Task 1 [40 points]: Implementation of Power Iteration Algorithm.

Task 1(A) [25 points] Implement the power iteration algorithm in matrix form to calculate the rank vector r, without teleport, using the PageRank formulation: r(t+1) = M · r(t) The matrix M is an adjacency matrix representing nodes and edges from your downloaded dataset, with rows representing destination nodes and columns representing source nodes. This matrix is sparse2. Initialize r(0) = [1/N, . . . , 1/N]T . Let the stop criteria of your power iteration algorithm be ||r(t+1) – r(t)||1 < 0.02 (please note the stop criteria involves the L1 norm). Spider traps and dead ends are not considered in this first task.

Task 1(B) [15 points] Run your code on the Berkeley-Stanford web data to calculate the rank score for all the nodes. Report: (1) The running time of your power iteration algorithm; (2) The number of iterations needed to stop; (3) The IDs and scores of the top-10 ranked nodes.

In [16]:
```
import numpy as np
import time
from scipy.sparse import csc_matrix
# Read data and create sparse adjacency matrix
edges = []
with open("web-BerkStan-final.txt", "r") as file:
    for line in file:
        src, dest = map(int, line.strip().split())
        edges.append((src, dest))
nodes = sorted(list(set(src for src, _ in edges).union(dest for _, dest in e
# N is the total number of nodes
N = len(nodes)
print("Node number: ",N)
mapping = {node: i for i, node in enumerate(nodes)}
data = [1] * len(edges)
rows = [mapping[src] for src, _ in edges]
cols = [mapping[dest] for _, dest in edges]
adj_matrix = csc_matrix((data, (rows, cols)), shape=(len(nodes), len(nodes))
transposed_matrix = adj_matrix.T
column_sums = np.array(transposed_matrix.sum(axis=0)).flatten()
non_zero_cols = column_sums != 0

# Create a new CSC matrix with all zeros
normalizing_matrix = csc_matrix((N, N), dtype=np.float64)

# Set the non-zero column sums to their reciprocal values
normalizing_matrix[non_zero_cols, non_zero_cols] = 1 / column_sums[non_zero_

# Normalize the transposed_matrix by multiplying it with the normalizing_mat
transposed_matrix = transposed_matrix.dot(normalizing_matrix)

# Power iteration
def power_iteration(M,N, tol=0.02, max_iter=1000):
    # rank vector 0
    r_prev = np.ones([N,1]) / N
    convergence_history = []
    for i in range(max_iter):
        r_next  = M.dot(r_prev)

        diff = np.linalg.norm(r_next - r_prev, ord=1)
        convergence_history.append(diff)

        if diff < tol:
```

```
            return r_next, convergence_history, i+1

        r_prev = r_next

    return r_prev, convergence_history, i+1

# run the power iteration and time it
start_time = time.time()
result_vector, convergence, iterations = power_iteration(transposed_matrix,N
end_time = time.time()

# Flatten the result_vector before getting top indices
flattened_vector = result_vector.flatten()

#calculate the time
runtime = end_time - start_time
print("Time: ",runtime)

# Get top 10 pages' ID and value using the flattened vector
top_10_indices_flattened = np.argsort(flattened_vector)[-10:][::-1]
top_10_scores_flattened = flattened_vector[top_10_indices_flattened]
top_10_ids_flattened = [list(mapping.keys())[list(mapping.values()).index(i)

print("Iteration number: ",iterations)
print("Top-10 ranked nodes:")

for id, score in zip(top_10_ids_flattened, top_10_scores_flattened):
    print("ID:", id, "| Score:", score)
```

```
Node number:  685230
Time:  4.553022861480713
Iteration number:  501
Top-10 ranked nodes:
ID: 49175 | Score: 0.006557865729524351
ID: 50301 | Score: 0.005644215480665075
ID: 316711 | Score: 0.004668535815546852
ID: 590181 | Score: 0.003338668844440769
ID: 50306 | Score: 0.0028003062603820196
ID: 50307 | Score: 0.0028003062603820196
ID: 446912 | Score: 0.0025982933774186365
ID: 66243 | Score: 0.0022288212078001753
ID: 68948 | Score: 0.0022271931820592424
ID: 68947 | Score: 0.0022073080058437485
```

Task 2(A) Calculate and report the number of dead-end nodes in your matrix M.

In [18]:

```
row_sums = np.array(transposed_matrix.sum(axis=0)).flatten()
dangling_nodes = np.where(row_sums == 0)[0]
num_dead_ends = len(dangling_nodes)
print("Number of dead-end nodes:", num_dead_ends)
```

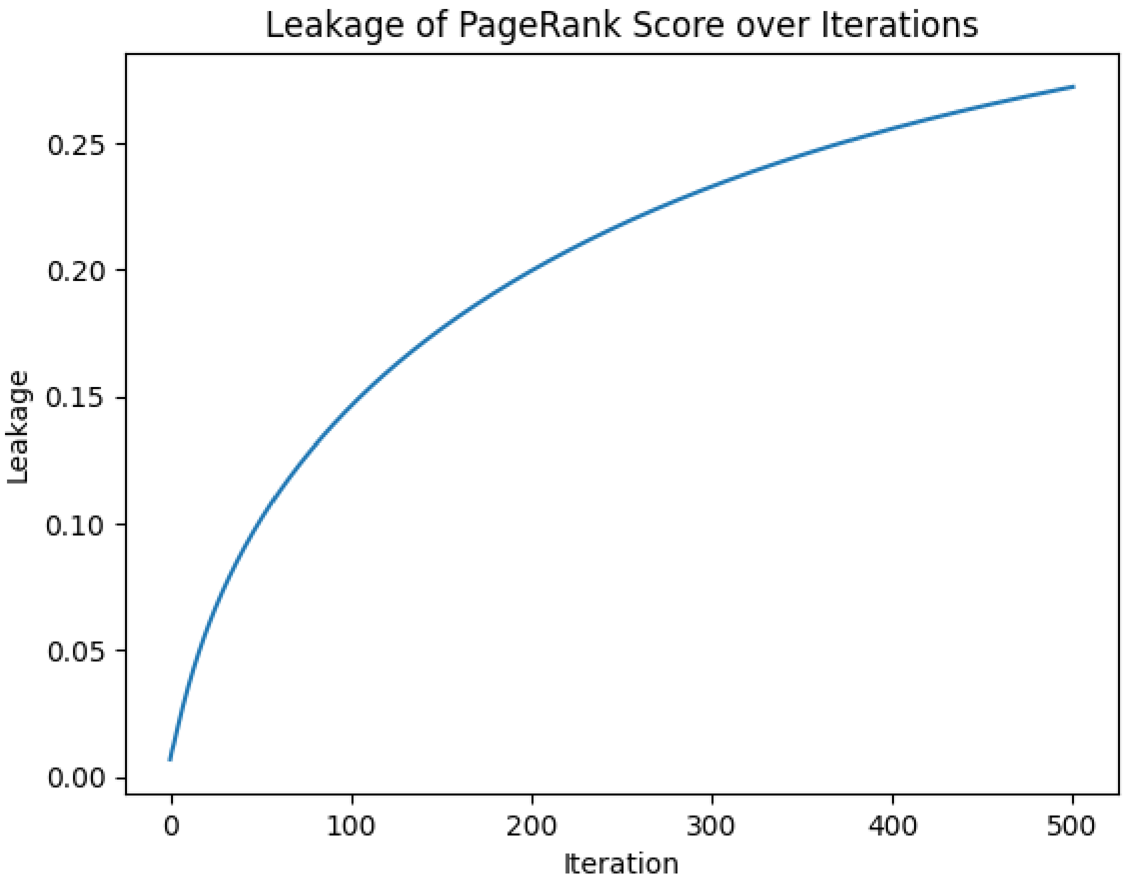
```
Number of dead-end nodes: 4744
```

Task 2(B) Calculate the leaked PageRank score in each iteration of Task 1(B)

In [11]:

```
def power_iteration_with_leakage(M, N, tol=0.02, max_iter=1000):
    r_prev = np.ones([N,1]) / N
    convergence_history = []
    leakage_history = []

    for i in range(max_iter):
        r_next = M.dot(r_prev)
        # Calculate leakage
        leakage = 1.0 - np.sum(r_next)
```

```
        leakage_history.append(leakage)

        diff = np.linalg.norm(r_next - r_prev, ord=1)
        convergence_history.append(diff)

        if diff < tol:
            return r_next, convergence_history, leakage_history, i+1

        r_prev = r_next

    return r_prev, convergence_history, leakage_history, i+1

# Run the function
result_vector_leakage, convergence_leakage, leakage, iterations_leakage = pc
import matplotlib.pyplot as plt

plt.plot(leakage)
plt.title('Leakage of PageRank Score over Iterations')
plt.xlabel('Iteration')
plt.ylabel('Leakage')
plt.show()
```



