CS 124 Homework 7: Spring 2024

Anusha Murali

Collaborators: None

No. of late days used on previous psets: 5
No. of late days used after including this pset: 6

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.

Solution: Let C^* and S^* are a minimum vertex cover and a maximum independent set of a graph $G = (V, E)$. Let $n = |V|$. Therefore, $|C^*| + |S^*| = n$ or $|S^*| = n - |C^*|$.

Let C be a 2-approximation vertex cover (Lemma 2 of lecture 19), so $|C| \leq 2|C^*|$.

Let S be the corresponding independent set of C. Hence $|S| = n - |C| \geq n - 2|C^*|$. But, we also know that $|S| \leq |S^*|$, so $|S| \leq n - |C^*|$.

We want to show that S is not within a factor of a constant $c > 1$ of its corresponding 2-approximation minimum vertex cover C.

Let us claim that $|S| = c|S^*|$. So, we have,

$$\begin{aligned}
n - 2|C^*| &\leq |S| \leq n - |C^*| \\
\frac{n - 2|C^*|}{n - |C^*|} &\leq \frac{|S|}{n - |C^*|} \leq \frac{n - |C^*|}{n - |C^*|} \\
\left(1 - \frac{|C^*|}{n - |C^*|}\right) &\leq \frac{|S|}{|S^*|} \leq 1 \\
1 - \epsilon &\leq c \leq 1
\end{aligned}$$

Since $\epsilon \geq 0$, the LHS of the above inequality is < 1, so we find that $c \leq 1$.

Hence 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, where $c > 1$. □

(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?)

Solution As suggested in the hint, let us consider a starting graph $G = (V, E)$, and consider the graph $G' = G \times G = (V', E')$, 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 $\{u, w\} \in E$ or $\{v, x\} \in E$.

Let $s(G)$ be the size of the maximum independent set in G. We have the following claim.

Claim: For every graph $G = (V, E)$, the size of the maximum independent set of G' is the square of the size of the maximum independent set of G. In other words, $s(G') = s(G \times G) = (s(G))^2$.

Proof. Let $S \subseteq V$ be the maximum independent set in G. Hence, the size of the maximum independent set is $|S| = s(G)$.

From the property of an independent set, there are no edges between the vertices in S. Hence there are no edges between the vertices in $S \times S$, which implies that $S \times S$ is an independent set in $G \times G$. Therefore, we find,

$$s(G') = s(G \times G) \geq |S \times S| = (s(G))^2. \tag{1}$$

Let $S' \subseteq V'$ be the maximum independent set in $G' = G \times G$. Note that $V' = V \times V$. Since $S' \subseteq V'$, we know that every vertex in S' consists of a pair of vertices (u, v), each corresponds to a vertex in G. Therefore, if we project the first coordinate of (u, v) onto x-axis and the second coordinate onto y-axis for all such pairs of vertices $(u, v) \in S'$, we obtain the independent sets S'_x and S'_y in G respectively. Therefore, we have,

$$s(G') = s(G \times G) = |S'| \leq |S'_x \times S'_y| \leq (s(G))^2. \tag{2}$$

Therefore, from Equations 1 and 2, we find,

$$s(G') = s(G \times G) = (s(G))^2.$$

Therefore, the size of the maximum independent set of G' is the square of the size of the maximum independent set of G as claimed. □

Now assume that we have a 2-approximate algorithm for the maximum independent set problem. Let us apply this 2-approximate algorithm to the graph $G' = G \times G$. From our claim above, the resulting algorithm will be $2^{1/2}$-approximate for G, which can be written as $1 + \epsilon$, where $\epsilon = \sqrt{2} - 1$. □

We can extend the idea given in the hint to an arbitrary graph $G' = G \times G \times \cdots \times G = G^m$, where G' is obtained by taking the product of G with itself m times. Hence, generalizing the above claim, the Equation 1 becomes,

$$s(G') = s(G \times G \times \cdots \times G) \geq |S \times S \times \cdots \times S| = (s(G))^m. \tag{3}$$

Similarly, for the general case, Equation 2 becomes,

$$s(G') = s(G \times G \times \cdots \times G) = |S'| \leq |S'_1 \times S'_2 \times \cdot \times S'_m| \leq (s(G))^m, \tag{4}$$

where we have used the subscripts $1, 2, \cdots, m$ for the coordinates of the individual vertices of a vertex tuple in G'. Hence from Equations 3 and 4, we find,

$$s(G') = s(G \times G \times \cdots \times G) = (s(G))^m.$$

In other words, the size of the maximum independent set of $G' = G \times G \times \cdots \times G$ is the m-th power of the size of the maximum independent set of G.

Now, as stated in the problem, let us assume that there exists a 2-approximate polynomial algorithm for the maximum independent set problem. Let us apply this 2-approximate algorithm to the graph $G' = G \times G \times \cdots \times G = G^m$. From our results above, the resulting algorithm will be $2^{1/m}$-approximate for G, which can be written as $1+\epsilon$, where $\epsilon = \sqrt[m]{2} - 1$.

For example, if $m = 5$, $2^{1/m} \approx 1.1487$, so $\epsilon = 0.1487$. If $m = 1000$, $2^{1/m} \approx 1.00069$, so $\epsilon = 0.00069$.

Therefore, if there exists a 2-approximate polynomial algorithm for the maximum independent set problem, we find that for every $\epsilon > 0$, there is a $(1+\epsilon)$-approximate polynomial time algorithm for the maximum independent set problem. □

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 $0 \le j \le n-1$ 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 $\mathrm{cut}_j(T)$ 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 $\mathrm{cut}_j(T)$ and m_j.

   Solution: Let the set of vertices indexed by T by V_T, and let E_T be the set of all edges among the vertices in V_T. Let $E_{j'}$ be the set of all edges that have at least one vertex in $\{j+1, \ldots, n\}$. For $e \in E_{j'}$, let $\mathbb{1}_e$ be the indicator that is 1 when e lies in S, otherwise it is 0. Then

   $$\mathbb{E}[\mathbb{1}_e] = \mathbb{P}[\mathbb{1}_e = 1] = \frac{1}{2},$$

   because each vertex in $\{j+1, \ldots, n\}$ is chosen i.i.d. uniformly at random, and in either case where v is in V_T or $V_{[j]\setminus T]}$, the same probability will hold. In case when both the end points lie within $\{j+1, \ldots, n\}$, the analysis from the lecture on MAX-CUT will still hold and give the same expectation. Therefore,

   $$C_j(T) = \mathrm{cut}_j(T) + \sum_{e \in E_{j'}} \mathbb{E}[\mathbb{1}_e] = \mathrm{cut}_j(T) + \frac{m_j}{2},$$

   which is consistent with the case where $j = 0$. □

(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.

Solution: We use the following idea. At any given point, let S_T denote the set of vertices indexed by T, $S_{T'}$ denote the set of vertices indexed by $\{1,\ldots,j\} \setminus T$, and $S_{T''}$ denote the set of vertices indexed by $\{j+1,\ldots,n\}$. Next, for a vertex $v \in S_{T''}$, let $E_T(v)$ denote the set of all edges between S_T and v, and $E_{T'}(v)$ denote the set of all edges between $S_{T'}$ and v. Let $f_{T,T'} : V \to \mathbb{R}$, $g_{T,T'} : V \to \mathbb{R}$ be the following functions that we define for $v \in S_{T''}$.

$$f_{T,T'}(v) = \frac{|E_{T'}(v)| - |E_T(v)|}{2}$$
$$g_{T,T'}(v) = \frac{|E_T(v)| - |E_{T'}(v)|}{2}$$

For v_{j+1}, if $f_{T,T'}(v_{j+1}) \geq g_{T,T'}(v_{j+1})$, then we put v_{j+1} in T to create T', otherwise we put it in $\{1,\ldots,j\} \setminus T$.

The idea is that if we add v_{j+1} to T, $C_{j+1}(T') - C_j(T) = f_{T,T'}(v_{j+1})$, otherwise it is equal to $g_{T,T'}(v_{j+1})$ (can be verified via some basic algebra, which we omit for brevity). Whichever leads to a more positive change, we would want to execute that move in order to maximize the expectation. □

(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.

Solution: We use the following algorithm.

Let $T_1 \leftarrow \{v_1\}, T_2 \leftarrow \emptyset$
for $i \leftarrow 2,\ldots,n$ **do**
  Compute $f_{T_1,T_2}(v_i)$ and $g_{T_1,T_2}(v_i)$ as in Part (b)
  if $f_{T_1,T_2}(v_i) \geq g_{T_1,T_2}(v_I)$ **then**
    $T_1 \leftarrow T_1 \cup \{v_I\}$.
  else
    $T_2 \leftarrow T_2 \cup \{v_I\}$.
  end if
end for
return T_1.

The analysis for the runtime requires fixing data structures being used for T_1, T_2. The loop has n iterations, so we have a baseline of $O(n)$ running time for the algorithm. Next, computing f_{T_1,T_2} and g_{T_1,T_2} is the main computation to consider. Each edge in E only gets considered twice in the whole algorithm, e.g., for edge $e = (u,v)$, we consider it once while working with u and once while working with v. To compute $E_T(v)$, we have to just compute that function once in the entire run of the algorithm, and that involves just counting the number of incident edges on v, and the same holds while computing $E_T(v)$. If we think of T_1 and T_2 has hash maps, we can check whether a neighbor of v is in T_1 or T_2 in $O(1)$ time, and since we have

adjacency list representation, this process would take $O(|E(v)|)$ time, where $E(v)$ is the set of incident edges on v. Therefore, we have $O(1)$ computation associated with each edge in the graph, so the total running time comes out to be $O(m+n)$.

We claim that this algorithm finds a 2-approximation to the MAX-CUT of every graph. We claim that by the end of the iteration with some value of i, if the number of edges in the sub-graph formed by vertices $\{1,\ldots,i\}$ is m_i, then the size of the cut discovered by this algorithm is at least $\frac{m_i}{2}$. We can prove this via induction on i. We will sketch out this proof informally. In the base case where $i = 1$, the sub-graph formed by v_1 has no edges (assuming no self-loops), so the claim trivially holds because the cut so far has no edges either. The inductive step assumes that the claim holds for some $i = k$. For $i = k+1$, we are essentially considering which one of T_1 and T_2 has more neighbors with v_{k+1}, and just putting v_{k+1} in the set which has fewer neighbors with v_{k+1}. Just as an example,

$$f_{T_1,T_2}(v) \geq g_{T_1,T_2}(v) \iff |E_{T_1}(v)| \geq |E_{T_2}(v)|,$$

which implies that T_1 has fewer neighbors with v_{k+1}, so we put that vertex in T_1, and vice versa. This means that we have added at least $\frac{m_{k+1}}{2}$ edges to our cut. Therefore, the claim holds by induction. By the end of the final iteration, we have a cut having at least $\frac{m}{2}$ edges, hence, we have a 2-approximation to the MAX-CUT. □

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.)

   Solution: Let $G(V,E)$ be an input compatibility graph, where a vertex $u \in V$ denotes a patient-donor pair, (P_i, D_i) and edge $e = (u,v) \in E$ denotes that the donor in vertex u can donate to the patient in vertex v. Our goal is to maximize the number of transplants.

   Since we want to maximize the number of transplants, we would like to maximize the number of incident directed edges. In other words, given a directed graph $G(V,E)$, we are looking for the maximum subset of edges $E' \subseteq E$ such that every vertex $\in V$ has at most one directed edge incident from E', or equivalently, we would like to maximize the cardinality of the set of matching edges, E'.

