LABORATORIO 04

Grafos: Listas de adyacencia

1. Ejercicios:

1.1. Considere el problema de decidir si un vértice v es aislado en un grafo G. ¿Cuánto tiempo consume la solución del problema?. Dé su respuesta en función del número de vértices del grafo.

Un vértice es aislado si no tiene arcos de entrada ni de salida, es decir, su grado de entrada y de salida son cero.

- **Grado de Salida:** Para determinar el grado de salida, solo se necesita verificar si la lista de adyacencia del vértice v está vacía (G->adj[v] == NULL). Esta es una operación de tiempo constante, O(1).
- **Grado de Entrada:** Para determinar el grado de entrada se debe recorrer las listas de adyacencia de todos los vértices del grafo para ver si alguno de ellos tiene un arco que apunte hacia v. En el peor de los casos esto implica revisar cada arco del grafo. Por lo tanto la complejidad es proporcional a la suma del número de vértices y arcos: O(V + A).

La complejidad total: $O(1) + O(V + A) = O(V + A)$

1.2. Escriba una función GRAPHindeg() que calcule el grado de entrada de un vértice v de un grafo G. Escribe una función GRAPHoutdeg() que calcule el grado de salida de v.

```
int GRAPHoutdeg(Graph G, vertex v) {
  int count = 0;
  for (link a = G->adj[v]; a != NULL; a = a->next) {
    count++;
  }
  return count;
}

int GRAPHindeg(Graph G, vertex v) {
  int count = 0;
  for (vertex i = 0; i < G->V; ++i) {
    for (link a = G->adj[i]; a != NULL; a = a->next) {
      if (a->w == v) {
        count++;
      }
    }
  }
  return count;
}
```

1.3. Consideremos el problema de decidir si dos vértices son adyacentes en un grafo G. ¿Cuánto tiempo se tarda en resolver este problema? De su respuesta en función del número de vértices y arcos del grafo

Para ver si un vértice w es adyacente a un vértice v debe de existir un arco v->w.

El tiempo de ejecución es proporcional a la longitud de la lista de adyacencia de v, que es su grado de salida. Por lo tanto, la complejidad es O(out-degree(v)). En el peor caso un vértice puede estar conectado a todos los demás, por lo que la complejidad sería $O(V)$.

1.4. Escribe una función GRAPHshow() que imprima todos los vértices adyacentes a v en una línea para cada vértice v del grafo G.

```
void GRAPHshow(Graph G) {
  printf("Grafo con %d vertices y %d arcos\n", G->V, G->A);
  for (vertex v = 0; v < G->V; ++v) {
    printf("%2d:", v);
    for (link a = G->adj[v]; a != NULL; a = a->next) {
      printf(" %2d", a->w);
    }
    printf("\n");
  }
}
```

1.5. Escribe una función GRAPHdestroy() que destruya la representación de un grafo G, liberando el espacio que la representación ocupa en memoria.

```
void GRAPHdestroy(Graph G) {
  if (G == NULL)
    return;
  for (vertex v = 0; v < G->V; ++v) {
    link current = G->adj[v];
    while (current != NULL) {
      link temp = current;
      current = current->next;
      free(temp);
    }
  }
  free(G->adj);
  free(G);
}
```

1.6. Eliminación de arcos. Escriba una función GRAPHremoveArc() que tome dos vértices v y w de un grafo G representado por listas de adyacencia y elimine el arco v-w de G.

```
void GRAPHremoveArc(Graph G, vertex v, vertex w) {
  link current = G->adj[v];
  link prev = NULL;

  while (current != NULL && current->w != w) {
    prev = current;
    current = current->next;
  }

  if (current == NULL)
    return;
```

```
  if (prev == NULL) {
    G->adj[v] = current->next;
  } else {
    prev->next = current->next;
  }
  free(current);
  G->A--;
}
```

1.7. ¿No dirigido? Escriba una función GRAPHundir() que decida si un grafo dado es no dirigido.

```
int GRAPHundir(Graph G) {
  for (vertex v = 0; v < G->V; ++v) {
    for (link a = G->adj[v]; a != NULL; a = a->next) {
      vertex w = a->w;
      int found = 0;

      for (link b = G->adj[w]; b != NULL; b = b->next) {
        if (b->w == v) {
          found = 1;
          break;
        }
      }
      if (!found) {
        return 0;
      }
    }
  }
  return 1;
}
```

1.8. Inserción de aristas. Escribe una función UGRAPHinsertEdge() que inserte una arista v-w en un grafo no dirigido G.

```
void UGRAPHinsertEdge(Graph G, vertex v, vertex w) {
  GRAPHinsertArc(G, v, w);
  GRAPHinsertArc(G, w, v);
}
```

1.9. Eliminación de aristas. Escribe una función UGRAPHremoveEdge() que elimine una arista dada v-w de un grafo no dirigido G.

```
void UGRAPHremoveEdge(Graph G, vertex v, vertex w) {
  GRAPHremoveArc(G, v, w);
  GRAPHremoveArc(G, w, v);
}
```

1.10. Grado Máximo. Escribe una función UGRAPHdegrees() que toma un grafo no dirigido y devuelve el grado máximo de un grafo.

```
```

```
int UGRAPHmaxDegree(Graph G) {
  int max_deg = 0;
  for (vertex v = 0; v < G->V; ++v) {
    int deg = GRAPHoutdeg(G, v);
    if (deg > max_deg) {
      max_deg = deg;
    }
  }
  return max_deg;
}
```

2. Ejecucion:

```
❯ ./main.out
Estado inicial de G1:
Grafo con 6 vertices y 7 arcos
 0:  5  1
 1:  5  0
 2:  4
 3:  1
 4:
 5:  3
Grado de salida del vertice 0: 2
Grado de entrada del vertice 1: 2
Grado de entrada del vertice 5: 2

Eliminar arco 1->5 de G1:
Grafo con 6 vertices y 6 arcos
 0:  5  1
 1:  0
 2:  4
 3:  1
 4:
 5:  3

Es G1 no-dirigido?: No

Estado inicial de G2:
Grafo con 4 vertices y 10 arcos
 0:  2  3  1
 1:  2  0
 2:  0  3  1
 3:  0  2

Es G2 no-dirigido?: Si

Grado maximo en G2: 3

Eliminar arista 0-2 de G2:
Grafo con 4 vertices y 8 arcos
 0:  3  1
 1:  2  0
 2:  3  1
 3:  0  2

Grado maximo en G2: 2
```

Figure 1: