

Introduction to Lambda Calculus

Santiago Palacio Gómez

Universidad EAFIT

12 de marzo de 2013

Contenido

- 1 Tarea Semana Anterior
- 2 Pila
- 3 Cola
- 4 BFS: Breadth-First Search
- 5 DFS: Depth-First Search
- 6 Tarea

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",
17 day);
18         else printf("failure on day %d\n", day);
19     }
20     return 0;
```

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 un 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
7      enteros
8      s.push(10);            // Insertar 10
9      s.push(-1);            // Insertar -1
10     cout << s.top() << endl; // Imprimir -1
11     s.pop();                // Eliminar -1
12     cout << s.top() << endl; // Imprimir 10
13     cout << s.size() << endl; // El tamaño del stack es 1
14     return 0;
15 }
```

Cola o Queue

Cola o Queue

Es un 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

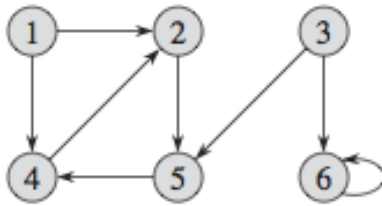
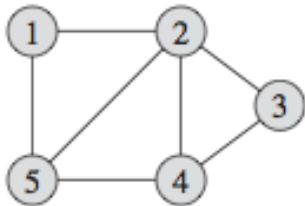
Implementación usando la librería de C++

```
1  #include <iostream>
2  #include <queue>           // Incluir queue
3  using namespace std;
4
5  int main(){
6      queue <int> q;         // Crear una cola de
7      enteros
8      q.push(10);            // Insertar 10
9      q.push(-1);            // Insertar -1
10     cout << q.front() << endl; // Imprimir 10
11     q.pop();                // Eliminar 10
12     cout << q.front() << endl; // Imprimir -1
13     cout << q.size() << endl;  // El tamaño de la cola
14     es 1
15     return 0;
16 }
```

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

- ¿Cuántas veces visto cada nodo?
- ¿Cuántas 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

- ¿Cuántas veces visto cada nodo?
- ¿Cuántas 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?

Algoritmo BFS

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

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

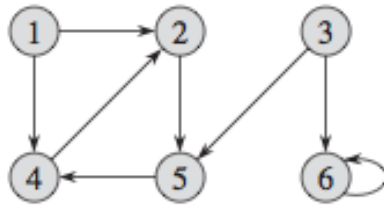
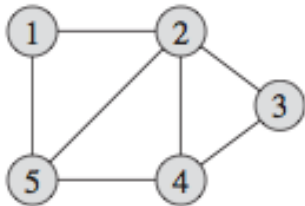
Complejidad de BFS

- 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

- 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

- ¿Cuántas veces visto cada nodo?
- ¿Cuántas 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

- ¿Cuántas veces visto cada nodo?
- ¿Cuántas 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?

Algoritmo 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
4  = 3
5
6  void dfs(int u){
7      color[u] = GRAY;          // Marcar el nodo como
8  semi-visitado
9      for (int i = 0; i < g[u].size(); ++i){
10         int v = g[u][i];
11         if (color[v] == WHITE) dfs(v); // Visitar mis
12 vecinos
13     }
14     color[u] = BLACK;          // Marcar el nodo como visitado
15 }
16
17 void call_dfs(int n){
18     // Marcar los nodos como no visitados
19     for (int u = 0; u < n; ++u) color[u] = WHITE;
20     // Llamar la funcion DFS con los nodos no visitados
21     for (int u = 0; u < n; ++u) if (color[u] == WHITE) dfs(u);
22 }
```


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 en lugar de poner el color pongo el número del nodo con el que hice la llamada inicial desde `call_dfs`, puedo hallar todos los nodos alcanzables desde este nodo 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>
- Para cada problema pensar: ¿Cómo se construiría el grafo? ¿Es dirigido o no dirigido? ¿Debo usar BFS, DFS o cualquiera de las dos?

Ayuda para la tarea

Ayuda para cada problema

- A El grafo es general para todos los casos de prueba (no cambia). Cuando se vaya a visitar un nodo, verificar que no esté en la lista de los prohibidos.
- B Si uno de mis vecinos ya está visitado, verificar que su color sea el contrario al mío si no es así retorne falso.
- C Pensar ¿Cuáles son las fichas que seguro tengo que derribar a mano?. Empujar esas fichas y las que ellas derriben (marcarlas como visitadas). Si luego de esto me quedan fichas sin tumbar ¿Qué característica especial cumplen esas fichas? ¿Cómo las debo derribar?