CS1200: Intro. to Algorithms and their Limitations Anshu & Vadhan

Problem Set 6

Harvard SEAS - Fall 2024 *Due: Wed Oct. 30, 2024 (11:59pm)*

Your name: Ethan Veghte
Collaborators:
No. of late days used on previous psets: 7
No. of late days used after including this pset: 8

The purpose of this problem set is practice proving optimality and efficiency of a greedy algorithm, practice modelling problems using graphs, reinforce understanding of the matching algorithm covered we learned, and think about ethical issues raised when modelling real-world problems for algorithmic solution.

1. (Greedy Coloring for Interval Scheduling) The IntervalScheduling-Optimization problem we studied in class finds the largest group of nonintersecting intervals. In many applications, it is also natural to consider the *coloring* version of the problem, where we want to partition the input intervals into as few groups as possible so that each group is nonintersecting.

   In this problem, you will prove that Greedy Coloring in order of *increasing start time* gives optimal coloring for interval scheduling. (Note the contrast with the *increasing finish time* ordering we used for the version studied in class. It is a common phenomenon that different orderings are better for coloring vs. independent-set problems; for example decreasing vertex degree is a good heuristic for greedy coloring of general graphs, while increasing vertex degree is a good heuristic for independent set.) Let $x = (x_0, \ldots, x_{n-1})$ be an instance of IntervalScheduling, where each x_i is an interval $[a_i, b_i]$ with $a_i, b_i \in \mathbb{Q}$. Let k be the maximum number of input intervals that contain any value $t \in \mathbb{Q}$. That is,

$$k = \max_{t \in \mathbb{Q}} |\{i \in [n] : t \in x_i\}|.$$

   (a) Prove that every proper coloring for IntervalScheduling uses at least k colors.

   The easiest intuition for this proof is that if k is is the maximum number of edges out of a single node (i.e. interval conflicts) there is no way to have a proper coloring without using all k colors to color all the connecting nodes or conflicting intervals. A proper coloring is one where all overlapping intervals have a unique color, and by definition there is at most k conflicts at any given interval. However lets prove this by contradiction:
   Assumption: *Assume there is a way to do a proper coloring using $c < k$ colors.*
   Now by definition of k there must be a time t where there are k interval scheduling conflicts. Thus k distinct colors are necessary to color all of those conflicts properly.
   Contradiction: *We assumed there was a way to color the schedule in c colors however, we showed that there will be a point t where k colors are necessary thus it is not possible for $c < k$.*
   Conclusion: *By contradiction that there is no way to have c colors where $c < k$ in the IntervalScheduling problem, and properly color the graph, we prove that every proper coloring uses at least k colors.*

(b) Show that the Greedy Coloring in order of *increasing start time* uses at most k colors. (To develop your intuition, carry out the algorithm on a few examples.)

Greedy Coloring works by putting a series of 'nodes' into a specified order and working down the order, and at each node coloring it the lowest color that is not being used by the neighboring nodes. The algorithm is finished once it makes it through the list as it will have colored all the nodes. In this scenario, if you order the intervals based on start time, you will use at most k colors. To use an example:
$I_1 = (1, 4)$
$I_2 = (3, 7)$
$I_3 = (2, 4)$
$I_4 = (8, 11)$
Once ordered you get $[I_1, I_3, I_2, I_4]$. *For greedy coloring then you would color* $I_1 \to C_1$, *because* C_1 *would be lowest possible color,* $I_2 \to C_2$ *because* I_1 *is conflicting thus lowest possible color is* C_2. *Next you get* $I_3 \to C_3$ *because it conflicts with* I_1 *and* I_2 *thus* C_3 *is lowest color. Finally* $I_4 \to C_1$ *because it conflicts with nothing and* C_1 *is lowest available color. Here* $k = 3$ *because* I_1, I_2, I_3 *all conflict thus we needed 3 distinct colors in order to properly color these intervals.*
By ordering the intervals based on start time, every interval will be next to its conflicting interval if there is one. This is because you cannot have a conflict be separated by another interval if that interval is also not a conflict. So as you work down the ordering, you can color accordingly to the conflicts that you run into. Since there will be at most k conflicts then you will only ever need to use k colors.

