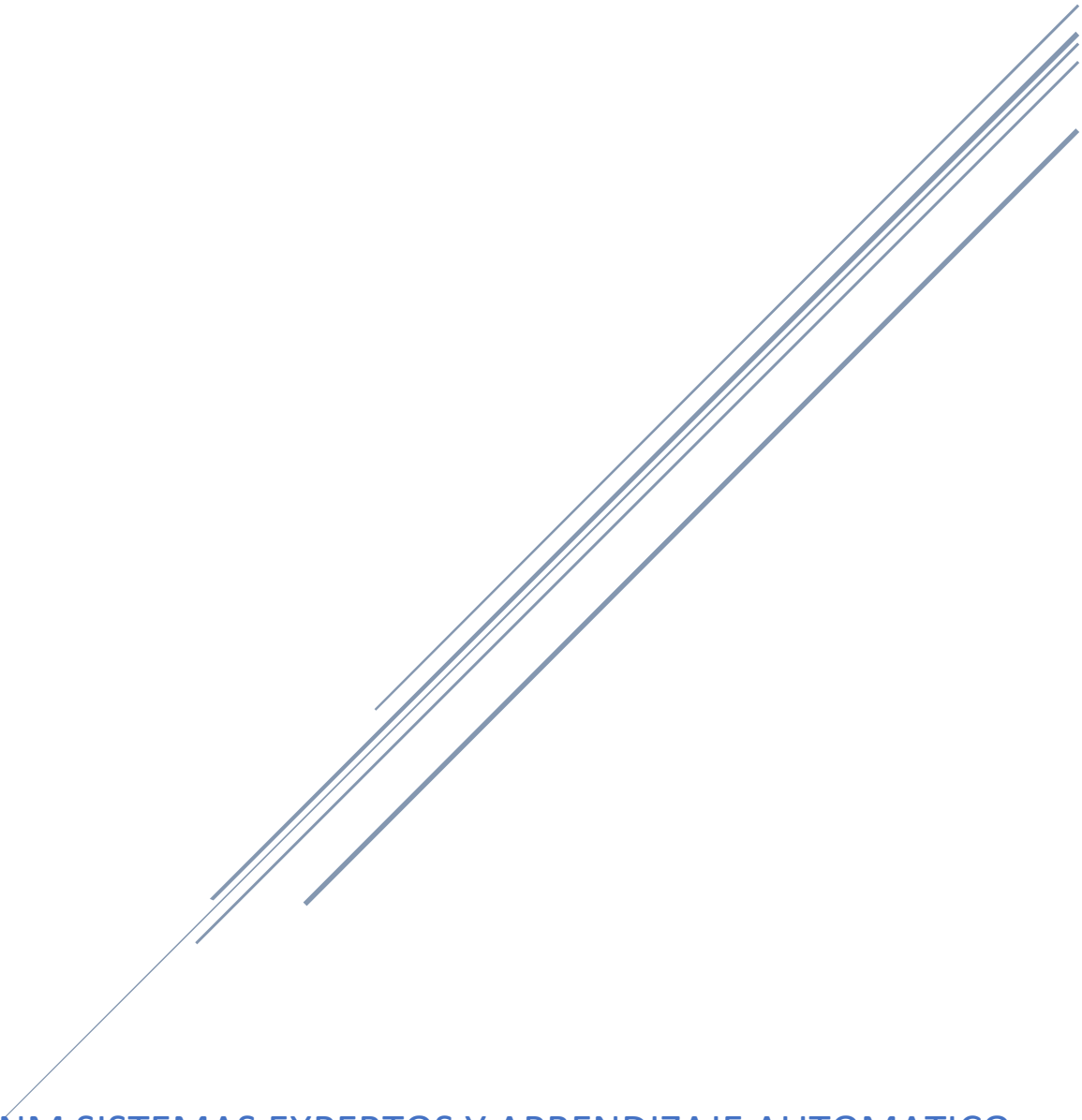
ALGORITMOS IA

DESARROLLO-ALGORITMOS-CODIGOS-EJEMPLOS



TEC NM SISTEMAS EXPERTOS Y APRENDIZAJE AUTOMATICO
JUAN PABLO MARTINEZ A. - EDUARDO LARA GOMEZ

INTRODUCCION

En el presente trabajo se expondrán y definirán los ejemplos de los algoritmos de los capítulos 3 , 4 , 5 y 6 del libro “INTELIGENCIA ARTIFICIAL UN ENFOQUE MODERNO Segunda edición de Stuart J. Russell y Peter Norvig”, todos los ejemplos estarán desarrollados y se implementaran en Python 3

CONCIDERACIONES

- Uso de Python 3
- VS CODE
- Configuraciones de variables de entorno
- Gestor de datos de Python

Contenido

ALGORITMOS CAPITULO 3 TEORIA CODIGO EJEMPLOS

Los agentes resolventes-problemas deciden qué hacer para encontrar secuencias de acciones que conduzcan a los estados deseables. Comenzamos definiendo con precisión los elementos que constituyen el «problema» y su «solución», y daremos diferentes ejemplos para ilustrar estas definiciones. Entonces, describimos diferentes algoritmos de propósito general que podamos utilizar para resolver estos problemas y así comparar las ventajas de cada algoritmo

3.3.1 BEST-FIRST SEARCH

Función de evaluación, la Figura 3.7 muestra el algoritmo. En cada iteración elegimos un nodo en la frontera con un valor mínimo, lo devolvemos si su estado es un estado objetivo y, de lo contrario, aplicamos EXPAND para generar nodos secundarios. Cada nodo hijo se agrega a la frontera si no se ha alcanzado antes, o se vuelve a agregar si ahora se llega con una ruta que tiene un costo de ruta más bajo que cualquier ruta anterior. El algoritmo devuelve una indicación de falla o un nodo que representa un camino hacia un objetivo. Al emplear diferentes funciones, obtenemos diferentes algoritmos específicos, que cubrirá este capítulo.

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Algunos autores han utilizado "la mejor búsqueda primero" para referirse específicamente a una búsqueda con una heurística que intenta predecir qué tan cerca está el final de una ruta a una solución, de modo que las rutas que se consideran más cercanas a una solución se extienden primero . Este tipo específico de búsqueda se denomina búsqueda codiciosa de lo mejor primero o búsqueda heurística pura

EJEMPLO

BESTFS.py

Un algoritmo codicioso es cualquier algoritmo que sigue la heurística de resolución de problemas de hacer la elección localmente óptima en cada etapa En muchos problemas, una estrategia codiciosa no suele producir una solución óptima, pero, no obstante, una heurística codiciosa puede producir soluciones localmente óptimas que se aproximen a una solución globalmente óptima en un período de tiempo razonable.

En el mundo real, elegir la mejor opción es un problema de optimización y, como resultado, tenemos la mejor solución con nosotros. En matemáticas, la optimización es un tema muy amplio cuyo objetivo es encontrar el mejor ajuste para los datos / problema. Estos problemas de optimización pueden resolverse utilizando el algoritmo codicioso ("Un algoritmo codicioso es un algoritmo que sigue la heurística de resolución de problemas de tomar la elección localmente óptima en cada etapa con la intención de encontrar un óptimo global"). Esta es la definición de Wikipedia y encontramos una de las soluciones óptimas teniendo en cuenta las limitaciones. Este es uno de los algoritmos más simples utilizados para la optimización.

Consideremos un problema en el que Hareus obtiene 1500 $ como dinero de bolsillo. Es un anfitrión y necesita comprar lo esencial para el mes. Entonces, reserva 1000 $ para lo esencial y ahora tiene el resto de los 500 $ para sus gastos. Fue al supermercado y allí tuvo que decidir qué comprar según el valor (una medida de cada artículo relacionada con la productividad) y además tener una restricción de 500 $. Este es uno de los problemas de optimización y el siguiente es el código para elegir los elementos de una de las mejores formas.

Idea clave: Máxima productividad con 500 $.

```
# Greedy Algorithm for a Optimisation Problem

# Defined a class for item,
# with its name, value and cost
class Itm(object):
    def __init__(self, name, val, cost):
        self.name = name
        self.val = val
        self.cost = cost

    def getvalue(self):
        return self.val

    def getcost(self):
        return self.cost

    def __str__(self):
        return self.name
```

```
# Defining a function for building a List
# which generates list of items that are
# available at supermart
def buildlist(names, values, costs):
    menu = []
    for i in range(len(names)):
        menu.append(Itm(names[i], values[i], costs[i]))
    return menu

# Implementation of greedy algorithm
# to choose one of the optimum choice
def greedy(items, maxcal, keyfunction):
    itemscopy = sorted(items, key = keyfunction, reverse = True)

    result = []
    totalval = 0
    totalcal = 0

    for i in range(len(items)):
        if (totalcal + itemscopy[i].getcost() <= maxcal):
            result.append(itemscopy[i])
            totalval = totalval + itemscopy[i].getvalue()
            totalcal = totalcal + itemscopy[i].getcost()

    return (result, totalval)


# Main Function
# All values are random
names = ['Ball', 'Gloves', 'Notebook', 'Bagpack', 'Charger', 'Pillow', 'Cakes', 'Pencil']
values = [89,90,95,100,90,79,50,10]
costs = [123,154,25,145,365,150,95,195]
Itemrs = buildlist(names, values, costs)
maxcost = 500 # maximum money he have to spend

taken, totvalue = greedy(Itemrs, maxcost, Itm.getvalue)

print('Total vaule taken : ', totvalue)

# Printing the list of item slected for optimum value
for i in range(len(taken)):
    print('  ', taken[i])
```

BREADTH-FIRST-SEARCH O BFS

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure

function UNIFORM-COST-SEARCH(problem) returns a solution node, or failure
  return BEST-FIRST-SEARCH(problem, PATH-COST)
```

Figure 3.9 Breadth-first search and uniform-cost search algorithms.

En Ciencias de la Computación, Búsqueda en anchura (en inglés BFS - Breadth First Search) es un algoritmo de búsqueda no informada utilizado para recorrer o buscar elementos en un grafo (usado frecuentemente sobre árboles). Intuitivamente, se comienza en la raíz (eligiendo algún nodo como elemento raíz en el caso de un grafo) y se exploran todos los vecinos de este nodo. A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el árbol.

Formalmente, BFS es un algoritmo de búsqueda sin información, que expande y examina todos los nodos de un árbol sistemáticamente para buscar una solución. El algoritmo no usa ninguna estrategia heurística.

Si las aristas tienen pesos negativos aplicaremos el algoritmo de Bellman-Ford en alguna de sus dos versiones

Procedimiento

- Dado un vértice fuente s, Breadth-first search sistemáticamente explora los vértices de G para “descubrir” todos los vértices alcanzables desde s.
- Calcula la distancia (menor número de vértices) desde s a todos los vértices alcanzables.
- Después produce un árbol BF con raíz en s y que contiene a todos los vértices alcanzables.
- El camino desde dt a cada vértice en este recorrido contiene el mínimo número de vértices. Es el camino más corto medido en número de vértices.
- Su nombre se debe a que expande uniformemente la frontera entre lo descubierto y lo no descubierto. Llega a los nodos de distancia k, sólo tras haber llegado a todos los nodos a distancia k-1

EJEMPLO

BFS.py

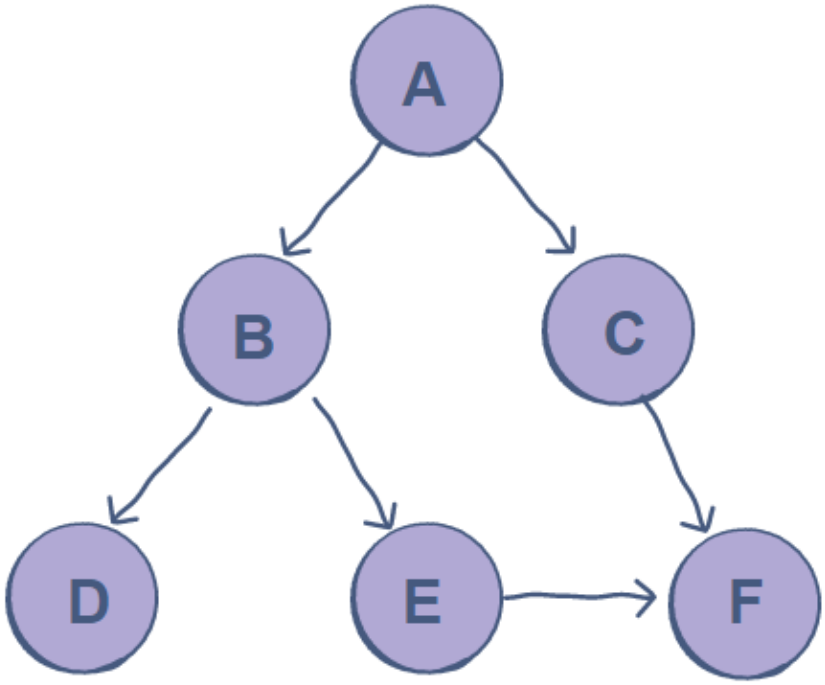
La búsqueda primero en el aliento (BFS) es un algoritmo que se utiliza para recorrer árboles en gráficos o estructuras de datos de árboles. BFS se puede implementar fácilmente utilizando recursividad y estructuras de datos como diccionarios y listas.

El algoritmo

1. Elija cualquier nodo, visite el vértice adyacente no visitado, márquelo como visitado, muéstrelo e insértelo en una cola.
2. Si no quedan vértices adyacentes restantes, elimine el primer vértice de la cola.
3. Repita el paso 1 y el paso 2 hasta que la cola esté vacía o se encuentre el nodo deseado.

Implementación

Considere el gráfico, que se implementa en el siguiente código



```
graph = {
  'A' : ['B','C'],
  'B' : ['D', 'E'],
  'C' : ['F'],
  'D' : [],
  'E' : ['F'],
  'F' : []
}

visited = [] # List to keep track of visited nodes.
queue = []     #Initialize a queue

def bfs(visited, graph, node):
  visited.append(node)
  queue.append(node)

  while queue:
    s = queue.pop(0)
    print (s, end = " ")

    for neighbour in graph[s]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)

# Driver Code
bfs(visited, graph, 'A')
```

Explicación

- **Líneas 3-10** : el gráfico ilustrado se representa mediante una *lista de adyacencia* . Una forma fácil de hacer esto en Python es usar una *estructura de datos de diccionario* , donde cada vértice tiene una lista almacenada de sus nodos adyacentes.
- **Línea 12** : visitedes una lista que se utiliza para realizar un seguimiento de los nodos visitados.
- **Línea 13** : queuees una lista que se utiliza para realizar un seguimiento de los nodos que se encuentran actualmente en la cola.
- **Línea 29** : Los argumentos de la bfsfunción son la visitedlista, el graphen forma de diccionario y el nodo inicial A.
- **Líneas 15-26** : bfssigue el algoritmo descrito anteriormente:
    1. Comprueba y agrega el nodo inicial a la visitedlista y el queue.
    2. Luego, mientras la cola contiene elementos, sigue sacando nodos de la cola, agrega los vecinos de ese nodo a la cola si no están visitados y los marca como visitados.
    3. Esto continúa hasta que la cola está vacía.

Complejidad del tiempo

Dado que se visitan todos los nodos y vértices, la complejidad de tiempo para BFS en un gráfico es O (V + E)O (V + E); dóndeVV es el número de vértices y mimi es el número de aristas.

3.4.2 DIJKSTRA'S ALGORITHM OR UNIFORM-COST SEARCH

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
  return BEST-FIRST-SEARCH(*problem*, PATH-COST)

La búsqueda primero en anchura es óptima cuando todos los costos son iguales, porque siempre expande el nodo no expandido más superficial. Con una extensión sencilla, podemos encontrar un algoritmo que es óptimo con cualquier función costo. En vez de expandir el nodo más superficial, la búsqueda de costo uniforme expande el nodo n con el camino de costo más pequeño. Notemos que si todos los costos son iguales, es idéntico a la búsqueda primero en anchura.

El algoritmo de Dijkstra, también llamado algoritmo de caminos mínimos, es un algoritmo para la determinación del camino más corto, dado un vértice origen, hacia el resto de los vértices en un grafo que tiene pesos en cada arista. Su nombre alude a Edsger Dijkstra, científico de la computación de los Países Bajos que lo concibió en 1956 y lo publicó por primera vez en 1959.

La idea subyacente en este algoritmo consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices; cuando se obtiene el camino más corto desde el vértice origen hasta el resto de los vértices que componen el grafo, el algoritmo se detiene. Se trata de una especialización de la búsqueda de costo uniforme y, como tal, no funciona en grafos con aristas de coste negativo (al elegir siempre el nodo con distancia menor, pueden quedar excluidos de la búsqueda nodos que en próximas iteraciones bajarían el costo general del camino al pasar por una arista con costo negativo).[cita requerida]

Una de sus aplicaciones más importantes reside en el campo de la telemática. Gracias a él, es posible resolver grafos con muchos nodos, lo que sería muy complicado resolver sin dicho algoritmo, encontrando así las rutas más cortas entre un origen y todos los destinos en una red.

PSEUDOCODIGO

