H9:ER6QT

Collaboration Policy: You are encouraged to collaborate with up to 4 other students, but all work submitted must be your own *independently* written solution. List the computing ids of all of your collaborators in the `collabs` command at the top of the tex file. Do not seek published or online solutions for any assignments. If you use any published or online resources (which may not include solutions) when completing this assignment, be sure to cite them. Do not submit a solution that you are unable to explain orally to a member of the course staff. **Please remember that you are not allowed to share any written notes or documents (these include but are not limited to Overleaf documents, LaTeX source code, homework PDFs, group discussion notes, etc.). Any solutions that share similar text/code will be considered in breach of this policy. Please refer to the syllabus for a complete description of the collaboration policy.**

Collaborators: zh2yn, zz9ek

Sources: Chenghan (bless her soul)

PROBLEM 1 *Dominoes Strike Back! Picture Edition*

Rather than tile a board, we would like to use dominoes to tile a pixel-art image, such as Figure 1. Unfortunately, not every such image can be tiled with dominoes. Write an algorithm that takes as input a black-and-white pixel-art image (with m rows and n columns) and determines whether or not that image's black pixels can be exactly tiled using 2-by-1 dominoes. The running time of your algorithm should be a polynomial in n and m. Prove the correctness and running time of your algorithm. (*Hint: Think about max flow.*)

Figure 1: A pixel-art dinosaur. Can you tile the the black pixels using standard 2-by-1 dominoes?

Algorithm

Note: We can disregard all of the white pixels in this algorithm as we do not care about the tiling for these.

In this algorithm, we are taking in a black-and-white pixel-art image that consists of m rows and n columns as input, and determines whether or not that image's black pixels can be exactly tiled using 2-by-1 dominoes, as mentioned in the problem. We will first assign each pixel a node in the pixel-art image from the top to the bottom. Then, we will assign each node to either team Black or team Red by alternating (similar to a chessboard pattern). We will continue this until all nodes are assigned to team Black or team Red. Next, we need to create a bipartite graph using Ford-Fulkerson, similar to the TA dog graph shown in class by connecting the nodes in team Black to all adjacent nodes in team Red through the use of edges. We will then proceed to add a source node and a sink node to the two sets of nodes and connect the nodes in team Black to the source, and connect the nodes in team Red to the sink and make each edge capacity 1. It is to be noted that an adjacent node is a node that is above, below, to the right or to the left of the node. The next step is to compute a max flow graph. Then, we need to make sure that "the sum of the middle edges in the max flow result is equivalent to the number of black nodes divided by two in the pixel-art image." We need to do this to verify that black pixels are tilable by 2-by-1 dominoes, and if this is not true, we can conclude that the image is not tilable by 2-by-1 dominoes

Correctness

In this algorithm, each flow will represent a domino. The creation of the bipartite graph is significant in how it represents the matches of the tiling. Each node from the "black set is connected to a node from the red set, which is then matched to the correct tile." Since the white pixels are not being considered/tiled, they are disregarded. A distinct set of nodes are created for each set since each pixel is assigned to either the black set or red set, as mentioned before. Drawn out, we can visualize a set of nodes on the left and on the right. In doing this, we are able to create the bipartite graph. This graph connects the adjacent nodes from the black set to the red set through edges. As discussed in class, we can then apply the maximum bipartite matching using max flow. We will add a source node and connect all the nodes in the black set to it, and then add a sink node and connect all the nodes in the red set to it, Make the edge capacity 1.

This algorithm implemented can help us decipher whether an image can be tiled with dominoes. The significance behind dividing the result of the maximum flow by 2 is that it allows us to check to see if the number of dominoes matches. By solving the max flow problem, we are able to solve the max bipartite matching problem, which is basically like the domino tiling problem.

Runtime

When we compute the max flow of the bipartite graph through Ford-Fulkerson, it has a runtime of $O(V * E)$, where V and E represent the vertices and edges, respectively. Likewise, the bipartite graph has the runtime of $O(V * E)$. In regard to this problem, the runtime would be $4m^2 * n^2$, in which 4 describes the number of edges surrounding each node (4 way), and V can be $m * n$ while E can be up to $m * n$.
The overall runtime of this algorithm is then $O(m^2n^2)$.

