Network Flow (Graph Algorithms II)

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Australia

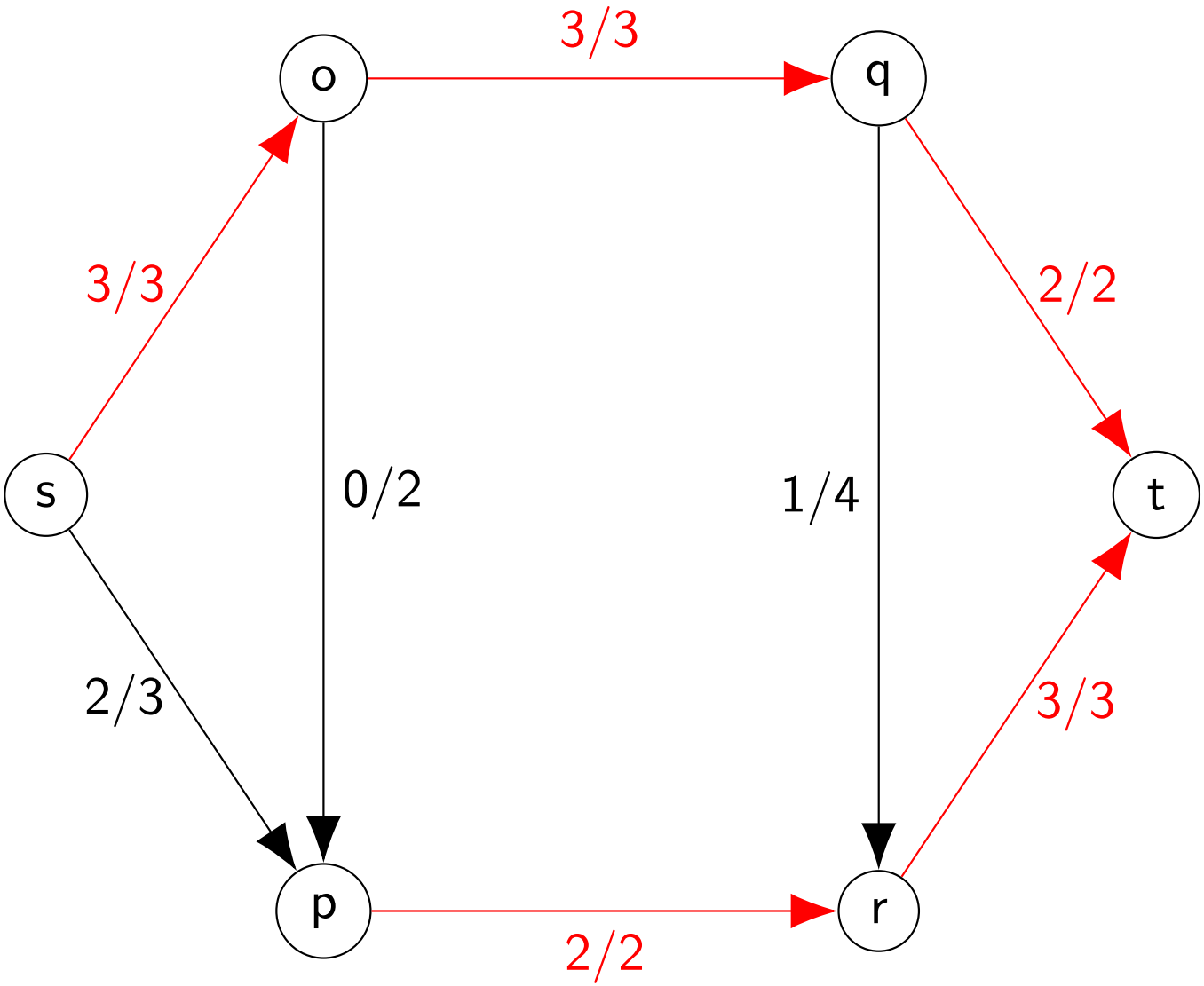
Table of Contents

Flow networks

- A *flow network*, or a *flow graph*, is a directed graph where each edge has a *capacity* that *flow* can be pushed through.
- Usually, there are two distinguished vertices, called the source (s) and the sink (t) that the flow comes from and the flow goes to.
- Intuitively, flow graphs can be likened to networks of pipes, each with a limit on the volume of water that can flow through it per unit of time.
- We won't be going through theoretical terms like *skew symmetry* and *flow conservation*.