In the PageRank algorithm, leakage refers to the loss of PageRank scores due to "dead ends" (nodes with no outgoing links) or "traps" (nodes whose outgoing links all point back to themselves or other traps). When a PageRank vector is multiplied by the transition matrix, these "dead ends" and "traps" prevent some of the PageRank scores from being passed on to other nodes, thereby reducing the overall sum of the PageRank scores in the system.

In a simple PageRank model that doesn't include random jumps (or a "damping factor"), this kind of leakage accumulates over the course of the iterations. This means that the leakage may be relatively small initially, but as the number of iterations increases, this leakage tends to grow larger and stabilize. This chart prove this fact.

Task3 [50 points]: Implementation of Power Iteration with Teleport. Task 3(A): Extend your PageRank code to handle both spider traps and dead ends using the idea of teleport. In this task, your implementation will allow to teleport randomly to any node. Code the PageRank with teleport formulation that, using the sparse matrix M, for each iteration works in three steps

In [12]:
```
def power_iteration_with_teleport(M, N, beta=0.9, tol=0.02, max_iter=1000):
    # Initialize r
    r_prev = np.ones([N, 1]) / N
    convergence_history = []

    for i in range(max_iter):
        # Step 1: calculate r_new
        r_new = beta * M.dot(r_prev)

        # Step 2: calculate S
        S = np.sum(r_new)

        # Step 3: update r_new with teleport
        r_new += (1 - S) / N

        # Check convergence
        diff = np.linalg.norm(r_new - r_prev, ord=1)
        convergence_history.append(diff)
        if diff < tol:
            return r_new, convergence_history,i+1

        r_prev = r_new

    return r_prev, convergence_history,i+1
```

Task 3(B) Run your code on the Berkeley-Stanford web data to calculate the rank score for all the nodes. Report: (1) The running time; (2) The number of iterations needed to stop; (3) The IDs and scores of the top-10 ranked nodes.

In [19]:
```
import time
import numpy as np
from scipy.sparse import csc_matrix, diags

from scipy.sparse import lil_matrix, csr_matrix
import numpy as np
# Initialize an empty list to hold edges
edges = []

# Read data from file and append to the edges list
with open("web-BerkStan-final.txt", "r") as file:
    for line in file:
        src, dest = map(int, line.strip().split())
        edges.append((src, dest))
nodes = sorted(list(set(src for src, _ in edges).union(dest for _, dest in e
# N is the number of nodes
N = len(nodes)
mapping = {node: i for i, node in enumerate(nodes)}
data = [1] * len(edges)
rows = [mapping[src] for src, _ in edges]
cols = [mapping[dest] for _, dest in edges]
```

```
adj_matrix = csc_matrix((data, (rows, cols)), shape=(len(nodes), len(nodes))
transposed_matrix = adj_matrix.T
column_sums = np.array(transposed_matrix.sum(axis=0)).flatten()
non_zero_cols = column_sums != 0

# Create a new CSC matrix with all zeros
normalizing_matrix = csc_matrix((N, N), dtype=np.float64)

# Set the non-zero column sums to their reciprocal values
normalizing_matrix[non_zero_cols, non_zero_cols] = 1 / column_sums[non_zero_

# Normalize the transposed_matrix by multiplying it with the normalizing_mat
transposed_matrix = transposed_matrix.dot(normalizing_matrix)

# run the power iteration and time it
start_time = time.time()
result_vector_teleport, iterations_teleport, iterations_teleport = power_ite
end_time = time.time()
# Flatten the result_vector before getting top indices
flattened_vector_teleport = result_vector_teleport.flatten()
# Calculate the time
runtime = end_time - start_time
print("Time: ",runtime)
# Get top 10 pages' ID and value using the flattened vector
top_10_indices_flattened = np.argsort(flattened_vector_teleport)[-10:][::-1]
top_10_scores_flattened = flattened_vector_teleport[top_10_indices_flattened
top_10_ids_flattened = [list(mapping.keys())[list(mapping.values()).index(i)

print("Iteration number: ",iterations_teleport)
print("Top-10 ranked nodes:")

for id, score in zip(top_10_ids_flattened, top_10_scores_flattened):
    print("ID:", id, "| Score:", score)
```

```
Time:  0.12322306632995605
Iteration number:  12
Top-10 ranked nodes:
ID: 272918 | Score: 0.009991150893727348
ID: 438237 | Score: 0.007287766055015052
ID: 210375 | Score: 0.004514288537947627
ID: 210304 | Score: 0.004406323191149426
ID: 601655 | Score: 0.003994730583009043
ID: 571447 | Score: 0.003662503247658825
ID: 316791 | Score: 0.0030866844561316226
ID: 571446 | Score: 0.0023195258104476307
ID: 319208 | Score: 0.0022364853338768913
ID: 184093 | Score: 0.002231966754764037
```

