Proyecto 1 Inteligencia Artificial

Saldaña Hernandez Norman Brandon Guerrero Ferrusca Francisco

González Lorenzo Daniela Michel Tenorio Veloz Alisson Dafne

September 2021

1 Planteamiento del Problema

Diseñar un agente Inteligente que encuentre una asignación de exámenes dentro de una red social de Alumnos, tal que dos alumnos que sean amigos no tengan un mismo examen y la asignación debe ser una de menor número de exámenes.

2 Solución basada en búsqueda

El problema requiere como salida una asignación de exámenes, es decir una regla de correspondencia definida, donde se relaciona a los alumnos de la red social con sus tipos de exámenes correspondientes.
Como entrada al agente inteligente a diseñar se debe dar un problema, es decir, los alumnos de la red social y sus respectivas relaciones de amistad, ya que esto nos permite evaluar que asignaciones son válidas y cuales no
Entonces según lo visto en clase y tomando en cuenta lo dicho anteriormente, planteamos una solución con las siguientes características:

- Estados (Nodos): Asignaciones de examen a cada alumno (cumplan o no las restricciones), en este caso usaremos un diccionario:

```
    {'Juan':1,'Jorge':2,'Diana':3}# donde el numero representa el n mero de examen
```

- Estado Inicial: Asignación vacía

```
    {'Juan':0,'Jorge':0,'Diana':0}
```

- Espacio de estados: El espacio de estados equivale al árbol de estados que debemos recorrer para encontras las soluciones, esta es un grafo del tipo arbol, donde los nodos son los estados previamente definidos.
- Función sucesor: Asignar un examen al alumno siguiente.
- Prueba meta: Asignar todos los examenes y probar que la asignación cumpla las restricciones.
- Costo: Una unidad por examen distinto.
- Problema de entrada: El problema será definido con un diccionario, donde la llave es el nombre del alumno y a cada alumno se le asocia una lista con los nombres de sus amigos.

```
Problema ={ # es la definici n del problema, a cada clave (alumno) se le relaciona una lista de
                                                    amigos
  'Juan':['Jorge','Diana','Laura'],
  'Jorge':['Juan'],
  'Diana':['Jorge','Laura'],
  'Laura':['Diana','Juan'],
  'Carlos':[]
}
```

2.1 Árbol de estados

El árbol de estados tiene como nodo raiz el nodo con los alumnos asignados todos con el examen 0, el examen 0 es un examen nulo, es decir no hay tipo de examen 0, es un indicador de que no ha sido asignado examen. Conforme el arbol aumente en profundidad significa que se asignan examenes, la profundidad es igual al número de alumnos, si hay 3 alumnos el arbol es de profundidad 3.
Otra observación interesante del problema es que en el peor de los casos, es decir donde haya un mayor número de exámenes, sucede cuando el problema indica que todos los alumnos son amigos entre sí, es decir que no se puede asignar un mismo examen a dos alumnos, dado que todos son amigos, por lo tanto se tendrá que asignar un examen diferente a cada alumno. Este caso y el hecho de que no sabemos en que nodo hoja hay una solucion nos dice que al no saber que tipo de problema nos darán como entrada, debemos considerar N exámenes donde N es el número de alumnos, por lo tanto cada nodo del árbol tendrá N ramificaciones y como se dijo anteriormente la profundidad del árbol de estados también es N. Con esto podemos concluir que el árbol de estados tiene N^N nodos hoja.
Otra consideración es que solo los nodos hoja (nodos finales) son aquellos que nos interesan, dado que son los que contienen una asignación completa o en otras palabras, una asignación donde a ningún alumno le haga falta examen, el resto de nodos son asignaciones incompletas.

2.2 Búsqueda en profundidad

La búsqueda en profundidad se adapta perfectamente a nuestro problema, dado que recorre el árbol de estados llegando a los nodos hoja, en este punto podemos evaluar si los nodos hoja cumplen con nuestras restricciones, y de ser el caso podemos tener un candidato a asignación óptima, ya que recordando el planteamiento del problema buscamos también una asignación con el menor número de exámenes para ahorrar trabajo al profesor.
Por lo tanto, además de la búsqueda en profundidad, hay que evaluar los nodos hojas, y de ser el caso almacenarlos si su coste es mas bajo que el de algún nodo con el record de ser el de coste mas bajo, así nos aseguramos de que al final de la ejecución del programa tengamos la asignación de examenes con menor número de exámenes y que además cumpla las restricciones de amistad.
Una ventaja de búsqueda en profundidad es que el espacio en memoria es mucho menor al que podríamos llegar a ocupar si usamos búsqueda a lo ancho, esto porque en búsqueda por anchura necesitaríamos almacenar todos los estados a la vez, y como sabemos que vamos a recorrer todo el árbol, es mas conveniente hacerlo por profundidad.

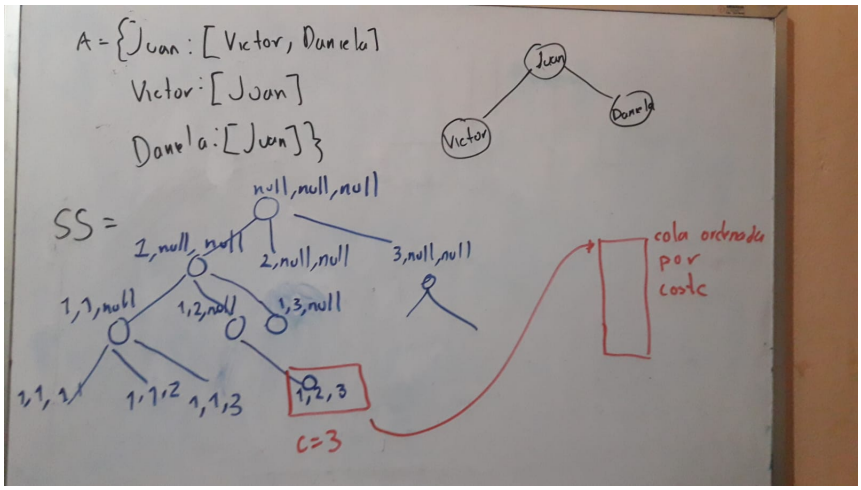

Figure 1: Espacio de estados

2.3 Desarrollo del código

Lo primero que hicimos para resolver el problema es hacerlo más pequeño, que cumpla las características, pero limitandonos a un problema mucho más simple, Como el siguiente:

```
Problema = {
  'Juan':['Jorge','Diana'],
  'Jorge':['Juan'],
  'Diana':['Juan'],
}
```

El árbol de estados es como el de la Figura 2, en donde se puede notar que parte de una asignación nula, y conforme bajamos en profundidad se asignan examenes a mas alumnos.

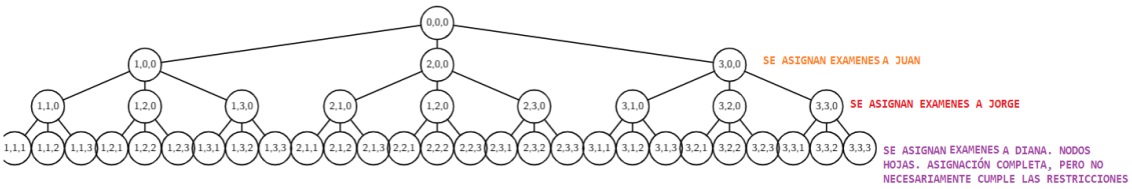

Figure 2: Arbol de estados para 3 alumnos

Lo primero es definir las variables, se define una lista con exámenes, con 3 tipos de exámenes, para asegurarnos que se asignen todos los exámenes posibles a los alumnos y tener el arbol de estados completo, en este caso como máximo debemos tener 3 tipos de examen dado que son 3 alumnos.
Depues debemos hacer una iteración en cada alumno,para obtener las combinaciones posibles.
El pseudocódigo sería el siguiente:

```
#Obtencion de todas las asignaciones completas para el problema de 3 alumnos
for examen_alumno1 en lista_examenes
    Asignar examen_alumno1  a alumno1
    for examen_alumno2 en lista_examenes
        Asignar examen_alumno2  a alumno2
        for examen_alumno3 en lista_examenes
            Asignar examen_alumno3  a alumno3
```

