Algorithm Design and Analysis (Fall 2023)
Assignment 5
Deadline: Jan 2, 2023

1. (30 points) [**Menger's Theorem**] Let $G = (V, E)$ be a connected undirected graph. The *edge connectivity* of $s, t \in V$ is the minimum number of edges whose removal disconnects s and t. The *vertex connectivity* of s, t is the minimum number of graph elements (any vertex in $V \setminus \{s, t\}$ or any edge in E) whose removal disconnects s and t. Menger's Theorem states that the edge connectivity of s, t is exactly the number of edge-disjoint paths between s and t and the vertex connectivity of s, t is exactly the number of *internally* vertex-disjoint paths between s and t. (Obviously, two paths between s and t are not vertex-disjoint since they intersect at s and t; *internally* vertex disjoint means they are disjoint except for s and t.)

   Prove that Menger's Theorem is just a consequence of the max-flow-min-cut theorem. Be sure the network you construct is *directed* and *capacitated.*

   Proof
   First, transfer G into a flow network by letting $c(u, v) = c(v, u) = 1$ for all edges $(u, v) \in E$.
   (1) Prove that the edge connectivity of s, t is exactly the number of edge-disjoint paths between s and t:
   · First, it's obvious that the minimum number of edges whose removal disconnects s and t is the number of edges in the min-cut by definition. Since we define $c(u, v) = c(v, u) = 1$, the number of edges in the min-cut is equal to the value of the max-flow, k.
   · Next, we prove k is equal to the number of edge-disjoint paths. Since $c(u, v)$ is an integer, k must be an integer and there must exists a min-cut that contains k edges with $f(u, v) = c(u, v) = 1$. If there are more than k edge-disjoint paths, there will be some paths which have to share edges so that the overall max-flow is k, which contradicts to the definition of edge-disjoint paths. So the number of edge-disjoint paths is less than k.
   · Else, if there are less than k edge-disjoint paths, at least two edges in a path have to go through the cut. In this case, there must be an edge in this path whose capacity is at least 2 because $f_{in} = f_{out}$ and there is a vertex whose $f_{out} = f_{in} \geq 2$ and the number of edges in its two sides(in, out) are not equal or the path can be divided into more than 1 distinct edge-disjoint paths. It contradicts to the definition that $c(u, v) = c(v, u) = 1$ and $f(u, v) < c(u, v)$. So the number of edge-disjoint paths is more than k. Therefore, the number of edge-disjoint paths is equal to k.
   · So, the minimum number of edges whose removal disconnects s and t is equal to the number of edge-disjoint paths.

(2) Prove that the vertex connectivity of s, t is exactly the number of *internally* vertex-disjoint paths between s and t.

· First, transfer each vertex in V into v_{in} and v_{out}. Let $c(v_{in}, v_{out}) = 1$ and $c(v_{out}, v_{in}) = 0$. Then, transfer each edge in E into two directed edges (u_{out}, v_{in}) and (v_{out}, u_{in}) if $(u, v) \in E$. Meanwhile, let $c(u_{out}, v_{in}) = c(v_{out}, u_{in}) = 1$

· Similar to (1), in a min-cut where each edge has a $f = 1$, the number of the cut is equal to the number of minimum number of graph elements because an edge from v_{in} to v_{out} in the cut means a vertex and an edge from v_{out} to u_{in} in the cut means an edge. Meanwhile, all these edge has a capacity of exactly 1, so the value of max-flow = the number of edges in the min-cut, k.

· Similarly, to prove the number of internally vertex-disjoint paths = k, we can apply the same method as (1) because in the above-constructed directed graph, the internally vertex-disjoint paths is exactly the edge-disjoint paths(use an edge to connect v_{in} and v_{out} so vertex-disjoint $\rightarrow$ edge-disjoint). Therefore, the number of internally vertex-disjoint paths is equal to k.

