Grafos

Búsqueda en profundidad

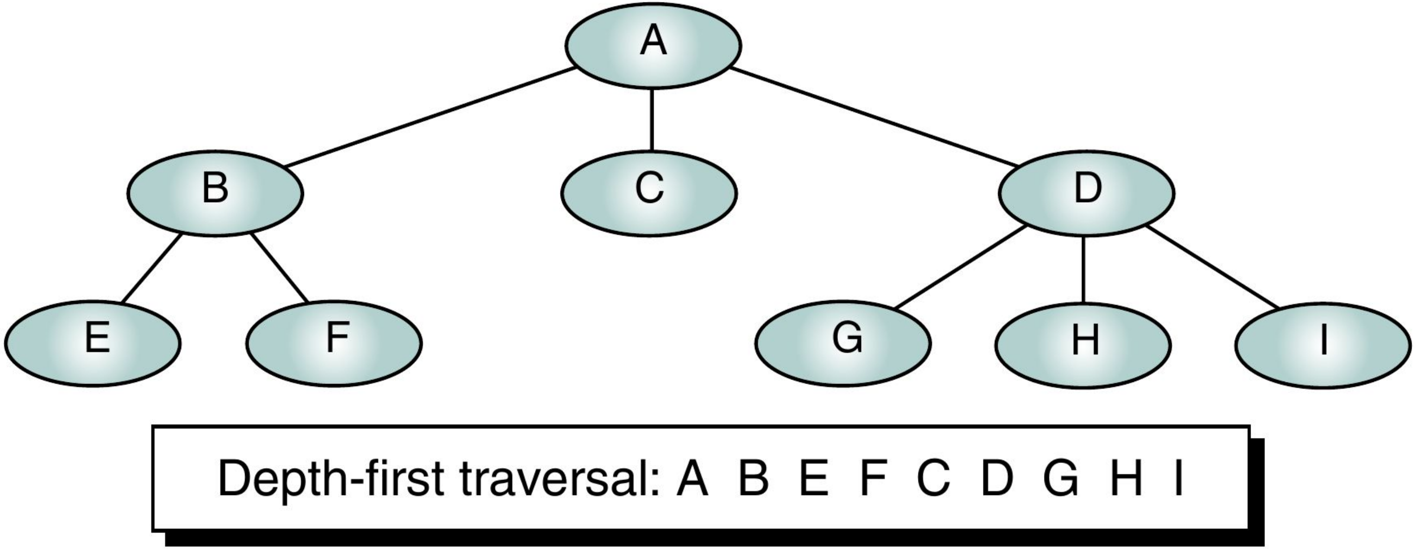
Profesor: Néstor Suat-Rojas. Ing., M.Sc.

Recorridos

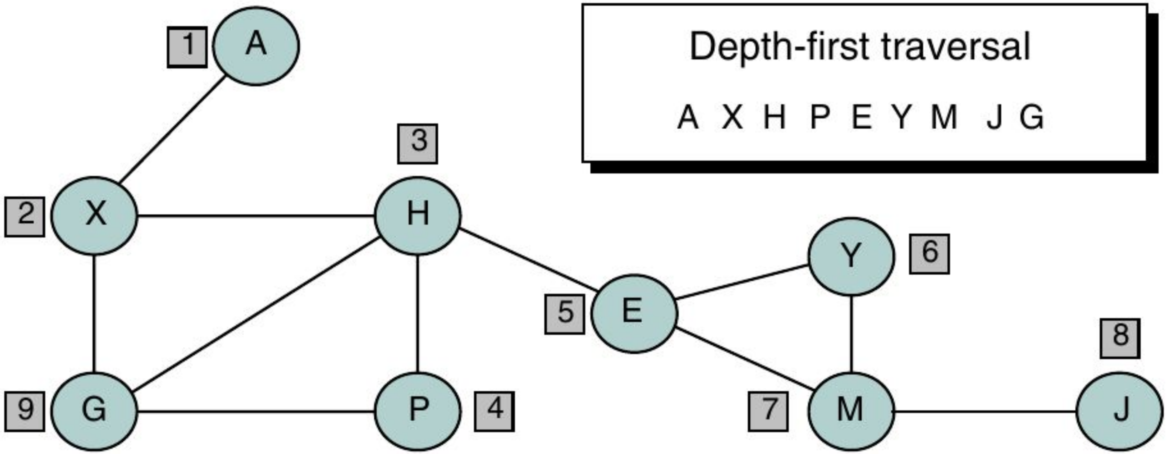
- Más de un algoritmo requiere recorrer todos los vértices en el grafo, procesarlos una sola vez.
- Sin embargo, dada la definición de un grafo, tenemos más de una ruta para llegar a un vértice.
- Soluciones tradicionales incluyen una bandera en cada vértice para determinar si el vértice ya fue visitado o no.

Recorrido en profundidad

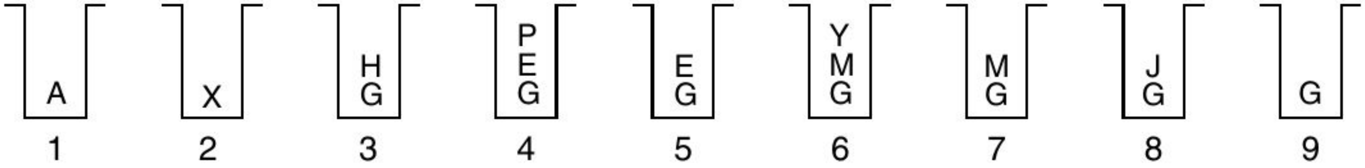
Procesa todos los vértices descendientes de un vértice antes de continuar con los vértices adyacentes. (Preorder).



Recorrido en profundidad



(a) Graph



(b) Stack contents

Búsqueda en profundidad

El problema de la gira del caballo

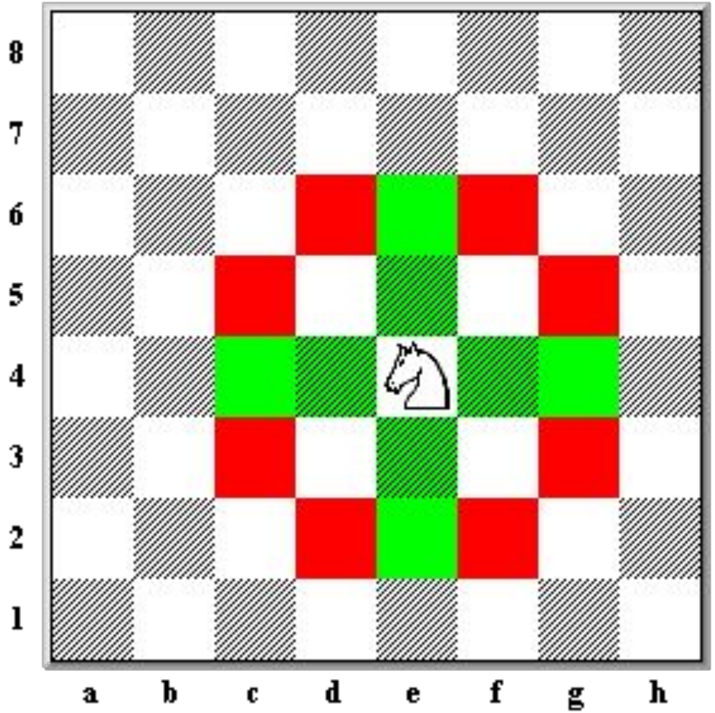
Entrada:

- Se juega en un tablero de ajedrez con una sola pieza de ajedrez, el caballo.

Salida:

- Encontrar una secuencia de movimientos que permitan al caballo visitar cada cuadro en el tablero exactamente una vez.

El problema de la gira del caballo



Video de Derivando: https://www.youtube.com/watch?v=xub1KmUgrdk

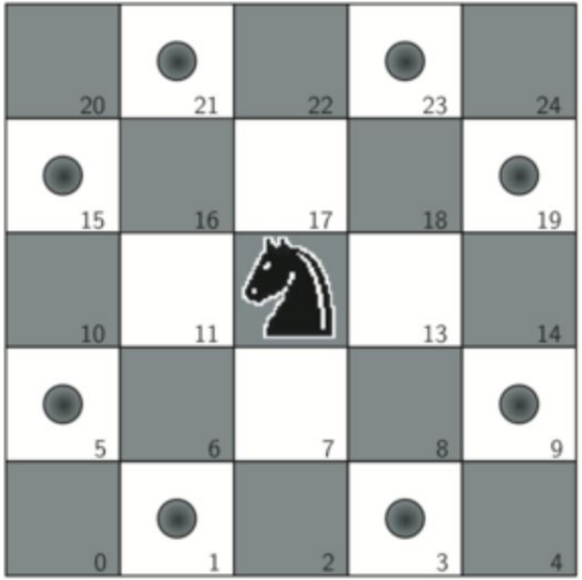
El problema de la gira del caballo

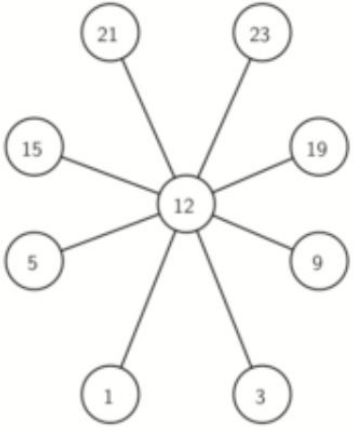
- Representar como un grafo los movimientos legales de un caballo en el tablero de ajedrez.
- Usar un algoritmo de grafos para encontrar la ruta de longitud `filas x columnas - 1` donde cada vértice del grafo se visita una vez.

Construcción del grafo

Dos ideas:

- Cada cuadro como un vértice.
- Cada movimiento legal como una arista.





Construcción del grafo

```
Graph<int> KnightGraph(int boardSize){
   Graph<int> g;
   for(int row=0; row<boardSize; row++){
       for(int col=0; col<boardSize; col++){
           int nodeIdOrigin = positionToId(row, col, boardSize);
           list<tuple<int, int>> newPositions = generateLegalMoves(row, col, boardSize);
           for(auto[r,c] : newPositions){
               int nodeIdDestination = positionToId(r, c, boardSize);
               g.addEdge(nodeIdOrigin, nodeIdDestination);
           }
       }
   }
   return g;
}
```

Construcción del grafo

```
Graph<int> KnightGraph(int boardSize){
    Graph<int> g;
    for(int row=0; row<boardSize; row++){
        for(int col=0; col<boardSize; col++){
            int nodeIdOrigin = positionToId(row, col, boardSize);
            list<tuple<int, int>> newPositions = generateLegalMoves(row, col, boardSize);
            for(auto[r,c] : newPositions){
                int nodeIdDestination = positionToId(r, c, boardSize);
                g.addEdge(nodeIdOrigin, nodeIdDestination);
            }
        }
    }
    return g;
}
```

- En cada cuadro del tablero la función `KnightGraph` llama a la función auxiliar `generateLegalMoves`, para crear una lista de movimientos legales para esa posición.

Construcción del grafo

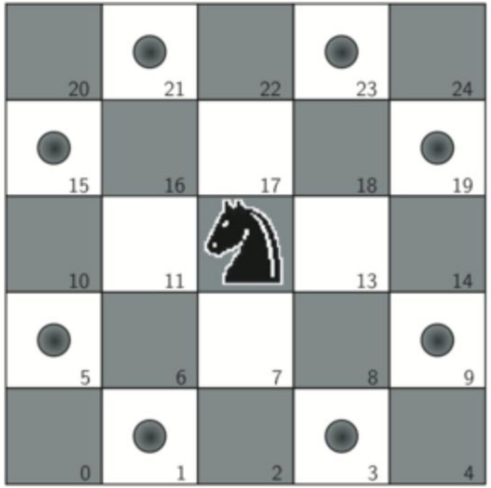
```
Graph<int> KnightGraph(int boardSize){
    Graph<int> g;
    for(int row=0; row<boardSize; row++){
        for(int col=0; col<boardSize; col++){
            int nodeIdOrigin = positionToId(row, col, boardSize);
            list<tuple<int, int>> newPositions = generateLegalMoves(row, col, boardSize);
            for(auto[r,c] : newPositions){
                int nodeIdDestination = positionToId(r, c, boardSize);
                g.addEdge(nodeIdOrigin, nodeIdDestination);
            }
        }
    }
    return g;
}
```

- Todos los movimientos legales se convierten en aristas.

