C4

Grafos y DFS

Grafos

Son formas de representar una situación de la vida real, por ejemplo, una red de computadores interconectados, una red de tuberías, transiciones entre estados, etcétera. Un grafo **G** es un conjunto de nodos **V** y un conjunto de aristas **E** que unen pares de nodos.

Existen **grafos cíclicos** que conforman un camino cerrado en el que no se repite ningún vértice a excepción del primero que aparece dos veces como principio y fin del camino. Por lo tanto, el número de vértices es igual al número de aristas. Además, cada vértice tiene grado par.

Relación 'es posterior a'

Una tarea Y es posterior a una tarea X si se cumple que:

$$X \to Y$$

o

Existe una tarea Z tal que $X \to Z$, y Y **es posterior a** Z

Lo que significa que X debe realizarse antes que Y. Por lo tanto, si una tarea forma parte de un ciclo si es posterior a sí misma. Para identificar las tareas posteriores a una tarea, se tiene el siguiente algoritmo:

```
posteriores(X):
    if X está pintado: return ∅
    pintar X
    P ← ∅
    for Y tal que X → Y:
        P ← P ∪ {Y}
        P ← P ∪ posteriores(Y)
    return P
```

A partir de lo anterior, si el nodo recién descubierto, Y, está pintado, hay dos posibilidades:

1. Si lo descubrió un nodo posterior a Y, hay ciclo.
2. Si lo descubrió un nodo anterior a Y, no hay ciclo (aún).

Hasta que `posteriores(X)` retorne, todos los nodos explorados son posteriores a X. Luego, nace el siguiente algoritmo:

```
hay ciclo luego de(X):
    if X está pintado de gris: return true
    if X está pintado de negro: return false
    Pintar X de gris
    for Y tal que X → Y:
        if hay ciclo luego de(Y), return true
    Pintar X de negro
    return false

hay ciclo en(G(V, E)):
    for X ∈ V:
        if X está pintado, continue
        if hay ciclo luego de (X):
            return true
    return false
```

A estos algoritmos se les conoce como **búsqueda en profundidad (DFS)** ya que llegan al final de una rama antes de empezar a explorar la otra. Su complejidad dependerá netamente de la implementación de la búsqueda.

Representación de grafos en memoria

Hay dos maneras de representar un grado:

1. Lista de adyacencias: Útiles para representar grafos no direccionados, en donde cada nodo tiene una lista de los nodos a los que tiene una arista. La complejidad es de $O(V+E)$.
2. Matriz de adyacencias: Útiles para representar grafos direccionados, en donde la coordenada x, y de la matriz indica si la arista (x, y) está en el grafo. La complejidad es de $O(V^2+E)$.

DFS

En la exploración en profundidad las aristas son exploradas a partir del vértice v descubierto más recientemente que aún tiene aristas no exploradas que salen de él. Cuando todas las aristas de v han sido exploradas, la exploración retrocede para explorar aristas que salen del vértice a partir del cual v fue descubierto. Luego, el algoritmo busca más profundamente en G mientras sea posible.

A medida que se realiza la exploración, DFS va 'pintando' los vértices; inicialmente son todos blancos, un vértice se pinta de gris cuando es descubierto (se registra el tiempo, **v.d**), y un vértice se pinta de negro cuando su lista de adyacencias ha sido examinada exhaustivamente (se registra el tiempo, **v.f**).

```
dfs():
  for each u in V:
     u.color = white
     π[u] = null
  time = 0
  for each u in V:
     if u.color == white:
        time = dfsVisit(u, time)

dfsVisit(u, time):
 u.color = gray
 time = time+1
 u.d = time
 for each v in α[u]:
    if v.color == white:
       π[v] = u
       time = dfsVisit(v, time)
 u.color = black
 time = time+1
 u.f = time
 return time
```

Si un intervalo, ya sea `[u.d, u.f]` o `[v.d, v.f]` está contenido dentro de otro, entonces se dirá que dicho vértice es decendiente del otro en un árbol DFS, en caso contrario, son disjuntos y no son descendientes.

Se definen cuatro tipos de aristas `(u, v)`:

1. De árbol: Si v fue descubierta por primera vez al explorar `(u, v)`.
2. Hacia atrás: Si u conecta a un ancestro v.
3. Hacia adelante: Si u conecta a un descendiente v (solo ocurre en grafos direccionales).
4. Cruzadas: Toda otra arista (solo ocurre en grafos direccionales).

Un grafo G es **direccional acíclico (DAG)** si y solo si DFS no produce aristas hacia atrás. La gracia de este tipo de grafos es que pueden ser ordenados topológicamente, es decir, ordenados linealmente acorde a todos los vértices tal que si G contiene la arista `(u, v)`, entonces u aparece antes que v en la ordenación (imposible de lograr en grafos cíclicos). Este algoritmo es el siguiente:

`topSort()`

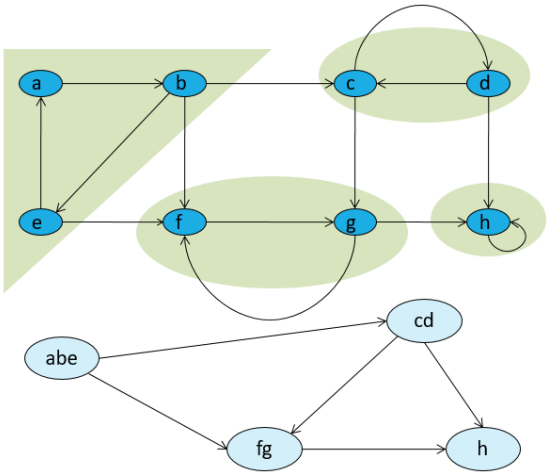
1) Ejecutamos `dfs()` para calcular los tiempos *f* para cada vértice

2) Cada vez que calculamos el tiempo *f* para un vértice, insertamos ese vértice al frente de una lista ligada

3) `return` la lista ligada de vértices

SCC's

Las **componentes fuertemente conectadas (SCC's)** de un grado direccional son conjuntos de vértices tales que para todo par de vértices u y v. son mutuamente alcanzables. De lo anterior se deduce que G y G^T tienen las mismas SCC's, por lo tanto, podemos definir G^{SCC} como un grafo direccional acíclico:



Una propiedad entre los SCC's y tiempos de finalización es que si hay una arista `(u, v)` entre dos componentes fuertemente conectados C y D (distintos) de un grafo G, entonces `f(C) > f(D)` pero si el grafo es G^T entonces `f(C) < f(D)`.

El algoritmo para encontrar SCC's de un grafo G es el siguiente:

1. Realizar DFS de G para calcular los tiempos de finalización de cada vértice.
2. Determinar G^T.
3. Realizar DFS de G^T, pero en el ciclo principal consideramos los vértices en orden decreciente en cuanto al tiempo de finalización.
4. Los vértices de cada árbol en el bosque DFS recién formado son una SCC diferente.

Tres problemas en grafos *con costos*

a) Grafos no direccionales:

- encontrar el *árbol de cobertura de costo mínimo*

b) Grafos direccionales:

- encontrar la *ruta más corta desde un vértice a todos los otros*

c) Grafos direccionales:

- encontrar las *rutas más cortas entre todos los pares de vértices*

CSP y Backtracking

CSP (Satisfacción de restricciones)

Similar a los problemas de optimización y de planificación que pueden ser abordados a partir de SAT tal que queremos encontrar una asignación a cada variable de una fórmula en lógica proposicional de tal forma que sea verdadera. Luego, si existiera un algoritmo para resolver CSP de manera eficiente, podríamos usarlo para resolver SAT, por lo que CSP es al menos tan difícil como SAT.

Resolución Backtracking

Dadas variables $x_1, \ldots, x_n$ con dominios $d_1, \ldots, d_n$, y un set de restricciones R, hay que encontrar una asignación para cada x que respete R:

```
asignar salas y horarios(C, S, i):
    if i = |C|, return true
    c = C_i
    for m ∈ Módulos:
        if c.profesor está ocupado al módulo m, continue
        for s ∈ S:
            if s está ocupada en el módulo m, continue
            Asignar clase c al horario m y sala s
            if asignar salas y horarios(C, S, i + 1):
                return true
            Desasignar clase c al horario m y sala s
    return false
```

La idea de backtracking es ir descartando permutaciones que violan alguna restricción, lo que significa que siempre resolverá el problema igual o más rápido que mediante 'fuerza bruta'.

```
is solvable(X, D, R, i):
    if i > |X|, return true
    x ← x_i
    for v ∈ d_i:
        if x = v viola R, continue
        x ← v
        if is solvable(X, D, R, i + 1):
            return true
        x ← ∅
    return false
```

Mejoras a Backtracking

1. Podas: Son restricciones adicionales deducibles de las originales que le ponemos al problema, pueden ser más costosas de revisar, pero suelen valerlo ya que cortan una parte del árbol de búsqueda.

```
is solvable(X, D, i):
    if i > |X|, return true
    x ← x_i
    for v ∈ d_i:
        if x = v no es válida, continue
        x ← v
        if is solvable(X, D, i + 1):
            return true
        x ← ∅
    return false
```

2. Propagación: Como no todos los valores de un dominio son siempre válidos, es posible invalidar valores del dominio de una variable cuando se asgina otra, por lo que es útil propagar esta información luego de una asignación para reducir efectivamente los dominios de las variables vecinas, por lo que si el tamaño del dominio es 1, entonces podemos asignar una variable y volver a propagar.

```
is solvable(X, D, R, i):
    if i > |X|, return true
    x ← x_i
    for v ∈ d_i:
        if x = v viola R, continue
        x ← v, propagar
        if is solvable(X, D, R, i + 1):
            return true
        x ← ∅ , propagar
    return false
```

3. Heurísticas: Al resolver un problema de asignación, el orden en que se asginan las variables afecta bastante el desempeño, por lo que resulta útil aproximar cuál es la mejor variable a asignar y su respectivo valor.

$is\ solvable(X, D, R)$:

$if\ X = \emptyset,\ return\ true$

$x \leftarrow$ la mejor variable de X

$for\ v \in d_i$, de mejor a peor:

$if\ x = v$ viola R, $continue$

$x \leftarrow v$

$if\ is\ solvable(X - \{x\}, D, R)$:

$return\ true$

$x \leftarrow \emptyset$

$return\ false$

Luego, usando las tres técnicas, es posible ver lo siguiente:

$is\ solvable(X, D)$:

$if\ X = \emptyset,\ return\ true$

$x \leftarrow$ la mejor variable de X

$for\ v \in d_i$, de mejor a peor:

$if\ x = v$ no es válida, $continue$

$x \leftarrow v$, propagar

$if\ is\ solvable(X - \{x\}, D)$:

$return\ true$

$x \leftarrow \emptyset$, propagar

$return\ false$