ESCUELA NACIONAL DE ESTUDIOS SUPERIORES
UNIDAD MORELIA

Universidad Nacional Autónoma de México

Escuela Nacional de Estudios Superiores
Unidad Morelia

Tecnologías para la Información en Ciencias

Inteligencia Artificial

Búsquedas con utilidad

Situación problema

Este trabajo se realizó con el apoyo del Programa UNAM-DGAPA-PAPIME dentro del Proyecto para la innovación y mejoramiento para la enseñanza, con clave PE110324, Diseño didáctico multi-seriado para potenciar habilidades de modelación matemática y reflexión crítica en el aprendizaje de la Inteligencia Artificial.

Presentación

El presente diseño didáctico lleva por título “El juego del 31” y es una situación-problema que admite representaciones algebraicas, computacionales, combinatorias y probabilísticas, para modelar un problema de juego discreto con información completa. Dentro de la asignatura de Inteligencia Artificial, corresponde al contenido temático de *representación y búsqueda de soluciones*; en él, permite representar un problema de forma reducida para buscar su solución de forma dinámica, a partir de la interacción con una persona.

Su elaboración incluye un documento y un recurso interactivo. El documento tiene, a su vez, dos secciones. La primera es un manual para estudiantes que incorpora un documento de presentación de la situación, una hoja de trabajo y un manual donde el cuaderno interactivo se ha editado para impresión; la segunda es un manual para docentes con una guía para el uso de material concreto, orientaciones didácticas para su implementación y una edición del cuaderno interactivo que permite anticipar las respuestas que dará el alumnado.

El material se desarrolló en un proceso metodológico iterativo de investigación basada en diseño, que permitió observar su incidencia positiva en el aprendizaje de habilidades de representación, motivando que el estudiantado (1) genere modelos a partir de una situación concreta, considerando distintas representaciones, que (2) sean apropiados para que un meta-modelo pueda encontrar la solución a partir del modelo propuesto; en este caso, el meta-modelo es el método de decisión Minimax.

ESCUELA NACIONAL DE ESTUDIOS SUPERIORES
UNIDAD MORELIA

Universidad Nacional Autónoma de México

Escuela Nacional de Estudios Superiores
Unidad Morelia

Tecnologías para la Información en Ciencias

Inteligencia Artificial

Búsquedas con utilidad

Manual para estudiantes

Este trabajo se realizó con el apoyo del Programa UNAM-DGAPA-PAPIME dentro del Proyecto para la innovación y mejoramiento para la enseñanza, con clave PE110324, Diseño didáctico multi-seriado para potenciar habilidades de modelación matemática y reflexión crítica en el aprendizaje de la Inteligencia Artificial.

El juego del 31

En el juego del 31 participan dos personas, en turnos alternados. Para jugar se colocan fichas sobre el tablero que se muestra a continuación:

1	2	3	4	5	6
1	2	3	4	5	6
1	2	3	4	6	6
1	2	3	4	6	6

Figura 1: Tablero para jugar el juego del 31.

En su turno, cada persona coloca una ficha en alguna de las casillas del juego que no esté ocupada y dice en voz alta la suma acumulada de todos los números bajo las fichas del tablero. La primera persona que menciona una suma mayor a 31, pierde el juego.

A continuación se muestran los primeros tres turnos en una partida:

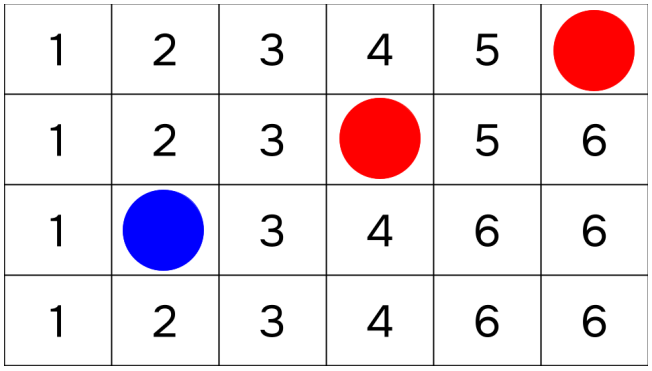

Figura 2: Primeros tres turnos en una partida de el juego del 31.

Primer turno: La primera persona coloca ficha en 4. Entonces grita: ¡cuatro!

Segundo turno: La segunda persona coloca ficha en 2. Entonces grita: ¡seis!

