Semillero de Programación

Pila, Cola, DFS y BFS

Ana Echavarría Juan Francisco Cardona

Universidad EAFIT

22 de febrero de 2013

Contenido

10055 - Hashmat the Brave Warrior

```
1  int main(){
2      long long a, b;
3      while(cin >> a >> b){
4          long long diff = abs(b - a);
5          cout << diff << endl;
6      }
7       return 0;
8  }
```

100 - The 3n + 1 problem

```
1  int main(){
2      int i, j;
3      while(cin >> i >> j){
4          cout << i << " " << j << " ";
5          if (i > j) swap(i, j);
6          int best = 0;
7          for (int k = i; k <= j; k++){
8              int count = 1;
9              int n = k;
10             while (n > 1){
11                 if (n % 2 == 0) n /= 2;
12                 else n = 3 * n + 1;
13                 count++;
14             }
15             best = max(count, best);
16         }
17         cout << best << endl;
18     }
19     return 0;
20 }
```

573 - The Snail

```
1  int main(){
2     int H, U, D, F;
3     while (cin >> H >> U >> D >> F){
4        if (H == 0) break;
5        int day = 0;
6        double height = 0.0;
7        double climb = U;
8        double fatigue = (1.0 * U * F) / 100;
9        while (height >= 0){
10          height += climb;
11          day++;
12          if (height > H) break;
13          height -= D;
14          climb = max(0.0, climb - fatigue);
15       }
16       if (height >= H) printf("success on day %d\n", day);
17       else printf("failure on day %d\n", day);
18    }
19     return 0;
20 }
```

483 - Word Scramble

```
1  vector <string> v;
2  int main(){
3     string line;
4     while (getline(cin, line)){
5        stringstream ss(line);
6        v.clear();
7        string s;
8        while (ss >> s) v.push_back(s);
9
10       for (int i = 0; i < v.size(); ++i){
11          reverse(v[i].begin(), v[i].end());
12          if (i > 0) cout << " ";
13          cout << v[i] << " ";
14       }
15       cout << endl;
16    }
17    return 0;
18 }
```

Pila o Stack

Pila o Stack

Es una contenedor dinámico en el cual sólo se pueden insertar elementos al final y sólo se pueden extraer elementos del final. **El último elemento que se insertó es el primer elemento en salir (LIFO).**

Un ejemplo sería una pila de libros.

Operaciones

Sobre una **pila** se pueden realizar las siguientes operaciones

- Push(x) inserta el elemento x al stack.
- Pop() elimina el último elemento del stack.
- Top() retorna el último elemento almacenado.

Implementación usando la librería de C++

```
1  #include <iostream>
2  #include <stack>                // Incluir stack
3  using namespace std;
4
5  int main(){
6     stack <int> s;               // Crear un stack de enteros
7     s.push(10);                  // Insertar 10
8     s.push(-1);                  // Insertar -1
9     cout << s.top() << endl;     // Imprimir -1
10    s.pop();                     // Eliminar -1
11    cout << s.top() << endl;     // Imprimir 10
12    cout << s.size() << endl; // El tamano del stack es 1
13    return 0;
14 }
```

Cola o Queue

Cola o Queue

Es una contenedor dinámico en el cual sólo se pueden insertar elementos al final y sólo se pueden extraer elementos del principio.
El primer elemento que se insertó es el primer elemento en salir (FIFO).

Un ejemplo sería una cola en un banco.

Operaciones

Sobre una **cola** se pueden hacer las siguientes operaciones

- Push(x) inserta el elemento x a la cola.
- Pop() elimina el primer elemento de la cola.
- Front() retorna el primer elemento de la cola.
- Back() retorna el último elemento de la cola

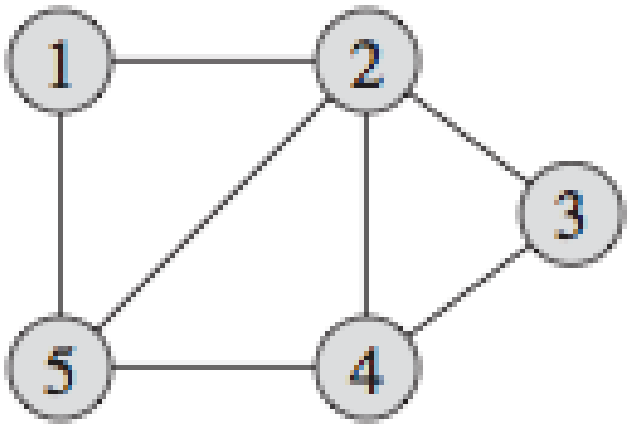
Implementación usando la librería de C++

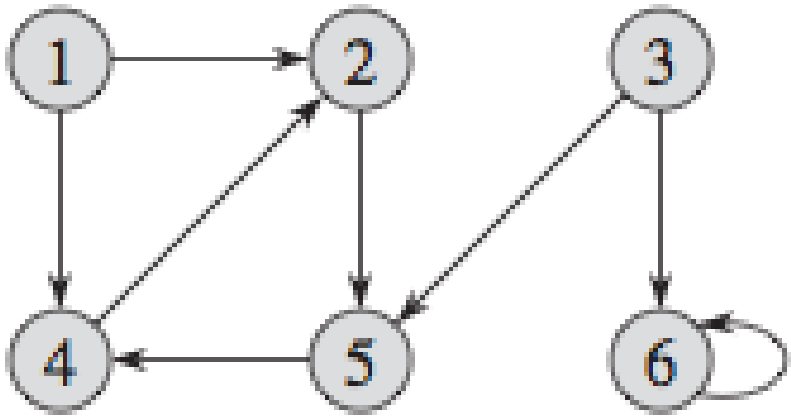
```cpp
#include <iostream>
#include <queue>                    // Incluir queue
using namespace std;

int main(){
    queue <int> q;                  // Crear una cola de enteros
    q.push(10);                     // Insertar 10
    q.push(-1);                     // Insertar -1
    cout << q.front() << endl;      // Imprimir 10
    q.pop();                        // Eliminar 10
    cout << q.front() << endl;      // Imprimir -1
    cout << q.size() << endl;       // El tamano de la cola es 1
    return 0;
}
```

BFS

- Algoritmo para recorrer o buscar elementos en un grafo.
- Se comienza desde uno nodo y se exploran todos los vecinos de este nodo.
- Luego, para cada uno de los vecinos, se exploran sus respectivos vecinos (que no se hayan visto antes).
- Se continúa de esta manera hasta que se haya recorrido todo el grafo.

Ejemplos





Pregunta

Complejidad

- ¿Cúantas veces visto cada nodo?
- ¿Cúantas veces visito cada arista?
- De acuerdo a lo anterior ¿Cuál sería una aproximación a la complejidad del algoritmo?

Estructura de datos

De las estructuras de datos mostradas anteriormente ¿Cuál sería la apropiada para almacenar los nodos que tengo pendientes por visitar?

Pregunta

Complejidad

- ¿Cúantas veces visto cada nodo?
- ¿Cúantas veces visito cada arista?
- De acuerdo a lo anterior ¿Cuál sería una aproximación a la complejidad del algoritmo?

Estructura de datos

De las estructuras de datos mostradas anteriormente ¿Cuál sería la apropiada para almacenar los nodos que tengo pendientes por visitar?

Algortimo BFS

```
1  vector <int> g[MAXN];    // La lista de adyacencia
2  int d[MAXN];             // Aristas usadas desde la fuente
3
4  void bfs(int s, int n){ // s = fuente, n = numero de nodos
5     // Marcar todos los nodos como no visitados
6     for (int i = 0; i < n; ++i) d[i] = -1;
7     queue <int> q;
8     q.push(s);                    // Agregar la fuente a la cola
9     d[s] = 0;                     // La distancia de la fuente es 0
10    while (q.size() > 0){
11       int cur = q.front(); q.pop();
12       for (int i = 0; i < g[cur].size(); ++i){
13          int next = g[cur][i];
14          if (d[next] == -1){        // Si todava no lo he visitado
15             d[next] = d[cur] + 1;  // La distancia que llevo + 1
16             q.push(next);          // Agregarlo a la cola
17          }
18       }
19    }
20 }
```

Aplicaciones

- Buscar o recorrer elementos en un grafo.
- Hallar mínimo número de aristas para llegar de la fuente a cualquier nodo.
- Hallar los nodos alcanzables desde la fuente (Ver si existe un camino de la fuente a cualquier nodo).
- Si se guarda el nodo del que vine (padre), se pueden hallar las aristas pertenecientes al camino "más corto" desde la fuente hasta cada nodo.
- Con algunas modificaciones sirve para ver si existe un ciclo en algún camino que sale desde la fuente.

Complejidad

Complejidad de BFS

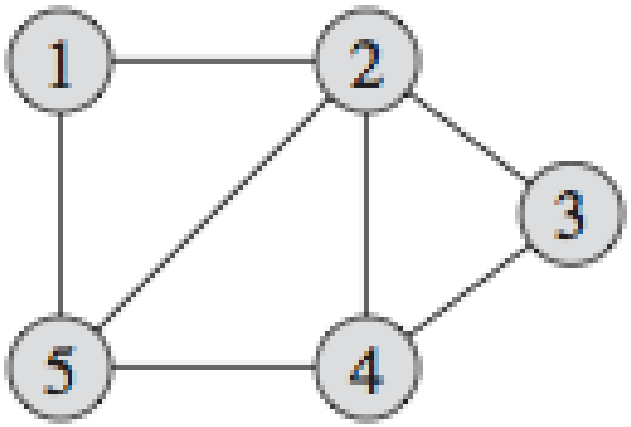
- Si se representa usando la **lista de adyacencia** la complejidad del BFS es $O(V + E)$.
- Si se representa usando la **matriz de adyacencia** la complejidad del BFS es $O(V^2)$.

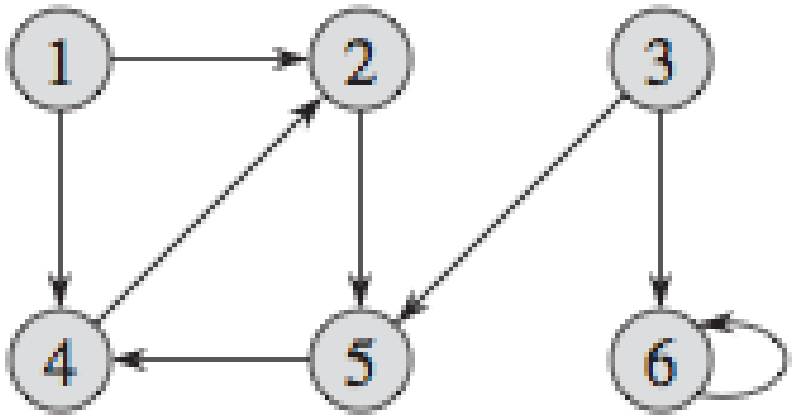
DFS

Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa (Backtracking), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado

- Algoritmo para recorrer o buscar elementos en un grafo.
- Se comienza desde uno nodo y se marca como gris (parcialmente visitado).
- Se explora cada uno de los vecinos de ese nodo.
- Cuando termino de visitar todos los vecinos, marco el nodo como negro (visitado).
- En otras palabras, no visito un nodo hasta haber visitado todos sus vecinos.

Ejemplos





Pregunta

Complejidad

- ¿Cúantas veces visto cada nodo?
- ¿Cúantas veces visito cada arista?
- De acuerdo a lo anterior ¿Cuál sería una aproximación a la complejidad del algoritmo?

Estructura de datos

De las estructuras de datos mostradas anteriormente ¿Cuál sería la apropiada para almacenar los nodos que tengo pendientes por visitar?

Pregunta

Complejidad

- ¿Cúantas veces visto cada nodo?
- ¿Cúantas veces visito cada arista?
- De acuerdo a lo anterior ¿Cuál sería una aproximación a la complejidad del algoritmo?

Estructura de datos

De las estructuras de datos mostradas anteriormente ¿Cuál sería la apropiada para almacenar los nodos que tengo pendientes por visitar?

Algortimo DFS

```
1  vector <int> g[MAXN];        // La lista de adyacencia
2  int color[MAXN];             // El arreglo de visitados
3  enum {WHITE, GRAY, BLACK}; // WHITE = 1, GRAY = 2, BLACK = 3
4
5  void dfs(int u){
6     color[u] = GRAY;   // Marcar el nodo como semi-visitado
7     for (int i = 0; i < g[u].size(); ++i){
8        int v = g[u][i];
9        if (color[v] == WHITE) dfs(v);  // Visitar mis vecinos
10    }
11    color[u] = BLACK;  // Marcar el nodo como visitado
12 }
13
14 void call_dfs(int n){
15    // Marcar los nodos como no visitados
16    for (int u = 0; u < n; ++u) color[u] = WHITE;
17    // Llamar la funcion DFS con los nodos no visitados
18    for (int u = 0; u < n; ++u)
19       if (color[u] == WHITE) dfs(u);
20 }
```

Aplicaciones

- Buscar o recorrer elementos en un grafo.
- Si se guarda el nodo del que vine (padre), se pueden hallar un camino desde la fuente hasta cada nodo.
- Ver si existe un ciclo en el grafo (si uno de mis vecinos es gris cuando lo voy a visitar).
- Se puede hacer que el DFS retorne algún valor y verificar con él condiciones en el grafo.
- Si se modifica el color por el ciclo en el que visité el nodo puedo hallar los nodos alcanzables desde el nodo en el que hago el llamado inicial.

Complejidad

Complejidad de DFS

- Si se representa usando la **lista de adyacencia** la complejidad del DFS es $O(V + E)$.
- Si se representa usando la **matriz de adyacencia** la complejidad del DFS es $O(V^2)$.

Tarea

Tarea

Resolver los problemas de
http://contests.factorcomun.org/contests/50

Problema A Construir el grafo (es general para todos los casos de prueba). Cuando se vaya a visitar un nodo, verificar que no esté en la lista de los prohibidos.

Problema B Se puede utilizar cualquiera de las dos técnicas (BFS o DFS). Si uno de mis vecinos ya está visitado, verificar que su color sea el contrario al mío si no es así retorne falso.