Elijah Chou

CS 526

Dr. Michelangelo Grigni

9/22/2023

**HW 2**

1. A) To argue that a tree decomposition with width k has at most |V| nodes, consider the following. Suppose we pick any arbitrary root node in T. For each vertex in V, while there may be multiple bags containing v, all these bags will form a subtree of T. This ultimately implies that there must be one bag Xi that contains v that is the closest to root r. If the bag isn’t the root, this means that Xi has a parent Xp that does not contain v. Considering this, every bag Xj will be a “closest to root r” bag for at least one vertex in Xj. We can show this by arguing that there cannot be two bags containing v that are “equidistant” from root r. If two bags that contain v are the same distance away from r, this means that they should have a common parent bag moving towards the root. This would mean that they are adjacent bags and based on the definition of the tree decomposition in the question, we should have merged all adjacent bags already. Since we can essentially map every vertex to one “closest to r” bag, the number of nodes/bags in T is at most |V|.  
     
   B) First, we will show that A∩B = Xi ∩ Xj. We know that A contains Xi and that B contains Xj. In the other direction, suppose that v is in both A and B. Then v is in Xa where a is a node in TA, and v is also in Xb where b is a node in TB. Utilizing the third property of the definition of a tree decomposition, we can deduce that v appears in every bag along the path in T between a and b. Therefore, v is in both Xi and Xj, so v is in Xi ∩ Xj.   
     
   Following this logic, we can move on to prove that if x is in A, and y is in B, and edge {x, y} is an edge of G, then at least one of x or y is in S = Xi∩Xj. According to the second property of a tree decomposition, for every edge {v, w} in the graph, there must be a bag Xi that contains both v and w. This also means that there must have been a bag Xz that contained both x and y if there is an edge {x, y} in G. Utilizing the third property of a tree decomposition, we know that, if x is in bag Xi, there must be a unique path between Xi and Xz because they both contain vertex x. Using the same logic, there should also be a unique path between Xj and Xz because they both contain y. This means that the bag Xz is part of a path connecting Xi and Xj. If Xz is indeed in S because it is not in A – S or B – S, then we have shown that Xz, which includes both x and y, are in S, and therefore the initial statement of this paragraph is true. In the case where Xz is not necessarily in S, we can show that at least x or y is in S. If x is in A – S, or in other words only in A and not in S, y must be in S because there must be a path from x to y, which is in B, since there is a edge {x, y} in G. Because there is a path that exists between x and y, the path must have a “first step” into B. This indicates that y exists in S since a first step from A to B must be an intersection between A and B and therefore S.  
     
   Since we showed that at least one of x or y is in S = Xi∩Xj, this also shows that S separates A and B. This is because we previously argued that S = A∩B, which implies that G has no edge between A – S and B – S. If u is in S, then we are done! Otherwise, u is in A – S, v is in B, and G has a path from u to v. Starting from u, the path must have a first step into B. With the claim proven in the previous paragraph, y must be in S.  
     
   C) To modify the tree decomposition (X, T) to make it smooth, follow these steps: For each node i in T: If |Xi| > k+1, then split Xi into two bags, with one containing k + 1 vertices. Then for all “small” bags, find a neighbor that has k+1 vertices and copy vertices from the larger neighbor into the small bag. This ensures that |Xi| = k+1 for all nodes i.   
     
   For each tree edge {i, j} in T: Compute the intersection of bags Xi and Xj, denoted as S = Xi ∩ Xj. In the case where |S| < k, we can try adding a new bag, Xz, in between Xi and Xj that has vertices from both Xi and Xj such that |Xi ∩ Xz| = k and |Xz ∩ Xj| = k. By doing this, we can ensure that all edges {i, j} in the new modified tree decomposition has |Xi ∩ Xj| = k.

By performing these two steps, you will create a modified tree decomposition (X, T) where each node has exactly k+1 vertices in its bag, and each tree edge {i, j} has exactly k vertices in their intersection, making the tree decomposition smooth according to the given definitions.

1. A) To compute B(s, i, j), we want to check whether the partial coloring s on Xi∩Xj can be extended to a 3-coloring of Di (the subtree rooted at node i). We can use the following boolean formula: B(s, i, j) = A(s', i) ∧ (s agrees with s' on Xi∩Xj). Here, s' represents a 3-coloring on Xi, and we are checking if there exists an extension of s' to Di such that it agrees with s on Xi∩Xj. We're essentially checking if we can extend the colorings from the children of node i to node j while satisfying the color restrictions on Xi∩Xj.

In regards to the range of choices for s’, there are 3 main cases:

i) when there is a vertex v in parent Xj but not in child Xi: this case will be true if all neighbors u of v don’t have the same coloring as v.

ii) when there is a vertex v in child Xi but not in parent Xj: this case will be true if one of 3 extensions to child Xi is true.

iii) when 2 children, Xi1 and Xi2, have the same vertices as parent Xj: this case will be true if the coloring does indeed extend to both Xi1 and Xi2.

