Network Flow (Graph Algorithms II)

COMP4128 Programming Challenges

School of Computer Science and Engineering
UNSW Australia

Table of Contents

Flow networks

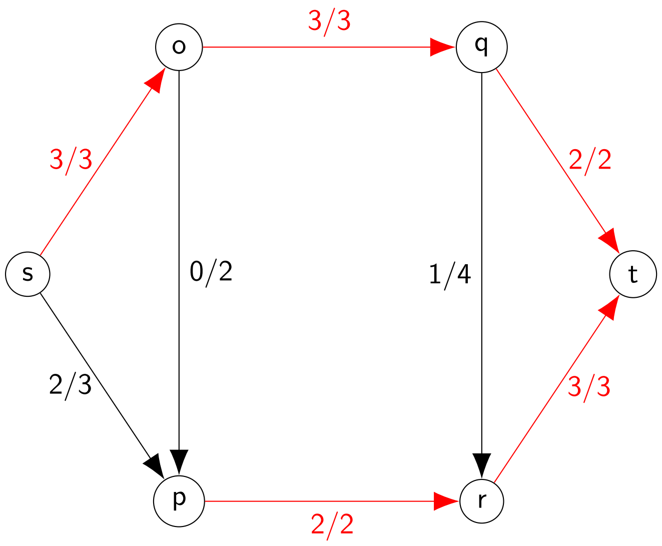
- A *flow network*, or a *flow graph*, is a directed graph where each edge has a *capacity* that *flow* can be pushed through.
- There are (usually) two distinguished vertices, called the source (s) and the sink (t) that the flow comes from and the flow goes to.
- Intuitively, flow graphs can be likened to networks of pipes, each with a limit on the volume of water that can flow through it per unit of time.
- The source has an infinite supply of water, the sink can drain an infinite amount of water. But for every other node, water in $=$ water out. For every edge, water in $\leq$ capacity.
- Formally defined as a system of inequalities, involving theoretical terms like *skew symmetry* and *flow conservation*. Not necessary for our purposes.

Table of Contents

Maximum Flow

- 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. This subclass is already interesting enough. There are subtle traps with real capacities. In particular, DONT use Ford-Fulkerson with real flows.
- 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.

Maximum Flow



Minimum cut problem

- The dual of max flow.
- 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.
- In a directed graph, define a s-t cut to be a partition of the vertices into two sets, one containing the source s, the other containing the sink t. The capacity of such a cut is defined to be the sum of capacities of edges going from the source partition to the sink partition. The min cut is the minimum capacity of all s-t cuts.

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.
- We restrict ourselves to integer weights for same reason as with flow.

Residual graphs

- 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.
- Flowing along our original edges is like adding water to a pipe. Flowing along a residual edge does the opposite, it removes water from the pipe.

Residual graphs

- We define an *augmenting path* as a path from s to t as a path that travels only on edges with capacity strictly greater than current flow.
- 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.

Residual graphs

- Whenever we flow along an edge, we should also do the opposite on the residual edge. So if we increase flow by 1, we should either decrease flow by 1 on the residual or, equivalently, increase capacity by 1. In this way, the sum of slacks of both edges remains constant.
- Intuitively, this is a mechanism allowing us to modify our already chosen augmenting paths. Since we keep the sum of slacks of both edges constant, flowing using the residual edge actually removes flow from the original edge. So in our augmenting path, it represents replacing the flow in the original edge with a new source of flow.

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.

Graph Representations: Advanced Edge Lists

- Flow is one of the few times we prefer not to use adjacency lists. It is a bit of a nuisance to get the residual edge of an edge with adjacency lists.
- We will instead use a single edge list. It's a bit easier to refer to specific edges in an edge list.
- However, we still need to find all edges adjacent to a vertex in $O(\text{num edges})$. There is an elegant but not entirely straightforward solution to this, effectively involving multiple linked lists interleaved in one array.

Graph Representations: Advanced Edge Lists

- Instead of $O(V)$ lists, we will put all our edges into a single list. An edge and its residual will always be in adjacent positions, in indices $2i$ and $2i+1$. So the edge at index i has its residual at index i^1.
- In addition, we add V linked list structures over the edge list, one for each source vertex. Each edge will have a pointer to the first edge before it in the list that has the same source. For each vertex v, we store the last edge in the list with v as the source. Traversing this linked list allows us to enumerate all edges starting at v in optimal time.

Graph Representations: Advanced Edge Lists

```
// 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.
- As a summary, it just keeps finding augmenting paths and pushing flow through them until there are none left.
- 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.

Ford-Fulkerson algorithm

