A solution to Knuth's TALG brain teaser

James A. Muir
School of Computer Science
Carleton University
jamuir@scs.carleton.ca

Time-stamp: <05/09/30 00:49:10 jamuir>

1 Introduction

A "Problems Column" by Khuller appears in the inaugural edition of the journal TALG (ACM Transactions on Algorithms, Vol. 1, No. 1, July 2005, pp. 157–159). One of the contributions to the column, entitled "Searching Graphs", is a brain teaser by Knuth. Here we present a solution to Knuth's exercise.

If you think you might like to try solving the problem yourself, then please stop reading and get to it. If you have any comments or corrections on what you read here, then I would be glad to receive them.

2 The Problem

Here is the problem as it is displayed in the journal:

> A graph with n vertices $\{0, 1, ..., n-1\}$ and m edges can be specified by an array of $2m$ integers, where the edges join vertices $\{a[2k], a[2k+1]\}$ for $0 \le k < m$.
>
> Let v be a given vertex and assume that d is an integer array of size n. Find `<Statement1>` and `<Statement2>` such that the C-language subroutine `alldistances` shown below will set $d[u]$ to the distance from vertex v to vertex u, for $0 \le u < n$. (If u is unreachable from v, the "distance" between them is considered to be n.) Does the program run in linear time?

```
void alldistances(int m, int a[], int n, int d[], int v)
{
   register int i,j,k,l;
   int *b=(int*)malloc(n*sizeof(int));
   int *link=(int*)malloc(2*m*sizeof(int));
   for (j=0; j<n; j++) b[j]=-1, d[j]=n;
   for (k=0; k<m+m; k++) <Statement1>;
   d[v]=0, k=b[v], b[v]=-1, j=-1, l=1;
   while (k>=0) {
      v=a[k];
      if (b[v]>=0) {
         d[v]=l;
         for (i=b[v]; link[i]>=0; i=link[i]);
         <Statement2>;
      }
      k=link[k];
      if (k<0) l++, k=j, j=-1;
   }
}
```

> `<Statement1>` begins "`link[k]=`", and `<Statement2>` begins "`link[i]=`".

3 Solution

3.1 Statement1

```
link[k]=b[a[k^1]], b[a[k^1]]=k
```

In C, the expression `X^Y` evaluates to a bitwise xor of the values `X` and `Y`. If you don't like bitwise operations, then, for our purposes, the expression `k^1` could be replaced with `(k%2)?(k-1):(k+1)`. This last expression employs C's ternary operator.

3.2 Statement2

```
link[i]=j, j=b[v], b[v]=-1
```

3.3 Correctness

The arrays `b` and `link` are used to hold indices into the array `a`. Immediately after the "for" loop containing `<Statement1>` is executed, for any vertex u, we have that

`b[u]` is the largest index, `i`, into `a` such that the vertex `a[i]` is a neighbour of `u`; if no such index exists, then `b[u]=-1`.

`link[b[u]]` is the second largest index, `i`, into `a` such that the vertex `a[i]` is a neighbour of `u`; if no such index exists, then `link[b[u]]=-1`.

`link[link[b[u]]]` is the third largest index, `i`, into `a` such that the vertex `a[i]` is a neighbour of `u`; if no such index exists, then `link[link[b[u]]]=-1`.

... and so on.

So, the "for" loop containing `<Statement1>` essentially constructs an adjacency list for each vertex in the graph. With these adjacency lists at hand, it now becomes easier to see how the algorithm might implement a breadth-first search.

`<Statement2>` accomplishes the following. As the algorithm discovers all vertices at, say, level ℓ in the graph (i.e. all vertices, u, with $d(u,v) = \ell$), `<Statement2>` concatenates their adjacency lists together. To discover all vertices at level $\ell+1$, the algorithm then traverses this list. Some of the vertices in this list have been previously visited by the algorithm (they are the vertices, u, with $b[u] = -1$). The other vertices are correctly identified as being at level $\ell+1$.

3.4 Running Time

We need to determine if the program runs in $O(n+m)$ time.

In its initialization stage, the algorithm takes $O(n) + O(2m)$ time. However, after that, things get a bit more tricky. We bound the running time by considering the number of vertices that are visited in the remaining part of the algorithm.

For example, if the input is the 3-cycle, `[0,1, 1,2, 2,0]`, with source vertex $v = 0$, then the algorithm continues after the initialization stage by visiting 0. Then, we visit the vertices adjacent to 0 (i.e.

its neighbours), namely 2 and 1. When we visit vertex 2 we must also visit its neighbours, and the same goes for vertex 1. In all, the algorithm makes 11 visits to vertices. The visits are summarized as follows:

$v = 0$	$v = 2$, and neighbours 1, 0	$v = 0$
	$v = 1$, and neighbours 2, 0	$v = 1$
		$v = 0$
		$v = 2$

Counting visits in this way, if we consider the algorithm applied to the complete graph, K_n, where $m = n(n-1)/2$, we can determine the number of visits to be exactly $1+(n-1)n+(n-1)(n-1) = 2n^2-3n+2$.

For the complete graph, the adjacency lists are as large as possible, so this gives us an upper bound. Now,

$$1 + (n-1)n + (n-1)(n-1) = 1 + 2m + O(2m) = O(m).$$

The running time is now bounded as $O(n) + O(2m) + O(m) = O(n+m)$. So, yes, the running time of the program is linear.

```
void alldistances(int m, int a[], int n, int d[], int v)
{
   register int i,j,k,l;
   int *b=(int*)malloc(n*sizeof(int));
   int *link=(int*)malloc(2*m*sizeof(int));
   for (j=0; j<n; j++) b[j]=-1, d[j]=n;
   for (k=0; k<m+m; k++) link[k]=b[a[k^1]], b[a[k^1]]=k;
   d[v]=0, k=b[v], b[v]=-1, j=-1, l=1;
   while (k>=0) {
      v=a[k];
      if (b[v]>=0) {
         d[v]=l;
         for (i=b[v]; link[i]>=0; i=link[i]);
         link[i]=j, j=b[v], b[v]=-1;
      }
      k=link[k];
      if (k<0) l++, k=j, j=-1;
   }
}
```