   Let us define a variable x_{uv} such that $x_{uv} = 1$ if the donor of pair u donates to the patient of pair v and $x_{uv} = 0$ otherwise. So, we have,

$$x_{uv} = \begin{cases} 1 & \text{If the donor of pair } u \text{ donates to the patient of pair } v \\ 0 & \text{Otherwise.} \end{cases}$$

   Since a donor can only donate one kidney, we have the constraint that $\sum_{v:(u,v)\in E} x_{uv} \leq 1$.

   From our definition of the variable x_{uv} above, we also have the constraint that $x_{uv} \in \{0,1\}$, where $(u,v) \in E$.

   Since we want to maximize the number of transplants, our objective function that needs to be maximized is $\sum_{(u,v)\in E} x_{uv}$.

   Therefore, we can write out all of our constraints for our integer program as,

$$\max \sum_{(u,v)\in E} x_{uv} \tag{5}$$

$$x_{uv} \in \{0,1\} \qquad \forall (u,v) \in E \tag{6}$$

$$\sum_{v:(u,v)\in E} x_{uv} \leq 1 \qquad \forall v \in V \tag{7}$$

[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.

The above constraints specified in Equations 5, 6, and 7 transform our original graph problem into an Integer Programming problem. The constraint in Equation 5 uses one variable (i.e: x_{uv}) per edge, which therefore has a total of $|E|$ assignments. The constraint in Equation 6 assigns either 0 or 1 to the variable x_{uv} if the edge corresponds to a donor of pair u donating to the patient of pair v. This determination can be made in $O(|E|)$ time. The constraint in Equation 7 can be determined in $O(|V||E|)$ time as we need to check $|V|$ vertices each at most $|E|$ number of times. Therefore, the reduction from the kidney donation problem to an integer programming problem takes $O(|V||E|)$ or polynomial time.

Since integer programming is NP-hard, we can convert the above problem into a linear programming problem by letting the constraint $x_{uv} \in \{0,1\}$ (specified in Equation 6) to be an inequality as $0 \leq x_{uv} \leq 1$. Hence the constraints for our linear program are as follows:

$$\max \sum_{(u,v)\in E} x_{uv} \tag{8}$$

$$0 \leq x_{uv} \leq 1 \qquad \forall (u,v) \in E \tag{9}$$

$$\sum_{v:(u,v)\in E} x_{uv} \leq 1 \qquad \forall v \in V \tag{10}$$

Solving the integer linear program will give us the values of all x_{uv}, where $(u,v) \in E$. If $x_{uv} = 1$ for a given (u,v), then the donor in vertex u will donate to the patient in vertex v. If $x_{uv} = 0$ for a given (u,v), then the donor in vertex u cannot donate to the patient in vertex v. Using these assignments, we can maximize the number of kidney transplantations. □

(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.

Solution: Given that each donor is only willing donate a kidney if their corresponding patient gets a kidney, we have the following additional constraint:

$$\sum_{u:(u,v)\in E} x_{uv} = \sum_{v:(v,u)\in E} x_{vu}.$$

The above constraint ensures that the patient of pair u receives a kidney if and only if that the donor of pair u donates a kidney. Strictly speaking, this summation constraint ensures that the number of donor kidneys a patient-donor pair donates is the same as the number of donor kidneys it receives. Therefore, our kidney donation problem has been reduced to the following integer linear program:

$$\max \sum_{(u,v)\in E} x_{uv} \tag{11}$$

$$\sum_{u:(u,v)\in E} x_{uv} = \sum_{v:(v,u)\in E} x_{vu} \qquad \forall v \in V \tag{12}$$

$$0 \leq x_{uv} \leq 1 \qquad \forall (u,v) \in E \tag{13}$$

$$\sum_{v:(u,v)\in E} x_{uv} \leq 1 \qquad \forall v \in V \tag{14}$$

The above constraint uses check on the sum of the variables x_{uv} for all (u, v), therefore has $O(|E|)$ time requirement. Hence the entire reduction process is again dominated by $O(|V||E|)$ or polynomial time.

