CS170 Final - Solutions

Name:

SID:

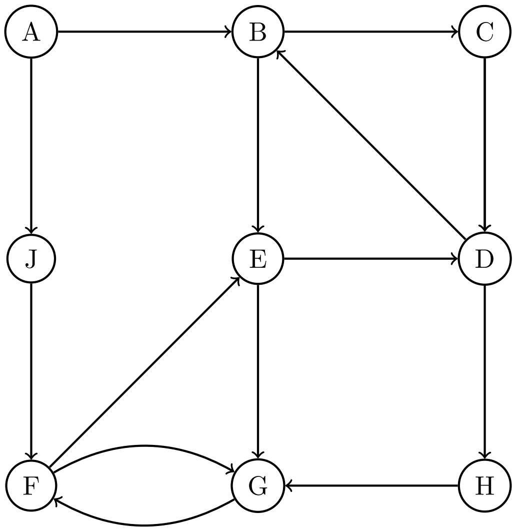
GSI and section time:

Answer all questions. Read them carefully first. Be precise and concise. The number of points indicate the amount of time (in minutes) each problem is worth spending. Not all parts of a problem are weighted equally. Write in the space provided, and use the back of the page for scratch. By "reduction" in this exam it is always meant "polynomial-time reduction." Also, when you are asked to prove that a problem is **NP**-complete, no need to show that it is in **NP**, unless asked to do so.

Good luck!

1 DFS *(10 points)*

For the directed graph below, draw the DAG of the strongly connected components.



The strongly connected components are $\{A, J, BCDEFGH\}$

2 Linear programming (10 points)

Your linear program is this:

$$
\begin{aligned}
&\max \;\; x_1 + x_2 \\
&x_1 \leq 1 \\
&x_2 \leq 1 \\
&x_1, x_2 \geq 0
\end{aligned} \tag{1}
$$

(a) Draw the feasible region. What is the optimal solution? How many steps will Simplex take from $(0, 0)$? What are the multipliers that prove optimality?

(Feasible region, not drawn here, is the square whose vertices are $(0, 0)$, $(1, 0)$, $(0, 1)$, and $(1, 1)$.) The optimal solution is at $x_1 = 1, x_2 = 1$ with value 2. Simplex will take two steps to reach this optimal solution. The multipliers that prove optimality are $y_1 = 1, y_2 = 1$.

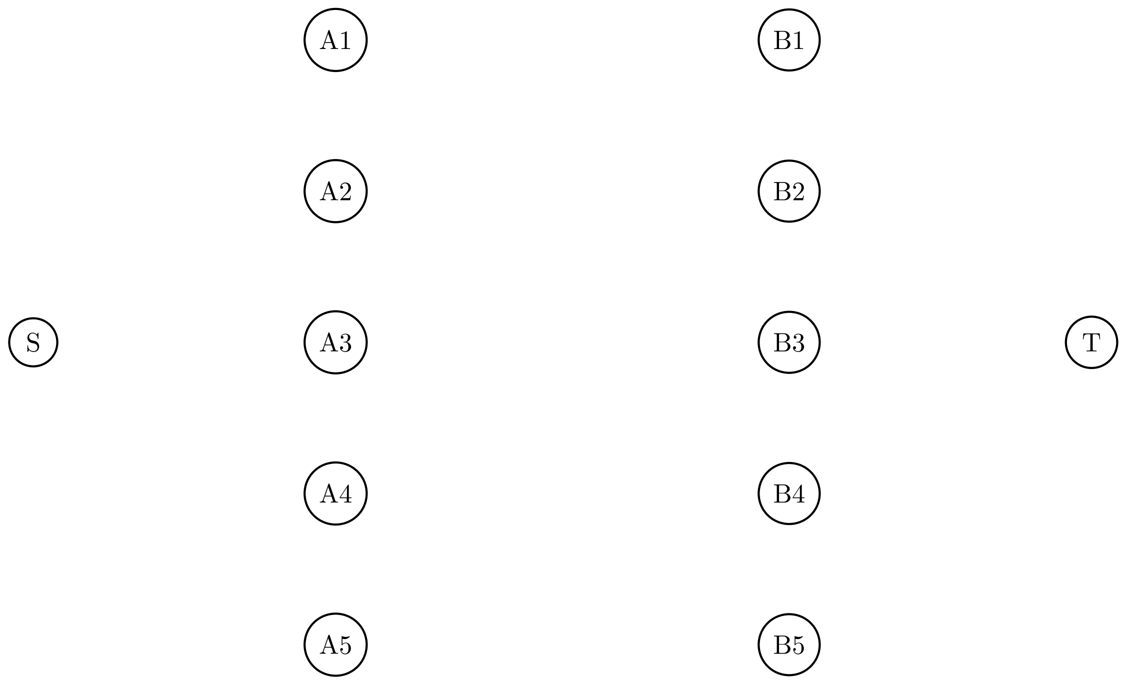
(b) The above linear program had a unique optimal solution. Can a linear program have exactly two distinct optimal solutions? Justify your answer.