```cpp
// 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;
}
```

Edmonds-Karp algorithm

- The f in the time complexity of Ford-Fulkerson is not ideal, because f could be 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 $O(\min(VE^2, Ef))$.

Edmonds-Karp algorithm

- Proof (Sketch): Let G_f be the residual network after enacting flow f (so only leave edges that have more capacity than flow). Let $d_i(v)$ be the unweighted shortest distance from source s to v in G_f^i (the residual graph after step i of Edmonds-Karp).
- The main claim is that $d_i(v) \leq d_{i+1}(v)$, i.e: that Edmonds-Karp only increases distances from the source.
- You can roughly convince yourself of this by looking at what a step of Edmonds-Karp does to the BFS tree of G_f^i.
- It saturates some edges, removing them from G_f^{i+1}. It may also introduce some residue edges into G_f^{i+1}.
- But removing edges doesn't help decrease distances (obvious). And introducing residue edges doesn't either since all those residue edges go backwards in the BFS tree (intuitively believable but you probably want to induct here if you want a rigorous proof).

Edmonds-Karp algorithm

- Imagining the BFS tree also helps prove our actual main claim: that an edge can only be saturated at most $n/2$ times.
- Say an edge $e : u \to v$ is saturated in iteration i, with u being d distance from the source. Then for e to be unsaturated, there needs to be some flow through its residue.
- But since distances don't decrease, for the residue to appear in the BFS tree, at some point u's distance needs to be 1 more than v's. But v's distance is already $d + 1$ so for this to happen u's distance must be $d + 2$. Hence the next time e is saturated, $d(u)$ has increased by 2. This can occur at most $n/2$ times.
- Finally, each round of Edmonds-Karp saturates at least one edge by design. So there can be at most $O(EV)$ rounds. Each is a BFS ($O(E)$), hence running time is $O(E^2V)$.

Edmonds-Karp algorithm

```
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;
}
```

Dinic's Algorithm

- The most useful one for contests.
- Essentially just a more optimized implementation of Edmonds-Karp.
- The basic idea is the exact same, we repeatedly augment along the shortest augmenting path.
- However, we work in phases. In each phase, we augment until the shortest distance from source to sink has increased. Hence there are $O(V)$ phases.
- The key is to make the BFS tree part of the earlier proof more explicit (this is often called the level graph). The speedup comes from realising that in each phase we're working on the same BFS tree. This allows us to find each augmenting path in $O(V)$ amortized.
- Each phase has $O(E)$ augmenting paths. Hence, we get $O(V^2E)$ overall.

Dinic's Algorithm

- At the start of each phase, construct the BFS tree. Ignore all edges not in it.
- The main claim is: once there is no flow in the BFS tree, the distance from the source to sink has increased. The same ideas as the earlier proof sketch work here too.
- Now that we are working on the BFS tree, any path we find is a shortest augmenting path. So we can use a DFS to find these.

Dinic's Algorithm

- The **key** is, say we are at node u. If we DFS to child c of node u and there are no augmenting paths from c to the sink, then child c is useless and we delete child c from node u.
- So our DFS will always find an augmenting path by trying the very first child of each node, except $O(E)$ times when we mess up. Amortized out, this means we get $O(V)$ per augmenting path, with an extra **overall** cost of $O(E)$ for the times we have to delete a child.
- Hence each phase is $O(EV)$, there are at most $O(E)$ augmenting paths and each is found in $O(V)$ (the extra $O(E)$ is dwarfed since it is $O(E)$ **per** phase).
- Overall complexity: $O(\min(V^2E, Ef))$.

Dinic's Algorithm

- As an implementation note, we don't actually delete children.
- We instead use our edge list (and in particular the embedded linked list in it) as our children array.
- For each node, we store the index of the first child that we have not "deleted".

Dinic's Algorithm

```
// assumes the residual graph is constructed as in the previous section
// n = #nodes, s = source, t = sink
int n, s, t;
// stores dist from s.
int lvl[N];
// stores first non-useless child.
int nextchld[N];

// constructs the BFS tree.
// Returns if the sink is still reachable.
bool bfs() {
  for (int i = 0; i < n; i++) lvl[i] = -1;
  queue<int> q;
  q.push(s); lvl[s] = 0;
  while (!q.empty()) {
    int u = q.front(); q.pop();
    nextchld[u] = start[u]; // reset nextchld
    for (int e = start[u]; ~e; e = succ[e]) {
      if (cap[e] > 0) {
        int nxt = to[e];
        if (lvl[nxt] != -1) continue; // already seen
        lvl[nxt] = lvl[u] + 1;
        q.push(nxt);
      }
    }
  }
  return lvl[t] != -1;
}
```

Dinic's Algorithm

```
// Finds an augmenting path with up to cflow flow.
int aug(int u, int cflow) {
  if (u == t) return cflow; // base case.
  // Note the reference here! We will keep decreasing nextchld[u]
  // Till we find a child that we can flow through.
  for (int &i = nextchld[u]; i >= 0; i = succ[i]) {
    if (cap[i] > 0) {
      int nxt = to[i];
      // Ignore edges not in the BFS tree.
      if (lvl[nxt] != lvl[u] + 1) continue;
      int rf = aug(nxt, min(cflow, cap[i]));
      // Found a child we can flow through!
      if (rf > 0) {
        cap[i] -= rf;
        cap[i^1] += rf;
        return rf;
      }
    }
  }
  lvl[u]=-1;
  return 0;
}

