CS 577-2: Introduction to Algorithms Fall 2019

Solutions to the Midterm 2 Practice Problems

Instructor: Dieter van Melkebeek

Problem 1

Part (a)

Intuitively, we want the first components of the pairs in P to be as large as possible, and the second components of the pairs in P to be as small as possible. This suggests the following greedy strategy could work: pair the largest n elements with the smallest n elements. The exact pairing doesn't matter.

We need to prove that this solution works. Let P be an arbitrary pairing. We show that the greedy pairing performs at least as well as P. We use a greedy stays ahead approach.[1]

Let L be the set of indices appearing as the first component in a pair in P, and let R be the set of indices appearing as the second component in a pair in P. For $k = 0, 1, \ldots, n$, let L_k be the largest k elements of L, and let R_k be the smallest k elements of R. (If $k > |L| = |R|$, set $L_k = L$ and $R_k = R$.)

Similarly with respect to the greedy solution, for $k = 0, 1, \ldots, n$, let L_k^* be the indices of the largest k numbers among $a_1, \ldots, a_{2n}$, and let R_k^* be the indices of the smallest k numbers. The greedy pairing is a way to pair elements of L_n^* with elements of R_n^*.

Our greedy-stays-ahead argument is captured by the following claim. Specialized to $k = n$, the left-hand side is the value of the greedy solution, and the right-hand side is the value of P, so this proves optimality of the greedy solution.

Claim 1. *For all $k = 0, \ldots, n$,*

$$\sum_{i \in L_k^*} a_i - \sum_{i \in R_k^*} a_i \geq \sum_{i \in L_k} a_i - \sum_{i \in R_k} a_i.$$

Proof. We argue this by induction on k from 0 to n.

Base case: When $k = 0$, both sides are empty sums, which are zero, and $0 \geq 0$.

Inductive step: For $n \geq k > 0$, let ℓ^* be the index of the k-th largest number among $a_1, \ldots, a_{2n}$, and let r^* be the index of the k-th smallest number. We can write

$$\begin{aligned}\sum_{i \in L_k^*} a_i - \sum_{i \in R_k^*} a_i &= \left(\sum_{i \in L_{k-1}^*} a_i - \sum_{i \in R_{k-1}^*} a_i \right) + (a_{\ell^*} - a_{r^*}) \\ &\geq \left(\sum_{i \in L_{k-1}} a_i - \sum_{i \in R_{k-1}} a_i \right) + (a_{\ell^*} - a_{r^*}).\end{aligned}$$

[1] One can also use an exchange argument. See claim 4 in part (b).

where the inequality follows by the inductive hypothesis. In the case where $k > |L|(= |R|)$, $L_k = L_{k-1}$ and $R_k = R_{k-1}$, so the claim follows since $a_{\ell^*} \geq a_{r^*}$. When $k \leq |L|$, let ℓ be the unique element in $L_k \setminus L_{k-1}$, and let r be the unique element in $R_k \setminus R_{k-1}$. It suffices to show that $a_{\ell^*} - a_{r^*} \geq a_\ell - a_r$. By definition, a_ℓ is smaller than at least $k-1$ other elements among $a_1, \ldots, a_{2n}$, namely the elements in L_{k-1}. So $a_{\ell^*} \geq a_\ell$. Analagously, a_r is larger than the $k-1$ elements in R_{k-1}, so $a_{r^*} \leq a_r$. Together, these imply $a_{\ell^*} - a_{r^*} \geq a_\ell - a_r$.

□

Given the optimality of the greedy solution, it remains to describe how to compute such a pairing in linear time. The idea is to find the median element among $a_1, \ldots, a_{2n}$ in linear time, and then use that to partition it into the larger and smaller halves in linear time. From here, computing a final pairing takes linear time.

