Basic problems:

1. [Dasgupta et al., Ex. 7.10] For the following network, with edge capacities as shown, find the maximum flow from $S$ to $T$, along with a matching cut:

   ![Network Diagram]

2. [Dasgupta et al., Ex. 5.1/5.2] Consider the following graph.

   ![Graph Diagram]

   (a) Run Kruskal’s algorithm on this graph. In what order are the edges added to the MST? For each edge in this sequence, give a cut that justifies its addition.

   (b) Run Prim’s algorithm on the same graph. Whenever there is a choice of vertices, always use alphabetic ordering (e.g. start from vertex $A$). Draw a table showing the intermediate values of the cost array.

   (c) How many minimum spanning trees does the graph have altogether?

3. [Dasgupta et al., Ex. 5.3] Consider the following task.

   Input: A connected, undirected graph $G$.

   Question: Is there an edge you can remove from $G$ while still leaving $G$ connected?

   Can you decide on the existence of such an edge in time $O(|V|)$? How about finding one?

   Hint: Consider the DFS search tree of the graph.

4. [Dasgupta et al., Ex. 5.20] Give a linear-time algorithm that takes as input a tree and determines whether it has a perfect matching: a set of edges that touches each vertex exactly once.

   Hint: If such a matching exists, then it is unique and can be determined bottom-up, starting from the leaves, by any systematic tree-search method (DFS, BFS). Mark first all the vertices as “unmatched” and change their status as “matched” and compute the corresponding matching as you move up (i.e. back towards your chosen starting vertex) in the tree.
Advanced problems:

5. [Dasgupta et al., Ex. 5.21] A feedback edge set of an undirected graph \( G = (V, E) \) is a subset of edges \( E' \subseteq E \) that intersects every cycle of the graph. Thus, removing the edges \( E' \) will render the graph acyclic.

Give an efficient algorithm for the following problem:

**Input:** Undirected graph \( G = (V, E) \) with positive edge weights \( w_e \).

**Output:** A feedback edge set \( E' \subseteq E \) of minimum total weight \( \sum_{e \in E'} w_e \).

**Solution:**

Let us prove the following property of a minimum \( fbes \) (feedback edge set, abbreviation defined for this solution only):

Let \( G \) be an undirected graph. \( E' \subseteq E \) is a minimum weight \( fbes \) of \( G \) iff \( E' = E \setminus E' \) is maximum weight spanning forest of \( G \).

By a spanning forest of a graph \( G \) with connected components \( G_1, G_2, \ldots, G_k \) we mean a sub-forest consisting of \( k \) trees, each spanning a different component \( G_i \).

**Proof.** Assume \( E_1 \) is a min-weight \( fbes \). This implies directly that \( E_1 \) is acyclic. For the sake of contradiction, assume \( E_1 \) is not connecting all nodes in some \( G_i \). Then one can move an edge \( f \) from \( E_1 \) to \( E_1 \) such that no cycles appear in \( E_1 \). Thus \( E_1 \setminus \{f\} \) is still intersecting each cycle while it has smaller weight than \( E_1 \), contradicting minimality of \( E_1 \). We conclude \( E_1 \) must be connecting each \( G_i \) - thus it must be a spanning forest.

Note that we did not yet argue why \( E_1 \) is actually a max-weight spanning forest. However, let us continue from assuming another set \( E_2 \subset E \) is a max-weight spanning forest. It has no cycles so \( E_2 \) must be a \( fbes \).

Let us calculate some weights now. Denote the weight of a set \( X \subset E \) by \( W_X = \sum_{e \in X} w_e \) and notice the property \( W_X = W_E - W_{E \setminus X} \). The max-weight property \( W_{E_2} \geq W_{E_1} \) implies \( W_{E_2} \leq W_{E_1} \). But \( E_1 \) was a min-weight \( fbes \), implying \( W_{E_2} = W_{E_1} \) and \( E_2 \) is a min-weight \( fbes \) as well. Similarly we get \( W_{E_2} = W_{E_1} \) and \( E_1 \) is max-weight spanning tree.