(c) Show that the Greedy Coloring in order of increasing start time can be implemented in time $O(n \log n)$. Hints:

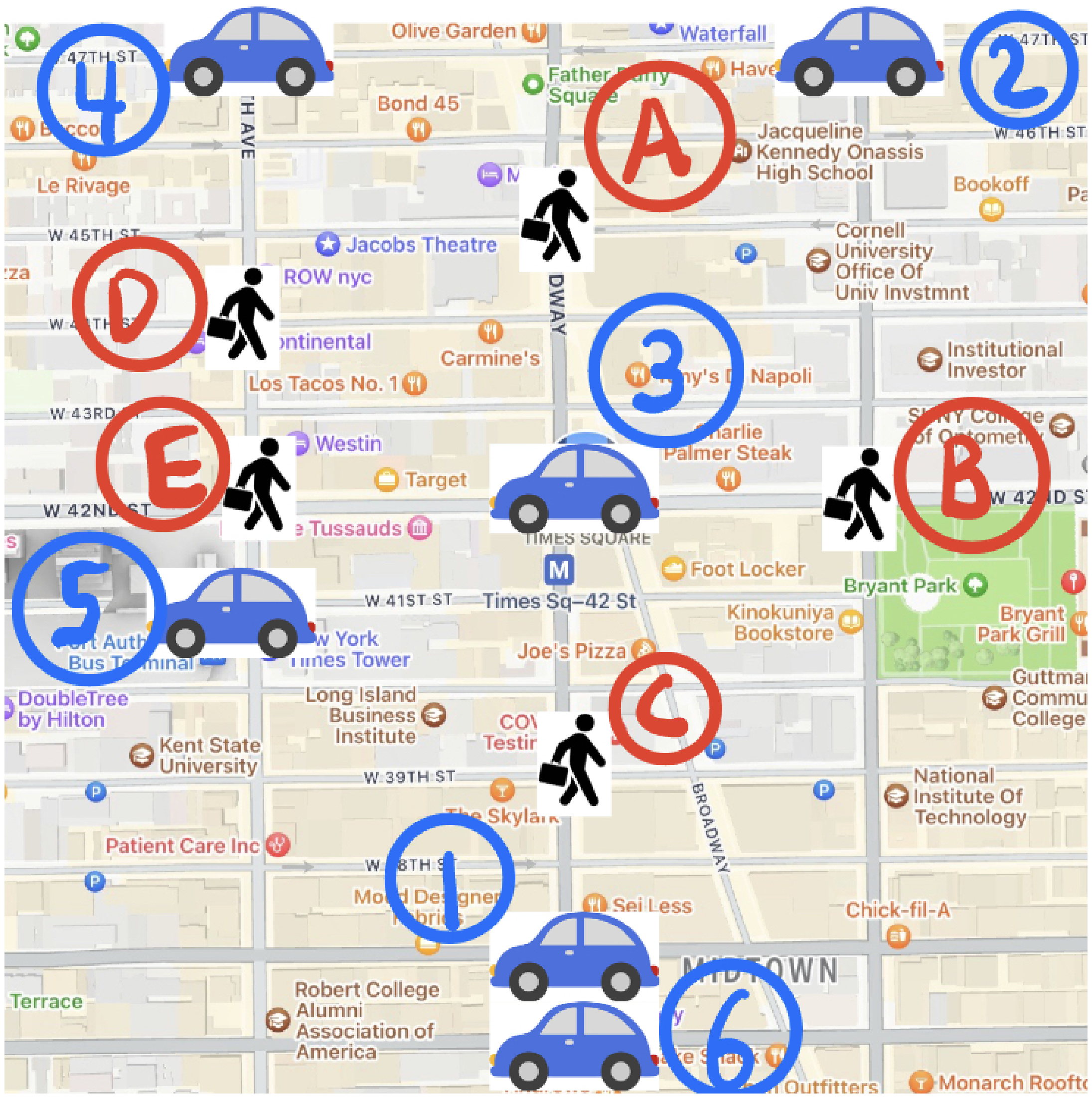
i. Keep track of the end times of the most recently scheduled intervals assigned to each color, and use an appropriate data structure to ensure that you spend only $O(\log k)$ rather than $O(k)$ time per iteration, where k is the number of colors used.