· So, the minimum number of elements whose removal disconnects s and t is equal to the number of internally vertex-disjoint paths. Therefore, Menger's Theorem is just a consequence of the max-flow-min-cut theorem.

2. (30 points) **[Perfect Matching on Bipartite Graph]** A graph is *regular* if every vertex has the same degree. Let $G = (A, B, E)$ be a regular bipartite graph with $|E| > 0$.

   (a) (10 points) Prove that $|A| = |B|$.

   (b) (15 points) Let $n = |A| = |B|$. Can we conclude that G must contain a matching of size n? If so, prove it; if not, provide a counterexample.

(a) Assume every vertex has a degree d, since every edge has its one endpoint in A and the other in B, we have $|E| = |A|d$ and $|E| = |B|d$, which indicates $|E| = |A|d = |B|d \rightarrow |A| = |B|$.

(b) Yes.

First, prove n is the maximum size of G's matching. Since $|V| = |A| + |B| = 2n$ and any edge connects at most 2 vertices, the size of any matching is at most $|V|/2 = n$.

So, to prove that G must contain a matching of size n, we just need to prove that G's maximum matching has a size of n. Because the size of maximum matching is equal to the value of max-flow in G'(proved in class), we just need to show that the value of max-flow in G' is n.

Then, construct G'. First, add a source vertex s and connect it to every vertex in A with $c(s, v) = 1$ for all $v \in A$. Then, add a sink vertex t and connect every vertex in B to it with $c(u, t) = 1$ for all $u \in B$. Next, transfer every edge in G into directed edge (u, v), $u \in A$ and $v \in B$ with $c(u, v) = \frac{1}{d}$.

Then, apply maximum flow and Ford-Fulkerson Algorithm. Since all vertex has a degree d, every vertex in A has d edges out and every vertex in B has d edges in, it's easy to conclude that every edge has its flow = capacity. In this case, the flow is maximum because we can't find any path from s to t. Since all edge has its flow = capacity, the value of maximum flow is equal to $\sum f(s, v) = n$ for all $v \in A$. Therefore, the value of max-flow is n, which implies that G must contain a matching of size n

3. (40 points) **[König-Egerváry Theorem]** In the class, we have seen that the maximum matching problem can be formulated by the following linear program

$$\begin{aligned} \text{maximize} \; & \sum_{e \in E} x_e \\ \text{subject to} \; & \sum_{e: e=(u,v)} x_e \le 1 && (\forall v \in V) \\ & x_e \ge 0 && (\forall e \in E) \end{aligned}$$

and the minimum vertex cover problem can be formulated by the following linear program

$$\begin{aligned} \text{minimize} \; & \sum_{u \in V} x_u \\ \text{subject to} \; & x_u + x_v \ge 1 && (\forall (u,v) \in E) \\ & x_u \ge 0 && (\forall u \in V) \end{aligned}$$

We have also seen that the second linear program is the dual program of the first.

(a) (20 points) Prove that both linear programs have integral optimal solutions if the graph is bipartite.

(b) (10 points) Using the result in the first part, prove König-Egerváry Theorem, which states that the size of the maximum matching equals to the size of the minimum vertex cover in a bipartite graph.

(c) (10 points) Provide a counterexample showing that the claim fails for non-bipartite graphs.

(a) Let A be the incidence matrix of G. An incidence matrix is a $|V| \times |E|$ matrix and $a_{ij} = 1$ if v_i is adjacent to e_j, else, $a_{ij} = 0$.
Then, the LP problem of maximum matching can be written as (x is a column vector of all edges, c is a column vector of 1):

$$\begin{aligned} \text{maximize} \; & c^T x \\ \text{subject to} \; & Ax \le 1 \\ & x \ge 0 \end{aligned}$$

The LP problem of minimum vertex cover can be written as (y is a column vector of all vertices, b is a column vector of 1):

$$\begin{aligned} \text{minimize} \; & b^T y \\ \text{subject to} \; & A^T y \ge 1 \\ & y \ge 0 \end{aligned}$$