No. Consider the line segment between these two solutions. This line is within the feasible region, since the region is convex. All of these points on the line have the same value as the two solutions. So, given two distinct optimal solutions, we can generate infinitely many more.

3 Assigning students to Sections by Max Flow *(15 points)*

Professor Reynard–Ulysses Nutts is teaching a class with 100 students and five sections A1, A2, ..., A5, with 20 students each, and you are his TA. Now, the professor believes that it is important for students to mingle in a class. For this reason, after the midterm, he wants to redistribute the students in five new sections, B1, B2, ... , B5, so that each of the new sections contains no more than 4 students from each of the old sections.

(a) You create a max-flow problem that accomplishes this. The nodes of the network are shown below. Indicate in the sketch the edges with the appropriate capacities (no need to draw all 35 edges). Then describe the max-flow showing that the professor's scheme is doable.



- Add edges (s, A_i) with capacity 20.
- Add edges (A_i, B_j) with capacity 4.
- Add edge (B_j, t) with capacity 20.

The max flow solution completely saturates every edge.

(b) “You were lucky,” professes the professor. “But what if the max flow you found was fractional? You cannot transfer 3.4 students from section A2 to Section B1.” How do you respond to that?

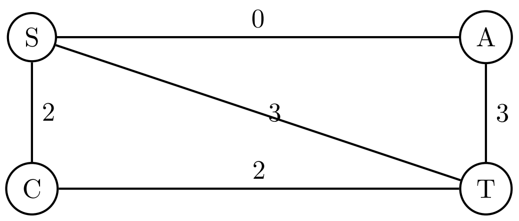
Using any standard flow algorithm on a graph with integer capacities will lead to a flow that is integer. So, we will not find a max flow that is fractional.

(c) Five more students register in the class. “No problem,” says the professor. “Just increase the section capacities to 21 students.” How do you prove to Professor R.-U. Nutts that his scheme is now impossible?

The cut $\{S, a_1, \ldots, a_5\}$ has weight 100, but the total flow needed is 105. Since any cut is an upper bound on the maximum flow, we know the max flow can not exceed 100.

4 Bottleneck Dijkstra *(35 points)*

In the *bottleneck path problem* you are given an *undirected* graph (V, E) with $|V| = n$ nodes, $|E| = m$ edges (as always we are assuming that $n < m$), and distances $\ell(i, j)$ on the edges, and two nodes s and t, and you are asked to find the *bottleneck* B, that is, the smallest number such that there is a path from s to t on which all edges have length B or less. You do not need to find the actual path. For example, in the graph shown, the best bottleneck is 2, because of the path $S - C - T$.



(a) You can solve this problem by a simple modification of Dijkstra. Show how would you change the statement

$$\ldots\texttt{if}\ dist(v) > dist(u) + \ell(u, v)\ \texttt{then}\ dist(v) = dist(u) + \ell(u, v)\ldots$$

in Dijkstra's algorithm to accomplish this. Argue very briefly that it works.

