AG2 - Actividad Guiada 2

Nombre y Apellidos:
Gloria Angelina Estrada Galindo

Url: https://github.com/Angegloria/03MAIR---Algoritmos-de-Optimizacion---2019/blob/master/AG2/Gloria_Angelina_Estrada_Galindo_AG2.ipynb

• Desarrollar algoritmos de búsqueda en amplitud para resolver problemas

Este algoritmo se usa para buscar elementos en un grafo, en este caso con el algoritmo de busqueda de amplitud se uso para buscar el camino mas corto. Tambien se le denomina busqueda de anchura y la idea es recorrer un arbol por niveles; teniendo un nodo v, se visitan primero todos los nodos adyacentes a v, luego los nodos que esten a distancia 2 (no visitados),inmediatamente a distancia 3, y similarmente hasta recorrer todos los nodos.

In [1]:
```
import numpy as np
# breadth first search )algoritmo de busqueda de amplitud, para buscar
 el camino mas corto
graph = {'A':['B','E','C'],
         'B':['A','E','D'],
         'C':['A','F','G'],
         'D':['B','E'],
         'E':['A','B','D'],
         'F':['C'],
         'G':['C']}

def bfs(graph,start,end):

    # los nodos visitados
    visited = []

    # los caminos posibles
```

```
    paths = [[start]]

    if start == end:
        print('El nodo que empieza es el mismo que el destino.')

    while paths:
        # captura el primer camino
        path = paths.pop(0)
        node = path[-1]
        if node not in visited:
            neighbors = graph[node]
            for neighbor in neighbors:
                new_path = path.copy()
                new_path.append(neighbor)
                paths.append(new_path)
                if neighbor == end:
                    return new_path
            visited.append(node)

    return 'No hay un camino entre estos nodos'

bfs(graph,'G','A')
```

Out[1]:
```
['G', 'C', 'A']
```

• Desarrollar algoritmos de búsqueda en profundidad para resolver problemas

La busqueda en profundidad es un algoritmo de busqueda que se usa para recorrer todos los nodos de un grafo o arbol de forma ordenada,pero no uniforme. Este algortimo se encarga de buscar profundamente un determinado elemento de manera que primero analiza los nodos más profundos y por ultimo el resto.

In [2]:
```
grafo = {1:0, 2:0, 3:0, 4:1, 5:1, 7:2, 8:3, 9:3, 10:4, 11:5, 6:7, 10:9,
 12:11, 13:11}

# La variable que encuentra el numero requerido con el camino mas corto
camino=[]
```

```
def busca_profundidad (inicio, objetivo):

    """
    La funcion de busqueda en profundidad necesita encontrar
    -El valor inicial
    -El valor a encontrar
    Regresar el valor encontrado o  el 0  en caso de no encontrarlo

    """
    camino.append(inicio)


# En caso de encontrar un valor, se regresa

    if inicio == objetivo:
          return(objetivo)


# se caminan todos los elementos para encontrar el valor inicial

    for k,v in grafo.items():


        if v==inicio:

# Se nombra a la funcion recursivamente a traves del nuevo padre
            result=busca_profundidad(k, objetivo)


# Si hay un resultado significa que se encontro el elemento buscado

            if result:
                return result



    camino.pop()

 # Cuando se esta en este punto es porque se llego al fin de la profund
```

```
idad
    return 0

result=busca_profundidad(0,10)

if result:
    print(camino)

else:
    print("no encontrado")
```

```
[0, 3, 9, 10]
```

• Desarrollar algoritmos con la técnica de ramificación y poda para resolver problemas

El algoritmo de ramificacion y poda se define como un arbol de soluciones, donde cada rama nos lleva a una posible solucion, tambien es un algoritmo de optimizacion. En este ejemplo, se obtiene uno por uno los items del arbol de decision y se obtiene el beneficio de los hijos de cada item tambien, se actualiza el beneficio maximo (maxProfit) en cada paso y se encuentra el maximo beneficio posible.