It's easy to observe that 1 is an integer vector, so we only need to prove that A is totally unimodular in order to prove that both linear programs have integral optimal solutions because A^T is totally unimodular if A is totally unimodular.
Prove A is totally unimodular by induction.
Base case: Each cell of A is either 0 or 1 by definition of A.
Inductive step: Suppose every $k \times k$ submatrix of A has determinant belongs to $\{0, 1, -1\}$.
Consider any $(k+1) \times (k+1)$ submatrix A'.
Case 1: If a column of A' is all-zero, then $det(A') = 0$.
Case 2: If a column of A' contains only one 1, then $det(A')$ equals to ± 1 times the determinant of a $k \times k$ submatrix. $det(A') \in \{0, -1, 1\}$ by induction hypothesis.
Case 3: If every column of A' has two 1. Because the graph is bipartite, vertices can be divided into A and B with no edge connecting two vertices in the same part. So we can rewrite A by shuffling rows of A' as: $\begin{bmatrix} A_1 \\ A_2 \end{bmatrix}$ where vertices in A_1 means they belong to part A and A_2 means part B so that each column of A_1 and A_2 has only one 1 respectively. Therefore, we can add up all the rows of A_1 and then minus all the rows of A_2 to get a zero vector. So we have $det(A') = 0$.
By induction, we prove that A is totally unimodular, thus proving that both linear programs have integral optimal solutions.

(b) First, we have the maximum matching problem defined by letting $x_e = 0, 1$:

$$\begin{aligned} \text{maximize} \ & \sum_{e \in E} x_e \\ \text{subject to} \ & \sum_{e:e=(u,v)} x_e \leq 1 && (\forall v \in V) \\ & x_e \in \{0, 1\} && (\forall e \in E) \end{aligned}$$

Then, we have its LP version:

$$\begin{aligned} \text{maximize} \ & \sum_{e \in E} x_e \\ \text{subject to} \ & \sum_{e:e=(u,v)} x_e \leq 1 && (\forall v \in V) \\ & x_e \geq 0 && (\forall e \in E) \end{aligned}$$

In (1), we have proved that both linear programs have integral optimal solutions. And in the LP version, we have $\sum_{e:e=(u,v)} x_e \leq 1$ and $x_e \geq 0$, so x_e be either 0 or 1, which stands for the maximum matching problem. Then, the two problem has the same integral optimal solutions.

Similarly, we have the minimum vertex cover problem:

$$\begin{aligned}
\text{minimize } & \sum_{u \in V} x_u \\
\text{subject to } & x_u + x_v \geq 1 && (\forall (u,v) \in E) \\
& x_u \in \{0,1\} && (\forall u \in V)
\end{aligned}$$

Then, we have its LP version:

$$\begin{aligned}
\text{minimize } & \sum_{u \in V} x_u \\
\text{subject to } & x_u + x_v \geq 1 && (\forall (u,v) \in E) \\
& x_u \geq 0 && (\forall u \in V)
\end{aligned}$$

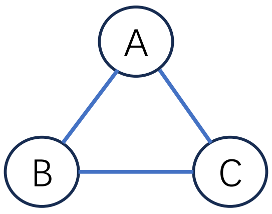
Similarly, we can let x_u be either 0 or 1 because if $x_u > 1$, we can always decrease x_u to 1 to decrease the objective function while satisfying all the constraints. According to (1), the two problem has the same integral optimal solutions.
Then, the LP problem of maximum matching can be written as (x is a column vector of all edges, c is a column vector of 1):

$$\begin{aligned}
\text{maximize } & c^T x \\
\text{subject to } & Ax \leq 1 \\
& x \geq 0
\end{aligned}$$

The LP problem of minimum vertex cover can be written as (y is a column vector of all vertices, b is a column vector of 1):

$$\begin{aligned}
\text{minimize } & b^T y \\
\text{subject to } & A^T y \geq 1 \\
& y \geq 0
\end{aligned}$$

The dual problem of LP maximum matching problem is exactly the LP minimum vertex cover problem. Since the strong duality holds (easy to be seen by Convex Optimization), we have: maximum $c^T x =$ minimum $b^T y$. So, their relaxed version(0,1 version) is also equal, which indicates that the size of the maximum matching equals to the size of the minimum vertex cover in a bipartite graph.
(c) Counterexample is shown below. The size of the maximum matching is 1 while the size of the minimum vertex cover is 2.



