UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA
FACULTAD DE INGENIERIA DE PRODUCCION Y SERVICIOS
ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

LABORATORIO ESTRUCTURAS DISCRETAS

DOCENTE: EDITH PAMELA RIVERO TUPAC

ESTUDIANTE: MELODY MARISOL RAMOS CHALLA

Arequipa 2021

GRAFOS

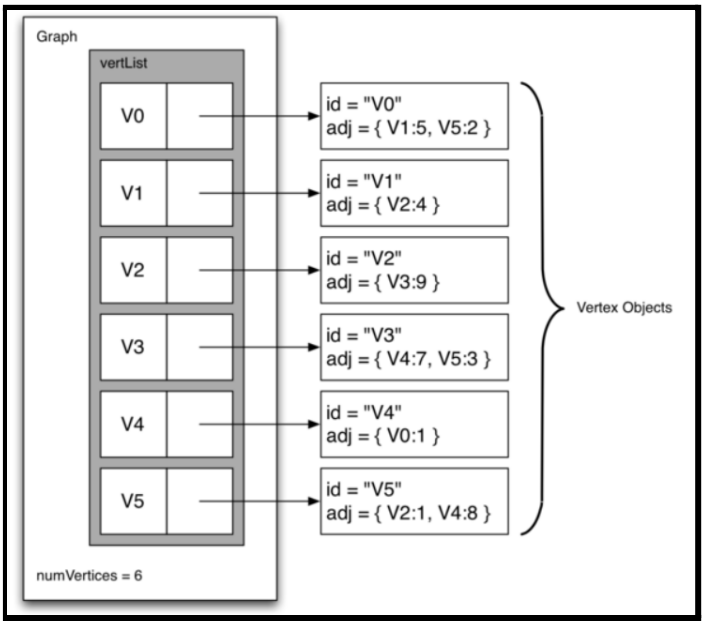
PROBLEMAS PROPUESTOS

1. **Propuesto 2**

   Implementación lista de Adyacencia

   Una forma más eficiente de implementar un gráfico escasamente conectado es utilizar una lista de adyacencia. En una implementación de lista de adyacencia, mantenemos una lista maestra de todos los vértices en el objeto Graph y luego cada objeto de vértice en el gráfico mantiene una lista de los otros vértices a los que está conectado.

   La ventaja de la implementación de la lista de adyacencia es que nos permite representar de forma compacta un gráfico disperso. La lista de adyacencia también nos permite encontrar fácilmente todos los enlaces que están conectados directamente a un vértice en particular.




```
Lista de adyacencia
0=>[(5,2), (1,5)]
1=>[(2,4)]
2=>[(3,9)]
3=>[(5,3), (4,7)]
4=>[(0,1)]
5=>[(4,8), (2,1)]
```

```java
import java.util.*;
public class ListaAdyacencia{
    public class Nodo{
        int nodo,w;
        public Nodo(int nodo,int w) {
            this.nodo=nodo;
            this.w=w;
        }
        @Override
        public String toString(){
            return "("+nodo+","+w+")";
        }
    }
    LinkedList<Nodo>listaNodo[];
    public ListaAdyacencia(int n){
        listaNodo=new LinkedList[n];
        for(int i=0;i<listaNodo.length;i++)
            listaNodo[i]=new LinkedList<Nodo>();
    }
    public void addNodo(int u,int v,int w){
        listaNodo[u].add(0,new Nodo(v,w));
    }
    @Override
    public String toString(){
        String result="";
        for(int i=0;i<listaNodo.length;i++)
            result+=i+"=>"+listaNodo[i]+"\n";
        return result;
    }
}
```