Part (b)

As in part (a), intuitively, we want the first components of the pairs in P to be as large as possible, and the second components to be as small as possible. For the purposes of part (b), this is easier when we assume that the sequence of values is sorted in decreasing order, i.e.,

$$a_1 > a_2 > \ldots > a_{2n}. \tag{1}$$

This is without loss of generality, since we can afford $O(n \log n)$ running time in part (b).

With this assumption in hand, our intuition suggests to pair, for some k, the first k numbers with the last k numbers. In particular, we want to make k as large as possible (up to n). Intuitively, this is best accomplished by pairing the two sequences of k numbers in order. All together, we make the following claim:

Claim 2 (Key Observation)**.** *There exists an optimal solution P^* such that the following holds. Let L^* be the set of first-components of pairs in P^*, and let R^* be the set of second-components. Then L^* is a prefix of $[2n]$ and R^* is a suffix of $[2n]$, and the elements are paired up in order. In other words,*

$$P^* = \{(i, 2n - k - 1 + i)\}_{i=1}^{k}. \tag{2}$$

for some $0 \leq k \leq n$.

We prove the claim below. Assuming it holds, what remains is to efficiently determine the optimal value k^* of k for part (b). Since larger values of k can only increase the value of the objective function, k^* equals the largest nonnegative $k \leq n$ such that (2) is a valid solution.

To do this, given that $a_1, \ldots, a_{2n}$ is already sorted, observe that we can test whether (2) is a valid solution for a given value of k in time $O(n)$ by checking whether $a_i \geq 2a_{2n-k-1+i}$ holds for each $i \in [k]$. Morover, this property is monotone in k. Thus, we can use a binary search to determine k^*. This requires testing $O(\log n)$ values of k, for a total running time of $O(n \log n)$ (with or without sorting). Once we have k^*, we can compute the maximum objective value in linear time. This results in an overall running time of $O(n \log n)$.

Algorithm 1 Finding k^*

Input: $a_i \in \mathbb{R}^+$ for $i = 1, 2, \ldots, 2n$ satisfying (1)
Output: k^*

```
k ← 0
i ← 1
j ← n + 1
while j ≤ 2n do
    if a_i ≥ 2a_j then
        k ← k + 1 (including the pair (i, j))
        i ← i + 1
    j ← j + 1
return k
```

Alternate solution As a side note, we point out that once the sequence is sorted, the optimal value k^* can actually be found in linear time using the following pointer-based approach.

This algorithm runs in time $O(n)$. Its correctness follows from the following greedy-stays-ahead argument combined with the fact that in an optimal solution P^* no second component is smaller than $n+1$. Let the pairs the above algorithm implicitly includes be $\{(1, \widehat{j}_1), (2, \widehat{j}_2), \ldots, (\widehat{k}, \widehat{j}_{\widehat{k}})\}$.

Claim 3. *For every valid solution $P = \{(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)\}$ with $i_1 < i_2 < \ldots < i_k$ and $n+1 \le j_1 < j_2 < \cdots < j_k$, it is the case that $\widehat{k} \ge k$ and $\widehat{j}_k \le j_k$.*

Proof. By induction on k.

Base case: When $k = 0$, the claim holds vacuously. When $k = 1$, we observe that since the pairing (i_1, j_1) exists in P, it holds that $a_{i_1} \ge 2 \cdot a_{j_1}$. Since the a's are sorted, $a_1 \ge 2 \cdot a_{j_1}$ as well. Since $j_1 \ge n+1$, it follows that Algorithm 1 finds some index $j \le j_1$ so that $a_1 \ge 2 \cdot a_{j_1}$. This implies that $\widehat{k} \ge 1 = k$, and $\widehat{j}_1 = j \le j_1$.

Inductive step: For $k > 1$, let $P' = \{(i_1, j_1), (i_2, j_2), \ldots, (i_{k-1}, j_{k-1})\}$ be the first $k-1$ pairs in P. By the inductive hypothesis, we have $\widehat{k} \ge k-1$ and $\widehat{j}_{k-1} \le j_{k-1}$. Since the pair (i_k, j_k) exists in P, we have that $a_{i_k} \ge 2 \cdot a_{j_k}$. Since also $i_k \ge k$, we have $a_k \ge a_{i_k}$, so $a_k \ge 2 \cdot a_{j_k}$. Now consider Algorithm 1 from the first iteration of the loop when $i = k$ (just after adding $(k-1, \widehat{j}_{k-1})$ to $\widehat{P}$). We have $j = \widehat{j}_{k-1} + 1$. Since $\widehat{j}_{k-1} \le j_{k-1} < j_k$, it also holds that $j \le j_k$. Since $a_k \ge 2 \cdot a_{j_k}$, it follows that over the next few iterations, the algorithm will find an index $j \le j_k$ so that $a_k \le 2 \cdot a_j$. Therefore $\widehat{k} \ge k$, with $\widehat{j}_k = j \le j_k$. This establishes the inductive step. □

Proof of the Key Observation

All that remains is to prove Claim 2. We do so in two steps, using an exchange argument for both.

Claim 4. *There exists an optimal solution $P = \{(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)\}$ such that*

$$i_1 < i_2 < \ldots < i_k < j_1 < j_2 < \ldots < j_k. \tag{3}$$

Proof. First note that for every pair $(i, j) \in P$, it has to be the case that $i < j$. Otherwise, replacing (i, j) in P by (j, i) would result in a higher objective value.

Consider any two pairs (i, j) and (i', j') in P with $i < i'$. By the above, $i < j$ and $i' < j'$. In principle, this leaves three possible orderings:

1. $i < i' < j < j'$
2. $i < i' < j' < j$
3. $i < j < i' < j'$

We argue that case 3 cannot happen, and exhibit an exchange for case 2 that transforms it into case 1 while maintaining validity and without decreasing the objective value.

First consider case 3. We have that $a_i \geq 2a_j$ (since the pair (i, j) is valid), $a_j \geq a_{i'}$ (since $j < i'$), so $a_i \geq 2a_{i'}$, i.e., the pair (i, i') is valid. For similar reasons, the pair (j, j') is valid. This means we can replace $\{(i, j), (i', j')\}$ by $\{(i, i'), (j, j')\}$ in P while preserving validity. This has the effect of changing the sign of $a_{i'}$ in the objective function from $+$ to $-$, and the sign of a_j from $-$ to $+$. Since $j < i'$, we have $a_j > a_{i'}$, and hence the net change in the objective value is positive, This implies that case 3 cannot occur in an optimal solution.

In case 2, the fact that $a_{i'} \geq 2a_{j'}$ (since the pair (i', j') is valid) and the presumed ordering implies that $a_i i \geq 2a_{j'}$ and $a_{i'} \geq 2a_j$, i.e., the pairs (i, j') and (i', j) are valid. Thus, we can use those pairs to replace the original pairs (i, j) and (i', j'). This exchange does not affect the value of the objective function.

Consider any optimal solution $P = \{(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)\}$. By relabeling we can always make sure that $i_1 < i_2 < \ldots < i_k$. If (3) fails, there have to be two consecutive pairs (i, j) and (i', j') with $i = i_\ell$ and $i' = i_{\ell+1}$ that are in case 2. Applying the above exchange reduces the number of inversions of the sequence of second components with respect to the sequence of first components, without affecting the objective value. Keep repeating this process reduces the number of inversion to zero, yielding a valid solution satisfying (3) and of optimal objective value. □

Claim 5. *There exists an optimal solution $P^* = \{(i_1, j_1), (i_2, j_2), \ldots, (i_k, j_k)\}$ satisfying (3) and such that $L^* \doteq \{i_1, i_2, \ldots, i_k\}$ is a prefix of $[2n]$, and $R^* \doteq \{j_1, j_2, \ldots, j_k\}$ is a suffix.*

Proof. If an index i is skipped, replacing the next index $i' \in L^*$ by i maintains validity and improves the objective value. A similar statement holds if an index j is skipped. Thus, no indices can be skipped in L^* or in R^* for an optimal solution P^*. □

Problem 2

At first glance, this problem presents itself as a kind of matching problem: match team members to seats at a table. This suggests using a flow-based approach to find the matching. This can be made to work: make a bipartite graph with one left vertex per team, one right vertex per table, and two additional nodes s and t. For each team, add an edge from s to the team's vertex with capacity the size of that team. For each table, add an edge from that table's vertex to t with capacity the capacity of that table. From each team to each table, add an edge with capacity one. Then there is a way to assign team members to table-seats if and only if this network has a maximum flow whose value is the total number of team members.

That is all correct; however, such an algorithm will not satisfy the running time requirements for this problem. We will have to try a different strategy.

To get some intuition, let's consider some extremal cases. One possibility is that some team has size equal to the number of tables. In that case, we have to give that team one spot at every table. The next simplest case is when a team has size one less than the number of tables. Then the choice is to decide which table should *not* get a member from that team. A natural heuristic is to leave out a table with the smallest capacity. After some thought, we can actually prove that choice will always work with the following exchange argument.

Let j^* be the table with smallest capacity, and suppose team 1 is the team assumed to have size $m-1$. Suppose there exists a way to seat the teams at the tables, and let j be the table with no members of team 1 under this arrangement. If $j = j^*$, then we are done. Otherwise, it suffices to rearrange team members so that the member of team 1 at j^* is moved to a different table. We move that team member to table j as follows. If table j is not full, then we just make the move and we are done. Otherwise, table j is full, and we have a situation as in figure 1. In this case, members from t_j different teams are seated at table j, and none are from team 1. Meanwhile, there are at most $t_{j^*} - 1$ teams beside 1 seated at table j^*. Since j^* is a table with smallest capacity, $t_{j^*} - 1 < t_j$, so there is some team with a member at table j and with no member at table j^*. It follows that we can swap that person and the member of team 1 and have a feasible assignment of team members to tables (see figure 1). After the swap, table j^* has no member of team 1, as desired. This completes the exchange argument.

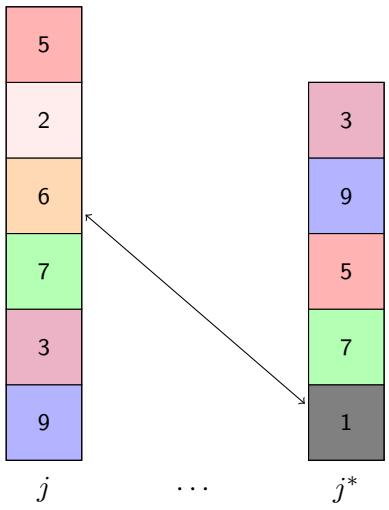

Figure 1: Example demonstrating the exchange done in the case where table j is full. Table j is at least as large as table j^* (here it is drawn strictly larger), and has no member of team 1, so there exists a team (here, 2 or 6; the figure chooses 6) that has a member at table j with no teammates at table j^*.

With that observation in hand, we take a second look at the argument just made. The assumption that the size of team 1 is $m-1$ is not actually required to make the exchange. While that assumption helps us pick j^* and j, we can see that no matter the size of team 1, a similar exchange is possible as long as table j^* has a member of team 1, table j has no members of team 1, and—crucially—table j has capacity at least that of table j^*. (Contrast with a modification of

figure 1, in which both the team 2 and team 6 members are removed from table j.) Indeed, for any feasible assignment of teams to tables, and any team 1, if the arrangement does not place a member of team 1 at table j, but does place one at a table j^* with smaller capacity, then there is an exchange so that team 1 instead places that person at table j. In other words, the members of team 1 may as well be placed at the tables with the largest capacity. That leads to the following greedy strategy.