ii. To make life easier for yourselves, you may instead implement a *variant* of Greedy Coloring in which, at every step, you assign a vertex *any color* not assigned to its neighbours that's also less than the largest color (as opposed to standard Greedy Coloring in which you assign the smallest color).

First the set of intervals must be ordered based on start times. This can be done in $O(n \log n)$. *This will generally end up being the asymptotically dominant operation, but we are not done with the algorithm.*
You also need a data structure such that you can easily compare the interval start time that you are looking at with the end times of other intervals. This is perfect for a balanced BST. To do this, each time you go through a node you add its end time to the tree. You also remove any items from the tree that are less than the start time you are at. This is because any subsequent start time you are coloring will have a higher start time, so there is no way to conflict with an interval that has already ended. That is, the only intervals in the tree are active and still relevant ones to the current start time. Since the BST is at most k elements large (by definition of k no interval will have more than k 'active' intervals whose end times

need to be tracked), and the height of the tree is $O(\log k)$, insertion and extraction from the tree can be done in $O(\log k)$. Insertion and extraction are the relevant operations for this algorithm. Furthermore, it makes the highest and lowest elements easily extractable (only lowest is relevant in this case, because it is most likely to lose relevancy first). Worst case $k = n$ so an operation would take $O(\log n)$ for each interval, and since each interval only needs to be operated on once it is still $O(n \log n)$.
To do the coloring, the color value should be held as the value in the key-value pairs of the tree. That way you maintain a record of what colors have been used in the current conflicts. Instead of maintaining what is the lowest color not being used, you can instead just keep track of highest color used so far, and those not being used by a certain interval conflict. That way to choose a color you simply choose any of the ones not being used and less than highest color randomly. If all are used up until highest color then a new distinct color is needed. These are simple $O(1)$ operations that will not affect asymptotic runtime.
In the end you are left with sorting ($O(n \log n)$), then comparing, which is worst-case $O(n \log n)$ as mentioned, still not exceeding $O(n \log n)$. As such, the insertion, extraction, and coloring of the intervals in the BST are $O(\log k)$ operations for each node, for all n intervals it is $O(n \log k)$ which is worst case $O(n \log n)$ if $n = k$. Thus no operations exceed $O(n \log k)$, so this process in the end takes $O(n \log n)$ time.

2. (Matching Algorithms) One practical application of matching algorithms is planning logistics, like in the following example from (fictional) ridesharing service Lyber in (real) New York City's Times Square. When a customer books a Lyber ride, the ride request is sent to a Lyber server and combined with others to create a schematic like the one drawn in the map below:

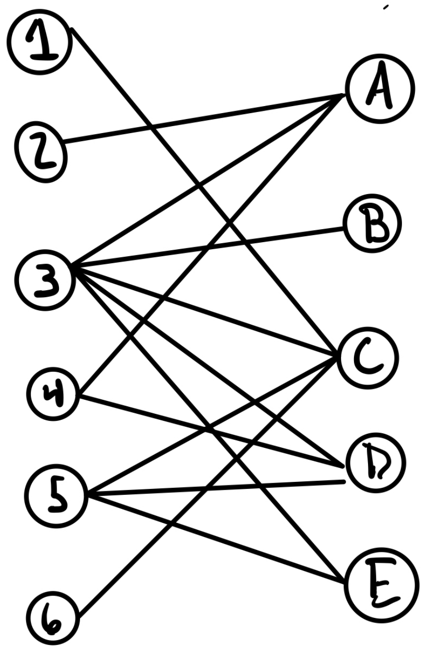


Given a schematic like this, Lyber's goal is to serve as many customers (labeled A–E in the map) as possible, by assigning each one to a driver (labeled 1–6 in the map). For simplicity, each customer and driver is at an intersection, and assume driving between adjacent streets (vertical segment) takes 30 seconds, and driving between adjacent avenues (horizontal segments) takes 1 minute. However, the one twist is that they want to make sure that *no customer is waiting for longer than 2 minutes*. They also do not want to assign a driver to more than one customer at once, since serving a single customer can take more than 2 minutes.

(a) To perform the assignment, they reduce to Maximum Matching in bipartite graphs. Draw a bipartite graph corresponding to the drivers and customers in the map above.

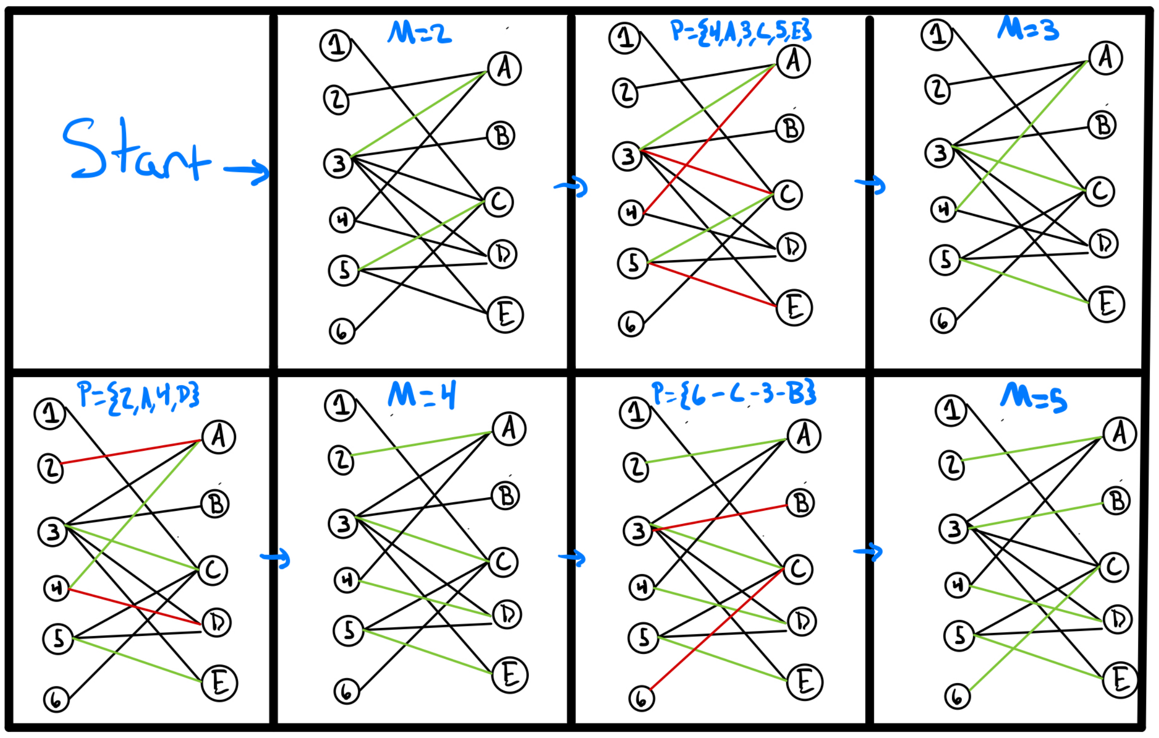
The limiting factor that creates the edges in the bipartite graph in this scenario is time.