In [3]:
```
from scipy.optimize import minimize
import numpy as np

def branch_and_bound(fun,cons,x0,bnds):
    nr_variables = len(x0) # numero de variables de decision

    # calcula una solucion inicial con valores enteros o no enteros
    res = minimize(fun, x0, method='SLSQP', bounds=bnds,
                   constraints=cons)

    # inicializa una fila para guardar los nodos
    queue = []
    queue.append((res,bnds)) # adiciona el nodo inicial

    mejor_sol = res # guarda la mejor solucion

    # Hace iteracion hasta tener todas las variables de decision como e
```

```
nteros
    while len(queue) > 0:

        res,bnds = queue.pop(0)

        # de la solucion anterior, busca las variables de decision con
 mayores valores y guarda los indices
        # que ordenen estas variables
        idx_sort = np.argsort(res.x)[::-1]
        ubnds = bnds.copy() # inicializa upper bound
        lbnds = bnds.copy() # inicializa lower bound
        # itera en las variables de decision para fijar en 0 o 1 el val
or mayor que no este ya fijado
        for idx in idx_sort:
            if bnds[idx] == (0,None):
                ubnds[idx] = (1,1)
                lbnds[idx] = (0,0)
                break

        if (0,None) not in bnds: # cuando todos los limites (bounds) es
tan fijados, retorna la respuesta
            return res

        # busca la solucion optima con los limites superiores e inferio
res, generando dos nodos hijos del nodo anterior.
        ubres = minimize(fun, x0, method='SLSQP', bounds=ubnds, constra
ints=cons)
        lbres = minimize(fun, x0, method='SLSQP', bounds=lbnds, constra
ints=cons)


        # Caso 1 comprueba si la mejor solucion es mayor que la solucio
n actual, en caso contrario cambia el valor guardado
        # de mejor solucion
        if abs(mejor_sol.fun) > abs(ubres.fun):
            pass
        else:
            if ubres.success == True:
                mejor_sol = ubres
```

```
        if abs(mejor_sol.fun) > abs(ubres.fun):
            pass
        else:
            if lbres.success == True:
                mejor_sol = lbres

        # comprueba si el nodo es factible (success = True), y en caso
 positivo se guarda en la fila el nodo con mayor valor
        if ubres.success == True:
            if lbres.success == True:
                if abs(ubres.fun) > abs(lbres.fun):
                    queue.append((ubres,ubnds))
                else:
                    queue.append((lbres,lbnds))
            else:
                queue.append((ubres,ubnds))
        else:
            if lbres.success == True:
                queue.append((lbres,lbnds))
```

In [4]:
```
fun = lambda x: -(15*x[0] + 12*x[1] +4*x[2] + 2*x[3])
cons = ({'type': 'ineq', 'fun': lambda x:  -(8*x[0] + 5*x[1] + 3*x[2] +
 2*x[3]-10)})
bnds = [(0, None), (0, None),(0,None),(0,None)]
x0 = (0,0,0,0)
branch_and_bound(fun,cons,x0,bnds)
```

Out[4]:
```
     fun: -18.0
     jac: array([-15., -12.,  -4.,  -2.])
 message: 'Optimization terminated successfully.'
    nfev: 6
     nit: 1
    njev: 1
  status: 0
 success: True
       x: array([0., 1., 1., 1.])
```

In [5]:
```
# Problema de la mochila 0/1
# los items tienen valores 40, 50, 100, 95, 30 y se quiere maximizar el
 valor total
# la capacidad de la mochila es 10
# El peso de los items es 2, 3.14, 1.98, 5 y 3
# solo puedo poner 1 item de cada tipo (o no ponerlo)
fun = lambda x: -(40*x[0] + 50*x[1] +100*x[2] + 95*x[3] + 30*x[4])
cons = ({'type': 'ineq', 'fun': lambda x:  -(2*x[0] + 3.14*x[1] + 1.98*
x[2] + 5*x[3] + 3*x[4] -10)})
bnds = [(0, None), (0, None),(0,None),(0,None),(0,None)]
x0 = (0,0,0,0,0)
branch_and_bound(fun,cons,x0,bnds)
```

Out[5]:
```
     fun: -235.0
     jac: array([ -40.,  -50., -100.,  -95.,  -30.])
 message: 'Optimization terminated successfully.'
    nfev: 7
     nit: 1
    njev: 1
  status: 0
 success: True
       x: array([1., 0., 1., 1., 0.])
```

• Modelar problemas y elección del algoritmo adecuado.

