P THINK FAST S

Competitive Programming

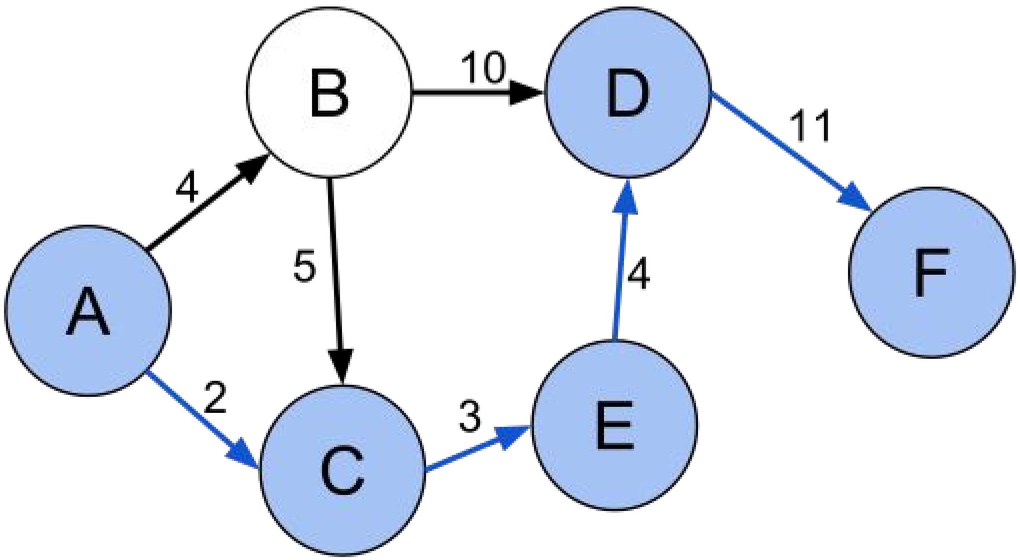
From Problem 2 Solution in O(1)

Graph Theory

Min Cost Max Flow using SSP

Mostafa Saad Ibrahim
PhD Student @ Simon Fraser University

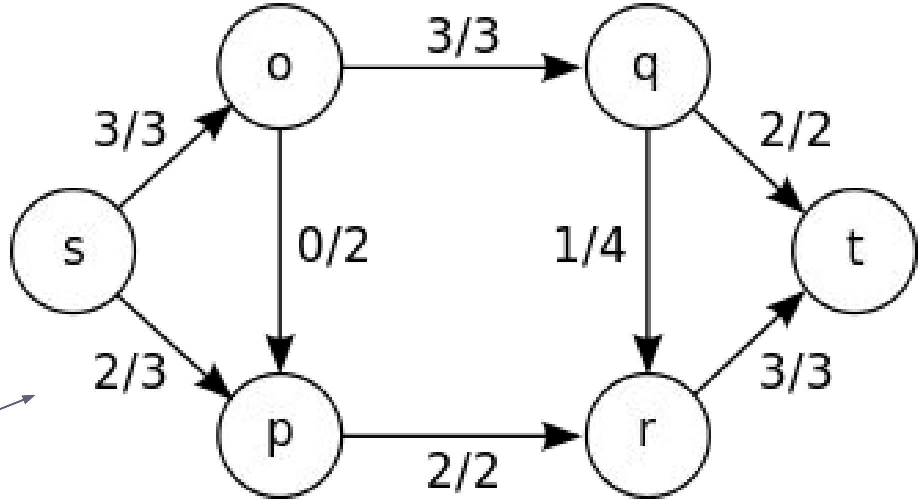
Recall: Shortest Path



Path cost: 2 + 3 + 4 + 11

Src: https://en.wikipedia.org/wiki/Shortest_path_problem

Recall: Max Flow



Edge capacity = 3
optimal solution, send 2 units through this edge

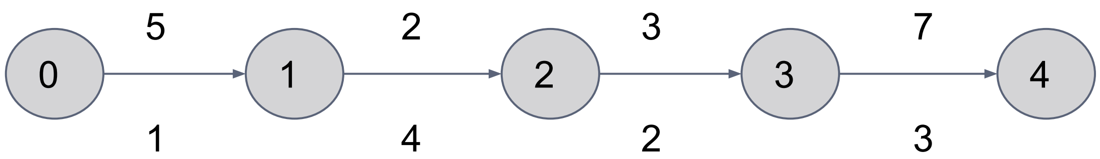
Input flow for r = 3 units ⇒ same as output