int mf() {
  int tot = 0;
  while (bfs())
    for (int x = aug(s,INF); x; x = aug(s,INF)) tot+=x;
  return tot;
}
```

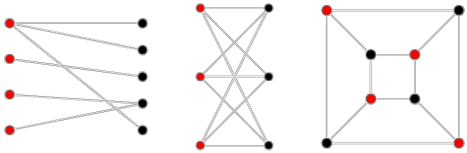
Faster flow algorithms

- Faster flow algorithms exist, but I've never actually used them. Dinic's is the best default for programming competitions.
- Augmenting path-based algorithms tend to run much faster in practice than their worst case time complexity suggests. In particular, Dinic's pretty much never behaves anything close to the worst case.
- Kind of annoying for competitions. If the graph has any structure, probably expect to easily run a $O(10^{12})$ in sub a second...
- Many flow algorithms. Some of interest: Capacity scaling $O(E^2 \log C)$ where C is the max capacity of an edge, and Highest Label Preflow Push $O(V^2\sqrt{E})$.
- Recently, it was shown that this problem can be solved in total $O(VE)$ time.

Table of Contents

Bipartite matching

- 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.


1

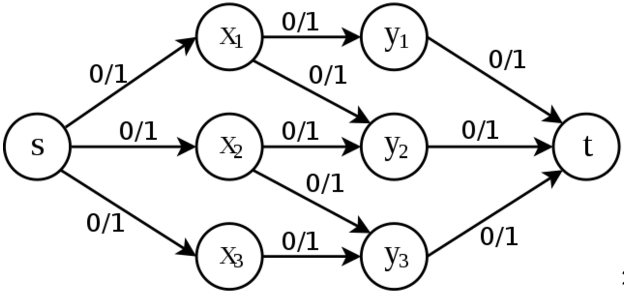
- The maximum bipartite matching problem, given a bipartite graph, is to choose the largest possible set of edges in that graph such that no one vertex is incident to more than one chosen edge.
- There is a clear flow formulation for this problem.

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

Bipartite matching

- 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.

Bipartite matching



[2]

[2] https://en.wikipedia.org/wiki/File:Maximum_bipartite_matching_to_max_flow.svg

Bipartite matching

- 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 and Edmonds-Karp run in $O(VE)$.
- Fact: For *unit graphs*, Dinic's runs in $O(E\sqrt{V})$. So Dinic's solves maximum bipartite matching in $O(E\sqrt{V})$. A unit graph is one where every non-source, non-sink node, say u, satisfies at least one of the following:
  - u has a single incoming edge and this edge has capacity one.
  - u has a single outgoing edge and this edge has capacity one.

  In other words, no vertex appears in more than one flow path. E.g: flow graphs arising from bipartite matching.

Bipartite matching

- Aside: Augmenting paths look very nice in bipartite graphs. There's the matched component and unmatched vertices and augmenting paths just alternate between following unmatched edges and matched edges.
- Also on bipartite graphs we get nice theorems like Hall's Marriage Theorem.
- For these reasons, we can say a lot more about the structure of bipartite graphs. Also certain problems on bipartite graphs are a lot easier than on general graphs.
- But won't get time to see this in this course.

Table of Contents

Min-Cut

- The dual to max flow is min cut.
- This is interesting in its own right! Frequently this shows up in unexpected places.
- Helps solve some optimal assignment problems.

Extracting the min-cut

- To get the value of the minimum cut, just run max flow.
- 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 remaining 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. (compare to the definition of a s-t cut)
- The residual edges need to be ignored!

Extracting the min-cut

```
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;
}
```

Project Selection

- One of the canonical examples of an assignment problem solvable by flow.
- **Problem Statement:** There are N projects and M machines. Each machine has a cost c_j. Each project has a revenue r_i. Each project depends on a set of machines but machines can be shared among projects. Determine a set of projects and machines that maximizes profit.
- **Input Format:** First line contains 3 integers N, M. $1 \leq N, M \leq 1000$. Next line contains N integers, the revenues. After that is a line with M integers, the costs. All such values are non-negative. Next line contains one integer, D, the number of dependencies. $0 \leq D \leq 1000$. The following D lines each contain a pair $a_i, b_j, 1 \leq a_i \leq N, 1 \leq b_j \leq M$. denoting that project a_i is dependent on b_j.

Project Selection

- **Sample Input:**

```
3 3
100 200 150
200 100 50
4
1 1
1 2
2 2
3 3
```

- **Sample Output:**

```
200
```

- **Explanation:** The optimal choice is to purchase machines 2 and 3 and select projects 2 and 3.

Project Selection

- Here we have a classical optimization problem with constraints. Let's make these explicit.
- Let P be the projects selected and Q the machines purchased. We are maximizing

$$\sum_{i \in P} r_i - \sum_{j \in Q} c_j$$

- **Useful trick:** Because we want to *minimize* cost, we want to rewrite our problem as a minimization. So let P^c be the set of projects *not* selected. Then we need to minimize:

$$\sum_{i \in P^c} r_i + \sum_{j \in Q} c_j$$

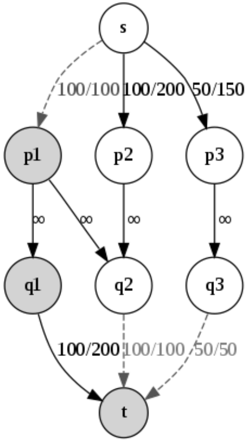
Project Selection

- Our constraints are, for all dependences i, j, $i \in P \implies j \in Q$. In terms of P^c, this can be rephrased as
$$i \in P^c \vee j \in Q$$
- We can formulate this as a min cut! We will have edges representing projects and machines, with costs r_i and c_j respectively. Cutting an edge represents adding i to P^c or j to Q.
- To formulate our constraints, for each dependence i, j we want a path from source to sink using the edge representing project i and the edge representing machine j.

Project Selection

- **Formal Construction:** We will form a biparitte graph. One bipartition represents projects, the other machines.
- Projects are connected to the source with edges representing their revenues.
- Machines are connected to the sink with edges representing their costs.
- **Useful Trick:** An edge with capacity infinity connects project i to machine j if i is dependent on j.
- The implementation is just constructing the flow network then calling your flow code.

Project Selection



[3]

[3] https://en.wikipedia.org/wiki/File:Max-flow_min-cut_project-selection.svg

Project Selection

