# CSC373 Summer '22
# Assignment 2

Due Date: June 29, 2022, 11:59pm ET

**Instructions**

1. Typed assignments are preferred (e.g., PDFs created using LaTeX or Word), especially if your handwriting is possibly illegible or if you do not have access to a good quality scanner. Either way, you need to submit a single PDF named "hwk2.pdf" on MarkUS at `https://markus.teach.cs.toronto.edu/2022-05`

2. You will receive 20% of the points for a (sub)question when you leave it blank (or cross off any written solution) and write "I do not know how to approach this problem." If you leave it blank but do not write this or a similar statement, you will receive 10%. This does not apply to any bonus (sub)questions.

3. You may receive partial credit for the work that is clearly on the right track. But if your answer is largely irrelevant, you will receive 0 points.

**Q1 [20 Points] Gene Alignment**

In bioinformatics, a common task involves determining *alignment* between two genes (represented as strings over the alphabet $\Sigma = \{A, C, G, T\}$). For example, a possible alignment of $x = ATGCC$ with $y = TACGCA$ is:

$$\begin{array}{cccccccc} - & A & T & - & G & C & C \\ T & A & - & C & G & C & A \end{array}$$

As you can see, this involves writing both strings in columns so that

- the characters of each string appear in order,
- each column contains a character from at least one string,
- subject to the previous constraint, columns can contain "gaps" (represented as "$-$").

The score of a possible alignment is specified through a "scoring matrix" $\delta$ that gives a value for each possible column in the alignment. In our example, the total score would be equal to:

$$\delta(-, T) + \delta(A, A) + \delta(T, -) + \delta(-, C) + \delta(G, G) + \delta(C, C) + \delta(C, A).$$

Your task is to come up with an efficient algorithm that takes as inputs two strings $x, y \in \{A, C, G, T\}^*$ with a $[5 \times 5]$ scoring matrix $\delta$, and that returns the highest possible score for alignment between $x$ and $y$. (Assume that $\delta(-, -) = -\infty$, representing the fact that no alignment can align two gaps together.)

**(a)** [5 Points] Define the array(s) or table(s) that your solution would compute. Clearly explain what each entry means, and how you would compute the final answer given all the entries in your array(s) or table(s).

**(b)** [7 Points] Write a Bellman equation and briefly justify its correctness.

**(c)** [4 Points] Describe a bottom-up implementation of your Bellman equation, and analyze its worst-case running time and space complexity.

**(d)** [4 Points] Now, modify your algorithm to give a way to print the highest-scoring *alignment* itself. For this *reconstruction of the solution*, you may want to define an additional array, although that is not strictly necessary. Re-analyze the running time and space complexity of the modified algorithm.
HINT: You may find it useful to read Section 15.4 of the textbook.

**Solution to Q1**

We are to write an efficient algorithm that takes as inputs two strings $x, y \in \{A, C, G, T\}^*$ along with a $[5 \times 5]$ scoring matrix $\delta$ (with $\delta(-,-) = -\infty$), and that returns the highest-scoring alignment between $x$ and $y$.

**Optimal Substructure Property:** Let $x = x_1 \cdots x_m$ and $y = y_1 \cdots y_n$ where each $x_i$ and each $y_i$ is in $\{A, C, G, T\}$ and $m \geq 0$ and $n \geq 0$ (but not both $m$ and $n$ are 0).
Consider an optimal alignment of $x$ and $y$. Either $x_m$ is aligned with $y_n$, or $x_m$ is aligned with a gap, or $y_n$ is aligned with a gap. In every case, the alignment for the rest of $x$ and $y$ must have the maximum possible score – else it would be possible to increase the total score of the current optimum alignment.

**(a)** Let $C[k, \ell]$ be the score of the highest scoring alignment between the initial segments $x_1 \cdots x_k$ and $y_1 \cdots y_\ell$, where $0 \leq k \leq m$ and $0 \leq \ell \leq n$.
The highest score among all possible alignments of $x$ and $y$ is given by $C[m, n]$.

**(b)**

$$C[k,\ell] = \begin{cases} 0 & \text{if } k = 0 \text{ and } \ell = 0, \\ \delta(x_k, -) + C[k-1, \ell] & \text{if } k > 0 \text{ and } \ell = 0, \\ \delta(-, y_\ell) + C[k, \ell-1] & \text{if } k = 0 \text{ and } \ell > 0, \\ \max\big(\delta(x_k, -) + C[k-1, \ell], \delta(-, y_\ell) + C[k, \ell-1], & \\ \delta(x_k, y_\ell) + C[k-1, \ell-1]\big) & \text{if } k > 0 \text{ and } \ell > 0, \end{cases}$$

for all $0 \leq k \leq m$ and $0 \leq \ell \leq n$, based on the recursive structure of optimal solutions discussed above.

**(c)** Algorithm 1 computes the values of $C[k, \ell]$ directly from the recurrence above and runs in worst-case time $\Theta((n+1)(m+1)) = \Theta(nm)$. The space complexity is also $\Theta(nm)$ as this is the size of the array $C$.

**(d)** Once we have computed the array values $C[k, \ell]$, we can use them to print an optimal alignment as described in Algorithm 2: starting at $C[m, n]$, test the current value against all three possibilities in the recurrence relation to determine the best alignment for the last character(s) of $x$ and $y$, until both sequences are completely aligned.

**Algorithm 1** Algorithm to compute score of best alignment

**Require:** $x = x_1 \cdots x_m, y = y_1 \cdots y_n, \delta$
**Ensure:** Score of the highest-scoring alignment between $x$ and $y$

```
C[0,0] ← 0
for k = 1, ..., m do
    C[k,0] ← δ(x_k, −) + C[k−1, 0]
end for
for ℓ = 1, ..., n do
    C[0,ℓ] ← δ(−, y_ℓ) + C[0, ℓ−1]
    for k = 1, ..., m do
        C[k,ℓ] ← max(δ(x_k, −) + C[k−1, ℓ], δ(−, y_ℓ) + C[k, ℓ−1], δ(x_k, y_ℓ) + C[k−1, ℓ−1])
    end for
end for
```

**Algorithm 2** Algorithm to print the best alignment

**Require:** $x = x_1 \cdots x_m, y = y_1 \cdots y_n, \delta, C$
**Ensure:** Highest-scoring alignment between $x$ and $y$

```
A = [ ]   # current alignment, stored as a list of pairs
(k,ℓ) ← (m,n)
while k > 0 and ℓ > 0 do
    if C[k,ℓ] = δ(x_k, −) + C[k−1, ℓ] then
        A ← [(x_k, −)] + A
        k ← k − 1
    else
        if C[k,ℓ] = δ(−, y_ℓ) + C[k, ℓ−1] then
            A ← [(−, y_ℓ)] + A
            ℓ ← ℓ − 1
        else
            A ← [(x_k, y_ℓ)] + A
            k ← k − 1
            ℓ ← ℓ − 1
        end if
    end if
end while
while k > 0 do
    A ← [(x_k, −)] + A
    k ← k − 1
end while
while ℓ > 0 do
    A ← [(−, y_ℓ)] + A
    ℓ ← ℓ − 1
end while
return A
```

This requires additional time $\Theta(m+n)$ in the worst case, and no significant additional space usage (other than storing the values of the counters which is only $\Theta(\log m + \log n)$).

**Q2 [20 Points] Board Cutting**

A sawmill gets orders for different lengths of boards $\ell_1, \ell_2, \ldots, \ell_n$. All of the boards have to be cut from *planks*, long pieces of wood with a fixed length $L$, and for technical reasons, the boards have to be cut in the order they are given. No matter how the planks are cut into boards, there is usually some amount of waste left over from each plank.
For example, if the board lengths are $2, 4, 1, 5$ and $L = 7$, then we can cut the boards in many different ways — some of them are clearly silly:

- cut board 2 from one plank, 4 from another, 1 from another, and 5 from a fourth plank (leaving four waste pieces with lengths $5, 3, 6, 2$), or
- cut board 2 from one plank, 4 from another, and $1, 5$ from a third plank (leaving three waste pieces with lengths $5, 3, 1$), or
- cut board 2 from one plank, $4, 1$ from another, and 5 from a third plank (leaving three waste pieces with lengths $5, 2, 2$), or
- cut boards $2, 4$ from one plank, 1 from another, and 5 from a third plank (leaving three waste pieces with lengths $1, 6, 2$), or
- cut boards $2, 4$ from one plank and $1, 5$ from another (leaving two waste pieces of length 1 each), or
- cut boards $2, 4, 1$ from one plank and 5 from another (leaving only one waste piece of length 2).

Instructions like "cut boards $2, 5$ from one plank and $4, 1$ from another" are *not* valid solutions because they change the order of the boards.
From the point of view of the sawmill, what matters most is *not* how many planks are used, nor how much waste is left in total. What matters is that the waste pieces all be the same length, as much as possible, because this allows those pieces to be reused more easily for other purposes. So in the example above, the best choice is the second-last ($2, 4$ together and $1, 5$ together) — the last solution is not quite as good because the lengths of the waste pieces are 0 and 2 (the plank with no waste is considered to have waste of length 0).
Formally, consider the following "Board Cutting" problem:

**Input:** Board lengths $\ell_1, \ell_2, \ldots, \ell_n$ and plank length $L$, where each length is a positive integer, each $\ell_i \leq L$, and the board lengths are not necessarily all distinct.

**Goal:** A division of $\ell_1, \ell_2, \ldots, \ell_n$ into groups $(\ell_{i_0+1}, \ldots, \ell_{i_1}); (\ell_{i_1+1}, \ldots, \ell_{i_2}); \ldots; (\ell_{i_{k-1}+1}, \ldots, \ell_{i_k})$, where $i_0 = 0$, $i_k = n$, $\ell_{i_j+1} + \ell_{i_j+2} + \cdots + \ell_{i_{j+1}} \leq L$ for $0 \leq j \leq k-1$ (intuitively: the total length of each group is no more than the length of one plank), and $\sum_{j=0}^{k-1} \left(L - \ell_{i_j+1} - \ell_{i_j+2} - \cdots - \ell_{i_{j+1}}\right)^2$ is minimized (intuitively: the lengths of the leftover pieces of plank are all as close to each other as possible).

**(a)** [4 Points] Define the array(s) or table(s) that your solution would compute, in order to obtain the *value* of the minimized objective function. Clearly explain what each entry means, and how you would compute the final answer given all the entries in your array(s) or table(s).

**(b)** [6 Points] Write a Bellman equation and briefly justify its correctness.

**(c)** [5 Points] Describe a bottom-up implementation of your Bellman equation, and analyze its worst-case running time and space complexity.

**(d)** [5 Points] Now, modify your algorithm to output the optimal *division* itself. You may define an additional array to do so, in which case, specify its definition clearly. Re-analyze the running time and space complexity of the modified algorithm.

**Solution to Q2**

In every optimal solution $(\ell_{i_0+1}, \ldots, \ell_{i_1}); (\ell_{i_1+1}, \ldots, \ell_{i_2}), \ldots, (\ell_{i_{k-1}+1}, \ldots, \ell_{i_k})$, the division of the first $k-1$ groups (the ones that correspond to input $\ell_1, \ldots, \ell_{i_{k-1}}$) must be done optimally – otherwise, we could replace the first $k-1$ groups to get a better overall solution.

**(a)** We define an array that stores optimal values for arbitrary sub-problems. For $j = 0, 1, \ldots, n$, let $A[j]$ denote the value of an optimal solution to the problem with input $\ell_1, \ldots, \ell_j$. Then, $A[n]$ would be the answer to the original problem.

**(b)** Now, we give a recurrence relation for the array values:
$A[0] = 0$ (there are no solutions to consider).
For $j = 1, 2, \ldots, n$, $A[j] = \min\left\{A[i] + (L - \ell_{i+1} - \cdots - \ell_j)^2 : 0 \leq i < j \text{ and } \ell_{i+1} + \cdots + \ell_j \leq L\right\}$ (consider every possible last group).

**(c)** Write a bottom-up algorithm to compute the array values, following the recurrence.

**Algorithm 3** Algorithm to minimize the objective function

```
A[0] ← 0
for j ∈ [1, 2, ..., n] do
    A[j] ← A[j − 1] + (L − ℓ_j)^2
    for i ∈ [j − 2, j − 3, ..., 1] do
        while ℓ_{i+1} + ℓ_{i+2} + ··· + ℓ_j ≤ L do
            if A[i] + (L − ℓ_j − ··· − ℓ_{i+1})^2 < A[j] then
                A[j] ← A[i] + (L − ℓ_j − ··· − ℓ_{i+1})^2
            end if
        end while
    end for
end for
```

Runtime: $\Theta(n^2)$ for the two nested loops, in the worst-case. Space: $\Theta(n)$ (the size of the array).

