Graphs (part 2/2)

Competitive programming

Prof. Rhadamés Carmona

Content

- BFS
- Floyd-Warshall
- Nwtwork Flow
- Bipartite matching
- Other graph problems

BFS

- **Breadth First Search:** A queue is used to insert discovered vertices.
- The algorithm starts by visiting and inserting one vertex into the queue.
- At each step the head is removed from the queue, and its adjacent unvisited vertices are visited and enqueued. This ensures a level-by-level path from the initial vertex.

BFS

We may visit the node when it is queued or dequeued; depends!

- **Breadth First Search, o búsqueda en anchura**:

```
void BFS(graph &g, int k) {
    for(int i = 0; i < g.size(); i++)
        g[i].visited = -1;
    list<int> q; // the queue
    // Mark as visited and enqueue it
    g[k].visited = 1; q.push_back(s);
    while(!q.empty()) {
        // Dequeue
        k = q.front();
        q.pop_front();
        for (edge *p = g[k].adj; p; p=p->next)
            if (g[p->v]->visited == -1) {
                g[p->v]->visited = 1;
                q.push_back(p->v);
            }
    }
}
```

BFS

- **Breadth First Search**, some problems/algorithms:
- Shortest path and minimum spanning tree.
- On social networks, search for friends, friends of friends, and successively up to N levels.
- Broadcast on a real network from a node.
- Network flow, the Ford–Fulkerson algorithm.
- Check if a network is bipartite.
- Prim's minimum spanning tree.
- Disjkstra.

BFS

- **Spanning tree:** or minimal spanning tree for undirected and weighted graphs (in arcs), is a connected subgraph without cycles that includes all vertices.
- **Minimum spanning tree (MST):** This is the spanning tree of minimum sum of arcs. It might not be unique. Applications: in communications networks, hydraulics, air and land routes, biological networks, etc. We'll assume connected graphs.
- We will study **Prim and Kruskal**

BFS

- **Prim's algorithm**: is a voracious or greedy algorithm; starting from any vertex, it adds one arc at a time to the tree. Selects the minimum weight arc that joins a vertex of the tree with a vertex that is not in the tree.

BFS

- **Prim's algorithm:** How to quickly find the candidate arc?. Each time an arc a (and the end vertex v) is selected, v is marked as visited, and the arcs (v,w) of ady(v) are added to the priority queue (ordered by weight). Some arcs might contain both marked vertices. These will be ignored by the algorithm (lazy priority queue update).

```
struct vertex {
        edge *ady;
        bool visited = false;
};
```

BFS

```
struct edge {
        int  left, right;
        float weight;
        edge *next;
};
```

- **Prim's algorithm**:

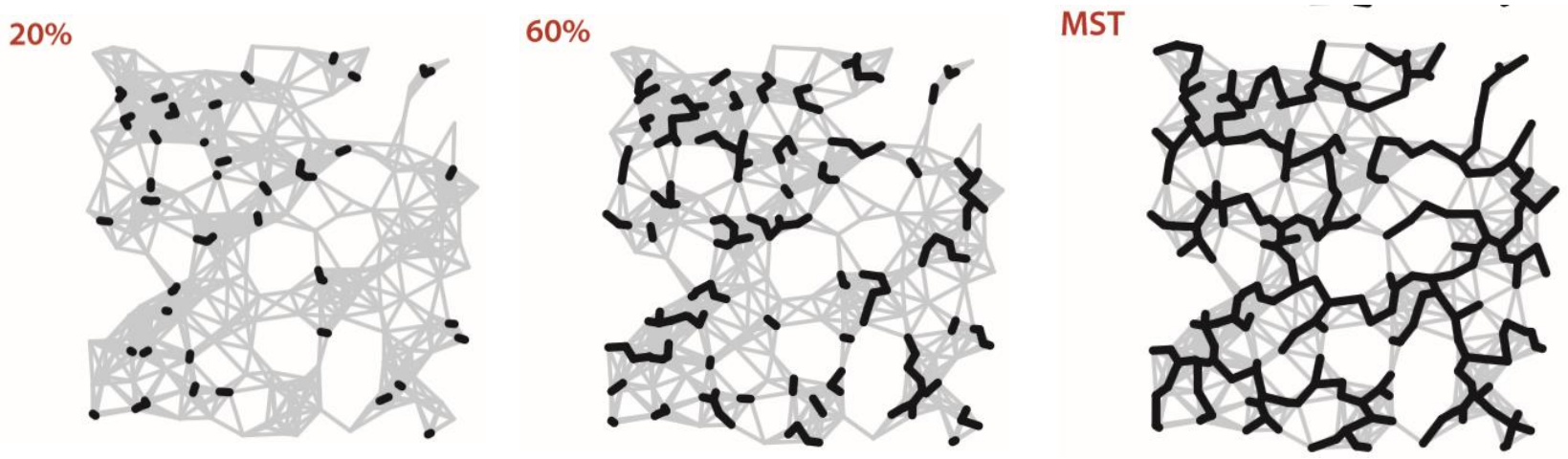
```
void LazyPrimMST(graph &g, list<edge *> &mst)   {       // O(m log n)
        PriorityQueue<edge*> pq;  // sorted ascendantly by edge::weight
        mst.clear();
        visit(g, 0, pq);
        while (!pq.empty())     {
           edge *e = pq.deleteMin();  // get lowest-weight
           if (g[e->left].visited && g[e->right].visited) continue;  //lazy
           mst.push_back(e);
           if (!g[e->left].visited) visit(g, e->left, pq);      // add vertex to tree
           if (!g[e->right].visited) visit(g, e->right, pq);  // the same
        }
}
void visit(graph g, int v, PriorityQueue<edge *> pq)   {
   g[v].visited = true;
   for (edge *p = g[v].adj ; p != NULL; p=p->next)
      if ( ! g[p->right].visited) pq.insert(e);
}
```

BFS

- **Kruskal algorithm:** in this case, all arcs are simply sorted ascending by weight.
- Starting from a forest of trees (each tree is just one vertex at the beginning), these arcs are added one by one if they do not form a cycle, until V arcs (V-1 vertices) are completed.
- Each added arc "joins" two trees in the set.

BFS

- **Kruskal algorithm**:



BFS

```
struct edge {
        int  left, right;
        float weight;
        edge *next;
};
```

- **Kruskal algorithm**:

```
Void KruskalMST(graph &g, queue<edge*> &mst)   {
  PriorityQueue<edge *> pq;  // heaps de mínimos
  insertAllEdges(pq, g);          // sorted by weight; O(m.logm)
  disjointSet <int>ds;            // yeah, disjoint set
  while (!pq.empty() && mst.size() < g.size()-1)     {// O(m)
          edge *e = pq.deleteMin();                  // O(logm)
          if (ds.root(e->left) != NULL &&
              ds.root(e->left) == ds.root(e->right)) continue;
          ds.unionFind(e->left, e->right, true);   // Merge
          mst.push_back(e);
  }
}
```

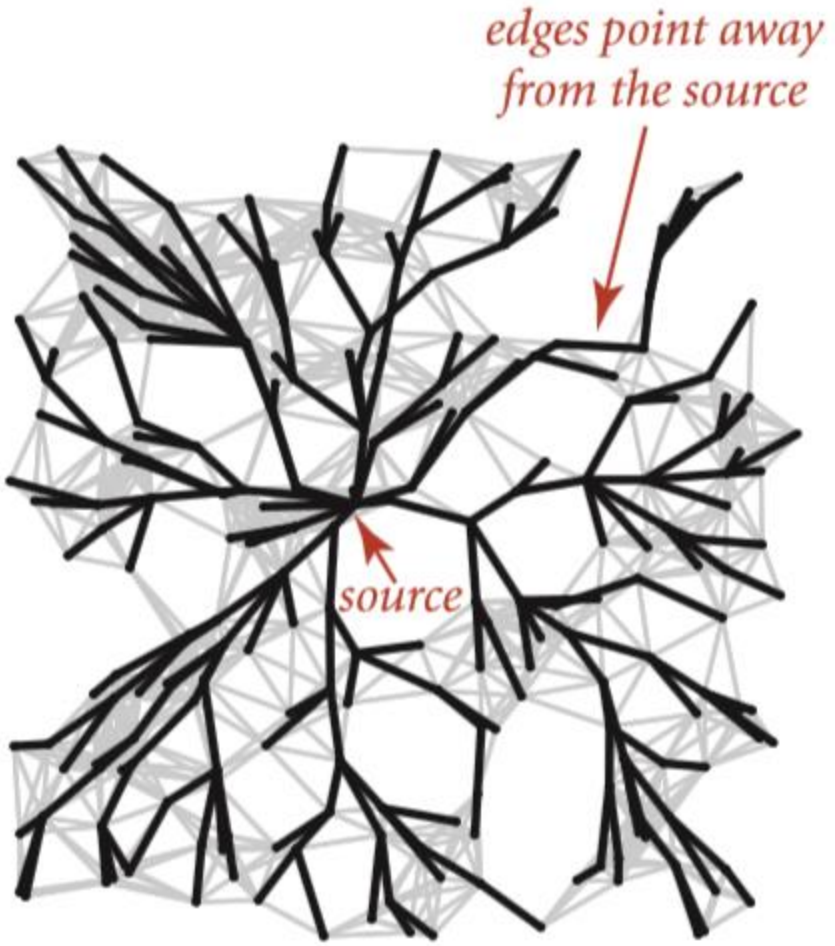
Disjoint set operations are O*(1)
Then, Kruskal is O(m.logm), with m = |E|

BFS

- **Shortest path:** Find the shortest path between a pair of vertices.
- Let's assume it's a directed graph, but it also applies to undirected graphs.
- The weight of the arc can mean time, length, cost, etc. In any case, it is something that you want to minimize. We will initially assume weights >= 0.

BFS

- **Shortest path**: a shortest path tree is usually constructed; from a vertex s, a subgraph containing a s is generated, and specifically a rooted tree in s, where each path from s to any vertex of the tree is the shortest in the original graph. Then you can do many queries from s.