Para empezar el código, se define el problema con un diccionario y se inicializa el diccionario que contiene las asignaciones, este es el formato de los nodos.

```
examenes=[1,2,3]#lista de ex menes
Asignacion = {}#inicializaci n de diccionario de asignaci n vacio
Problema = {
  'Juan':['Jorge','Diana'],
  'Jorge':['Juan'],
  'Diana':['Juan'],
}
#Problema.keys obtiene todos los alumnos del problema
for node in Problema.keys(): #inicializamos el diccionario de asignaciones con los alumnos, asignando 0
                                                    a todos (no se ha asignado examen)
  Asignacion[node] = 0 # Asigna 0 a cada alumno, donde node el es alumno, rellena el diccionario

print(Asignacion)#imprime ['Juan':0,'Jorge':0,'Diana':0], el diccionario tiene el formato adecuado en
                                                    este punto
```

El código de a continuación genera todos los nodos completamente asignados, es decir, los nodos hoja del problema de 3 alumnos.
Esto se puede hacer como se ve en el código con una anidación de ciclos for, donde se asignen los exámenes por niveles, y en el nivel mas bajo se imprimen los resultados, es decir las asignaciones completas o nodos hoja

```
##Este programa obtiene todos los nodos hoja del  rbol  de estados, es decir, todas las asignaciones
                                                    completas
## Lo hace para 3 alumnos dado que solo hay 3 ciclos for anidados
examenes=[1,2,3]#lista de ex menes
Asignacion = {}#inicializaci n de diccionario de asignaci n vacio
Problema = {
  'Juan':['Jorge','Diana'],
  'Jorge':['Juan'],
  'Diana':['Juan'],
}
#Problema.keys obtiene todos los alumnos del problema
for node in Problema.keys(): #inicializamos el diccionario de asignaciones con los alumnos, asignando 0
                                                    a todos (no se ha asignado examen)
  Asignacion[node] = 0 # Asigna 0 a cada alumno, donde node el es alumno, rellena el diccionario

print(Asignacion)#imprime ['Juan':0,'Jorge':0,'Diana':0], el diccionario tiene el formato adecuado en
                                                    este punto

for i in examenes: #iteramos 3 veces; i itera en la lista [1 2 3]
  Asignacion['Juan'] = i #asignamos el valor de i a Juan en el diccionario de asignacion
  for j in examenes: #iteramos 3 veces; j itera en la lista [1 2 3]
    Asignacion['Jorge'] = j #asignamos el valor de j a Juan en el diccionario de asignacion
    for k in examenes: # iteramos 3 veces; k itera en la lista [1 2 3]
      Asignacion['Diana'] = k # asignamos el valor de k a Juan en el diccionario de asignacion
      print(Asignacion) # imprime la asignaci n Final
```

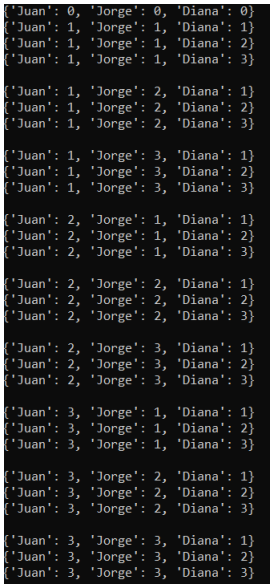

Figure 3: Output del código anterior: todas las asignaciones posibles

Si agregamos una función que valide si una asignación cumple con las restricciones, quedaría del siguiente modo:

```
def IsValid(Asig):  #testea si una asignaci n cumple con las restricciones
  for alumno in Alumnos:  # itera sobre todos los alumnos [Juan, Jorge,Diana]
    for friend in Problema.get(alumno): # obtiene los amigos del alumno en ese momento
      if Asig.get(friend) == Asig.get(alumno): #compara el examen asignado de un amigo con el alumno
        return 0        #si 2 amigos tienen el mismo examen se habr n violado las restricciones, por
                                              lo tanto la funcion devuelve 0
  return 1 # si ning n par de amigos han tenido el mismo examen, devuelve 1
```

Al agregar esta función al código que obtiene todas las asignaciones posibles, y en el ultimo for anidado ponemos la condicion de que si la asignacion es valida la imprima, entonces solo se imprimirán asignaciones válidas como en la Figura 4