Tercer turno: La primera persona coloca ficha en 6. Entonces grita: ¡doce!

1. Propón una estrategia ganadora y regístrala en la hoja de trabajo adjunta.
2. Desarrolla un algoritmo de búsqueda para encontrar la estrategia ganadora al jugar con una persona.

Hoja de trabajo

Nombre:

¿Habrá una forma de ganar siempre? ¿Qué estrategia propondrías para ello?

Registra un juego concreto donde tu estrategia funciona.

¿Puedes convencerte de que tu estrategia funciona en muchos casos? Utiliza una representación conveniente para comunicar tus ideas con mayor claridad.

¿Habrá algo que pueda suceder que evite que la estrategia funcione? Una buena sugerencia para revisar es jugar con otras personas.

¿Cómo podemos aprovechar la computadora en nuestra búsqueda de esta estrategia?

Cuaderno de Jupyter

Representación

Tablero de juego

```
import copy
```

A continuación se define la clase ***Gameboard*** que representará un estado del tablero de juego. En ***tokens*** se almacena un diccionario con la cantidad de casillas por cada valor que han sido utlizadas y ***value*** nos indica cual es la suma de estas casillas.

```
class Gameboard:

    def __init__(self,tokens,value):
        self.tokens = tokens
        self.value  = value


    def __eq__(self,other):
        return (self.tokens == other.tokens)


    def __copy__(self):
        return Gameboard(self.tokens, self.value)


    def __deepcopy__(self,memo):
        tokens = copy.deepcopy(self.tokens,memo)
        value  = copy.deepcopy(self.value ,memo)
        return Gameboard(tokens,value)


    def print(self):
        print( "En el tablero analizado se ha usado:" )
        for value,times_used in self.tokens.items():
            print( "\t {} veces la casilla {}".format(times_used,value) )
        print( "La suma del tablero es: {}".format(self.value) )
```

En la siguiente celda definiremos las dimensiones del tablero, la cantidad de filas y columnas que tendrá. También definiremos el máximo número que será válido antes de que no existan movimientos posibles.

```
total_rows        = 4
max_token_value   = 6
max_allowed_value = 31
```

Para ejemplificar el funcionamiento de la clase ***Gameboard*** utilizaremos el tablero inicial y mostraremos en pantalla su contenido.

```
initial_gameboard = Gameboard( { i:0 for i in range(1,max_token_value+1) }, 0 )
```

```
initial_gameboard.print()
```

```
>>> En el tablero analizado se ha usado:
>>>     0 veces la casilla 1
>>>     0 veces la casilla 2
>>>     0 veces la casilla 3
>>>     0 veces la casilla 4
>>>     0 veces la casilla 5
```

```
>>>     0 veces la casilla 6
>>> La suma del tablero es: 0
```

Nodo

A continuación se define la clase ***Node***, la cual esta conformada por un tablero, el estatus (servirá para la evaluación con el algoritmo Minimax) y el nivel que ocupa en nuestro árbol de búsqueda. Es importante mencionar que se ha definido el método ***__lt__*** con lo que podremos ordenar los nodos de menor a mayor basados en el valor de su tablero.

```
class Node:

    def __init__(self,gameboard,level=0):
        self.gameboard = gameboard
        self.status    = None
        self.level     = level


    def __eq__(self, other):
        return self.gameboard.value == other.gameboard.value


    def __lt__(self, other):
        if( self.gameboard.value == other.gameboard.value ):
            return self.level < other.level
        return self.gameboard.value < other.gameboard.value


    def __copy__(self):
        return Node(self.gameboard,self.level)


    def __deepcopy__(self,memo):
        gameboard = copy.deepcopy(self.gameboard,memo)
        level     = copy.deepcopy(self.level    ,memo)
        return Node(gameboard,level)


    def print(self):
        print( 'El nodo tiene las siguientes características:' )
        print( '\t\t Tablero: {} '.format(self.gameboard.tokens) )
        print( '\t\t Valor  : {} '.format(self.gameboard.value) )
        print( '\t\t Nivel  : {} '.format(self.level) )
        print( '\t\t Status : {} '.format(self.status) )
```

En la siguientes dos celdas planteamos un ejemplo de su instanciación mediante el tablero inicial definido anteriormente.

```
initial_node = Node(initial_gameboard,0)
```

```
initial_node.print()
```

```
>>> El nodo tiene las siguientes características:
>>>     Tablero: {1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 0}
>>>     Valor  : 0
>>>     Nivel  : 0
>>>     Status : None
```

Exploración del árbol de búsqueda

Antes de comenzar a explicar el código de las siguientes celdas es importante entender cuál será el procedimiento. Cuando hacemos la elección de una casilla de valor ***a*** como primer movimiento y el adversario elige una casilla de valor ***b***, la situación que tendremos en el tercer movimiento será equivalente a que los valores se hubiesen escogido en el orden inverso. Teniendo esto en cuenta podemos definir a nuestro árbol de búsqueda basados en las primeras apariciones de un mismo estado.