En el siguiente ejercicio se trata de buscar la raiz de una funcion no lineal.La construccion de modelos revela problemas que no se ven a primera vista en otros modelos,cuando se construye un modelo matematico es posible obtener propiedades y caracteristicas de las relaciones que de otra forma estarian ocultas, tambien se reflejan situaciones complejas que no se pueden hacer en otro tipo de modelos y de ahi la importancia de modelizar y escoger el algoritmo adecuado.

In [6]:
```
# El problema de buscar la raiz(es) de una funcion no lineal
f = lambda x: (1-x[0])**2 + 100*(x[1] - x[0]**2)**2

xn_1 = np.array([2,2]) # x(n-1)
xn = np.array([0,0]) # x(n)
fn_1 = f(xn_1)
fn = f(xn)
```

```
iter_ = 0

print(f'iter \t x \t \t \t f')

while fn > 1e-6:

    xnew = xn - (xn-xn_1)/(fn-fn_1)*fn
    fn_1 = fn
    xn_1 = xn
    xn = xnew
    fn = f(xnew)
    iter_ += 1
    print(f'{iter_} \t {xn[0]:.3f},{xn[1]:.3f} \t {fn:.3e}')
```

```
iter     x                       f
1        -0.005,-0.005   1.013e+00
2        0.398,0.398     6.106e+00
3        -0.085,-0.085   2.032e+00
4        -0.326,-0.326   2.050e+01
5        -0.059,-0.059   1.506e+00
6        -0.037,-0.037   1.227e+00
7        0.056,0.056     1.169e+00
8        1.953,1.953     3.469e+02
9        0.049,0.049     1.124e+00
10       0.043,0.043     1.087e+00
11       -0.135,-0.135   3.620e+00
12       0.120,0.120     1.882e+00
13       0.395,0.395     6.075e+00
14       -0.004,-0.004   1.010e+00
15       -0.084,-0.084   1.995e+00
16       0.077,0.077     1.361e+00
17       0.423,0.423     6.290e+00
18       -0.018,-0.018   1.070e+00
19       -0.108,-0.108   2.674e+00
20       0.042,0.042     1.081e+00
21       0.145,0.145     2.262e+00
22       -0.051,-0.051   1.398e+00
23       -0.368,-0.368   2.725e+01
24       -0.034,-0.034   1.195e+00
25       -0.019,-0.019   1.076e+00
```

```
26      0.119,0.119     1.869e+00
27      -0.205,-0.205   7.578e+00
28      0.225,0.225     3.635e+00
29      0.621,0.621     5.683e+00
30      -0.479,-0.479   5.226e+01
31      0.755,0.755     3.479e+00
32      0.843,0.843     1.774e+00
33      0.935,0.935     3.770e-01
34      0.959,0.959     1.535e-01
35      0.976,0.976     5.389e-02
36      0.986,0.986     2.056e-02
37      0.991,0.991     7.708e-03
38      0.995,0.995     2.932e-03
39      0.997,0.997     1.114e-03
40      0.998,0.998     4.246e-04
41      0.999,0.999     1.619e-04
42      0.999,0.999     6.179e-05
43      1.000,1.000     2.359e-05
44      1.000,1.000     9.006e-06
45      1.000,1.000     3.439e-06
46      1.000,1.000     1.313e-06
47      1.000,1.000     5.017e-07
```

In [ ]: