CS 124 Homework 7: Spring 2024

Collaborators: Charlie, Annabel, Sophie, Ryan

No. of late days used on previous psets: 7
No. of late days used after including this pset: 9

Homework is due Friday Apr 26 at 11:59pm ET. Please remember to select pages when you submit on Gradescope. Each problem with incorrectly selected pages will lose 5 points.

Please also note that late days count half for this pset. In other words, every 12 hours used past the deadline will count as one late day. The latest submission deadline is Saturday Apr 27 at 11:59pm ET with the use of two late days.

Try to make your answers as clear and concise as possible; style may count in your grades. Assignments must be submitted in pdf format on Gradescope. If you do assignments by hand, you will need to scan your papers to turn them in.

Collaboration Policy: You may collaborate on this (and all problem sets) only with other students currently enrolled in the class, and of course you may talk to the Teaching Staff or use Ed. You may also consult the recommended books for the course and course notes linked from the timetable. You may not use generative AI or large language models, or search the web for solutions, or post the questions on chat forums. Furthermore, you must follow the "one-hour rule" on collaboration. You may not write anything that you will submit within one hour of collaborating with other students or using notes from such sources. That is, whatever you submit must first have been in your head alone, or notes produced by you alone, for an hour. Subject to that, you can collaborate with other students, e.g. in brainstorming and thinking through approaches to problem-solving.

For all homework problems where you are asked to give an algorithm, you must prove the correctness of your algorithm and establish the best upper bound that you can give for the running time. Generally better running times will get better credit; generally exponential-time algorithms (unless specifically asked for) will receive no or little credit. You should always write a clear informal description of your algorithm in English. You may also write pseudocode if you feel your informal explanation requires more precision and detail, but keep in mind pseudocode does NOT substitute for an explanation. Answers that consist solely of pseudocode will receive little or no credit. Again, try to make your answers clear and concise.

Problems

1. Let $G = (V, E)$ be a graph. A *vertex cover* of G is a set $C \subseteq V$ such that all edges in E have at least one endpoint in C. That is, each edge is adjacent to at least one vertex in the vertex cover. The **(Minimum) Vertex Cover (Search)** problem is, given a graph G, to find a vertex cover with a minimum number of vertices. Minimum Vertex Cover is NP-complete.

   Given a graph $G = (V, E)$, a set $S \subseteq V$ is an independent set if there are no edges within S. The **Independent Set (Search)** problem is, given a graph G, to find an independent set with a maximum number of vertices. Independent Set is also NP-complete.

   Independent Set and Minimum Vertex Cover (like all pairs of NP-complete problems) are equivalent problems: given an algorithm A that outputs a minimum vertex cover on every input graph

G, the algorithm that outputs $V \setminus A(G)$ on input G solves the maximum independent set problem. Below you will prove that this equivalence does not extend to approximations and indeed a 2-approximation to Minimum Vertex Cover does not yield a c-approximation to Independent Set for any constant c.

(a) **(5 points)** For every constant $c > 1$, show that there exists a graph and a 2-approximation of its minimum vertex cover such that the corresponding independent set is not within a factor of c of the maximum independent set.

Proof. Let use have a family of graphs $G = (V, E)$ defined by $2n$ vertices and n edges. Each vertex will only be connected by 1 edge to another vertex, so the entire graph has n different pairs of 2 vertices which are only connected to each other by 1 edge, and not connected to any other pair of vertices. Then, the optimal vertex cover of G is clearly of size n because for all n disjoint pairs of vertices connected by an edge, we simply must choose 1 of them to include in the optimal vertex cover. In addition, the max independent set is size n because having more than n vertices would mean that at least one of the vertices is connected by edge to another.

However, from the 2-approximation vertex cover algorithm in lecture notes 19, we know the 2-approximation for the vertex cover of G is the set of all $2n$ vertices (which is within a factor of 2 from the optimal solution). It follows that the max independent set corresponding to this vertex cover must be the empty set (size 0).

However, for the approximation independent set to be within a factor of c of the maximum independent set, this must mean the following must be true:

$$\frac{1}{c}n \leq 0 \leq cn$$

Where we derived the size of the approximation independent set is 0 and the size of the optimal independent set is n. However, this inequality is evidently false for all $c > 1$ because the number of vertices in the graph must be non-zero (n > 0). Therefore, for this family of graphs, the corresponding independent set for its 2-approximation of minimum vertex cover is not within a factor of c of the max independent set for all $c > 1$. □

(b) **(24 points)** Prove that if there exists a polynomial time algorithm for approximating the size of the maximum independent set in a graph G to within a factor of 2, then for every $\epsilon > 0$, there is a polynomial time algorithm for approximating the size of the maximum independent set in a graph to within a factor of $(1+\epsilon)$. The degree of the polynomial may depend on ϵ. (**Hint:** for a starting graph $G = (V, E)$, consider the graph $G \times G = (V', E')$, where the vertex set V' of $G \times G$ is the set of ordered pairs $V' = V \times V$, and $\{(u, v), (w, x)\} \in E'$ if and only if

$$\{u, w\} \in E \text{ or } \{v, x\} \in E.$$

If G has an independent set of size k, then how large an independent set does G' have? If you have an α approximation to the independent set size in G' what kind of approximation to the independent set size in G do you get?)

Proof. We define the polynomial time algorithm for approximating the size of the MIS in a graph G to within a factor of $1+\epsilon$ for all $\epsilon > 0$.

For the sake of readability, we define $a = \lceil -\log\log(1+\epsilon) \rceil$, which varies with $\epsilon > 0$. First, we construct graph G^{2^a} by repeatedly taking the Cartesian product of G for 2^a times. Thus, we can compute it as such:

$$G^{2^a} = \underbrace{G \times G \times \cdots \times G}_{2^a \text{ times}}$$

To get this, we can do:

$$G^2 = (G \times G)$$

$$G^4 = (G \times G) \times (G \times G)$$

$$G^8 = (G \times G) \times (G \times G) \times (G \times G) \times (G \times G)$$

$$\cdots$$

And so on to compute G^{2^a}. Then, we use the 2-approximation algorithm as assumed in the problem description to get a 2-approximation of the MIS which we call A. Finally, we return the a-th square root the size of A, $|A|^{(1/2)^a}$, which yields a approximation within a factor of $(1+\epsilon)$ to the size of the MIS.

Correctness: We will prove that our algorithm returns a $(1+\epsilon)$ approximation of the size of the MIS of G.

We will first prove that if the size of the MIS in G is k, then the size of the MIS in G^2 is exactly k^2. To prove this, we will first show that the size of the MIS of G^2 is less than or equal to k^2 and the size of the MIS of G^2 is greater than or equal to k^2. Before we begin, let us define the MIS of G as $I = \{x_1, ..., x_k\}$. We also denote the MIS of G^2 as I'.