```
// Construction:
// Nodes 0 to N-1 are the project nodes.
// Nodes N to N+M-1 are the machine nodes.
// Node N+M is the source, N+M+1 is the sink.

int main() {
    int N, M, D;
    scanf("%d %d", &N, &M);
    int totalRevenue = 0;
    int source = N+M, sink = N+M+1;
    for (int i = 0; i < N; i++) {
        int c; scanf("%d", &c);
        add_edge(source, i, c);
        totalRevenue += c;
    }
    for (int i = 0; i < M; i++) {
        int c; scanf("%d", &c);
        add_edge(i+N, sink, c);
    }
    scanf("%d", &D);
    for (int i = 0; i < D; i++) {
        int a, b; scanf("%d %d", &a, &b);
        a--; b--;
        add_edge(a, b+N, INF); // INF > all possible outputs.
    }
    printf("%d\n", totalRevenue - get_max_flow());
}
```

Project Selection

- **Summary:** It's a classic example but it shows the general gist of how many min cut problems work.
- First we rephrase our optimiztion problem into a minimization problem. Often we have to negate variables for this. We want cutting an edge to represent something with a cost.
- Next we try to phrase the constraints as cuts. Here it's important to have a good grasp on what the edges are, we are aiming to form paths from source to sink so that our constraints translate are equivalent to all paths having a cut edge.
- There are useful constructions that appear repeatedly. Adding infinite weighted edges is common.
- Often we also hope we can reduce the problem to something similar to the bipartite example here.

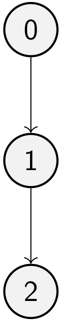
Table of Contents

Vertex capacities

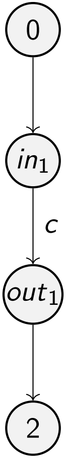
- A graph has vertex capacities if there are also capacity restrictions on how much flow can go through a vertex. (this means there's a limit to total flow incoming/outgoing from a vertex).
- This is solved by splitting each vertex into two vertices, an "in" vertex and an "out" vertex.
- For some vertex u with capacity c_u, we add an edge from in_u to out_u with capacity c_u.
- Incoming edges go to in_u and outgoing edges leave from out_u with their original capacities.
- Useful for flow problems where vertices have restrictions.
- Also useful for cut problems where costs are associated with vertices.

Vertex capacities

Original graph:



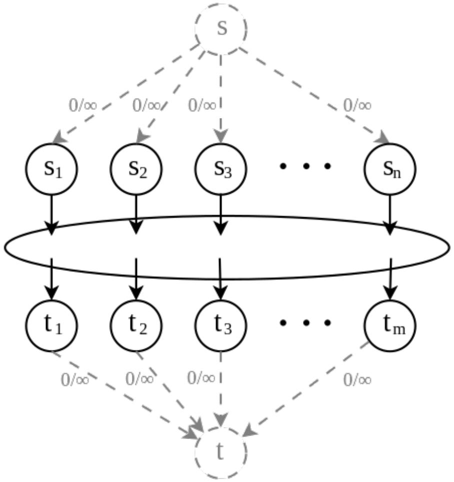
Node 1 has capacity c:



Multiple Sources and Sinks

- Sometimes a problem will naturally have multiple sources or sinks.
- Just make all the sources into regular nodes and connect them with infinite edges to a "supersource", the actual source of the flow graph.
- Same with sinks.

Multiple Sources and Sinks



Undirected Graphs

- Sometimes you want to find flow or min cut in an undirected graph.
- Just duplicate each edge, one going forwards, one going backwards.

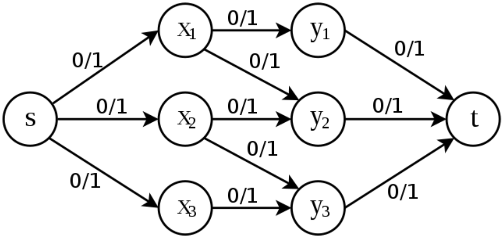
Maximum edge-disjoint paths in a graph

- 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.

Minimum vertex cover in a bipartite graph

- 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.
- Actually this is just min-cut max-flow. Using our earlier construction of a flow network for bipartite graphs, the min cut corresponds to a max vertex cover.

Bipartite matching

- 

- Edges directly connected to source and sink represent vertices. If these are cut, the corresponding vertex is in the min vertex cover.
- If the cut contains the intermediary edges, pick either endpoint to be in the min vertex cover.

Table of Contents

How to Flow?

- We just covered the basic theory of flow.
- But the truly remarkable part of flow (and especially min cut) is it appears in problems you don't expect it to!
- To me it is because flow and min cut try to solve a very general problem, either assignment or optimization under constraints.
- Both in full generality are NP-hard and encompass most problems we care about. So even being able to solve a small subclass of these is pretty remarkable.

How to Flow?

- There are 2 main parts to flow problems: recognizing they are flow and constructing the right flow graph.
- The former is difficult, but you get some intuition after a while. Flow problems tend to be unsolvable by anything except flow.
- I feel a decent approach to start with is, if the problem requires assignments with no obvious greedies then flow is worth considering. If the problem requires optimization with constraints with no obvious greedy, consider min cut.
- Constructing the right flow graph requires a lot of practice but there are common reusable constructions.
- I will try to show some useful basic examples and things to keep in mind.

Example: Magic Hours

- **Problem Statement:** Disneyland has N tourists and M attractions. Each tourist is interested in a set of attractions and will be happy if they can visit at least one. Each attraction has a limit on the number of tourists that can visit.
  In addition, certain tourists are entitled to visit during magic hours. Only some of the rides are open during magic hours. The normal limits don't apply but at most K tourists can visit during magic hours.
  What is the maximum number of tourists that can be made happy?

Example: Magic Hours

- **Input Format:** First line 3 integers, N, M, K.
  $1 \leq K, M, N \leq 1000$.
  Next line contains M integers, the limits for each of the attractions. Next line contains M boolean values. The ith is 1 iff attraction i is open during magic hours.
  The next N lines each describe a tourist. Each line is of the form `b C x1 ... xC` where b is 1 if this tourist is eligible for magic hours, 0 otherwise, C is the number of attractions the tourist is interested in and $\{x1, \ldots, xC\}$ is the set of attractions. Attractions are 1 indexed.
  There are at most 2000 tourist attraction pairs.

Example: Magic Hours

- **Sample Input:**