PROBLEM 2 *Summer Camp*

You are on the activities committee for a summer camp and are trying to identify a suitable set of activities for the attendees. Your plan is to split the n attendees into $k \le n$ groups based on their interests and choose an activity for each of the groups. To help you plan, you have gathered a list of interests from each of the attendees. Based on the responses, you have defined the dissimilarity $d_{i,j} \ge 0$ to measure how different the i^{th} and j^{th} attendees are (e.g., a large $d_{i,j}$ indicates that the attendees have very different interests). Your goal is to assign each of the n attendees to a group $G_1, \ldots, G_k \subset \{1, \ldots, n\}$ in a way that *maximizes* the smallest dissimilarity between groups. We define the dissimilarity between two groups G_s and G_t to be

$$\text{dissimilarity}(G_s, G_t) = \min_{i \in G_s, j \in G_t} d_{i,j}.$$

Namely, the dissimilarity between two groups G_s, G_t is the smallest dissimilarity between two individuals in G_s and G_t. Give an algorithm that splits the n attendees into k groups $G_1, \ldots, G_k$ that maximizes

$$\max_{G_1,\ldots,G_t} \min_{s,t \in \{1,\ldots,k\}} \text{dissimilarity}(G_s, G_t) = \max_{G_1,\ldots,G_t} \min_{s,t \in \{1,\ldots,k\}} \min_{i \in G_s, j \in G_t} d_{i,j}.$$

Note that every group must contain at least 1 attendee. Your algorithm should run in time that is polynomial in n. Prove the correctness and running time of your algorithm. (*Hint: Try constructing a reduction to a problem we have seen before.*)

Algorithm

In this summer camp problem, we will reduce it to a minimum spanning tree problem. Our algorithm will split the n attendees into k groups, $G_1, \ldots, G_k$, that maximizes the given equations in the problem.

In our algorithm, we will first create a graph for the MST. A single node will represent each of the n attendees. Furthermore, edges will represent the dissimilarity. Additionally, the number given for dissimilarity will be the edge weights, where the smallest edge weight is given by the smallest dissimilarity between two groups. Using this graph, we can create a MST. Next, after creating the MST, we will cut the graph based on where the highest edge weight is. The purpose of this is to split the attendees into two groups. The latter step will then be repeated until k groups are created, where each cut will be the new highest edge weight. The total number of cuts should be $k - 1$ (the number of groups needed minus one).

Correctness

The correctness of our algorithm can be proved using the exchange argument. Using this, we will be able to prove why the heaviest weight must be removed each time. In the problem, the last edge removed, which is the $k - 1$ edge, is the minimum of all the max edges removed. After removing this edge, $k - 1$ cuts will have been made and we will gave a total of k groups. If we denote the last edge removed to d, the weight of d is going to be either equal to or more than ($d \geq$) any edge in any of the groups we have. Now, let's assume we have two random nodes (individuals), i and j, that put in the same cluster, or group, after the last cut ($k - 1$) is made. Taking into consideration our algorithm, this means that the distance/dissimilarity between i and j is less than or equal to the distance d (i and $j \leq d$) which was cut. In a different optimal algorithm, let's say i and j are in different clusters, or groups. Keeping in mind that the distance between i and j is less than or equal to d, because our algorithm removed the heaviest cut (d), it means that the new algorithm failed to maximize, or find the max heaviest cut d; instead, this weight is in one of the groups that have been split.

In the context of our problem, this new algorithm places the two individuals, i and j, into different groups, despite the fact that they were more alike than the two that were split up in our algorithm when the $k - 1$th cut happened. Thus, because our algorithm/solution removes the heaviest edge at the $k - 1$th cut, it is more optimal than this optimal algorithm .

Runtime

The overall runtime of this algorithm comes from creating of the minimum spanning tree and the removing of edges as each cut is made. Since the number of removed edges is less than the number of edges made in the creation of the MST, the removing of edges is outweighed by the creation of the tree.

If Kruskal's algorithm is used to implement/create the MST, our runtime would be $O(n^2 logn)$ in the worst case. This is due to the fact that Kruskal's algorith has a runtime of $O(ElogV)$. To clarify the comparison, the most amount of edges that there are going to be in the creation of the MST is n^2. In other words, there can be up to n^2 number of edges if every n node is connected to every other node. Furthermore, the vertices would be the number of attendees, n. Overall, this gives a runtime of $O(n^2 logn)$.

PROBLEM 3 *Backpacking, Revisited*

After your first successful backpacking adventure (HW6, Problem 2), you have decided to return to Shenandoah National Park. Similar to before, you and your friend have completed your packing list, and you need to bring n items in total, with the weights of the items given by $W = (w_1, \dots, w_n)$. Your goal this time is to divide the items between the two of you such that the difference in weight is as small as possible. There is no longer a restriction on the total number of items that each of you should carry. Here, we will define a decisional version of this Backpacking problem:

Backpacking: Given a sequence of non-negative weights $W = (w_1, \dots, w_n)$ and a target weight difference t, can you divide the items among you and your friend such that the weight difference between backpacks is at most t?