Table of Contents

- The maximum flow problem is to find, given a flow graph with its edge capacities, what the maximum flow from the source to the sink is.
- We restrict ourselves to integer capacities because flow problems with real capacities are hard.
- The integrality theorem states that if all the edges in the graph have integer capacities, then there exists a maximum flow where the flow through every edge is an integer.
  - This doesn't mean that you can't find a maximum flow where the flow in some edges *isn't* integer, only that you won't need to.



- Given a graph with a source, a sink and edge weights, find the set of edges with the smallest sum of weights that needs to be removed to disconnect the source from the sink.
- We restrict ourselves to integer weights because cut problems with real weights are hard.

Max-flow/min-cut theorem

- It turns out that these two problems are actually equivalent!
- The max-flow/min-cut theorem states that the value of the minimum cut, if we set edge weights to be capacities in a flow graph, is the same as the value of the maximum flow.

- To see why this is so, we examine a modification of the original flow graph called the residual graph.
- The residual graph of a flow graph has all the same edges as the original graph, as well as a new edge going in the other direction for each original edge, with capacity zero.

Residual graphs

- We define an *augmenting path* as a path from s to t as a path that travels only on edges with positive capacity.
- Clearly, if we can find an augmenting path in our graph, we can increase the total flow of the graph.
- If we keep finding augmenting paths, and decreasing the capacities of the edges they pass through, we can keep increasing the flow in our graph.

- We also need to increase the capacity of the residual edge when we decrease the capacity of the original edge, so their sum stays constant.
- Intuitively, this is a mechanism allowing us to "fix" any mistakes that we make in choosing augmenting paths, because there are usually many different ones available, and we need to be able to change our minds if we choose a suboptimal path.
  - This works because we're allowed to use residual edges in an augmenting path (if they have positive capacity): then we decrease the capacity of the residual edge we use and increase the capacity of the corresponding original edge, effectively (partially) "cancelling out" a previous use of that edge by another augmenting path.

Max-flow/min-cut theorem

- It can be seen that if we can't find an augmenting path in our flow graph, then we have a maximum flow.
- Furthermore, this implies that in our residual graph, there exists no path from s to t, so we have discovered a cut.
- This cut must be a minimum cut, because if there were another smaller cut, it would be impossible to push as much flow as we have already pushed.

- So far, we haven't had much need to represent graphs as edge lists.
- One time this is necessary is when we want an easy way to refer to specific edges. For example, being able to find the residual edge of an edge.
- Another reason is being able to iterate over edges in $O(E)$ time without wasting an extra $O(V)$ time going over the adjacency list.
- There is an elegant but not entirely straightforward solution to this, effectively involving multiple linked lists interleaved in one array.

- **Implementation for a residual graph**

```
// the index of the first outgoing edge for each vertex, initialised to -1
int start[V];
fill(start, start + V, -1);
// if e is an outgoing edge from u, succ[e] is another one, or -1
// cap[e] is the capacity/weight of the e
// to[e] is the destination vertex of e
int succ[E], cap[E], to[E];

int edge_counter = 0;
void add_edge(int u, int v, int c) {
  // set the properties of the new edge
  cap[edge_counter] = c, to[edge_counter] = v;
  // insert this edge at the start of u's linked list of edges
  succ[edge_counter] = start[u];
  start[u] = edge_counter;
  ++edge_counter;
}

for (/* edge u -> v with capacity c in the original graph */) {
  add_edge(u, v, c); // original
  add_edge(v, u, 0); // residual edge
}

// edges are in pairs so we can easily go between residuals & originals
int inv(int e) { return e ^ 1; }

// easily iterate through all of u's outgoing edges (~(-1) == 0)
for (int e = start[u]; ~e; e = succ[e]) /* do something */;
```

Table of Contents

Ford-Fulkerson algorithm

- The previously described augmenting paths algorithm for finding maximum flows is called the Ford-Fulkerson algorithm.
- It runs in $O(Ef)$ time, where E is the number of edges in the graph and f is the maximum flow, since we need to perform an $O(E)$ graph search in the worst case to find a single unit of flow for each of the f units.