2. **Propuesto 3**

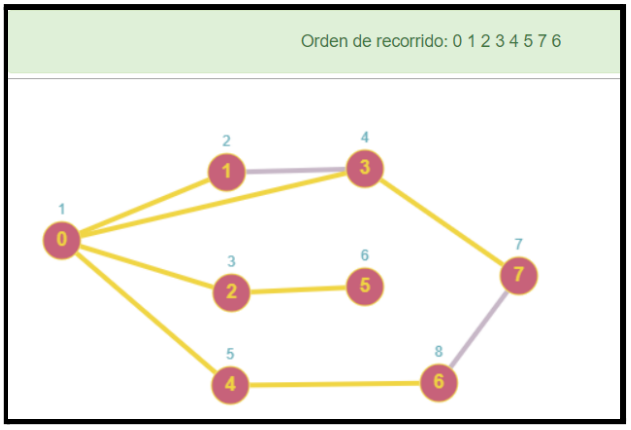
BSF

La primera búsqueda de amplitud es una técnica general de recorrer un gráfico. La primera búsqueda de amplitud puede usar más memoria, pero siempre encontrará primero la ruta más corta. En este tipo de búsqueda, el espacio de estados se representa en forma de árbol. La solución se obtiene atravesando el árbol. Los nodos del árbol representan el valor inicial o el estado inicial, varios estados intermedios y el estado final. En esta búsqueda se utiliza una estructura de datos de cola y se recorre nivel por nivel.

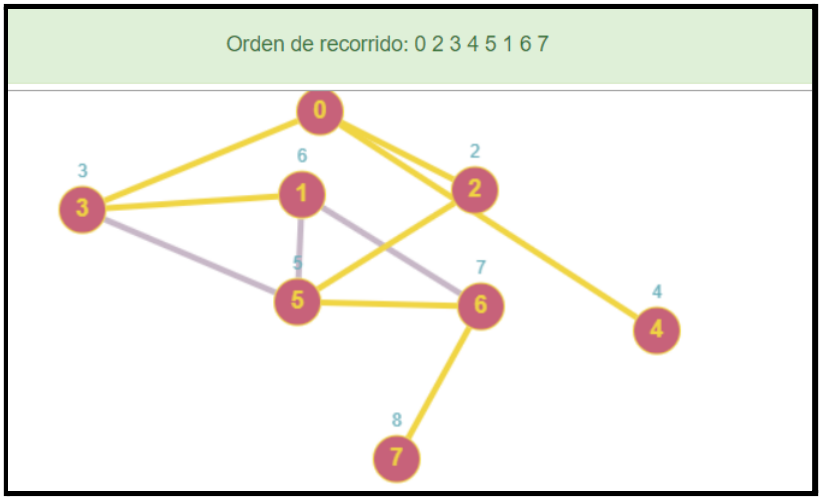
Una ventaja es que no sigue un solo camino infructuoso durante mucho tiempo. Encuentra la solución mínima en caso de múltiples rutas.Tiene como desventajas el consumir gran espacio de memoria. Su complejidad temporal es mayor. Tiene rutas largas, cuando todas las rutas a un destino se encuentran aproximadamente en la misma profundidad de búsqueda.

```java
public static void VisitBFS(int v,boolean[] visitado) {
    queue.add(v);
    visitado[v]=true;
    int a;
    while(!queue.isEmpty()) {
        a=queue.remove();
        System.out.print(a+" ");
        Iterator<Integer> miIter=listaNodo[a].listIterator();
        int b;
        while(miIter.hasNext()) {
            b=miIter.next();
            if(visitado[b]!=true) {
                queue.add(b);
                visitado[b]=true;
            }
        }
    }
}
public static void bFS(int v) {
    boolean visitado[]= new boolean[nodo];
    VisitBFS(v,visitado);
    for(int i=0;i<nodo;i++) {
        if(visitado[i]!=true)
            VisitBFS(i,visitado);
    }

}
```




```
Recorrido BFS:
0 1 2 3 4 5 7 6
```