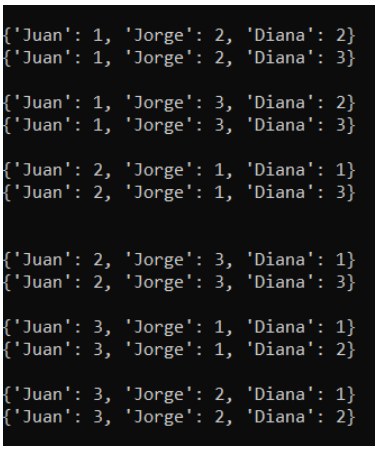

Figure 4: Caption

Y como podemos observar se filtran las asignaciones a las que solo son válidas, obteniendo un menor número de impresiones

Ahora, hay que generalizar este último algoritmo para N número de alumnos, lo que se hará aquí será una función recursiva que llame a un ciclo for y lo haga por profundidad. Así la funcion recursiva dejará de llamarse cuando alcance la máxima profundidad y termine los ciclos for. El pseudocódigo es el siguiente

```
Funcion_Recursiva(profundidad)
    if profundidad < numero_de_alumnos:
        for examen en lista_de_examenes
            Asignacion(alumno en profundidad actual) <- examen
            Funcion_Recursiva(profundidad + 1)
    else
        Imprimir asignacion
```

Implementando a código el pseudocódigo tenemos lo siguiente, esto obtiene los nodos hojas y recorre todo el arbol al igual que lo algoritmos anteriores pero con la ventaja de que puede ser ingresado cualquier problema.

```
##Este programa obtiene todos los nodos hoja del  rbol  de estados, es decir, todas las asignaciones
                                                        completas
## Lo hace para N alumnos gracias a la funcion recursiva
def GetAsig(prof):# Funcion recursiva busqueda en profundidad
  if prof < len(Alumnos):# si la profundidad del arbol de busqueda aun no es la maxima (no hemos
                                                        llegado a los nodos hoojas)
    for i in examenes: # iterando en los valores de los posibles examenes [1 2 3]
      Asignacion[Alumnos[prof]] = i #se le asigna el valor de el examen i al alumno de profundidad prof
                                                        , p.e Juan tiene profundidad 0, Jorge profundidad 1
                                                        , Diana profundidad 2 en el arbol de estados
      GetAsig(prof + 1)  # Se llama a esta misma funcion para bajar en profundidad, sumando profundidad
                                                         mas 1, asi se baja en el arbol de estados
  else:          # si la profundidad es maxima tenemos una asignacion completa (se ha asignado un examen
                                                        a cada alumno)
    print(Asignacion) # Imprimimos la asignacion completa


examenes=[1,2,3]
Problema ={
  'Juan':['Jorge','Diana'],
  'Jorge':['Juan','Laura'],
  'Diana':['Jorge'],
  'Laura':['Jorge'],
  'Carlos':[]
}
Asignacion = {}
Alumnos = list(Problema.keys())#Hacemos una lista con los alumnos [Juan, Jorge, Diana, Laura, Carlos]
for alumno in Problema.keys(): #inicializamos el diccionario de asignaciones con los alumnos, asignando
                                                        0 a todos (no se ha asignado examen)
  Asignacion[alumno] = 0

GetAsig(0)# llamamos a la funcion recursiva iniciando con una profundidad 0
```

La salida para un problema de 5 alumnos sería como la que se muestra a continuación, tomando en cuenta que no se alcanzó a capturar todas las asignaciones que se muestran en consola dado que son demasiadas