```
3 2 1
1 1
1 0
1 1 1
0 2 1 2
0 1 1
```

- **Sample Output:** 3
- **Explanation:** Person 2 visits attraction 2 and person 3 visits attraction 1. Person 1 visits attraction 1 during magic hours.

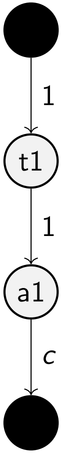
Example: Magic Hours

- Seems like an assignment problem with no obvious greedy (e.g: try $K = 0$). So flow seems suitable here.
- The task now is to translate the requirements into a flow graph.
- And until you become familiar, this step will likely be a lot of trial and error.
- So I'm just going to iterate through some examples.
- I think it is also important to be able to determine if a flow graph is correct so expect some wrong flow graphs :P

Example: Magic Hours

- Let's start with basics. What happens if there are no magic hours?
- Question is just bipartite matching with constraints on attractions.

Example: Magic Hours

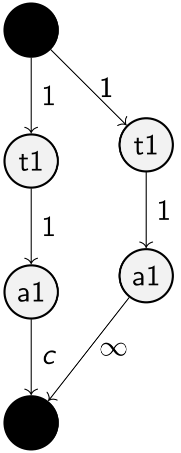


where c is capacity of the attraction.

Example: Magic Hours

- So far so good. How to handle magic hours?
- **Try 1:** Create a copy of the graph for magic hours. This graph only contains tourists that are entitled to magic hours and rides open during magic hours.

Example: Magic Hours

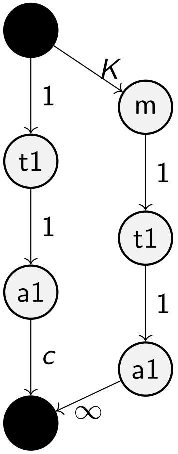


Assume t1 and a1 are both applicable to magic hours.
What's wrong?

Example: Magic Hours

- An obvious problem. We aren't accounting for K, the limit on tourists during magic hours.
- There's a standard fix here, add an extra source to the magic hours side of the graph that has a capacity of K.

Example: Magic Hours

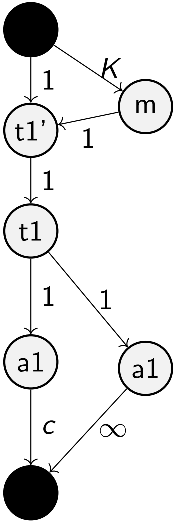


What's wrong?

Example: Magic Hours

- The same tourist might be double counted, they might visit an attraction during the day and we would count them again during magic hours.
- This is a common mistake with flow graphs, you try to add in a new constraint and in the process you invalidate an old constraint.
- So we need a single bottleneck per tourist.
- We can try to pass the magic hours source through the tourists.

Example: Magic Hours

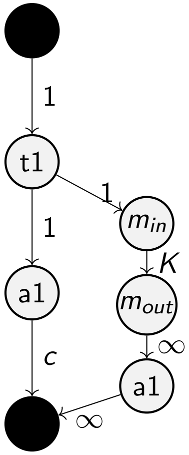


What's wrong?

Example: Magic Hours

- Many things. The magic hours source doesn't actually *do* anything. More concretely, we are allowing the tourists visiting during magic hours to visit all attractions, not just the magic hours attractions.
- Mixing up different paths is another very common mistake with flow graphs.
- Okay, let's try putting the magic hours constraint after the constraint on the tourists.

Example: Magic Hours

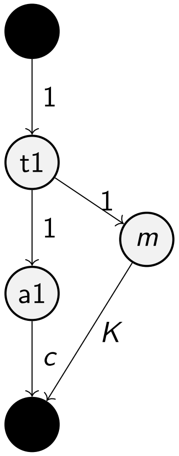


Note we split the magic hours node into 2 nodes since we have a constraint on the node itself.
What's wrong?

Example: Magic Hours

- Not obvious when there's only 1 tourist. But with many, by passing through the node for magic hours, you're allowing them to visit ANY of the attractions open during magic hours, not just the ones they are interested in seeing. How to fix?
- Note we don't actually care which attraction a tourist visits during magic hours, just that there is at least one attraction they are interested in and is open during magic hours.
- So precalculate which tourists can actually visit an attraction during magic hours.
- After that, we don't care about the attractions connected to the magic hours node. So we can connect the magic hours node directly to the sink.

Example: Magic Hours



Example: Magic Hours

```
// Construction:
// Nodes 0 to N-1 are the tourists.
// Nodes N to N+M-1 are the attractions.
// Node N+M is the magic hour, N+M+1 source, N+M+2 sink.

int main() {
    scanf("%d %d %d", &N, &M, &K);
    int magicnode = N+M, source = N+M+1, sink = N+M+2;
    for (int i = 0; i < M; i++) {
        int c; scanf("%d", &c);
        add_edge(i+N, sink, c);
    }
    vector<int> magicOpen(M);
    for (int i = 0; i < M; i++) {
        scanf("%d", &magicOpen[i]);
    }
    for (int i = 0; i < N; i++) {
        add_edge(source, i, 1);
        int b, C;
        scanf("%d %d", &b, &C);
        for (int j = 0; j < C; j++) {
            int x; scanf("%d", &x); x--;
            add_edge(i, x+N, 1);
            if (b && magicOpen[x]) add_edge(i, magicnode, 1);
        }
    }
    add_edge(magicnode, sink, K);
    // run max flow
}
```

Example: Magic Hours

- **Complexity?** $O(N + M)$ vertices, $O(N + M + A)$ edges where $A \leq 2000$ is the number of tourist attraction pairs. So about 2000 vertices and 4000 edges.
- Dinic's is $O(E^2 V)$ which directly evaluated gives about 8 billion. But the graph is so structured I would expect it to run in a second.
- Keep in mind these traps! Whenever you think you have a flow network it should be because you think each plausible flow path means something. You should check:
  - Every possible choice is represented by some flow path.
  - Every flow path corresponds to some valid choice.
- It would be a good idea to double check you understand why every edge has the capacity it has in this example.
- Only real method to learn is probably practice. When I'm doing flow, I still find myself spending most of my time iterating through flow graphs.

Example problem: Irrigation

- **Problem Statement:** My garden is a $N \times M$ grid, some squares containing flowers. As I learnt recently, flowers need water...
  To water them, I can purchase some line sprinklers. Each line sprinkler can water a single row or column. What is the minimum number of line sprinklers I need.
- **Input Format:** The first line contains 2 integers, N, M, $1 \leq N, M \leq 1000$. The following N lines each contain M bits. A 1 denotes a square with a flower.

Example problem: Irrigation

- **Sample Input:**

  ```
  3 3
  001
  111
  001
  ```

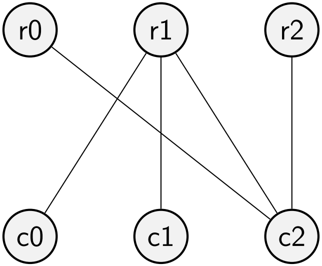
- **Sample Output:** 2

Example problem: Irrigation

- What are our constraints here?
- We have one for each flower. Each flower says we need to pick its row or its column (maybe both) In our flow formulation, it would make sense that each flower is an edge.
- So our nodes represent the rows and columns.
- **Key Observation:** This graph is naturally bipartite. If our flowers are edges, they can only connect a row node to a column node.

Example problem: Irrigation

Sample Input again:



Example problem: Irrigation

- What are we being asked for?
- Minimum set of vertices adjacent to at least one edge.
- Min vertex cover!
- We could have also solved the flow directly without realising this. Noting it is a bipartite graph is key however.

Example problem: Irrigation

```
// Construction:
// Nodes 0 to N-1 are the row vertices.
// Nodes N to N+M-1 are the column vertices.
// Node N+M is the source, N+M+1 is the sink.

int N, M;
int main() {
    scanf("%d %d", &N, &M);
    int source = N+M, sink = N+M+1;
    for (int r = 0; r < N; r++) {
        for (int c = 0; c < M; c++) {
            char inp; scanf(" %c", &inp);
            if (inp == '1') {
                add_edge(r, c+N, 1);
            }
        }
    }
    for (int r = 0; r < N; r++) {
        add_edge(source, r, 1);
    }
    for (int c = 0; c < M; c++) {
        add_edge(c+N, sink, 1);
    }
    printf("%d\n", get_max_flow());
}
```

Example problem: Irrigation

- **Complexity?** $O(N+M)$ vertices, $O(NM)$ edges, with Dinic's it is $O(NM\sqrt{N+M})$, about 30 mil. More than fast enough, especially for a flow problem.
- How could you get the feeling this might involve flow?
- You're roughly being asked to minimize a value, subject to some constraints.
- If you play with it a bit, you'll realize a greedy solution is hard. So the next natural thing to try is flow.
- Also the graph is very naturally bipartite.

Aside: Common Bipartite Graphs

- It is useful to be able to recognize if a problem involves a bipartite graph quickly.
- A sometimes useful characterization: Bipartite graphs are graphs with no odd cycles.
- Also a graph is bipartite iff it can be 2-colored. This allows you to check bipartiteness in $O(V + E)$ with a DFS.
- Common examples:
  - Any time you naturally have two different kinds of nodes.
  - On grids, 4-adjacency forms a bipartite graph. Why?
  - For same reason, so does knight jumps...
  - The rows and columns of a grid.

Example problem: Jumping Frogs

- **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.
- Freddy's journey will consist of a series of jumps. Can you find a path that minimizes the longest jump that Freddy will perform?

Example problem: Jumping Frogs

- 2 constraints here, we both have a constraint on maximum distance and constraints on paths.
- Want to shave a constraint, it will be annoying to have to deal with both at the same time. How?
- **Binary search!**
- We can binary search for the smallest maximum edge weight. So from now on we have a decision problem:
- Given we can jump up to M distance, is there a path there and back for Freddy Frog?

Example problem: Jumping Frogs

- 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.
- **Observation 1:** 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. Edges with infinite capacities connect 2 stones that are within distance M.
- But this formulation will only allow us to visit any of the "big" vertices once.

Example problem: Jumping Frogs

- 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.
- As always, pick infinite to be larger than any necessary finite capacity.

Example problem: Jumping Frogs

- We need $O(\log DE)$ (where E depends on required precision) iterations of our binary search to find this maximum edge weight. We can do a max-flow computation in $O(Ef) = O(V^2)$, because we can terminate the max-flow once f hits 2.
- The total runtime is $O(V^2 \log DE)$.
- **Implementation**

```
double lo = 0, hi = 1e10;
for (int it = 0; it < 70; it++) {
  double mid = (lo + hi) / 2;
  if (cando(mid)) hi = mid;
  else lo = mid;
}
printf("%lf\n", lo);
```

Example problem: Jumping Frogs

```
// Each stone is represented by 2 vertices, (0,1), (2, 3), ...
// Source is 2N, fake sink is 2N+1. real sink is 2N+2.
// Edge of cap 2 from fake to real sink to limit max flow to 2.
double sqdist(double x1, double y1, double x2, double y2) {
    return (x1-x2)*(x1-x2) + (y1-y2)*(y1-y2);
}

bool cando(double maxdist) {
    if (D < maxdist) return true;
    int source = 2*N;
    int fakesink = 2*N+1;
    int realsink = 2*N+2;
    for (int i = 0; i < N; i++) {
        int cap = isbig[i] ? INF : 1;
        add_edge(2*i, 2*i+1, cap);
    }
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            if (sqdist(x[i], y[i], x[j], y[j]) < maxdist*maxdist)
                add_edge(2*i+1, 2*j, INF);
        }
    }
    for (int i = 0; i < N; i++) {
        if (x[i] < maxdist) add_edge(source, 2*i, INF);
        if (x[i] + maxdist > D) add_edge(2*i+1, fakesink, INF);
    }
    add_edge(fakesink, realsink, 2);
    return get_max_flow() == 2;
}
```

Example: Image Segmentation

- **Problem Statement:** I have an image made of N pixels. Each can be either in the background or foreground. For the ith pixel, you get f_i points if it is in the foreground, b_i points if it is in the background.
  Furthermore, there are M pairs of pixels that you would prefer to have the same assignment. For the kth pair, you pay a penalty of p_k if pixel a_k has a different assignment to pixel b_k.
  Maximize points - penalties.
- **Input Format:** First line, 2 integers `N M`. $1 \leq N, M \leq 1000$. Next line contains N integers, the values f_i. Next line contains N integers, the values b_i. Next M lines each contain a triplet, `ak bk pk`, describing one of the penalty pairs.

Example: Image Segmentation

- **Sample Input:**