Modify the statement to if $dist(v) > \max(dist(u), \ell(u, v))$ then $dist(v) = \max(dist(u), \ell(u, v)$. The proof works very similarly to Dijkstra's proof. We can use induction. Assume that the first i vertices removed from the queue have the correct bottleneck values. For the $i + 1$th vertex say u which was the minimum in the queue, if there was a shorter bottleneck path $s \rightarrow u$, consider the last vertex v on the path athat is among the first i vertices. Let w be the next vertex. Then w must be the minimum in the queue, a contradiction unless $w = v$.

(b) But suppose that you want instead to use divide-and conquer to solve this problem. You want to divide-and-conquer the *edges* of the graph according to their lengths. If the graph has m edges (assume that m is a power of 2), you want your recurrence to be

$$T(m) = T(\frac{m}{2}) + O(m).$$

Would this algorithm run faster than Dijkstra's algorithm implemented using d-ary heap? Justify briefly your answer.

Yes, this algorithm is linear in the number of edges, while Dijkstra's algorithm is proportional to $O(m \log m)$ (this is assuming on a connected graph where the number of edges is at least the number of nodes).

(c) Your divide-and-conquer algorithm, call it `bottleneck(`V, E`)`, works in two steps.

In the first step you divide the m edges of the graph into two *disjoint* sets H (for high) and L (for low) such that $|L| = |H| = \frac{m}{2}$, and every edge in L has length $\leq$ to that of every edge in H. Can this step be done in $O(m)$ time (as required in the recurrence)? Justify briefly.

Yes. We can first find the median in $O(n)$ time. Then, we can split the edges into two sets of equal size in linear time once we have the median.

(d) The second step is this recursive call:

```
2.  If there is a path from $s$ to $t$ in the graph $(V, L)$ return bottleneck( ...  )
```

Fill in the dots and justify your answer briefly.

return bottleneck(V, Ls, t). If there is a path from s to t in the graph (V, L), the the bottleneck is due to an edge in L, and it doesn't increase on deleting the edges of H.

(e) The last line of the algorithm handles the case where there is no path from s to t in (V, L):

```
else return bottleneck(V',E'), where $V'$ is ...  and $E'$ is ...
```

Describe carefully how the set of nodes V' and the set of edges E' is constructed, and briefly justify . Remember that the graph is undirected, E' must contain at most $\frac{|E|}{2}$ edges, and this step must also run in linear time.

Let V' be the connected components of (V, L). Let E' be the edges in H that go across the connected components in V'. We can do this in linear time by doing a dfs on (V, L) to find connected components. Then, adding the edges in E' just requires iterating through edges in H.

(f) Could you solve the same way the *Bottleneck spanning tree* problem (finding the smallest number B such that there exists a spanning tree whose edges all have weight at most B)? Justify very briefly.

We can still do this divide and conquer. But now, we will modify our check so that we check if everything is connected, rather than only s and t being connected.

5 Complete the sentences: *(20 points)*

(a) The solution of the recurrence $T(n) = 4T(\frac{n}{2}) + n^2$ is...

By master theorem, the solution to this is $T(n) = O(n^2 \log n)$

(b) You can linearize a dag by doing depth-first search and then ordering the nodes by...

post order, from largest to smallest.

(c) Dijkstra's algorithm works when the edge lengths are ...

non-negative. (A common answer that only got half credit was "positive", but edges lengths can be 0.)

(d) An operation of the union-find data structure with path compression on n elements can never take longer than...

$O(\log n)$ time. (Note that the possible runtime of a single operation is not the same as its amortized runtime.)

(e) A graph (direct or undirected) has a cycle if and only if depth-first search finds a...

back edge. (This is not the same as "an edge already visited", since this latter category includes cross edges, which are not necessarily part of a cycle).

(f) In a Huffman tree with all frequencies different, the item with the second-lowest frequency will always be placed at...

lowest level (leaf is not accepted, since every item will be at a leaf).