Construcción del grafo

```
Graph<int> KnightGraph(int boardSize){
   Graph<int> g;
   for(int row=0; row<boardSize; row++){
       for(int col=0; col<boardSize; col++){
           int nodeIdOrigin = positionToId( row, col, boardSize);
           list<tuple<int, int>> newPositions = generateLegalMoves( row, col, boardSize);
           for(auto[r,c] : newPositions){
               int nodeIdDestination = positionToId( r, c, boardSize);
               g.addEdge(nodeIdOrigin , nodeIdDestination );
           }
       }
   }
   return g;
}
```

```
int positionToId(int row, int col, int boardSize){
    return (row * boardSize) + col;
}
```



- `positionToId` convierte una posición en el tablero (fila y columna) en un número de vértice lineal.

Construcción del grafo

```
list<tuple<int, int>> generateLegalMoves (int x, int y, int boardSize){
   list<tuple<int, int>> newMoves;
   list<tuple<int, int>> moveOffsets;
   moveOffsets.push_back(tuple<int,int>(-1,-2));
   moveOffsets.push_back(tuple<int,int>(-1,2));
   moveOffsets.push_back(tuple<int,int>(-2,-1));
   moveOffsets.push_back(tuple<int,int>(-2,1));
   moveOffsets.push_back(tuple<int,int>(1,-2));
   moveOffsets.push_back(tuple<int,int>(1,2));
   moveOffsets.push_back(tuple<int,int>(2,-1));
   moveOffsets.push_back(tuple<int,int>(2,1));

   for(auto[row,col] : moveOffsets){
       int newX = x + row;
       int newY = y + col;
       if( legalCoord(newX, boardSize) && legalCoord( newY, boardSize)){
           newMoves.push_back(tuple<int, int>(newX, newY));
       }
   }
   return newMoves;
}
```

La función `generateLegalMoves` toma la posición del caballo y genera 8 movimientos legales.

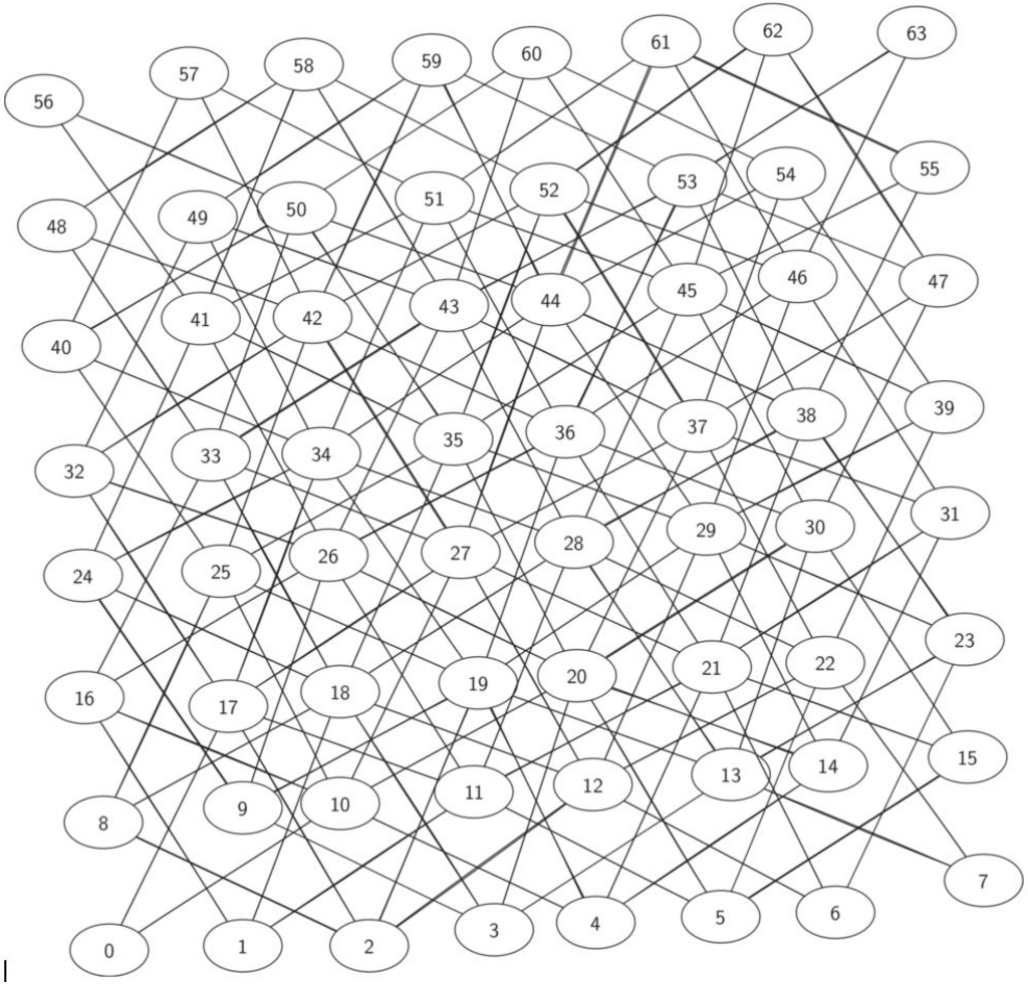
Construcción del grafo

```
list<tuple<int, int>> generateLegalMoves (int x, int y, int boardSize){
   list<tuple<int, int>> newMoves;
   list<tuple<int, int>> moveOffsets;
   moveOffsets.push_back(tuple< int,int>(-1,-2));
   moveOffsets.push_back(tuple< int,int>(-1,2));
   moveOffsets.push_back(tuple< int,int>(-2,-1));
   moveOffsets.push_back(tuple< int,int>(-2,1));
   moveOffsets.push_back(tuple< int,int>(1,-2));
   moveOffsets.push_back(tuple< int,int>(1,2));
   moveOffsets.push_back(tuple< int,int>(2,-1));
   moveOffsets.push_back(tuple< int,int>(2,1));

   for(auto[row,col] : moveOffsets){
       int newX = x + row;
       int newY = y + col;
       if( legalCoord(newX, boardSize) && legalCoord( newY, boardSize)){
           newMoves.push_back(tuple< int, int>(newX, newY));
       }
   }
   return newMoves;
}
```

La función `generateLegalMoves` toma la posición del caballo y genera 8 movimientos legales.

```
bool legalCoord(int x, int boardSize){
    if(x >= 0 && x < boardSize) return
true;
    else return false;
}
```

La función `legalCoord` asegura que un movimiento particular que se genere todavía esté aún dentro del tablero.



Construcción del grafo

- Grafo de un tablero de 8x8.
- 336 aristas.
- Vértices en las esquinas tienen menos aristas.
- Grafo: 336 aristas de 4096 posibles (8,2%).

Implementación de la gira del caballo

Algoritmo de búsqueda en profundidad (BEP o DFS).

- La exploración en profundidad del grafo es exactamente lo que necesitamos para encontrar una ruta que tiene exactamente 63 aristas.
- Si el algoritmo encuentra un callejón sin salida, él retrocede al siguiente vértice más profundo que permita realizar un movimiento legal.

Implementación de la gira del caballo

- `knightTour` recibe cuatro parámetros:
  - `n`, profundidad actual.
  - `ruta`, lista de vertices visitados.
  - `u`, vértice actual que se está explorando.
  - `limite`, el número de nodos en la ruta.

```
bool knightTour(int n, vector<int> &path, Vertex<int>* u, int
limit){
    bool done;
    u->color = 'g';
    path.push_back(u->data);
    if(n < limit){
        vector<Edge<int>*> neighbors = u->connectedTo;
        int i = 0;
        done = false;
        while(i < neighbors.size() && !done){
            if( neighbors[i]->to->color == 'w' ){
                done = knightTour(n+1, path, neighbors[i]->to,
limit);
            }
            i++;
        }
        if(!done){
            path.pop_back();
            u->color = 'w';
        }
    }else{
        done = true;
    }
    return done;
}
```

Implementación de la gira del caballo

- `knightTour` es recursiva:
  - **`Base:`** si tenemos una ruta que contiene 64 vértices, regresamos `true`.
  - **`Recursivo`**: si no tiene la profundidad esperada seguimos explorando.

```
bool knightTour(int n, vector<int> &path, Vertex<int>* u, int
limit){
    bool done;
    u->color = 'g';
    path.push_back(u-> data);
    if(n < limit){
        vector<Edge<int>*> neighbors = u->connectedTo;
        int i = 0;
        done = false;
        while(i < neighbors.size() && !done){
            if( neighbors[i]->to->color == 'w' ){
                done = knightTour(n+1, path, neighbors[i]->to,
limit);
            }
            i++;
        }
        if(!done){
            path.pop_back();
            u-> color = 'w';
        }
    }else{
        done = true;
    }
    return done;
}
```

Implementación de la gira del caballo

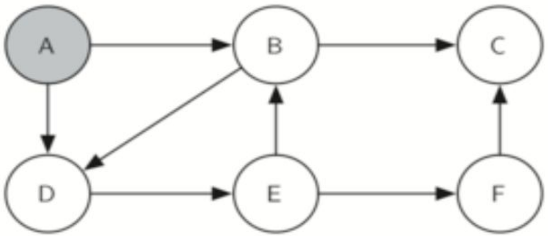
- BEP realiza seguimiento de los vértices visitados con colores.
  - w, no visitado.
  - g, visitado.
- Callejón sin salida si todos los vecinos han sido explorados.
  - Retrocedemos, devolviendo `false`.
  - El ciclo `while` continúa examinando el siguiente vértice.

```
bool knightTour(int n, vector<int> &path, Vertex<int>* u, int
limit){
    bool done;
    u->color = 'g';
    path.push_back(u-> data);
    if(n < limit){
        vector<Edge<int>*> neighbors = u->connectedTo;
        int i = 0;
        done = false;
        while(i < neighbors.size() && !done){
            if( neighbors[i]->to->color == 'w' ){
                done = knightTour(n+1, path, neighbors[i]->to,
limit);
            }
            i++;
        }
        if(!done){
            path.pop_back();
            u-> color = 'w';
        }
    }else{
        done = true;
    }
    return done;
```

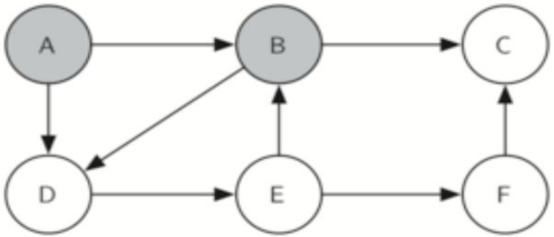
Implementación de la gira del caballo

```
knightTour(0, ruta, A, 6)
```

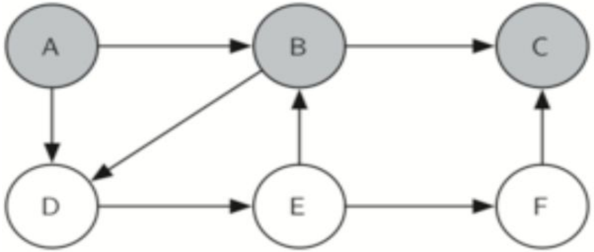
(1)



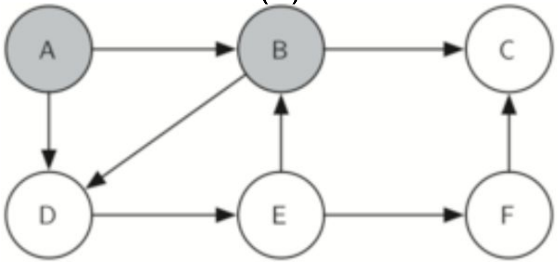
(2)



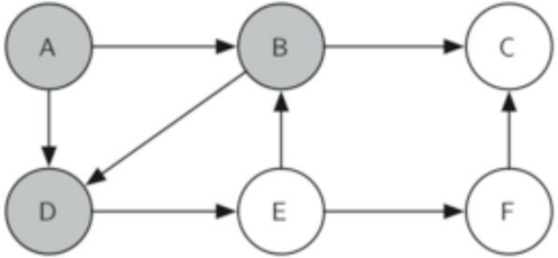
(3)



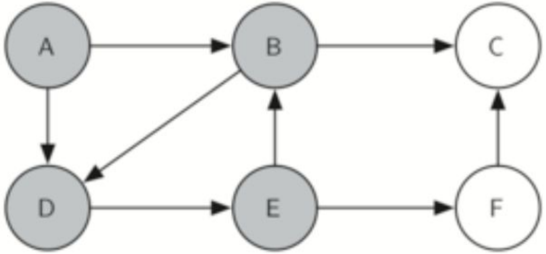
(4)



(5)



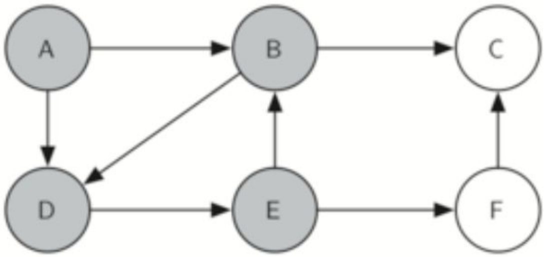
(6)



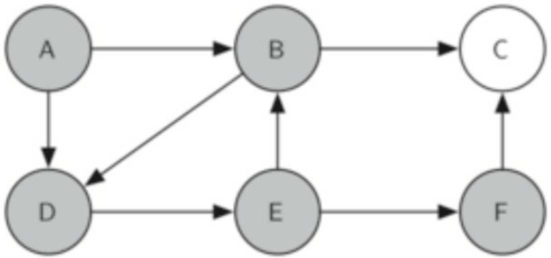
Implementación de la gira del caballo

```
knightTour(0, ruta, A, 6)
```

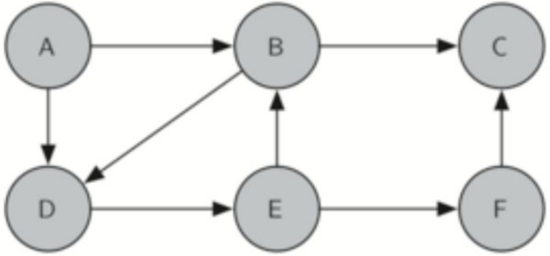
(6)



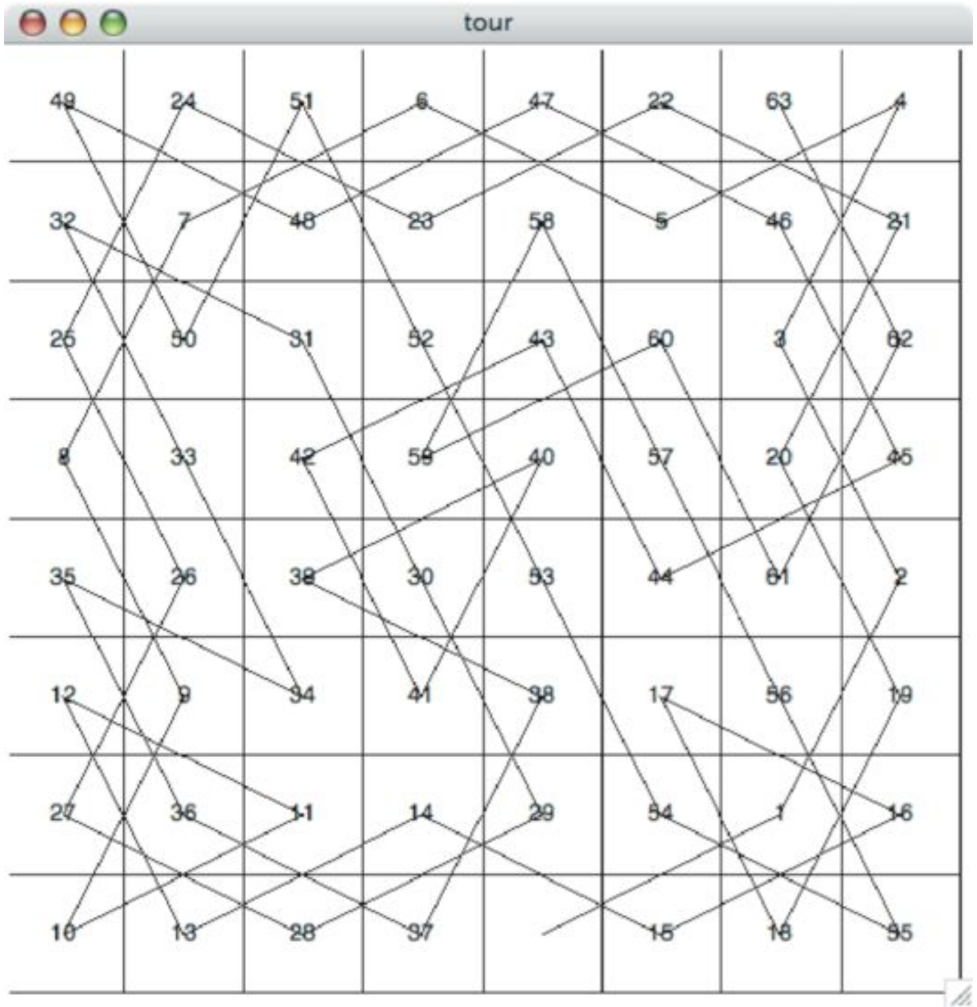
(7)



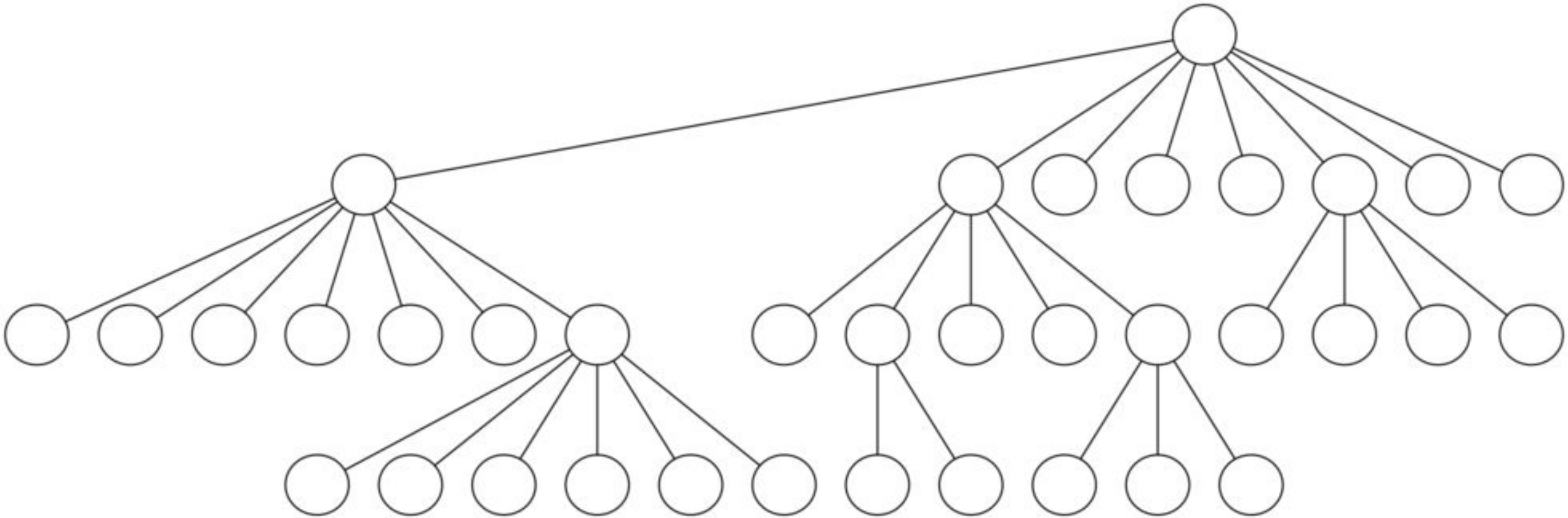
(8)



Implementación de la gira del caballo



Análisis de la gira del caballo $O(k^N)$



Análisis de la gira del caballo $O(k^N)$

2	3	4	4	4	4	3	2
3	4	6	6	6	6	4	3
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
4	6	8	8	8	8	6	4
3	4	6	6	6	6	4	3
2	3	4	4	4	4	3	2

Análisis de la gira del caballo $O(k^N)$

Número de nodos de un árbol binario es $2^{N+1} - 1$

Algoritmo es exponencial $k^{N+1} - 1$

Tablero 5x5 → 25 niveles → k = 3.8 → 3.12 x 10^{24}.

Tablero 6x6 → 36 niveles → k = 4.4 → 1.5 x 10^{23}.

Tablero 8x8 → 64 niveles → k = 5.25 → 1.3 x 10^{46}.

Análisis de la gira del caballo.

```
vector<Vertex<int>*> orderByAvailable (Vertex<int>* vertex){
    vector<Vertex<int>*> result;
    vector<tuple<int, Vertex<int>*>> listWeight;
    for(Edge<int>* v : vertex->connectedTo){
        if(v->to->color == 'w'){
            int c = 0;
            for(Edge<int>* w : v->to->connectedTo){
                if(w->to->color == 'w'){
                    c++;
                }
            }
            listWeight.push_back(tuple< int, Vertex<int>*>(c, v->to));
        }
    }
    sort(listWeight.begin(), listWeight.end(), [](tuple<int, Vertex<int>*> t1, tuple<int, Vertex<int>*>t2){
        return get<0>(t1) < get<0>(t2);
    });
    for(auto[c, v] : listWeight){
        result.push_back(v);
    }
    return result;
}
```

Solución final

```
int main() {
   Graph<int> g;
   int boardSize = 8;
   g = KnightGraph(boardSize);
   vector<int> path;
   bool done = knightTour(0, path, g.getVertex(4),
boardSize*boardSize-1);
   if(done) {
       for (int value: path) {
           cout << value << "\t ";
       }
   }else{
       cout << "No pudo alcanzar una solución" ;
   }

   return 0;
}
```

Búsqueda en profundidad general

Búsqueda en profundidad general

Buscar lo más profundamente posible, conectando tantos nodos en el grafo como sea posible y ramificando donde sea necesario.

```
template<class T>
class Vertex{
public:
    ...
    int discovery;
    int finish;
    ...
};
```

* tiempo de descubrimiento

** tiempo de finalización

* Tiempo de descubrimiento: número de pasos para encontrar el vértice.
** Tiempo de finalización: número de pasos para pintar de negro.

Implementación de búsqueda en profundidad general

```
void bep(Vertex<string>* curVertex, int &tiempo){
    curVertex->color = 'g';
    tiempo++;
    curVertex->discovery = tiempo;
    for(Edge<string>* neighbor: curVertex->getConnectedTo()){
        if(neighbor->to->color == 'w'){
            neighbor->to->predecessor = curVertex;
            bep(neighbor->to, tiempo);
        }
    }
    curVertex->color = 'b';
    tiempo++;
    curVertex->finish = tiempo;
}
void bep(Graph<string> &graph){
    int tiempo = 0;
    for(Vertex<string>* v: graph.vertexList){
        if(v->color == 'w'){
            bep(v, tiempo);
        }
    }
}
```

Implementación de búsqueda en profundidad general

```
void bep(Vertex<string>* curVertex, int &tiempo){
    curVertex->color = 'g';
    tiempo++;
    curVertex->discovery = tiempo;
    for(Edge<string>* neighbor: curVertex->getConnectedTo()){
        if(neighbor->to->color == 'w'){
            neighbor->to->predecessor = curVertex;
            bep(neighbor->to, tiempo);
        }
    }
    curVertex->color = 'b';
    tiempo++;
    curVertex->finish = tiempo;
}
void bep(Graph<string> &graph){
    int tiempo = 0;
    for(Vertex<string>* v: graph.vertexList){
        if(v->color == 'w'){
            bep(v, tiempo);
        }
    }
}
```

El método `bep` itera sobre todos los vértices llamando a la función recursiva.