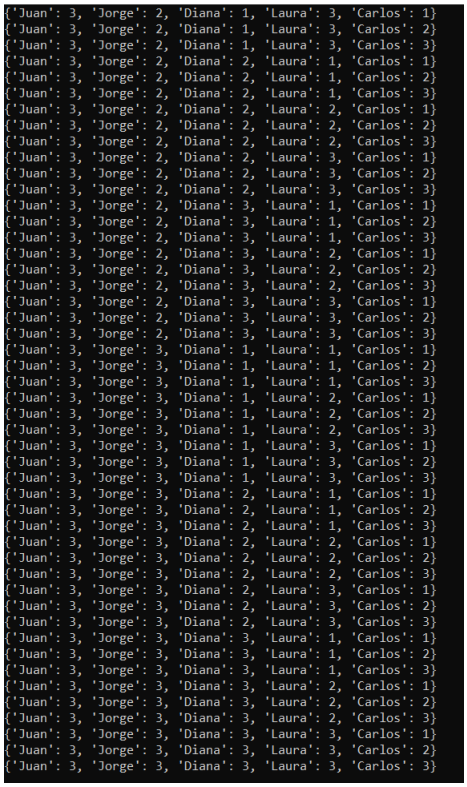

Figure 5: Caption

Ahora que podemos recorrer el árbol de búsqueda, y que además podemos filtrar las asignaciones válidas, es necesario diseñar una función de coste, que nos diga el coste de una asignación para asi poder evaluar y almacenar la que nos dé el mejor coste y devolver esa función al finalizar la ejecución.
En el siguiente código se muestra una función de coste adecuada

```
def cost(Asig):
  exdif =[] #lista vacia llamada examenes distintos abreviado
  for alumno in Alumnos:
    if Asig.get(alumno) not in exdif: #si el examen asignado al alumno N no esta en la lista, agregarlo
      exdif.append(Asig.get(alumno))

  costo = len(exdif)#exdif contendra solo examenes distintos, no repetidos, por lo tanto su longitud es
                                                    el numero de examenes asignados y eso corresponde
                                                   al coste
  return costo
```

Para finalizar se unen el código que recorre el espacio de estados con una función recursiva, la función de coste y la función de evaluación. Y solo se requiere agregar una sección de código adicional en el "else" de la función recursiva, donde además de lo que se hacia antes de validar la asiganción tambien se deberá obtener el coste de la función válidada y comparar con un coste de referencia, este se irá actualizando si se encuentra una asignación con coste menor, por lo tanto al finalizar el recorrido del arbol de estados tendrémos la solución de menor coste que tambien hemos almacenado. Se puede ver que tambien se agregaron unas funciones del tipo getters y setters, para obtener ese coste de referencia y guardarlo si se actualiza, lo mismo para la asigancion con ese respectivo coste, aunque se debe decir que hubo problemas a la hora de manejar estas variables, porque al final la asignacion minima no se guardaba correctamente, así que por eso imprimimos dentro de la función recursiva las asignaciones que vayan teniendo menor coste, así al final obtendremos la de menor coste impresa en pantalla.

Código Final

```
def cost(Asig):
  exdif =[]
  for alumno in Alumnos:
    if Asig.get(alumno) not in numexa:
      exdif.append(Asig.get(alumno))

  costo = len(exdif)
  return costo

def IsValid(Asig):
  for alumno in Alumnos:
    for friend in Problema.get(alumno):
      if Asig.get(friend) == Asig.get(alumno):
        return 0
  return 1

def getCostMin():
  global costmin
  return costmin
def setCostmin(value):
  global costmin
  costmin = value
def getBestAsig():
  global mejorAsig
  return mejorAsig
def setBestAsig(value):
  global mejorAsig
  mejorAsig = value

def GetAsig(prof):
  if prof < len(Alumnos):
    for i in examenes:
      Asignacion[Alumnos[prof]] = i
      GetAsig(prof +1)
  else:
    if IsValid(Asignacion):
      if  cost(Asignacion) < getCostMin():
        setCostmin(cost(Asignacion))
        setBestAsig(Asignacion)
        print("Una asignacion optima es: "+str(getBestAsig()))
        print("Su costo : " + str(cost(getBestAsig())))

import time
inicio = time.time()
#Declaracion de variables y del problema
mejorAsig ={}
examenes=[] # [1,2,3,4,5,.., N]
Asignacion = {}
Problema ={
  'Juan':['Jorge','Diana','Carlos'],
  'Jorge':['Juan','Laura'],
  'Diana':['Juan','Laura','Josue'],
  'Laura':['Jorge','Diana'],
  'Carlos':['Juan'],
  'Josue':['Diana','America'],
  'America':['Josue'],
  'Oscar':[]

}

Alumnos = list(Problema.keys())
i=0
for alumno in Problema.keys():
  Asignacion[alumno] = 0
  i = i +1
  examenes.append(i)

costmin =len(examenes)

GetAsig(0)
fin = time.time()
print("tiempo de ejecucion: "+str(fin-inicio))
```

3 Experimentos

3.1 Experimento de baja dificultad

3.1.1 Experimento de 2 alumnos

```
Problema ={
'Luis':[],
  'Maria':[]
}
```

```
C:\Users\Gferr\Desktop\Inteligencia>py IA4.py
Una asignación optima es: {'Luis': 1, 'Maria': 1}
Su costo : 1
tiempo de ejecución: 0.0009980201721191406
```