Upper Bound Direction: Consider an independent set P in the graph G^2 (not necessarily the MIS). Let us denote A as the set of all first coordinates of the vertices in P. Let us denote B as the set of all second coordinates of the vertices in P.

Thus, by the construction of the Cartesian product $G^2 = G \times G$, we know that A and B are both independent sets. Assume for the sake of contradiction that A and B were not independent sets. Then, it follows from the problem description that there would exist two points $\{(u,v),(w,x)\}$ in G^2 with $u, w \in A$ and $v, x \in B$ such that $(u,w) \in E$ or $(v,x) \in E$. Therefore, there would exist an edge between two points in P, violating the independence of P. Therefore, we have reached a contradiction and the statement is proved true.

We also have that A and B are of at most size k since the I has size k (by optimality, A and B cannot have a bigger size than the maximum). This is equivalent to $|A| \leq k$ and $|B| \leq k$. Note that P can be constructed from a subset of the Cartesian

product $A \times B$ because the first coordinate is from A and the second coordinate is from B. Therefore, we also know that $|P| \leq |A| * |B| \leq k^2$. Therefore, the size of an independent set P in G^2 is upper bounded by k^2.

Lower Bound Direction: We will now prove that the size of the I' is lower bounded by k^2. Recall that we have the MIS of G as $I = \{x_1, ..., x_k\}$. Then, we simply construct an independent set on G^2 by selecting vertices (x_i, x_j) in G^2 where $x_i, x_j \in I$. Note that the independence of this set comes from the fact that any $x_i, x_j \in I$ must be independent because they belong in I. Therefore, for any two points $\{(x_i, x_a), (x_j, x_b)\}$ with $x_a, x_b, x_i, x_j \in I$, no edge $(x_a, x_b), (x_i, x_j) \notin E$, meaning this independent set is indeed independence.

Moreover, this independent set on G^2 is of size k^2 because x_i has k values it can take on and x_j also has k values it can take on. Thus, since we have proved that there indeed exists an independent set of size k^2 in G^2, it follows that the MIS of G^2 must be *at least* this large, so the MIS of G must have at least size k^2.

Therefore, since we have proved that $k^2 \leq |I'| \leq k^2$, we have proved that the size of the MIS of G^2 is exactly k^2 if the MIS of G has size k.

Next, we will use this fact to prove that our algorithm returns a $(1+\epsilon)$ approximation of the size of the MIS of G.

Let k be the size of I, the MIS of G. We assume the 2-approximation algorithm discussed in the problem is correct. Thus, we denote A as the size of the 2-approximation given by this algorithm on our graph G^{2^a}. Finally, note that using the proof above, we can inductively show that if I has size k, then the MIS of G^{2^a} has size k^{2^a}. Thus, by the definition of 2-approximation, we have:

$$\frac{1}{2}k^{2^a} \leq A \leq 2k^{2^a}$$

Next, in our algorithm definition, we take the square root of A a times. This yields:

$$\frac{1}{2^{(1/2)^a}}k \leq A^{(1/2)^a} \leq 2^{(1/2)^a}k$$

Thus, our algorithm returns the value $A^{(1/2)^a}$ which we will next prove is within a factor of $(1+\epsilon)$ from the size of I. For simplicity, we denote $A^{(1/2)^a}$ as S.

$$\frac{1}{2^{(1/2)^a}}k \leq S \leq 2^{(1/2)^a}k$$

Since our solution cannot be more than the size of I by optimality, then the upper bound narrows to k.

$$\frac{1}{2^{(1/2)^a}}k \leq S \leq k$$

Thus, since the upper bound is satisfied already, to prove that our outputted solution size is within a factor of $1+\epsilon$ from the optimal, we must prove that our outputted solution size is greater or equal to $\frac{k}{1+\epsilon}$. Recall that we defined a at the beginning to be $a = \lceil -\log\log(1+\epsilon) \rceil$. Here, we plug in $a = -\log\log(1+\epsilon)$ in:

$$\frac{1}{2^{(1/2)^{(-\log\log(1+\epsilon))}}} k \leq S \leq k$$

Simplifying, we get:

$$\frac{1}{1+\epsilon} k \leq S \leq k$$

Therefore, since $k \leq (1+\epsilon)k$ for $\epsilon > 0$ and $a = \lceil -\log\log(1+\epsilon) \rceil \geq -\log\log(1+\epsilon)$, then we have proved that our algorithm returns a $(1+\epsilon)$ approximation of the size of the maximum independent set of G.

Runtime: Let $n = |V|$ and $m = |E|$. First, our algorithm must construct the graph G^{2^a}. This graph by definition contains a total of n^{2^a} vertices and $n^{2^{a+1}}$ edges. Since we assume that edge and vertex creation takes $O(1)$ time, this step takes $O(n^{2^a} + n^{2^{a+1}}) = O(n^{2^{a+1}})$ time. Next, we call the 2-approximation algorithm as discussed in the problem description, which is given as polynomial. Therefore, we can express its runtime as $O(n^{c2^a})$ with a constant factor c. Then, we take a total square roots of the output of this polynomial algorithm. Assuming each square root takes $O(1)$ time, this takes $O(a)$ time. Therefore, our total runtime is $O(n^{2^{a+1}} + n^{c2^a} + a) = O(n^{c2^a})$. However, if substitute $a = \lceil -\log\log(1+\epsilon) \rceil$, we get a runtime of $O(n^{c2^{\lceil -\log\log(1+\epsilon) \rceil}})$ in terms of ϵ which we assume is constant. Simplifying, we have a final runtime of:

$$O(n^{\frac{c}{\log(1+\epsilon}})$$

□

2. Recall the MAX CUT problem from class where the goal is to output a set S of vertices of a given graph so as to maximize the number of edges with exactly one endpoint in S.

   Recall the randomized algorithm from class for this problem which picks S by flipping an independent random coin for each of the vertices $v_1, \ldots, v_n$ of the graph and adding v_i to S if and only if the i-th coin is heads. We asserted in the class that the expected size of the cut output is $m/2$ where m is the number of edges in the graph.

   Now consider the following algorithm: Given $0 \leq j \leq n-1$ and $T \subseteq \{1, \ldots, j\}$, let $C_j(T)$ denote the expected value of the cut S defined as follows. Among the vertices $v_1, \ldots, v_j$, S only contains the vertices indexed by T, and among the remaining vertices $v_{j+1}, \ldots, v_n$, each one is included in S independently with probability $1/2$.

   In this notation, we can interpret what we showed in class as saying that $C_0(\emptyset) = m/2$.

   (a) **(7 points)** Let $\text{cut}_j(T)$ denote the number of edges crossing from $\{1, \ldots, j\} \cap T$ to $\{1, \ldots, j\} \setminus T$. Let m_j denote the number of edges that touch at least one vertex among $\{j+1, \ldots, n\}$. Give an expression for $C_j(T)$ in terms of $\text{cut}_j(T)$ and m_j.

Proof. To find an expression for $C_j(T)$, we will group every edge into one of two cases: either both of their endpoints lie in the set $\{v_1, ..., v_j\}$ or at least one endpoint lies in the set $\{v_{j+1}, ..., v_n\}$. Thus, we will split all edges of G into one of two cases:

Case 1: Here, we will again split up case one, where at least one endpoint in $(u, v) \in E$ is in the set $\{v_{j+1}, ..., v_n\}$, into 3 different cases:

Case 1.1: We first consider the case where endpoint of (u, v) lies in the set $\{v_{j+1}, ..., v_n\}$ and the other endpoint lies in the vertex set indexed by $\{1, \ldots, j\} \cap T$. Let us say w.l.o.g. that u lies in the set $\{v_{j+1}, ..., v_n\}$ and v lies in the vertex set indexed by $\{1, \ldots, j\} \cap T$. Then, the probability that the edge (u, v) is in the cut is if the vertex u is NOT assigned to S, since v is already in S. From the problem description, we know that the probability that u is assigned to S is $1/2$. Therefore, the probability of the complement, the probability that u is not assigned to S is $1 - 1/2 = 1/2$. Therefore, the probability that the edge in this case is in the cut S is $1/2$.

Case 1.2: Next, we consider the case where endpoint of (u, v) lies in the set $\{v_{j+1}, ..., v_n\}$ and the other endpoint lies in the vertex set indexed by $\{1, \ldots, j\} \setminus T$. Let us say w.l.o.g. that u lies in the set $\{v_{j+1}, ..., v_n\}$ and v lies in the vertex set indexed by $\{1, \ldots, j\} \setminus T$. Then, the probability that the edge (u, v) is in the cut is if the vertex u IS assigned to S, since v is not in S (and cannot be randomly assigned to S). From the problem description, we know that the probability that u is assigned to S is $1/2$. Therefore, the probability that the edge in this case is in the cut S is $1/2$.

Case 1.3: Finally, we consider the case when both endpoints of (u, v) lie in the set $\{v_{j+1}, ..., v_n\}$, then the only way for this edge to be put in the cut S is if ONE of the endpoints is randomly included. The probability that u is included in S and v is not is $1/2 * 1/2 = 1/4$ and the probability that u is not included in S and v is included is $1/2 * 1/2 = 1/4$. Therefore, the probability that an edge in this case is included in the cut S is $1/4 + 1/4 = 1/2$.

Case 2: We will now consider the case when both endpoints of an edge $(u, v) \in E$ lie in $\{v_1, ..., v_j\}$. The only way for this edge to be in the cut S is if, without loss of generality, u belongs to the set of vertices indexed by $\{1, \ldots, j\} \cap T$ and v belongs to the set indexed by $\{1, \ldots, j\} \setminus T$. Note that this is by definition the value of $\text{cut}_j(T)$. Thus, the exact number of edges in the cut for case 1 is the value $\text{cut}_j(T)$ (with no randomness).

These cases cover all of the edges in the graph. Note that in all cases in case 1, every edge has a probability to be included in the cut S with probability $1/2$ while the number of edges included in the cut S in case 1 is just $\text{cut}_j(T)$. Since the number of edges grouped into case 2 is by definition m_j and each edge has a probability of $1/2$ of adding one edge to the cut S, we know by the linearity of expectation that case 2 contributes an expected value of $\frac{1}{2}m_j$ edges to $C_j(T)$. Therefore, we can express $C_j(T)$, the expected number of edges in the cut, as the sum of the expected value of

all of these cases:

$$C_j(T) = E(\text{Case 2}) + E(\text{Case 1})$$

$$C_j(T) = E(\text{cut}_j(T)) + \frac{1}{2}m_j$$

$$C_j(T) = \text{cut}_j(T) + \frac{1}{2}m_j$$

□

(b) **(5 points)** Given j and T as above, let T' be obtained by either adding v_{j+1} to T or not. Give an algorithm for deciding whether or not to add v_{j+1} to T so as to maximize $C_{j+1}(T')$, and provide justification.

Proof. We will describe an algorithm that decides whether or not to add v_{j+1} to T so as to maximize $C_{j+1}(T')$, given j and T, as such: for vertex v_{j+1}, find the total number of edges it has to the vertices belonging to the vertex set indexed by $\{1,\ldots,j\} \setminus T$ and the total number of edges it has to vertices belonging to the vertex set indexed by $\{1,\ldots,j\} \cap T$. Let us denote the former quantity as P and the latter quantity as M. Thus, we decide to add vertex v_{j+1} if and only if $P \geq M$.

We provide justification our algorithm as follows: we know that $C_j(T') = \text{cut}_j(T') + \frac{1}{2}m'_j$ from part (a). In addition, $\text{cut}_j(T)$ is by definition the number of edges crossing from $\{1,\ldots,j\} \cap T$ to $\{1,\ldots,j\} \setminus T$.

If we choose to not add v_{j+1} into the set T, then every edge that connects from v_{j+1} to vertices in $\{1,\ldots,j\}$ T will then be an edge crossing the cut. Thus, if we do not add v_{j+1} to T, then $\text{cut}_{j+1}(T') = \text{cut}_j(T) + M$ intuitively. Note that we do not subtract P from $\text{cut}_j(T)$ because the edges counted in P are not edges crossing from $\{1,\ldots,j\} \cap T$ to $\{1,\ldots,j\} \setminus T$ in the first place. Moreover, $m'_j = m_j$ because it is the number of vertices indexed by $\{j+1,...,n\}$ and v_{j+1} still belongs to this set. Thus, if we choose not to include v_{j+1} in T', then:

$$C_j(T') = \text{cut}_j(T) + M + \frac{1}{2}m_j$$

If we choose to add v_{j+1} into the set T, then every edge that connects from v_{j+1} to vertices in $\{1,\ldots,j\} \setminus T$ will then be an edge crossing from $\{1,\ldots,j\} \cap T$ to $\{1,\ldots,j\} \setminus T$. Thus, if we add v_{j+1} to T, then $\text{cut}_j(T') = \text{cut}_j(T) + P$ intuitively. We do not subtract M for similar reasons as above. Moreover, $m'_j = m_j$ for the same reasons as above as well. Therefore, if we include v_{j+1} in T', then

$$C_j(T') = \text{cut}_j(T) + P + \frac{1}{2}m_j$$

Note that to maximize $C_j(T')$, we must pick the larger of P or M since the remaining elements of the expression for both cases is the same. Thus, since our algorithm chooses to include v_{j+1} in the set T' if and only if $P \geq M$, then it follows that we will always chose the option which maximizes $C_j(T')$. □

(c) **(10 points)** Use the above to give a deterministic $O(m+n)$ time algorithm which achieves a 2-approximation for MAX CUT. [You may assume that the input graph is given in adjacency list format.] You must clearly describe the algorithm and analyze its run time. Briefly (assuming previous parts) also argue that the algorithm achieves a 2-approximation.