Implementación de búsqueda en profundidad general

```
void bep(Vertex<string>* curVertex, int &tiempo){
    curVertex->color = 'g';
    tiempo++;
    curVertex->discovery = tiempo;
    for(Edge<string>* neighbor: curVertex->getConnectedTo()){
        if(neighbor->to->color == 'w'){
            neighbor->to->predecessor = curVertex;
            bep(neighbor->to, tiempo);
        }
    }
    curVertex->color = 'b';
    tiempo++;
    curVertex->finish = tiempo;
}
void bep(Graph<string> &graph){
    int tiempo = 0;
    for(Vertex<string>* v: graph.vertexList){
        if(v->color == 'w'){
            bep(v, tiempo);
        }
    }
}
```

El método `bep` itera sobre todos los vértices llamando a la función recursiva.

El método recursivo comienza **`curVertex`** y explora todo sus vecinos en profundidad.

Implementación de búsqueda en profundidad general

```
void bep(Vertex<string>* curVertex, int &tiempo){
    curVertex->color = 'g';
    tiempo++;
    curVertex->discovery = tiempo;
    for(Edge<string>* neighbor: curVertex->getConnectedTo()){
        if(neighbor->to->color == 'w'){
            neighbor->to->predecessor = curVertex;
            bep(neighbor->to, tiempo);
        }
    }
    curVertex->color = 'b';
    tiempo++;
    curVertex->finish = tiempo;
}
void bep(Graph<string> &graph){
    int tiempo = 0;
    for(Vertex<string>* v: graph.vertexList){
        if(v->color == 'w'){
            bep(v, tiempo);
        }
    }
}
```

El método **bep** itera sobre todos los vértices llamando a la función recursiva.

El método recursivo comienza **curVertex** y explora todo sus vecinos en profundidad.

Aquí está la diferencia con la búsqueda en anchura, y es la búsqueda inmediata de su sucesor, en lugar de agregarlo a la cola.

Implementación de búsqueda en profundidad general

```
void bep(Vertex<string>* curVertex, int &tiempo){
    curVertex->color = 'g';
    tiempo++;
    curVertex->discovery = tiempo;
    for(Edge<string>* neighbor: curVertex->getConnectedTo()){
        if(neighbor->to->color == 'w'){
            neighbor->to->predecessor = curVertex;
            bep(neighbor->to, tiempo);
        }
    }
    curVertex->color = 'b';
    tiempo++;
    curVertex->finish = tiempo;
}
void bep(Graph<string> &graph){
    int tiempo = 0;
    for(Vertex<string>* v: graph.vertexList){
        if(v->color == 'w'){
            bep(v, tiempo);
        }
    }
}
```

El método **bep** itera sobre todos los vértices llamando a la función recursiva.

El método recursivo comienza **curVertex** y explora todo sus vecinos en profundidad.

Aquí está la diferencia con la búsqueda en anchura, y es la búsqueda inmediata de su sucesor, en lugar de agregarlo a la cola.

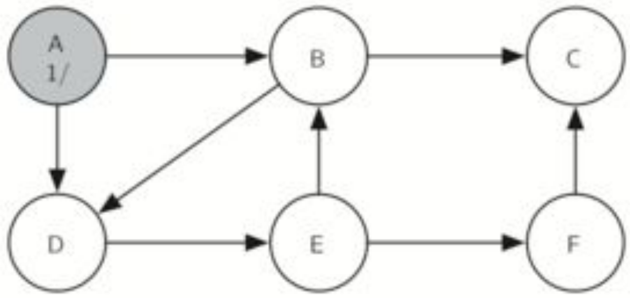
Note el cambio de color y los tiempos de descubrimiento y finalización. ¿Hay alguna razón para que tiempo sea referencia?

Ejemplo de búsqueda en profundidad general

tiempo | 0
vértice padre | None

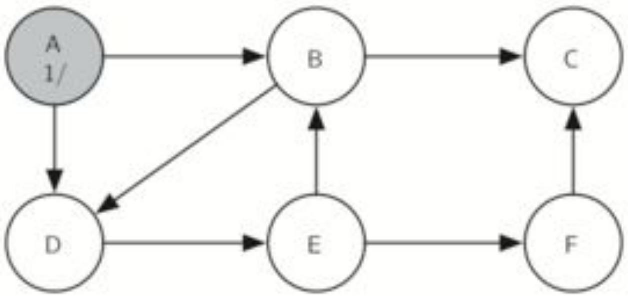
Ejemplo de búsqueda en profundidad general

tiempo | 1
vértice padre | None

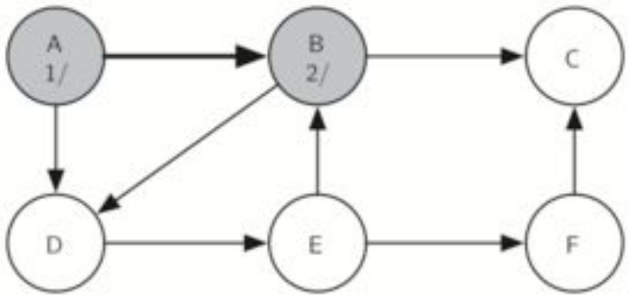


Ejemplo de búsqueda en profundidad general

tiempo | 1
vértice padre | None

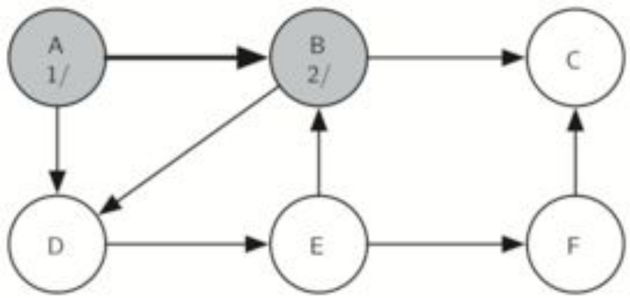


tiempo | 2
vértice padre | A



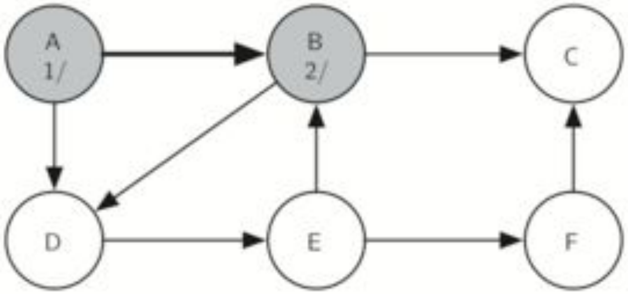
Ejemplo de búsqueda en profundidad general

tiempo | 2
vértice padre | A

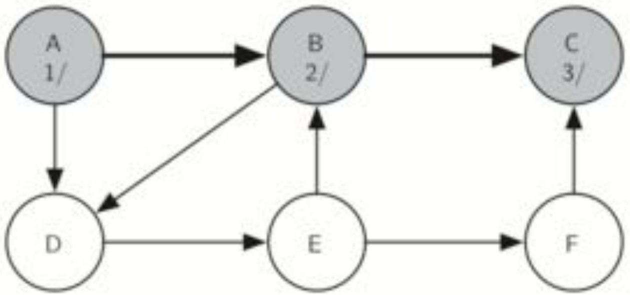


Ejemplo de búsqueda en profundidad general

tiempo | 2
vértice padre | A

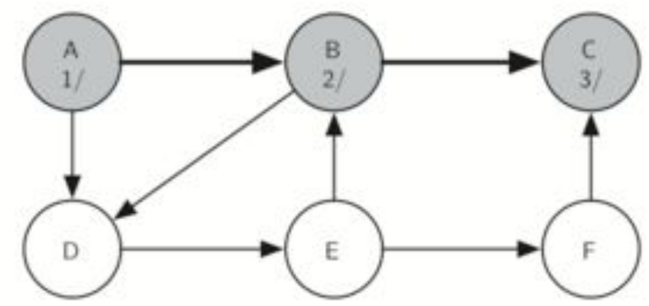


tiempo | 3
vértice padre | B



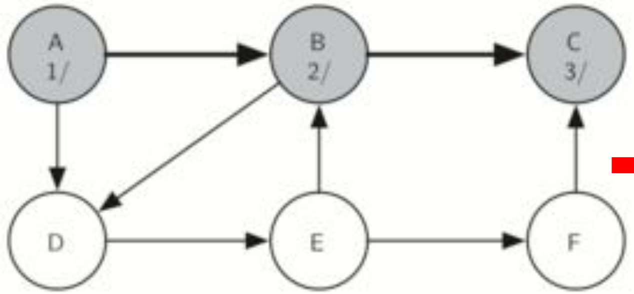
Ejemplo de búsqueda en profundidad general

tiempo | 3
vértice padre | B

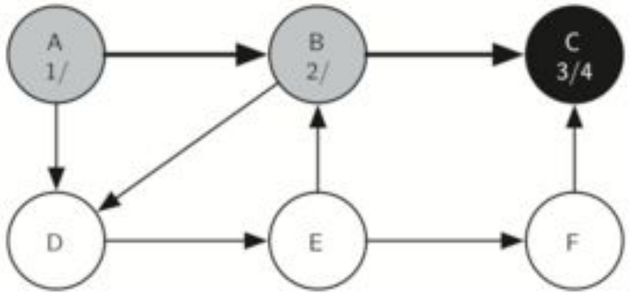


Ejemplo de búsqueda en profundidad general

tiempo | 3
vértice padre | B

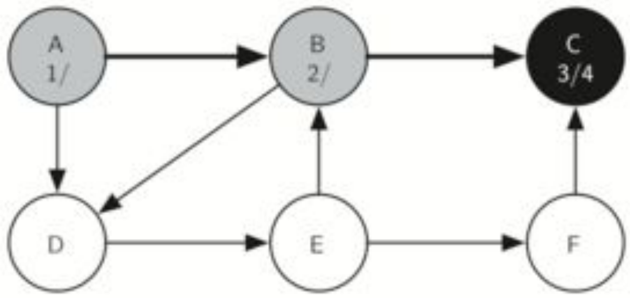


tiempo | 4
vértice padre | B



Ejemplo de búsqueda en profundidad general

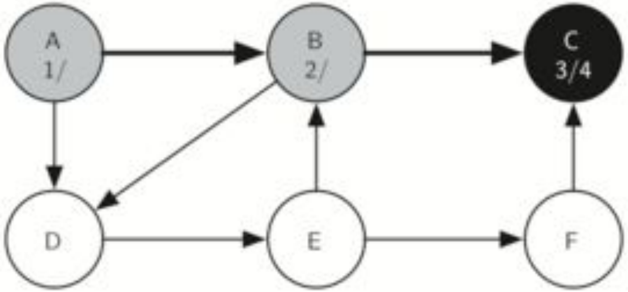
tiempo | 4
vértice padre | B



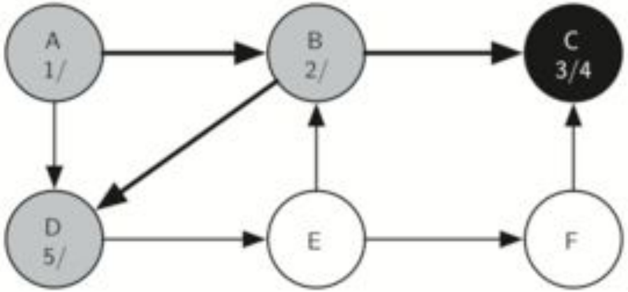
tiempo | 4
vértice padre | B

Ejemplo de búsqueda en profundidad general

tiempo | 4
vértice padre | B

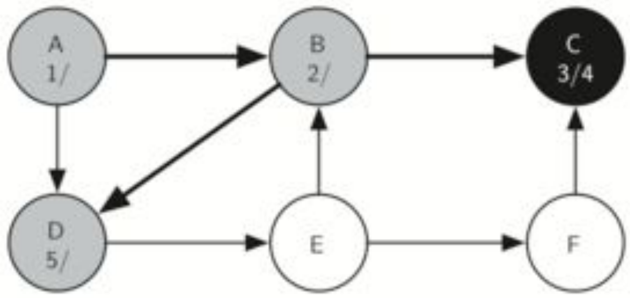


tiempo | 5
vértice padre | B



Ejemplo de búsqueda en profundidad general

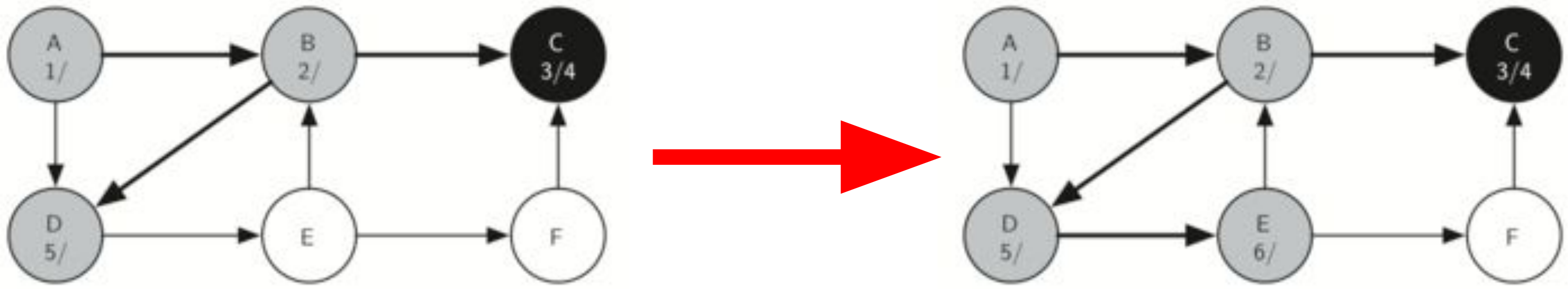
tiempo | 5
vértice padre | B



Ejemplo de búsqueda en profundidad general

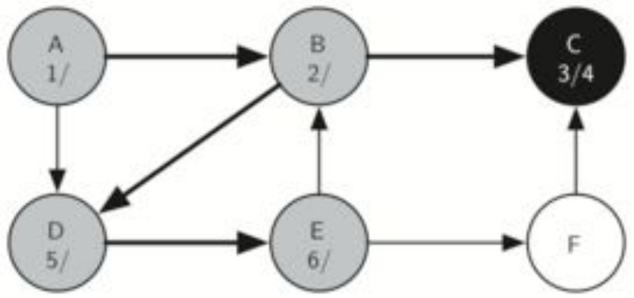
tiempo | 5
vértice padre | B

tiempo | 6
vértice padre | D



Ejemplo de búsqueda en profundidad general

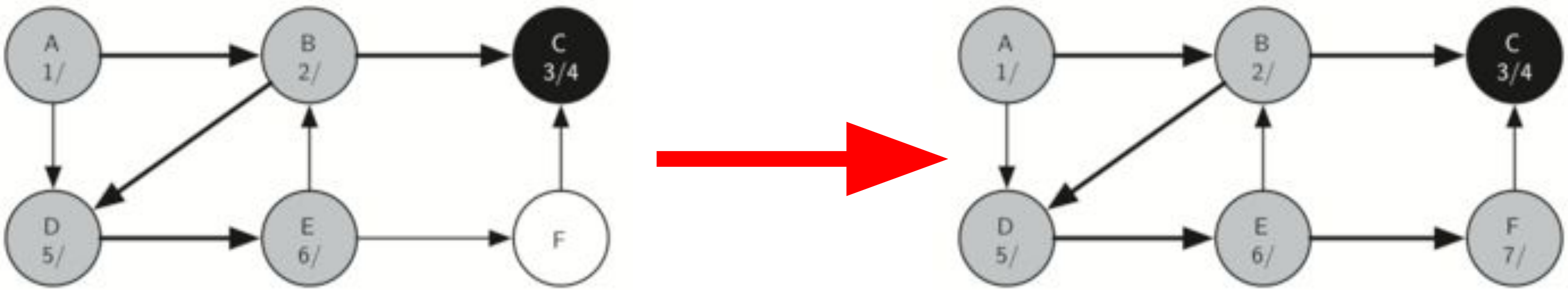
tiempo | 6
vértice padre | D



Ejemplo de búsqueda en profundidad general

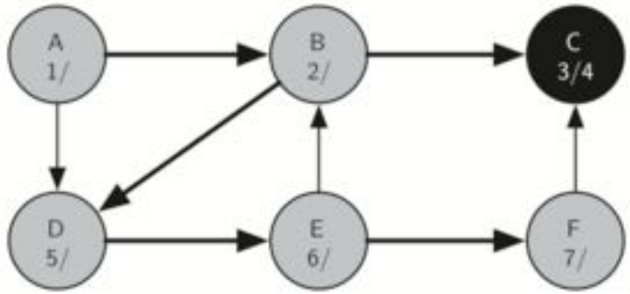
tiempo | 6
vértice padre | D

tiempo | 7
vértice padre | E



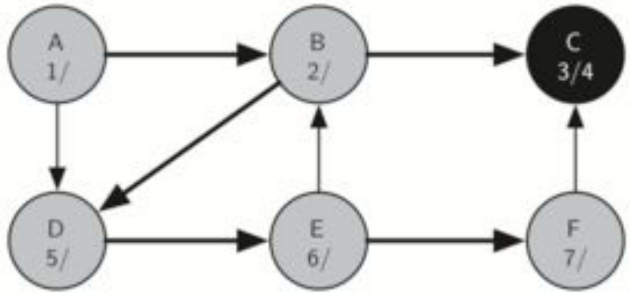
Ejemplo de búsqueda en profundidad general

tiempo | 7
vértice padre | E

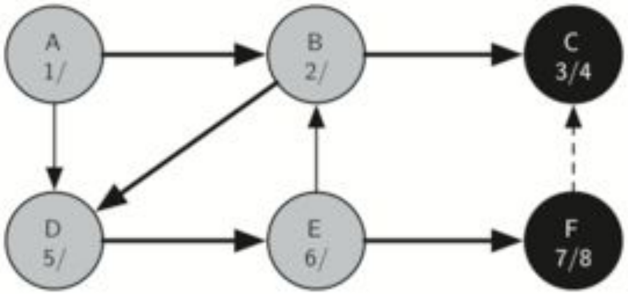


Ejemplo de búsqueda en profundidad general

tiempo | 7
vértice padre | E

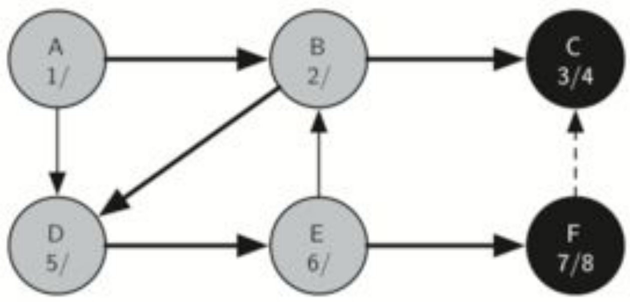


tiempo | 8
vértice padre | E



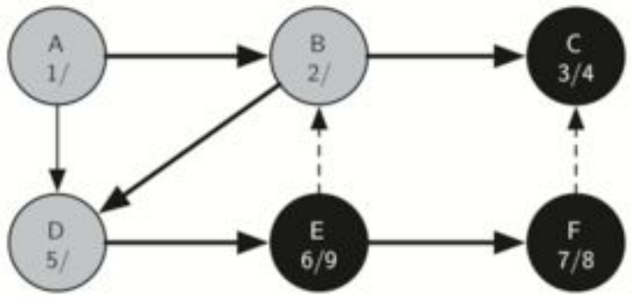
Ejemplo de búsqueda en profundidad general

tiempo | 8
vértice padre | E



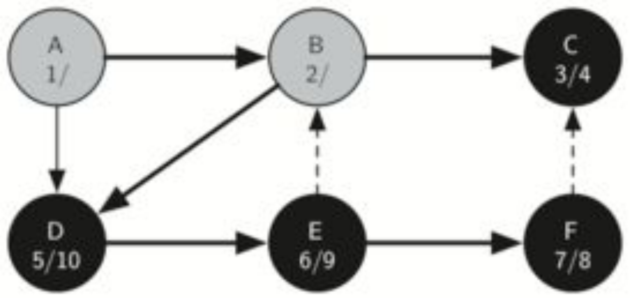
Ejemplo de búsqueda en profundidad general

tiempo | 9
vértice padre | D



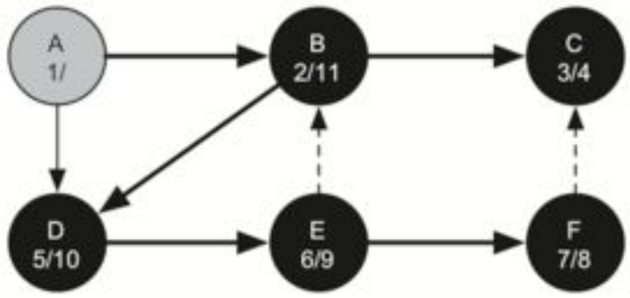
Ejemplo de búsqueda en profundidad general

tiempo | 10
vértice padre | B



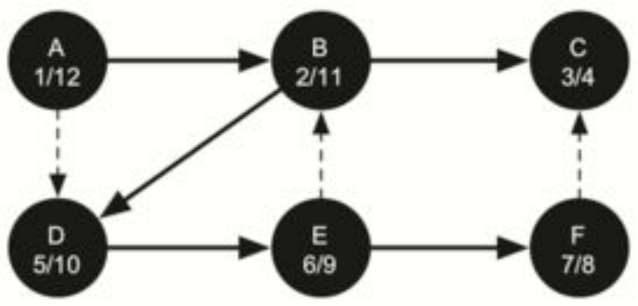
Ejemplo de búsqueda en profundidad general

tiempo | 11
vértice padre | A

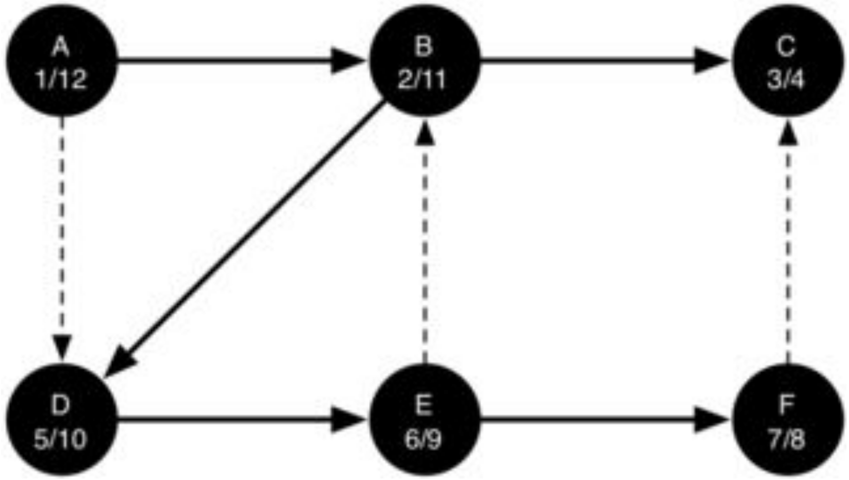


Ejemplo de búsqueda en profundidad general

tiempo | 12
vértice padre | None



Ejemplo de búsqueda en profundidad general O(V + E)