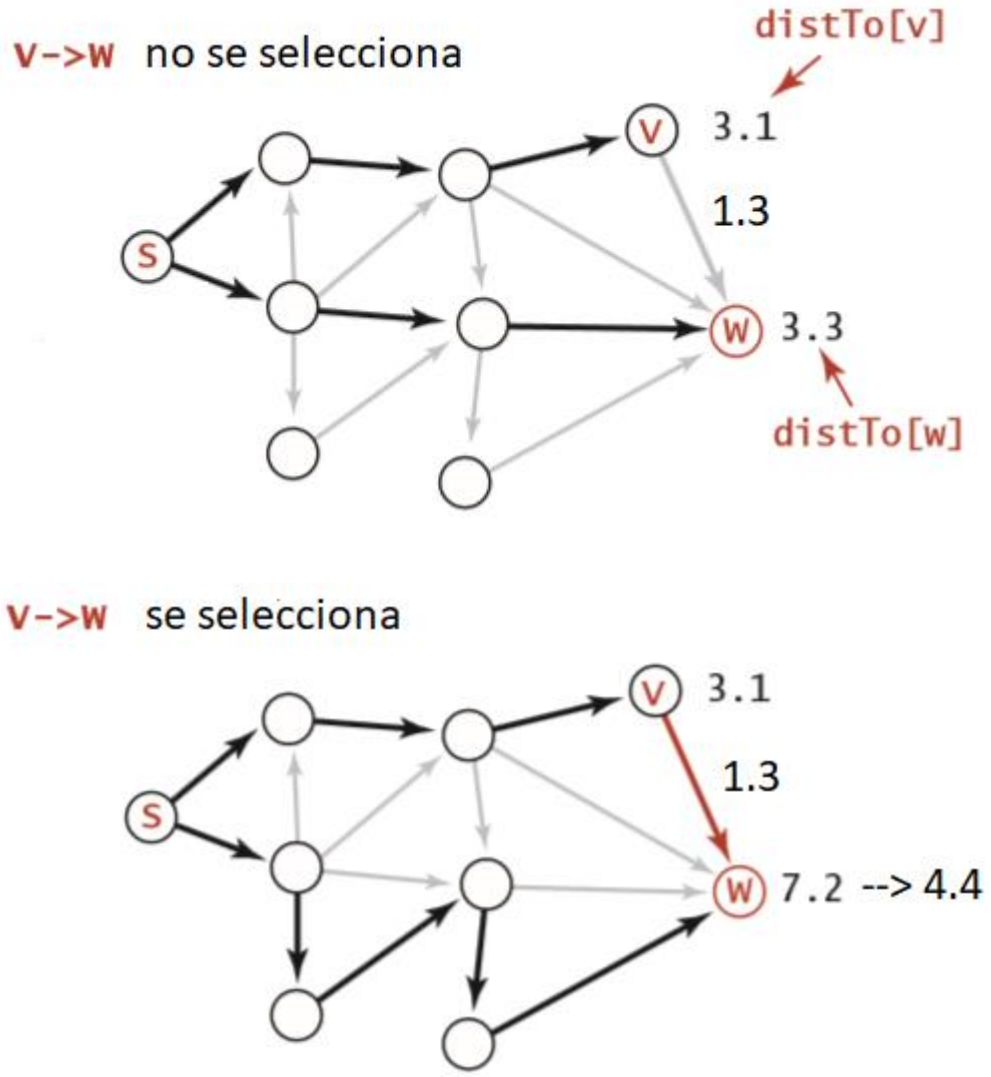


BFS

- **Shortest path:**
- Let edgeTo be an array of n integers (indexes to vertices), such that edgeTo[v] contains the index to the parent of v in the tree.
- Let distTo be an array of n floating point numbers, such that distTo[v] is the minimum distance from s to v.
- distTo[s] = 0.
- Initially, distTo[v] = ∞ ($v \neq s$), and edgeTo[v]=1 for every vertex v.
- There are 2 classic algorithms: edge relaxation and vertex relaxation.

BFS

- **Shortest path (edge relaxation):** On each step a side (v,w) is considered. When processing this side you simply check if it is shorter to go through the arc (v,w) or not?.



BFS

- **Shortest path (edge relaxation)**: an Edge is relaxed

```
vector<edge *> edgeTo;
vector<float> distTo;

void relax(edge *e) {
        int v = e->left;
        int w = e->right;
        if (distTo[w] > distTo[v] + e->weight)   {
                distTo[w] = distTo[v] + e->weight;
                edgeTo[w] = e;
        }
}
```

BFS

- **Shortest path (vertex relaxation)**: a vertex is relaxed by relaxing all its edges:

```
vector<edge *> edgeTo;
vector<float> distTo;

void relax(digraph &g, int v) {
   for (edge *p = g[v]->ady; p; p=p->next)   {
      int w = p->right;
      if (distTo[w] > distTo[v] + p->weight)  {
         distTo[w] = distTo[v] + p->weight;
         edgeTo[w] = v;
      }
   }
}
```

BFS

- **Shortest path (Dijkstra):** similar to Prim's algorithm. The difference is that Prim selects the minimum weight arc that joins a vertex of the tree with a vertex that is not in the tree. Dijkstra instead adds the nearest vertex in distance to the "source" s.
- We start by queueing (in a priority queue) the vertex s with distance 0. Then we extract the first vertex of the tail, "relax" the vertex, queueing the adjacent unvisited vertices (or updating its weight if visited).