Proof. Let us define a deterministic $O(m+n)$ algorithm which achieves a 2-approximation for MAX CUT as follows:

i. Input: Adjacency list of graph

ii. Initialize set T to be empty, j = 0, and set T' to be empty

iii. Initialize collection of sets E_j. Let E_j be the set of edges which our algorithm "considers" on the $j-th$ run of the inner loop.

iv. For every j from 1 to n

   A. Set T = T'

   B. Initialize P = 0, M = 0

   C. For vertex v_j, go to its place in the adjacency list. For every vertex u encountered in its adjacency list, if $u \in \{1,\ldots,j\} \setminus T$, increment P by one and add every edge (v,u) to E_j. If $u \in \{1,\ldots,j\} \cap T$, increment M by one and add every edge (v,u) to E_j.

   D. if $P \geq M$, then add vertex v_j to T'.

v. Return T'

Correctness: We will argue that our algorithm achieves a 2-approximation of the MAX CUT problem.

Let us denote S as the cut which our algorithm outputs at termination and let S' be the optimal max cut. Thus, S is the number of edges which crosses the final return value of T'. Then, consider our collection of sets E. From its definition, each set E_j is simply the set of edges connecting to $\{1,...,j\}$ which we look at in step j when determining whether or not to add v_j to T'. Importantly, since we only add an edge to E_j if it corresponds to a vertex already "covered" by a previous iteration of j, then it follows that the union of all E_j is the collection of all edges within the graph (we "build up" to the set E). Therefore:

$$E = \bigcup_{j=1,\ldots,n} E_j, |E| = \sum_{j=1,..,n} |E_j|$$

Let us now consider the subset of each E_j which are actually "added" to the max cut– that is, the subset of edges which cross the cut after either adding or not adding v_j. Let us denote this subset of edges for every E_j as E_{j_c}. Intuitively, since these edges in all of

the E_{j_c} do not "change" in the sense that they will always remain in the cut after they are added. Thus, if we start off with 0 edges in the cut, then:

$$S = \bigcup_{j=1,...,n} E_{j_c}, |S| = \sum_{j=1,...,n} |E_{j_c}|$$

Finally, note that for all E_j, its corresponding E_{j_c} must have the property $|E_{j_c}| \geq \frac{1}{2}|E_j|$ since our algorithm would otherwise simply choose the other option which maximizes $C_j(T')$ (to either omit or include v_j). Therefore, since the maximum cut $|S'| \leq E$, we have

$$\sum_{j=1,...,n} |E_{j_c}| \geq \frac{1}{2} \sum_{j=1,..,n} |E_j|$$

$$|S| \geq \frac{1}{2}|E| \geq \frac{1}{2}|S'|$$

We have thus proved the lower bound of the approximation. The upper bound is trivial, as by definition of optimality, we know that the size of the optimal max cut must be greater than or equal to the size of the approximation. Thus,

$$|S| \leq |S'|$$

Therefore, we have proved our output is a valid 2 approximation:

$$\frac{1}{2}|S'| \leq |S| \leq |S'|$$

Runtime: Let $m = |E|$ denote the number of edges in the graph and let $n = |V|$ denote the number of vertices in the graph. The outer loop, which iterates through all v_j from $j = 1, ..., n$ loops $O(n)$ times. However, note that we only ever look at edges E_j at step j which connect to vertices already covered in iterations $i < j$ (since we consider only vertices which lie the subset from $\{1, ..., j\}$). Therefore, by inductive reasoning, we only ever consider each edge *once*, ie. checking if the other endpoint is in $\{1, \ldots, j\} \setminus T$ or $\{1, \ldots, j\} \cap T$. This is backed up by our earlier proof that $E = \bigcup_{j=1,...,n} E_j$ and $|E| = \sum j = 1, ..., n|E_j|$. Thus, since we do exactly two comparisons for each edge in the graph, this takes $O(2m)$ time. Thus, our algorithm takes a total of $O(n+2m) = O(n+m)$ time. □

3. Patients who require a kidney transplant but do not have a compatible donor can enter a kidney exchange. Usually, the demand for kidneys is far greater than the supply (in the US in 2010, more than 90,000 people were on the wait-list for a transplant, but only 15,000 kidneys were available). Thus, we must decide whom to allocate the kidneys to.

   The kidney donation problem has as input a compatibility graph $G(V, E)$, where each patient-donor pair is a vertex, and there is a directed edge $e = (u, v) \in E$ if the donor in u can donate to the patient in v. We wish to maximize the number of transplants.

(a) **(10 points)** Give a (polynomial-time) reduction [1] from the kidney donation problem to an integer linear program. (For this problem, donors are willing to donate a kidney regardless of whether their patient gets a kidney.)

Proof. Consider the following linear program which can be used to solve the kidney donation problem. Let us define variables $x_1, ..., x_m$ such that each variable x_i corresponds to a unique directed edge in the input compatibility graph. Every variable x_i, where i denotes the directed edge (u, v), can take on values of either 1 or 0, where 1 means that the donation from donor u to patient v should occur and 0 means that the donation should not. Our goal is to maximize the sum of these variables which is clearly equivalent to maximizing the total number of donations occur. We are subject to the following constraints:

$$\forall v \in V, \sum_{i \text{ s.t.} (i,v) \in E} x_{(i,v)} \leq 1$$

$$\forall u \in V, \sum_{i \text{ s.t.} (u,i) \in E} x_{(u,i)} \leq 1$$

Thus, we can express our linear program in canonical form to be solved by the simplex method as such:

$$\textbf{maximize} \sum_{i=(u,v) \in E} x_i$$

$$\textbf{subject to } \forall v \in V : \sum_{i \text{ s.t.} (i,v) \in E} x_{(i,v)} \leq 1$$

$$\forall u \in V : \sum_{i \text{ s.t.} (u,i) \in E} x_{(u,i)} \leq 1$$

$$\forall i \in E : x_i \geq 0$$

$$\forall i \in E : x_i \leq 1$$

Although we will prove correctness later, the logic for these constraints is that every donor can at most donate one kidney and every patient should receive only one kidney.

In order to reduce the the kidney donation problem to this integer linear program, we first iterate through all vertices in G and get all the outgoing edges from a given vertex, creating the constraints and variables:

$$\forall u \in V, \sum_{i \text{ s.t.} (u,i) \in E} x_{(u,i)} \leq 1$$

[1] Creating a reduction simply involves transforming the input of a problem A to the input of another problem B so that you can use B to solve A. In this case, you are transforming the input of the kidney donation problem to the input of an ILP. A polynomial-time reduction means that the process of transforming the input is polynomial-time, not that the entire process of solving problem A is polynomial-time. For more details, please see Section 10.

Then, we compute G^R, the reverse of G, and using the outgoing edges from every vertex in G^R, we create the constraints and variables:

$$\forall u \in V, \sum_{i \text{ s.t.}(u,i) \in E} x_{(u,i)} \leq 1$$

because an outgoing edge in G^R is the same as an incoming edge in G.