Figure 6: Arbol de estados para 2 alumnos

3.1.2 Experimento de 3 alumnos

```
Problema ={
  'Juan':['Jorge','Diana'],
  'Jorge':['Juan'],
  'Diana':['Juan'],
}
```

```
C:\Users\Gferr\Desktop\Inteligencia>py IA4.py
Una asignación optima es: {'Juan': 1, 'Jorge': 2, 'Diana': 2}
Su costo : 2
tiempo de ejecución: 0.0007903575897216797
```

Figure 7: Arbol de estados para 3 alumnos

3.1.3 Experimento de 6 alumnos

```
Problema ={
  'Juan':['Jorge','Diana','Carlos'],
  'Jorge':['Juan','Laura'],
  'Diana':['Juan','Laura','Jose'],
  'Laura':['Jorge','Diana'],
  'Carlos':['Juan'],
  'Josue':['Diana']

}
```

```
C:\Users\Gferr\Desktop\Inteligencia>py IA4.py
Una asignación optima es: {'Juan': 1, 'Jorge': 2, 'Diana': 2, 'Laura': 1, 'Carlos': 2, 'Josue': 1}
Su costo : 2
tiempo de ejecución: 0.07882475852966309
```

Figure 8: Arbol de estados para 6 alumnos

3.2 Experimentos de media dificultad

3.2.1 Experimento de 10 alumnos

```
Problema ={
  'Laura':['Jorge','Juan','Nancy'],
  'Jose':['Diana','Fernanda','Carlos','Diana','Axel'],
  'Liliana':['Axel','Fernanda'],
  'Juan':['Carlos','Laura','Jorge'],
  'Jorge':['Laura','Juan','Axel'],
  'Diana':['Jose','Nancy','Jose'],
  'Carlos':['Juan','Jose','Fernanda'],
  'Nancy':['Diana','Laura'],
  'Axel':['Liliana','Jorge','Jose'],
  'Fernanda':['Jose','Liliana','Carlos']
}
```

```
C:\Users\Gferr\Desktop\Inteligencia>py IA4.py
Una asignación optima es: {'Laura': 1, 'Jose': 1, 'Liliana': 1, 'Juan': 2, 'Jorge': 3, 'Diana': 2, 'Carlos': 3, 'Nancy':
 3, 'Axel': 2, 'Fernanda': 2}
Su costo : 3
```

Figure 9: Arbol de estados para 10 alumnos

3.2.2 Experimento de 7 alumnos

```
Problema ={
  'Juan':['Jorge','Diana','Carlos'],
  'Jorge':['Juan','Laura'],
  'Diana':['Juan','Laura','Josue'],
  'Laura':['Jorge','Diana'],
  'Carlos':['Juan'],
  'Josue':['Diana','America'],
  'America':['Josue'],

}
```

```
C:\Users\Gferr\Desktop\Inteligencia>py IA4.py
Una asignación optima es: {'Juan': 1, 'Jorge': 2, 'Diana': 2, 'Laura': 1, 'Carlos': 2, 'Josue': 1, 'America': 2}
Su costo : 2
tiempo de ejecución: 1.5816540718078613
```

Figure 10: Arbol de estados para 7 alumnos

3.2.3 Experimento de 8 alumnos

```
Problema ={
  'Juan':['Jorge','Diana','Carlos'],
  'Jorge':['Juan','Laura'],
  'Diana':['Juan','Laura','Josue'],
  'Laura':['Jorge','Diana'],
  'Carlos':['Juan'],
  'Josue':['Diana','America'],
  'America':['Josue'],
  'Fernanda':['Jorge','America']

}
```

```
C:\Users\Gferr\Desktop\Inteligencia>py IA4.py
Una asignación optima es: {'Juan': 1, 'Jorge': 2, 'Diana': 2, 'Laura': 1, 'Carlos': 2, 'Josue': 1, 'America': 2, 'Fernanda': 1}
Su costo : 2
```

