1. Problem 1: Approximating MaxCut

An algorithm $A$ is an $\alpha$-approximation algorithm for MaxCut for some $0 < \alpha \leq 1$ if for every graph $G$ the algorithm returns a cut $A(G)$ of size

$$\frac{|A(G)|}{|OPT(G)|} \geq \alpha$$

where $OPT(G)$ is the maximum cut of $G$.

(i) Show that there is an efficient $\frac{1}{2}$-approximation algorithm for MaxCut.

Theorem 1. There is a polynomial-time $\frac{1}{2}$-approximation algorithm for MaxCut.

Proof. Enter your answer here. □

Hint. You may want to design a greedy algorithm that considers the vertices one at a time in an arbitrary order.

(ii) Let DirectedMaxCut be the variant of MaxCut where the goal is to find the partition $(S, V \setminus S)$ of the vertices of a directed graph $G = (V, E)$ that maximizes the number of directed edges going from $S$ to $E \setminus S$. Show that this problem can also be efficiently approximated.

Theorem 2. There is a polynomial-time $\frac{1}{4}$-approximation algorithm for DirectedMaxCut.

Proof. Enter your answer here. □

(iii) Bonus. Can you improve either of these bounds?
2. Problem 2: Travelling with few weights

An special case of the Metric TSP problem is the (1, 2)-TSP problem where each edge weight in the complete graph is either 1 or 2. You will show how to improve on the best-known approximation algorithm for Metric TSP in this case.

(i) A minimum double matching in a weighted graph is a minimum-weight collection $M$ of edges such that each vertex is adjacent to exactly 2 edges in $M$. Prove that there is an efficient algorithm for finding a minimum double matching. You may use results from previous courses (e.g., CS 341) without proof in your answer if you give precise statements and references.

Lemma 1. There is a polynomial-time algorithm for finding a minimum double matching in the complete weighted graph.

Proof. Enter your answer here. □

(ii) Use the result above to obtain a competitive TSP tour.

Theorem 3. There is a polynomial-time $\frac{4}{3}$-approximation algorithm for the (1, 2)-TSP problem.

Proof. Enter your answer here. □

(iii) Bonus. Can you generalize the result to obtain approximation algorithm for the 1-to-$k$-TSP problem where the weights in the graphs must be integers in the set $\{1, \ldots, k\}$ for some $k \geq 3$?
3. Problem 3: Covering vertices

There is a very natural algorithm for the VertexCover problem that at each step adds the vertex which covers the largest number of still-uncovered edges:

**Algorithm 1: VC2(V,E)**

\[
S \leftarrow \emptyset; \\
\text{while } E \neq \emptyset \text{ do} \\
\quad \text{Let } v \in V \text{ be the vertex that covers the largest number of edges in } E; \\
\quad S \leftarrow S \cup \{v\}; \\
\quad E \leftarrow E \setminus \{(u,v) : u \in V\}; \\
\text{Return } S;
\]

The goal of this question is to show that this algorithm is a \(O(\log n)\)-approximation algorithm for VertexCover.

(i) Define the cost of an edge \(e \in E\) by running the VC2 algorithm, determining the number \(t_e\) of (previously-uncovered) edges covered by the vertex \(v\) that covers \(e\) in the algorithm, and setting \(\text{cost}(e) = 1/t_e\).

**Lemma 2.** The size of the vertex cover returned by VC2 satisfies

\[|S| = \sum_{e \in E} \text{cost}(e).\]

**Proof.** Enter your answer here. □

(ii) Let \(e_1, \ldots, e_n\) be an ordering of the edges in \(E\) according to the order in which they are covered in the algorithm VC2.

**Lemma 3.** For every \(k \leq n\), the cost of edge \(e_k\) satisfies

\[\text{cost}(e_k) \leq \frac{\text{OPT}}{n - k + 1}\]

where OPT is the size of the optimal vertex cover of the graph.

**Proof.** Enter your answer here. □

(iii) The harmonic numbers satisfy the bound \(H_n := 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} = O(\log n)\). Use this fact and the results above to show that VC2 is a valid approximation algorithm for VertexCover.

**Theorem 4.** VC2 is a \(O(\log n)\)-approximation algorithm for VertexCover.

**Proof.** Enter your answer here. □

(iv) **Bonus.** Show that the bound in the theorem is tight for some graphs.
4. Problem 4: Variant of disjointness

Let $2^{[n]}$ denote the set of subsets of $[n] = \{1, 2, \ldots, n\}$ and fix $k \geq 1$. The function $\text{Disjoint}_k : 2^{[n]} \times 2^{[n]} \to \{0, 1\}$ is defined by

$$
\text{Disjoint}_k(S, T) = \begin{cases} 
1 & \text{if } S \cap T = \emptyset, |S| \leq k, \text{ and } |T| \leq k \\
0 & \text{otherwise.}
\end{cases}
$$

Establish the following bounds on the communication complexity of the $\text{Disjoint}_k$ function.

(i) Show that when $k$ is small, the communication complexity of $\text{Disjoint}_k$ is also (fairly) small.

**Theorem 5.** For every $k \geq 1$, $D^{\text{CC}}(\text{Disjoint}_k) = O(k \log n)$.

*Proof.* Enter your answer here.

(ii) Show that, conversely, when $k$ is large, the communication complexity of $\text{Disjoint}_k$ is also (fairly) large.

**Theorem 6.** For every $k \geq 1$, $D^{\text{CC}}(\text{Disjoint}_k) \geq k$.

*Proof.* Enter your answer here.

(iii) **Bonus.** Show that the communication complexity of $\text{Disjoint}_k$ must depend on $n$.

**Theorem 7.** For every $k \geq 1$, $D^{\text{CC}}(\text{Disjoint}_k) \geq \log(\frac{n}{k})$.

*Proof.* Enter your answer here.

For an extra bonus challenge: close the gap between the upper and lower bounds on the communication complexity of $\text{Disjoint}_k$! (Warning: I suspect the true communication complexity is $\Theta(k \log n)$, but I don’t have a proof of that fact at the moment—and if that bound is correct, it’s possible that it is not achievable via the fooling set method.)
5. Problem 5: Palindromes

A simple Turing machine algorithm is an algorithm that receives its input $x \in \{0, 1\}^n$ on a tape and has only $O(1)$ bits of additional memory. Initially, the algorithm sees the first bit of the input. At each time step, an algorithm in this model can overwrite the current bit that it sees with an $X$ or leave the current bit as is, then move one step to the left or to the right. The time complexity of an algorithm in this model is the number of time steps that elapse before the algorithm terminates and outputs its answer.

The Palindrome : $\{0, 1\}^n \rightarrow \{0, 1\}$ function determines if its input $x = (x_1, \ldots, x_n)$ is identical to its reverse $x^R = (x_n, \ldots, x_1)$. Formally,

$$\text{Palindrome}(x) = \begin{cases} 1 & \text{if } x = x^R \\ 0 & \text{otherwise.} \end{cases}$$

It’s not hard to see that there is a simple Turing machine algorithm that computes Palindrome in time $O(n^2)$. (How?)

(i) Show that the simple algorithm for Palindrome is optimal.

**Theorem 8.** Every simple Turing machine algorithm for Palindrome has time complexity $\Omega(n^2)$.

**Proof.** Enter your answer here. □

*Hint.* You may want to consider inputs of the form $x00\cdots0y^R$ for some $x, y \in \{0, 1\}^n$.

(ii) **Bonus.** Show that the same $\Omega(n^2)$ bound holds even when we consider general Turing machine algorithms, which can overwrite characters on the input tape with any bit (and not just overwrite them with $X$s).