Correctness: To prove the correctness of this algorithm, we will prove that the reduction to the linear program accurately captures the kidney problem. First, we will define the actual constraints of the real life kidney problem. Evidently, we want to maximize the total number of donations that occur. In addition, no patient should receive more than 1 kidney and no donor should give more than 1 kidney (since the donor would die, and the extra kidney would be wasted on a patient). We will show that our linear constraints and objective function accurately reflect these real world constraints, which we have partially explained in the algorithm description.

In the input graph, every edge represents a possible donation. Therefore, since we define a variable x_i for all $i = (u, v) \in E$, every variable x_i represents a donation that *should* occur. In addition, since the variables x_i take on values from 0 to 1, it follows naturally that our objective function **maximize** $\sum_{i=(u,v)\in E} x_i$ maximizes the total number of donations which occur.

Then, consider our constraints $\forall v \in V$, $\sum_{i \text{ s.t.}(i,v)\in E} x_{(i,v)} \leq 1$ and $\forall u \in V$, $\sum_{i \text{ s.t.}(u,i)\in E} x_{(u,i)} \leq 1$. In the input graph, the set of all edges i s.t.$(i, v) \in E$ represents all the possible donations which a given patient v can receive. Thus, if we constrain the sum of the variables x_i which represent incoming donations to patient v to *at most* one, then we guarantee that each patient can not receive more than one donation of a kidney. Similarly, the set of all edges i s.t.$(u, i) \in E$ represents all the possible donations that a given donor u can receive. Thus, if we constrain the sum of the variables x_i which represent outgoing donations from a donor u to *at most* 1, then we guarantee that each donor can not donate more than one of their kidneys. Therefore, our constraints satisfy the real life constraints of the kidney problem.

Runtime: Let $m = |E|$ denote the number of edges in the input graph and $n = |V|$ denote the number of patient donor pairs in the input graph. First, to create the variables x_i for all $i = (u, v) \in E$ and write out our objective function, we must iterate through all edges of the graph, which will take $O(m)$ time.

Then, to create the outgoing edge constraint (no donor can donate more than 1 kidney) for the integer linear program, we must iterate through all vertices in the adjacency list form of G and sum together the x_i's which correspond to the outgoing edges of a given vertex, taking $O(n + m)$ time. Next, to create the incoming edge constraint (no patient should receive more than one kidney), we must first construct G^R which takes $O(n + m)$ time. Then, we must iterate through all vertices in the

adjacency list form of G^R and sum together the x_i's which correspond to the outgoing edges of a given vertex in G^R, again taking $O(n+m)$ time. This will naturally get us the sum of the x_i's which correspond to the ingoing edges of the same vertex in G, which we can then use to create our constraint.

Thus, the reduction takes a total runtime of $O(m+3(n+m)) = O(n+m)$ time. □

(b) **(5 points)** Suppose that each donor is only willing to donate a kidney if their corresponding patient gets a kidney (so donations form a set of cycles in the graph). Give a (polynomial-time) reduction from this restricted kidney donation problem to an integer linear program. For your solution, it suffices to specify what needs to be changed in the integer linear program from the previous part.

Proof. Consider the same algorithm outlined in part (a) but with a slight modification to the constraints which we will describe below. While we keep all the constraints we originally created, we add the extra constraint as outlined below:

$$\forall v \in V, \sum_{i \text{ s.t.}(i,v)\in E} x_{(i,v)} = \sum_{i \text{ s.t.}(u,i)\in E} x_{(u,i)}$$

Thus, we can express our linear program in canonical form to be solved by the simplex method as such:

$$\textbf{maximize} \sum_{i=(u,v)\in E} x_i$$

$$\textbf{subject to } \forall v \in V, \sum_{i \text{ s.t.}(i,v)\in E} x_{(i,v)} \leq 1$$

$$\forall u \in V, \sum_{i \text{ s.t.}(u,i)\in E} x_{(u,i)} \leq 1$$

$$\forall v \in V, \sum_{i \text{ s.t.}(i,v)\in E} x_{(i,v)} = \sum_{i \text{ s.t.}(u,i)\in E} x_{(u,i)}$$

$$x_i \in \{0,1\} \text{ for all } i$$

Correctness: To prove the correctness of this modified algorithm, we appeal to the correctness of the algorithm we proved in part (a). Thus, we will now prove the correctness of the additional constraint which we added to the algorithm in part (a). We will show that this additional constraint enforces the condition that each donor can only donate a kidney if and only if their corresponding patient gets a kidney.

We define the new constraint as $\forall v \in V$, $\sum_{i \text{ s.t.}(i,v)\in E} x_{(i,v)} = \sum_{i \text{ s.t.}(u,i)\in E} x_{(u,i)}$. Recall that each x_i corresponds to a possible donation (edge in the input graph) and takes on the value 1 if the donation does occur or 0 if the donation does not. In addition, recall that the set of all edges i s.t.$(i,v) \in E$ represents all the possible donations which a given patient v can receive and the set of all edges i s.t.$(u,i) \in E$ represents all the possible donations that a given donor u can receive. Thus, setting the two

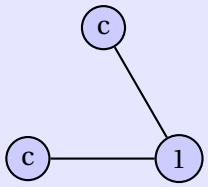
sums in the constraint equal to each other enforces that the number of donations a donor in the patient-donor pair gives is equal to the number of donations the patient in the same pair receives. Since we already have the constraints that the two sums in the condition are less than or equal to one based on the other constraints from part (a), we are essentially enforcing that either the donor gives 1 kidney and the patient receives 1 kidney, or the donor gives 0 kidneys and the patient also receives 0 kidneys. Therefore, this additional constraint enforces the condition that each donor only donates a kidney if and only if their corresponding patient also gets a kidney. Thus, we have shown the correctness of this modified program from part (a). □

(c) **(0 points, optional)** Algorithmic matches are not guaranteed to work in practice (due to, e.g., tissue-type incompatibility). Therefore, for each edge e, there is an associated probability f_e that the donation fails. If any of the donations in a cycle fails, the whole cycle fails and no transplants occur. To minimize the issue (and the logistical difficulty of having many transplants occurring at the same time and place), a hospital has decided to cap the length of kidney-donation cycles at 4. Again, each donor is only willing to donate a kidney if their corresponding patient gets a kidney. We wish to maximize the expected number of transplants that succeed. Give a (polynomial-time) reduction from this restricted kidney donation problem to an integer linear program.

(Hint: It may be helpful to calculate the set $C = C_1, C_2, \ldots$ of cycles in G of length at most 4. How big can the set C be?)

4. In the Weighted Vertex Cover problem the input is a graph $G = (V, E)$ with positive integer weights $w = \{w(u) | u \in V\}$ on the vertices of G. The goal is to output a vertex cover of minimum total weight given the graph G and weights w. I.e., the output should be a vertex cover C and among all vertex covers it should output a cover of minimum total weight.

