DFS Applications

Recall from DFS
typedef struct {
    int *order;    // The order the vertex was visited
    int *spanTree; // The parent of the vertex
    int count;     // internal use
} Search;

Properties of a DFS Forest
• DFS is a preorder traversal of graph
• To find more features of graphs, classify edges as one of four types
• Order array is a pre-order numbering of nodes
• Introduce spanTree array - assigned in dfsR

Modified DFS
// Purpose: Helper function for graphSearch performing recursive calls
// input: graph to be searched, the search object of storing results,
//        and the current edge being traversed
// output: none
void dfsR (Graph *theGraph, Search *info, Edge *curEdge) {
    int nextStep, curPos = curEdge->to;
    Edge newEdge;
    info->order[curPos] = info->count++;  
    info->spanTree[curPos] = curEdge->from;
    for (nextStep = 0; nextStep < theGraph->vertices; nextStep++) {
        if (theGraph->adjMatrix[curPos][nextStep] != 0) {
            if (info->order[nextStep] == NOT_VISITED)
                dfsR(theGraph, info, createEdge(curPos, nextStep, &newEdge));
        }
    }
}

Connected Components
• What would you do to identify connected components?
**Link Types**

- **tree link**
  - if `spanTree[toVertex] == fromVertex`
    - This is the edge the recursive call is made on

- **parent link**
  - if `spanTree[fromVertex] == toVertex`

- **back link**
  - if `order[fromVertex] > order[toVertex]`

- **down link**
  - if `order[fromVertex] < order[toVertex]`

*Each edge is associated with two types - depending on the vertex (mirrors the adjacency lists)*
Cycle Detection

- Does a graph have a cycle?
- What type of link indicates a cycle?

How would you write the algorithm?

```cpp
bool cycleR (Graph *theGraph, Search *info, Edge *curEdge) {
    int nextStep, curPos = curEdge->to;
    Edge newEdge;
    info->order[curPos] = info->count++;
    info->spanTree[curPos] = curEdge->from;

    for (nextStep = 0; nextStep < theGraph->vertices; nextStep++) {
        if (theGraph->adjMatrix[curPos][nextStep] != 0)
            if (info->order[nextStep] == NOT_VISITED)
                cycleR(theGraph, info, createEdge(curPos,nextStep, &newEdge));
    }
}
```

Spanning Forest

- Given a connected graph with V vertices, find a set of V-1 edges that connects the vertices.
  - If the graph has C components, find a spanning forest (with V-C edges)
- Why only V-C edges?

Two colorability

- Is there a way to assign one of two colors to each vertex of a graph such that no edge connects two vertices of the same color?
  - Another name is bipartitiness
- Property: any graph with an odd length cycle is not two colorable

![Graph with odd cycle]
// Globals for determining two-colorability
#define RED 0
#define BLUE 1
#define NO_COLOR -1

// Purpose: determine if the given graph is 2-colorable
// Input: the graph
// Output: true if it is 2-colorable and otherwise false
bool graphTwoColor(Graph *theGraph) {
    int vertex;
    int *color = (int *) calloc(sizeof(int), theGraph->vertices);
    CHECKP(color);
    for (vertex = 0; vertex < theGraph->vertices; vertex++)
        color[vertex] = NO_COLOR;
    for (vertex = 0; vertex < theGraph->vertices; vertex++)
        if (color[vertex] == NO_COLOR)
            if (!dfsRcolor(theGraph, color, vertex, RED)) {
                free(color);
                return false;
            }
    free(color);
    return true;
}

// Purpose: Helper function using depth first search to assign colors
//          to vertices
// Input: the graph, a colorArray initialized to NO_COLOR, the current
//        vertex, the last assigned color
// Output: True if colors sucessfully assigned to all reachable
//        vertices and false otherwise
bool dfsRcolor(Graph *theGraph, int *colorArray, int fromVertex,
                int curColor) {
    int toVertex;
    colorArray[fromVertex] = 1-curmColor;
    for (toVertex = 0; toVertex < theGraph->vertices; toVertex++) {
        if (theGraph->adjMatrix[fromVertex][toVertex] == 1) {
            if (!dfsRcolor(theGraph, colorArray, toVertex, curColor))
                return false;
        }
    }
    return true;
}
**Statement**
A graph is two-colorable if and only if it contains no odd cycle.

Prove that the statement is true. *Hint:* Prove by induction that `graphTwoColor` determines whether or not any given graph is two-colorable.

**Base Case**

**Inductive Hypothesis**

**Inductive Step**