Computing Science 204 (B1)
2-Colouring a Graph
Friday, August 11

Algorithm for 2-Colouring a Graph

An undirected graph $G=(V,E)$ is said to be *2-colourable* if and only if it is possible to colour the vertices of G so that no adjacent vertices have the same colour. A *cycle* in G is a sequence

$$u_1, e_1, u_2, e_2, \cdots, u_k, e_k, u_{k+1},$$

where $k \geq 0$, each u_i is a vertex, and each e_i is the edge $\{u_i, u_{i+1}\}$, such that the vertices $u_1, u_2, \cdots, u_k$ are distinct, the edges $e_1, e_2, \cdots, e_k$ are distinct, and $u_{k+1} = u_1$. The *length* of the cycle is k, the number of edges in it. There is a well known theorem due to *König* which states that a graph is 2-colourable if and only if it has no cycles of odd length.

Given a graph G with no odd cycles, there is a simple algorithm for finding a 2-colouring of G. First we may assume that G is connected, since we could colour each component separately. Now choose an arbitrary vertex x. Colour x blue. Next colour all neighbours of x white. For each of these neighbours, colour its uncoloured neighbours blue. Continue in this way until all the vertices of G are coloured.

To implement this algorithm formally, vertices which have been coloured, but whose neighbours have not yet been coloured, are saved in an ordered list called a *queue*. At each stage of the algorithm, we find the first vertex y in the queue and remove it from the queue. We find its uncoloured neighbours and colour them the opposite colour of y. We then add these neighbours to the end of the queue. We continue until all vertices have been coloured. The algorithm is stated formally below, here Q is the queue.

2Colour
Input: A graph $G=(V,E)$ which is connected and has no cycles of odd length.
Output: A colouring of the vertices of G using the two colours blue and white.

(i) Initially, all vertices in V are uncoloured and Q is empty.

(ii) Choose an x in V, colour x blue, and put x in Q.

(iii) Let y be the first vertex in Q. Remove y from Q.

(iv) Find all uncoloured neighbours of y. Colour each with the colour opposite that used for y. Add them to the end of the queue Q in arbitrary order.

(v) If all vertices are coloured, stop. Otherwise, return to (iii).

We implement the above algorithm in C, the procedure takes as its input an undirected graph G which is connected and contains no odd cycles, and has as its output a valid 2-colouring for the graph G.

For each graph the required output is:

- A list of the vertices and edges (i.e. echo the input).
- If the graph is not 2-colourable, a message so indicating.
- If the graph is 2-colourable, a valid 2-colouring in the form of two lists labelled *Blue* and *White.*

The algorithm is implemented below, it uses the *Breadth First Search* algorithm implemented using a queue.

```
#include<stdio.h>
#define maxV 100

struct node
{
      int v;
      struct node *next;
} *z,*adj[maxV];

struct qnode
{
      int v;
      struct qnode *next;
} *qz;

struct queue
{
      struct qnode *head;
      struct qnode *tail;
} Q;

int V,E,bipartite = 1;
int colour[maxV],visited[maxV];

initarrays()
{
      int j;
      for (j = 0; j < maxV; j++)
      {
           colour[j] = 0;
           visited[j] = 0;
      }
}
```

```
void adjlist()
{
      int j,x,y;
      struct node *t;

      printf("Enter the number of Vertices and Edges:\n");
      scanf("%d %d",&V,&E);

      z = (struct node *) malloc(sizeof *z);
      z->v = 0;
      z->next = z;

      for (j = 0; j <= V; j++)
           adj[j] = z;

      printf("Enter the edges x y:\n");

      for (j = 1; j <= E; j++)
      {
           scanf("%d %d",&x,&y);

           t = (struct node *) malloc(sizeof *t);
           t->v = x;
           t->next = adj[y];
           adj[y] = t;

           t = (struct node *) malloc(sizeof *t);
           t->v = y;
           t->next = adj[x];
           adj[x] = t;
      }
}

void queueinit()
{
      struct qnode *head,*tail;

      qz = (struct qnode *) malloc(sizeof *qz);
      qz->next = qz;

      head = (struct qnode *) malloc(sizeof *head);
      head->next = qz;

      tail = (struct qnode *) malloc(sizeof *tail);
      tail->next = qz;

      Q.head = head;
      Q.tail = tail;
}
```

```
int queueempty()
{
      return (Q.head == qz && Q.tail == qz);
}

void enqueue(k)
int k;
{
      struct qnode *q;

      q = (struct qnode *)malloc(sizeof *q);
      q->v = k ;
      q->next = qz;

      if (Q.head == qz && Q.tail == qz)
      {
           Q.head = q;
           Q.tail = q;
      }
      else
      {
           Q.tail->next = q;
           Q.tail = q;
      }
}

int dequeue()
{
      int k;
      struct qnode *q;

      if (Q.head == qz && Q.tail == qz)
      {
           printf("Queue is empty!");
           return 0;
      }
      else
      {
           q = Q.head;
           Q.head = Q.head->next;

           if (Q.head == qz)
               Q.tail = qz;
           k = q->v;
           free(q);
           return k;
      }
}
```

```
void visitnode(k)
int k;
{
      int n;
      struct node *t;

      colour[k] = 1;
      enqueue(k);

      while (!queueempty())
      {
           n = dequeue();
           t = adj[n];

           while (t != z && bipartite)
           {
               if (visited[t->v] == 1)
               {
                   if (colour[t->v] == colour[n])
                       bipartite = 0;
               }
               else if (visited[t->v] == 0)
               {
                   visited[t->v] = 1;
                   if (colour[n] == 1)
                       colour[t->v] = 2;
                   if (colour[n] == 2)
                       colour[t->v] = 1;
                   enqueue(t->v);
               }
               t = t->next;
           }
      }
}

void breadthfirstsearch()
{
      int j;

      for (j = 1; j <= V; j++)
      {
           if (visited[j] == 0)
               visitnode(j);
      }
}
```

```
void printoutput()
{
      int j;
      struct node *p;

      for (j = 1; j <= V; j++)
      {
           printf("\nadj[%d] :  ",j);
           p = adj[j];
           while (p != z)
           {
               printf("%4d",p->v);
               p = p->next;
           }
           printf("\n");
      }
      printf("\n");

      if (bipartite)
      {
           printf("\nBLUE: ");

           for (j = 1; j <= V; j++)
           {
               if (colour[j] == 1)
                   printf("%3d",j);
           }
           printf("\n");

           printf("\nWHITE: ");

           for (j = 1; j <= V; j++)
           {
               if (colour[j] == 2)
                   printf("%3d",j);
           }
           printf("\n");

      }
      else if (!bipartite)
      {
           printf("\nTHE GRAPH IS NOT 2-COLOURABLE. \n");
      }
}

main()
{
      adjlist();
      initarrays();
      queueinit();
      breadthfirstsearch();
      printoutput();
}
```