Si estamos en un nodo cuyo valor en la suma de las casillas seleccionadas puede llevarnos a estados que no rebasen la máxima suma permitida y a la vez a estados que la rebasen, ¿cuáles deberían de tener más prioridad? Basados en esto hemos definido que los nodos que se exploren primero sean aquellos de menor valor en la evaluación mencionada. Así cuando lleguemos al primer nodo cuyo estado rebase máxima suma permitida, todos los estados de nodos no fueron explorados serán ***posiciones perdedoras***.

Lo anterior nos dice que los nodos explorados pudieron ser posiciones perdedoras o ganadoras, pero todos los que no fueron explorados definitivamente serán posiciones perdedoras.

```
# Dado un nodo inicial y las restricciones de tablero y máximo valor:
#    - Explora el árbol de búsqueda
#    - Genera una lista de los nodos explorados y un set de los tableros asociados a ellos.
#    - Devuelve las estructuras mencionadas
def exploration(initial_node,total_rows,max_token_value,max_allowed_value):

    # Inicializamos la frontera
    frontier = [ initial_node ]

    # Tendremos dos sets:
    #    - Uno con los tableros que hemos visto
    #    - Otro con los tableros que hemos explorado
    seen_before_gameboards     = set( [str(list(initial_node.gameboard.tokens.values()))] )
    explored_before_gameboards = set()

    # Una lista de los nodos que se exploraron,
    #    dada la manera de hacer la búsqueda estará ordenada de manera ascendente.
    explored_before_nodes = []


    # Mientras tengamos elementos en la frontera
    while( frontier ):

        # Tomamos el primer nodo de la frontera (el menor)
        actual_node = frontier[0]

        # Si es mayor que nuestro máximo valor permitido,
        #    ?¨para qué explorarlo o a los siguientes nodos?
        if( actual_node.gameboard.value > max_allowed_value ):
            # Entonces salimos del while
            break

        # Al no ser menor quitamos ese elemento de la frontera
        frontier = frontier[1:]

        # Añadimos su tablero al set de tableros explorados
```

```
        explored_before_gameboards.add( str(list(actual_node.gameboard.tokens.values()))
    )

        # Añadimos el nodo al final de la lista de nodos explorados
        explored_before_nodes.append(actual_node)

        # Generamos todos los nodos hijo a través de una función sucesor
        children_nodes = succesor_function(actual_node,total_rows,max_token_value)

        # Para cada nodo hijo revisaremos:
        for new_node in children_nodes:

            # Que NO se haya visto su tablero asociado anteriormente
            if( not (str(list(new_node.gameboard.tokens.values())) in
    seen_before_gameboards) ):

                # De ser cierto lo anterior:

                # Agregamos el tablero a la lista de tableros visto anteriormente
                seen_before_gameboards.add( str(list(new_node.gameboard.tokens.values()))
     )

                # Agregamos el nodo al final dela frontera y reordenamos
                frontier.append( new_node )
                frontier = sorted(frontier)

    # Al tener todos los nodos explorados regresaremos:
    #    - La lista de nodos explorados (está ordenada de menor a mayor)
    #    - El set de nodos explorados (para que la consulta no sea lineal)
    return explored_before_nodes,explored_before_gameboards
```