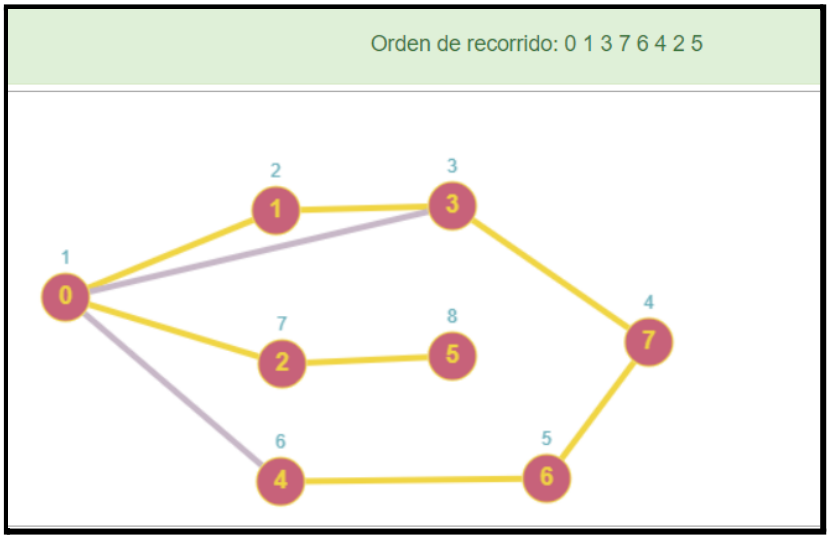

```
Recorrido BFS:
0 2 3 4 5 6 7 1
```

DFS

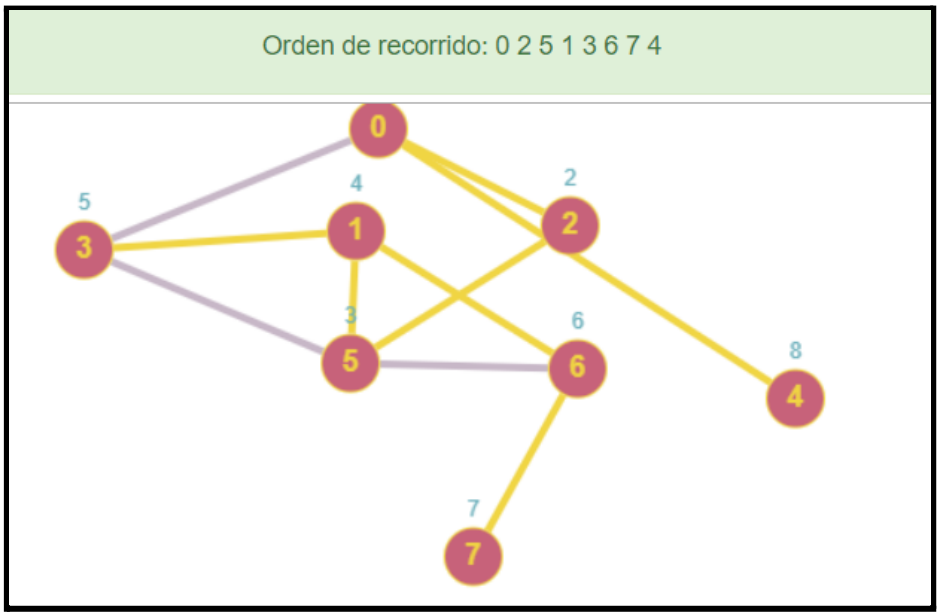
El algoritmo DFS es un algoritmo recursivo que utiliza la idea de retroceder. Implica búsquedas exhaustivas de todos los nodos avanzando, si es posible, o retrocediendo.Aquí, la palabra retroceso significa que cuando avanza y no hay más nodos a lo largo de la ruta actual, se mueve hacia atrás en la misma ruta para encontrar nodos que atravesar. Todos los nodos se visitarán en la ruta actual hasta que se hayan atravesado todos los nodos no visitados, después de lo cual se seleccionará la siguiente ruta.Esta naturaleza recursiva de DFS se puede implementar utilizando pilas. La idea básica es la siguiente:elegir un nodo inicial y empujar todos los nodos adyacentes en una pila, luego sacar un nodo de la pila para seleccionar el siguiente nodo a visitar y empujar todos los nodos adyacentes en una pila.

```
public static void VisitDFS(int v,int[] visitado) {
    visitado[v]=1;
    System.out.print(v+" ");
    Iterator<Integer> miIter=listaNodo[v].listIterator();
    while(miIter.hasNext()) {
        int b=miIter.next();
            if(visitado[b]==0) {
                VisitDFS(b,visitado);
            }
    }
}

public static void dFS(int v) {
    int visitado[]=new int[nodo];
    VisitDFS(v,visitado);
    for(int i=1;i<nodo;i++) {
        if(visitado[i]==0)
            VisitDFS(i,visitado);
    }
```




```
Recorrido DFS
0 1 3 7 6 4 2 5
```