Algorithm Consider all the teams one by one (in an arbitrary order). For each team i, we assign that team's members to the s_i tables *with largest remaining capacity*. If this is impossible, then we output that no arrangement is possible. Otherwise, after all teams are considered, we say that there exists an arrangement of the teams to tables. The following pseudocode gives one way to implement this; since we don't need to output the arrangement itself, we just keep track of the remaining capacity of each table.

Algorithm 2 Seat Assignment

Input: $S = \{s_1, s_2, \ldots, s_n\}$ and $T = \{t_1, t_2, \ldots, t_m\}$.
Output: YES if seating arrangement is possible; NO otherwise.

```
Q ← {t_j ∈ T | t_j > 0}
for i = 1, ..., n do
    if |Q| < s_i then
        return NO
    Extract the largest s_i values from Q; call them t_{ℓ_1}, t_{ℓ_2}, ..., t_{ℓ_{s_i}}
    for j = 1, ..., s_i do
        t_{ℓ_j} ← t_{ℓ_j} − 1
        if t_{ℓ_j} > 0 then
            Push t_{ℓ_j} into Q
return YES
```

Correctness We precisely formulate and prove the general version of the exchange argument stated above in order to prove the correctness of our algorithm. It suffices to prove the following claim.

Claim 6. *Let there be n teams of sizes $s_1, s_2, \ldots, s_n$, and m tables, of capacities $t_1 \geq t_2 \geq \cdots \geq t_m$. Suppose there is a feasible assignment for all these n teams. Then there exists a feasible assignment in which we just assign team 1 to the first s_1 seats.*

Correctness follows since, once team 1 is assigned to the first s_1 tables, the remaining problem is the same, except with only teams $2, \ldots, n$ present and the first s_1 tables each having one less capacity. So (after possibly reordering the tables) the claim says we can seat team 2 members at the largest s_2 remaining tables, and so on.

Proof. By an exchange argument. Suppose that some assignment places all the teams at the tables. Suppose that for some table $j \leq s_1$, team 1 has no member assigned to table j. Then there is a table $j^* > s_1$ containing a member of team 1. If table j is not full, we just move the team 1 member from table j^* to table j. Otherwise, table j is full, so t_j teams have members at that table. At most

$t_{j^*} - 1$ teams besides team 1 have members at table j^*. Since the tables are sorted by capacity, and $j \leq s_1 < j^*$, we have $t_j > t_{j^*} - 1$. So there is a person at table j with no teammates at table j^*. So we can swap them with the team 1 member at table j^*. (See figure 1 for an illustration.)

With each step of the exchange, we decrease the number of tables among the first s_1 that have no members of team 1. We can do a swap as long as the number of such tables is positive. So after some sequence of swaps, the assumed feasible assignment is transformed into one where team 1 has a member seated at tables $1, \ldots, s_1$. This is what we set out to prove. □

Implementation details and run-time analysis We need to create Q, and then for each $1 \leq i \leq n$, we need to extract the largest s_i values from Q, update them, and possibly replace them in Q. Using a priority queue for Q, the initial creation takes $O(|Q|)$ time, and each extract-update-replace sequence costs $O(\log |Q|)$ time. As $|Q| \leq m$, this leads to a running time of $O\left(m + (\sum_{i=1}^{n} s_i) \log m\right) = O(N \log N)$.

Problem 3

We solve this problem using network flow. First note that we can construct a directed graph $G = (V, E)$ from the input by creating one vertex v_i for each coupon i and adding an edge from v_i to v_j if v_i can be exchanged for v_j. Note that we interpret coupon exchanges as being unidirectional. To model the problem using bidirectional exchanges it suffices to add edges in both directions and proceed in the same way. Figure 2 shows the graph representation for the example given in the problem statement:

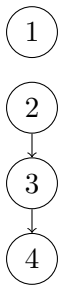

Figure 2: Graph representation of the relations between coupons given as an example.

Now we want to transform this graph into a network N in a way that a maximum flow in N represents a strategy for trying to get at least one of each coupon. Let c_i for $i = 1, 2, \ldots, n$ represent how many coupons of type i we have. We first add a source vertex s and add edges from s to each v_i with capacity c_i, signaling that we have this many coupons of type i. We also add a sink vertex t and add edges from each v_i to t with capacity 1, representing that in the end we want exactly one of each coupon. Lastly, as the exchanges we make are only limited to how many coupons of each type we have, we assign capacity ∞ to the edges connecting two vertices representing coupons that can be exchanged. Figure 3 shows the network we construct from the example given in the problem statement.

After constructing N, we compute a maximum flow f for N and output yes if and only if the value of the flow is n, i.e. $\nu(f) = n$.

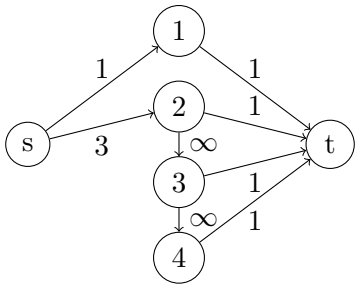

Figure 3: Network constructed to solve the example input.

Correctness Intuitively, correctness follows from the observation that if there is a flow of value n in N, then there is a way to "redistribute" the extra coupons we have in a way that makes it possible to obtain every single coupon, and if there is not then this is just not possible. As we have edges of infinite capacity between coupons that can be traded, the only limits are the ones imposed by how many coupons we have of each type, which are realized by respecting conservation of flow. More formally, we can argue correctness through the following claim.

Claim 7. *Let N be the network constructed by the algorithm and f a maximum flow in N, then $\nu(f) = n$ if and only if there is a sequence of coupon exchanges that ends with one coupon of each type.*

Proof. ($\implies$) Assume $\nu(f) = n$. As all edge weights are integer, there is an integer valued flow f' such that $\nu(f') = n$ as well. To show this implies there is a sequence of coupon exchanges that ends with one coupon of each type, we construct such a sequence based on f'. For each unit of flow that passes from a vertex representing a coupon to another, make that exchange. Conservation of flow implies that every exchange made in this way is valid, as we can only exchange a coupon if we actually had it in the first place. As the fact that $\nu(f') = n$ implies that every edge from vertices representing coupons to t has exactly one unit of flow pushed though it, this means this sequence of exchanges ends with exactly one coupon of each type, as desired.

($\impliedby$) Assume there is a sequence of coupon exchanges that ends with one coupon of each type. We use this to construct a flow of value n in N. First push c_i units of flow in each edge connecting s to vertex v_i. For each exchange, push one unit of flow in the edge connecting the coupon that is traded to the one it is traded for. As we can only trade coupons if we had them in the first place and we end up with one coupon of each type, this means after pushing all flows in this manner there is still one unit of flow that needs to be pushed from each vertex v_i. By pushing one unit of flow from each v_i to t, we guarantee conservation of flow, and since n units of flow leave s, this flow has capacity n. □

Running time Given n coupons as input, we construct a network N that has $n + 2$ vertices and at most $n^2 + 2n = O(n^2)$ edges. This network can be constructed in time $O(n^2)$. We then compute a maximum flow in N, which takes time $O(n \cdot n^2) = O(n^3)$, and finally compare the value of the flow to n in constant time. This results in an algorithm that runs in time $O(n^3)$, which is polynomial in n.

Problem 4

We reduce this problem to computing the maximum value of a flow in a network. For each point (x_i, y_i) we first find all possible candidates to be in B_i. That is we find all points at distance less than or equal to d from (x_i, y_i). Let's call all the sensors that are at distance at most d from i the neighbors of i.