Src: https://en.wikipedia.org/wiki/Maximum_flow_problem

Path Flow & Cost

Capacity

Cost

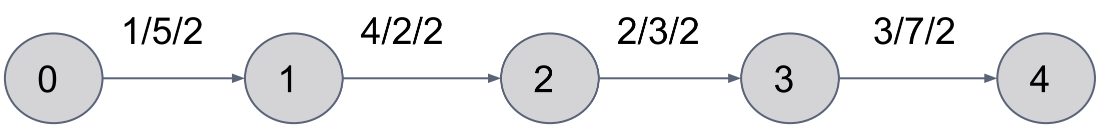


Flow cap = min(5, 2, 3, 7) = 2

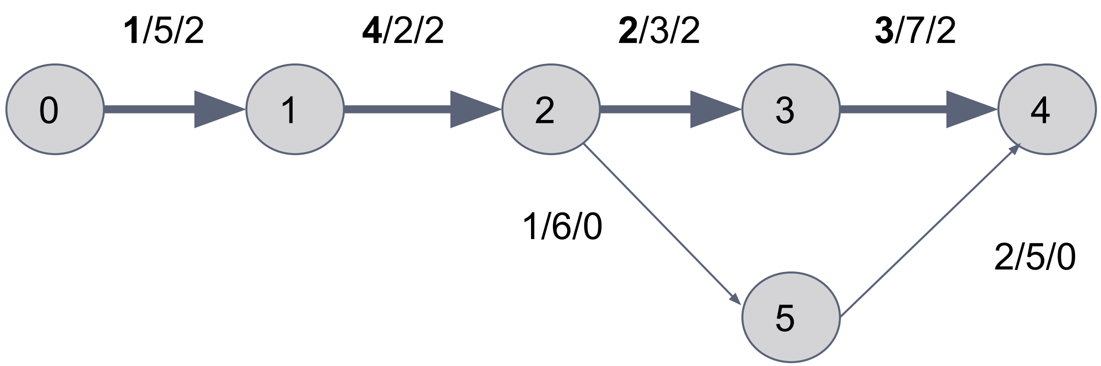
Path cost = 1+4+2+3 = 10

Flow cost = ∑ edge cost * flow = 2 * 10 = 20

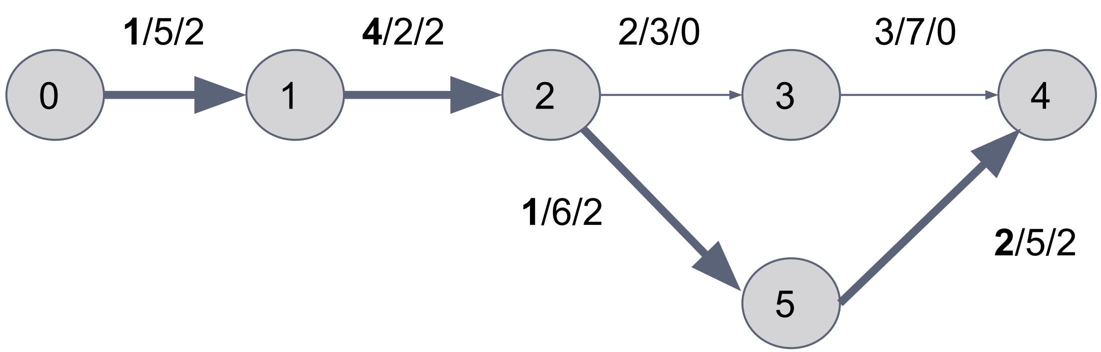
Notation: edge cost / edge cap / edge flow



Multiple Max-Flow solutions!



Max Flow = 2, Cost = 2***10** = 20



Max Flow = 2, Cost = 2***8** = 16

Set all costs = 0
⇒ Max Flow Problem

Remove Capacity Constraint
⇒ Shortest Path Problem

Min Cost Max Flow

- Among different **max** flow solutions, select one with min cost
- E.g. **first** criteria **max flow**, second min cost

Successive shortest path algorithm

- Generalization of Ford–Fulkerson algorithm
- Instead of finding any path, find **shortest path**
- So keep finding optimal flow, but one with shortest value, hence lowest flow cost
- $O(n^2mB)$ using bellmanford
  - B is assigned to an upper bound on the largest supply of any node
  - Efficient Dijkstra with potentials:
  - O(m*log(m) * min(flow, n*flow_cost))

Code consideration

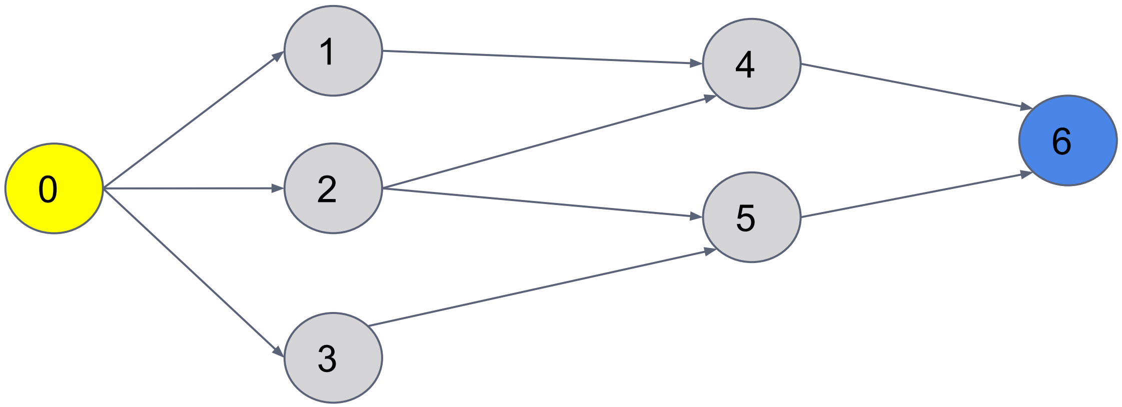
- Recall, augmenting path can partially go with original edge (i, j) or cancel flow in edge (j, i)
  - So cost of reverse edge need to be -ve
- For input edge(i, j): Flow = f, Cost = c
  - cap[i][j] = f, cost[i][j] = c, cost[j][i] = - cost[i][j]
- Then, every edge causes a cycle in cost matrix, but its sum is zero (e.g. no negative cycle)
  - However, this cycle doesn't exist in FIRST iteration as the flow in back edges is zero [Dijsktra with potentials]
- We need bellman ford algorithm (-ve values)