As in Part (a), solving the integer linear program will give us the values of all x_{uv}, where $(u, v) \in E$. If $x_{uv} = 1$ for a given (u, v), then the donor in vertex u will donate to the patient in vertex v. If $x_{uv} = 0$ for a given (u, v), then the donor in vertex u cannot donate to the patient in vertex v. Using these assignments, we can maximize the number of kidney transplantations. □

(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.

   Solution: For every $c > 1$, define the graph $G = (V, E)$ as follows. We can assume that the graph has $2n$ (where $n \geq 1$) vertices. We will construct a disconnected bipartite graph. Let $V = \bigcup_{i=1}^{n} \{u_i, v_i\}$, such that $E = \bigcup_{i=1}^{n} \{(u_i, v_i)\}$. Put weight 1 on all u_i's and weight $\lceil c \rceil$ on all v_i's. The algorithm from the lecture will output V as the vertex cover, whose weight would be $n(1+\lceil c \rceil)$, which is strictly greater than cn (the upper bound on the weight of a c-approximation to the minimum vertex cover $\{u_1, \ldots, u_n\}$, which is n).

   (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.)

   Solution: For a graph $G = (V, E)$ define a variable $x_v \in \{0, 1\}$ for each $v \in V$, such that $x_v = 1$ if v is chosen in the vertex cover, otherwise it is 0. Let w_v be the weight for a vertex $v \in V$. Our constraint set is as follows.

$$\forall v \in V,\ x_v \in \{0, 1\} \tag{15}$$

$$\forall e = (u, v) \in E,\ x_u + x_v \geq 1 \tag{16}$$

   Our goal is to minimize the following function.

$$f(V) = \sum_{v \in V} w_v x_v$$

   We will show that our constraint set to the integer LP accurately represents the problem instance, i.e., an assignment of all x_v's satisfies (16) if and only if the corresponding selection of the v's for the output set of vertices is a vertex cover.

   We first show that an assignment of x_v's that satisfies (16) results in a vertex cover. Suppose we have an assignment of x_v's that satisfies (16). This means that for each edge $e = (u, v) \in E$, $x_u + x_v \geq 1$, which implies that at least one of x_u or x_v must be assigned the value 1, which means that the corresponding vertex (or vertices) must be there in the chosen set of vertices (by construction). This implies that e is covered by that set. Since this is true for all the edges in E, the resulting choice of vertices must be a vertex cover.

   Next, we show that an assignment of x_v's that does not satisfy (16) does not result in a vertex cover. Suppose there is an edge $e = (u, v) \in E$ that does not satisfy (16).Then it must be the

case that $x_u = x_v = 0$. However, by our construction, it means that neither u nor v was chosen in the output set, which implies that e was not covered by that set. This shows that the chosen set is not a vertex cover.

The running time of this reduction is just linear in n and m, i.e., $O(n+m)$. It just needs to create and store the n variables, and create the m constraints. The actual running time of the solver of this integer LP is known to be exponential. □

(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.)

Solution: We relax the above integer linear program to a regular linear program by modifying (15) as follows.

$$\forall v \in V,\ x_v \in [0,1]$$

We first solve this relaxed LP. Next, in the solution, if $x_v \geq \frac{1}{2}$, we set it to 1, otherwise we set it to 0. We return this final, rounded assignment. This will give us a 2-approximation in polynomial time.

Running time. The running time is polynomial in n because regular linear programs can be solved in time that is polynomial in n.

Obtaining a vertex cover. This follows because in this relaxed LP, (16) is always satisfied. When an edge constraint is satisfied for $e = (u,v) \in E$, it means that at least one of x_u or x_v must have a value of at least $\frac{1}{2}$, which we would round up to 1, which would still satisfy that constraint. Since we know from Part B that any assignment satisfying all the edge constraints would give us a vertex cover, this rounding scheme will give us a vertex cover, as well.

