1. Suppose you are given an $n \times n$ checkerboard with some of the squares deleted. You have a large set of dominos, just the right size to cover two squares of the checkerboard. Describe and analyze an algorithm to determine whether one tile the board with dominos—each domino must cover exactly two undeleted squares, and each undeleted square must be covered by exactly one domino.

   Your input is a boolean array $Deleted[1..n, 1..n]$, where $Deleted[i, j] = \textsc{True}$ if and only if the square in row i and column j has been deleted. Your output is a single boolean; you do ***not*** have to compute the actual placement of dominos.

Solution (Reduction to bipartite perfect matching): Given the array $Deleted[1..n, 1..n]$, we first construct a bipartite graph $G = (L \cup R, E)$ as follows:

- $L := \{(i, j) \mid Deleted[i, j] = \textsc{False} \text{ and } i + j \text{ is even}\}$ — Intuitively, L is the set of undeleted white squares.
- $R := \{(i, j) \mid Deleted[i, j] = \textsc{False} \text{ and } i + j \text{ is odd}\}$ — Intuitively, R is the set of undeleted black squares.
- $E := \{(i, j)(i', j') \mid (i, j) \in L \text{ and } (i', j') \in R \text{ and } |i - i'| + |j - j'| = 1\}$ — Intuitively, E is the set of all adjacent pairs of undeleted squares.

Every domino must cover one white square and one black square, so if $|L| \neq |R|$, we can immediately return False. Otherwise, we compute a maximum-cardinality matching in G; if this matching is perfect, we return True, and otherwise, we return False.

If we use the matching algorithm presented in class, our algorithm runs in $O(VE) = \boldsymbol{O(n^4)}$ ***time.*** ■

Solution (Reduction to maximum flow): Given the array $Deleted[1..n, 1..n]$, we first construct a flow network $G = (V, E)$ as follows:

- $V := \{s, t\} \cup \{(i, j) \mid Deleted[i, j] = \textsc{False}\}$.
- There are three types of edges:
  - $s \rightarrow (i, j)$ for all $(i, j) \in V$ where $i + j$ is even.
  - $(i, j) \rightarrow t$ for all $(i, j) \in V$ where $i + j$ is odd.
  - $(i, j) \rightarrow (i', j')$ for all $(i, j), (i', j') \in V$ where $i + j$ is even and $|i - i'| + |j - j'| = 1$
  - Every edge in G has capacity 1.

  Intuitively, we have an edge from s to every white square, from every white square to every adjacent black square, and from every adjacent black square to t.

Now we compute a maximum (s, t)-flow in G. If the value of this flow is exactly half the number of undeleted squares, return True; otherwise, return False.

The maximum flow value is equal to the largest number of non-overlapping dominos we can place on the board, which is trivially at most $n^2/2$. Thus, if we use Ford-Fulkerson to compute the maximum flow, our algorithm runs in $O(n^2 E) = \boldsymbol{O(n^4)}$ ***time.*** If we use Orlin's algorithm to compute the maximum flow, our algorithm still runs in $O(VE) = \boldsymbol{O(n^4)}$ ***time.*** ■

Rubric (Standard rubric for graph reductions): 10 points:

- 4 for correct input transformation
  - + 2 for vertices (including capacities or other necessary data)
  - + 2 for edges (including directedness, capacities, costs, or other necessary data)
- 3 for problem and algorithm
  - + 2 for specifying the problem to be solved, including all necessary input parameters (for example, "maximum cardinality matching" or "integer maximum flow from s to t")
  - + 1 for identifying a correct black-box algorithm for that problem
- 2 for correct output transformation
- 1 for time analysis (in terms of the original input parameters, not just the flow network)

For each problem, we have a target running time in mind, although this may not be the fastest algorithm for the problem. Algorithms slower than the target running time are worth fewer points; faster algorithms are worth extra credit.

2. A ***k-orientation*** of an undirected graph G is an assignment of directions to the edges of G so that every vertex of G has at most k incoming edges.

   Describe and analyze an algorithm that determines the smallest value of k such that G has a k-orientation, given the undirected graph G input. Equivalently, your algorithm should find an orientation of the edges of G such that the maximum in-degree is as small as possible. For example, given the cube graph as input, your algorithm should return 2.

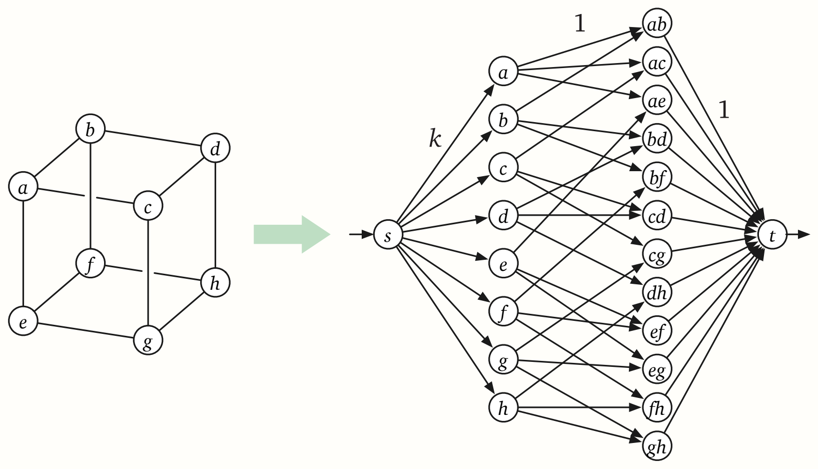
Solution: Our algorithm performs a binary search for the smallest k such that G has a k-orientation; for each value of k we consider, we intuitively look for an ***assignment*** of at most k incoming edges to each vertex. More concretely, we solve the decision problem as a generalized matching or pair-selection problem, where the two resource sets are the vertices and edges of G.