It is a gridlock city and so every person is accessible by every car, but the car has to be able to reach the person in 2 minutes to be considered an edge.



Edges:
Person A: (A, 2), (A, 3), (A, 4)
Person B: (B, 3)
Person C: (C, 1), (C, 3), (C,5), (C,6)
Person D: (D, 3), (D, 4), (D, 5)
Person E: (E, 3), (E, 5)

(b) The Lyber app first prioritizes customers on Broadway, so they initially assign customer A to driver 3 and customer C to driver 5. Using the algorithm from class, find a *maximum matching* in the bipartite matching graph you've drawn, starting from the initial matching of A to 3 and C to 5. Draw pictures showing the sequence of matchings and augmenting paths you find. (No need to break down the steps of the algorithm to find the augmenting paths.)



3. (Vertex-Weighted Matching) For a graph $G = (V, E)$ and a subset $F \subseteq E$, let $V(F)$ denote the set $\bigcup_{f \in F} f$ of vertices that are an endpoint of at least one edge in F.

   (a) Prove that if $G = (V, E)$ is a graph and $M \subseteq E$ is a matching in G, then there is a maximum-size matching M' such that $V(M) \subseteq V(M')$. (Hint: consider constructing a maximum matching via augmenting paths, but starting with $M_0 = M$ rather than $M_0 = \emptyset$. What can you say about the $V(M_i)$'s?)
   The basic logic to this proof stems out of the fact that for any matching M, M_i is the i-th iteration of augmenting paths in order to increase number of matches. Thus M_i comes out of augmenting M_{i-1}. And since an augmented path only ever uses vertices that are included in the matching then $V(M_{i-1}) \subseteq V(M_i)$. In order to flesh out this proof we will use induction.
   Base Case: $M = M_0$
   This is trivially proved to be true because by definition of M_0 and M, $V(M) \subseteq V(M_0)$.
   Inductive hypothesis: *Assume that for M_i where $i > 0$, $V(M) \subseteq V(M_i)$.*
   Inductive step:
   If M_i is a maximum matching then $M' = M_i$ and then $V(M) \subseteq V(M_i) = V(M')$ simply by the inductive hypothesis.
   If M_i is not a maximum matching then we can continue creating augmenting paths that gets M_i closer to M'. For each augmentation, the size of M_i only increases by 1. We also know that by definition of augmenting paths, the subset of vertices already defined in $V(M_i)$ is still contained in $V(M_{i+1})$. This is because in augmenting only edges are swapped our the vertices are maintained and added to. So essentially $V(M_i) \subseteq V(M_{i+1})$. The Inductive hypothesis also says that $V(M) \subseteq V(M_i)$ thus $V(M) \subseteq V(M_{i+1})$.

There will be a point where the jth iteration after i reaches the maximum matching. In other words after some number of augmentations $V(M_{i+j}) = V(M_k) = V(M')$ where k is the number of total augmentation iterations to reach M'. At that point since we know $i \leq i + j$, and $V(M_i) \subseteq V(M_{i+j}) = V(M_k) = V(M')$ and $V(M) \subseteq V(M_i)$ by inductive hypothesis then, after k iterations we will reach M' where $V(M) \subseteq V(M')$
Conclusion: *Thus by the induction we have proved that $V(M) \subseteq V(M')$.*

(b) In the Embedded EthiCS module, we saw how simply maximizing the *size* of a matching may not always be the right objective. Thus, it is natural to consider weighted versions of the matching problem. Suppose we consider vertex-weighted graphs $G = (V, E, w)$, w is an array specifying a nonnegative vertex weight $w(v)$ for every $v \in V$. (For example, the weight assigned to a patient might correspond to the number of extra years of life they would gain from a donation.) The goal of the *vertex-weighted maximum matching problem* is to find a matching M maximizing its *total weight*

$$w(M) = \sum_{\{u,v\} \in M} (w(u) + w(v)).$$

(This corresponds to the utilitarian objective discussed in Embedded EthiCS module.) Using Part 3a, prove that every graph G has a matching M^* that simultaneously maximizes both total weight and size. That is, for every matching M in G, we have both $w(M) \leq w(M^*)$ and $|M| \leq |M^*|$.

