Đại học quốc gia TP.HCM
Trường đại học công nghệ thông tin

Môn học: Phân tích và thiết kế thuật toán
CS112.N21.KHTN

Bài tập Brute Force

Sinh viên thực hiện:
Trương Thanh Minh - 21520064
Lê Châu Anh - 21521821

Mục lục

1 Đề bài **2**

2 Solution **2**

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

2.2 Design a BFS-based algorithm for checking whether a graph is bipartite . . 3

1 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.)

2 Solution

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

- **Below is the algorithm to check for bipartiteness of a graph:**
  - Use a $color[]$ array which stores 0 or 1 for every node which denotes opposite colors.
  - Call the function DFS from any node.
  - If the node u has not been visited previously, then assign $!color[v]$ to $color[u]$ and call DFS again to visit nodes connected to u.
  - If at any point, $color[u]$ is equal to $color[v]$, then the node is not bipartite.

- **Below is the pseudocode of the above approach:**

```
bool isBipartite(vector<int> adj[], int v, vector<bool>& visited,
    vector<int>& color)
    for (u: adj[v])
        # if vertex u is not explored before
        if visited[u] == false
            # mark present vertices as visited
            visited[u] = true

            # mark its color opposite to its parent
            color[u] = !color[v]

            # if the subtree rooted at vertex v is not bipartite
            if (!isBipartite(adj, u, visited, color)
                return false

        # if 2 adjacent are colored with same color then the graph is
            not bipartite
        else if color[u] == color[v]
            return false
    return true
```

- **Complexity Analysis**:
  - Time Complexity: $O(N)$

– Space Complexity: $O(N)$

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

- **Below is the algorithm to check for bipartiteness of a graph:**
  - Assign RED color to the source vertex (putting into set U).
  - Color all the neighbors with BLUE color (putting into set V).
  - Color all neighbor's neighbor with RED color (putting into set U).
  - This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where $m = 2$.
  - This way, assign color to all vertices such that it satisfies all the constraints of m way coloring problem where $m = 2$.

- **Below is the pseudocode of the above approach:**

```
bool isBipartite(int G[][V], int src)

    # Create a color array to store colors assigned to all vertices.
       Vertex number is used as index in this array. The value -1 of
       colorArr[i] is used to undicate that no color is assigned to
       vertex 'i'.
    fill(colorArr, colorArr+n+1, -1)
    colorArr[src] = 1

    # Create a queue of vertex numbers and enqueue source vertex for
       BFS traversal
    queue <int> q
    q.push(src)

    # Run while there are vertices in queue (similar to BFS)

    while q is not empty:
        u = q.font()
        q.pop()

        # Return false if there is a self-loop
        if G[u][u] == 1
            return false

        # Find all non-colored adjacent vertices
        for (int v = 0; v < V; ++v)
            # An edge form u to v exists and destination v is not
               colored
            if G[u][v] && colorArr[v] == -1
                # Assign alternate color to this adjacent v of u
```

```
                colorArr[v] = 1 - colorArr[u]
                q.push(v)

            # An edge from u to v exists and destination v is colored
                with same color as u
            else if G[u][v] && colorArr[v] == colorArr[u]
                return false
    # If we reach here, them all adjacent vertices can be colored
        with alternate color
    return true
```

- **Complexity Analysis**:
  - Time Complexity: $O(V+E)$
  - Space Complexity: $O(V)$