(a) **(5 points)** For every $c > 1$, give an example of a weighted graph G with weights w such that the 2-approximation algorithm from the lecture of the (unweighted) Vertex Cover problem on G does not output a c-approximation to the weighted Vertex cover on G and w.



Proof. Consider the graph above with weights as labeled. The optimal vertex cover is just the middle vertex connected to 2 edges with a size of 1. However, the 2 approximation max vertex cover algorithm from lecture notes 19 would choose one of the two edges and add both of its vertex endpoints, throwing out the third vertex from the vertex cover approximation. No matter which edge this algorithm picks, it will always pick an edge connected to the middle vertex with size 1 and one of the other 2 vertices with size c.

Thus, the weight of the approximation of the vertex cover will always be $1 + c$. Since the optimal weighted vertex cover has weight 1, then the 2-approximation algorithm from lecture 19 gives a weighted vertex cover which is a factor of $1 + c$ away from the optimal. Since $1 + c > c$, then we have proved that the 2-approximation weighted vertex cover on this graph is not a c approximation to the weighted Vertex cover on G and w. □

(b) **(10 points)** Give a reduction from the weighted Vertex Cover problem to an Integer Linear Program. I.e., describe an algorithm that takes G and w as input and outputs an integer program whose value is exactly the minimum vertex cover size in G and w. You don't need to solve the integer linear program. (Hint: the following question will be easier if your solution to this has one variable per vertex and no other variables.)

Proof. Consider the following reduction from the weighted Vertex Cover problem to an Integer Linear Program.

Define indicator variables x_v for all $v \in V$ that take on values of either 0 or 1, where $x_v = 1$ if and only if $v \in V$ is in the minimum weighted vertex cover.

Our objective function is:

$$\textbf{minimize} \sum_{v \in V} w(v) * x_v$$

Subject to the constraints:

$$\forall (u, v) \in E : x_u + x_v \geq 1$$

$$\forall v \in V : x_v \geq 0$$

$$\forall v \in V : x_v \leq 1$$

Correctness: We will show that our objective function and constraints accurately capture the minimum weighted Vertex Cover problem.

Note that every variable x_v can only take on integer values from 0 to 1 from the second and third constraints. Thus, each variable can only take on either 1 or 0. Therefore, every variable x_v indicates whether vertex v should be included in the minimum weighted vertex cover of G.

Then, our first constraint $\forall (u, v) \in E : x_u + x_v \geq 1$ enforces the condition that the outputted solution is indeed a valid vertex cover in the sense that no edge is left uncovered. We enforce this by making sure every edge (u, v) in the graph has the sum of the variables corresponding to its endpoints greater or equal to 1. Note that this means $x_u = 1$ and/or $x_v = 1$. Equivalently, this makes sure that *at least* one of its endpoints u or v is included in the minimum weighted vertex cover, thus making sure every edge is covered by the vertex cover.

Finally, our objective function aims to minimize the sum $\sum_{v \in V} w(v) * x_v$. Recall that variables x_v take on values of 1 if they are included in the cover and 0 if they are excluded. Therefore, this sum is just the sum of all weights of vertices which we include in our weighted cover since all weights of vertices not included in the cover are multiplied by 0. Thus, minimizing this sum accurately will minimize the total weight of the weighted vertex cover, which is what we want.

Thus, the constraints and objective function of our integer linear program accurately reflect the goals and specifications of the minimum weighted vertex cover problem.

Runtime: Let $n = |V|$ denote the number of vertices in G and $m = |E|$ denote the number of edges in G. We also assume that getting the weight of any vertex through $w(v)$ takes O(1) time. In order to create our variables and write out our objective function, we create a variable x_v corresponding to every vertex in G, summing the products of these variables with their respective vertex weights. We also write out constraints 2 and 3 here, which also take $O(1)$ time because they are simple comparisons. Since we iterate through all $v \; in V$ to do this and perform only constant time operations per iteration, this step takes $O(n)$ time.

Next, to create constraint 1, we must iterate through all edges $(u, v) \in E$ to create the constraint $x_u + x_v \geq 1$ for each edge. Since this sum and comparison take $O(1)$ time, then the runtime of this is the runtime of iterating through all edges which is $O(m)$.

Therefore our total runtime is $O(n+m)$ runtime. □

(c) **(15 points)** Give a deterministic polynomial time 2-approximation to the Weighted Vertex Cover problem. (Hint: You may use the fact that LPs are solvable in polynomial time to solve the LP relaxation of the integer program from Part (b), then try to turn the real-valued solution into one that corresponds to an actual vertex cover.)

Proof. We transform our reduction to an integer linear program in part (b) to a reduction to a LP in the following way: we keep all the same constraints of the integer linear program which we defined. However, we make the clarification that every variable x_v does not need to take on integer values. Thus, the objective function for our LP program is:

$$\textbf{minimize} \sum_{v \in V} w(v) * x_v$$

Subject to the constraints:

$$\forall (u, v) \in E : x_u + x_v \geq 1$$

$$\forall v \in V : x_v \geq 0$$

$$\forall v \in V : x_v \leq 1$$

After solving this linear program, we iterate through all variables x_v and round the outputted solution to the nearest integer (which is either 0 or 1). As in part (b), if a variable $x_v = 1$ at the end after rounding, v is included in the approximation of the weighted vertex cover. If $x_v = 0$ after rounding, then v is excluded.

Correctness: We will now prove that our algorithm outputs a 2-approximation to the weighted vertex cover problem and this cover is also a valid vertex covering.

First, we will prove that the rounding of the solution maintains the property that the resulting vertex cover output will indeed cover all edges (prove that the output is indeed a valid vertex covering).

The constraints $\forall v \in V : x_v \geq 0$ and $\forall v \in V : x_v \leq 1$ enforce the fact that after rounding, our variables can only have values of 0 or 1. Thus, we can assert that for any variable with $x_v = 1$, $v \in V$ is in the approximation of the minimum weighted vertex cover.

Next, consider our constraint which we proved enforces this property in part (a): $\forall (u, v) \in E : x_u + x_v \geq 1$. In our modified linear program which now allows x_v to be non-integer, this constraint will now only be true if and only if x_v OR x_u is greater or equal to 0.5. To prove this, let us assume, for the sake of contradiction, that both $x_v, x_u < 0.5$ and $x_u + x_v \geq 1$. Immediately, we arrive at a contradiction since the sum of two numbers less than 0.5 can never be greater than double of 0.5. Therefore, since at least one of x_v or x_u is greater or equal to 0.5, then at least one of them will be rounded to 1 at the end of our algorithm and included in the vertex cover. Thus, for every edge $(u, v) \in E$, the constraint $x_u + x_v \geq 1$ still enforces the property that this edge will be covered in the minimum weighted vertex cover for our linear program.

