INF280: Competitive programming

More advanced graph algorithms

Louis Jachiet

Implicit graphs

Implicit graph

The majority of the algorithms that you will implement can be seen as graph algorithms:

Implicit graph

The majority of the algorithms that you will implement can be seen as graph algorithms:

- directly an application of a graph algorithm

Implicit graph

The majority of the algorithms that you will implement can be seen as graph algorithms:

- directly an application of a graph algorithm
- a modified version of a graph algorithm

Implicit graph

The majority of the algorithms that you will implement can be seen as graph algorithms:

- directly an application of a graph algorithm
- a modified version of a graph algorithm
- an application of a graph algorithm over a *hidden* graph

Implicit graph

The majority of the algorithms that you will implement can be seen as graph algorithms:

- directly an application of a graph algorithm
- a modified version of a graph algorithm
- an application of a graph algorithm over a *hidden* graph

$\Rightarrow$ describing your problem as a graph problem *usually helps*

Rabbit

We have a graph where nodes are cells of the grid and edge are between nodes that are neighbors in the grids. *Find the path between two given points?*

Piggyback

Given a weighted graph G defining a distance d between nodes.

Find the node v minimizing $Bd(v, 1) + Ed(v, 2) + Pd(v, n)$.

Examples 3/3

Moocast

G is the graph where nodes are cows and an edge (a, b) exists when b can hear a.

Find the node that can reach most other nodes.

Why explicit the implicit graphs?

Help you reason over the problem:

- is it exactly the same problem?
- what are the properties of this implicit graph?
- can the problem on the implicit graph be simplified?
- can we reduce the number of nodes? of transitions?
- are we lacking important properties from the original graph?

Help you code the problem

The more standard algorithms you use the less likely you are to have bugs.

Union-find

Union-Find purpose

Maintain a collection of non-overlapping sets with the following operations

- Add a new element, in its own set
- Get the set of an element
- Merge two sets

Queries we might need to answer

- Given two elements, are they in the same component?
- What the size of the component of x?
- What is the number of components?

Union-Find

```
repr[x] ; // initialized to -1
int find(int x) {
  if(repr[x] < 0) return x;
  return repr[x]=find(repr[x]); // path compression
}
bool unite(int a, int b) {
  a = find(a);
  b = find(b);
  if(a==b) { return false; }
  if(repr[a] > repr[b]) { swap(a,b); } // size
  repr[a] += repr[b] ;
  repr[b] = a;
  return true;
}
```

Minimum Spanning Trees (MST)

Minimum spanning tree

Spanning tree

Given a connected graph $G = (V, E)$ a spanning tree is a selection of $E' \subseteq E$ such that E' forms a tree covering all nodes in G.

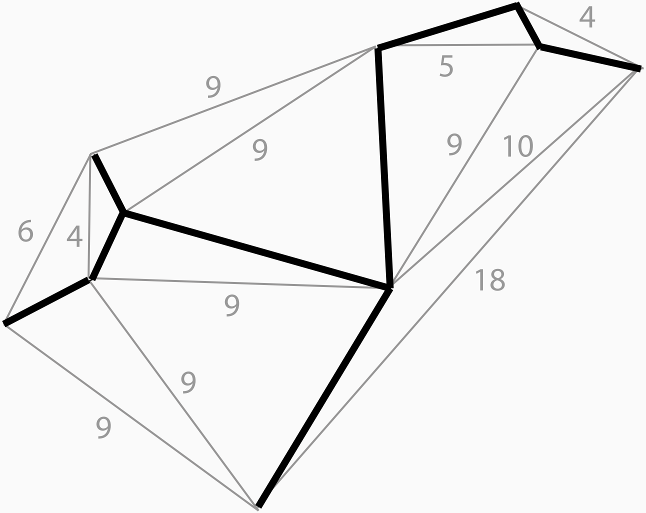
MST Problem

Find the spanning tree that has minimal total weight.

Properties

The MST also minimizes the maximal weight of an edge.