```
DIJKSTRA (Grafo G, nodo_fuente s)
    para u ∈ V[G] hacer
        distancia[u] = INFINITO
        padre[u] = NULL
        visto[u] = false
    distancia[s] = 0
    adicionar (cola, (s, distancia[s]))
    mientras que cola no es vacía hacer
        u = extraer_mínimo(cola)
        visto[u] = true
        para todos v ∈ adyacencia[u] hacer
            si ¬ visto[v]
                si distancia[v] > distancia[u] + peso (u, v) hacer
                    distancia[v] = distancia[u] + peso (u, v)
                    padre[v] = u
                    adicionar(cola,(v, distancia[v]))
```

EJEMPLO

DIJ.py

```
import networkx as nx
G=nx.Graph()
e=[('a','b',1),('b','c',2),('a','c',3),('c','d',4),('d','e',2),('b','e',1)]
G.add_weighted_edges_from(e)
print(nx.dijkstra_path(G,'a','e'))
```

3.4.3 DEPTH-FIRST SEARCH

Una Búsqueda en profundidad (en inglés DFS o Depth First Search) es un algoritmo de búsqueda no informada utilizado para recorrer todos los nodos de un grafo o árbol (teoría de grafos) de manera ordenada, pero no uniforme. Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa (Backtracking), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado

PSEUDOCODIGO

```
DFS(grafo G)
  PARA CADA vértice u ∈ V[G] HACER
        estado[u] ← NO_VISITADO
        padre[u] ← NULO
  tiempo ← 0
  PARA CADA vértice u ∈ V[G] HACER
        SI estado[u] = NO_VISITADO ENTONCES
              DFS_Visitar(u,tiempo)
DFS_Visitar(nodo u, int tiempo)
  estado[u] ← VISITADO
  tiempo ← tiempo + 1
  d[u] ← tiempo
  PARA CADA v ∈ Vecinos[u] HACER
        SI estado[v] = NO_VISITADO ENTONCES
              padre[v] ← u
              DFS_Visitar(v,tiempo)
  estado[u] ← TERMINADO
  tiempo ← tiempo + 1
  f[u] ← tiempo
```

EJEMPLO

DFS.py

La búsqueda en profundidad (DFS) es un algoritmo para el recorrido de árboles en estructuras de datos de árboles o gráficos. Se puede implementar fácilmente usando recursividad y estructuras de datos como diccionarios y conjuntos.

El algoritmo

1. Elija cualquier nodo. Si no está visitado, márquelo como visitado y repita en todos sus nodos adyacentes.
2. Repita hasta que se visiten todos los nodos o se encuentre el nodo que se buscará.

Implementación

Considere este gráfico, implementado en el siguiente código:

```
# Using a Python dictionary to act as an adjacency list
graph = {
   'A' : ['B','C'],
   'B' : ['D', 'E'],
   'C' : ['F'],
   'D' : [],
   'E' : ['F'],
   'F' : []
}

visited = set() # Set to keep track of visited nodes.

def dfs(visited, graph, node):
   if node not in visited:
      print (node)
      visited.add(node)
      for neighbour in graph[node]:
         dfs(visited, graph, neighbour)

# Driver Code
dfs(visited, graph, 'A')
```

Explicación

- **Líneas 2-9** : el gráfico ilustrado se representa mediante una **lista de adyacencia** ; una forma fácil de hacerlo en Python es utilizar una estructura de datos de *diccionario* . Cada vértice tiene almacenada una lista de sus nodos adyacentes.
- **Línea 11** : visitedes un conjunto que se utiliza para realizar un seguimiento de los nodos visitados.
- **Línea 21** : Se dfsllama a la función y se le pasa el visitedconjunto, graphen forma de diccionario, y A, que es el nodo inicial.
- **Líneas 13-18** : dfssigue el algoritmo descrito anteriormente:
    1. Primero verifica si el nodo actual no está visitado; en caso afirmativo, se agrega al visitedconjunto.
    2. Luego, para cada vecino del nodo actual, la dfsfunción se invoca nuevamente.
    3. El caso base se invoca cuando se visitan todos los nodos. Luego, la función regresa.

Complejidad del tiempo

Dado que se visitan todos los nodos y vértices, la complejidad de tiempo promedio para DFS en un gráfico es O (V + E)O (V + E), dónde VV es el número de vértices y mimies el número de aristas. En el caso de DFS en un árbol, la complejidad del tiempo esO (V)O (V), dónde VV es el número de nodos.

3.4.4 DEPTH-LIMITED AND ITERATIVE DEEPENING SEARCH

IDS O IDDFS

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution node or failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
```

Búsqueda iterativa de profundización en profundidad

En informática , la búsqueda iterativa de profundización o más específicamente la búsqueda iterativa de profundización primero en profundidad (IDS o IDDFS) es una estrategia de búsqueda de espacio / gráfico de estado en la que una versión de profundidad limitada de búsqueda en profundidad primero se ejecuta repetidamente con una profundidad creciente límites hasta encontrar la meta. IDDFS es óptimo como la búsqueda en amplitud , pero usa mucha menos memoria; en cada iteración, visita los nodos en el árbol de búsqueda en el mismo orden que la búsqueda en profundidad, pero el orden acumulativo en el que los nodos se visitan por primera vez es efectivamente en amplitud

Propiedades

IDDFS combina la eficiencia espacial de la búsqueda en profundidad y la integridad de la búsqueda en amplitud (cuando el factor de ramificación es finito). Si existe una solución, encontrará una ruta de solución con la menor cantidad de arcos.

Dado que las visitas de profundización iterativas indican varias veces, puede parecer un desperdicio, pero resulta no ser tan costoso, ya que en un árbol la mayoría de los nodos están en el nivel inferior, por lo que no importa mucho si los niveles superiores se visitan varias veces. veces.

La principal ventaja de IDDFS en la búsqueda del árbol del juego es que las búsquedas anteriores tienden a mejorar las heurísticas comúnmente utilizadas, como la heurística asesina y la poda alfa-beta , de modo que una estimación más precisa de la puntuación de varios nodos en la búsqueda de profundidad final puede ocurrir, y la búsqueda se completa más rápidamente ya que se realiza en un mejor orden. Por ejemplo, la poda alfa-beta es más eficaz si busca primero los mejores movimientos.

Una segunda ventaja es la capacidad de respuesta del algoritmo. Debido a que las primeras iteraciones usan valores pequeños para{\ Displaystyle d}re, se ejecutan extremadamente rápido. Esto permite que el algoritmo proporcione indicaciones tempranas del resultado casi de inmediato, seguidas de refinamientos como{\ Displaystyle d}reaumenta. Cuando se utiliza en un entorno interactivo, como en un programa de juego de ajedrez , esta función permite que el programa juegue en cualquier momento con el mejor movimiento actual encontrado en la búsqueda que ha completado hasta el momento. Esto se puede expresar como cada profundidad de la búsqueda co recursiva producir una mejor aproximación de la solución, aunque el

trabajo realizado en cada paso es recursivo. Esto no es posible con una búsqueda tradicional en profundidad, que no produce resultados intermedios.

Complejidad del tiempo

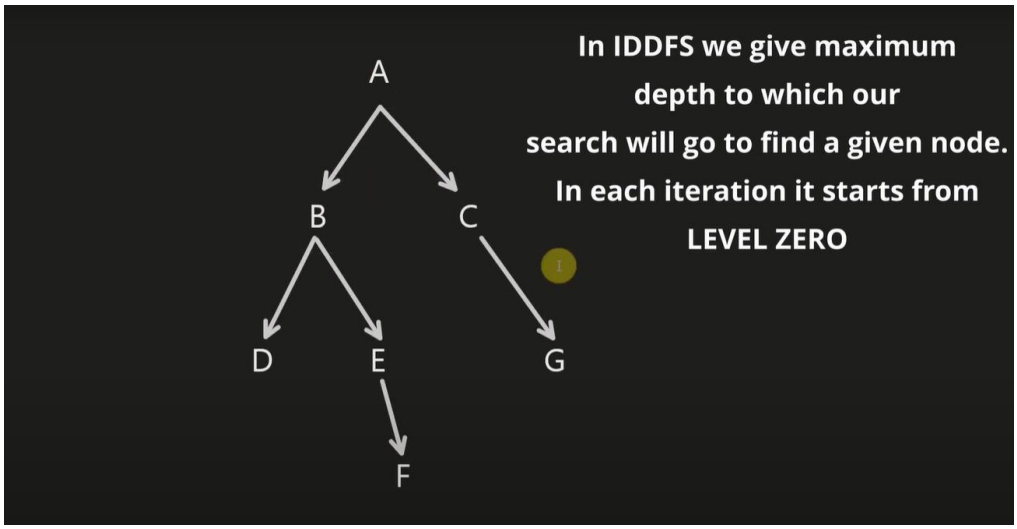
La complejidad temporal de IDDFS en un árbol (bien equilibrado) resulta ser la misma que la búsqueda en amplitud, es decir{\ Displaystyle O (b ^ {d})}O (b ^ {d}), donde{\ Displaystyle b}segundo es el factor de ramificación y {\ Displaystyle d}re es la profundidad de la meta.

PSEUDOCODIGO

```
La función Build-Path (s, μ, B) es
   π ← Find-Shortest-Path (s, μ) (Calcular recursivamente la ruta al nodo de retransmisión)
   eliminar el último nodo de π

   volver π       B (Agregar la pila de búsqueda hacia atrás)
función Profundidad-Búsqueda-limitada-hacia adelante (u, Δ, F) es
   si Δ = 0 entonces

      F ← F       {u} (marca el nodo)
      de retorno
   foreach niño de T hacen
      Búsqueda de profundidad limitada hacia adelante (secundaria, Δ - 1, F)
La función Profundidad-Búsqueda-Limitada-Atrás (u, Δ, B, F) es
   anteponer u a B
   si Δ = 0 entonces
      si u en F   entonces
         devuelve u (Alcanzó el nodo marcado, úselo como un nodo de retransmisión)
      quitar el nodo principal de B
      devolver null
   foreach matriz de T hacen
      μ ← Búsqueda limitada en profundidad hacia atrás (padre, Δ - 1, B, F)

      si μ       nulo  luego
         devuelve μ
   quitar el nodo principal de B
   devolver nulo
la función Find-Shortest-Path (s, t) es
  si s = t entonces
     devuelve <s>
  F, B, Δ ← ∅, ∅, 0
  por siempre hacer
     Búsqueda de profundidad limitada hacia adelante (s, Δ, F)
     foreach δ = Δ, Δ + 1 hacer
        μ ← Búsqueda limitada en profundidad hacia atrás (t, δ, B, F)

        si μ       null luego
           devuelve Build-Path (s, μ, B) (Encontré un nodo de retransmisión)
```

EJEMLO

IDDFS.py



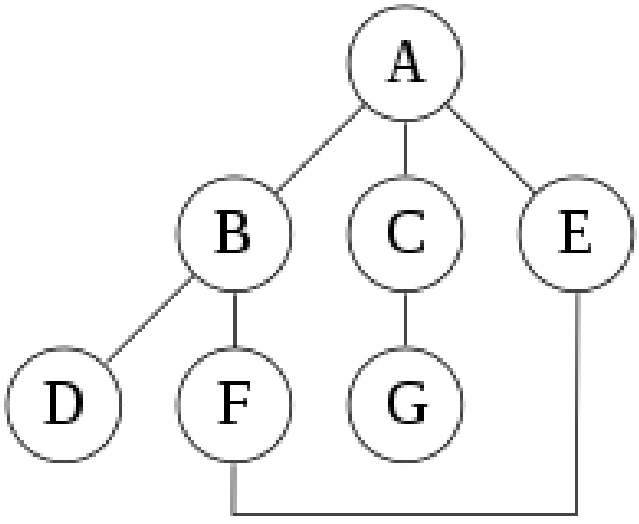

```
graph = {
    'A': ['B', 'C'],
    'B': ['D','E'],
    "C": ['G'],
    'D': [],
    'E': ['F'],
    'G': [],
    'F':[]
}

path = list()

def DFS(currentNode,destination,graph,maxDepth,curList):
    print("Checking for destination",currentNode)
    curList.append(currentNode)
    if currentNode==destination:
        return True
    if maxDepth<=0:
        path.append(curList)
        return False
    for node in graph[currentNode]:
        if DFS(node,destination,graph,maxDepth-1,curList):
            return True
        else:
            curList.pop()
    return False

def iterativeDDFS(currentNode,destination,graph,maxDepth):
    for i in range(maxDepth):
```

```
        curList = list()
        if DFS(currentNode,destination,graph,i,curList):
            return True
    return False

if not iterativeDDFS('A','E',graph,4):
    print("Path is not available")
else:
    print("A path exists")
    print(path.pop())
```

FUNCIONAMIENTO

El algoritmo de búsqueda DFID (Depth-First Iterative-Deepening) fue introducido por primera vez en el año 1977 cuando Slate y Atkin presentaron un programa de ajedrez. El algoritmo combina las ventajas de la búsqueda en anchura y de la búsqueda en profundidad. De la primera guarda el concepto de encontrar la solución siempre a la menor profundidad, mientras que de la segunda conserva el estilo de búsqueda con lo cual asegura que su necesidad de memoria será lineal. Para asegurar la completud del algoritmo, el DFID introduce un límite de profundidad en cada una de sus iteraciones. El desarrollo del algoritmo es el siguiente:

En cada iteración el algoritmo realiza una búsqueda limitada en profundidad. Si en dicha iteración no se alcanza la solución óptima, entonces se incrementa dicho límite en una unidad de profundidad y se vuelve a realizar la búsqueda.



Una búsqueda en profundidad empezando en A, asumiendo que los lados izquierdos del gráfico se toman antes que los derechos y asumiendo que la búsqueda recuerda los nodos visitados previamente y no los repite (dado que esto es un pequeño grafo), visitará los nodos en el siguiente orden: *A, B, D, F, E, C, G*.

Realizando la misma búsqueda sin recordar los nodos previamente visitados, el resultado no tendrá fin: *A, B, D, F, E, A, B, D, F, E, etc.*, esto ocurre por el ciclo entre *A, B, D, F y E*, lo que no permite alcanzar *C* o *G*.

La búsqueda en profundidad iterativa nos soluciona estos bucles y alcanzará los nodos de los siguientes niveles. Asumiendo que procede de izquierda a derecha como antes:

- 0: A
- 1: A (repetido), B, C, E

(Nótese que BPI ha visitado C, lo que no ocurre con la búsqueda en profundidad.)

- 2: A, B, D, F, C, G, E, F

(Nótese que aún visita C, pero aparece más tarde. También visita E por un camino distinto, pero vuelve a F dos veces.)

- 3: A, B, D, F, E, C, G, E, F, B

Para este grafo, cuanta más profundidad se añade, los ciclos "ABFE" y "AEFB" simplemente se alargan antes de que el algoritmo abandone e intente otra rama. Puede recorrer varias veces al mismo nodo siempre y cuando no sea la solución

El siguiente pseudocódigo muestra una BPI implementada en términos de una búsqueda en profundidad limitada recursiva. (LLamada BP).

```
BPI(raíz, objetivo)
{
  profundidad = 0
  repetir
  {
    resultado = BPL(raíz, objetivo, profundidad)
    Si (resultado es una solución)
      devolver resultado
    profundidad = profundidad + 1
  }
}
```

DEPTH-LIMITED-SEARCH

DLS

```
function DEPTH-LIMITED-SEARCH(problem, ℓ) returns a node or failure or cutoff
  frontier ← a LIFO queue (stack) with NODE(problem.INITIAL) as an element
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        add child to frontier
  return result
```

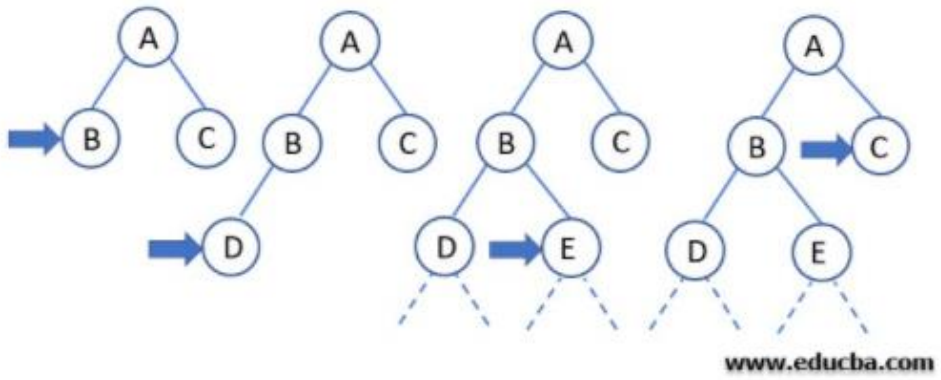
La búsqueda de profundidad limitada es el nuevo algoritmo de búsqueda para búsquedas desinformadas. El problema del árbol ilimitado aparece en el algoritmo de búsqueda de profundidad primero, y se puede solucionar imponiendo un límite o un límite a la profundidad del dominio de búsqueda. Diremos que este límite es el límite de profundidad que hace que la estrategia de búsqueda DFS sea más refinada y organizada en un ciclo finito. Denotamos este límite por 1 y, por lo tanto, esto proporciona la solución al problema de la ruta infinita que se originó anteriormente en el algoritmo DFS. Por lo tanto, la búsqueda limitada en profundidad se puede llamar una versión extendida y refinada del algoritmo DFS. En pocas palabras, podemos decir que para evitar el estado de bucle infinito durante la ejecución de los códigos, el algoritmo de búsqueda de profundidad limitada se está ejecutando en un conjunto finito de profundidad llamado límite de profundidad.

