PHÂN TÍCH VÀ THIẾT KẾ THUẬT TOÁN

TUẦN 5 – BRUTE FORCE

**Practice problems: Brute-force assignment**

**Đề bài**: A graph is said to be bipartite if all its vertices can be partitioned into two disjoint subsets X and Y so that every edge connects a vertex in X with a vertex in Y. (One can also say that a graph is bipartite if its vertices can be colored in two colors so that every edge has its vertices colored in different colors; such graphs are also called 2-colorable.)

1. Design a DFS-based algorithm for checking whether a graph is bipartite.

2. Design a BFS-based algorithm for checking whether a graph is bipartite.

**Trả lời:**

# DFS-based algorithm for checking bipartiteness:

The algorithm can be implemented using a modified depth-first search (DFS) traversal of the graph. Here's the step-by-step algorithm:

1. Start by initializing an empty set called 'visited' to keep track of visited vertices and a dictionary called 'color' to store the color assignment for each vertex. Initially, color[vertex] = None for all vertices.
2. For each unvisited vertex v in the graph, perform the following steps:
   * Call a helper function, DFS(v, 0), with v as the starting vertex and 0 as the color for v.
   * If DFS(v, 0) returns False, the graph is not bipartite, and the algorithm can terminate.
   * If DFS(v, 0) returns True, continue with the next unvisited vertex.
3. The DFS function takes a vertex v and a color c as input and performs the following steps:
   * Mark vertex v as visited by adding it to the 'visited' set.
   * Assign color c to vertex v by setting color[v] = c.
   * For each neighbor u of v, do the following:
     + If u is not visited, recursively call DFS(u, 1 - c) with u as the vertex and 1 - c as the color. Here, 1 - c represents the opposite color of c.
     + If u is visited and color[u] is the same as color[v], return False, indicating that the graph is not bipartite.
   * If all the neighbors are visited and the algorithm reaches this point, return True, indicating that the graph is bipartite.
4. After completing the DFS traversal for all unvisited vertices and none of the DFS calls return False, the algorithm can conclude that the graph is bipartite.

The time complexity of this algorithm is O(V + E), where V is the number of vertices and E is the number of edges in the graph.

# BFS-based algorithm for checking bipartiteness:

The algorithm can be implemented using a modified breadth-first search (BFS) traversal of the graph. Here's the step-by-step algorithm:

1. Start by initializing an empty set called 'visited' to keep track of visited vertices and a dictionary called 'color' to store the color assignment for each vertex. Initially, color[vertex] = None for all vertices.
2. For each unvisited vertex v in the graph, perform the following steps:
   * Enqueue v into a queue data structure.
   * Assign color 0 to vertex v by setting color[v] = 0.
   * Mark vertex v as visited by adding it to the 'visited' set.
   * While the queue is not empty, do the following:
     + Dequeue a vertex u from the front of the queue.
     + For each neighbor n of u, do the following:
       - If n is not visited, enqueue n into the queue.
       - If n is visited and color[n] is the same as color[u], return False, indicating that the graph is not bipartite.
       - If n is visited and color[n] is None, assign the opposite color of color[u] to n by setting color[n] = 1 - color[u].
     + If all the neighbors are visited and the algorithm reaches this point, return True, indicating that the graph is bipartite.
3. After completing the BFS traversal for all unvisited vertices and none of the BFS iterations return False, the algorithm can conclude that the graph is bipartite.

The time complexity of this algorithm is also O(V + E), where V is the number of vertices and E is the number of edges in the graph.