4. (Bonus 60 points) [**Dinic's Algorithm**] In this question, we are going to work out *Dinic's algorithm* for computing a maximum flow. Similar to Edmonds-Karp algorithm, in each iteration of Dinic's algorithm, we update the flow f by increasing its value and then update the residual network G^f. However, in Dinic's algorithm, we push flow along *multiple* s-t paths in the residual network instead of a *single* s-t path as it is in Edmonds-Karp algorithm.

   In each iteration of the algorithm, we find the *level graph* of the residual network G^f. Given a graph G with a source vertex s, its level graph $\overline{G}$ is defined by removing edges from G such that only edges pointing from level i to level $i+1$ are kept, where vertices at level i are those vertices at distance i from the source s. An example of level graph is shown in Fig. 1.

   Next, the algorithm finds a *blocking flow* on the level graph $\overline{G}^f$ of the residual network G^f. A blocking flow f in G is a flow such that each s-t path contains at least one *critical edge*. Recall that an edge e is *critical* if the amount of flow on it reaches its capacity: $f(e) = c(e)$. Fig. 2 gives examples for blocking flow.

   Dinic's algorithm is then described as follows.

   1. Initialize f to be the empty flow, and $G^f = G$.
   2. Do the following until there is no s-t path in G^f:
      - construct the level graph $\overline{G}^f$ of G^f.
      - find a blocking flow on $\overline{G}^f$.
      - Update f by adding the blocking flow to it, and update G^f.

   Complete the analysis of Dinic's algorithm by solving the following questions.

   (a) (15 points) Prove that, after each iteration of Dinic's algorithm, the distance from s to t in G^f is increased by at least 1.

   (b) (15 points) Design an $O(|V| \cdot |E|)$ time algorithm to compute a blocking flow on a level graph.

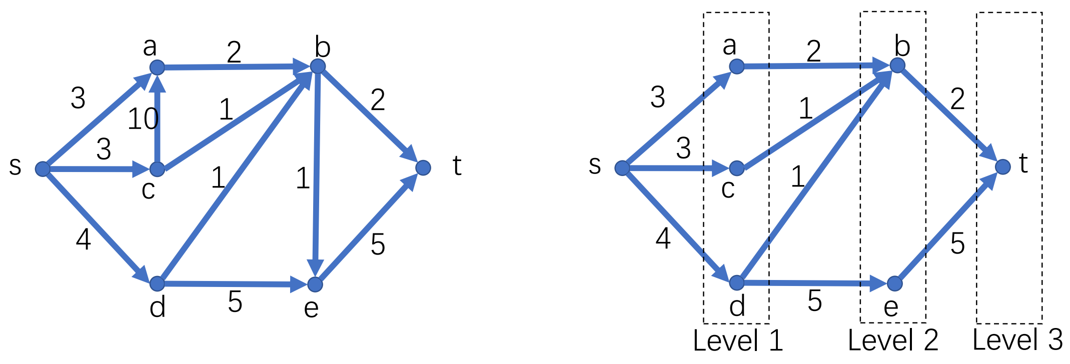


Figure 1: The graph shown on the right-hand side is the level graph of the graph on the left-hand side. Only edges pointing to the next levels are kept. For example, the edges (c, a) and (b, e) are removed, as they point at vertices at the same level. If there were edges pointing at previous levels, they should also be removed.