From the above, we can extract both parts of the standard “lhs-implies-rhs and rhs-implies-lhs” argument, and the proof is complete.

We remark that Kruskal’s min-weight spanning tree algorithm actually obtains a min-weight spanning forest for graphs that are not connected. A simple max-weight spanning forest algorithm is obtained from reversing the choice between large and small weights in Kruskal’s algorithm. The complement is the asked minimum \( fbes \).

A pseudocode is shown below. The union-find operations are run in the graph \((V, X)\).
Algorithm 1: Algorithm for finding min-weight feedback edge set in undirected graphs

1 function fbe-set(G);
   \textbf{Input:} Undirected graph } G = (V, E) \text{ with weights } w_e > 0 \forall e \in E
   \textbf{Output:} Minimum weight feedback edge set } E' \subset E
2 \textbf{for each} u \in V: \textbf{do}
3     \text{makeset}(u);
4 \textbf{end}
5 X = \{\}\;;
6 E' = \{\}\;;
7 \text{Sort the edges } E \text{ by weight};
8 \textbf{for all edges} \{u, v\} \in E, \text{ in decreasing order of weight: do}
9     \textbf{if} find(u) \neq find(v): \textbf{then}
10        add edge \{u, v\} to } X;
11        union(u; v);
12 \textbf{end}
13 \textbf{else}
14        add edge \{u, v\} to } E'
15 \textbf{end}
16 \textbf{end}
17 \textbf{return} E'\;;

6. [Dasgupta et al., Ex. 7.24] \textit{Direct bipartite matching.} Let } G = (V_1 \cup V_2, E) \text{ be a bipartite graph (so that each edge has one endpoint in } V_1 \text{ and one in } V_2\text{), and let } M \subseteq E \text{ be a matching in the graph (that is, a set of edges that don’t touch). A vertex is said to be \textit{covered} by } M \text{ if it is the endpoint of one of the edges in } M. \text{ An alternating path is a path of odd length that starts and ends with a non-covered vertex, and whose edges alternate between } M \text{ and } E \setminus M.

(a) In the bipartite graph below, a matching } M \text{ is shown in bold. Find an alternating path.

(b) Prove that a matching } M \text{ is maximum if and only if there does not exist an alternating path with respect to it.

(c) Design an algorithm that finds an alternating path in } O(|V| + |E|) \text{ time using a variant of breadth-first search.

(d) Give a direct } O(|V| \cdot |E|) \text{ algorithm for finding a maximum matching in a bipartite graph.

3
Solution:

(a) Some examples, with $\equiv$ denoting an edge in $M$:

$$D - F = B - E = A - H = C - G$$

$$D - F = B - E = A - H = C - I$$

$$D - F = B - G.$$

(b) If for a matching $M$ there exists an alternating path $P$ with edges $E_P$, consider the edge set defined as the symmetric difference $M' = M \triangle E_P = (M \setminus E_P) \cup (E_P \setminus M)$ with size $|M'| = |M| + 1$. In other words, moving from $M$ to $M'$ flips the inclusion of each $E_P$. Since the endpoints of $P$ are not endpoints of $M$ edges and $M'$ touches no other new vertices, we conclude $M'$ is a matching as well.

Thus, the existence of an alternating path for a matching $M$ implies $M$ is not a maximum matching.

For the other implication, assume $M$ is not a maximum matching. Then there exists a matching $M'$ with $|M'| > |M|$. Consider the multi-set $M'' = M \uplus M'$ of edges and the corresponding multi-graph $H = (V, M'')$. In this graph, no vertex has degree greater than two, so each connected component must be a path or a cycle.

Note that no path or cycle in $H$ can have two consecutive edges in either $M$ or $M'$. This follows because both $M$ and $M'$ were matchings. In particular, there cannot be a cycle of odd length in $H$.

Let us now look on the balance between $M$ and $M'$ edges. Since there were more $M'$ than $M$ edges, at least one connected component of $H$ must have also more $M'$ than $M$ edges. Due to the alteration of $M$ and $M'$ edges along the path, this is possible only if the component is a path both starting and ending with an $M'$ edge. Now this path is an alternating path with respect to $M$. The proof is finished.

