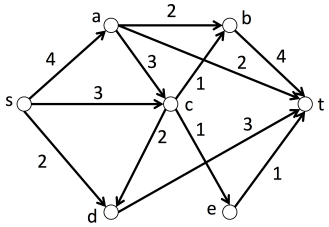
# CS 124 Data Structures and Algorithms — Spring 2015

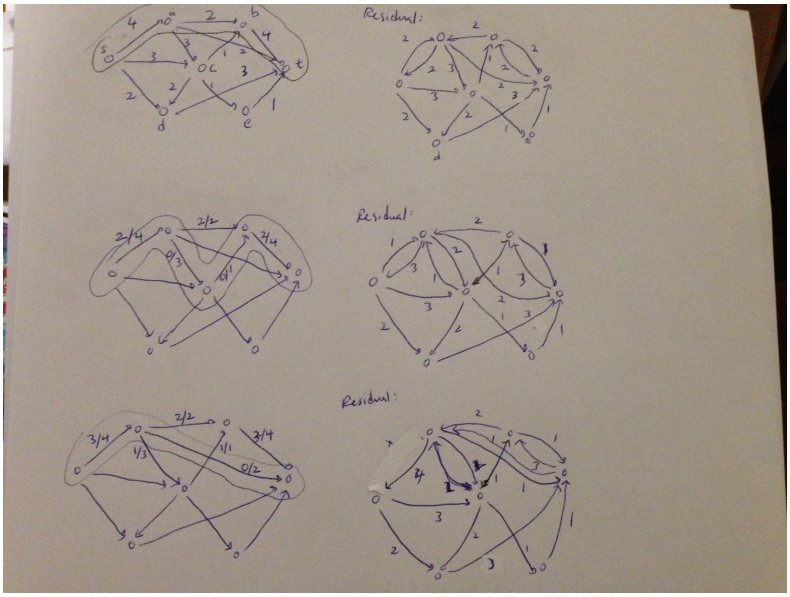
## Problem Set 7 solutions

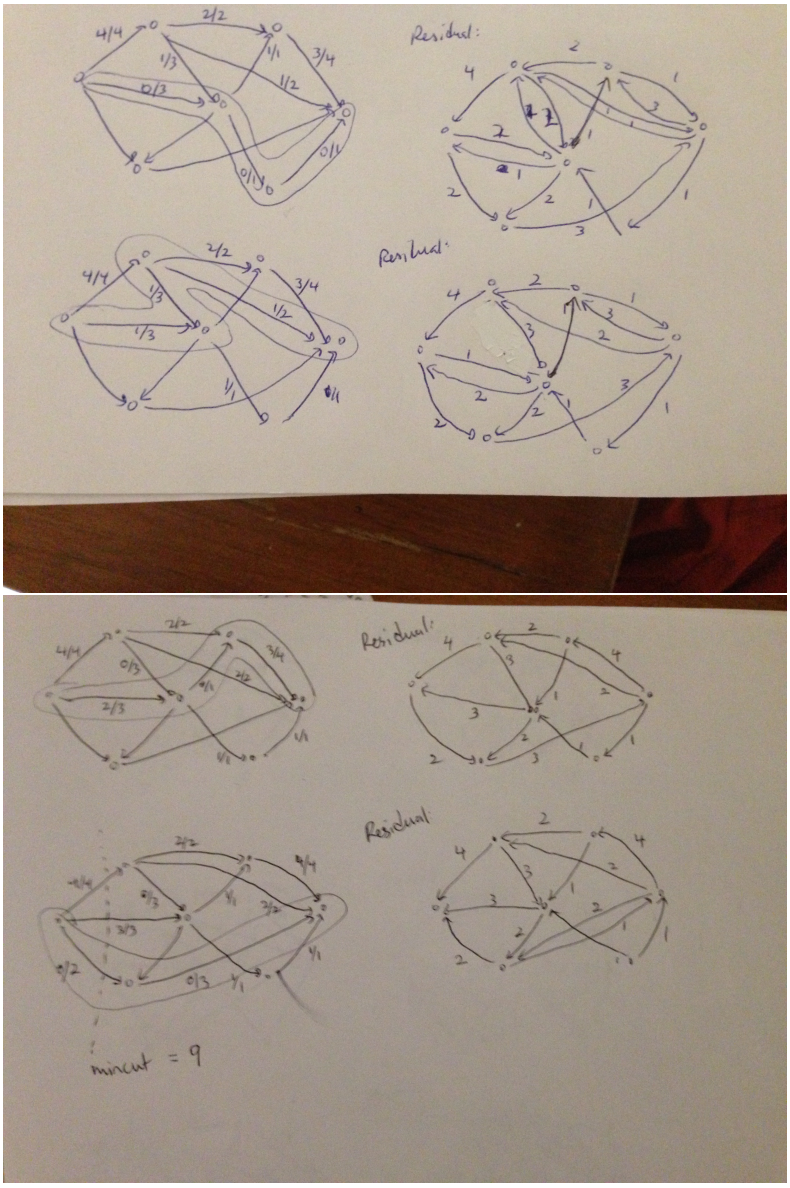
**Problem 1**

Find the maximum flow from $s$ to $t$ and the minimum cut between $s$ and $t$ in the network below, using the method of augmenting paths discussed in class. (This means give the flow along each edge, along with the final flow value; similarly, give the edges that cross the cut, along with the final cut value.) Show the residual network at the intermediate steps as you build the flow. (If you need more information on the algorithm, it's called the Ford-Fulkerson algorithm.)




**Solution:**






**Problem 2**

You have been given a square plot of land that has been divided into $n$ rows and columns, yielding $n^2$ square subplots. Some of these subplots have rocky ground and cannot support plant growth, while others have soil. You would like to plant palm trees on a subset of the square subplots so that every row and every column has exactly the same number $p$ of palm trees. Furthermore, you would like to do this so that $p$ is as large as possible. Devise an efficient algorithm to determine how to accomplish this.

**Solution:** First we will discuss how to determine whether it is possible to achieve a particular value of $p$. Set up a bipartite graph with $n$ vertices on the left to represent each

row and $n$ on the right to represent each column. Connect a row vertex to a column vertex with capacity 1 if the corresponding square has soil. Make a super-source that connects to each row vertex with capacity $p$, and a super-sink that connects to each column vertex with capacity $p$.

Run a max flow algorithm on this graph; note that if a flow of $pn$ exists (i.e. all super-source and super-sink capacities are saturated), then this corresponds to a feasible planting configuration on the original grid. This is because each unit of flow corresponds to a match between row and column, representing a palm tree on the corresponding square.