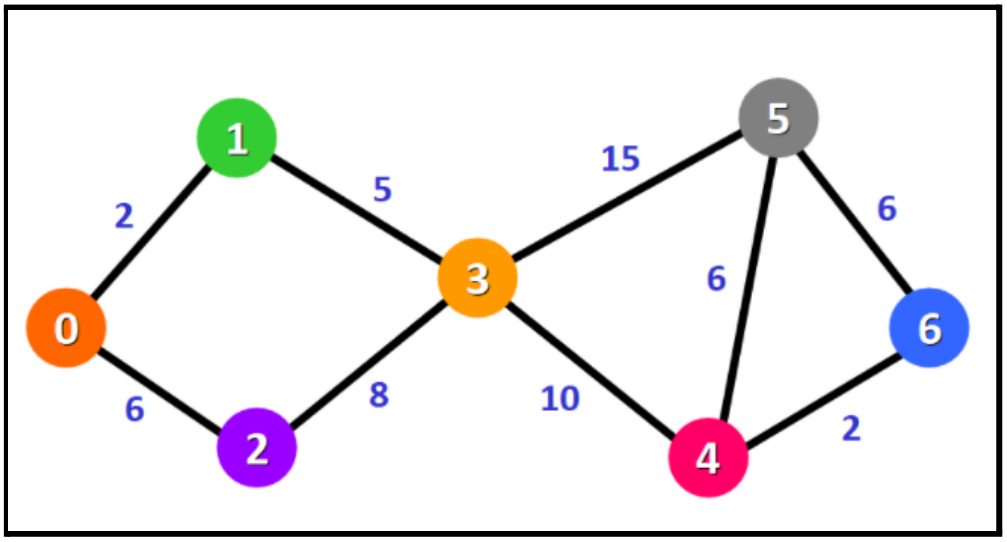

```
Recorrido DFS
0 2 5 1 3 6 7 4
```

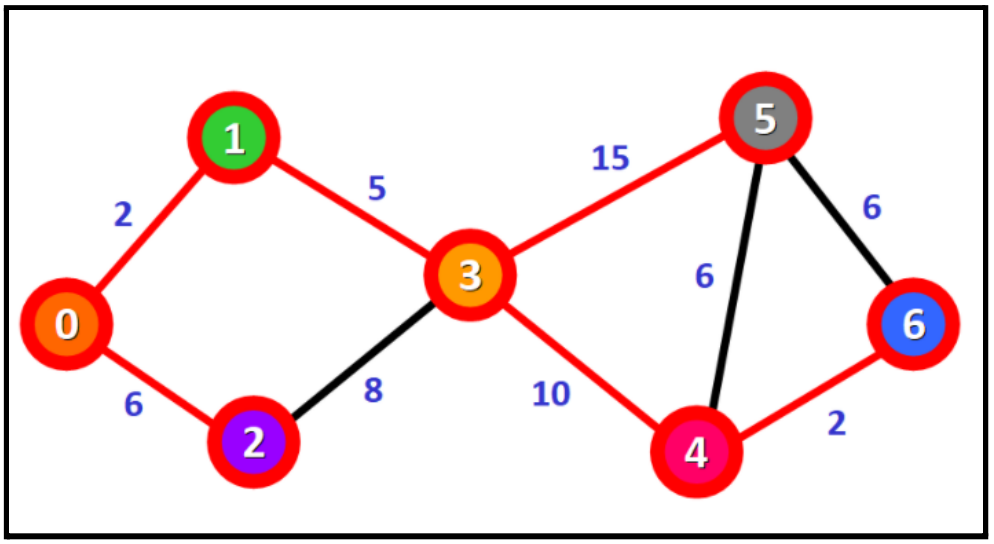
DIJKSTRA

El algoritmo de Dijkstra utiliza pesos de los bordes para encontrar la ruta que minimiza la distancia total entre el nodo fuente y todos los demás nodos. Este algoritmo también se conoce como el algoritmo de ruta más corta de fuente única.

Para la implementación del algoritmo el primer paso es marcar todos los nodos como no visitados,marcar el nodo inicial elegido con una distancia actual de 0 y los nodos restantes con infinito, luego arreglar el nodo inicial como el nodo actual. Para el nodo actual, se analiza todos sus vecinos no visitados y se mide sus distancias sumando la distancia actual del nodo actual al peso del borde que conecta el nodo vecino y el nodo actual,para comparar la distancia medida recientemente con la distancia actual asignada al nodo vecino y convertirla en la nueva distancia actual del nodo vecino. Después de eso, se considera todos los vecinos no visitados del nodo actual, marcando el nodo actual como visitado,si el nodo de destino se ha marcado como visitado, se detiene. De lo contrario, se elige el nodo no visitado que está marcado con la menor distancia, se marca como el nuevo nodo actual y se repite el proceso nuevamente.

```java
public Dijkstra(int num) {
    this.num = num;
    dist = new int[num];
    visitado = new ArrayList<Integer>();
    pq = new PriorityQueue<Nodo>(num, new Nodo());
}
public void DijkstraA(List<List<Nodo> >nodoLista, int a){
    this.nodoLista = nodoLista;
    for (int i = 0; i < num; i++)
        dist[i] = Integer.MAX_VALUE;
    pq.add(new Nodo(a, 0));
    dist[a] = 0;
    while(visitado.size()!= num){
        int u = pq.remove().nodo;
        visitado.add(u);
        add(u);
    }
}
```






```
Algoritmo de Dijkstra:
Inicio  Nodo    Distancia
0         0       0
0         1       2
0         2       6
0         3       7
0         4       17
0         5       22
0         6       19
```

3. **Propuesto 4**

   El grafo de palabras se define de la siguiente manera: cada vértice es una palabra en el idioma Inglés y dos palabras son adyacentes si difieren exactamente en una posición. Por ejemplo, las cords y los corps son adyacentes, mientras que los corps y crops no lo son.

   a) Dibuje el grafo definido por las siguientes palabras: words cords corps coops crops drops drips grips gripe grape graph

```java
import java.util.*;
public class Adyacencia{
    public class Nodo{
        String str;
        boolean isEnd;
        boolean visit;
        HashMap<Character,Nodo> hijo;

        public Nodo() {
          this.isEnd = false;
          this.visit = false;
          this.str = "";
          hijo = new HashMap<Character,Nodo>();
        }
        @Override
        public String toString(){
            return "("+str+")";
        }
    }
    public Nodo root = new Nodo();
    public int tam = Integer.MAX_VALUE;
    public int longitud(String pala1, String pala2, Set<String> lista) {
        if (pala1 == null || pala2 == null || pala1.length() != pala2.length() || lista== null || lista.size() == 0) {
            return 0;
        }
        for (String s:lista) {
            insert(s);
        }
        DFS(pala1,pala2, 1);
        return tam;
    }
```

```java
public void insert (String s){
  char[] arr = s.toCharArray();
  Nodo node = root;
  for (int i = 0; i < arr.length; i++) {
    char c = arr[i];
    if (!node.hijo.containsKey(c)) {
      node.hijo.put(c, new Nodo());
    }
    node = node.hijo.get(c);
    if (i == arr.length - 1) {
      node.isEnd = true;
      node.str = s;
    }
  }
}

public HashMap<Character,Nodo> obtenerActual(String s) {
    char[] arr = s.toCharArray();
    Nodo node = root;
    for (int i = 0; i < arr.length; i++) {
        char c = arr[i];
        if (!node.hijo.containsKey(c)) {
          return null;
        }
        if (i == arr.length - 1) {
            return node.hijo;
        }
        node = node.hijo.get(c);
    }
    return null;
}
public HashMap<Character,Nodo> obtenerHijo(String s) {
    if (s == null || s.length() == 0) {
      return root.hijo;
    }
    char[] arr = s.toCharArray();
    Nodo node = root;
    for (int i = 0; i < arr.length; i++) {
        char c = arr[i];
        if (!node.hijo.containsKey(c)) {
          return null;
        }
        node = node.hijo.get(c);
    }
    return node.hijo;
}
public int compare(String pala1, String pala2) {
  int cont = 0;
  for (int i = 0; i < pala1.length(); i++) {
    cont += pala1.charAt(i) != pala2.charAt(i) ? 1 : 0;
    if (cont > 1) {
      return cont;
    }
  }
  return cont;
}
```

4. **Propuesto 5**

```
public class Grafo {
    public class Node{
        int data;
        Node left, right;

        Node(int item){
            data = item;
            left = right;
        }
    }
    public class GrafoVer{
        Node root1, root2;
        public boolean verificacion(Node n1, Node n2){
            if (n1 == null && n2 == null)
                return true;
            if (n1 == null || n2 == null)
                return false;
            if (n1.data != n2.data)
                return false;
            return (verificacion(n1.left, n2.left) && verificacion(n1.right, n2.right))
            || verificacion(n1.left, n2.right) && verificacion(n1.right, n2.left);
        }
```