BFS

- **Shortest path (Dijkstra)**:

```
void Dijkstra(digraph &g, int s) {
        for (int u=0; u<g.size(); u++) {
                distTo[u] = INFINITO ; edgeTo[u] = NULL; g[u].visited = false;
        }
        distTo[s] = 0; queue <pair<int, double>> q;
        q.insert(make_pair<s, distTo[s]>);
        while (! queue.empty()) {
           int u = q.min().index; q.pop();
           g[u].visited= true;
           for (edge *e = g[u].ady; e; e = e->next) /*if (!g[e->right].visited)*/ {
                v = e->right;
                if (distTo[v] > distTo [u] + e->weight) {
                        distTo [v] = distTo [u] + e->weight;
                        edgeTo [v] = u;
                        q.insert(make_pair<v, distTo[v]>);
}       }  }    }
```

BFS

- **Shortest path (Dijkstra)**:
  - O(*m*log*n*) using priority queue, as each arc (m arcs) is inserted into a priority queue with at most *n* vertices (log*n* per insertion)
  - There is an O(*n*log*n* + *m*) implementation using fibonacci heap

Floyd-Warshal

- **Floyd-Warshall**: the goal is to have a structure that gives us the shortest path between any pair of nodes. Used for multiple queries (shortest path between 2 vertices) on same graph.
- This is equivalent to transitive closure, but storing as weight the distance between the two vertices.
- This algorithm is n^3 if we use the adjacency matrix. It is based on the idea of edge relaxation.

Floyd-Warshal

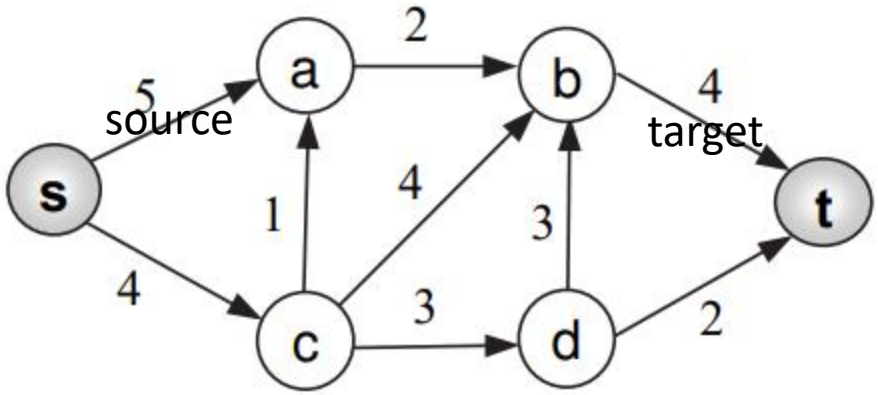
- **Floyd-Warshall**:

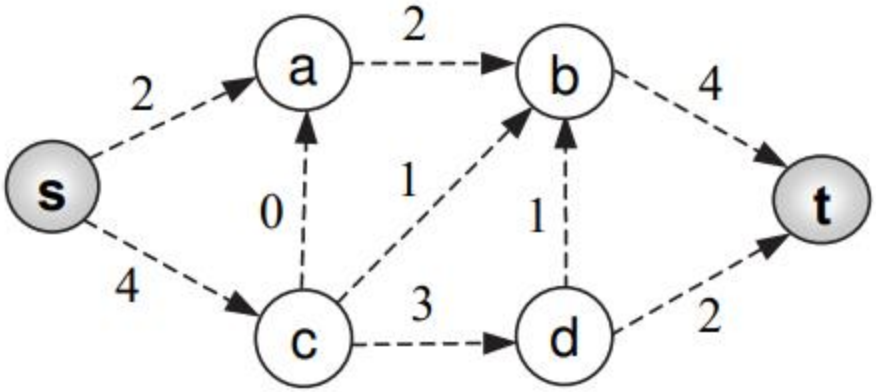
```
adjacency matrix. If the arc doesn't exist, then it's infinite
it is further assumed that ady[i][i] = 0 (zero diagonal)
Returns an array with minimum distances
void floydWarshall(int ady[n][n], int path[n][n]) {
          memcpy(path, ady, n*n*sizeof(int));
          for (int k = 0; k < n; k++)
                    for (int i = 0; i < n; i++)
                              for (int j = 0; j < n; j++){
                                        int d= path[i][k] + path[k][j];
                                        if (path[i][j] > d)
                                            path[i][j] = d;
                              }
}
```

What is better, n Dijkstras, o one Floyd-Warshall?

Maximum network flow

- **Maximum network flow:** The problem with maximum flow is to calculate the maximum amount of flow that can be transported from a starting point or origin (source or fountain) to an arrival or destination point (target or water well).