This still leaves the question of whether there efficient algorithms to optimize vertex-weighted matching. This problem can be reduced to the maximum-flow problem, which is covered in CS1240.

The basic intution to this problem is that from 3a we learned that $V(M) \subseteq V(M')$. From this we can understand that given M_i, any augmented path will increase the number of vertices included in the matching. It will also interestingly, increase the total weight because with weights on vertices, more vertices $\Rightarrow$ more weight. Thus we know $w(M_i) \leq w(M_{i+1})$ and $|M_i| \leq |M^{i+1}|$.
We can imagine then that there will be a matching M^* which maximizes size and weight simultaneously. This is because if lets say j-th iteration of M gets you to M^* (i.e. $M_j = M^*$) then we know $|M_{j-1}| \leq |M_j|$ and with nonnegative weights on vertices $w(M_{j-1}) \leq w(M_j)$. So if there is an opportunity to increase size then it will also increase weight (more vertices $\Rightarrow$ more weight). Also if there is a different matching to increase weight then unless it is a maximum matching there will be a way to further increase weight (augment path further). This is because the starting point matching does not affect final result (i.e. any valid initial matching will reach to the same maximum size). There may be different maximum matches, but some might maximize size and weight simultaneously.

(c) (optional[1]) Show how to reduce matching with the *maximin* objective to vertex-weighted

[1]This problem won't make a difference between N, L, R-, and R grades. As this problem is purely extra credit, course staff will deprioritize questions about this problem at office hours and on Ed.

matching,[2] and deduce that there is always a matching M that simultaneously maximizes the maximin objective and $|M|$. For simplicity, you may assume that there are no ties in how well off the patients are prior to treatment. (Hint: use weights that are powers of 2.)

4. (EthiCS Reflection) Suppose there are two patients in need of an immediate kidney transplant, but only one donor is currently available. The donor's kidney is compatible with both patients. Patient A starts at 30 QALYs and is expected to live 3 additional QALYs as a result of the transplant. Patient B starts at 45 QALYs, and is expected to live 10 additional QALYs as a result of the transplant. *All else being equal*, **which patient should the kidney go to, and why?** Your response should take the form of a short paragraph (3-4 sentence) reflection. In explaining your ethical reasoning about the case, be sure to draw on at least one concept discussed in class.

   Note: As with the previous psets, you may include your answer in your PDF submission, but the answer should ultimately go into a separate Gradescope submission form.

   I believe the kidney should go to patient B. Although it is obviously an extremely difficult decision, the basis of my decision is making best use of the kidney. The kidney here is the scarce resource, and its function is simple, keeping people alive. Thus to use a kidney for something other than one that maximizes keeping people alive (largest additional QALYs boost), is to waste the scarce resource. This is generally a utilitarian perspective that maximizes according to the 'benefit' to society as QALYs gained from the transplant. Even if I did want to factor in a maximin notion, I don't believe the difference in QALYs already lived by A and B is sufficient to offset the differences gained by the kidney. You could look at by saying patient A gets an additional 10% *of the life they have lived* ($\frac{3}{30}$)*, whereas patient B gets a boost of* 22% ($\frac{10}{45}$).

5. Once you're done with this problem set, please fill out this survey so that we can gather students' thoughts on the problem set, and the class in general. It's not required, but we really appreciate all responses!

[2]In lecture, Salil said that he did not know whether there were efficient algorithms to optimize the maximin objective, but afterwards we realized that this reduction allows it to be solved via maximum flow algorithms, as covered in CS1240.