En la celda anterior se usó la denominada función sucesor, en la siguiente celda podemos ver su implementación:

```
# Dado un nodo:
#    - Genera todos los posibles nodos hijo.
#    - Devuelve una lista con todos los nodos hijo.
def succesor_function(node,total_rows,max_token_value):

    # Lista para guardar los nodos hijo
    children_nodes = []

    # Vamos a iterar sobre los valores que podemos seleccionar en el tablero
    for i in range(1,max_token_value+1):

        # Si la cantidad de valores es menor que el límite (filas),
        #    es decir, se puede seleccionar el valor, entonces:
        if( node.gameboard.tokens[i] < total_rows ):

            # Crear una copia del nodo
            new_node = copy.deepcopy(node)
            # Agregar el movimiento hecho
            new_node.gameboard.tokens[i] += 1
            # Cambiar el valor de la suma total
            new_node.gameboard.value     += i
            # Aumentar en uno el nivel dentro del árbol de búsqueda
            new_node.level += 1
            # Agregar el nodo a la lista de nodos hijo
            children_nodes.append(new_node)

    # Devolver la lista de nodos hijo
    return children_nodes
```

En la siguiente celda podemos ver el resultado de la exploración para los límites planteados en las primeras celdas de este Notebook. También veremos que el total de nodos explorados es 3551, siendo que todos los posibles estados dentro de la representación para los límites planteados son 15625.

```
explored_nodes,explored_gameboards = exploration(initial_node,total_rows,max_token_value,
    max_allowed_value)
```

```
len(explored_nodes)
```

```
>>> 3551
```

Minimax

De los estados que no han sido etiquetados aún, veremos si son o no posiciones ganadoras, con la intención de averiguar para quién son. Para eso, revisaremos en orden de mayor a menor a los nodos. Recordemos que al ver una posición etiquetada como ganadora o perdedora se involucra también la información sobre qué persona recibe esa posición.

Aquí hay que tener mucho cuidado, para ejemplicar lo anterior tengamos los siguientes dos escenarios:

- ***Escenario 1:*** Se han jugado los números 5,6,6,1,6,2,5 *(suma 31)*.
- ***Escenario 2:*** Se han jugado los números 5,4,4,3,4,2,5,5 *(suma 31)*.

Por la definición de posición ganadora, ambos escenarios corresponden a una posición ganadora. ¿Quién gana en cada caso?

¡Coloca aquí tus respuestas!

Entonces podemos decir que una posición ganadora en el turno de A, será buena para A. Pero una posición ganadora en el turno de B, NO será positiva para A. Esto es super importante para proceder a la siguiente parte del Notebook, te recomendamos analizarlo.

Con esto en mente, procederemos a usar una función que nos dirá para cada nodo cuál es su ***"status"***, para ello definiremos que **si el valor es 1, es una posición que benificia al jugador A**, mientras que ***si el valor es -1, es una posición que NO beneficia a A***.

```
def check_strategy(explored_nodes,total_rows,max_token_value):

    # Diccionario que nos dice para un tablero asociado a un nodo,
    #    cuál es el status de nodo
    checked_status_gameboards = dict()

    # Iteramos sobre los nodos de mayor a menor
    for actual_node in reversed(explored_nodes):

```

```
        # Calculamos el status del nodo
        new_status = calculate_node_status(actual_node,checked_status_gameboards,
    total_rows,max_token_value)
        # Agregamos este valor a .status
        actual_node.status = new_status

    # Si el estado inicial (no tener ninguna ficha)
    #    tiene un valor de 1 significa que el estado beneficia al jugador A
    if( explored_nodes[0].status == 1 ):
        return checked_status_gameboards,"El jugador A tiene estrategia ganadora"
    else:
        return checked_status_gameboards,"El jugador A NO tiene estrategia ganadora"
```

En la celda anterior se hizo uso de la función ***calculate_node_status***, a continuación se muestra su definición:

```
def calculate_node_status(node,checked_status_gameboards,total_rows,max_token_value):

    # Obtenemos los nodos hijo del nodo que se analiza actualmente
    children_nodes = succesor_function(node,total_rows,max_token_value)

    # Comenzamos con la situación contraria a la que desean los jugadores:
    if( node.level%2 == 0 ):
        # Jugador A busca cambiar (MAXIMIZAR) el valor de status
        #    en busca de obtener el valor 1
        status = -1
    else:
        # Jugador B busca cambiar (MINIMIZAR) el valor de status
        #    en busca de obtener el valor -1
        status = 1

    # Iteramos sobre los nodos hijo
    for new_node in children_nodes:

        # Para mayor facilidad el key del tablero lo ponemos en una variable
        key_str = str(list(new_node.gameboard.tokens.values()))

        # Revisamos que el tablero del nodo hijo haya sido revisado,
        #    el caso de que esto no ocurra es porque su suma es mayor al máximo permitido
        if( key_str in checked_status_gameboards.keys() ):

            # Maximizar y minimizar dependiendo del jugador
            if( node.level%2 == 0 ):
                status = max(status,checked_status_gameboards[key_str])
            else:
                status = min(status,checked_status_gameboards[key_str])

    # Agregamos un registro al diccionario, así lo podemos usar en las siguientes
    iteraciones
    checked_status_gameboards[ str(list(node.gameboard.tokens.values())) ] = status

    return status
```

```
nodes_status,message = check_strategy(explored_nodes,total_rows,max_token_value)
```

```
len(nodes_status),message
```

```
>>> (3551, 'El jugador A tiene estrategia ganadora')
```