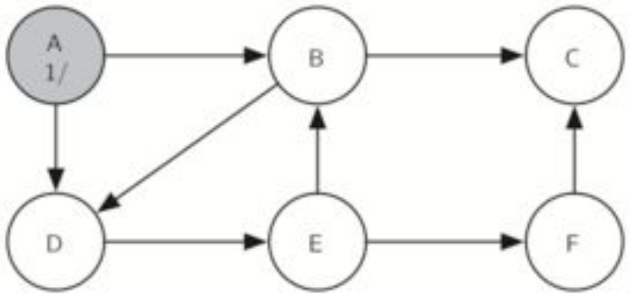


Los tiempos de inicio y finalización de cada nodo muestran una propiedad denominada **Propiedad del paréntesis**.

```cpp
int main() {
   Graph<string> g;
   g.addVertex("A");
   g.addVertex("B");
   g.addVertex("C");
   g.addVertex("D");
   g.addVertex("E");
   g.addVertex("F");

   g.addEdge("A", "B");
   g.addEdge("A", "D");

   g.addEdge("B", "C");
   g.addEdge("B", "D");

   g.addEdge("D", "E");

   g.addEdge("E", "B");
   g.addEdge("E", "F");

   g.addEdge("F", "C");

   cout << "Realizando BEP desde vertice"  << endl;
   bep(g);
   cout << "Imprimiendo recorrido"  << endl;
   traversal(g.getVertex("F")); // palabra final*/
   return 0;
}
```

output:

```
Imprimiendo recorrido
F
E
D
B
A
```

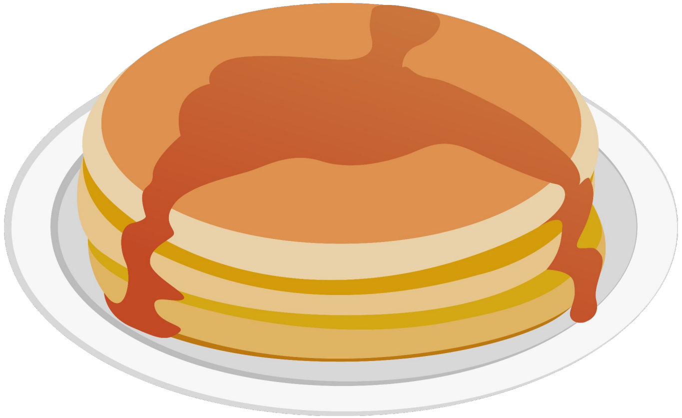


Ordenamiento topológico

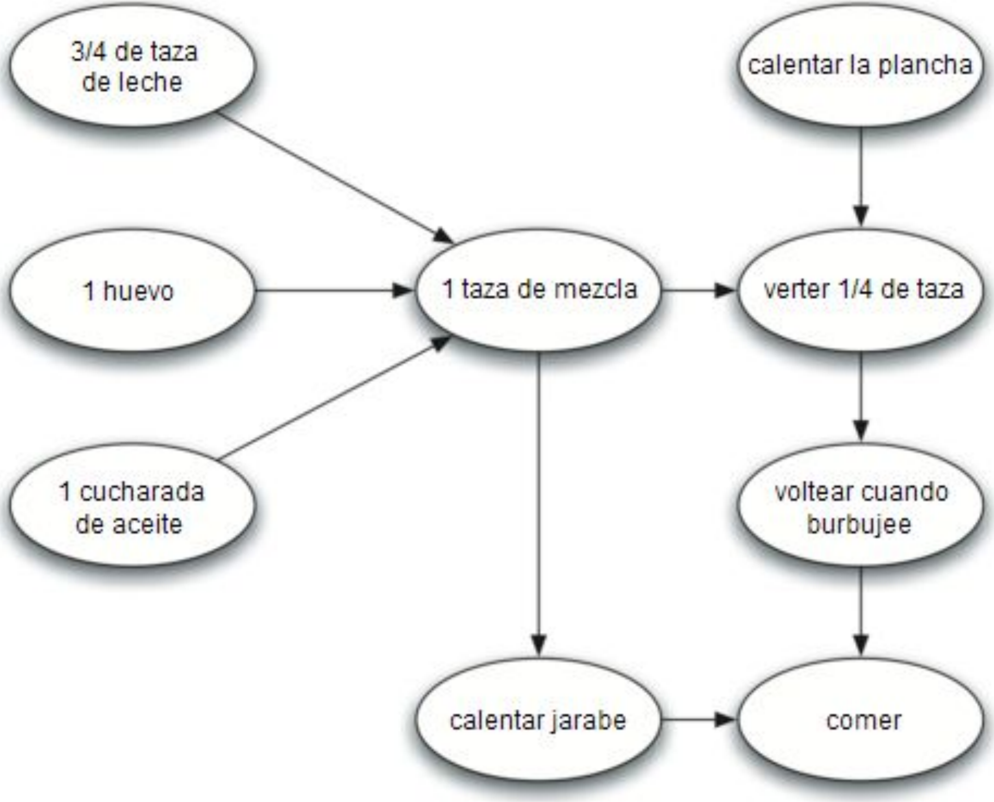
Motivación

Receta para panqueques:

" 1 huevo, 1 taza de mezcla de panqueques, 1 cucharada de aceite y 3/4 de una taza de leche. Para hacer panqueques usted debe calentar la plancha, mezclar todos los ingredientes y derramar la mezcla sobre una plancha caliente. Cuando los panqueques empiecen a burbujear, déles vuelta y deje que se cocinen hasta que estén dorados en la parte de abajo. Antes de comer sus panqueques, usted querrá calentar un poco de jarabe dulce".

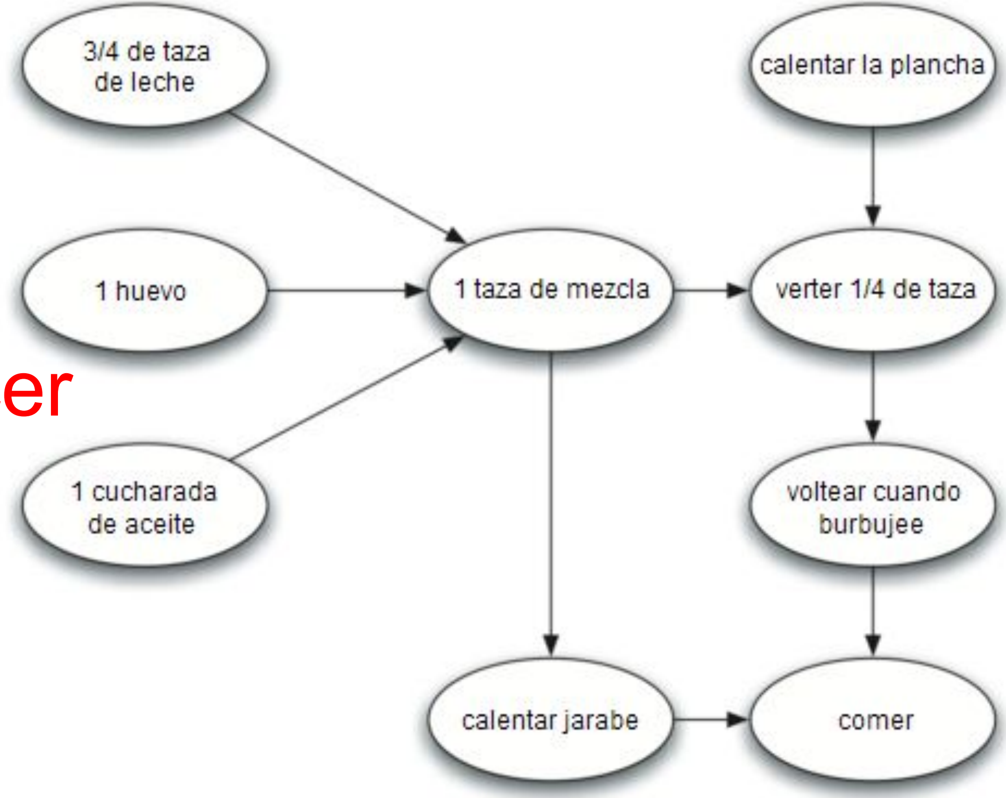


Motivación



3/4 de taza de leche
calentar la plancha
1 huevo
1 taza de mezcla
verter 1/4 de taza
1 cucharada de aceite
voltear cuando burbujee
calentar jarabe
comer

Motivación



¿Qué hacer primero?

Ordenamiento topológico

Toma un GAD o grafo acíclico dirigido ***G*** y produce un ordenamiento lineal de todos sus vértices.

- Para una arista ***(v, w)*** el vértice ***v*** está antes de ***w***.

Los GAD indican la procedencia de los eventos:

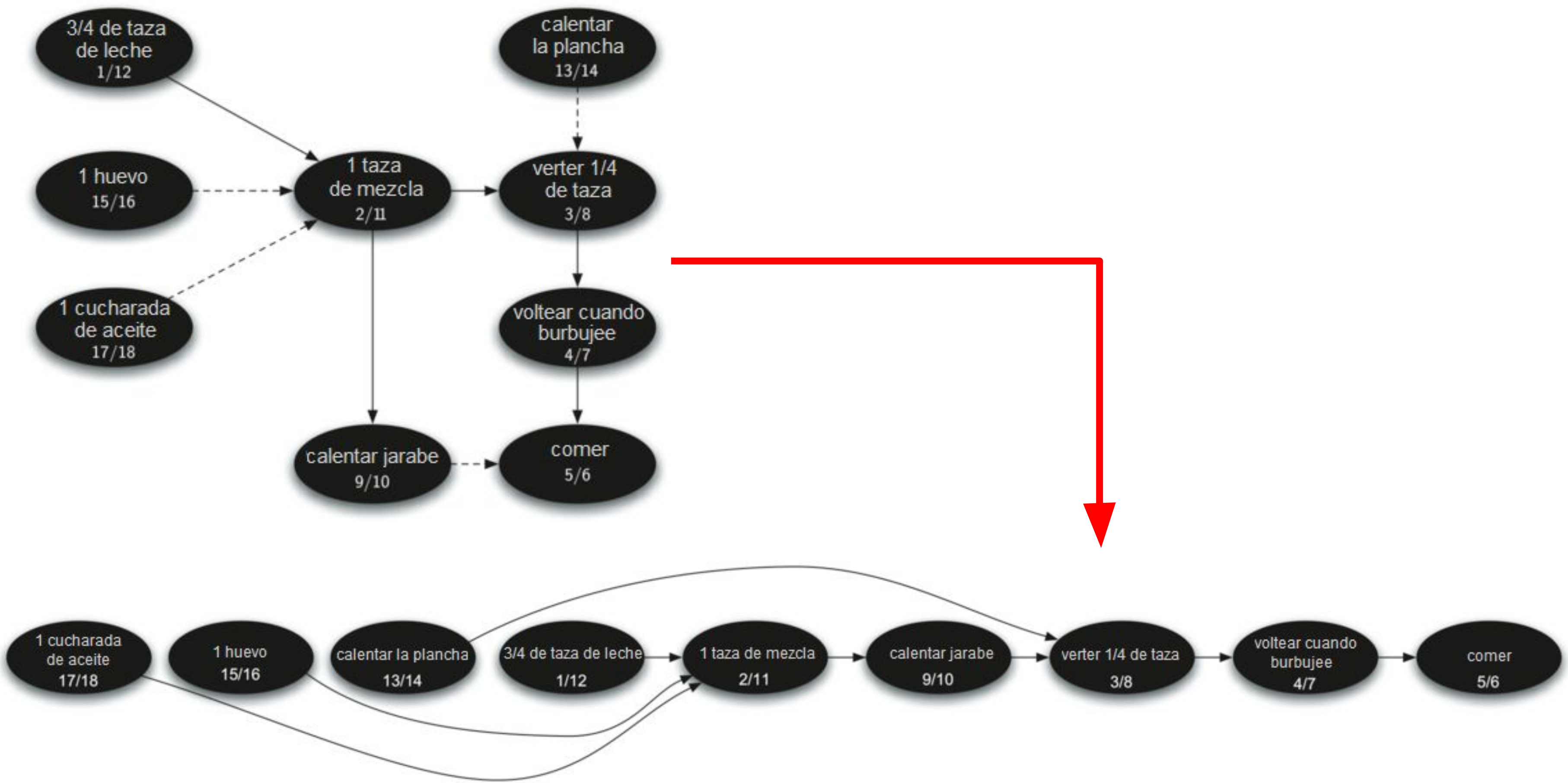
- Planificadores de proyectos.
- Grafos de precedencia para optimizar las consultas de bases de atos.
- Multiplicación de matrices.