In order to find the largest value of $p$ that works, one can run the above for $p = 1, 2, \ldots, n$ and return the largest $p$ that works. We have $2n$ vertices and $m \leq n^2$ edges, where $m$ is the number of soil plots. We henceforth assume $m \geq n$, since otherwise the optimal value of $p = 0$. Each max flow takes time $O((m+n)n) = O(mn)$, since the max flow value is at most $n$. The total time is thus $O(mn^2)$.

An improvement can be made in the running time via the following claim: we claim that if exactly $p$ is possible per row and column, then exactly $p'$ is possible for any $p' \leq p$. Before we prove the claim, this already means that we can improve the running time in various ways. First, suppose the optimal value of $p$ is $p_*$. Then we can halt the above loop as soon as some $p$ doesn't have max flow value $np$, then return $p_* = p - 1$. This gives running time $O(mp_*^2)$, which is better for small $p_*$. Even better: we could binary search on $p$; this would give running time $O((m+n)p_* \log n)$ (in fact this can even be made $O((m+n)p_* \log(p_*))$ by geometrically growing the right boundary point of the binary search). An even better solution using the claim is as follows: as before simply loop over $p$ in increasing order, and let $\tilde{p}$ be the first value of $p$ where it is impossible. Then return $p_* = \tilde{p} - 1$. However, once we have computed a max flow $f_p$ for $p$ and want it for $p+1$, we do not need to recompute the max flow from scratch! We can simply increase the capacities that are $p$ to $p+1$ to form a new graph $G_{p+1}$ from $G_p$. Note then that $f_p$ is still a feasible flow for $G_{p+1}$. We then just need to augment it to obtain a max flow for $f_{p+1}$ for $G_{p+1}$. But $f_p$ has flow value $np$, and $f_{p+1}$ cannot possibly have flow value more than $n(p+1)$. Thus the difference in flow values is at most $n$, and thus we have to perform at most $n$ augmentation steps. Thus, for each value of $p$ up to $p_* + 1$ we do at most $n$ augmentation steps, so the total time is $O(mnp_*)$. In terms of just $n$, we know $m = O(n^2), p_* \leq n$, so this running time is always $O(n^4)$.

