# Exercise 1

Design a 2-approximation algorithm for the maximum cut problem in an undirected graph G = (V, E). The maximum cut problem aims to partition the vertices V into two disjoint subsets S and V \ S such that the number of edges with one endpoint in S and the other in V \ S is maximized. Prove that your algorithm achieves a 2-approximation.

## Solution

(a) Here's a simple 2-approximation algorithm for the maximum cut problem:

APPROX\_MAXCUT(G)

S = ∅

for each vertex v ∈ V do

with probability 1/2

S = S ∪ {v}

return S

(b) To prove that this algorithm achieves a 2-approximation, let's analyze the expected size of the cut (S, V \ S) produced by the algorithm.

For each edge e = (u, v) ∈ E, the probability that e crosses the cut (i.e., one endpoint is in S and the other is in V \ S) is 1/2. This is because the algorithm independently assigns each vertex to S with probability 1/2.

Let X\_e be an indicator random variable for the event that edge e crosses the cut. Then, E[X\_e] = 1/2. The expected size of the cut is:

E[|cut(S, V \ S)|] = E[∑\_{e ∈ E} X\_e] = ∑\_{e ∈ E} E[X\_e] = |E|/2

Let OPT denote the size of a maximum cut in G. Since every cut has size at most |E|, we have OPT ≤ |E|. Therefore,

OPT / E[|cut(S, V \ S)|] ≤ |E| / (|E|/2) = 2

Thus, the algorithm achieves a 2-approximation in expectation. By using the method of conditional expectations or randomized rounding, we can derandomize this algorithm to obtain a deterministic 2-approximation algorithm

# Exercise 2

Consider a variant of the traveling salesperson problem (TSP) where the goal is to find a tour that visits all the vertices of a graph exactly once, except for the starting vertex, which must be visited twice (once at the beginning and once at the end of the tour). Assume that the graph is complete, and the edge weights satisfy the triangle inequality. Design a 2-approximation algorithm for this variant of TSP and prove its approximation ratio.

## Solution

(a) Here's a 2-approximation algorithm for the given variant of TSP:

APPROX\_TSP\_VARIANT(G)

Compute a minimum spanning tree T of G

Double each edge of T to obtain an Eulerian graph T'

Find an Eulerian tour in T' and shortcut it to obtain a tour H

return H

(b) To prove that this algorithm achieves a 2-approximation, let's analyze the cost of the tour H produced by the algorithm.

Let OPT denote the cost of an optimal tour for this TSP variant. Since the edge weights satisfy the triangle inequality, we have:

cost(T) ≤ OPT

The cost of the Eulerian tour in T' is exactly 2 · cost(T), as each edge is doubled. By applying the shortcutting step, we can only decrease the cost of the tour. Therefore,

cost(H) ≤ 2 · cost(T) ≤ 2 · OPT

# Exercise 3

Given a universe U of n elements and a collection of subsets of U, S = {S1, S2, ..., Sm}, the Set Cover problem aims to find the minimum number of subsets from S whose union covers all elements in U.

Design a greedy approximation algorithm for the Set Cover problem that achieves an approximation ratio of O(log n). Prove the approximation ratio using the definition of the approximation factor.

## Solution

The greedy approximation algorithm for Set Cover is as follows:

1. Initialize an empty set cover C and a set U' containing all uncovered elements of U.

2. While U' is not empty:

a. Select the subset Si from S that covers the maximum number of uncovered elements in U'.

b. Add Si to the set cover C.

c. Remove the elements covered by Si from U'.

3. Return the set cover C.

Proof of approximation ratio:

Let OPT be the size of an optimal set cover and let H\_n be the n-th Harmonic number, defined as H\_n = ∑\_{i=1}^n (1/i) ≤ ln(n) + 1.

At each iteration, the greedy algorithm selects the subset that covers the maximum number of uncovered elements. Let ui be the number of uncovered elements before the i-th iteration. The number of elements covered by the selected subset in the i-th iteration is at least ui / OPT, as an optimal solution must cover all ui elements with OPT sets.

After the i-th iteration, the number of remaining uncovered elements is at most:

ui+1 ≤ ui - (ui / OPT) = ui \* (1 - 1/OPT)

Applying this inequality iteratively, we get:

ui ≤ n \* (1 - 1/OPT)^(i-1)

The algorithm terminates when ui ≤ 1. Let k be the number of iterations until termination. Then:

1 ≤ n \* (1 - 1/OPT)^(k-1)

Taking the natural logarithm of both sides and solving for k, we get:

k ≤ OPT \* H\_n ≤ OPT \* (ln(n) + 1)

Therefore, the greedy algorithm returns a set cover of size at most OPT \* (ln(n) + 1), achieving an approximation ratio of O(log n).

Time complexity: O(n \* m \* min(n, m)), where n is the number of elements and m is the number of subsets.

# Exercise 4

Design a randomized algorithm to determine whether an undirected graph G = (V, E) is connected or not. The algorithm should have a one-sided error probability of at most 1/n, where n is the number of vertices in the graph. Analyze the time complexity of your algorithm.

## Solution

The randomized algorithm for graph connectivity is as follows:

1. Choose a vertex v uniformly at random from V.

2. Perform a depth-first search (DFS) or breadth-first search (BFS) starting from v.

3. If the search visits all vertices in the graph, return "Connected"; otherwise, return "Not Connected".

Analysis:

If the graph is connected, the algorithm will always return "Connected" as the search starting from any vertex will visit all vertices in the graph.

If the graph is not connected, the algorithm might return "Connected" only if the randomly chosen vertex v belongs to the largest connected component. Let C be the largest connected component in the graph, and let |C| denote the number of vertices in C. The probability of choosing a vertex from C is |C| / n < 1.

Therefore, the probability of the algorithm returning "Connected" when the graph is not connected is less than 1. By repeating the algorithm k times and returning "Not Connected" if any of the k iterations return "Not Connected", we can reduce the error probability to (1)^k.

To achieve an error probability of at most 1/n, we set (1)^k ≤ 1/n and solve for k:

k ≥ log\_2(n)

Time complexity: The time complexity of a single iteration is O(n + m) for the DFS or BFS, where m is the number of edges in the graph. By repeating the algorithm O(log n) times, the overall time complexity becomes O((n + m) \* log n).