Rick Sullivan
Professor Tran
Algorithms-COEN 179
7 May 2013

Homework #3

1.
```
class graph
{
public:

...

int bfs1(int start, std::vector<int> & parent)
{
    bool color[n()];
    std::queue<int> q;
    int ans = 0;

    // initialization
    parent[start] = start;
    color[start] = true;
    q.push(start);

    while (!q.empty())
    {
        int u = q.front();
        q.pop();
        ++ans;
        std::cout << u << std::endl;

        for (const_iterator x = data[u].begin(); x != data[u].end(); ++x)
            if (parent[*x] == -1)
            {
                parent[*x] = u;
                color[*x] = !color[u];
                q.push(*x);
            }
            else
            {
                if (parent[u] != *x)
                    acyclic = false;
                if (color[u] == color[*x])
                    bipartite = false;
            }
    }

    return ans;
}

...

bool is_bipartite()
```

```
{
    bipartite = true;

    std::vector<int> parent(n(), -1);
    for (int i = 0; i < n(); ++i)
        if (parent[i] == -1)
            bfs1(i, parent);

    return bipartite;
}

private:
    std::vector<std::list<int> > data;
    bool acyclic;
    bool bipartite;

}; // /graph
```

2.
```
void dfs1(int start, std::vector<int>& parent, std::vector<int>& first,
            std::vector<int>& last, int& time, std::vector<bool>& color)
{
    //initialize
    first[start] = time++;

    for (const_iterator x = data[start].begin(); x != data[start].end(); ++x)
    {
        if (parent[*x] == -1)
        {
            parent[*x] = start;
            color[*x] = !color[start];
            dfs1(*x, parent, first, last, time, color);
        }
        else if (color[*x] == color[start])
                bipartite = false;
    }

    last[start] = time++;
}

void dfs()
{
    std::vector<int> parent(n(), -1);
    std::vector<int> first(n(), -1);
    std::vector<int> last(n(), -1);
    std::vector<bool> color(n());

    int time = 0;
    int ncc = 0;

    for (int start = 0; start < n(); ++start)
    {
        if (parent[start] == -1)
        {
            parent[start] = start;
```

```
                color[start] = true;
                dfs1(start, parent, first, last, time, color);
                ++ncc;
            }
        }
    }

    bool is_bipartite()
    {
        bipartite = true;
        dfs();

        return bipartite;
    }
```

3.
```
    //Returns the length of the shortest cycle.
    //if the graph is acyclic, returns -1 for undefined.
    int girth()
    {
        if (is_acyclic())
            return -1;

        std::vector<int> parent(n(), -1);
        int minC = 2147483647;

        for (int i = 0; i < n(); ++i)
        {
            for (const_iterator j = data[i].begin(); j != data[i].end(); ++j)
            {
                remove_edge(i, *j);
                int d = distance(i, *j);
                if(d != -1)
                    minC = std::min(minC, d);
                add_edge(i, *j);
            }
        }
        return minC;
    }

    //Uses DFS to track the shortest distance between two vertices.
    //Returns -1 if no path exists.
    int distance(int start, int end)
    {
        std::vector<int> parent(n(), -1);
        std::queue<int> q;
        int d = 0;

        // initialization
        parent[start] = start;
        q.push(start);

        while (!q.empty())
        {
            int u = q.front();
```

```
        q.pop();
        if(u == end)
            return d;
        ++d;
        std::cout << u << std::endl;

        for (const_iterator x = data[u].begin(); x != data[u].end(); ++x)
            if (parent[*x] == -1)
            {
                parent[*x] = u;
                q.push(*x);
            }
    }

    return -1;
}

void remove_edge(int from, int to){
    assert(is_edge(from, to));
    data[from].remove(to);
    data[to].remove(from);
}
```