Example: Minimum Spanning Trees



https://commons.wikimedia.org/wiki/File:Minimum_spanning_tree.svg, Dcoetzee, public domain

Computing MST

Kruskal's algorithm

For all edges (a, b) by increasing weight

- if a and b not in the same component
    - link a and b

Computing MST

Kruskal's algorithm

For all edges (a, b) by increasing weight

- if a and b not in the same component
  - link a and b

Prim's algorithm

Make a modified Dijkstra:

- maintain a set S of nodes, initialized as $\{x\}$ for any node x
- while there remains a node not in S:
  - select an edge $\{n, n'\} \in E \cap (S, V \setminus S)$ minimizing $w(n, n')$
  - add $\{n, n'\}$ to E'

Kruskal

```
vector<pair<weight, pair<int,int> > > edges;
// ...
sort(edges.begin(),edges.end());
long long weight_mst = 0;
for(auto [w,p] : edges)
  if(unite(p.first,p.second))
    weight_mst += w;
```

Prim

```
long long dist[NB_NODES_MAX];
//...
fill(dist,dist+NB_NODES_MAX,INF);
set<pair<long long,int>> p_queue; // (weight, node)
p_queue.insert(make_pair(0,start_node));
dist[start_node] = 0;
while(!p_queue.empty()) {
  auto [node_dist, node] = *p_queue.begin() ; // c++17
  p_queue.erase(p_queue.begin());
  for(auto v : nxt[node])
    if(v.second < dist[v.first]) {
      p_queue.erase(make_pair(dist[v.first],v.first));
      dist[v.first] = v.second;
      p_queue.insert(make_pair(dist[v.first],v.first));
    }
}
```

Flows and matching

Flow network

Definition

A flow network G is a graph where each edge has a capacity value.
A flow network generally has a source s and an target t.

Flow

A flow in a G maps edges (a, b) to values $f_{a,b}$ such that:

- the flow along each edge is less than the capacity
- the source has an incoming flow equal to 0
- the sink has an outgoing flow equal to 0
- for other nodes, the total incoming flow is equal to the total outgoing flow

The value of a flow is the outgoing flow from s.

Max-flow = Min-Cut

Cut

In a flow network G with source s and target t, a cut is a partition of nodes into 2 partitions S and T such that $s \in S$, $t \in T$. The capacity of a cut is the sum of capacities of edges between S and T.

Theorem

$$\text{Max-Flow} = \text{Min-Cut}$$

This means that the maximal value of a flow is equal to the cut of minimum capacity.

Bipartite Matching

Matching in bipartite graph

In a weighted bipartite graph (V, E) with $V = X \sqcup Y$, a matching is a selection $E' \subseteq E$ of edges such that no nodes in (V, E') have degree higher than 1.

Maximum matching

A matching of maximal total weighted.

Reduction to max-flow

Create two new nodes s and t, link s to all nodes in X and t to all nodes in Y. All edges have capacity 1.

Ford-Fulkerson "algorithm" for flows

Residual graph

Given a flow network G and a flow f we can compute the residual flow network G' as G but where the capacity of an edge (a, b) is $c_{a,b} - f_{a,b}$. Notice than an edge is removed when $f_{a,b} = c_{a,b}$ and using the convention $f_{a,b} = -f_{b,a}$ an edge is created when $f_{b,a} < 0$.

Ford-Fulkerson Method

- Initialize f with empty flow
- While there exists a path p from s to t in the residual
    - increase f with the path p using maximal capacity

$\Rightarrow$ multiple algorithms to find the path lead to various complexities.

Ford-Fulkerson with DFS

```
int capa[Tm][Tm], flow[Tm][Tm]; // adjacency matrix
bool visited[Tm];
int dfs(int x, int max_flow) {
  if(visited[x]) return 0; // already search/ed for a flow
  if(x==target) return max_flow;// found our flow
  visited[x] = true; // stop visiting x
  for(int n: nxt[x]) // mixes adjacency lists with matrix
    if(flow[x][n] < capa[x][n]) { // residual
      const int sub_flow = dfs(n,
                min(max_flow,capa[x][n]-flow[x][n]));
      if(sub_flow > 0) {
        flow[x][n]+= sub_flow;
        flow[n][x]-= sub_flow;
        return sub_flow;
      }
  return 0; // haven't found a flow
}
```

Ford-Fulkerson with DFS

```
int totalFlow = 0, curFlow = 1 ;
while(curFlow > 0) {
  fill(visited,visited+Tm,false) ;
  curFlor = dfs(source,INF) ;
  totalFlow += curFlow ;
}
// in the worst case the flow increases by one each time
// hence in O(E) × F where F is the final flow
// if using integers
```

Recognize flow algorithms

Flow problems are usually a bit counter-intuitive and hard to recognize...

Multiple algorithms

The code above is for Ford-Fulkerson with DFS, this is not the fastest method but the simplest. You can replace the DFS with a BFS to improve the worst-case complexity.