**(d)** We now use the computed values to reconstruct an optimal solution.
We need a second array $B[j]$ to store the index $i$ such that $A[j] = A[i] + (L - \ell_j - \cdots - \ell_{i+1})^2$.
Now, we can reconstruct the solution one group at a time, starting from the end and working backward.

**Algorithm 4** Algorithm to define the helper array

$A[0] \leftarrow 0$
$B[0] \leftarrow 0$
**for** $j \in [1, 2, \ldots, n]$ **do**
    $A[j] \leftarrow A[j-1] + (L - \ell_j)^2$
    $B[j] \leftarrow j - 1$
    **for** $i \in [j-2, \ldots, 1]$ **do**
        **while** $\ell_{i+1} + \cdots + \ell_j \leq L$ **do**
            **if** $A[i] + (L - \ell_j - \cdots - \ell_{i+1})^2 < A[j]$ **then**
                $A[j] \leftarrow A[i] + (L - \ell_j - \cdots - \ell_{i+1})^2$
                $B[j] \leftarrow i$
            **end if**
        **end while**
    **end for**
**end for**

**Algorithm 5** Algorithm for reconstructing an optimal solution

$S \leftarrow \emptyset$
$j \leftarrow n$
**while** $j > 0$ **do**
    $S \leftarrow S \cup \{(\ell_{B[j]+1}, \ldots, \ell_j)\}$
    $j \leftarrow B[j]$
**end while**
return $S$

Runtime: $\Theta(n^2)$ to compute the values in $A[]$ and $B[]$ (reconstructing the solution takes only $\Theta(n)$). Space: $\Theta(n)$.

a) false

**Q3 [20 Points] Reducing Edge Capacities**

**(a)** [4 Points] Prove or disprove: if $N = (V, E)$ is a network, $f^*$ is a maximum flow in $N$, $e_0 \in E$ is an edge with $f^*(e_0) = c(e_0)$, and $N'$ is the same network as $N$ except that $c'(e_0) = c(e_0) - 1$, then the maximum flow $f'$ in $N'$ satisfies $|f'| < |f^*|$.

**(b)** [12 Points] Write an algorithm that takes a network $N = (V, E)$, maximum flow $f^*$ in $N$, and an edge $e_0 \in E$ with $f^*(e_0) = c(e_0)$, and that outputs a maximum flow for network $N'$, where $N'$ is the same as $N$ except that $c'(e_0) = c(e_0) - 1$.
Provide a brief argument that your algorithm is correct and analyze its time complexity. For full credit, your algorithm should be as efficient as possible.

**(c)** [4 Points] Write an algorithm that takes a network $N = (V, E)$ and that outputs a list of **all** edges $e_1, \ldots, e_k$ with the property that if the capacity of any edge in that list is reduced **by one unit**, the value of the maximum flow in $N$ is also reduced.

**Solution to Q3**

**(a)** We *disprove* the statement with the counter-example depicted in Figure 1, where $e_0 = (s, b)$ and the network $N$ and max flow $f^*$ are as depicted in the figure. If we reduce $c(s, b)$ from 3 to 2, it is possible to adjust the flow in the resulting network $N'$ by setting $f'(s, b) = 2$ and $f'(s, a) = f'(a, b) = 1$ so that $|f'| = 3 = |f^*|$.

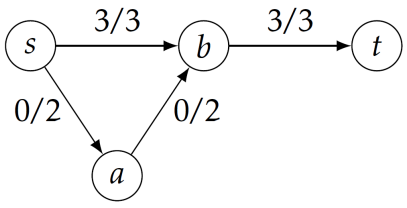



Figure 1: Counterexample to Part (a)

**(b)**

**Algorithm:** On input $N, f^*, e_0$ as described in the problem statement, let $e_0 = (u, v)$:

1. Start with $f'(e) = f^*(e)$ for all edges $e \in N'$.
2. Run BFS in $N'$ to find *reducing* paths starting from $v$ – where every forward edge has non-zero flow ($f'(e) > 0$) and every backward edge has unused capacity ($f'(e) < c(e)$). This yields one of two possibilities: a path from $v$ to $t$, or a path from $v$ to $u$ that does *not* go through $t$.
3. If this finds a path from $v$ to $u$ that does not go through $t$, then $N'$ contains a *reducing cycle* $C$ starting and ending at $u$.
   - Reduce flow $f'$ along cycle $C$, by subtracting 1 from the flow on every forward edge (including $e_0$) and adding 1 to the flow on every backward edge.