Maximum network flow

- **Maximum network flow:**
  - There are two special vertices: source and well
  - The weight of the arc represents the capacity of the arc. It cannot be exceeded
  - The final value of the flow is the total flow that comes out of the source, and equivalently the one that reaches the well
  - Capacity can be thought of as amount of water in pipe, bandwidth, etc.

Maximum network flow

- Naive solution: Each iteration checks for a path in the network (with capacity > 0 per arc) from the source to the well.
- If the path exists, increases the flow at each arc of the road in the resulting network as possible, while reducing the capacity of the arcs involved in the residual network.
- It is required to trace the capacity used (resulting network) and the remaining capacity.
- The process ends when there is no other path.

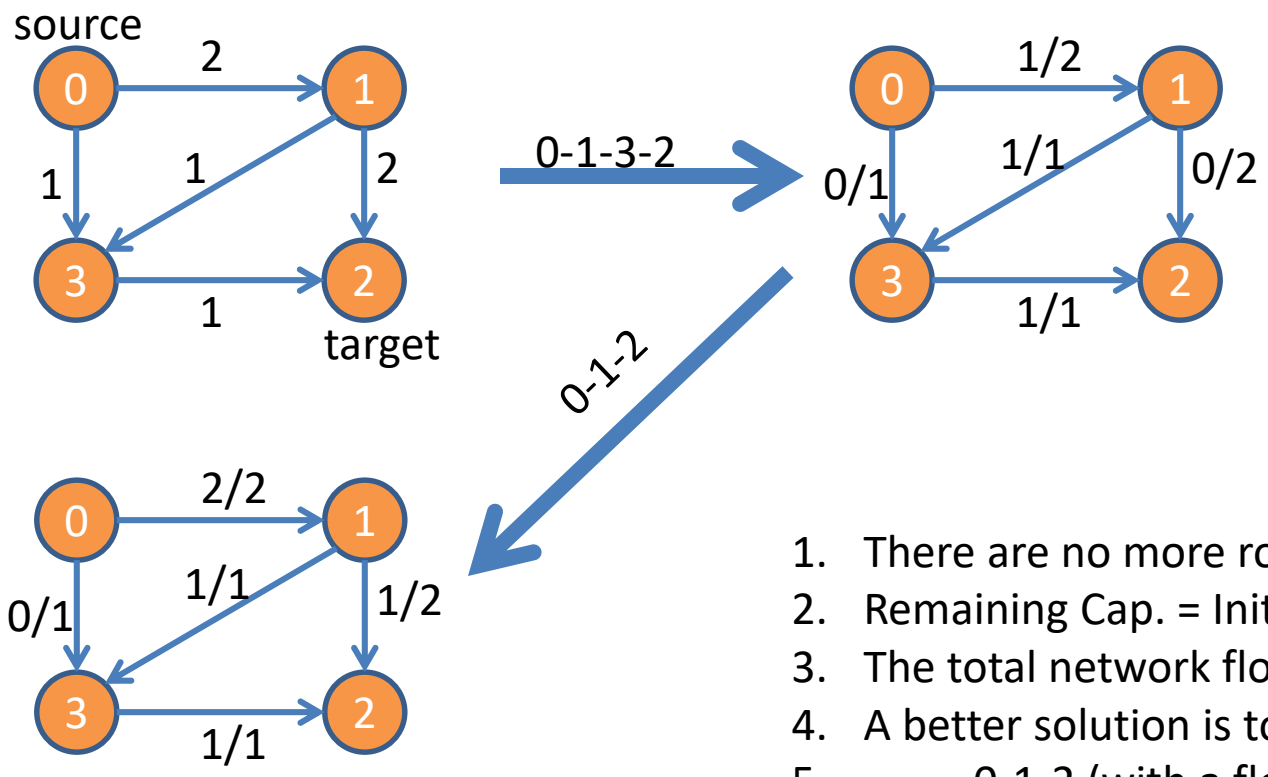
Maximum network flow

- **Naive approach:**

```
void naive(pair g[n][n], int s, int t) {  // NOT optimal
        vector<pair> path;
        while (findPath(g, s, t, path))    {
                int min = g[path[0]][path[1]].capacity;
                for (int i=1; i<path.size()-1; i++)
                        if (min > g[path[i]][path[i+1]].capacity)
                                min = g[path[i]][path[i+1]].capacity;
                for (int i=0; i<path.size()-1; i++) {
                        int u = path[i], v = path[i+1];
                        g[u][v].capacity -= min; // reduce capacity
                        g[u][v].flow += min;  // add flow
                }
        }
}
// for findPath, use DFS,BFS or A*
```