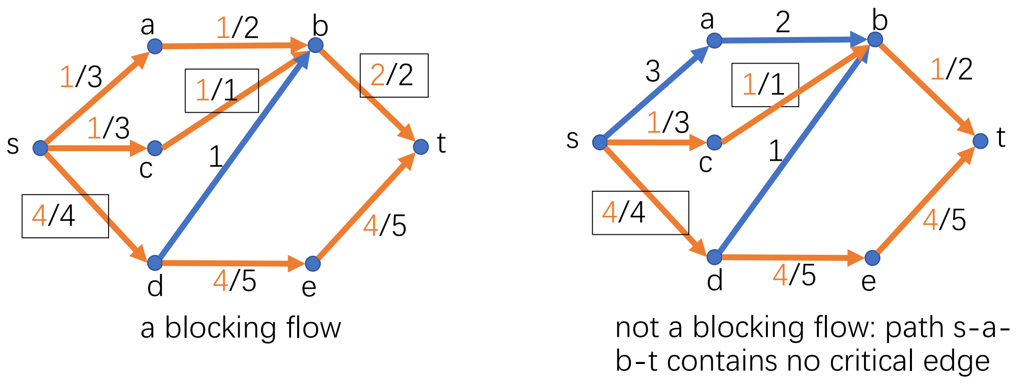


Figure 2: Blocking flow examples

(c) (10 points) Show that the overall time complexity for Dinic's algorithm is $O(|V|^2 \cdot |E|)$.

(d) (20 points) **(challenging)** We have seen in the class that the problem of finding a maximum matching on a bipartite graph can be converted to the maximum flow problem. Show that Dinic's algorithm applied to finding a maximum matching on a bipartite graph only requires time complexity $O(|E| \cdot \sqrt{|V|})$.

(a) Consider in the two iteration i and $i+1$, the distance from s to t in $i-th$ iteration is denoted as $d_i(s,t)$ and the level of t in $i-th$ iteration is denoted as $L_i(t)$. It's easy to observe that $d_i(s,t) = L_i(t)$, so we just need to show that $L_{i+1}(t) > L_i(t)$ because $L(t)$ can only be integer.
First, prove $L_{i+1}(t) \geq L_i(t)$. Consider any shortest path from s to t in G^f_{i+1} (G^f_{i+1} means the residual network in $i+1-th$ iteration). The path contains only two kinds of edges: 1. Edges from G^f_i. 2. Edges that are not from G^f_i. The second kind of edge(u,v) means $(v,u) \in G^f_i$ and $L_i(u) = L_i(v)+1$ by the nature of the algorithm that the blocking flow goes through (v,u). Now there is only two cases:
Case 1: If the path contains no second edge, then $L_{i+1}(t) \geq L_i(t)$ because the path also exists in $i-th$ iteration and it may not be the shortest in G^f_i.
Case 2: If the path contains the second edges. Assume the first such edge to be (u,v). Then $L_{i+1}(u) \geq L_i(u)$ as proved in case 1. By the above-mentioned nature of the second edge, $L_i(u) = L_i(v)+1$. Meanwhile, we have $L_{i+1}(v) = L_{i+1}(u)+1$. Combine the three equations and inequality, we have: $L_{i+1}(v) \geq L_i(v)+2$. Then, continue to find the second edge(u_1, v_1) in the path, we also have $L_{i+1}(u_1) \geq L_i(u_1)$ because there exists a path from v to u_1 and $L_{i+1}(v) \geq L_i(v)+2$...By repeating this method, we would finally have $L_{i+1}(t) \geq L_i(t)$.
Next, prove $L_{i+1}(t) > L_i(t)$. Suppose $L_{i+1}(t) = L_i(t)$, there is a shortest path from s to t in G^f_{i+1} that is exactly the same in G^f_i because the path can only contains only two kinds of edges and the second edge will only increase the distance in G^f_i. It is contradiction because the path is not blocked by the blocking flow. Therefore, combining two inequalities, we have $L_{i+1}(t) > L_i(t)$, which is what we need to prove the question.
(b) **Algorithm**
Use DFS to compute a blocking flow:

```
DFS(vertex $u$, vertex $t$, flow)
    if $u = t$ or flow$\leq 0$ return flow;
    Initialize ans$\leftarrow 0$.
    for every adjacent edge of u:
        if its another endpoint $v$'s level($v$) = level($u$)+1 and its flow < capacity:
            let k = DFS($v$, $t$, min(flow, capacity-flow))
        flow←flow-k and ans←ans+k
        update flow
    return ans
```

Time Complexity

The algorithm runs at most $O(|E|)$ rounds because in each round, it deletes at least one critical edge. And in each round, it go through $O(|V|)$ vertices so updating and others takes $O(|V|)$. Therefore, the overall time complexity is $O(|V||E|)$.

Correctness

In the algorithm, we push as mush flow as we can to each edge. If it isn't a blocking flow, we can still find a path from s to t that each edge has its $capacity - flow > 0$. However, it will never happen because the time we start DFS from s along this path, we will send the max flow to this path which is equal to the min(capacity-flow) of edges in this path by the nature of the algorithm, thus block such edge. It is a contradiction, so it is a blocking flow, thus proving the correctness of the algorithm.

(c) Each iteration increases the distance from s to t by at least 1, so the algorithm runs $O(|V|)$ rounds. Constructing the level graph can be achieved by BFS, which takes $O(|E|)$ to find the level. Finding a blocking flow takes $O(|V||E|)$ and updating takes $O(|E|)$. Therefore, the overall time complexity is $O(|V|^2|E|)$.

(d) To find a maximum matching means to find a max flow in a bipartite graph. Add two vertices s, t and let $c(s,u) = c(v,t) = 1, u \in A, v \in B$ (A, B are two sets of vertices where no edge connects two vertices in the same set). Then, let any edge point from u to v with $c(u,v) = 1$.

In this case, each iteration of the algorithm only takes $O(|E|)$. Constructing level graph takes $O(|E|)$. Finding the blocking flow here in the bipartite graph only takes $O(|E|)$ because each edge will be considered at most once (the capacity of each edge is equal to 1 so every time we push 1 flow to a path, thus every edge in the path being full and it's impossible to reach t from s along any of these edges). Updating takes $O(|E|)$. So each iteration takes $O(|E|)$.

Next, we aim to prove that the algorithm runs $O(\sqrt{|V|})$ rounds. Suppose the algorithm has run $\sqrt{|V|}$ rounds. By the conclusion of (a), every path from s to t is at least $\sqrt{|V|}$. Let f^* be the maximum flow and f be the current flow, then there is a flow with a value of $f^* - f$ in the current residual network. By the nature of the bipartite graph we constructed at the beginning of the question, the flow should be divided into $f^* - f$ vertex-disjoint paths with their length no less than $\sqrt{|V|}$. Since in each path, we have at least $\sqrt{|V|}$ vertices. So we have $(f^* - f)\sqrt{|V|} \leq |V|$ because the total number of vertices is no more than $|V|$, which implies that $f^* - f \leq \sqrt{|V|}$. Since in each round, the value of current flow will at least increase by 1 because of the Integral Theorem. So we still need at most $\sqrt{|V|}$ rounds to get the maximum flow. Therefore, the algorithm runs at most $\sqrt{|V|} + \sqrt{|V|} = O(\sqrt{|V|})$ rounds. Combined with the time complexity of each round $O(|E|)$, the overall time complexity to find a max-flow on a bipartite graph is $O(|E|\sqrt{|V|})$, which is equal to finding a maximum matching on a bipartite graph.

5. How long does it take you to finish the assignment (including thinking and discussion)? Give a score (1,2,3,4,5) to the difficulty. Do you have any collaborators? Please write down their names here.
   Two afternoons and nights.
   1. 5
   2. 5
   3. 4
   4. 5+

   Reference: https://cp-algorithms.com/graph/dinic.html