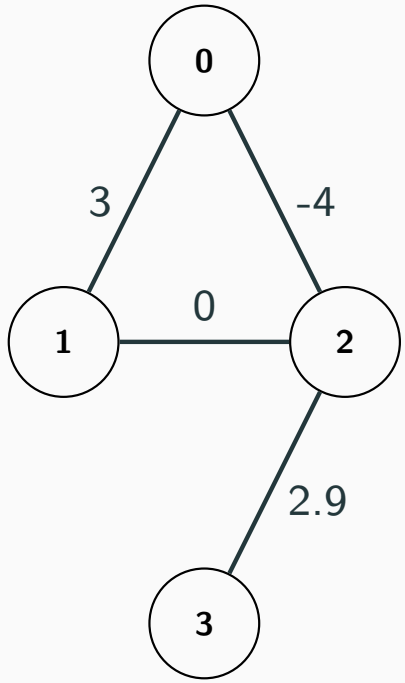
```
vector<edge> adj[4];

adj[0].push_back(edge(0, 1, 3));
adj[0].push_back(edge(0, 2, -4));

adj[1].push_back(edge(1, 0, 3));
adj[1].push_back(edge(1, 2, 0));

adj[2].push_back(edge(2, 0, -4));
adj[2].push_back(edge(2, 1, 0));
adj[2].push_back(edge(2, 3, 2.9));

adj[3].push_back(edge(3, 2, 2.9));
```



Minimum spanning tree

- We have an undirected weighted graph
- The vertices along with a subset of the edges in the graph is called a spanning tree if
  - it forms a tree (i.e. does not contain a cycle) and
  - the tree spans all vertices (all vertices can reach all other vertices)
- The weight of a spanning tree is the sum of the weights of the edges in the subset
- We want to find a minimum spannig tree

Minimum spanning tree

- Several greedy algorithms work

- Go through the edges in the graph in increasing order of weight
- Greedily pick an edge if it doesn't form a cycle (Union-Find can be used to keep track of when we would get a cycle)
- When we've gone through all edges, we have a minimum spanning tree

- This is Kruskal's algorithm
- Time complexity is $O(E \log E)$
- Other algorithms are Prim's and Boruvska's

Minimum spanning tree

```
bool edge_cmp(const edge &a, const edge &b) {
    return a.weight < b.weight;
}

vector<edge> mst(int n, vector<edge> edges) {
    union_find uf(n);
    sort(edges.begin(), edges.end(), edge_cmp);

    vector<edge> res;
    for (int i = 0; i < edges.size(); i++) {
        int u = edges[i].u,
            v = edges[i].v;

        if (uf.find(u) != uf.find(v)) {
            uf.unite(u, v);
            res.push_back(edges[i]);
        }
    }

    return res;
}
```

Shortest paths

- We have a weighted graph (undirected or directed)
- Given two vertices u, v, what is the shortest path from u to v?

- If all weights are the same, this can be solved with breadth-first search
- Of course, this is usually not the case...

Shortest paths

- There are many known algorithms to find shortest paths
- Like breadth-first search, these algorithms usually find the shortest paths from a given start vertex to all other vertices
- Let's take a quick look at Dijkstra's algorithm, the Bellman-Ford algorithm and the Floyd-Warshall algorithm

Dijkstra's algorithm

```
vector<edge> adj[100];
vector<int> dist(100, INF);

void dijkstra(int start) {
    dist[start] = 0;
    priority_queue<pair<int, int>,
                   vector<pair<int, int> >,
                   greater<pair<int, int> > > pq;
    pq.push(make_pair(dist[start], start));

    while (!pq.empty()) {
        int u = pq.top().second;
        pq.pop();

        for (int i = 0; i < adj[u].size(); i++) {
            int v = adj[u][i].v;
            int w = adj[u][i].weight;

            if (w + dist[u] < dist[v]) {
                dist[v] = w + dist[u];
                pq.push(make_pair(dist[v], v));
            }
        }
    }
}
```

Dijkstra's algorithm

- Time complexity is $O(V \log E)$
- Note that this only works for non-negative weights

Bellman-Ford algorithm

```
vector<edge> adj[100];
vector<int> dist(100, INF);

void bellman_ford(int n, int start) {

    dist[start] = 0;

    for (int i = 0; i < n - 1; i++) {
        for (int u = 0; u < n; u++) {
            for (int j = 0; j < adj[u].size(); j++) {
                int v = adj[u][j].v;
                int w = adj[u][j].weight;
                dist[v] = min(dist[v], w + dist[u]);
            }
        }
    }
}
```

Bellman-Ford algorithm

- Time complexity is $O(V \times E)$
- Can be used to detect negative-weight cycles, how?

Bellman-Ford algorithm

- Time complexity is $O(V \times E)$
- Can be used to detect negative-weight cycles, how?
- At most N edges in a path before it contains a cycle.

Bellman-Ford algorithm

- Time complexity is $O(V \times E)$
- Can be used to detect negative-weight cycles, how?
- At most N edges in a path before it contains a cycle.
- Do extra iterations on the outer loop.

Bellman-Ford algorithm

- Time complexity is $O(V \times E)$
- Can be used to detect negative-weight cycles, how?
- At most N edges in a path before it contains a cycle.
- Do extra iterations on the outer loop.
- If there is any change then there is at least one negative weight cycle.

Floyd-Warshall algorithm

- What about using dynamic programming to compute shortest paths?
- Let $\mathrm{sp}(k, i, j)$ be the shortest path from i to j if we're only allowed to travel through the vertices $0, \ldots, k$
- Base case: $\mathrm{sp}(k, i, j) = 0$ if $i = j$
- Base case: $\mathrm{sp}(-1, i, j) = \mathrm{weight}[i][j]$ if $(i, j) \in E$
- Base case: $\mathrm{sp}(-1, i, j) = \infty$
- $\mathrm{sp}(k, i, j) = \min \begin{cases} \mathrm{sp}(k-1, i, k) + \mathrm{sp}(k-1, k, j) \\ \mathrm{sp}(k-1, i, j) \end{cases}$

Floyd-Warshall algorithm

```
int dist[1000][1000];
int weight[1000][1000];

void floyd_warshall(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            dist[i][j] = i == j ? 0 : weight[i][j];
        }
    }

    for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
            }
        }
    }
}
```

Floyd-Warshall algorithm

- Computes all-pairs shortest paths
- Time complexity is clearly $O(n^3)$
- Very simple to code

Floyd-Warshall algorithm

```
int dist[1000][1000];
int weight[1000][1000];

void floyd_warshall(int n) {
    rep(i,0,n) rep(j,0,n)
        dist[i][j] = i == j ? 0 : weight[i][j];

    rep(k,0,n) rep(i,0,n) rep(j,0,n)
        dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
}
```