5. **Cuestionario**

a) ¿Cuántas variantes del algoritmo de Dijkstra hay y cuál es la diferencia entre ellas?
Dijkstra con cola de prioridad
Esta variante utiliza O(m log n) operaciones, siendo n la cantidad de nodos del problema y m la cantidad de aristas, se basa en que las estructuras que se usan para la cola de prioridad extraen e insertan un elemento en tiempo logarítmico.
Si el grafo es ralo, o sea, tiene pocas aristas, conviene utilizar la implementación con cola de prioridad (O(m log n)) Si el grafo es denso, o sea, tiene muchas aristas, conviene utilizar la implementación básica (O(n 2)).

b) Invetigue sobre los ALGORITMOS DE CAMINOS MINIMOS e indique, ¿Qué similitudes encuentra, qué diferencias, en qué casos utilizar y porque?
Dijkstra: resuelve el problema de los caminos más cortos desde un único nodo origen hasta todos los otros nodos del grafo (aunque aplicando una regla de repetición del algoritmo, se puede automatizar la resolución del problema desde todos los nodos de origen hasta todos los nodos del grafo).Sin embargo, este algoritmo tiene algunas limitaciones, siendo la fundamental que no funciona en grafos con aristas de coste negativo puesto que se producen inestabilidades en la estrategia de búsqueda. Y otra importante es que no es directamente paralelizable, algo muy importante para los procesos que se engloban dentro del Big Data. Pero sí que permite formulaciones muy flexibles para modelizar problemas de transporte y la programación de sumideros para evitar el uso de pesos negativos.

Bellman-Ford: resuelve el problema de los caminos más cortos desde un origen permitiendo que la ponderación de los nodos sea negativa. También funciona con pesos negativos. Esto puede funcionar cuando, en algunos casos, podemos obtener alguna ventaja al tomar un camino determinado,funciona bien para sistemas distribuidos, son lugar de trabajo de algoritmos dinámicos. Algunas limitaciones serían que sólo puede funcionar para gráficos dirigidos y requiere información local.

Algoritmo de Floyd-Warshall: resuelve el problema de los caminos más cortos entre todos los nodos.Obtiene la mejor ruta entre todo par de nodos.Trabaja con la matriz D inicializada con las distancias directas entre todo par de nodos.La iteración se produce sobre nodos intermedios, o

sea para todo elemento de la matriz se prueba si lo mejor para ir de i a j es a través de un nodo intermedio elegido o como estaba anteriormente, y esto se prueba con todos los nodos de la red. Una vez probados todos los nodos de la red como nodos intermedios, la matriz resultante da la mejor distancia entre todo par de nodos.El algoritmo da sólo la menor distancia; se debe manejar información adicional para encontrar tablas de encaminamiento.Hasta no hallar la última matriz no se encuentran las distancias mínimas.Su complejidad es del orden de N3.El algoritmo de Floyd-Warshall puede ser utilizado para resolver los siguientes problemas: Camino mínimo en grafos dirigidos, cierre transitivo en grafos dirigidos,ruta óptima,comprobar si un grafo no dirigido es bipartito.

Algoritmo de Johnson: resuelve el problema de los caminos más cortos entre todos los nodos y puede ser más rápido que el de Floyd-Warshall en grafos de baja densidad.El algoritmo de Johnson es una forma de encontrar el camino más corto entre todos los pares de vértices de un grafo dirigido disperso. Permite que las aristas tengan pesos negativos, si bien no permite ciclos de peso negativo. Funciona utilizando el algoritmo de Bellman-Ford para hacer una transformación en el grafo inicial que elimina todas las aristas de peso negativo, permitiendo por tanto usar el algoritmo de Dijkstra en el grafo transformado.

Algoritmo de Viterbi: resuelve el problema del camino estocástico más corto con un peso probabilístico adicional en cada nodo. Se usa como técnica de codificación convolucional diseñada para reducir la probabilidad de transmisión errónea a través de canales de telecomunicación ruidosos. Este algoritmo pretende encontrar una secuencia de estados denominada ruta de Viterbi que explique una secuencia determinada de observaciones. El algoritmo de Viterbi limita el número de secuencias a considerar para cada estado, de esta manera cuando encuentra la mejor secuencia el resto se descarta, para cada tiempo y estado se guarda la mejor secuencia obtenida y su valor.