(g) The greedy algorithm for Set Cover approximates the optimum by a factor of...

$O(\log n)$, where n is the number of objects to be covered. (Some students incorrectly made reference to k, the optimal number of sets).

(h) The FFT on n points can be run with n processors in parallel time...

$O(\log n)$

(i) The FFT on n points can be run with $\sqrt{n}$ processors in parallel time...

$O(\sqrt{n} * \log(n))$

(j) In Multiplicative Weights with n experts and T periods you can come O(...) close to the optimum total performance.

$\sqrt{\frac{\ln n}{T}}$.

6 True or false? Circle the right answer. No explanation needed (45 points)

(No points will be subtracted for wrong answers, so guess all you want!)

T **F** The Floyd-Warshall algorithm for all-pairs shortest paths works even when there are negative cycles.

False The Floyd-Warshall algorithm will still fail.

T **F** The reduction we did in class is from MAX FLOW to MATCHING, not the other way.

False

T **F** The Bellman-Ford algorithm for shortest paths runs in $O(|V|^2)$ time.

False This runs in $O(|V||E|)$ time.

T **F** The dynamic programming algorithm for KNAPSACK WITHOUT REPETITION runs in polynomial time.

False Knapsack still runs in exponential time with respect to the input.

T **F** The algorithm for MST that is best to parallelize is Kruskal's.

False The algorithm that is best to parallelize is Bovruka's.

T **F** Huffman's algorithm runs in linear time.

False We need to use a heap to implement this efficiently, which makes the runtime $O(n\ logn)$.

T **F** Euclid's algorithm on n-bit numbers can run faster than $O(n^3)$ time.

False Euclid's algortihm runs in $O(n^3)$ time (there are n iterations, each of which requires an $O(n^2)$ time division).

T **F** Doubling the capacities in MAX FLOW just doubles the values of the maximum flow

True If we look at the value of any cut, we can see that doubling the capacities will just double the value of that cut. Since all cuts will be doubled, and since the min cut equals the max flow, the max flow will also be doubled.

T **F** Adding 1 to the capacities in MAX FLOW just adds 1 to the values of the max flow.

False Consider the following counterexample with 3 nodes A, B, C, with edges $A \to B, B \to C, A \to C$, all with capacity 1, and we want the max flow from A to C. Then, the max flow is originally 2, but adding 1 to all capacities will make the max flow 4.

In the remaining questions there are four possible answers: (1) True (T); (2) False (F); (3) True if and only if **P** = **NP** (=); (3) True if and only if **P** ≠ **NP** (≠). Circle one.
Note: By "reduction" in this exam it is always meant "polynomial-time reduction."

T **F** = ≠ The minmax strategy in a zero-sum game can be found in polynomial time.

True We know that finding the minmax strategy is in P.

T **F** = ≠ There is no reduction from RUDRATA PATH to MAX FLOW.

≠ Any **NP**-complete problem can always be reduced to any other **NP**-complete problem, although the reduction may not be clear or direct.

T **F** = ≠ There is a polynomial-time algorithm for INDEPENDENT SET.

= Only if **NP**-complete problems are all within **P**.

T F $=$ $\neq$ There is a known polynomial-time algorithm for INDEPENDENT SET.

False In other words, we do not know that **P** = **NP**.

T F $=$ $\neq$ There is a reduction from FACTORING to RUDRATA PATH.

True Since Factoring is a problem in **NP**, and Rudrata Path is **NP**-complete, then there exists a reduction from Factoring to Rudrata path.

T F $=$ $\neq$ INTEGER LINEAR PROGRAMMING is **NP**-complete.

True This is true regradless of whether or not $P = NP$.

T F $=$ $\neq$ LINEAR PROGRAMMING is **NP**-complete.

= We know that Linear program is in **P** (and therefore **NP**). However, we do not know if it is **NP**-hard. There can be a reduction from an **NP**-complete problem to Linear Programming if and only if **P** = **NP**.