Nodes indexing

- Same experience as max flow
- From graph to another, you may need to index
- For simplicity, assume following graph
  - Src node, sink node
  - R starting nodes, C ending nodes
  - Src connects to R nodes, C nodes connect to sink
  - Total: R+C+2 nodes
  - Let src = 0, sink = r+c+1
  - R nodes indexed 1 to r
  - C nodes indexed r+1, r+c

Nodes indexing

- Let R = 3, C = 2, 5 edges between them



Flow values

- Some flow values will be given, others won’t
- Src/Sink edges may have value OO
- Intermediate edges typically have flow = 1
- Overall
  - Draw graph and index it based on problem nature
  - Think in missing flow values
  - Usually either OO or some constant
- When computing shortest path
  - Ignore any edge with **capacity = 0**

MCMF

- Code
  - As same as Ford code, just shortest path / cost added
- while(true)
  - Get **shortest path** from (src to dest)
    - Use only edges that has capacity > 0
  - If no path = done
  - Compute path flow (min value)
  - Compute path cost: flow value * shortest path value
  - Augment path (-flow, +flow)

MCMF

```
pair<int, int> mcmf(vector< vi > capMax, vector< vector<int> > & costMax,
        int src, int dest)
{
    int maxFlow = 0, minCost = 0;
    while(true) {
        vector<edge> edgeList;

        repa(capMax) if(capMax[i][j] > 0)
            edgeList.push_back( edge(i, j, costMax[i][j]) );

        // return 2 vectors: first distances, 2nd previous array for paths
        pair<vi, vi> p = BellmanFord(edgeList, sz(capMax), src, dest);
        if(p.first[dest] >= +OO)
            break;  // no more paths

        int bottleNeck = OO;    // get path flow
        lp(i, sz(p.second)-1 ) {
            int f = p.second[i], t = p.second[i+1];
            bottleNeck = min(bottleNeck, capMax[f][t]);
        }

        lp(i, sz(p.second)-1 ) {    // augument path
            int f = p.second[i], t = p.second[i+1];
            minCost += bottleNeck * costMax[f][t];
            capMax[f][t] -= bottleNeck, capMax[t][f] += bottleNeck;
        }
        maxFlow += bottleNeck;
    }
    return make_pair(maxFlow, minCost);
}
```

MCMF

```
lp(i, r)
    cin>>cap[0][i+1];
lp(j, c)
    cin>>cap[j+r+1][r+c+1];

int m;
cin>>m;

lp(k, m) {
    int i, j, cost;
    cin>>i>>j>>cost
    // Flow could be 1, 2...or could be given
    cap[i+1][j+r+1] = OO;
    cost[i+1][j+r+1] = v, cost1[j+r+1][i+1] = -cost[i+1][j+r+1];
}
```

Min cost bipartite matching

- Assignment problem

Persons	Job J1	J2	J3	J4
I	86	78	62	81
II	55	79	65	60
III	72	65	63	80
IV	86	70	65	71
V	72	70	71	60

Src: http://statistics-assignment.com/wp-content/uploads/2012/10/1285.png

Min cost bipartite matching

- Recall bipartite matching?
  - Once solved it by reducing it to max flow
  - Once solved it based on bipartite graph nature
- Minimum cost bipartite matching
  - Very similar style
  - Can be solved by reducing to MCMF .. just construct graph
  - Or solve based on bipartite graph nature (Hungarian)
- Hungarian algorithm solves it in O(N^3)
  - Link, Russian site - explain/code, Other code

Max Cost Max Flow

- And Max cost bipartite matching
- Recall that graph has no -ve cycles
- Hence, just multiple all costs with -1
- Compute cost, and multiply -1
  - E.g. Graph Costs *= -1
  - ComputeMCMF $\Rightarrow$ Flow = 10, cost = -20
  - cost *= -1 $\Rightarrow$ cost = 20 = Max Cost Max Flow

Reading

- Topcoder has 3 parts to read
- Bellman TLE?
  - Dijkstra with potentials. Code. Very special case, as 1st iteration has no -ve values. In next iterations, we can control with potentials arrays
  - **SPFA**. Same worst complexity, but generally faster. Code.
- Your TODOs: Min-cost circulation
- More readings:
  - Link 1. Link 2.
  - Cycle-Canceling Algorithm

تم بحمد الله

علمكم الله ما ينفعكم

ونفعكم بما تعلمتم

وزادكم علماً