Fix an arbitrary value of k. To decide whether G has a k-orientation, we construct a flow network $H = (V', E')$ as follows:

- $V' = V \cup E \cup \{s, t\}$. Except for the source s and target t, the vertices of H correspond to the vertices *and edges* of G. Clearly $|V'| = 2 + |V| + |E| = O(E)$.
- E' contains three types of edges:
  - An edge $s \to v$, for each vertex $v \in V$.
  - An edge $v \to e$, for each edge $e \in E$ and each endpoint v of e.
  - An edge $e \to t$, for each edge $e \in E$.

  Altogether we have $|E'| = |V| + 2|E| + |E| = O(E)$.
- Each edge $s \to v$ has capacity k; all other edges have capacity 1.

The following figure shows the resulting flow network for the cube graph:



Our construction guarantees a correspondence between k-orientations of G and integer (s,t)-flow in H that saturates every edge into t; specifically, each flow path from s to t in H corresponds to a choice of direction for one edge in G.

- For any k-orientation of G, we can construct an integer flow f in H as follows. For each directed edge $u \to v$ in the orientation of G, we send one unit of flow through h along the path $s{\to}v{\to}uv{\to}t$; the flow f is the sum of these E paths. Because each vertex of G has at most k incoming edges, we have $f(s{\to}v) \le k$ for every vertex v. Because each edge uv is either oriented into v or not, we have $f(v{\to}uv) \le 1$. Finally, because each edge of G has exactly one orientation, we have $f(e{\to}t) = 1$ for every edge E. We conclude that f is a feasible flow in H that saturates every edge into t.

- On the other hand, let f be any integer flow in H that saturates every edge into t. We can decompose f into E paths of the form $s{\to}v{\to}uv{\to}t$, each carrying one unit of flow. For each such path, assign edge uv the direction $u{\to}v$. Because $f(e{\to}t) = 1$ for every edge e in G, every edge e in G is assigned a unique direction. Because $f(s{\to}v) \le k$ for every vertex v of G, at most k edges in G are directed into v. So we have constructed a k-orientation of G.

Thus, to solve the decision problem for any fixed k, we construct the flow network H as described above, compute a maximum (s,t)-flow f^* in H, and then report success if and only if $|f^*| = E$. If we use Orlin's algorithm to compute the maximum flow, the decision algorithm runs in $O(V'E') = O(E^2)$ time.[a]

Finally, to solve the optimization problem, we perform a binary search over all possible values of k. Every graph has a V-orientation, and no graph has a (-1)-orientation, so we can limit our search to the range $0 \le k \le V - 1$. It follows that our binary search requires $O(\log V)$ iterations, and thus our entire algorithm runs in ***$O(E^2 \log V)$ time***. ■

[a]If we used Ford-Fulkerson here, our decision algorithm would run in $O(E^2k)$ time, and so the resulting optimization algorithm would run in $O(E^2 V \log V)$ time.

Rubric: 10 points; standard graph-reduction rubric. The proof of correctness (in gray) is not required for full credit. This is not the fastest algorithm for this problem.

3. Suppose you have a sequence of jobs, indexed from 1 to n, that you want to run on two processors. For each index i, running job i on processor 1 requires $A[i]$ time, and running job i on processor 2 takes $B[i]$ time. If two jobs i and j are assigned to different processors, there is an additional communication overhead of $C[i,j] = C[j,i]$. Thus, if we assign the jobs in some subset $S \subseteq \{1,2,\ldots,n\}$ to processor 1, and we assign the remaining $n - |S|$ jobs to processor 2, then the total execution time is

$$\sum_{i\in S} A[i] \;+\; \sum_{i\notin S} B[i] \;+\; \sum_{i\in S}\sum_{j\notin S} C[i,j].$$

Describe an algorithm to assign jobs to processors so that this total execution time is as small as possible. The input to your algorithm consists of the arrays $A[1\,..\,n]$, $B[1\,..\,n]$, and $C[1\,..\,n, 1\,..\,n]$.

Solution: We reduce this problem to a standard minimum-cut problem. Given the arrays A, B, and C, we construct a flow network $G = (V, E)$ as follows:

- $V = \{1,2,\ldots,n\} \cup \{s,t\}$; that is, G has a vertex for each job, plus a source vertex s and target vertex t.
- There are three types of edges in G:
  - An edge $s \to i$ with capacity $B[i]$ for each job i.
  - An edge $i \to t$ with capacity $A[i]$ for each job i.
  - Edges $i \to j$ and $j \to i$ with capacity $C[i,j]$ for each pair of jobs $i \neq j$.

Now let (S, T) be any (s,t)-cut in this network. Definition-chasing implies that the capacity of this cut is equal to the total execution time if all jobs in S are assigned to processor 1 and all jobs in T are assigned to processor 2:

$$\begin{aligned}\|S,T\| &= \sum_{i\notin S} c(s\to i) + \sum_{i\in S} c(i\to t) + \sum_{i\in S}\sum_{j\notin S} c(i\to j)\\ &= \sum_{i\notin S} B[i] \;+\; \sum_{i\in S} A[i] \;+\; \sum_{i\in S}\sum_{j\notin S} C[i,j]\end{aligned}$$

Thus, computing the *best* assignment of jobs to processors is equivalent to computing the *minimum* (s,t)-cut in G. If we compute the minimum (s,t)-cut in G using Orlin's maximum flow algorithm, our overall algorithm runs in ***$O(VE)$ time.*** ■

Rubric: 10 points: standard flow reduction metric.