

Problem Set 5: NP-Completeness and Approximations

Due: Friday 12/10 11:59PM

Feel free to work in groups of at most 4 for these - if you have a group of more than 4, please run it by me first. If you do work in a group, please include the names of those that you worked with. **However: each student should submit a separate copy where the solutions have been written by yourself.**

Our first goal is to show the following problem is NP-COMPLETE for $k \geq 3$:

k -COLOR

INPUT: A graph G (you can assume it's just one connected component) OUTPUT: TRUE if you can color the graph in k colors so that no two adjacent vertices are colored the same.

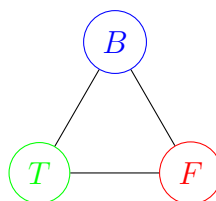
1. (5 points) Show that 2-COLOR is in P.

Solution: Consider a valid coloring for a 2-colorable graph: note that we could swap the color on every vertex and get a new valid coloring. As such, take an arbitrary starting vertex and color it, then proceed via BFS over the vertices. Since we are proceeding by BFS, whenever we reach a new vertex it must be neighbored with a vertex that is already colored, at which point we must color it a different color. If at any point we color a vertex the same color as any of its neighbors, we can immediately announce that the graph is not 2-colorable. If we are otherwise able to color the entire graph, then we have colored the graph such that no vertex is the same color as any neighbor of it, and hence the graph is 2-colorable. Since BFS can be implemented in polynomial time, this suffices to show that 2-COLOR is in P.

2. (10 points) Show that 3-COLOR is NP-COMPLETE.

Some helpful thoughts below: suppose we have an instance for 3SAT with variables $x_1, \dots, x_n$ and clauses $C_1, \dots, C_m$:

- i) Consider the following graph:



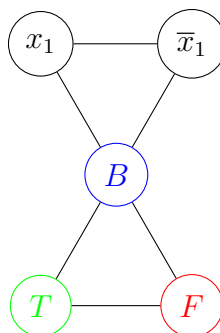
Convince yourself that these three vertices must be colored 3 different colors in a valid coloring.

Solution: If any of them were the same color, then we would have two adjacent vertices with the same color.

- ii) Consider the following subgraph:

Problem Set 5: NP-Completeness and Approximations

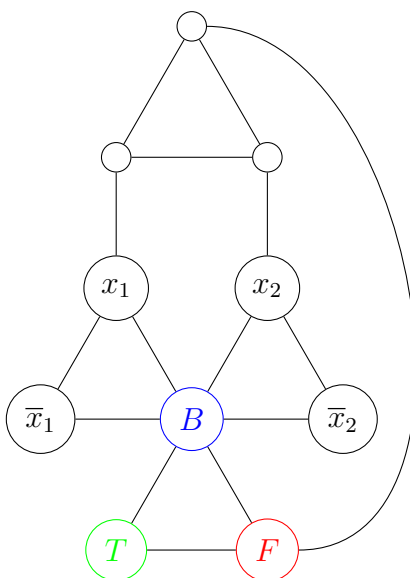
Due: Friday 12/10 11:59PM



Convince yourself that one of x_1 or $\bar{x}_1$ is colored the same color as T and the other must be colored the same as F .

Solution: As above, x_1 and x_2 can't be the same color, and neither can be the same color as B . Thus one must be the same color as T and one must be the same color as F .

iii) Consider the following subgraph:



Convince yourself that there's no valid coloring for this graph where both x_1 and x_2 are the same color as the F vertex.

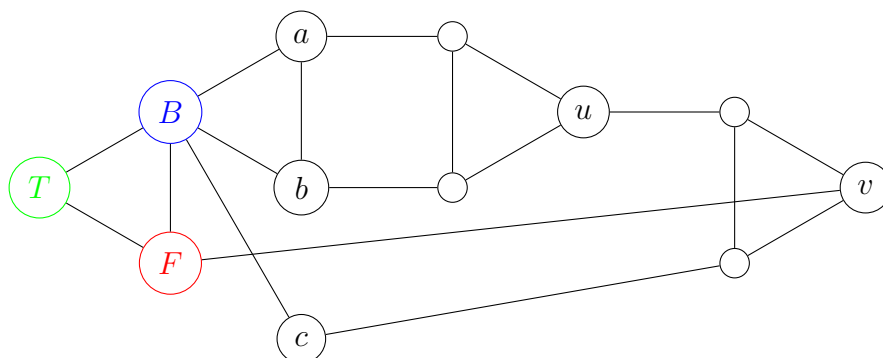
Solution: If both x_1 and x_2 were the same color as F , then one of the two vertices above them would have to be the same color as B and the other the same color as T , and thus the top vertex would have to be colored the same as F , but this would immediately be a problem since there is an edge between F and the top vertex. On the other hand, if at least one of x_1 or x_2 is the same color as T then the graph is 3 colorable.