Task 3(C) Vary the teleport probability β with numbers in the set: {1, 0.9, 0.8, 0.7, 0.6}. Report the number of iterations needed to stop for each β. Explain, in words, your findings from this experiment.

In [14]:
```
betas = [1, 0.9, 0.8, 0.7, 0.6]
iterations_for_betas = []

for beta in betas:
    _, _, iterations = power_iteration_with_teleport(transposed_matrix, N, b
    iterations_for_betas.append(iterations)

# Print the results
for beta, iterations in zip(betas, iterations_for_betas):
    print(f"For beta = {beta}, number of iterations = {iterations}")
```

```
For beta = 1, number of iterations = 520
For beta = 0.9, number of iterations = 12
For beta = 0.8, number of iterations = 8
For beta = 0.7, number of iterations = 6
For beta = 0.6, number of iterations = 5
```

My finds:

In the context of the PageRank algorithm, the variable β represents the "damping factor," which is the probability of following an outgoing edge from the current node during the random walk. When β = 1, the random walk strictly follows the links and does not include any random jumps. On the other hand, when β is less than 1, random jumps are introduced, meaning there's a chance of jumping to any random node instead of following the links.

1. When β the number of iterations needed for the algorithm to converge is 520, which is substantially high. This may be because there are "dead-ends" or "traps" in the graph that can cause PageRank leakage, making it harder for the algorithm to converge.

2. As β decreases from 1 to 0.6, the number of iterations needed for convergence dramatically drops. For β = 0.9 , it's just 12 iterations, and it goes as low as 5 iterations for β = 0.6 . This suggests that introducing random jumps (teleportation) makes the algorithm converge much faster.

The fast convergence for lower β values indicates that the introduction of random jumps effectively mitigates problems such as PageRank leakage, which helps in faster and more stable convergence. It also emphasizes the importance of including the damping factor in the PageRank algorithm for practical and efficient computation.

Additional experimentation：

It reveals that using random jumps effectively mitigates the leakage. Based on the results, the leakage is indeed very minimal.

In [20]:
```
def power_iteration_with_teleport_leak(M, N, beta=0.9, tol=0.02, max_iter=10
    # Initialize r
    r_prev = np.ones([N, 1]) / N
    convergence_history = []
    leakage_history = []

    for i in range(max_iter):
        # Step 1: calculate r_new
        r_new = beta * M.dot(r_prev)

        # Step 2: calculate S
        S = np.sum(r_new)

        # Step 3: update r_new with teleport
        r_new += (1 - S) / N

        leakage = 1.0 - np.sum(r_new)
        leakage_history.append(leakage)
```

```
        # Check convergence
        diff = np.linalg.norm(r_new - r_prev, ord=1)
        convergence_history.append(diff)
        if diff < tol:
            return r_new, convergence_history, leakage_history, i+1

        r_prev = r_new

    return r_prev, convergence_history, leakage_history, i+1

# Run the function
result_vector_leakage, convergence_leakage, leakage, iterations_leakage = pc
# Plotting
import matplotlib.pyplot as plt

plt.plot(leakage)
plt.title('Leakage of PageRank Score over Iterations')
plt.xlabel('Iteration')
plt.ylabel('Leakage')
plt.show()
```