T F $=$ $\neq$ There are problems in **NP** that cannot be solved in exponential time.

False As shown in homework, all problems in NP can be solved in exponential time.

T F $=$ $\neq$ There are computational problems that cannot be solved in exponential time.

True One example is the halting problem, which can not be solved in a finite amount of time.

T F $=$ $\neq$ There are problems in **NP** that are neither in **P** nor **NP**-complete.

$\neq$ If **P** $\neq$ **NP**, then we can prove that there must be problems in between the two, e.g. we believe factoring to be in this category. Otherwise, then every problem in **NP** is both **P** and **NP**-complete.

T F $=$ $\neq$ There is a problem in **NP** that is not **NP**-complete.

$\neq$ Only if **NP** $\neq$ **NP**-complete.

T F $=$ $\neq$ Any two problems in **P** can be reduced to each other in polynomial time.

True The reduction from problem A to problem B works as follows. First, solve A. If there is a solution, create an instance of B that has a solution. Otherwise, create an instance of B that does not have a solution.

T F $=$ $\neq$ Any two problems in **NP** can be reduced to each other in polynomial time.

= Any problem in **NP** can be reduced to any problem in **NP**-complete in polynomial time, so this is only true if **NP** = **NP**-complete.

T F $=$ $\neq$ There is a polynomial-time algorithm that approximates the TSP with triangle inequality with a factor of two.

True This was covered in the textbook.

T F $=$ $\neq$ There is a polynomial-time algorithm that approximates the TSP without triangle inequality with a factor of two.

= A polynomial time algorithm that optimally solves TSP will also approximate TSP with a factor of two. But the textbook mentions that if **P** $\neq$ **NP**, then there cannot be an approximation algorithm for TSP.

7 Dynamic Programming Plus *(25 points)*

(a) You are hired by a startup named Kale Tacos to plan its deployment of food trucks at street corners of Shattuck Ave. There are n street corners where a truck could be deployed, numbered 1 through n. You have researched carefully the revenue a truck will get from street corner i, but this amount depends on whether or not another truck is deployed at one of the adjacent corners $i-1$ or $i+1$ (street corners 1 and n have only one adjacent street corner). That is, your data consists of n number pairs $(h_i, \ell_i), i = 1, \dots, n$ where h_i (which is non-negative) is the revenue when there is no truck in any one of the adjacent corners, and ℓ_i (which is smaller than h_i and may be negative) is the revenue when there is another a truck in one of the adjacent corners, or if there are trucks in both. You want to find the optimum placement of trucks (there is no limit to the number of trucks you can deploy).

You use, of course, Dynamic Programming! You define $R(i), i = 1, \dots, n-1$ to be the revenue you can get from corners $1, 2, \dots, i$ assuming that there is no truck at corner $i+1$, and $R'(i), i = 1, \dots, n$ to be the revenue you can get from corners $1, 2, \dots, i$ assuming that *there is* a truck corner $i+1$.

Fill the blanks:

$R(1) = h_1$

$R'(1) = max(0, l_1)$

For $i = 2, 3, \dots, n-1 \quad R(i) = \max(R(i-1), R'(i-1) + l_i, R(i-2) + h_i)$

For $i = 2, 3, \dots, n-1 \quad R'(i) = \max(R(i-1), R'(i-1) + l_i)$

The optimum revenue is computed as $R(n)$. (Note that the above subproblem was only defined up to $R(n-1)$; both a correct expression for $R(n)$ or just writing "$R(n)$" was accepted.)

The running time is O(n), because there are $O(n)$ subproblems and it takes $O(1)$ time for each.

(b) What if your company only has only a fixed number $k < n$ of trucks? Describe the appropriate subproblem for this situation, and estimate and justify the time complexity of the algorithm (no need to give the precise recurrence).

Add a parameter in our dp state representing the number of trucks we use. That is, let $T(i, j)$ be the revenue from placing j trucks on the first i corners, with no truck at $i+1$, and let $T'(i, j)$ be the revenue from placing j trucks on the first i corners, with a truck at $i + 1$. The time complexity would then be $O(nk)$, since we have nk subproblems, and each would still takes constant time to evaluate.