Maximum network flow

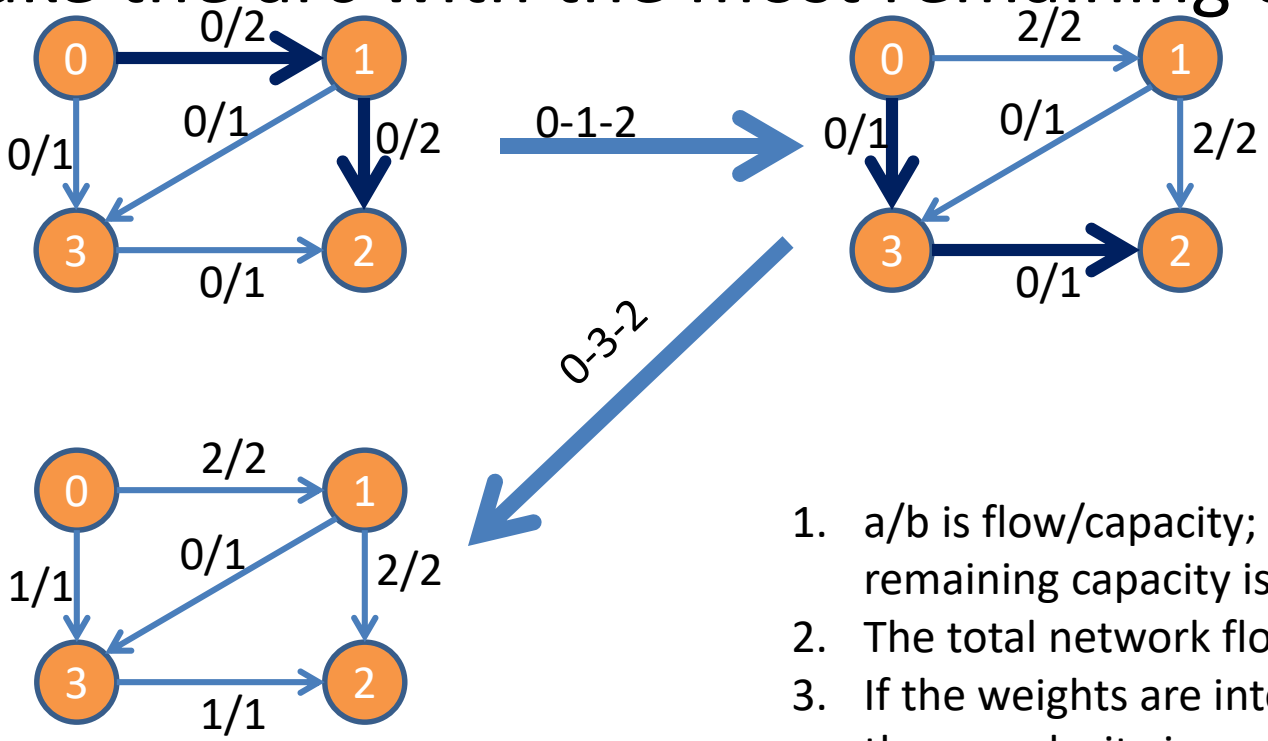
- **Naive approach:** example



1. There are no more roads to destiny
2. Remaining Cap. = Initial - Flow
3. The total network flow is 2
4. A better solution is to choose:
5. 0-1-2 (with a flow of 2)
6. 0-3-1 (with a flow of 1), adding 3
7.

Maximum network flow

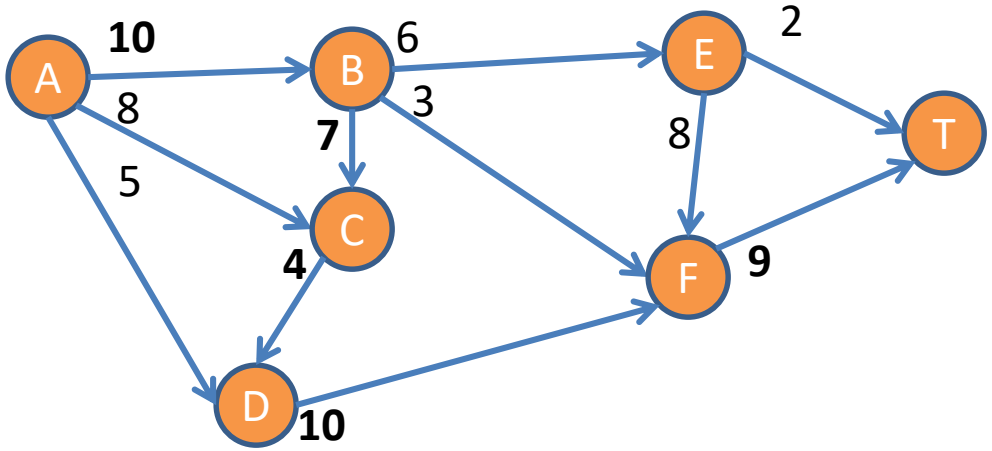
- **Another maximum flow algorithm:** at each step I take the arc with the most remaining capacity



1. a/b is flow/capacity; the remaining capacity is b-a
2. The total network flow is 3
3. If the weights are integers, the complexity is maxflow*|E|

Maximum network flow

- **Another maximum flow algorithm:** at each step I take the arc with the most remaining capacity