Now we just need to prove the claim. That is, we want to show if exactly $p$ trees per row and column is possible, then exactly $p-1$ trees is also possible. Note that by induction, this implies that exactly $i$ trees are possible for any $i < p$. The claim conveniently follows from the statement of Problem 3 in this same problem set! The claim is equivalent to saying that any bipartite and biregular graph which is $p$-regular has a perfect matching (since then if we remove the matching, what we are left with is exactly $p-1$ trees per row and column). By Problem 3, it suffices to show that any subset $U$ on the left has at least $|N(U)|$ neighbors on the right. The number of edges leaving the left is $p|U|$. Since the right is also $p$-regular, the number of nodes receiving these edges must be at least $p|U|/p = |U|$.

**Problem 3**

There are $n$ hungry amphibians sitting around a pond, when $n$ insects suddenly fly overhead. Each amphibian looks at the insects and decides on a subset of them that it is willing to eat. Suppose that for any subset $U$ of the amphibians, the collective set of insects they are willing to eat, denoted as $N(U)$, satisfies the condition $|N(U)| \geq |U|$. Prove that there is a way for every amphibian to eat exactly one desired insect without conflicts. **Hint:** Construct a graph and reason about flows.

**Solution(s):** This is a well-known theorem known as Hall's (marriage) theorem, and can be stated as:

**Theorem 1.** *Given a bipartite graph with bipartition $(L, R)$, $|L| = |R| = n$, there exists a matching of size $n$ if and only if for all $U \subseteq L$, $|N(U)| \geq |U|$. Here $N(U)$ is the set of vertices adjacent to $U$.*

A matching of size $n$ is also known as a perfect matching.
Note that one direction is obvious: if a perfect matching exists, then for any subset $U \subseteq L$, $N(U)$ must contain at least the matching vertices.
Thus, it suffices to consider the other direction.

**Solution 1:** Recall that bipartite matching can be modeled by network flow. Set up a flow network with supersource $s$ with edges to $L$ and supersink $t$ with edges from $R$, and set all edges to have capacity 1. Consider the $s, t$ min-cut of this graph; suppose the $s$ part of the cut contains vertices $L' \subseteq L, R' \subseteq R$.
The value of the min-cut consists of the number of edges across the cut; therefore, we have edges $s \to L - L', L' \to R - R', R' \to t$, with total value

$$|L - L'| + |R'| + \#edges(L', R - R')$$

Note that $|R'| + \#edges(L', R - R') \geq |N(L')|$, since each edge from $L'$ to $R - R'$ can contribute at most one vertex to the neighborhood of $L'$. Therefore, we have that the cut value is

$$\geq |L - L'| + |N(L')| \geq |L - L'| + |L'| = n$$

Since the min-cut is of value at least $n$, the max-flow is also at least $n$, implying a matching of $n$ vertices (i.e. a perfect matching) must exist.

**Solution 2:** We use induction. For the base case $n = 1$, the result is obvious.
Now consider a bipartite graph with $n$ vertices on each side. We consider two cases:
Case 1: Suppose that we have $|N(U)| \geq |U| + 1$ for all $U \subseteq L$. Then we can pick an arbitrary edge $(v, w)$ for $v \in L, w \in R$ for our matching, and the remaining graph of $n - 1$ vertices will still satisfy the given condition and thus has a perfect matching by induction.

Case 2: Suppose that for some $U \subseteq L$, $|N(U)| = |U|$. I claim that the graph induced by $L - U, R - N(U)$ still satisfies the condition $|N'(S)| \geq S$ for $S \subseteq L - U$ and the neighbors $N'(S) \subseteq R - N(U)$.
Indeed, suppose this isn't true for some $S \subseteq L - U$, so that its neighbors $N'(S) \subseteq R - N(U)$ has $|N'(S)| < |S|$ (the rest of $S$'s neighbors must be in $N(U)$). Then if we consider $S \cup U \subseteq L$, we have that its neighborhood $N(S \cup U) \subseteq R$ is the disjoint union of $N'(S)$ and $N(U)$, so that

$$|N(S \cup U)| = |N'(S)| + |N(U)| < |S| + |N(U)| = |S| + |U| = |S \cup U|$$

contradicting the condition in the problem statement for the original graph.
Therefore, since the condition holds for the subgraphs $U \cup N(U)$ and $(L - U) \cup (R - N(U))$, we can apply the inductive hypothesis to each and get a perfect matching for $(L, R)$.