Este algoritmo sigue esencialmente un conjunto de pasos similar al del algoritmo DFS.

1. **El nodo de inicio o el nodo 1 se agrega al comienzo de la pila.**
2. **Luego se marca como visitado y si el nodo 1 no es el nodo objetivo en la búsqueda, entonces colocamos el segundo nodo 2 en la parte superior de la pila.**
3. **A continuación, lo marcamos como visitado y comprobamos si el nodo 2 es el nodo objetivo o no.**
4. **Si no se encuentra que el nodo 2 sea el nodo objetivo, colocamos el nodo 4 en la parte superior de la pila**
5. **Ahora buscamos en el mismo límite de profundidad y nos movemos a lo largo de la profundidad para verificar los nodos objetivo.**
6. **Si el nodo 4 tampoco es el nodo objetivo y se alcanza el límite de profundidad, volvemos a los nodos más cercanos que permanecen sin visitar o sin explorar.**

7. **Luego los empujamos hacia la pila y los marcamos como visitados.**
8. **Continuamos realizando estos pasos de manera iterativa a menos que se alcance el nodo objetivo o hasta que se hayan explorado todos los nodos dentro del límite de profundidad para el objetivo.**

Si fijamos el límite de profundidad en 2, DLS se puede llevar a cabo de manera similar al DFS hasta que se encuentre que el nodo objetivo existe en el dominio de búsqueda del árbol.



Algoritmo del ejemplo

1. Comenzamos encontrando y arreglando un nodo de inicio.
2. Luego buscamos junto con la profundidad usando el algoritmo DFS.
3. Luego seguimos verificando si el nodo actual es el nodo objetivo o no.

Si la respuesta es no: entonces no hacemos nada

Si la respuesta es sí: entonces volvemos

1. Ahora comprobaremos si el nodo actual se encuentra por debajo del límite de profundidad especificado anteriormente o no.

Si la respuesta no es: entonces no hacemos nada

Si la respuesta es sí: Luego exploramos más el nodo y guardamos todos sus sucesores en una pila.

EJEMPLO

EL ALGORITMO ES MUY COMPLEJO SE PROGRAMA EN C O EN JAVA

3.4.5 BIDIRECTIONAL SEARCH OR BIDIRECTIONAL BEST-FIRST SEARCH

B1BFS

La idea de la búsqueda bidireccional es ejecutar dos búsquedas simultáneas: una hacia delante desde el estado inicial y la otra hacia atrás desde el objetivo, parando cuando las dos búsquedas se encuentren en el centro (Figura 3.16). La motivación es que bd2 es mucho menor que bd, o en la figura, el área de los dos círculos pequeños es menor que el área de un círculo grande centrado en el inicio y que alcance al objetivo.

La búsqueda bidireccional se implementa teniendo una o dos búsquedas que comprueban antes de ser expandido si cada nodo está en la frontera del otro árbol de búsqueda; si esto ocurre, se ha encontrado una solución. Por ejemplo, si un problema tiene una solución a profundidad d 6, y en cada dirección se ejecuta la búsqueda primero en anchura, entonces, en el caso peor, las dos búsquedas se encuentran cuando se han expandido todos los nodos excepto uno a profundidad 3. Para b 10, esto significa un total de 22.200 nodos generados, comparado con 11.111.100 para una búsqueda primero en anchura estándar. Verificar que un nodo pertenece al otro árbol de búsqueda se puede hacer en un tiempo constante con una tabla hash, así que la complejidad en tiempo de la búsqueda bidireccional es O(bd2). Por lo menos uno de los árboles de búsqueda se debe mantener en memoria para que se pueda hacer la comprobación de pertenencia, de ahí que la complejidad en espacio es también O(bd2). Este requerimiento de espacio es la debilidad más significativa de la búsqueda bidireccional. El algoritmo es completo y óptimo (para costos uniformes) si las búsquedas son primero en anchura; otras combinaciones pueden sacrificar la completitud, optimización, o ambas.

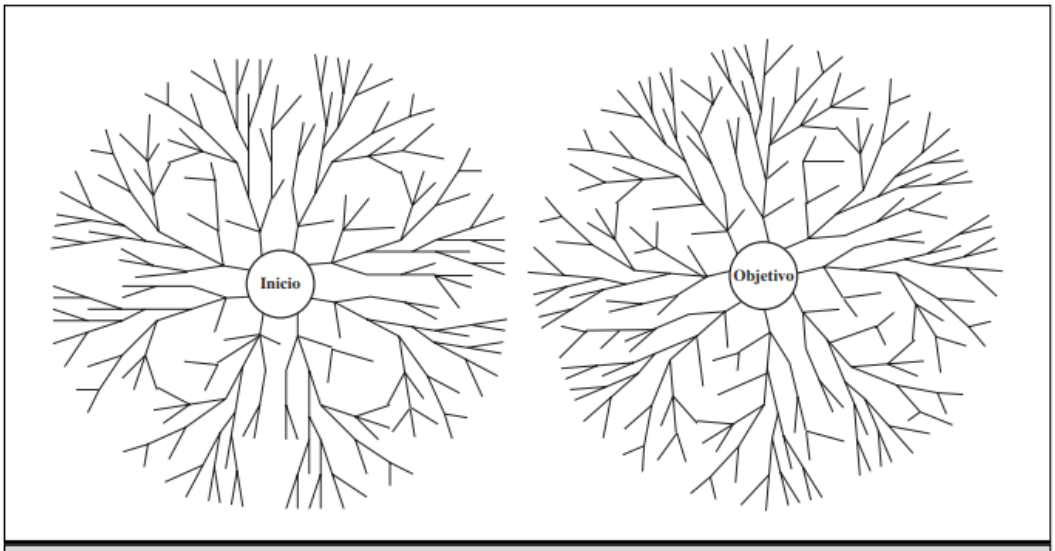


Figura 3.16 Un esquema de una búsqueda bidireccional que está a punto de tener éxito, cuando una rama desde el nodo inicio encuentra una rama desde el nodo objetivo.

Buscar un gráfico es un problema bastante famoso y tiene mucho uso práctico. Ya hemos discutido aquí cómo buscar un vértice objetivo a partir de un vértice fuente usando BFS . En la búsqueda de gráficos normal usando BFS / DFS, comenzamos nuestra búsqueda en una dirección, generalmente desde el vértice de origen hacia el vértice de destino, pero ¿qué pasa si comenzamos la búsqueda en ambas direcciones simultáneamente?

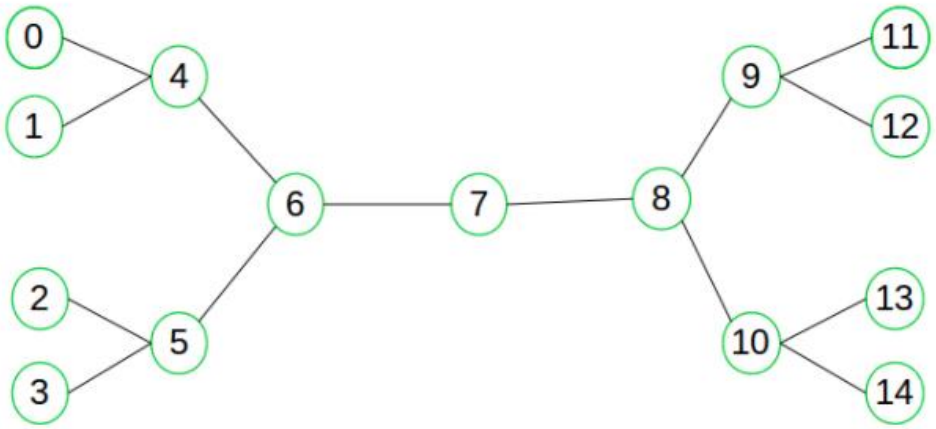
La búsqueda bidireccional es un algoritmo de búsqueda de gráficos que encuentra la ruta más pequeña desde la fuente hasta el vértice de la meta. Ejecuta dos búsquedas simultáneas:

1. Fuente del formulario de búsqueda hacia adelante / vértice inicial hacia el vértice de la meta
2. Objetivo de formulario de búsqueda hacia atrás / vértice de destino hacia el vértice de origen

La búsqueda bidireccional reemplaza el gráfico de búsqueda único (que probablemente crezca exponencialmente) con dos subgráficos más pequeños, uno que comienza desde el vértice inicial y otro que comienza desde el vértice del objetivo. La búsqueda termina cuando dos gráficos se cruzan.

Al igual que el algoritmo A * , la búsqueda bidireccional puede guiarse por una estimación heurística de la distancia restante desde la fuente hasta el objetivo y viceversa para encontrar el camino más corto posible.

Considere seguir un ejemplo simple:



Suponga que queremos encontrar si existe un camino desde el vértice 0 al vértice 14. Aquí podemos ejecutar dos búsquedas, una desde el vértice 0 y otra desde el vértice 14. Cuando tanto la búsqueda hacia adelante como hacia atrás se encuentran en el vértice 7, sabemos que tenemos encontró una ruta del nodo 0 al 14 y la búsqueda se puede terminar ahora. Podemos ver claramente que hemos evitado con éxito exploraciones innecesarias.

Debido a que en muchos casos es más rápido, reduce drásticamente la cantidad de exploración requerida.
Suponga que si el factor de ramificación del árbol es **by la** distancia del vértice de la meta desde la fuente es **d** , entonces la complejidad de búsqueda normal de BFS / DFS sería $O(b^d)$. Por otro lado, si ejecutamos dos operaciones de búsqueda, entonces la complejidad sería $O(b^{d/2})$ para cada búsqueda y la complejidad total sería $O(b^{d/2} + b^{d/2})$ que es mucho menor que $O(b^d)$.

¿Cuándo utilizar el enfoque bidireccional?

Podemos considerar un enfoque bidireccional cuando:

1. Tanto el estado inicial como el objetivo son únicos y están completamente definidos.
2. El factor de ramificación es exactamente el mismo en ambas direcciones.

Medidas de desempeño

- Integridad: la búsqueda bidireccional está completa si se utiliza BFS en ambas búsquedas.
- Optimidad: es óptimo si se utiliza BFS para la búsqueda y las rutas tienen un costo uniforme.
- Complejidad de tiempo y espacio: La complejidad de tiempo y espacio es $O(b^{d/2})$.

A continuación se muestra una implementación muy simple que representa el concepto de búsqueda bidireccional usando BFS. Esta implementación considera caminos no dirigidos sin ningún peso.

```
# Python3 program for Bidirectional BFS
# Search to check path between two vertices

# Class definition for node to
# be added to graph
class AdjacentNode:

    def __init__(self, vertex):

        self.vertex = vertex
        self.next = None

# BidirectionalSearch implementation
class BidirectionalSearch:

    def __init__(self, vertices):

        # Initialize vertices and
        # graph with vertices
```

```
        self.vertices = vertices
        self.graph = [None] * self.vertices

        # Initializing queue for forward
        # and backward search
        self.src_queue = list()
        self.dest_queue = list()

        # Initializing source and
        # destination visited nodes as False
        self.src_visited = [False] * self.vertices
        self.dest_visited = [False] * self.vertices

        # Initializing source and destination
        # parent nodes
        self.src_parent = [None] * self.vertices
        self.dest_parent = [None] * self.vertices

    # Function for adding undirected edge
    def add_edge(self, src, dest):

        # Add edges to graph

        # Add source to destination
        node = AdjacentNode(dest)
        node.next = self.graph[src]
        self.graph[src] = node

        # Since graph is undirected add
        # destination to source
        node = AdjacentNode(src)
        node.next = self.graph[dest]
        self.graph[dest] = node

    # Function for Breadth First Search
    def bfs(self, direction = 'forward'):

        if direction == 'forward':

            # BFS in forward direction
            current = self.src_queue.pop(0)
            connected_node = self.graph[current]

            while connected_node:
                vertex = connected_node.vertex
```

```
            if not self.src_visited[vertex]:
                self.src_queue.append(vertex)
                self.src_visited[vertex] = True
                self.src_parent[vertex] = current

            connected_node = connected_node.next
        else:

            # BFS in backward direction
            current = self.dest_queue.pop(0)
            connected_node = self.graph[current]

            while connected_node:
                vertex = connected_node.vertex

                if not self.dest_visited[vertex]:
                    self.dest_queue.append(vertex)
                    self.dest_visited[vertex] = True
                    self.dest_parent[vertex] = current

                connected_node = connected_node.next

    # Check for intersecting vertex
    def is_intersecting(self):

        # Returns intersecting node
        # if present else -1
        for i in range(self.vertices):
            if (self.src_visited[i] and
                self.dest_visited[i]):
                return i

        return -1

    # Print the path from source to target
    def print_path(self, intersecting_node,
                   src, dest):

        # Print final path from
        # source to destination
        path = list()
        path.append(intersecting_node)
        i = intersecting_node
```

```
        while i != src:
            path.append(self.src_parent[i])
            i = self.src_parent[i]

        path = path[::-1]
        i = intersecting_node

        while i != dest:
            path.append(self.dest_parent[i])
            i = self.dest_parent[i]

        print("*****Path*****")
        path = list(map(str, path))

        print(' '.join(path))

    # Function for bidirectional searching
    def bidirectional_search(self, src, dest):

        # Add source to queue and mark
        # visited as True and add its
        # parent as -1
        self.src_queue.append(src)
        self.src_visited[src] = True
        self.src_parent[src] = -1

        # Add destination to queue and
        # mark visited as True and add
        # its parent as -1
        self.dest_queue.append(dest)
        self.dest_visited[dest] = True
        self.dest_parent[dest] = -1

        while self.src_queue and self.dest_queue:

            # BFS in forward direction from
            # Source Vertex
            self.bfs(direction = 'forward')

            # BFS in reverse direction
            # from Destination Vertex
            self.bfs(direction = 'backward')

            # Check for intersecting vertex
            intersecting_node = self.is_intersecting()
```

```
            # If intersecting vertex exists
            # then path from source to
            # destination exists
            if intersecting_node != -1:
                print(f"Path exists between {src} and {dest}")
                print(f"Intersection at : {intersecting_node}")
                self.print_path(intersecting_node,
                                src, dest)
                exit(0)
        return -1

# Driver code
if __name__ == '__main__':

    # Number of Vertices in graph
    n = 15

    # Source Vertex
    src = 0

    # Destination Vertex
    dest = 14

    # Create a graph
    graph = BidirectionalSearch(n)
    graph.add_edge(0, 4)
    graph.add_edge(1, 4)
    graph.add_edge(2, 5)
    graph.add_edge(3, 5)
    graph.add_edge(4, 6)
    graph.add_edge(5, 6)
    graph.add_edge(6, 7)
    graph.add_edge(7, 8)
    graph.add_edge(8, 9)
    graph.add_edge(8, 10)
    graph.add_edge(9, 11)
    graph.add_edge(9, 12)
    graph.add_edge(10, 13)
    graph.add_edge(10, 14)

    out = graph.bidirectional_search(src, dest)

    if out == -1:
        print(f"Path does not exist between {src} and {dest}")
```

```
# This code is contributed by Nirjhari Jankar
```

3.5.1 GREEDY BEST-FIRST SEARCH

CODICIOSO.py

ES LO MISMO QUE BFS

Un algoritmo codicioso es cualquier algoritmo que sigue la heurística de resolución de problemas de hacer la elección localmente óptima en cada etapa En muchos problemas, una estrategia codiciosa no suele producir una solución óptima, pero, no obstante, una heurística codiciosa puede producir soluciones localmente óptimas que se aproximen a una solución globalmente óptima en un período de tiempo razonable.

EJEMPLO DE LAS MONEDAS

Algoritmo codicioso para encontrar el número mínimo de monedas

Última actualización: 30-09-2020

Dado un valor V, si queremos hacer un cambio para V Rs, y tenemos un suministro infinito de cada una de las denominaciones en moneda india, es decir, tenemos un suministro infinito de {1, 2, 5, 10, 20, 50, 100, 500, 1000} monedas / billetes valorados, ¿cuál es el número mínimo de monedas y / o billetes necesarios para realizar el cambio?

Entrada: V = 70
Salida: 2
Necesitamos una nota de 50 rupias y una nota de 20 rupias.

Entrada: V = 121
Salida: 3
Necesitamos un billete de 100 R, un billete de 20 R y una moneda de 1 R.

Solución: Enfoque codicioso.

Enfoque: una intuición común sería tomar las monedas de mayor valor primero. Esto puede reducir la cantidad total de monedas necesarias. Comience con la denominación más grande posible y siga agregando denominaciones mientras el valor restante sea mayor que 0.

Algoritmo:

1. Ordene la matriz de monedas en orden decreciente.
2. Inicializar el resultado como vacío.
3. Encuentre la denominación más grande que sea menor que la cantidad actual.