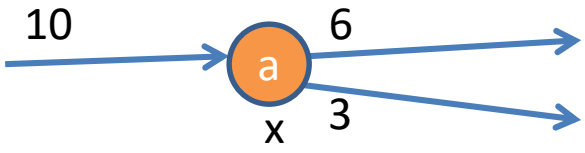


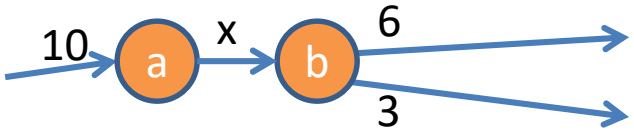
Maximum network flow

- **Another maximum flow algorithm:** as you can see, these algorithms are not optimal.
- Optimal algorithms (require a residual graph):
  - Ford Furkelson
  - Edmons-Karp

Maximum network flow

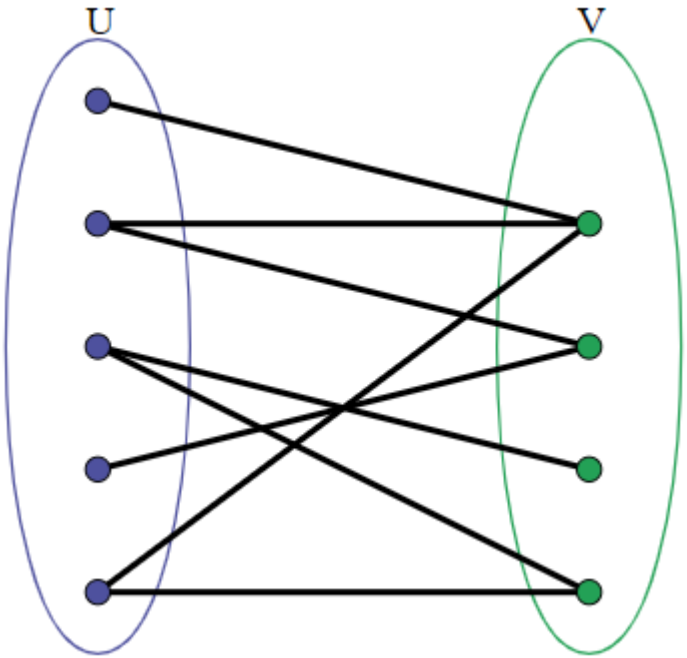
- If there are multiple sources (or targets), add a dummy source (or target) with a link to each actual source (or target). What would be the capacity to each source (or target)?
- If there is an “vertex with capacity=x", we replace it with 2 vertices and a link with that capacity





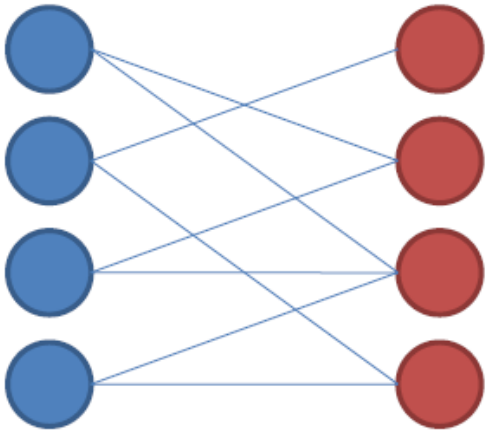
Bipartite graphs

- It is a graph whose vertices can be separated into two disjoint sets U and V, where there are no arcs connecting 2 nodes of U, nor two nodes of V, but there can be arcs connecting nodes from U to V.

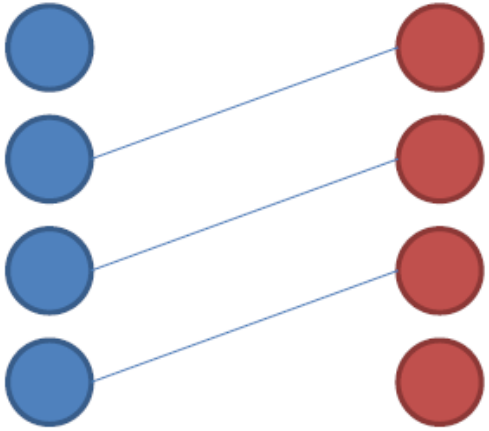


Bipartite graphs

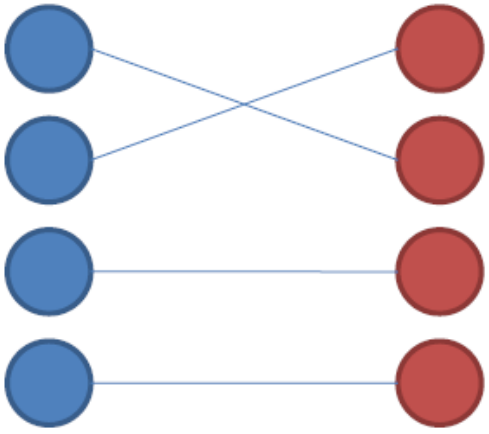
- A "matching" consists of selecting arcs from U to V so that vertex has (as much) one adjacent arc.
- Maximum bipartite matching (MBM) means that this matching has to be maximum in number of arcs.