Implementación

Adaptación de búsqueda en profundidad (BEP).

1. Llamar `bep(g)` para algún grafo `g`. ← calcular los tiempo de finalización.
2. Almacenar los vértices en una lista en orden decreciente a los tiempos de finalización.
3. Devolver la lista ordenada.

Implementación


3/4 de taza de leche 1/12
calentar la plancha 13/14
1 huevo 15/16
1 taza de mezcla 2/11
verter 1/4 de taza 3/8
1 cucharada de aceite 17/18
voltear cuando burbujee 4/7
calentar jarabe 9/10
comer 5/6
1 cucharada de aceite 17/18
1 huevo 15/16
calentar la plancha 13/14
3/4 de taza de leche 1/12
1 taza de mezcla 2/11
calentar jarabe 9/10
verter 1/4 de taza 3/8
voltear cuando burbujee 4/7
comer 5/6

```cpp
int main() {
   Graph<string> g;
   g.addVertex("3/4 de taza de leche");
   g.addVertex("calentar la plancha");
   g.addVertex("1 huevo");
   g.addVertex("1 cucharada de aceite");
   g.addVertex("1 taza de mezcla");
   g.addVertex("calentar jarabe");
   g.addVertex("verter 1/4 de taza");
   g.addVertex("voltear cuando burbujee");
   g.addVertex("comer");

   g.addEdge("3/4 de taza de leche","1 taza de mezcla");
   g.addEdge("1 huevo","1 taza de mezcla");
   g.addEdge("1 cucharada de aceite","1 taza de mezcla");
   g.addEdge("1 taza de mezcla","verter 1/4 de taza");
   g.addEdge("1 taza de mezcla","calentar jarabe");
   g.addEdge("calentar la plancha","verter 1/4 de taza");
   g.addEdge("verter 1/4 de taza","voltear cuando burbujee");
   g.addEdge("voltear cuando burbujee","comer");
   g.addEdge("calentar jarabe","comer");

   ...
}
```

```cpp
int main() {
   Graph<string> g;
   g.addVertex( "3/4 de taza de leche" );
   g.addVertex( "calentar la plancha" );
   g.addVertex( "1 huevo" );
   g.addVertex( "1 cucharada de aceite" );
   g.addVertex( "1 taza de mezcla" );
   g.addVertex( "calentar jarabe" );
   g.addVertex( "verter 1/4 de taza" );
   g.addVertex( "voltear cuando burbujee" );
   g.addVertex( "comer" );

   g.addEdge( "3/4 de taza de leche" ,"1 taza de mezcla" );
   g.addEdge( "1 huevo" ,"1 taza de mezcla" );
   g.addEdge( "1 cucharada de aceite" ,"1 taza de mezcla" );
   g.addEdge( "1 taza de mezcla" ,"verter 1/4 de taza" );
   g.addEdge( "1 taza de mezcla" ,"calentar jarabe" );
   g.addEdge( "calentar la plancha" ,"verter 1/4 de taza" );
   g.addEdge( "verter 1/4 de taza" ,"voltear cuando burbujee" );
   g.addEdge( "voltear cuando burbujee" ,"comer");
   g.addEdge( "calentar jarabe" ,"comer");

   ...
}
```

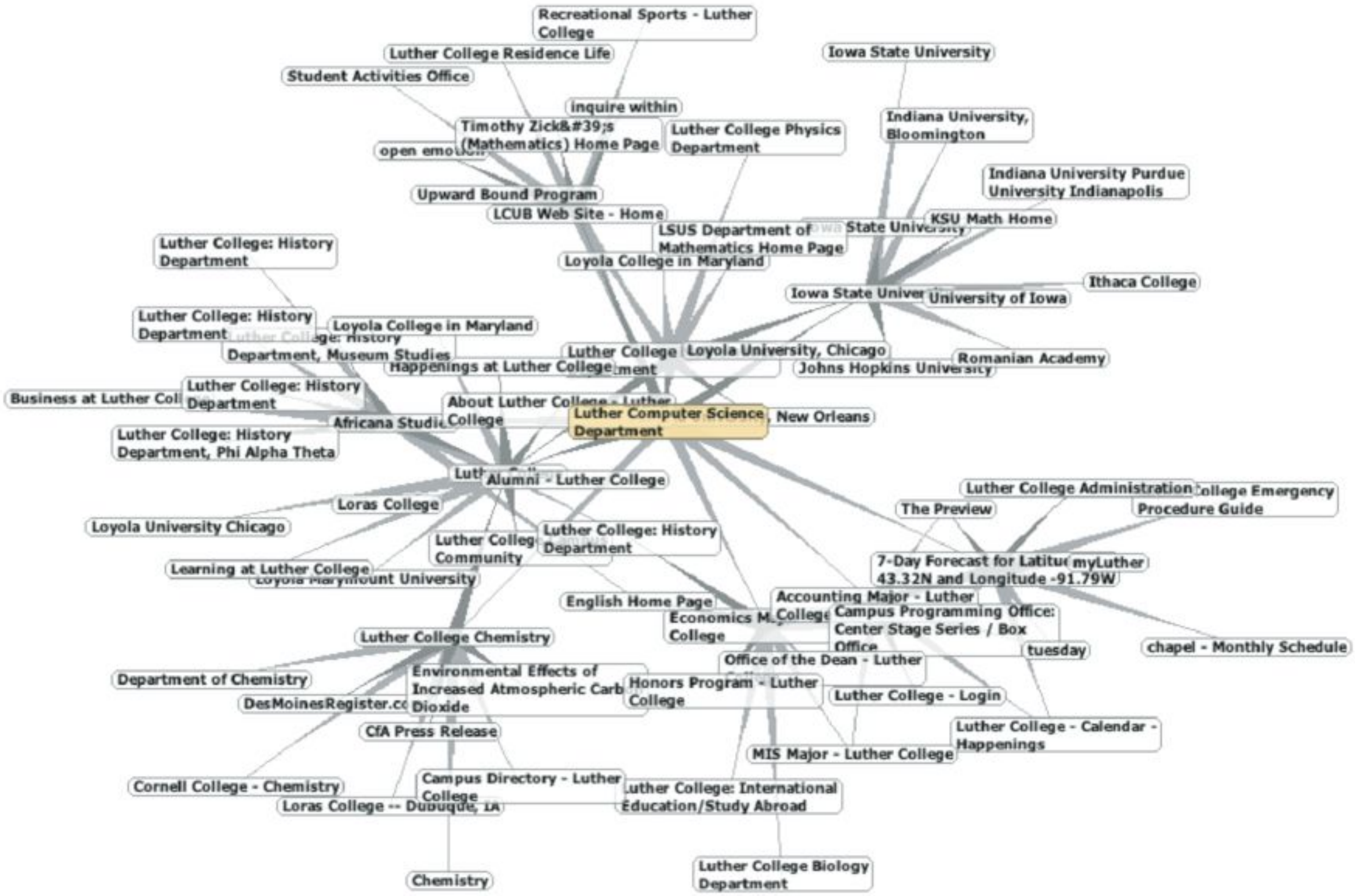
output:

```
1 cucharada de aceite
1 huevo
calentar la plancha
3/4 de taza de leche
1 taza de mezcla
verter 1/4 de taza
voltear cuando burbujee
calentar jarabe
comer
Recorrido
comer
calentar jarabe
1 taza de mezcla
3/4 de taza de leche
```

```cpp
int main() {
   ...
   bep(g);
   vector<Vertex<string>*> vertexList = g.vertexList;
   sort(vertexList.begin(), vertexList.end(), [](Vertex<string>* a, Vertex<string>* b){
       return a->finish > b->finish;
   });
   for(Vertex<string>* v : vertexList){
       cout << v->data << endl;
   }
   cout << "Recorrido" << endl;
   traversal(g.getVertex("comer"));
   return 0;
}
```

Componentes fuertemente conectados

Motivación



Recreational Sports - Luther College
Luther College Residence Life
Student Activities Office
Iowa State University
inquire within
Timothy Zick's (Mathematics) Home Page
Luther College Physics Department
Indiana University, Bloomington
Indiana University Purdue University Indianapolis
Upward Bound Program
LCUB Web Site - Home
KSU Math Home
LSUS Department of Mathematics Home Page
Luther College: History Department
Loyola College in Maryland
Ithaca College
Iowa State University
University of Iowa
Loyola College in Maryland
Department, Museum Studies
Luther College
Loyola University, Chicago
Romanian Academy
Happenings at Luther College
Johns Hopkins University
Luther College: History Department
About Luther College - Luther College
Luther Computer Science Department
New Orleans
Africana Studies
Luther College: History Department, Phi Alpha Theta
Alumni - Luther College
Luther College Administration
College Emergency Procedure Guide
Loras College
The Preview
Loyola University Chicago
Luther College: History Department
Luther College Community
7-Day Forecast for Latitude 43.32N and Longitude -91.79W
myLuther
Learning at Luther College
Loyola Marymount University
English Home Page
Accounting Major - Luther College
Economics Major College
Campus Programming Office: Center Stage Series / Box Office
Luther College Chemistry
tuesday
chapel - Monthly Schedule
Office of the Dean - Luther College
Department of Chemistry
Environmental Effects of Increased Atmospheric Carbon Dioxide
Honors Program - Luther College
Luther College - Login
CfA Press Release
Luther College - Calendar - Happenings
MIS Major - Luther College
Cornell College - Chemistry
Campus Directory - Luther College
Luther College: International Education/Study Abroad
Loras College -- Dubuque, IA
Luther College Biology Department
Chemistry

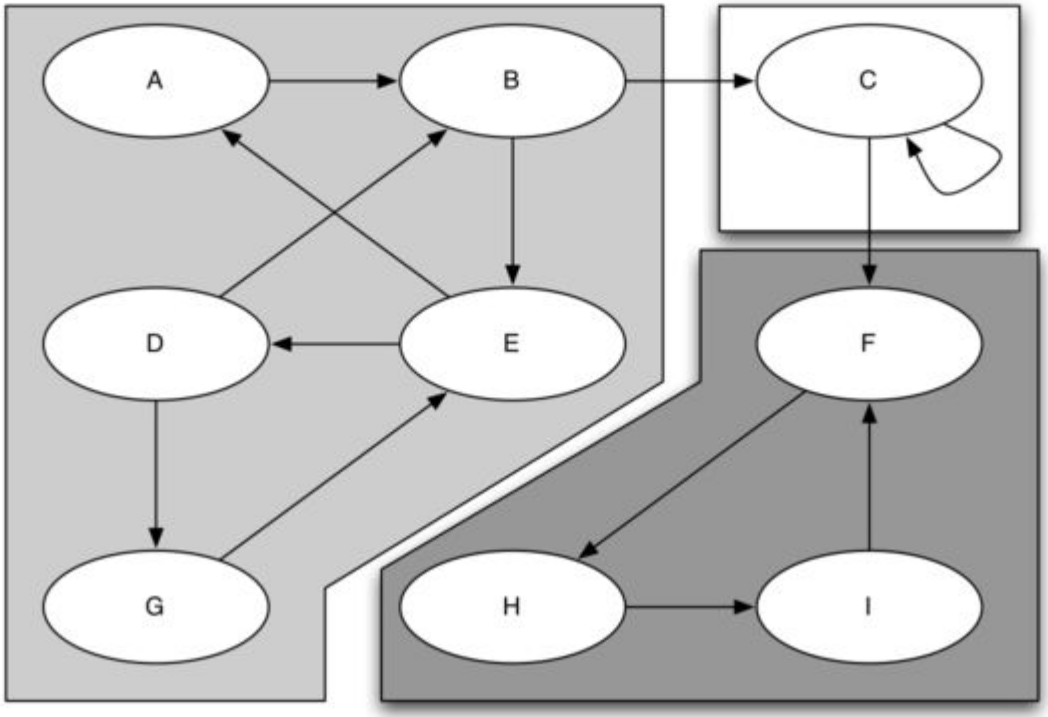
Componentes fuertemente conectados (CFC)

Ayudar a encontrar grupos de vértices altamente interconectados en un grafo.