First, to see that 3-COLOR is in NP, note that we can verify a solution by asking what color each vertex should be, then looping through the edges and making sure no two endpoints of any edge have the same color. So we can verify a solution in a polynomial amount of time.

Problem Set 5: NP-Completeness and Approximations

Due: Friday 12/10 11:59PM

Next, we will show that 3-COLOR is NP-HARD by means of a reduction from 3SAT. Given an input of 3SAT with variables $x_1, \dots, x_n$ and clauses $C_1, \dots, C_m$, we first create 3 vertices as in i) above. Then for every variable x_i we create adjacent vertices x_i and $\bar{x}_i$ and connect them both to the vertex B , as in ii. Then, for each clause of the form (a, b, c) we connect the corresponding vertices with the gadget as drawn below:



Note that by iii) it must be the case that if this component can be colored then v must not be the same color as F , which can only happen if at least one of u or c is not the same color as F . Note that (again by iii) if u is not the same color as F it must be the case that at least one of a or b is not the same color as F . Hence each gadget is colorable so long as one of a , b , or c is not colored the same as F .

Suppose we have a satisfying assignment to the variables $x_1, \dots, x_n$ so that at least one literal in every clause is True. In the graph we've created, give B , T , and F each their own color, and then assign each literal a with the same color as T if a has been assigned True or F if it has been assigned False. Then for every gadget at least one the literal nodes has the same color as T , so all of the gadgets are colorable. Hence the whole graph is colored so that no two endpoints of an edge have the same color.

Suppose we have a valid coloring using 3 colors on the graph we've created. We know that each gadget is colorable only if at least one literal node in each clause has the same color as T . Assign variables to be True if they have the same color as T in the graph, and otherwise False: under this assignment, there is at least one true literal in every clause so the whole formula evaluates to true, giving us a satisfying assignment.

The next problem is based on https://en.wikipedia.org/wiki/Wolf,_goat_and_cabbage_problem.

3. (10 points) Imagine we have a collection of n items to transport across a river and a boat that can hold ourselves plus k items, under the constraint that certain items cannot be left on the same side of the river as each other. Show that determining whether we are able to get all the items across the river is NP-COMplete.

Hint: You're free to reduce from whatever you want, but I think that VERTEX COVER might be the most straightforward?

Solution: First I claim this problem is in NP: for a solution I'm going to ask what gets taken across on every boat ride. To verify that solution, I check to make sure that

Problem Set 5: NP-Completeness and Approximations

Due: Friday 12/10 11:59PM

what gets left on either side of the river is all compatible: this is $O(n^2)$. The number of rides I take should be $O(n)$, so in total this is $O(n^3)$ verification.

Next we show $\text{VERTEXCOVER} \leq_p \text{RIVERCROSSING}$, showing that RIVERCROSSING is NP-HARD. Given our graph G and parameter k for VERTEXCOVER , we create an instance for RIVERCROSSING as follows: Each vertex becomes an item, where the edges denote incompatibility. We also create an extra pair of items a, b that are incompatible with each other. Lastly, we set the size of the boat to be $k + 1$.

Note that if there is a vertex cover of size k , then by removing that vertex cover from the graph there are no edges remaining. Hence, if there is a vertex cover of size k , we can solve the RIVERCROSSING instance by taking the vertex cover and a , leaving a on the other side, then sequentially taking 1 item across from the rest of the items left on the first side other than b , until only b remains on the other side, at which point we can take b and the vertex cover and achieve victory.

Now, suppose there is a way to solve the river crossing problem. Then on the very first trip that is taken, at least one of a or b must be taken, since they are incompatible to leave on the same side. Note that whatever remains of the items on the island, there cannot be any incompatibilities. This means that after the $\leq k$ items were put on the boat, all remaining items were compatible, ie all items that came from the original graph did not have any edges between each other. Which means that the $\leq k$ items that were put onto the boat covered all the edges in the graph, ie we have a vertex cover of size $\leq k$ (if it is $< k$, by adding any vertex we get a vertex cover of size k).

On the other hand, if there is no vertex cover of size k , then

The rest of the problem set is based around investigating the online k -cache problem. For this problem, imagine we are writing an algorithm overseeing a memory cache in a computer which our client is interacting with: sequentially the client will request specific “pages” of memory $r_1, r_2, \dots$. If the requested page is not in our memory cache at the moment of its request, we suffer a “miss” and must add it to our memory. However, we can only store k pages in our cache: if we are to add another page when we are already storing k pages, we have to evict a page. Our goal is to minimize how many misses we suffer.