- Return $f'$.

4. Else, this finds some reducing path $P_1$ from $v$ to $t$.
   - Run BFS again in $N'$ to find a reducing path $P_2$ from $s$ to $u$.
   - Reduce flow $f'$ along path $P_2 + e_0 + P_1$, by subtracting 1 from the flow on every forward edge (including $e_0$) and adding 1 to the flow of every backward edge.
   - Run BFS in $N'$ to find an augmenting path; if one exists, augment flow $f'$ along that path.
   - Return $f'$.

**Correctness:** Finding a reducing path allows us to adjust the initial flow $f' = f^*$ so that $f'(e_0) = c'(e_0)$ while preserving flow conservation (for every node on the reducing path, the total flow in is still equal to the total flow out after the adjustment). After this adjustment, $f'$ is a valid flow for $N'$. Moreover, if the flow is adjusted along a reducing cycle, then this does not reduce the total flow in the network so the resulting $f'$ is still maximum. Else, the only difference between $N$ and $N'$ is that $c'(e_0) = c(e_0) - 1$, and we have $|f'| = |f^*| - 1$ after the reduction, so either $f'$ is a maximum flow for $N'$ (in which case we will not find an augmenting path and the algorithm will return $f'$), or it is possible to augment $f'$ by 1 unit (in which case we will find an augmenting path and the algorithm will return the updated $f'$).

**Runtime:** Each execution of BFS takes time $\Theta(|V|+|E|)$; each flow reduction and augmentation takes time $\Theta(|V|)$ (because each path has length at most $|V|$). The total time is therefore linear: $\Theta(|V| + |E|)$.

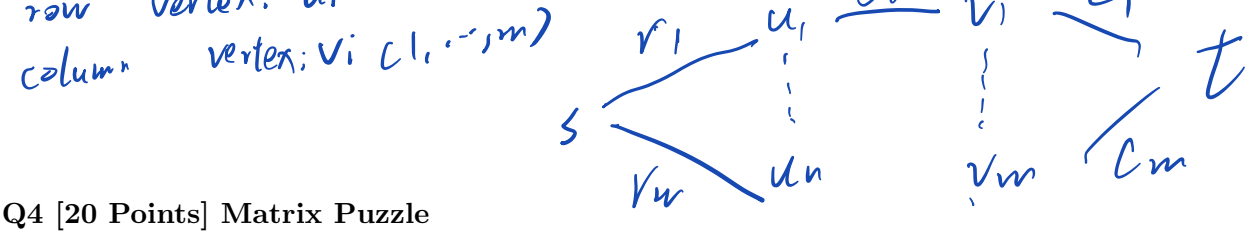
**(c)**

---

**Require:** $N = (V, E)$
**Ensure:** A list of all edges with the desired property
  $S = \emptyset$
  Run an implementation of Ford-Fulkerson to find a maximum flow $f$.
  **for** every edge $e \in E$ such that $f(e) = c(e)$ **do**
    Run the algorithm from the previous part with $e_0 = e$ to update the maximum flow to $f'$.
    **if** $|f'| < |f|$ **then**
      add $e$ to $S$
    **end if**
  **end for**
  Return $S$.

---

The algorithm runs in time $O(|E| \times (|V|+|E|))$, and it is correct because it directly verifies whether or not reducing $c(e)$ reduces the value of the max flow, for each edge $e$ with $f(e) = c(e)$.




**Q4 [20 Points] Matrix Puzzle**

You want to fill a matrix of $n$ rows and $m$ columns with non-negative integers so that the sum of each row or each column corresponds to a predetermined number. Each matrix entry also has an upper bound constraint. Specifically, given **non-negative** integers $r_1, \ldots, r_n$, $c_1, \ldots, c_m$, and $b_{1,1}, \ldots, b_{n,m}$, your task is to find integers $a_{1,1}, \ldots, a_{n,m}$ such that:

1. $0 \leq a_{i,j} \leq b_{i,j}$ for all $1 \leq i \leq n$, $1 \leq j \leq m$;
2. $r_i = \sum_{k=1}^{m} a_{i,k}$ for all $1 \leq i \leq n$;
3. $c_j = \sum_{k=1}^{n} a_{k,j}$ for all $1 \leq j \leq m$.

**(a)** [10 Points] Design an algorithm to find such integers. Your algorithm should return NIL if such integers do not exist.

**(b)** [8 Points] Prove the correctness of your algorithm.

**(c)** [2 Points] Analyze the running time of your algorithm.

**Solution to Q4**

**(a)**

**Algorithm:** Construct a bipartite graph $G = (V_1, V_2, E)$ as follows.

1. Create one vertex $u_i \in V_1$ for each row; create one vertex $v_j \in V_2$ for each column.
2. For each pair $i, j$ where $1 \geq i \geq n$ and $1 \leq j \leq m$, create an edge $(u_i, v_j)$ with capacity $c(u_i, v_j) = b_{i,j}$.
3. Create a source $s$ and an edge from $s$ to each $u_i \in V_1$, with $c(s, u_i) = r_i$.
4. Create a sink $t$ and an edge from each $v_j \in V_2$ to $t$, with $c(v_j, t) = c_j$.

Then, run a max-flow algorithm to find a maximum flow on the constructed network. If the resulting flow $f$ does not saturate the edges connecting to $s$ and $t$, then there is no possible matrix solution. Otherwise $a_{i,j} = f(u_i, v_j)$ gives a solution.

**(b)**

**Correctness:** Suppose the max-flow $f$ saturate all edges from $s$ and all edges to $t$. By the conservation constraint on $u_i \in V_1$, we have:

$$r_i = c(s, u_i) = f(s, u_i) = \sum_{k=1}^{m} f(u_i, v_k) = \sum_{k=1}^{m} a_{i,k}$$

Similarly by the conservation constraint on $v_i \in V_2$, we have:

$$c_j = c(v_j, t) = f(v_j, t) = \sum_{k=1}^{m} f(u_k, v_j) = \sum_{k=1}^{m} a_{k,j}$$

**Algorithm 6** Algorithm to construct the matrix

$V_1 \leftarrow \{u_1, u_2, \ldots, u_n\}$
$V_2 \leftarrow \{v_1, v_2, \ldots, v_m\}$
$E \leftarrow \big\{(s, u_i) \text{ with } c(s, u_i) = r_i : 1 \leq i \leq n\big\} \cup \big\{(v_j, t) \text{ with } c(v_j, t) = c_j : 1 \leq j \leq m\big\} \cup \big\{(u_i, v_j) \text{ with } c(u_i, v_j) = b_{i,j} : 1 \leq i \leq n, 1 \leq j \leq m\big\}$
Find max-flow $f$ in the network $N = (\{s, t\} \cup V_1 \cup V_2, E)$
**if** $\exists u_i \in V_1, f(s, u_i) < c(s, u_i)$ or $\exists v_i \in V_2, f(v_i, t) < c(v_i, t)$ **then**
    return NIL
**else**
    Build a matrix $A$ such that $a_{i,j} = f(u_i, v_j)$
    return $A$
**end if**

Therefore the returned matrix $A$ is a valid solution.
On the other hand, for a valid matrix solution $A$, consider the following flow $f$ by:

- $\forall u_i \in V_1, f(s, u_i) = c(s, u_i)$
- $\forall v_j \in V_2, f(v_j, t) = c(v_j, t)$
- $\forall u_i \in V_1, v_j \in V_2, f(u_i, v_j) = a_{i,j}$

$f$ is a flow that satisfies the conservation constraint on all vertices and $|f| = \sum_{i=1}^{n} r_i = \sum_{j=1}^{m} c_j$. Therefore if the max-flow algorithm outputs a flow that does not saturate the edges from $s$ and edges to $t$, it means that no such matrix solution $A$ exists.

**(c)**

**Runtime:** Constructing the network takes time $O(nm)$, and so does constructing a solution matrix $A$ from maximum flow values. Finding a maximum flow is the most time-consuming part of the algorithm, requiring time $O(nm^2)$ if we use the Edmonds-Karp implementation of the Ford-Fulkerson algorithm.

## BONUS QUESTION

### Q5 [10 Points] Knight Coexistence

You want to place knights on an $n \times n$ chessboard. The chessboard has $n^2$ positions from $(0, 0)$ to $(n-1, n-1)$. A knight at position $(x, y)$ is able to attack up to eight different positions: $(x+2, y+1)$, $(x+2, y-1)$, $(x+1, y+2)$, $(x+1, y-2)$, $(x-1, y+2)$, $(x-1, y-2)$, $(x-2, y+1)$, and $(x-2, y-1)$ – for each position that lies on the board, of course. Moreover, some positions are blocked so that you cannot place knights on them. Given $n$ and a list of all blocked positions, find a way to place as many knights as possible on the chessboard so that the placed knights cannot attack each other.

**Solution to bonus problem.** For every non-blocked chessboard position $(x, y)$, create a vertex $v_{x,y}$. For every pair of non-blocked positions $(x_1, y_1)$ and $(x_2, y_2)$ on which knights can attack each other, create an edge between $v_{x_1,y_1}$ and $v_{x_2,y_2}$. Note that the resulting graph is a bipartite graph because each odd position (i.e., $x + y$ is odd) can only attack even positions (i.e., $x + y$ is even) and vice versa. Therefore run the maximum independent set algorithm[1] on the bipartite graph to find a maximum independent set, and place a knight on a chessboard position iff the corresponding vertex is in the resulting independent set. Figure 2 describes the pseudo-code.

**Correctness:** Because any two vertices in an independent set do not share an edge, the placed knights therefore will not be able to attack each other. So every independent set yields a valid placement of knights. On the other hand, any knight coexistence placement maps back to some independent set on the constructed bipartite graph. Hence, the maximum number of placed knights is equal to the maximum size of any independent set.

**Runtime:** Constructing the network takes time $O(n^2)$, and so does constructing a solution from a maximum independent set. Finding a minimum cut (to find a maximum independent set) is the most time-consuming part of the algorithm, requiring time $O(nm^2)$ if we use the Edmonds-Karp implementation of the Ford-Fulkerson algorithm.

[1] An Independent Set in a graph is a collection of mutually non-adjacent vertices. Show a $1-1$ correspondence between maximum matchings in a bipartite graph and maximum independent sets; therefore, we may use the maximum matching algorithm for bipartite graphs from class to find a maximum independent set in this graph.

**Pseudo-code:**

$V \leftarrow \{v_{0,0}, v_{0,1}, \ldots, v_{n-1,n-1}\}$
$E \leftarrow \varnothing$
**for** $i$ **in** $0, 1, \ldots, n-1$:
    **for** $j$ **in** $0, 1, \ldots, n-1$:
        **if** $(i, j)$ and $(i-1, j-2)$ are non-blocking valid positions:
            $E \leftarrow E \cup \{(v_{i,j}, v_{i-1,j-2})\}$
        **if** $(i, j)$ and $(i-1, j+2)$ are non-blocking valid positions:
            $E \leftarrow E \cup \{(v_{i,j}, v_{i-1,j+2})\}$
        **if** $(i, j)$ and $(i+1, j-2)$ are non-blocking valid positions:
            $E \leftarrow E \cup \{(v_{i,j}, v_{i+1,j-2})\}$
        **if** $(i, j)$ and $(i+1, j+2)$ are non-blocking valid positions:
            $E \leftarrow E \cup \{(v_{i,j}, v_{i+1,j+2})\}$
        **if** $(i, j)$ and $(i-2, j-1)$ are non-blocking valid positions:
            $E \leftarrow E \cup \{(v_{i,j}, v_{i-2,j-1})\}$
        **if** $(i, j)$ and $(i-2, j+1)$ are non-blocking valid positions:
            $E \leftarrow E \cup \{(v_{i,j}, v_{i-2,j+1})\}$
        **if** $(i, j)$ and $(i+2, j-1)$ are non-blocking valid positions:
            $E \leftarrow E \cup \{(v_{i,j}, v_{i+2,j-1})\}$
        **if** $(i, j)$ and $(i+2, j+1)$ are non-blocking valid positions:
            $E \leftarrow E \cup \{(v_{i,j}, v_{i+2,j+1})\}$
$I \leftarrow$ find maximum independent set for $G = (V, E)$
place knights on all positions $(i, j)$ where $v_{i,j} \in I$

Figure 2: Algorithm to find maximum number of non-attacking knights