4. Agregue la denominación encontrada al resultado. Reste el valor de la denominación encontrada de la cantidad.
5. Si la cantidad se vuelve 0, imprima el resultado.
6. De lo contrario, repita los pasos 3 y 4 para obtener un nuevo valor de V.

Salida:

```
Lo siguiente es un número mínimo de cambios

para 93:50 20 20 2 1
```

Análisis de complejidad:

- **Complejidad del tiempo:** O (V).
- **Espacio auxiliar:** O (1) ya que no se utiliza espacio adicional.

```
# Python 3 program to find minimum
# number of denominations
def findMin(V):
    # All denominations of Indian Currency
    deno = [1, 2, 5, 10, 20, 50,
            100, 500, 1000]
    n = len(deno)

    # Initialize Result
    ans = []

    # Traverse through all denomination
    i = n - 1
    while(i >= 0):

        # Find denominations
        while (V >= deno[i]):
            V -= deno[i]
            ans.append(deno[i])
        i -= 1
    # Print result
    for i in range(len(ans)):
        print(ans[i], end = " ")
# Driver Code
if __name__ == '__main__':
    n = 93
    print("Following is minimal number",
        "of change for", n, ": ", end = "")
    findMin(n)

# This code is contributed by
# Surendra_Gangwar
```

3.5.2 A* SEARCH

A * (pronunciado "A-star") es un algoritmo de búsqueda de recorrido y recorrido de gráfico , que se utiliza a menudo en muchos campos de la informática debido a su integridad, optimización y eficiencia óptima. Un inconveniente práctico importante es su {\ Displaystyle O (b ^ {d})}O (b ^ d)complejidad del espacio, ya que almacena todos los nodos generados en la memoria. Por lo tanto, en los sistemas prácticos de enrutamiento de viajes , generalmente es superado por algoritmos que pueden preprocesar el gráfico para lograr un mejor rendimiento, [2] así como por enfoques limitados por memoria; sin embargo, A * sigue siendo la mejor solución en muchos casos.

Peter Hart , Nils Nilsson y Bertram Raphael del Stanford Research Institute (ahora SRI International) publicaron por primera vez el algoritmo en 1968. [4] Se puede ver como una extensión del algoritmo de 1959 de Edsger Dijkstra . A * logra un mejor rendimiento utilizando heurísticas para guiar su búsqueda.

A * es un algoritmo de búsqueda informado , o una búsqueda de mejor primero , lo que significa que se formula en términos de gráficos ponderados : a partir de un nodo inicial específico de un gráfico, su objetivo es encontrar una ruta al nodo objetivo dado que tiene el menor costo (menor distancia recorrida, menor tiempo, etc.). Lo hace manteniendo un árbol de caminos que se originan en el nodo de inicio y extendiendo esos caminos un borde a la vez hasta que se satisface su criterio de terminación.

En cada iteración de su bucle principal, A * necesita determinar cuál de sus caminos extender. Lo hace basándose en el costo de la ruta y una estimación del costo requerido para extender la ruta hasta la meta. Específicamente, A * selecciona la ruta que minimiza

Descripcion

donde *n* es el siguiente nodo en el camino, *g* (*n*) es el coste de la ruta desde el nodo de inicio a *n* , y *h* (*n*) es una heurística función que estima el coste de la ruta más barato de *n* a la meta. A * termina cuando la ruta que elige extender es una ruta desde el inicio hasta la meta o si no hay rutas elegibles para ser extendidas. La función heurística es específica del problema. Si la función heurística es admisible , lo que significa que nunca sobreestima el costo real para llegar al objetivo, se garantiza que A * devolverá una ruta de menor costo desde el principio hasta el objetivo.

Las implementaciones típicas de A * usan una cola de prioridad para realizar la selección repetida de nodos de costo mínimo (estimado) para expandir. Esta cola de prioridad se conoce como *conjunto abierto* o *franja* . En cada paso del algoritmo, el nodo con el más bajo *f* (*x*) valor se elimina de la cola, los *f* y *g* valores de sus vecinos se actualizan en consecuencia, y estos vecinos se añaden a la cola. El algoritmo continúa hasta que un nodo eliminado (por lo tanto, el nodo con el valor *f* más bajo de todos los nodos marginales) es un nodo objetivo. [a] La *f*El valor de esa meta es entonces también el costo del camino más corto, ya que *h* en la meta es cero en una heurística admisible.

El algoritmo descrito hasta ahora nos da solo la longitud del camino más corto. Para encontrar la secuencia real de pasos, el algoritmo se puede revisar fácilmente para que cada nodo en la ruta realice un seguimiento de su predecesor. Después de ejecutar este algoritmo, el nodo final apuntará a su predecesor, y así sucesivamente, hasta que el predecesor de algún nodo sea el nodo de inicio.

Por ejemplo, al buscar la ruta más corta en un mapa, $h(x)$ podría representar la distancia en línea recta hasta la meta, ya que es físicamente la distancia más pequeña posible entre dos puntos cualesquiera. Para un mapa de cuadrícula de un videojuego, usar la distancia de Manhattan o la distancia de octil se vuelve mejor según el conjunto de movimientos disponibles (4 o 8 direcciones).

Si la heurística h satisface la condición adicional $h(x) \leq d(x, y) + h(y)$ para cada borde (x, y) de la gráfica (donde d denota la longitud de ese borde), entonces h se llama monótono o consistente. Con una heurística consistente, se garantiza que A * encontrará una ruta óptima sin procesar ningún nodo más de una vez y A * es equivalente a ejecutar el algoritmo de Dijkstra con el costo reducido $d'(x, y) = d(x, y) + h(y) - h(x)$

El siguiente pseudocódigo describe el algoritmo:

```
función reconstruct_path ( cameFrom , current )
    total_path := {actual}
    mientras que actual en cameFrom . Claves :
        actual := cameFrom [ actual ]
        total_path . anteponer ( actual )
    return total_path

// A * encuentra un camino desde el principio hasta la meta.
// h es la función heurística. h (n) estima el costo para alcanzar la meta desde el nodo n.
function A_Star ( start , goal , h )
    // El conjunto de nodos descubiertos que pueden necesitar ser (re) expandidos.
    // Inicialmente, solo se conoce el nodo de inicio.
    // Esto generalmente se implementa como un min-heap o una cola de prioridad en lugar de un
conjunto de hash.
    openSet := {inicio}

    // Para el nodo n, cameFrom [n] es el nodo inmediatamente anterior a él en la ruta más barata
desde el inicio
    //
    an actualmente conocido. cameFrom := un mapa vacío

    // Para el nodo n, gScore [n] es el costo de la ruta más barata desde el principio hasta n
actualmente conocida.
    gScore := mapa con valor predeterminado de Infinity gScore [ inicio ] := 0


    // Para el nodo n, fScore [n]: = gScore [n] + h (n). fScore [n] representa nuestra mejor estimación
actual en cuanto a
    // qué tan corta puede ser una ruta de principio a fin si pasa por n.
    fScore := mapa con valor predeterminado de Infinity fScore [ inicio ] := h ( inicio )
```

```
   mientras openSet está no vaciar
      // Esta operación puede ocurrir en O (1) tiempo si openSet es un min-montón o una cola de
prioridad
      actual : = el nodo en openSet que tiene el más bajo fScore [] valor
      si actual = meta
         retorno reconstruct_path ( camefrom , actual )

      openSet . Eliminar ( actual )
      para cada vecino del actual
         // d (actual, vecino) es el peso del borde del actual al vecino
         // tentative_gScore es la distancia desde el inicio hasta el vecino hasta el actual
         tentative_gScore : = gScore [ actual ] + d ( actual , vecino )
         si tentative_gScore < gScore [ vecino ]
            // Esta ruta al vecino es mejor que cualquier anterior. ¡Grabe!
            vino de [ vecino] : = gScore actual
            [ vecino ] : = tentative_gScore fScore [ vecino ] : = gScore [ vecino ] + h ( vecino ) si el
vecino no está en openSet . agregar ( vecino )



   // conjunto abierto está vacía pero el objetivo no se alcanzó
   retorno fracaso
```