Common errors/mistakes: Forgetting that the DP state still needs to take into account the presence of a truck on the $i + 1$th corner; defining the subproblem as above but saying incorrectly that there were now k subproblems that took $O(n)$ time to run; and saying that because there are $O(nk)$ subproblems that the runtime would be $O(nk)$ (it is possible that a single subproblem takes more than constant time to solve).

(c) Suppose next that each location i has an arbitrary list $L_i = \{j_1, j_2, \ldots\}$ of competing locations which can be anywhere (instead of just the two competing locations $i - 1$ and $i + 1$), and the revenue at location i is h_i if there are no trucks in any competing location in the list, and it is ℓ_i otherwise. You have k trucks. Prove that the problem of optimum truck placement is now **NP**-complete. (*Hint:* Independent set..)

We reduce from indpendent set. Given a graph $G = (V, E)$. Let the set of trucks be V. Let the neighbors of each vertex i be the list of competing locations for truck i. Then, we set $h_i = 1, l_i = 0$ for all trucks i. We can show that we get an optimal value of k if and only if there exists an independent set of size k.

8 NP-completeness *(25 points)*

Note: By "reduction" in this exam it is always meant "polynomial-time reduction." For the reductions in Problem 8 mention the problem you are using, direction and construction of the reduction (*proofs are not necessary*). Also, when you are asked to show that a problem is **NP**-complete, no need to show that it is in **NP**, unless asked to do so.

(a) Recall the 3-Dimensional Matching problem, asking you to match n girls, n boys, and n pets given a list of compatible triples. We know that it is **NP**-complete. In the 4-Dimensional Matching problem you are given compatible *quadruples* of n boys, n girls, n pets, and n *homes*, and again you want to create n harmonious households accommodating them all. Fill the blanks in the following proof that 4-Dimensional Matching is **NP-complete**.

Proof: *We will reduce the problem 3D matching to the problem 4D matching*

Given an instance of the problem 3D matching we construct an instance of the problem 4D matching

as follows: Create n homes, which are compatible with every triple in our 3D Matching instance.

A common error was to add a home to each triple that was compatible with that triple.

■

(b) You are given a strongly connected directed graph $G = (V, E)$ and a budget B, and you are asked to find a subgraph (V, E') of G with $E' \subseteq E$ such that (1) (V, E') is strongly connected, and (2) $|E'| \leq B$. Fill in the blanks in the following proof that this problem is **NP**-complete.

Proof: *The above problem is a generalization of the Rudrata Cycle problem, because we can set $B = n$. We know that a strongly connected graph on n vertices with n edges is a cycle on the vertices.*

For every pair of search problems A and B, if problem A is a generalization of problem B then clearly,

problem B reduces to problem A.

Since we know that the problem Rudrata Cycle is NP-complete, the above problem is NP-complete. ■

(c) In the $\frac{1}{3}$-Independent Set problem you are given a graph (V, E) and you are asked to find an independent set of the graph of size exactly $\frac{|V|}{3}$. In other words, the target size g is not part of the input, but it is always $\frac{|V|}{3}$. Why is this special case of Independent Set **NP**-complete?

Consider the reduction from 3-SAT to Independent Set we studied in class. It takes an instance of 3-SAT and returns a graph that has an independent set of size exactly $|V|/3$ if and only if the 3-SAT instance has a satisfying assignment. Thus, this reduction is also a reduction from 3-SAT to $\frac{1}{3}$-Independent Set, which proves NP-completeness.

Alternatively, there is a messier reduction from Independent Set to $\frac{1}{3}$-Independent Set. Consider an instance of Independent Set with a graph G and target size g. If $g \geq |V|/3$, then we can add $3g - |V|$ vertices to G such that the new vertices all have edges to each other and the original vertices. We can then solve our modified G with $\frac{1}{3}$-Independent Set.