1. Show that the Backpacking problem defined above is NP-complete (namely, you should show that Backpacking $\in$ NP and that Backpacking is NP-hard). For this problem, you may use the fact that the SubsetSum problem is NP-complete:

   SubsetSum: Given a sequence of non-negative integers $a_1, \dots, a_n$ and a target value t, does there exist a subset $S \subseteq \{1, \dots, n\}$ such that $\sum_{i \in S} a_i = t$?

   In order to prove that the backpacking problem is NP-Complete, we have to show that the backpacking problem is within the bounds of NP AND that is NP-hard.

   First, we will show that the backpacking problem is NP-hard. This is done by reducing it from the SubsetSum problem (NP-hard as it is NP-complete). Lets assume we start with a set called S that consists of non-negative integers $a_1, \dots, a_n$, and its sum is equal to s, similar to the backpacking problem. Let's also assume that a subset B of S exists and its elements all sum up to our target value t. Hence, this would mean that the sum of all the elements outside of the subset B, which we can call B', has to be $s - t$. So no we know that we have two subsets B and B' whose sums are t and $s - t$, respectively. We are reducing the subset problem to our backpacking problem here.

   Note: In order to prevent confusion, we will relabel the t variable from the backpacking problem to t_p

   With these two subsets now, we can try to figure out if there is a way to guarantee that the difference of the sums of these two subsets is not greater than t_p from the backpacking problem (if not already). This would mean $\mid (s - t) - t \mid \leq t_p$. We can simplify this problem by setting $t_p = 0$, which is undoubtedly less than or equal to the original t_p from backpacking. This is because, in this case, you cannot have a negative difference (absolute value!). In order to confirm that the difference is zero, meaning both sums are the same, we can add more items to both sides, denoted as O_0 and O_1, giving us this equation $\rightarrow t + O_0 = s - t + O_1$. To do this, we can let the weights of one side be $O - 0 = 2s - t$, and the other side be $O_1 = s + t$. This way, we are in fact verifying that we do not mistakenly get negative weights.

   We then get the following:

$$\begin{aligned} t + O_0 &= s - t + O_1 \\ t + 2s - t &= s - t + s + t \\ 2s &= 2s \\ s &= s \end{aligned}$$

From doing the above, we have achieved our goal of a weight difference of zero between the two sides. The backpacking problem is NP as the main operations carried out in the backpacking problem we are iterating through all the items in the list of weights, which constitutes a linear time, a subset of polynomial time, and putting them accordingly into the two different subsets. Additionally, as discussed in the beginning, we need to calculate/check the difference between the two subsets, which is constant time. Furthermore, we exhibited that backpacking is NP-hard (shown from reducing it from our subset problem), as shown above. Therefore, we are able to prove that backpacking is in fact NP-complete.

Overall, we are able to find the difference of the problem in linear time, and hence, it occurs in polynomial time.

2. Your solution to HW6, Problem 2 can be adapted to solve this version of the BACKPACKING problem in time that is *polynomial* in n and M, where $M = \max(w_1, \dots, w_n)$ is the maximum weight of all of the items. Why did this not prove $\mathsf{P} = \mathsf{NP}$? (*Conversely, if you did prove that* $\mathsf{P} = \mathsf{NP}$*, there's a nice check waiting for you at the Clay Mathematics Institute.*)

In this problem, the run time of the verifier is $O(nlogm)$, where n denotes number of elements and M denotes max weight that is being read in from memory (in bits). This takes $logm$ time. Then, the runtime of the backpacking problem/the solver is $O(n^3 m)$, with n denoting the number of items and m denoting maximum weight.
In order to present that this equation is polynomial (in terms of the verifier), we would have to find a polynomial p in the equation $(logm)^p < M$ that would allow equality on both sides. We don't consider the comparison of n and n^3 as that meet the requirement of being polynomial. Contrastingly, there exists no polynomial p that would make them equal in the above case. This would work only in the case that p was as well an exponential function, which does not align with the requirements mentioned in the problem. Additionally, considering the the worst case, we can set $m = log_{10}(M)$. Let's say the total weight of all the items in the set is nm, the maximum possible weight. In order to solve this, we would take the inverse of the log, which gives us: $10^m(n^3)$. Hence, it cannot be NP as the latter result is not polynomial. Therefore, we have not proved that $P = NP$.