B) To compute A(s, i), we want to check whether the partial coloring s on Xi can be extended to a 3-coloring of Di. We can use the following boolean formula: A(s, i) = (∀j: children of i) [B(s’, j, i)]. This formula checks if, for all children j of node i, we can extend the partial coloring s on Xi to Di by considering the extensions obtained from each child j. If any of the children fails to extend the coloring, then A(s, i) will be false.   
  
If Xi has no children, A(s, i) should simply return true if there is indeed a 3-coloring within the vertices of Xi and return false if there is not.  
  
C) To estimate the time needed to decide whether G is 3-colorable, we can use dynamic programming over the tree decomposition (X, T) of width k. Let n be the number of vertices in G. In each subproblem A(s, i) or B(s, i, j), s represents a 3-coloring on a subset of vertices, and we need to consider all possible extensions. The total number of subproblems is O(3^k \* n) because for each vertex in a bag, there are 3 choices for its color (1, 2, or 3), and there are at most k+1 vertices in each bag. We can compute each subproblem in constant time by checking if the partial coloring satisfies the 3-coloring constraints. Therefore, the overall time complexity to decide whether G is 3-colorable is O(3^k \* n). When k is a constant, this simplifies to O(n), making it linear in the size of the input graph. This is similar to the MIS dynamic programming approach but adapted for 3-coloring.

1. A) An example of a suboptimal sequence of cards is the following: 1, 3, 100, 2. The greedy choice would have chosen 2 at the start, but this would give the other player to choose 100 later.  
     
   B) The algorithm will store the computed cases in a n by n matrix T, with each entry in the matrix storing 3 different variables. Each will store 1) the optimal choice (stored as “choice”), 2) the value the player gets from making said choice (stored as “value”), and 3) the next value that the opposite player will get (stored as “next”). To initialize the beginning values, we will use the following notation: T[i][i] -> value = si (or card i), choice = i, and next = 0. We then use recursion to do the following.  
     
   T[i][j].choice = i if si + T[i+1][j].next > sj + T[i][j-1].next OR = j if si + T[i+1][j].next <= sj + T[i][j-1].next  
     
   T[i][j].value = si + T[i+1, j].next if T[i][j].choice == i OR = sj + T[i][j-1].next if T[i][j].choice == j

T[i][j].next = T[i+1][j].value if T[i][j].choice == i OR = T[i][j-1].value if T[i][j].choice == j

1. A) Since the new weight is greater than the old one, there is no way to improve the MST just by adding this edge. There shouldn’t be a linear time algorithm to update the MST because edge e is still not in the MST.  
     
   B) With the reduced weight, edge e could replace some e’ from the same cycle in the original graph G. In order to find this cycle, something that an algorithm should be able to do is add a new edge e to tree T, run breadth first search in linear time to find a cycle containing edge e. Proceed by removing any edge with greater weight than e from the found cycle, but if no such edges were found, simply remove edge e.  
     
   C) Since edge e was in the MST before, reducing the weight of e would keep it inside the MST since it actually improves the MST. Therefore, an update would not be needed for this case.  
     
   D) Run breadth first search to get all the cycles that edge e belongs to. When doing so, make note of the heaviest and lightest edges on the current path. If edge e is the heaviest edge in a cycle, then remove it from T, the MST, and replace it with the lightest edge of cycle from running breadth first search earlier.
2. One way to solve this is to first set all the red edges to have weights of 1 and blue edges to have weights of 0. We will first find a minimum red possible tree T1 by running Kruskal’s algorithm so that it finds an MST in O(E log E) time. Then, we get a maximum red possible tree T2 by setting all the red edges to have weights of 0 and blue edges to have weights of 1 and then running Kruskal’s algorithm again to find another MST. If either T1 or T2 has k red edges then we are done! Or, if k < red edges in T1 or k > red edges in T2, we return false because it is impossible to get an MST with exactly k red edges. However, if the number of red in T1 < k < number of red in T2, then we need to a few more steps.  
     
   Once we have T1 and T2, we can use the fact that MSTs are bases. Given any edge x in T1-T2, we can find an edge y in T2-T1, so that T’ = T1 – {x} + {y} is still a spanning tree. Every time we do this step, this should bring us one step closer to the MST containing exactly k red edges. To do this, we would first need to find some edge x in T1 – T2, which could be done by labeling each edge beforehand. Once we find that edge, we can go ahead and remove it, which will create two separate components C1 and C2. Label each vertex in both C1 and C2 with their corresponding components for the next step. Now that we have removed x, we should now find an edge y in T2 – T1 such that y connects C1 and C2 again. Once we find this, we can add y to T’ and have an MST. This replacement overall would be linear time O(|V|) because in the worst case you check every edge, and there are at most V-1 edges in an MST.   
     
   In regards to time complexity, there are the two calls of Kruskal’s algorithm, so we have O(E log E) time. But then there are at most V calls of the linear time algorithm described in the previous paragraph. This brings the time complexity up to O(|V|^2) time. So overall the complexity would be O(|V|^2 + E log E) = O(|V|^2 + V log V) = O(|V|^2)