We now describe how to construct the flow network. We construct a bipartite graph with $2n$ nodes, that is 2 nodes per sensor, one on the left side and one on the right side. We connect with a directed edge of capacity 1, each node of the left side to its neighbors on the right side. To model the constraint that the cardinality of each set should be exactly r we set the capacities of the edges (s, i) to r. Finally, to model the constraint that each node should not belong to more that b sets B_i we set the capacities of the edges (i, t) to b. See Figure 4 for an illustration.

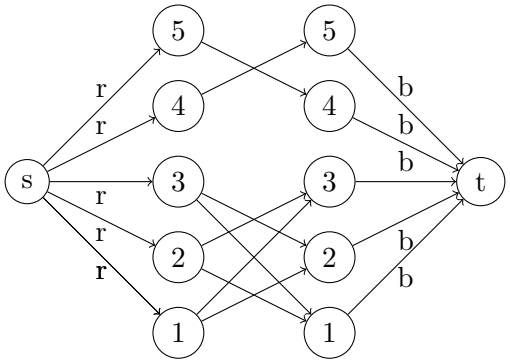

Figure 4: Flow network for Problem 4. The graph corresponds to a triangle of 3 points $1, 2, 3$ that have pairwise distances smaller than d and a pair of points $4, 5$ which are close to each other but far from $1, 2, 3$.

Since $r \leq b$ we have that the maximum flow is at most rn. Given an integral flow in this network we can construct sets B_i as follows. For each node i of the right side we include in B_i the nodes of the left side that are connected with an edge that carries 1 unit of flow. Since the flow is integral, the value of the maximum flow is rn if and only if sets B_i with the desired properties exist.

Since the network can be constructed in time polynomial in n, and the value of the maximum flow can be computed in polynomial time, the resulting algorithm runs in time polynomial in n.

Problem 5

Let's try to model this problem as a graph problem. To do so, we first construct an undirected graph $G = (V, E)$ that represents our problem. Given a set W of N words $w_1, w_2, \ldots, w_n$, we include one vertex v_i for each word w_i. Then for each pair of words w_i, w_j with $i < j$, we add an edge between both their vertices if w_i can be transformed into w_j by a single exchange. Now that we have the graph G, finding the size of the largest swap-free set in W reduces to computing the maximum size of an independent set in G.

There is no known efficient algorithm for finding an independent set in an arbitrary graph, and we may think this approach is doomed to fail. However, there is a crucial observation that allows

us to solve the problem efficiently for G, namely that G is bipartite. This may not be immediately clear. Think about each word in comparison with the alphabetically first word in the set, which we call w. Since every word is an anagram of w, each one has either an even or odd number of inversions in relation to w, and if word w_i is connected to word w_j by an edge, then one must necessarily have an odd number of inversions and the other an even number of inversions, as making a swap changes the parity of the number of inversions of a word in relation to w. Thus, we can partition the vertices into two groups – those with an even number of inversions with respect to w, and those with an odd number – such that there are no edges between vertices of the same group.

As we have seen in Homework 7, Exercise 2, the problem of finding a vertex cover in bipartite graphs can be solved efficiently by exploiting connections with Bipartite Matching and Maximum Flow. We have also seen in class that Vertex Cover and Independent Set are closely connected, and interreduce efficiently. We can combine these results to obtain an efficient algorithm for the Independent Set problem in bipartite graphs.

There is also a more direct way to reduce the Independent Set problem for bipartite graphs to the Minimum Cut problem, which we explore next. Suppose we have a bipartition of $G = (V, E)$ into $L, R \subseteq V$, where $L \cup R = V$, $L \cap R = \emptyset$, and $E \subseteq L \times R$. We do as in the reduction from Bipartite Matching to Minimum Cut from class: we add vertices s and t to G, add edges from s to each vertex in L with capacity 1, add edges from each vertex in R to t with capacity 1, orient the edges between L and R so that they go from vertices in L to vertices in R and give them infinite capacity to construct a network G'. Consider any st-cut (S, T) of finite capacity in G'. There can be no edge between L and R crossing the cut from the S-side to the T-side as such an edge would contribute ∞ to the capacity of the cut. Thus, the vertices in L that belong to S and the vertices in R that belong to T from an independent set $I = (L \cap S) \cup (R \cap T)$ of G. Moreover, as the edges that contribute to the capacity of the cut are exactly those that connect s to a vertex in $L \cap T$ and those that connect a vertex in $R \cap S$ to t, which all have unit capacity, it follows that

$$c(S,T) = |L \cap T| + |R \cap S| = (|L| - |L \cap S|) + (|R| - |R \cap T|) = (|L| + |R|) - (|L \cap S| + |R \cap T|) = N - |I|. \tag{4}$$

See Figure 5 for an illustration. In fact, we have the following.

Claim 8. *There is a one-to-one correspondence between st-cuts (S, T) of finite capacity in G' and independent sets I in G. Moreover, $|I| + c(S, T) = N$.*

Proof. We already explained how to get I out of (S, T). In the other direction, given I we can construct S as $S = \{s\} \cup (I \cap L) \cup \left(\overline{I} \cap R\right)$ and $T = \{t\} \cup \left(\overline{I} \cap L\right) \cup (I \cap R)$. Then none of the middle edges contribute to the capacity of the cut, and the contribution of the other edges is as described in (4). See again Figure 5. □

The claim implies that the maximum size of an independent set in the bipartite graph equals $N - \min(c(S, T))$, where $(S.T)$ ranges over all st-cuts in G'.

Implementation details To construct the graph G, we compare every pair of words w_i, w_j, with $i < j$, symbol by symbol to decide if a single swap can transform one word into the other, and if this is possible we add an edge (v_i, v_j) connecting the vertices that represent each word. More precisely, we count for each pair of words in how many positions their symbols differ. If this happens for exactly two positions, we add an edge between both, and if it happens for more, we do not. Then we use breadth-first search to, in linear time, find a bipartition of V into $L, R \subseteq V$ such that $L \cup R = V$, $L \cap R = \emptyset$, and $E \subseteq L \times E$. After that, we construct the network G' as

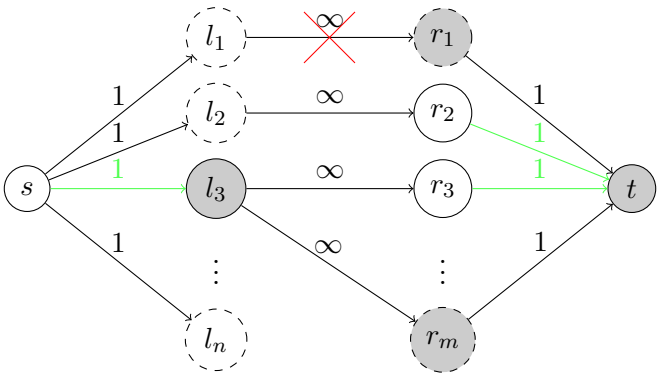

Figure 5: Vertices in white are in S, vertices in light gray are in T. Vertices that are part of I are depicted by dashed circles. Note there will never be an edge connecting two elements in I like l_1 and r_1 because the capacity of the cut is finite. Edges that contribute to the capacity of the cut are depicted in green.

mentioned before, compute a maximum flow f in G', and return $N - \nu(f)$ as the size of the largest independent cut in G. This works because $\nu(f) = \min(c(S,T))$ by the max-flow/min-cut theorem.

Running time We compare $O(N^2)$ pairs of words, and since each word can only have lowercase letters and no repetitions, there is a limit of 26 for the number of symbols in each word. This means we can compare each pair of words in constant time and construct G in time $O(N^2)$. Constructing G' out of G again takes time $O(N^2)$, and computing a maximum flow in G' takes time $O(N^3)$ as G' has $N + 2$ vertices and $O(N^2)$ edges. This results in a total running time of $O(N^3)$.