Definición:

- Un **componente fuertemente conectado**, C, de un grafo G, es el mayor subconjunto de vértices $C \subset V$ tal que para cada pareja de vértices $v, w \in C$ tenemos una ruta desde v hasta w y una ruta desde w hasta v.

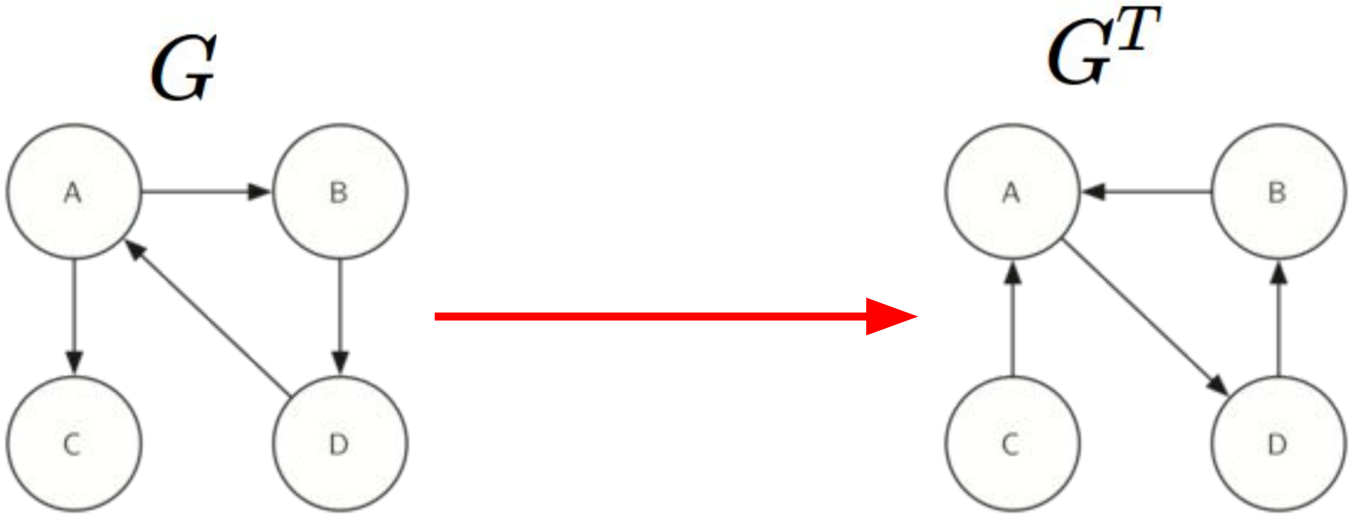
Componentes fuertemente conectados (CFC)



Un **componente fuertemente conectado**, ***C***, de un grafo ***G***, es el mayor subconjunto de vértices ***C*** ⊂ ***V*** tal que para cada pareja de vértices ***v***, ***w*** ∈ ***C*** tenemos una ruta desde ***v*** hasta ***w*** y una ruta desde ***w*** hasta ***v***.

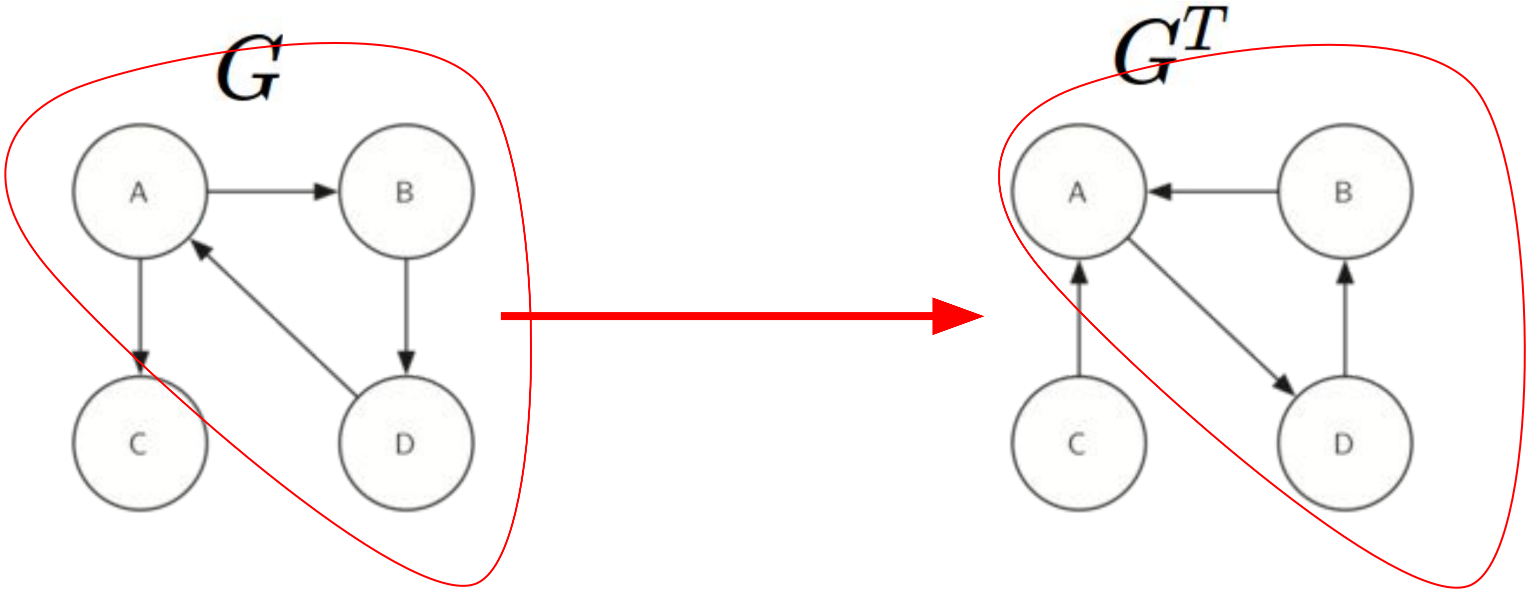
Transposición de un grafo. G^T

La transposición de un grafo G se define como el grafo G^T donde se ha invertido todas las aristas del grafo.



Transposición de un grafo. G^T

La transposición de un grafo G se define como el grafo G^T donde se ha invertido todas las aristas del grafo.



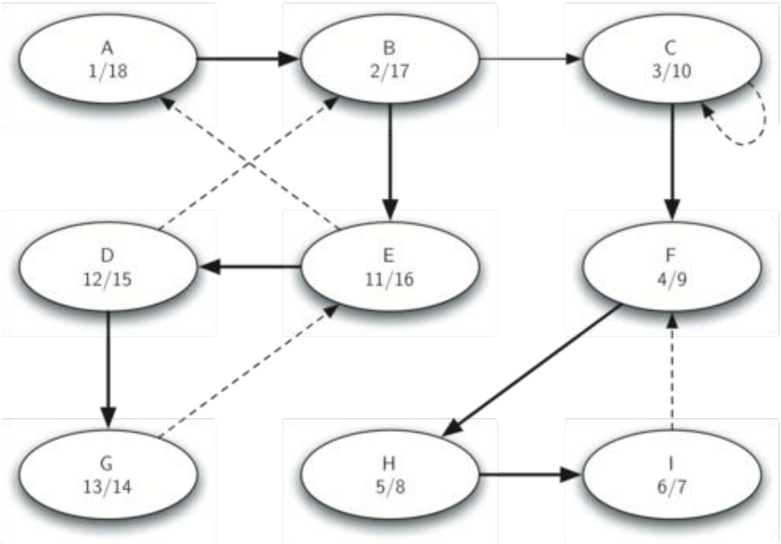
Implementación

Algoritmo para calcular los componentes fuertemente conectados:

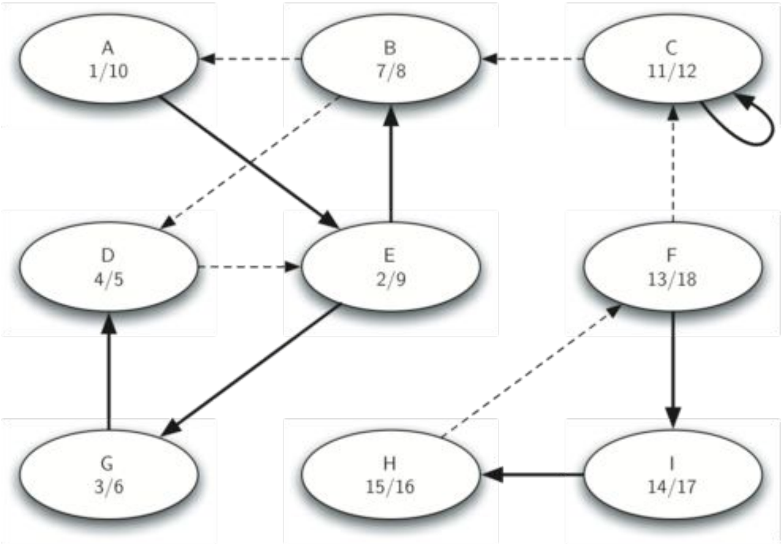
1. Llamar a bep(G) $\leftarrow$ calcular los tiempos de finalización de cada vértice.
2. Calcular G^T .
3. Llamar a bep G^T) $\leftarrow$ explorar cada vértice en orden decreciente de tiempo finalización del recorrido anterior.
4. Cada árbol de en bloque anterior es un componente fuertemente conectado.

Implementación

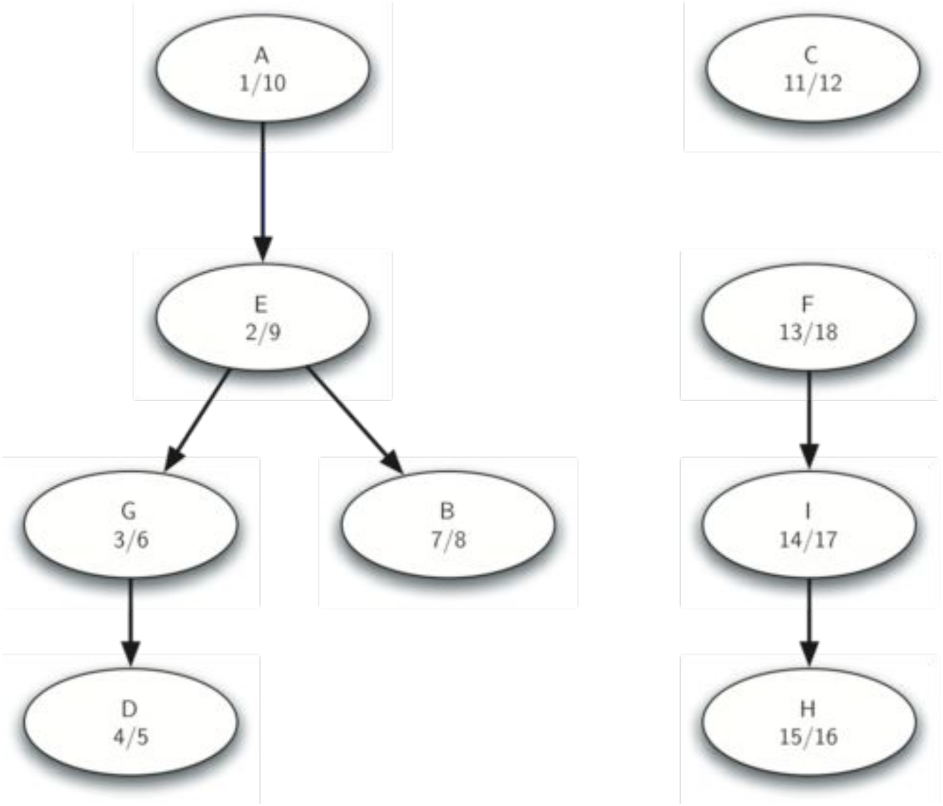
G



G^T



Implementación



Gracias