Example: Suppose our request sequence is A, B, A, C, A and $k = 2$. Then the algorithm suffers two misses immediately for the first two requests, after which we have a memory cache of $[A, B]$. When A is requested again, we do not have a miss since A is still in our cache. When C is requested, we have to evict a page: suppose we evict A . Then when A is requested one last time, we suffer another miss since we have evicted it for C .

Months ago we showed that the algorithm $\text{LONGEST-FORWARD-DISTANCE}$ (the algorithm that evicts the page that will be requested furthest in the future) is optimal. However, suppose we now consider this problem from an *online* setting: we can only base our decision of which pages to evict based on everything we’ve seen so far, and not what will come in the future.

4. (5 points) Convince me that no algorithm can expect to perfectly match the optimal solution on every input. That is, for every online algorithm, there is an input on which it incurs more misses than the optimal algorithm.

Problem Set 5: NP-Completeness and Approximations

Due: Friday 12/10 11:59PM

Solution: Let $k = 2$ and suppose the first 3 page requests are A, B, C . Then at the request C , one of A or B has to be evicted. If the algorithm evicts A , then it is suboptimal on the instance A, B, C, A ; if it instead evicts B , then it is suboptimal on the instance A, B, C, B .

Knowing this, we still want to rank our algorithms by how well they perform: let $C(A, I)$ denote the number of misses an algorithm A incurs on input I and $C(\text{OPT}, I)$ the optimal number of misses on input I (ie, the number of misses LFD would have). We say A is $\rho(A)$ -competitive if there is some constant c so that for all possible inputs I we have that $C(A, I) \leq \rho(A)C(\text{OPT}, I) + c$ (in English, for every possible input the cost of our algorithm on that input is at most a factor of $\rho(A)$ worse than the optimal cost on that input plus some constant).

5. (5 points) Consider the algorithm LFU that always evicts the page that has been requested least overall so far. Convince me that $\rho(\text{LFU}) \geq x$ for every number x (an immediate consequence of this is that there's no way to upper bound the competitive ratio of this algorithm).

Solution: Consider the input with $k = 2$ that repeats A $2x$ times, then switches off between pages C and D $2x$ times. The optimal solution has 3 misses, and removes A immediately to make room for both C and D . However, LFU will repeatedly swap C and D , giving us $1 + 4x$ misses. Hence we have more $\frac{4}{3}x > x$ as many misses as the optimal solution, showing that the competitive ratio of LFU is greater than x .

6. (10 points) Convince me that any algorithm must have competitive ratio at least k .

Solution: This solution uses $k + 1$ distinct pages $P_1, \dots, P_{k+1}$. Let $r_i = P_i$ for the first $k + 1$ requests. Let e_i denote the page that gets evicted for r_i , if such a page exists. By setting $r_i = e_{i-1}$ for $i > k + 1$, we guarantee that r_i is never in the cache at the time of its request. Hence, the algorithm suffers n misses over the first n requests. However, note that the optimal solution only gets a miss every k items (after the original k). Hence, for each n we can create an instance I so that $C(A, I) > n$ while $C(\text{OPT}, I) < \frac{n}{k} + k$. Hence the competitive ratio of any algorithm must be at least k .

7. (10 points) Let's try to break a sequence of pages into phases: each phase is the maximal length sequence of page requests since the previous phase that contains at most k distinct page requests. Let's call an algorithm "good" if in any phase it only evicts pages not yet seen in the current phase. Convince me that any "good" algorithm has competitive ratio $O(k)$.

Solution: If $k = 1$ then the statement is obvious, so we will assume $k > 2$. Within each phase, the algorithm can only have at most k misses (one for each item added to the cache). As for the optimal solution, let P be the first page requested in a particular phase. If P is not in the optimal solution's memory at that time, then the optimal solution must have a miss in this phase. If it is in the memory, there must be a miss other than the first page in the previous phase, since this page wasn't ever requested in the previous phase, which itself had k unique page requests.

Hence for each phase there is a unique request we know that the optimal solution must have missed on. Summing up over all phases gives us the desired answer.

Problem Set 5: NP-Completeness and Approximations

Due: Friday 12/10 11:59PM

8. (5 points) Consider the algorithm that uses phases as defined as above: this algorithm only evicts pages not requested in this phase, chosen arbitrarily. Convince me that this algorithm is ‘good’ as defined above.

Solution: This is really just a free 5 points to be honest. This algorithm only evicts pages not seen in the current phase, so it’s “good”.

9. (10 points, EXTRA CREDIT) Let’s say a randomized algorithm A is $\rho(A)$ competitive if $E[C(A, I)] \leq \rho(A)C(\text{OPT}, I) + c$ for all inputs I , where $E[C(A, I)]$ denotes the expected cost of A on input I . Consider the same algorithm as above, but it picks the pages to evict randomly. Show that this algorithm is $\log k$ -competitive.