23-4-2019

Análisis de rendimiento

Aplicativo ADA

Sergio Andrés Cardona García
Johan Sebastián Saldarriaga Villada
Análisis y Diseño de Algoritmos
Universidad de Caldas

Dentro de los algoritmos implementados en el aplicativo, decidimos realizar el costeo del algoritmo de agregar ambientes al arbol, esto debido a que la funcionlidad del aplicativo esta basada en su mayoria en el arbol, pues en el analisis de un algoritmo consideramos cada ejecucion como un nuevo ambiente(nodo), teniendo en cuenta esto, es fundamental agregar un ambiente(nodo) de forma correcta; otra razon para elegir este algoritmo, es que en su mayoria los algoritmos implementados son usando librerias ajenas a nosotros como apoyo a los demas procesos.

```
# Agregar hijos desde el nivel 2 en adelante
def agregarAmbiente(self,ambienteN, padre, valor):
    if self.getRaiz() == None:                          #C1
        return False
    nodoTemporal = self.buscarN(padre,self.getRaiz())   #C2 + T(buscarN)
    if nodoTemporal != None:                            #C3
        aux = ambiente(ambienteN,valor)                 #C4
        nodoTemporal.agregarNodo(aux)                   #C5
        return True
    return False

# Buscar nodo en el arbol de ejecucion
def buscarN(self,nodo,arbol):
    if arbol == None:                   #C1
        return None
    if arbol.nombre == nodo:            #C2
        return arbol
    for i in arbol.hijos:
        aux = self.buscarN(nodo,i)      #RECURRENCIA 1
        if aux != None:                 #C3
            return  aux
    return None
```

#RECURRENCIA 1

Realizando un seguimiento al comportamiento del algoritmo *buscarN,* se llego a la conclusión que es un problema de back tracking con complejidad NP-Completo, es decir, es el subconjunto de los problemas de decisión en NP tal que todo problema en NP se puede reducir en cada uno de los problemas de NP-completo.

Basándonos en el algoritmo suma de subconjuntos (ejercicio realizado en clase), el cual tiene como mejor solución 2^n-1, ya que su factor de ramificación es 2 y teniendo en cuenta que nuestro árbol es n-ario, podemos decir que su mejor solución sería $\Theta(n^n\text{-}1)$

El siguiente algoritmo tiene un comportamiento similar a lo dicho anteriormente.

```python
# Recorrer y mostrar el arbol de ejecuciones (ambientes)
def recorrerMostrarArbolE(self, arbol):
    if arbol!=None:
        cadena = arbol.nombre+ " "
        for i in arbol.hijos:
            cadena+=self.recorrerMostrarArbolE(i)
        return cadena
    return ""
```