(c) Assume $L$ and $R$ are the two parts such that each edge has one endpoint in $L$ and the other in $R$. (If these sets are initially unknown, a simple dfs search can be used to find the partition in $O(|V| + |E|)$ time.)

The key variation of the standard bfs is to use only $M$ edges when traveling from $R$ to $L$ and $E \setminus M$ edges when traveling from $L$ to $R$. One needs to store the to-be alternating path itself during the run, and as the bfs never revisits a node, one can just keep track of each node’s unique predecessor on the path candidates. The pseudocode is presented in Algorithm 2.
Algorithm 2: Algorithm for finding an alternating path

function alternating-path-bfs(G, M);

Input: Undirected bipartite graph G = (V, E), matching M ⊆ E

Output: alternating path in G, or empty path if no such path exists

for v ∈ V do
  mark v unvisited;
  mark v uncovered;
end

for e ∈ M do
  mark both endpoints of e covered;
end

use bipartiteness to mark each v ∈ V as belonging to L or R;

Q ← empty queue;
for each uncovered v ∈ L do
  mark v visited;
  v.predecessor ← none;
  if v has a neighbour then
    Q.add(v);
  end
end
while Q not empty do
  v ← Q.take first;
  for nbr neighbour of v do
    if nbr unvisited then
      if v ∈ L and (v, nbr) /∈ M then
        Q.add(nbr);
        nbr.predecessor ← v;
        if nbr uncovered then
          extract path P by going through predecessors recursively from nbr;
          return P;
        end
      end
      else if v ∈ R and (v, nbr) ∈ M then
        Q.add(nbr);
        nbr.predecessor ← v;
      end
    end
  end
end
return empty path
What is the running time of Algorithm 2? The initialization part up to line 17 consumes clearly $O(|V| + |E|)$ time. Only the while loop remains to be analyzed. Note that each $v$ inserted into $Q$ must be an endpoint of an edge, and each $v$ is pulled from $Q$ and checked for neighbours at most once. Thus the while-for selection on lines 18-20 is done in $O(|E|)$ time, with at most this many calls to the inner parts. These inner parts of the for loop starting at line 20 are run in constant time per iteration, except for the final processing step at line 26 which may cause an extra $O(|E|)$ once. Thus the while loop has running time $O(|E|)$ in total, and the whole algorithm has running time $O(|V| + |E|)$.

(d) Here we exploit the previous algorithm finding alternating paths, increasing the size of the matching by knowledge of the explicit alternating paths.

**Algorithm 3: Algorithm for finding maximum matching in a bipartite graph**

```
1 function maximum-matching:
   Input: Undirected bipartite graph $G = (V,E)$
   Output: Maximum matching $M \subseteq E$ in $G$
2 $M \leftarrow \emptyset$
3 while true do
4   $P \leftarrow$ alternating-path-bfs($G,M$);
5   if $P$ is empty path then
6     return $M$
7   end
8   for each edge $e$ on $P$ do
9     if $e \in M$ then
10        $M \leftarrow M \setminus \{e\}$
11     end
12     else
13        $M \leftarrow M \cup \{e\}$;
14     end
15   end
16 end
```

To find an upper bound to the running time, we argue that at most $O(\min\{|E|,|V|\})$ iteration rounds are needed. Indeed, at each round of finding an alternating path we put one more edge to $M$. As $|M| \leq |E|$, the process is guaranteed to finish after at most $|E|$ rounds. On the other hand, at each round we also make two previously exposed vertices unexposed. After $|V|/2$ rounds there cannot be any more exposed edges, and the process must have finished before $|V|$ rounds. Thus, $\min\{|E|,|V|\}$ is an upper bound to the number of iterations before the algorithm finishes.

Now the inner algorithm had running time $O(|V| + |E|)$, and

$$(|V| + |E|) \min\{|E|,|V|\} \leq 2|V||E|.$$ 

Thus the maximum matching algorithm has running time $O(|V||E|)$, as desired.