If $g < |V|/3$, then we add $(|V| - 3g)/2$ isolated vertices to G. We then solve our modified G with $\frac{1}{3}$-Independent Set and receive an independent set as an answer. From that independent set, we return only the vertices that were in our original graph.

(d) Fill the blanks in the following proof that the $\frac{1}{2}$-Independent Set problem (the variant where the goal is $\frac{|V|}{2}$) is **NP**-complete.

Proof: *From part (c), we know that* $\frac{1}{3}$-Independent Set *is NP-complete. So we will now reduce* $\frac{1}{3}$ *Independent set to* $\frac{1}{2}$*-Independent set. Given an instance* $G = (V, E)$ *of* $\frac{1}{3}$-Independent Set*, construct an instance* $G' = (V', E')$ *of* $\frac{1}{2}$*-Independent set as follows:*

define V' to be V with an additional $|V|/3$ isolated nodes and define E' to be exactly the same as E.

(Alternatively we could have reduced from Independent Set and considered the cases $g \geq |V|/2$ and $g < |V|/2$ in a similar manner to part (c)).

To complete a formal proof of the reduction, we will need to show that
G' has an independent set of size $|V'|/2$
if and only if

G has an independent set of size $|V|/3$. ■

(x) Extra credit, work on this only if you finished all else:

Recall the Non-Partisan Traveling Senator Problem (NPTSP) of the project. Suppose you want to prove that it is **NP**-complete, through a reduction. Which known **NP**-complete problem would you use? And which direction would this reduction go? Fill the spaces below:

I would reduce the problem TSP to the problem NPTSP

Show that the NPTSP is **NP**-complete.

Proof: Given an instance $G = (V, E)$ of TSP, we construct an instance of NPTSP as follows:

For each vertex $v \in V$, we construct a red vertex v_r and a blue vertex v_b. The edge (v_r, v_b) has cost 0, the edge (v_r, w_b) has cost infinity whenever v and w are distinct vertices in V, the edges (v_r, w_r) and (v_b, w_b) have the same cost as (v, w) in E.

Note that for any TSP path

$$v_1, v_2, \ldots, v_n$$

in G, we can construct a corresponding NPTSP path

$$v_{1,b}, v_{1,r}, v_{2,r}, v_{2,b}, v_{3,b}, v_{3,r}, \ldots, v_{n,r}, v_{n,b}$$

(depending on the parity of n, the last two vertices in the path may be $v_{n,b}, v_{n,r}$). By construction, we never have more than 3 vertices of the same color in a row. This NPTSP has the same cost as our TSP path, so the optimal NPTSP path has cost less than or equal to the cost of the optimal TSP path.

Now consider an optimal NPTSP path. If two consecutive vertices in this path have different colors, then they must be of the form v_r, v_b or v_b, v_r or else they would contribute a cost of infinity to the path. We know that three consecutive vertices cannot all have the same color. I claim that for any vertex $v \in V$, the vertices v_r and v_b must be consecutive vertices in the path. Suppose otherwise and without loss of generality say v_r appears first. From our above arguments, we know that the path from v_r to v_b must take the form

$$v_r, w_{1,r}, w_{1,b}, w_{2,b}, w_{2,r}, \ldots, w_{k,r}, w_{k,b}, v_b$$

Note that v_r and v_b must be the first and last vertices in our optimal path, because otherwise there would be three consecutive red vertices at the beginning or three consecutive blue vertices at the end. But then I could construct a cheaper path by moving v_b to the front. Thus, the optimal NPTSP path takes the form

$$v_{1,b}, v_{1,r}, v_{2,r}, v_{2,b}, v_{3,b}, v_{3,r}, \ldots, v_{n,r}, v_{n,b}$$

and there exists a corresponding TSP path

$$v_1, v_2, \ldots, v_n$$

in G with the same cost. It follows that the optimal TSP path has the same cost as the optimal NPTSP path.

If you had a correct solution to this bonus problem, then we took that into consideration when assigning letter grades. ■