```
3 2
5 1 3
1 5 2
1 2 1
2 3 5
```

- **Sample Output:** 11
- **Explanation:** Assign the pixels as (foreground, background, background). You pay a penalty of 1 because the first 2 pixels have different assignments.

Example: Image Segmentation

- Optimization problem with constraints and no obvious greedy. Reasonable to try min cut.
- We need to change this into a minimization. Write out what we are maximizing.

Example: Image Segmentation

- Let F be the set of pixels in the foreground and B be background. Say that $a_i = 1$ iff pixel i is in the foreground.
- Let p_{ij} be the penalty if pixel i has a different assignment to pixel j. This is 0 if there is no constraint involving pixels i and j
- Then we are maximizing

$$\sum_{i \in F} f_i + \sum_{i \in B} b_i - \sum_{a_i \neq a_j} p_{ij}$$

- This is equivalent to minimizing (S^c is the complement of S):

$$\sum_{i \in F^c} f_i + \sum_{i \in B^c} b_i + \sum_{a_i \neq a_j} p_{ij}$$

Example: Image Segmentation

- $$\sum_{i \in F^c} f_i + \sum_{i \in B^c} b_i + \sum_{a_i \neq a_j} p_{ij}$$
- This suggests what some of our edges should be and what cutting edges should mean.
- For each pixel i, we need to have an edge representing pixel i is in the foreground.
- It should have cost f_i and cutting it represents that pixel i is not in the foreground.
- Same with an edge representing the background.
- Call the first edge $foreground_i$ and the second edge $background_i$.
- We need an edge, say e_{ij} representing each penalty cost too. Cutting this edge represents that pixels i and j have different assignments.

Example: Image Segmentation

- What are our constraints?
- For each of the M penalties:

$$(\text{cut } e_{ij}) \vee (i \in F \iff j \in F)$$

- We can unroll this into two constraints involving just or statements.

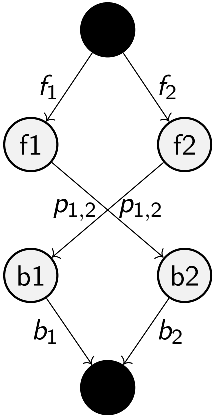
$$(\text{cut } e_{ij}) \vee (i \in F^c) \vee (j \in B^c)$$
$$(\text{cut } e_{ij}) \vee (i \in B^c) \vee (j \in F^c)$$

- Since we are defining $(i \in F^c)$ to be the same as cutting $foreground_i$ we can rewrite the first condition as:

$$(\text{cut } e_{ij}) \vee (\text{cut } foreground_i) \vee (\text{cut } background_j)$$

- This suggests what our paths should be.

Example: Image Segmentation

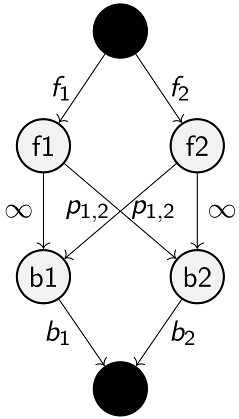


What's wrong?

Example: Image Segmentation

- Not every cut corresponds to a meaningful assignment.
- There's no guarantee we cut either $\textit{foreground}_1$ or $\textit{background}_1$. If we don't, this means pixel 1 is in both the foreground and background???
- So we can try to fix this by making a path that includes both $\textit{foreground}_1$ and $\textit{background}_1$. Here we apply the standard trick of introducing an ∞ edge.

Example: Image Segmentation



What's wrong?

Example: Image Segmentation

- Now at least one of *foreground*$_1$ or *background*$_1$ is cut. But both can be???
- This corresponds to having pixel 1 in neither the foreground nor background... Less obviously an issue but one can come up with a breaking case.
- There are 2 solutions to this problem, each showcasing a worthwhile construction.

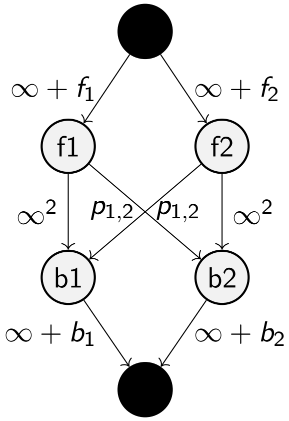
Solution 1

- The first is: to fix this construction, we need to enforce we cut exactly one of $foreground_i$ or $background_i$. Enforcing exactly one constraints is hard with min cut in general.
- Instead, we will force the constraint, we cut exactly N of the edges in

$$\{foreground_i\}_{i=1}^{N} \cup \{background_i\}_{i=1}^{N}$$

- Then since the earlier construction forces us to cut at least one of $foreground_i$ or $background_i$ this is equivalent to cutting exactly one of the 2 for each i.
- We enforce this by artificially inflating the flows of the edges $foreground_i$ and $background_i$ so that no optimal solution would cut more than the bare minimum necessary, which is N.

Solution 1



Pick ∞ so that $\infty > \sum_{i,j} p_{i,j}$.
Pick ∞^2 so that $\infty^2 > N \cdot \infty + \sum_i f_i$.

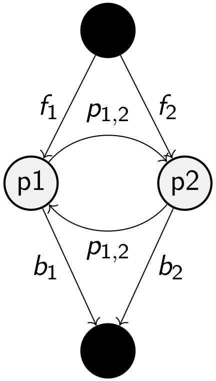
Solution 1

- Then, our choices of ∞ guarantee we never cut more than N edges with an ∞ (it's better to just cut all the $p_{i,j}$ edges instead) and never cut an edge with an ∞^2 (it's better to just cut all the $\infty + f_i$ edges).
- Our min cut will be of the form $N \cdot \infty + A$ where A is the actual answer.
- Worth noting: we introduced 2 edges for each $p_{i,j}$. This should give you pause but is okay here because no min cut will cut both.
- **Exercise:** Implement this.

Solution 2

- The alternative solution uses a slightly different construction and thinks of cutting as more a way of partitioning nodes. (compare to definition of a s-t cut)
- It turns out the problem here is actually that we separate each pixel into 2 nodes.
- If we instead don't, everything just happens to work out.

Solution 2



Solution 2

- This works because a min cut will partition the graph into three parts, nodes connected to source, nodes connected to sink, nodes connected to neither.
- If pixel i is connected to the source, then we must have $background_i$ cut. Furthermore, in a min cut $foreground_i$ won't be cut. **Why?**
- Vice versa for if pixel i is connected to the sink.
- Also, in a min cut, no pixels will be disconnected from both. **Why?**
- If pixel i is disconnected from both, then we can choose not to cut $foreground_i$ and get a better cut.
- So in a min cut, we will only cut exactly one of $foreground_i$ or $background_i$ for each i.
- And we will still cut the right $p_{i,j}$. Again, note it was okay to duplicate each of the $p_{i,j}$ edges.
- **Exercise:** Implement this.

Example: Image Segmentation

- **Complexity?** Both have $O(N)$ vertices and $O(N+M)$ edges, so Dinic's runs in worst case $O((N+M)^2 N)$.
- Like Project Selection, Image Segmentation is a useful subproblem to keep in mind.
- It is useful when you need to assign truth values to variables. The main pitfall is not assigning exactly one truth value to each variable and either of the solutions here circumvents this issue.
- Adding infinity to edges is also a generally useful gadget for controlling the number of edges cut. E.g: find the min cut that also cuts the minimum number of edges.