Now, we will prove that the outputted solution after rounding is a 2-approximation of the optimal minimum weighted vertex cover. Let us define U' as the optimal minimum weighted vertex cover. Let U be the outputted solution of our linear program after rounding and the transformation back into the actual solution (where we iterate through all vertices checking if $x_v = 1$). We wish to show that the total weight of the vertices in U is within a factor of 2 from the total weight of the vertices in U'. Note that we express the total weight of the vertices in U as:

$$\sum_{v \in U} w(v) = \sum_{v \in V} w(v) * \text{ round}(x_v)$$

Note that we do not need to multiply by a factor of x_v in the first sum since we are only considering vertices in U already. Then, we have:

$$\sum_{v \in V} w(v) * \text{ round}(x_v) \leq \sum_{v \in V} w(v) * (2 * x_v)$$

This follows from the fact that round$(x_v) \leq 2 * x_v$, which we will now explain. We consider two scenarios: if $x_v < 0.5$ or if $x_v \geq 0.5$. If $x_v < 0.5$, then round$(x_v) = 0$

and $2 * x_v \geq \text{round}(x_v) = 0$ since we also have the constraint $x_v \geq 0$. If $x_v \geq 0.5$, then $\text{round}(x_v) = 1$ and $2 * x_v \geq \text{round}(x_v)$ since $x_v \geq 0.5$ implies $2x_v \geq 1$.

Then, note that the sum $\sum_{v \in V} w(v) * (x_v)$ is the weight of the linear program outputted before the rounding. Moreover, $\sum_{v \in U'} w(v)$ represents the optimal minimum weight vertex cover which is equivalent to the weight of the solution outputted by the integer linear model (integer is important!). The solution/weight of the LP version of the program must be at least as good as the integer version of the program since the solution space of the LP version covers the integer version! Thus, since it is subject to one less constraint (the integer constraint) and thus has greater range of freedom, we know that the LP program solution will output a solution that is at least as small as the integer program solution. Therefore, we have

$$\sum_{v \in V} w(v) * (x_v) \leq \sum_{v \in U'} w(v)$$

Thus, combining all of these inequalities, we have:

$$\sum_{v \in V} w(v) * \text{ round}(x_v) \leq \sum_{v \in V} w(v) * (2x_v) \leq 2 \sum_{v \in U'} w(v)$$

Thus, we have shown that the upper bound of our approximation is two times the the optimal weight solution (which we found using the integer linear program). Next, we will show that:

$$0.5 \sum_{v \in U'} w(v) \leq \sum_{v \in V} w(v) * \text{ round}(x_v)$$

This is trivially proved since we know that $\sum_{v \in U'} w(v)$ is the smallest weight we can achieve for the vertex cover by definition of optimality. Therefore, we know:

$$\sum_{v \in U'} w(v) \leq \sum_{v \in V} w(v) * \text{ round}(x_v)$$

Therefore, the following inequality is true:

$$\sum_{v \in U'} w(v) \leq \sum_{v \in V} w(v) * \text{ round}(x_v) \leq 2 \sum_{v \in U'} w(v)$$

Thus, we have proved that the approximation of the vertex cover given by our algorithm is indeed a 2-approximation of the optimal.

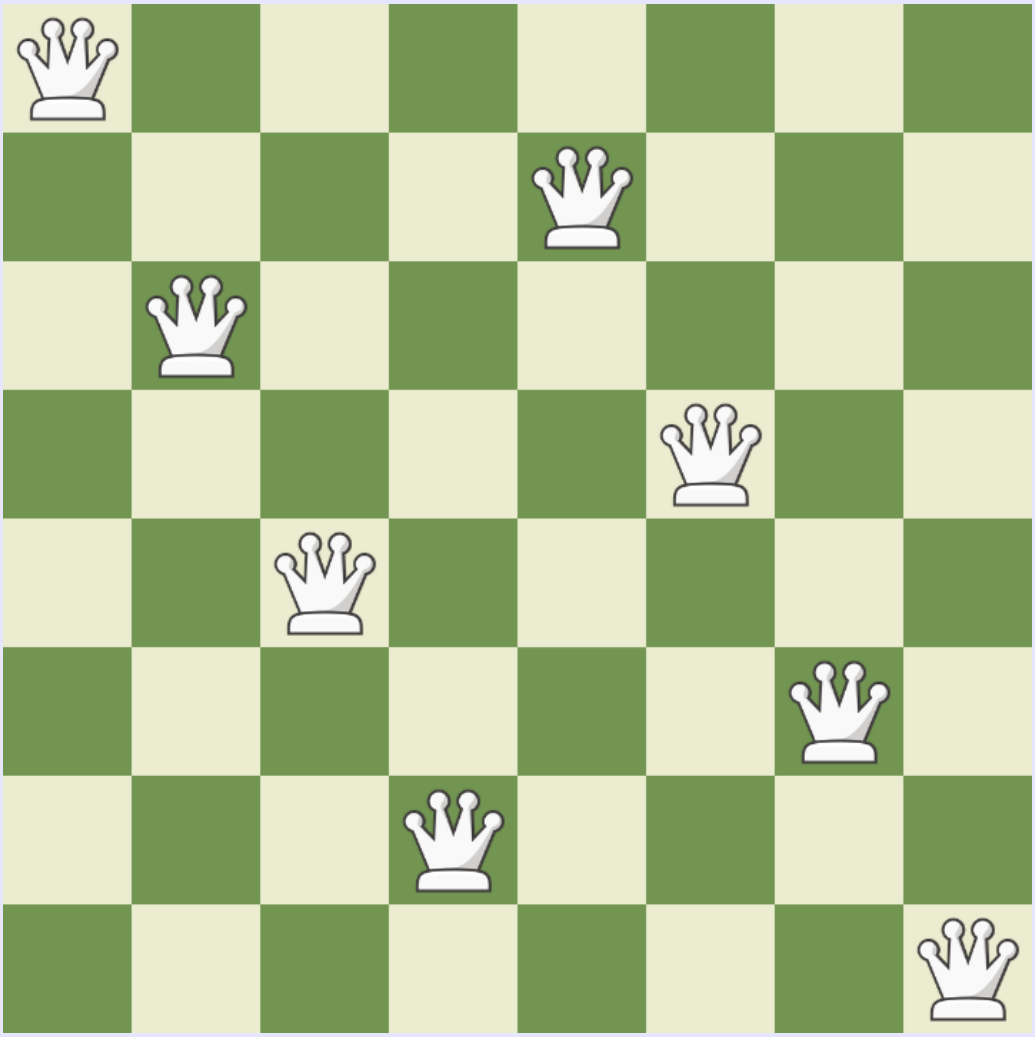
Runtime: From part (b), we proved that the runtime of the reduction to an integer linear program is $O(n+m)$ time. Since we are essentially writing out the same objective function, creating the same variables, and same constraints, we know the runtime of our reduction to a linear program is also $O(n+m)$ time. Then, by the problem description, we can solve the linear program in polynomial time. Finally, to convert our solution back to the actual form of the solutions, we simply iterate through all x_v and check if $x_v = 1$. If so, we add v to the vertex cover approximation. Since each variable corresponds to one vertex, this takes $O(n)$ time. Therefore, since every step of our algorithm takes polynomial time, then our entire algorithm takes polynomial time. □

