Description

Given an undirected connected graph, check if the graph contains a cycle.

> bool is_graph_cyclic(const vector<vector>& adj_list);
>
> You can assume that if u and v are adjacent, then adj_list[u] will contain v and
> adj_list[v] will contain u. Also, u will not appear in adj_list[u].

Solutions

- 深度优先DFS, time O(|V|+|E|). 用了递归没用stack

  When we do a DFS from any vertex v in an undirected graph, we may encounter **back-edge** that points to one of the ancestors of current vertex v in the DFS tree. Each "back edge" defines a cycle in an undirected graph. If the back edge is `x -> y` then since y is ancestor of node x, we have a path from y to x. So we can say that we have a path `y ~~ x ~ y` that forms a cycle. (Here `~~` represents one more more edges in the path and `~` represents a direct edge).

```
bool isCyclic(int u, bool visited[], int prev, const
vector<vector<int>> &adj_list) {
    visited[u] = true;  // mark the current node to be visited
    for (auto &i: adj_list[u]) {
        if (!visited[i]) {
            if (isCyclic(i, visited, u, adj_list))
                return true;
        } else if (i != prev)
            return true;
    }
    return false;
}  // If we use stack in is_graph_cyclic, we don't need isCyclic
function and recursion


bool is_graph_cyclic(const vector<vector<int>> &adj_list) {
    bool visited[adj_list.size()];  // whether visit or not
    for (auto &i: visited)
        i = false;
    for (int u = 0; u < adj_list.size(); u++) {  // DFS
        if (!visited[u])
            if (isCyclic(u, visited, -1, adj_list))
                return true;
    }
    return false;
}
```

- 广度优先BFS, time O(|V|+|E|). 用了queue没用递归

  When we do a BFS from any vertex v in an undirected graph, we may encounter **cross-edge** that points to a previously discovered vertex that is neither an ancestor nor a descendant of current vertex. Each "cross edge" defines a cycle in an undirected graph. If the cross edge is `x -> y` then since y is already discovered, we have a path from v to y (or from y to v since the graph is undirected) where v is the starting vertex of BFS. So we can say that we have a path `v ~~ x ~ y ~~ v`. that forms a cycle. (Here `~~` represents one more more edges in the path and `~` represents a direct edge).

```
bool isCyclic(const vector<vector<int>> &adj, int i,
vector<bool> &visited) {
    vector<int> parent(adj.size(), -1);
    queue<int> q;  // queue for BFS
    visited[i] = true;
    q.push(i);
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (auto v: adj[u]) {
            if (!visited[v]) {
                visited[v] = true;
                q.push(v);
                parent[v] = u;
            } else if (parent[u] != v)
                return true;
        }
    }
    return false;
}


bool is_graph_cyclic(const vector<vector<int>> &adj_list) {
    vector<bool> visited(adj_list.size(), false);
    for (int i = 0; i < adj_list.size(); i++)
        if (!visited[i] && isCyclic(adj_list, i, visited))
            return true;
    return false;
}
```

- Disjoint set/ Union find

  对于别的graph representation，这种方法好；在当前graph representation下，此法一般

```
// A utility function to find the subset of an element i
int find(vector<int> &parent, int i) {
    if (parent[i] == -1)
        return i;
```

```
    return find(parent, parent[i]);
}


// A utility function to do union of two subsets
void Union(vector<int> &parent, int x, int y) {
    parent[x] = y;
}


bool is_graph_cyclic(const vector<vector<int>> &adj_list) {
    vector<int> parent(adj_list.size(), -1);
    vector<bool> visited(adj_list.size(), false);
    // Iterate through all edges of graph, find subset of
    // both vertices of every edge, if both subsets are
    // same, then there is cycle in graph.
    for (int i = 0; i < adj_list.size(); ++i) {
        visited[i] = false;
        for (auto u: adj_list[i]) {
            if (!visited[u]) {
                int x = find(parent, i);
                int y = find(parent, u);
                if (x == y)
                    return true;
                Union(parent, x, y);
            }
        }
    }
    return false;
}
```