Obtaining a 2-approximation. The argument is as follows. Fix an optimal solution to an instance of this problem, and for all $v \in V$, let x_v^* denote the optimal assignment in terms of the integer linear program for that instance corresponding to this solution. Let the solution of this relaxed linear program be denoted by $\bar{x}_v$ for all $v \in V$, and let $\hat{x}_v$ be the rounded solution. Note that because of the nature of this rounding scheme, $\hat{x}_v \leq 2\bar{x}_v$ for all $v \in V$. Also, because the constraint set of the relaxed LP covers the constraint set of the integer LP (in the sense that each variable can take more values in the relaxed LP than in the integer LP), the solution to the relaxed LP would result in a weighted sum that is at most as large as the weighted sum from the solution to the integer LP (i.e., the optimal true solution to the weighted vertex cover problem). Then we have the following.

$$\begin{aligned}\sum_{v\in V} w_v \hat{x}_v &\leq \sum_{v\in V} 2w_v \bar{x}_v \\ &= 2\sum_{v\in V} w_v \bar{x}_v \\ &\leq 2\sum_{v\in V} w_v x_v^*\end{aligned}$$

Therefore, this gives us a 2-approximation. □

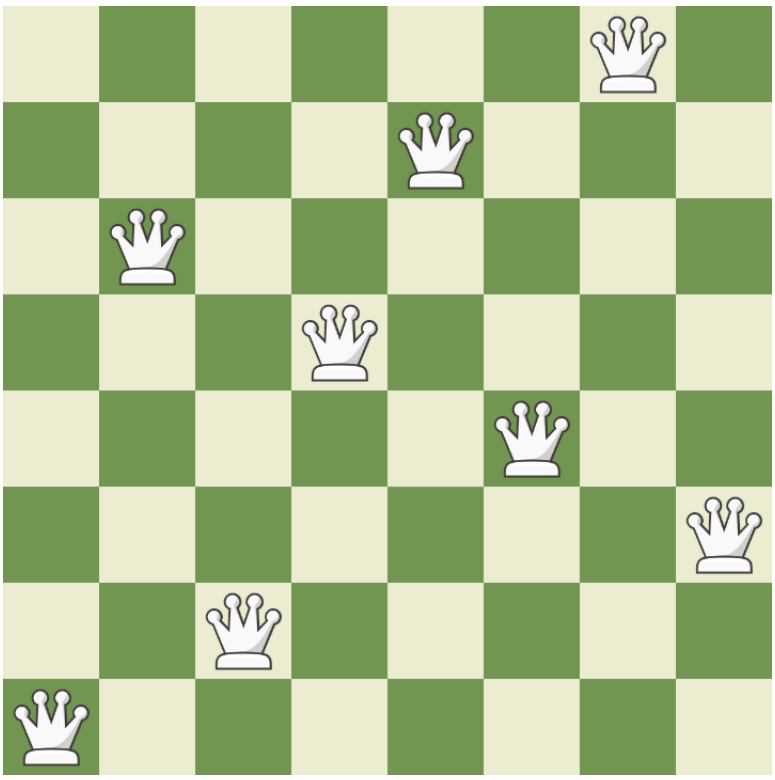
5. Suppose we want to place n queens on an $n \times n$ chessboard such that no pair of queens attacks each other.[2] In this problem, we will explore the behavior of a local search algorithm for this problem. The algorithm is as follows:

   - 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.

[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

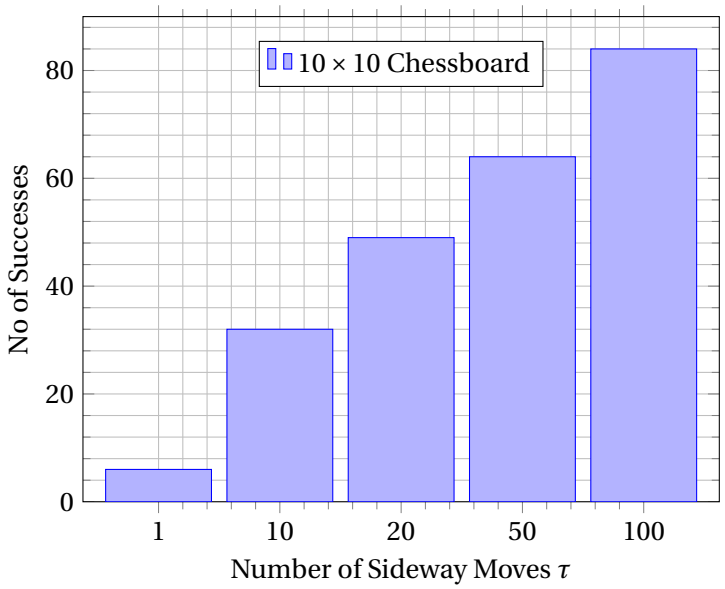
(a) **(5 points)** For $n = 8$, give an example of a placement of 8 queens, one per column, which is a local optimum (as defined in Step (B) above) where at least one pair of queens is attacking each other. *(Hint: you may find it helpful to use your code for the next question to find such a placement)*

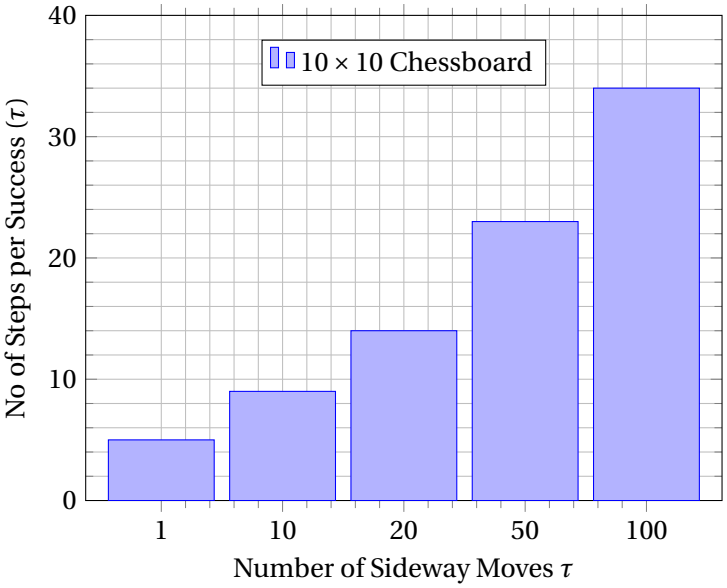
Solution: The cost of the current state in the following placement of the 8 queens is 1. This is because there is only one attacking pair (d5 and g8). There is no move of any of the 8 queens will strictly decrease the cost. This state is also not a global optimum due to the above one attacking pair. Therefore, the attached example 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.





I implemented the given $n \times n$ chessboard problem using the Hill Climbing algorithm, where I used τ to limit the number of sideways moves whenever the algorithm got stuck at a local maximum.

The major issue of using a Hill Climbing algorithm for solving an optimization problem is that it may get stuck at a local optimum. In fact there could be multiple local optima, and the algorithm can get stuck in any one of them. As indicated in step (B) of the algorithm in the problem statement, if no move strictly decreases the number of attacking pairs, it indicates that we are at a local optimum. In this case, if there is at least one move which achieves the same number of attacking pairs, then we can choose such a move uniformly at random. However, one problem with allowing sideway moves is that the algorithm may not terminate as we may be visiting the same states repeatedly. Therefore, we impose a limit on the number

of sideway moves. The first plot shows that as we increase the number of sideway moves from $\tau = 1$ to $\tau = 100$, the number of successes also increases.

With only 1 sideways move ($\tau = 1$), the first plot shows that the percentage of success over 100 runs is only about 6%, and the second plot shows that the Hill Climbing algorithm found each SUCCESS in 5 steps on average. On the other hand, when we increase the number of allowed sideways move to 100 ($\tau = 100$), the Hill Climbing algorithm is able to solve approximately 84% of the 100 runs. However, on the average, it took about 34 steps for it to find a SUCCESS. Therefore, we find that the larger the number of allowed sideway moves, the higher the number of successful solutions, as a larger τ increases the exploratory space for the sideway moves. □