5. Suppose we want to place n queens on an $n \times n$ chessboard such that no pair of queens attacks each other.[2] In this problem, we will explore the behavior of a local search algorithm for this problem. The algorithm is as follows:

   - Define an integer parameter $\tau \geq 0$.
   - For each column from 1 to n, place exactly one queen in a randomly chosen row.
   - Repeat the following:
     - Consider all $n(n-1)$ possible moves in which exactly one queen is moved to a different row within its respective column.
     - (A) If there is at least one move which strictly decreases the number of attacking pairs, pick the move which decreases this number the most. If the move results in zero attacking pairs, break out of the loop.
     - (B) If no move strictly decreases the number of attacking pairs, this is called a *local optimum*. In this case, if there is at least one move which achieves the same number of attacking pairs, then choose such a move uniformly at random. This is called a *sideways move*. Only τ many sideways moves are allowed, after which one must break out of the loop.
     - (C) Otherwise, if every move achieves a strictly greater number of attacking pairs, break out of the loop.
   - If the current configuration achieves zero attacking pairs, return SUCCESS. Otherwise, return FAIL.

   (a) **(5 points)** For $n = 8$, give an example of a placement of 8 queens, one per column, which is a local optimum (as defined in Step (B) above) where at least one pair of queens is attacking each other. *(Hint: you may find it helpful to use your code for the next question to find such a placement)*

[2] Recall that in chess, a queen is said to "attack" a position on the board if it can get to that position by moving in a straight line horizontally, vertically, or diagonally on the board

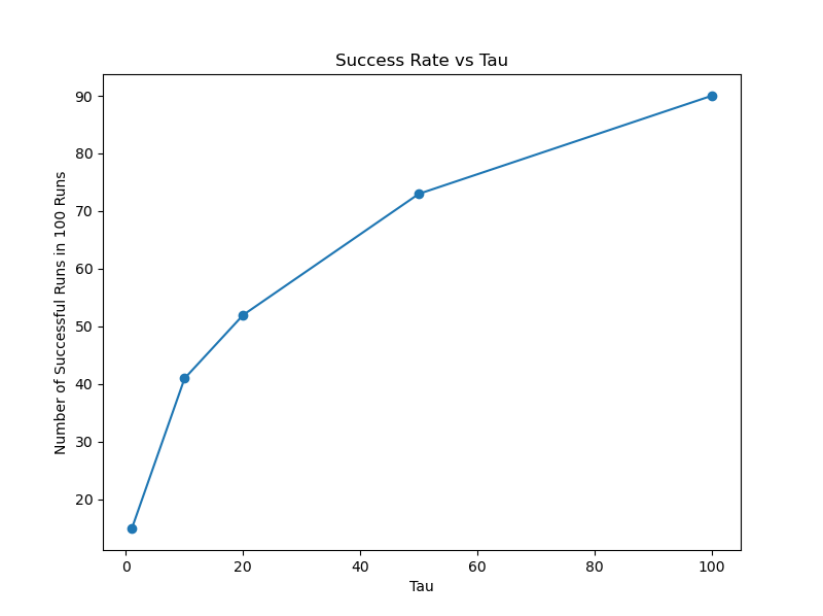


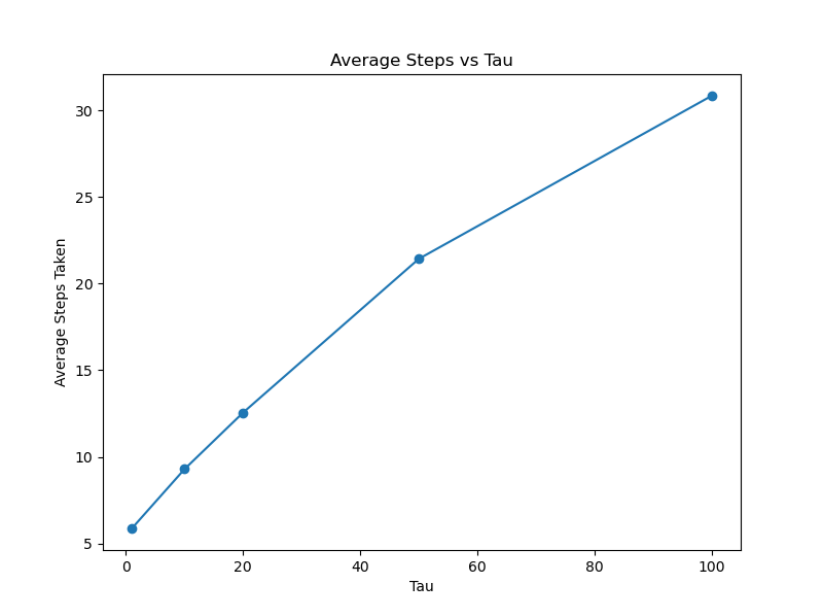
No queen can move to any one of the $n-1$ other rows in its column without attacking another queen in its new position. The two queens which are attacking each other are the queens at A8 and H1. However, if we try to move either of these queens to any other row, this will cause another pair of queens to attack each other since there is exactly 1 queen in every row of the board. Since none of the $n(n-1)$ moves strictly decreases the number of attacking pairs, this is a local optimum.

Required format for solution: In your solution, please attach an image of the board position generated as follows. Go to `https://www.chess.com/analysis?tab=analysis`, press "+ Set Up Position," press the trash can icon on the right, and drag eight white queens onto the board in a configuration of your choosing. Then press the share icon (bottom right), press "Image," and download and attach the JPEG file to your assignment.

(b) **(15 points)** Implement the above algorithm for $n = 10$. For each $\tau \in [1, 10, 20, 50, 100]$, run the

algorithm 100 times and generate the following two plots. The x-axis for both plots is τ. The y-axis for the first plot is the number of runs in which the algorithm returns SUCCESS. The y-axis for the second plot is the average number of steps taken by the algorithm among all trials that resulted in SUCCESS. Give a short explanation of what you see in these two plots.





In these plots, we can see that the success rate of the algorithm grows logarithmically since each increase in τ yields smaller and smaller increases in the number of suc-

cesses found. This diminishinig return makes sense because at a certain point, additional alloted steps to walk out of a local minimum only help in scenarios where walking along the local minimum takes many many steps, which is rare. However, the growth of the success rate in relation to τ makes since because we allow our algorithm more freedom to explore along the local minimum to potentially find a better solution. In addition, we see that the average number of steps increases roughly linearly with τ. This makes sense because everytime we increase τ, we are essentially giving our algorithm a greater degree of freedom in which it can step along local minima to find better solutions, rather than cutting the algorithm off soon after it finds a local minima.