- **Implementation**

```
// assumes the residual graph is constructed as in the previous section

int seen[V];

int inv(int e) { return e ^ 1; }

bool augment(int u, int t, int f) {
  if (u == t) return true;                          // the path is empty!
  if (seen[u]) return false;
  seen[u] = true;
  for (int e = start[u]; ~e; e = succ[e])
    if (cap[e] >= f && augment(to[e], t, f)) {      // if we can reach the end,
      cap[e] -= f;                                  // use this edge
      cap[inv(e)] += f;                             // allow "cancelling out"
      return true;
    }
  return false;
}

int max_flow(int s, int t) {
  int res = 0;
  fill(seen, seen + V, 0);
  while (augment(s, t, 1)) {
    fill(seen, seen + V, 0);
    res += 1;
  }
  return res;
}
```

- The f in the time complexity of Ford-Fulkerson is not ideal, because f is exponential in the size of the input.
- It turns out that if you always take the shortest augmenting path, instead of any augmenting path, and increase the flow by the minimum capacity edge on your augmenting path, you need to find at most $O(VE)$ augmenting paths total.
- This gives a total time complexity of $O(VE^2)$.

Edmonds-Karp algorithm

- Since we saturate (use the maximum capacity) of at least one edge every time we find an augmenting path in Edmonds-Karp, we can find at most $O(E)$ augmenting paths before the shortest path must increase in length.
- The shortest path can increase in length at most $O(V)$ times, giving our $O(VE)$ bound.

- **Implementation**

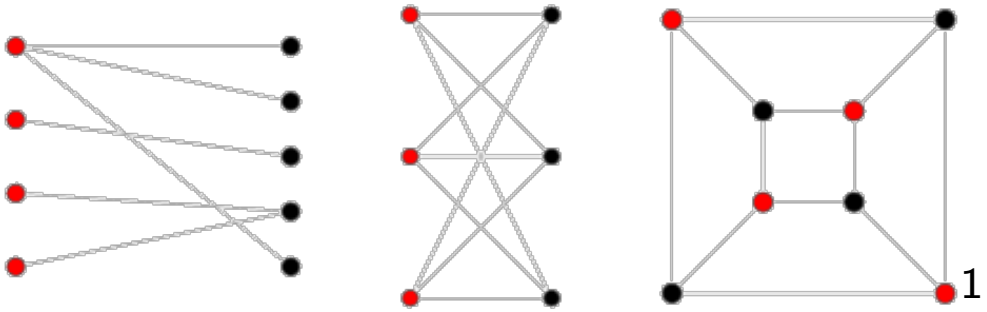
```
int augment(int s, int t) {
  // This is a BFS, shortest path means by edge count not capacity
  queue<int> q;
  // path[v] = which edge we used from to reach v
  fill(path, path + V, -1);
  for (q.push(s), path[s] = -2; !q.empty(); q.pop()) {
    int u = q.front();
    for (int e = start[u]; ~e; e = succ[e]) {
      // if we can use e and we haven't already visited v, do it
      if (cap[e] <= 0) continue;
      int v = to[e];
      if (path[v] == -1) {
        path[v] = e;
        q.push(v);
      }
    }
  }
  if (path[t] == -1) return 0; // can't reach the sink
  int res = INF;
  // walk over the path backwards to find the minimum edge
  for (int e = path[t]; e != -2; e = path[to[inv(e)]])
    res = min(res, cap[e]);
  // do it again to subtract that from the capacities
  for (int e = path[t]; e != -2; e = path[to[inv(e)]]) {
    cap[e] -= res;
    cap[inv(e)] += res;
  }
  return res;
}
```

Faster flow algorithms

- Faster flow algorithms exist, but they are outside the scope of this course.
- Augmenting path-based algorithms tend to run much faster in practice than their worst case time complexity suggests.
- The next algorithms to look at after Edmonds-Karp that still use augmenting paths run in $O(E^2 \log f)$ time (capacity scaling) and $O(V^2 E)$ (Dinic).
- Recently, it was shown that this problem can be solved in total $O(VE)$ time.

Table of Contents

- A bipartite graph is one where the vertices can be partitioned into two sets, where no vertices in the same set have an edge between them.



- The maximum bipartite matching problem, given a bipartite graph, is to choose the largest possible set of edges in that graph such that no one vertex is incident to more than one chosen edge.
- There is a clear flow formulation for this problem.

[1] http://mathworld.wolfram.com/BipartiteGraph.html

- Modify the original bipartite graph by making each edge a directed edge from the first set to the second set, with capacity 1.
- Attach an edge of capacity 1 from s to every vertex in the first set, and an edge of capacity 1 from every vertex in the second set to t.

