Graphs Overview for Competitive Programming 2
Minimum Spanning Tree (MST)

• Input: Undirected weighted graph
• Output: A spanning tree with minimal weight

• Properties:
  • The heaviest edge in a cycle will not be in any MST (aka “Red rule”)
  • The lightest edge in a cut will be in every MST (aka “Blue rule”)
  • All MSTs have the same number of edges of each weight
    • If the weights are unique, there is a unique MST
  • Corollary: Finding a maximum spanning tree is equivalent (just negate the weights)
Minimum Spanning Tree (MST)

• Prim and Kruskal suggested well-known algorithms for finding an MST

<table>
<thead>
<tr>
<th></th>
<th>Prim</th>
<th>Kruskal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>Grows a single component. At each step adding the lightest edge touching it.</td>
<td>Go over all edges by increasing weight. If adding the edge does not close a cycle, add it.</td>
</tr>
<tr>
<td>Implementation</td>
<td>Maintain a priority queue with the nodes adjacent to the component, along with the weight of the corresponding candidate edges. Take the minimum at each step.</td>
<td>Maintain a Union-Find data structure containing the components. An edge closes a cycle iff both the nodes belong to the same Union-Find component.</td>
</tr>
<tr>
<td>Time complexity</td>
<td>$O(</td>
<td>E</td>
</tr>
</tbody>
</table>
Minimax Paths

• Input: Weighted undirected graph, source and destination.
• Output: A path such the weight of the heaviest edge is minimal.
• Property: This is the path between the two nodes in a minimum spanning tree.
  • To find a minimax path, compute the MST and take the path between the two nodes.
• The opposite (maximizing the lightest edge) is called Widest Path.
• The widest path lies on a maximum spanning tree.
Max Flow

• Input: Directed graph with source and target, capacities on edges
• Output: The max possible flow from s to t

• Properties:
  • “Min Cut – Max flow”: max flow = min capacity of an s-t cut
  • Vertex capacity can be simulated by splitting the vertex to two, and adding an edge between them with the capacity

• Multiple sources can be simulated by adding a single source with outgoing edges to all sources (same for multiple targets)
Max Flow Applications

• Maximum matching on a bipartite graph
  • Find the max number of edges to keep such that each vertex touches at most one edge
  • Solve by connecting one side to s and the other side to t, using unit weights

• Maximum s-t edge-disjoint paths
  • Find the max number of s-t paths such that each edge appears in at most one path
  • Solve by assigning each edge with unit capacity

• Minimum path cover on a DAG
  • Find the min number of paths to cover the vertices in vertex-disjoint paths
  • Solve by duplicating each node and performing max matching on a bipartite graph
  • The size of the max matching is the number of paths needed
# Finding Max Flow

- Given a flow, the residual network represents the flow that can still be added. If there is an edge $u \rightarrow v$ with capacity $c$ and flow $f$, the residual network will have:
  - An edge $u \rightarrow v$ with capacity $c - f$
  - An edge $v \rightarrow u$ with capacity $f$

- **Ford-Fulkerson** is a general method to find max flow by repeatedly adding flow on a path from $s$ to $t$ on the residual network. **Edmonds-Karp** is a specific way of using Ford-Fulkerson by always choosing the shortest path. **Dinitz** is a more sophisticated way of finding a shortest path in the residual network.

<table>
<thead>
<tr>
<th></th>
<th>Ford-Fulkerson</th>
<th>Edmonds–Karp</th>
<th>Dinitz</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time</strong></td>
<td>(General method)*</td>
<td>$O(</td>
<td>V</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Build the residual graph. As long as there is an s-t path in the residual graph (called an augmenting path), send flow along one of the paths.</td>
<td>A specific version of Ford-Fulkerson. Choose the shortest augmenting path at each step. Use BFS from s to find the shortest path.</td>
<td>Similar to Edmonds-Karp. Improves on Edmonds-Karp by using some data structure, but is harder to implement.</td>
</tr>
</tbody>
</table>

* If the capacities are integers, bounded by $O(|E|F^*)$
You don’t have to remember everything!

• It is important to know what algorithms there are out there, to be able to choose to use them, and try to reduce the problems you are given to known problems with efficient algorithms.

• It is also useful to have an idea of how they work, in order to know when they can be used, and what problems may occur.

• You don’t have to remember the details or to be able to program them yourselves. This is why you have a formula sheet.

• A good formula sheet can make a huge difference in your success in solving problems quickly.

   It is highly recommended to have a good cheat sheet!

   (for the final competition, and for class)
Further Reading

• Algoritms 1 for proofs and more variations: http://webcourse.cs.technion.ac.il/234247

• Wikipedia tries to give intuitive explanations and usually contains code snippets.

• The Stanford ACM-ICPC Notebook contains optimized implementations for most graph algorithms: https://github.com/jaehyunp/stanfordacm