Ejercicio 1

El juego se ha desarrollado de la siguiente manera:

- Jugador A, selecciona el número 1.
- Jugador B, selecciona el número 6.
- Jugador A, selecciona el número 2.
- Jugador B, selecciona el número 2.
- Jugador A, selecciona el número 5.
- Jugador B, selecciona el número 2.
- Jugador A, selecciona el número 3.
- Jugador B, selecciona el número 2.
- Jugador A, selecciona el número 6.

¿Cuál será el valor del atributo ***.status*** para el nodo asociado a este estado del juego? Justifica tu respuesta.

¡Coloca aquí tus respuestas!

¿Cómo ganar?

Si tenemos estrategia ganadora será importante saber cuál es el movimiento que me permite seguir esta estrategia, para ello definiremos las siguientes dos funciones.

```
# Dado un nodo, las restricciones del tablero y qué jugador lo pregunta,
#   sugiere la mejor opción.
def what_to_do(node,nodes_status,total_rows,max_token_value, player="A"):

    # Obtener los nodos hijo del nodo que se revisa actualmente
    children_nodes = succesor_function(node,total_rows,max_token_value)

    # Revisar en orden de mayor a manor los hijos,
    #   solo para que el árbol sea más corto
    for new_node in reversed(children_nodes):

        # Tablero asociado al nodo hijo en un string
        key_str = str(list(new_node.gameboard.tokens.values()))

        # Si el nodo hijo fue evaluado en su status debe de aparecer
        if( key_str in nodes_status.keys() ):

            # Este es el status (1,-1) del nodo hijo
            child_status= nodes_status[key_str]

            # De acuerdo a qué jugador seamos decidimos si nos interesa el valor
            if( player == "A"  and  child_status == 1 ):
                return get_next_move(node,new_node)
            elif( player == "B"  and  child_status == -1 ):
                return get_next_move(node,new_node)
```

```
26
27     # Si llegamos aquí es porque ninguna opción nos beneficiaba
28     return "Ya no hay nada por hacer :("
```

```
1  # Una vez que hemos elegido un nodo hijo que nos beneficia
2  def get_next_move(parent_node, child_node):
3
4      # Obtener el valor de la casilla a seleccionar viendo cual valor es diferente (+1)
5      for a,b in zip(parent_node.gameboard.tokens.items(),child_node.gameboard.tokens.items
       ()):
6          if( a[1] != b[1] ):
7              return "Coloca una ficha sobre una casilla de número {}".format(a[0])
8
9      return None
```

En las siguientes celdas se presenta un ejemplo de funcionamiento para las funciones presentadas en las celdas anteriores:

```
1 my_tokens = { 1:1, 2:2, 3:0, 4:0, 5:3, 6:0 }
2 my_value  = 20
3 actual_gameboard = Gameboard(my_tokens,my_value)
```

```
1 actual_node = Node(actual_gameboard)
```

```
1 what_to_do(actual_node,nodes_status,total_rows,max_token_value,"A")
```

```
>>> 'Coloca una ficha sobre una casilla de número 4'
```

Ejercicio 2

Implementa una función que, dada una lista con los movimientos de cada jugador, nos indique qué movimiento debe realizar el jugador en el siguiente turno para ganar. Si ningún movimiento lo lleva a una posición ganadora también debe ser indicado.

```
1 # Escribe tu código en el cuaderno de Jupyter
```

Te pedimos probar con las siguientes sucesiones de movimientos:

- [5,6,6]
- [2,5,6,6,4]
- [1,5,4,3]
- [5,4,4,3,4,4,3]
- [5,2,5,2,5,2,5]

Por favor, deja indicada la respuesta de estas pruebas.

¡Coloca aquí tus respuestas!