- The size of the largest matching is the maximum flow in this graph.
- Since the flow of this graph is at most V, Ford-Fulkerson is preferred over Edmonds-Karp for this type of graph.

Table of Contents

Vertex capacities

- A graph has vertex capacities if the capacity restrictions are on the vertices instead of on the edges.
- This is solved by splitting each vertex into two vertices, an "in" vertex and an "out" vertex.
- For some vertex u with capacity c_u, we add an edge from in_u to out_u with capacity c_u.
- The edge capacities for the edges in the original graph are infinite.

Extracting the min-cut

- To extract the actual edges in the minimum cut, we use the fact that all of them must be saturated.
- We do a graph traversal starting from s and only traverse edges that have positive capacity and record which vertices we visit.
- The edges which have a visited vertex on one end and an unvisited vertex on the other will form the minimum cut.
- The residual edges need to be ignored!

Extracting the min-cut

- **Implementation**

```
void check_reach(int u) {
  if (seen[u]) return;
  seen[u] = true;
  for (int e = start[u]; ~e; e = succ[e]) {
    if (cap[e] > 0) check_reach(to[e]);
  }
}

vector<int> min_cut(int s, int t) {
  int total_size = max_flow(s, t);
  vector<int> ans;
  fill(seen, seen + V, 0);
  check_reach(s);
  // the odd-numbered edges are the residual ones
  for (int e = 0; e < edges; e += 2) {
    if (!seen[to[e]] && seen[to[inv(e)]]) {
      ans.push_back(e);
    }
  }
  return ans;
}
```

- A set of paths is edge-disjoint if no two paths use the same edge.
- To find the maximum number of edge-disjoint paths from s to t, make a flow graph where all edges have capacity 1.
- The maximum flow of this graph will give the answer.

- A vertex cover in a graph is a set of vertices which touches at least one endpoint of every edge.
- By Kőnig's theorem, the size of the maximum matching is equal to the number of vertices in a minimum vertex cover.

Table of Contents

- **Problem statement** Freddy Frog is on the left bank of a river. N ($1 \leq N \leq 100$) rocks are placed on the plane between the left bank and the right bank. The distance between the left and the right bank is D ($1 \leq D \leq 10^9$) metres. There are rocks of two sizes. The bigger ones can withstand any weight but the smaller ones start to drown as soon as any mass is placed on it. Freddy has to go to the right bank to collect a gift and return to the left bank where his home is situated.
- He can land on every small rock at most one time, but can use the bigger ones as many times as he likes. He can never touch the polluted water as it is extremely contaminated.
- Can you find a path such that the maximum distance Freddy has to jump is minimised?

- First, let's ignore the fact that we need to find the path with the smallest maximum edge length.
- We need to find a path from our source to our sink, and then back again, that does not cross any of the "small" vertices more than once.
- Keeping track of the state of the vertices after going to the sink and then coming back seems difficult, but we can transform it into the equivalent problem of finding two paths from the source to the sink.

Example problem: Jumping Frogs

- To find two vertex disjoint paths from the source to the sink, we need to find a flow of at least 2 in a flow graph with vertex capacities 1.
- But this formulation will only allow us to visit any of the "big" vertices once.

- The only thing that enforces that restriction in our graph is the vertex capacity of 1 in our original construction.
- So if we set the vertex capacity for every "big" vertex to be infinite, then we have a solution.
- Practically, an edge with capacity infinity is an edge with a large enough finite capacity that it'll never be reached.

Example problem: Jumping Frogs

- Now that we know how to find a valid path at all, how do we then find the one with the smallest maximum edge weight?
- It can be seen that if we can find a solution with the maximum edge weight being at most k, then we can also find a solution for any larger value j.
- So we can binary search for the smallest maximum edge weight, checking to see if we have a valid k by running our standard max-flow algorithm, ignoring any edges that have weight greater than k.

- We need $O(\log D)$ iterations of our binary search to find this maximum edge weight, and we can do a max-flow computation in $O(Ef) = O(V^2)$, because the maximum amount of flow we need is constant.
- The total runtime is $O(V^2 \log D)$.
- **Implementation**

```
double lo = 0, hi = 1e10;
for (int it = 0; it < 70; it++) {
  double mid = (lo + hi) / 2;
  if (max_flow(s, t, mid) >= 2) hi = mid;
  else lo = mid;
}
```