Bipartite graph



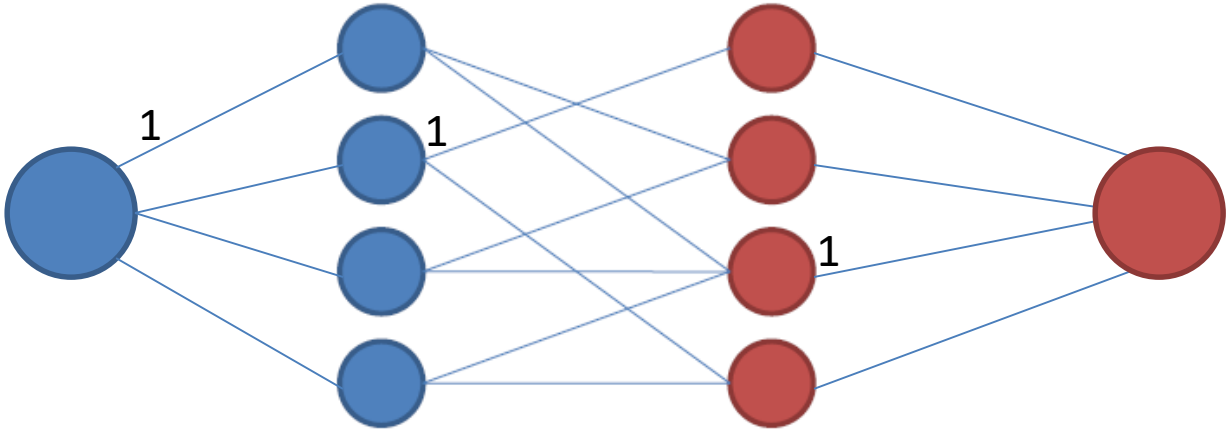
A matching



Maximum bipartite matching

Bipartite graphs

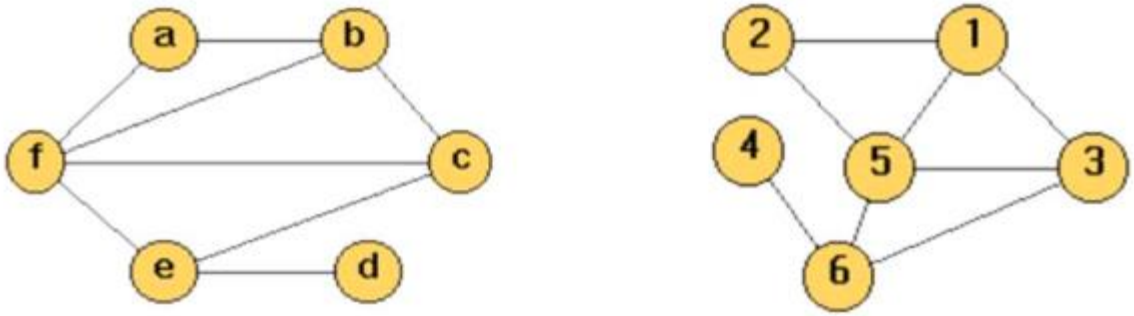
- Adding a source and target, we can convert this problema into a “maximum network flow”.



weight = 1 for each arc

Other problems with graphs

- Is the graph flat? That is, can you draw in 2D without crossing the arcs? It is O(n+m) using Tarjan Hopcroft Algorithm
- Isomorphism of graphs: find a bijection between vertices of two graphs, such that both graphs are equivalent. This is equal to re-labeling the vertices so that both graphs are equal.



{(a→2), (b → 1), (c → 3), (d → 4), (e → 6), (f → 5)}

Other problems with graphs

- Min Cost Max Flow
- Min-Cut: minimal cutting in graphs
- Hungarian algorithm
- RMQ (Range Minimum Query), Lowest Common Ancestor (LCA)
- PERT (program evaluation and review technique) CPM (critical path analysis)
- In network: Ford Furkenson, Edmonds-Karp algorithm, Push & Relabel

Spoj

There are many BFS problems

https://www.spoj.com/problems/tag/bfs

And graphs in general

https://www.spoj.com/problems/tag/graph

Minimum spanning tree

https://www.spoj.com/problems/MST/

Prim or Kruskal

https://www.spoj.com/problems/CSTREET/

Spoj

Interesting (DS + MST)

https://www.spoj.com/problems/IITWPC4I/

Just MST (Kruskal ?)

https://www.spoj.com/problems/BLINNET/

Network flow

https://www.spoj.com/problems/FASTFLOW/

https://www.spoj.com/problems/COCONUTS/

Spoj

Bipartite graphs

https://www.spoj.com/problems/MATCH/

Hungarian algorithm

https://www.spoj.com/problems/SCITIES

For next class (January 10)

- Group 1: study and prepare Ford Furkenson and/or Edmons-Karp (power point). Solve 1 problem related to network Flow and one problem related to bipartite matching. Try a third problem related to MST, Kruskal, Prim, Dijkstra, Floyd-Warshal
- Group 2: study and prepare Hungarian algorithm (power point). Solve one problem related to Hungarian algorithm. Solve 1 or more problems related to Prim, Kruskal, Dijkstra, Floyd-Warshal