Figure 11: Arbol de estados para 8 alumnos

3.3 Experimentos de alta dificultad

3.3.1 15 alumnos

```
Problema ={
  'Laura':['Jorge','Ana'],
  'Jose':['Diana','Gabriela'],
  'Liliana':['Axel','David','Gabriela'],
  'Juan':['Carlos','David','Gabriela'],
  'Jorge':['Laura','Diana'],
  'Diana':['Jose','Jorge','Nancy'],
  'Carlos':['Juan','Gabriela'],
  'Nancy':['Arturo','Diana','Angel'],
  'Axel':['Liliana','Gabriela'],
  'Fernanda':['Diego','Gabriela'],
  'Oscar':['David','Maria','Gabriela'],
  'Maria':['Gabriela','Oscar','Christian'],
  'Angel':['Christian','Nancy','Arturo'],
  'Arturo':['Nancy','Angel','Diego'],
  'Ana':['Karen','Laura','Gabriela'],
  'Diego':['Fernanda','Arturo','Karen'],
  'David':['Oscar','Liliana','Juan'],
  'Gabriela':['Maria','Jose','Carlos','Axel','Fernanda','Ana','Oscar','Liliana','Juan'],
  'Christian':['Angel','Maria'],
  'Karen':['Ana','Diego']
}
```

Por la complejidad exponencial son demasiados nodos para que de un resultado en un tiempo razonable, si calculamos la complejidad en tiempo podemos dedudir que como N^N es el numero de nodos hoja generados, y tenemos que $N = 15$ entonces $15^{15} = 2.37e17 nodos$, esto implica que si tomamos como referencia alguno de los experimentos anteriores, donde tenemos un tiempo de ejecución y tambien sabemos cuantos nodos se generan, aproximadamente podríamos calcular que cada nodo se tarda aproximadamente 3 microsegundos, aunque es un numero que puede variar dado que se usa la funcion de validacion en unicamente los nodos hoja por lo que no es un numero exacto de calcular, pero para hacer una referencia podriamos asumir que si cada nodo se ejecuta en 3 microsegundos, entonces el tiempo de ejecucion de este programa debería ser aproximadamente $3[\frac{\mu s}{nodo}] * 2.37E17[nodos] = 1.31E12[segundos] = 2.19E10[minutos] = 364,911,575[horas] = 15,204,640[dias] = 41656[años]$
Esto es solo una aproximación puesto que realmente se hacen mas visitas a nodos, porque esto solo es para nodos hoja, entonces probablemente sea mas tiempo de ejecución.

3.4 Experimento sin solución

```
Problema={}
```

No hay solución porque no hay asignaciones posibles, el algoritmo garantiza que encuentra una solución si la hay, siempre y cuando haya alumnos, la manera en la que no encontrara solución es que no haya alumnos.
La salida del programa sería la siguiente, dado que no hay asignaciones, no existe solución

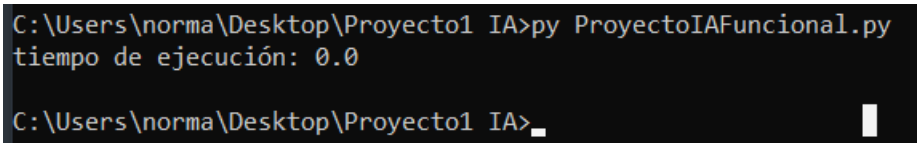

Figure 12: Caption

4 Conclusiones

El agente desarrollado hace su trabajo, el algoritmo garantiza que se encontrará una solución, sin embargo la complejidad en tiempo no es asumible, si se tienen grupos de mas de 9 alumnos el tiempo de procesamiento comenzará a ser excesivo. Como recomendación que se nos hizo fue una poda del árbol, que por cuestiones de tiempo no se pudo implementar pero el concepto se entiende, por lo que pensamos que aún con una poda del árbol no sería suficiente para mejorar el algoritmo a un punto tal que encuentre la solución para 15 alumnos en menos de 10 segundos, puesto que aun si podara el arbol lo suficiente creemos que siguen siendo excesivos el numero de nodos para 15 alumnos y no hablar de para 40 alumnos que sería el tamaño de un grupo en la vida real.
Estamos seguros que para este tipo de problemas que buscan satisfacciones de restricciones, no solo alguna satisfaccion de ellas, sino la más óptimas, es necesario recorrer todo el arbol de estados para asegurarse que no se ha dejado alguna solucion que podría ser mejor, sin embargo usar DFS o BFS para recorres espacios de estado completos es inasumible. Se deben buscar mejores soluciones y mas eficientes que no implique complejidades exponenciales ni en memoria ni en tiempo.

5 Referencias

Sloman, Leila (2021). Mathematician Answers Chess Problem About Attacking Queens, Quanta Magazine.