**Solution 3:** Assume for contradiction that no perfect matching exists, and consider the maximal matching $M$ of size $< n$. Denote the subset of matched vertices in $L$ as $L_M$ and similarly define $R_M$. We can represent the matched edges as directed backwards from $R_M$ to $L_M$ and the other edges as forward from $L$ to $R$ (in a similar spirit as a flow residual graph). Take some vertex $v_0$ in $L - L_M$ (which must exist since it's not a perfect matching). By the problem condition, it must have a neighbor in $R$, and since we already have a maximal matching, the neighbor must be in $R_M$. Let the neighbor be $w_1$, and let its match in $L_M$ be $v_1$. We can then draw a path $v_0 \to w_1 \to v_1$.
Using this idea, define an *alternating path* to be a path that begins from $v_0$ and alternates between $L$ and $R$ (recall that backward $R \to L$ edges come from the maximal matching). Denote $S$ as the set of vertices in $L$ reachable via alternating paths from $v_0$ (including $v_0$), and similarly define $T \subseteq R$.
I claim that $T \subseteq R_M$. Suppose for contradiction there exists an alternating path that ends in a vertex $w \in R - R_M$. Then by augmenting along this path (as we do for residual graphs), we obtain a matching that includes $v_0$ and $w$, adding another pair to the matching and contradicting the maximality of $M$.
Thus, since there are only back edges from $R_M \to L_M$, we also obtain $S \subseteq L_M$, and further that $|S| - 1 = |T|$ by the matching $M$ (remember that $v_0$ is included!).
I also claim that $N(S) \subseteq T$. Indeed, if not, then we can obtain an alternating path that ends in $R - T$, contradicting the definition of $T$. Thus, by the problem condition, we get that $|T| \geq |N(S)| \geq |S|$, contradicting the fact that $|S| - 1 = |T|$. Therefore, our initial assumption was false and a perfect matching must exist.

## Problem 4

Let $G = (V, E)$ be an unweighted, undirected graph with $n$ vertices and $m$ edges. Suppose that we do not want to find just one minimum cut, but want to count the *number* of minimum cuts (recall in class that we said the number of minimum cuts is never more than $\binom{n}{2}$, which is achieved by the $n$-cycle, but in general the number of minimum cuts could be any integer between 1 and $\binom{n}{2}$). Any running time of the form $O(n^C)$ for constant $C > 0$ receives full

credit, and your algorithm may be Monte Carlo with failure probability at most 1/3. What would you change if you wanted failure probability $P$ for some given $0 < P < 1/3$, and how would that affect your running time?

**Solution:** Recall that for some mincut $C$, Karger's algorithm in one iteration will find $C$ with probability at least $1/\binom{n}{2}$. We set up an array $A$ for each possible cut size, and run Karger's multiple times to produce some cut $c$, and we also record which cuts we have seen already (recording can be done with a bitmask for which vertices are in the cut). If we have a new cut, then we increment $A[c]$ for the cut size of $c$. The smallest cut size will then be assumed to be the mincut, and the number of mincuts will just be how many $A$ has recorded. We analyze the failure probability. Suppose we run Karger's $k$ times. Our algorithm fails if we missed any mincut, so suppose we have mincuts in our graph $C_1, \ldots, C_l$. Then

$$P(\text{missed some mincut}) = P(\text{missed } C_1 \text{ or missed } C_2 \ldots \text{ or missed } C_l) \leq$$

$$\leq \sum_i P(\text{missed } C_i) \leq \binom{n}{2} \cdot (1 - 1/\binom{n}{2})^k$$

by the union bound, and since we have at most $\binom{n}{2}$ mincuts. The failure probability of missing one mincut $C_i$ is at most $1 - 1/\binom{n}{2}$ in one iteration of Karger's, and we run it $k$ times.
We then have that

$$P(\text{missed some mincut}) \leq \binom{n}{2} \cdot (1 - 1/\binom{n}{2})^k \leq \binom{n}{2} \cdot e^{-k/\binom{n}{2}}$$

so note that if we choose

$$k \geq \binom{n}{2} \log \frac{\binom{n}{2}}{P}$$

then we get $P(\text{failure}) \leq P$ for our desired failure probability $P$.
One iteration of Karger's algorithm (contraction) takes time $O(m)$ time, since we essentially add a new vertex and connect all the previous graph's edges while removing the contracted pair of vertices. If we run it $k$ times, where $k$ is $O(n^2 \log(n/P))$, then we get time $O(mn^2 \log(n/P))$ which is polynomial as desired.

**Problem 5:**

Apply network flow. Create a source and sink vertex, and connect the source to all caterpillar nodes and all borders of the grid to the sink vertex, all with capacity 1. We set two antiparallel edges of capacity 1 to replace each undirected edge.
Because we want vertex-disjoint paths, we split each node $v$ into $v_{in}$ and $v_{out}$ nodes, with an edge going from $v_{in}$ to $v_{out}$ with capacity 1. All edges directed into $v$ now direct to $v_{in}$, and all edges directed out of $v$ now direct to $v_{out}$. The vertex-disjoint condition is now handled by the capacity of the edge $(v_{in}, v_{out})$.

Thus, if the max flow is found to be $< m$, we cannot route all caterpillars to the border of the grid, and the max flow value is the maximum number that can be routed. If the max flow is equal to $m$, then it is possible.
Here is a sample solution:

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <vector>
#include <queue>

#define MAXN 5600 // 50*50*2
#define MAXM 3000
#define INF 9999

using namespace std;

const int dx[] = {0, 1, -1, 0};
const int dy[] = {1, 0, 0, -1};
int n,m;

// Represent a vertex of the grid as 2*(n*x+y) + {0,1}
// where 0 is the in-vertex, 1 is the out-vertex

// source, sink
int S = 5500;
int T = 5501;

// Stores the locations of the caterpillars
int loc[MAXM];

// adjacency lists
int adj[MAXN][MAXN]; // largest will be for S, T
int adj_size[MAXN];

int maxflow = 0;
int cap[MAXN][MAXN]; // capacity is always 1
int flow[MAXN][MAXN];
int par[MAXN];

// augment a path
int augment(int v, int f) {
  if (v == S) {
```

```
    maxflow += f;
    return f;
  }

  int u = par[v];
  bool backflow = false;
  if (flow[v][u] > 0) {
    backflow = true;
  }
  if (backflow) {
    f = min(f, flow[v][u]);
  } else {
    f = min(f, cap[u][v] - flow[u][v]);
  }
  f = augment(u, f);
  if (backflow) {
    flow[v][u] -= f;
  } else {
    flow[u][v] += f;
  }
  return f;
}

void add_adj(int u, int v) {
  cap[u][v] = 1;
  adj[u][adj_size[u]] = v;
  ++adj_size[u];
  adj[v][adj_size[v]] = u;
  ++adj_size[v];
}

int main() {
  cin >> n >> m;

  for (int j=0;j<m;++j) {
    int x,y;
    cin >> x >> y;
    --x; --y;
    loc[j] = 2*(n*x + y);
    add_adj(S, loc[j]);
    cap[S][loc[j]] = 1;
  }
```

```
// set capacities and adjacent vertices
for (int x=0;x<n;++x) {
  for (int y=0;y<n;++y) {
    int u = 2*(n*x + y);
    add_adj(u, u+1);
    for (int k=0;k<4;++k) {
      int v = 2*(n*(x + dx[k]) + (y+dy[k]));
      if (v < 0 || v >= 2*n*n)
        continue;
      add_adj(u+1, v);
      add_adj(v+1, u);
    }
  }
}

// sink
// edges of grid
for (int x=0;x<n;++x) {
  int v = 2*(n*x) + 1;
  add_adj(v, T);
  v = 2*(n*x + n-1) + 1;
  add_adj(v,T);
}
for (int y=1;y<n-1;++y) {
  int v = 2*(y) + 1;
  add_adj(v,T);
  v = 2*(n*(n-1) + y) + 1;
  add_adj(v,T);
}

// run ford-fulkerson
while (1) {
  queue<int> q;
  memset(par, -1, sizeof(par));
  q.push(S);
  par[S] = S;
  bool augmented = false;
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int i=0;i<adj_size[u];++i) {
```

```
        int v = adj[u][i];
        if (par[v] != -1) continue;
        if (cap[u][v] - flow[u][v] > 0 || flow[v][u] > 0) {
          par[v] = u;
          if (v == T) {
            augment(T, INF);
            augmented = true;
            break;
          } else {
            q.push(v);
          }
        }
      }
      if (augmented) break;
    }
    if (!augmented) break; // no more augmenting paths
  }

  if (maxflow == m)
    printf("possible\n");
  else if (maxflow < m) {
    printf("not possible\n");
    printf("%d\n", maxflow);
  }

  return 0;
}
```