EJEMPLO A* COMO IR DE S A G MUY SIMILAR A DIJKSTRA

A.py

```
tree = {'S': [['A', 1], ['B', 5], ['C', 8]],
        'A': [['S', 1], ['D', 3], ['E', 7], ['G', 9]],
        'B': [['S', 5], ['G', 4]],
        'C': [['S', 8], ['G', 5]],
        'D': [['A', 3]],
        'E': [['A', 7]]}

tree2 = {'S': [['A', 1], ['B', 2]],
         'A': [['S', 1]],
         'B': [['S', 2], ['C', 3], ['D', 4]],
         'C': [['B', 2], ['E', 5], ['F', 6]],
         'D': [['B', 4], ['G', 7]],
         'E': [['C', 5]],
         'F': [['C', 6]]
         }

heuristic = {'S': 8, 'A': 8, 'B': 4, 'C': 3, 'D': 5000, 'E': 5000, 'G': 0}
heuristic2 = {'S': 0, 'A': 5000, 'B': 2, 'C': 3, 'D': 4, 'E': 5000, 'F': 5000, 'G': 0}

cost = {'S': 0}            # total cost for nodes visited


def AStarSearch():
    global tree, heuristic
    closed = []            # closed nodes
    opened = [['S', 8]]    # opened nodes

    '''find the visited nodes'''
    while True:
        fn = [i[1] for i in opened]    # fn = f(n) = g(n) + h(n)
        chosen_index = fn.index(min(fn))
        node = opened[chosen_index][0]  # current node
        closed.append(opened[chosen_index])
        del opened[chosen_index]
        if closed[-1][0] == 'G':       # break the loop if node G has been found
            break
        for item in tree[node]:
            if item[0] in [closed_item[0] for closed_item in closed]:
                continue
            cost.update({item[0]: cost[node] + item[1]})          # add nodes to cost dictionary
            fn_node = cost[node] + heuristic[item[0]] + item[1]    # calculate f(n) of current node
            temp = [item[0], fn_node]
```

```
            opened.append(temp)                                # store f(n) of current node in array opene
d

    '''find optimal sequence'''
    trace_node = 'G'                        # correct optimal tracing node, initialize as node G
    optimal_sequence = ['G']                    # optimal node sequence
    for i in range(len(closed)-2, -1, -1):
        check_node = closed[i][0]           # current node
        if trace_node in [children[0] for children in tree[check_node]]:
            children_costs = [temp[1] for temp in tree[check_node]]
            children_nodes = [temp[0] for temp in tree[check_node]]

            '''check whether h(s) + g(s) = f(s). If so, append current node to optimal sequence
            change the correct optimal tracing node to current node'''
            if cost[check_node] + children_costs[children_nodes.index(trace_node)] == cost[trace_
node]:
                optimal_sequence.append(check_node)
                trace_node = check_node
    optimal_sequence.reverse()              # reverse the optimal sequence

    return closed, optimal_sequence


if __name__ == '__main__':
    visited_nodes, optimal_nodes = AStarSearch()
    print('visited nodes: ' + str(visited_nodes))
    print('optimal nodes sequence: ' + str(optimal_nodes))
```

ALGORITMOS CAPITULO 4

TEORIA CODIGO

EJEMPLOS

HILL-CLIMBING SEARCH ALGORITHM

ALGORITMO DE BÚSQUEDA DE ESCALADA

Hill Climbing es una búsqueda heurística utilizada para problemas de optimización matemática en el campo de la inteligencia artificial.

Dado un gran conjunto de entradas y una buena función heurística, intenta encontrar una solución suficientemente buena al problema. Es posible que esta solución no sea el máximo óptimo global.

- En la definición anterior, **los problemas de optimización matemática** implican que la escalada resuelve los problemas en los que necesitamos maximizar o minimizar una función real dada eligiendo valores de las entradas dadas. Ejemplo- Problema del vendedor ambulante en el que necesitamos minimizar la distancia recorrida por el vendedor.
- "Búsqueda heurística" significa que este algoritmo de búsqueda puede no encontrar la solución óptima al problema. Sin embargo, dará una buena solución en **un tiempo razonable.**
- Una **función heurística** es una función que clasificará todas las alternativas posibles en cualquier paso de bifurcación en el algoritmo de búsqueda según la información disponible. Ayuda al algoritmo a seleccionar la mejor ruta entre las posibles rutas.

Características de la escalada

1. **Variante del algoritmo de generación y prueba:** es una variante del algoritmo de generación y prueba. El algoritmo de generación y prueba es el siguiente:

1. Generar posibles soluciones.
2. Pruebe para ver si esta es la solución esperada.
3. Si se ha encontrado la solución, salga, vaya al paso 1.

2 **Utiliza el enfoque codicioso :** en cualquier punto del espacio de estados, la búsqueda se mueve solo en esa dirección, lo que optimiza el costo de la función con la esperanza de encontrar la solución óptima al final.

En ciencia de la computación, el algoritmo hill climbing, también llamado Algoritmo de Escalada Simple o Ascenso de colinas es una técnica de optimización matemática que pertenece a la familia de los algoritmos de búsqueda local. Es un algoritmo iterativo que comienza con una solución arbitraria a un problema, luego intenta encontrar una mejor solución variando incrementalmente un único elemento de la solución. Si el cambio produce una mejor solución, otro cambio incremental se le realiza a la nueva solución, repitiendo este proceso hasta que no se puedan encontrar mejoras. Suele llamarse a esta búsqueda algoritmo voraz local, porque toma un estado vecino "bueno" sin pensar en la próxima acción.

El Algoritmo Hill climbing es interesante para encontrar un óptimo local (una solución que no puede ser mejorada considerando una configuración de la vecindad) pero no garantiza encontrar la mejor solución posible (el óptimo global) de todas las posibles soluciones (el espacio de búsqueda). La característica de que sólo el óptimo local puede ser garantizado puede ser remediada utilizando reinicios (búsqueda local repetida), o esquemas más complejos basados en iteraciones, como búsqueda local iterada, en memoria, como optimización de búsqueda reactiva y búsqueda tabú, o modificaciones estocásticas, como simulated annealing.

La relativa simplicidad de este algoritmo lo hace una primera elección popular entre los algoritmos de optimización. Es usado ampliamente en inteligencia artificial, para alcanzar un estado final desde un nodo de inicio. La elección del próximo nodo y del nodo de inicio puede ser variada para obtener una lista de algoritmos de la misma familia. Aunque algoritmos más avanzados tales como simulated annealing o búsqueda tabú pueden devolver mejores resultados, en algunas situaciones hill climbing opera sin diferencias. El hill climbing con frecuencia puede producir un mejor resultado que otros algoritmos cuando la cantidad de tiempo disponible para realizar la búsqueda es limitada, por ejemplo en sistemas en tiempo real. El algoritmo puede devolver una solución válida aún si es interrumpido en cualquier momento antes de que finalice.

Por ejemplo, el hill climbing puede ser aplicado al problema del viajante. Es fácil encontrar una solución inicial que visite todas las ciudades pero sería muy pobre comparada con la solución óptima. El algoritmo comienza con dicha solución y realiza pequeñas mejoras a esta, tales como intercambiar el orden en el cual dos ciudades son visitadas. Eventualmente, es probable que se obtenga una ruta más corta.

El problema del vendedor viajero, problema del vendedor ambulante, problema del agente viajero o problema del viajante (TSP por sus siglas en inglés (Travelling Salesman Problem)), responde a la siguiente pregunta: dada una lista de ciudades y las distancias entre cada par de ellas, ¿cuál es la ruta más corta posible que visita cada ciudad exactamente una vez y al finalizar regresa a la ciudad origen? Este es un problema NP-Hard dentro en la optimización combinatoria, muy importante en investigación operativa y en ciencias de la computación.

El problema fue formulado por primera vez en 1930 y es uno de los problemas de optimización más estudiados. Es usado como prueba para muchos métodos de optimización. Aunque el problema es computacionalmente complejo, se conoce gran cantidad de heurísticas y métodos exactos, así que es posible resolver planteamientos concretos del problema desde cien hasta miles de ciudades.

El TSP tiene diversas aplicaciones aún en su formulación más simple, tales como: la planificación, la logística y la fabricación de circuitos electrónicos. Un poco modificado, aparece como subproblema en muchos campos como la secuenciación de ADN. En esta aplicación, el concepto de "ciudad" representa, por ejemplo: clientes, puntos de soldadura o fragmentos de ADN y el concepto de "distancia" representa el tiempo de viaje o costo, o una medida de similitud entre los fragmentos de ADN. En muchas aplicaciones, restricciones adicionales como el límite de recurso o las ventanas de tiempo hacen el problema considerablemente difícil. El TSP es un caso especial de los Problemas del Comprador Viajante (travelling purchaser problem).

En la teoría de la complejidad computacional, la versión de decisión del TSP (donde, dada una longitud "L", el objetivo es decidir qué grafo tiene un camino menor que L) pertenece a la clase de los problemas NP-completos. Por tanto, es probable que en el caso peor el tiempo de ejecución para cualquier algoritmo que resuelva el TSP aumente de forma exponencial con respecto al número de ciudades.

EJEMPLO

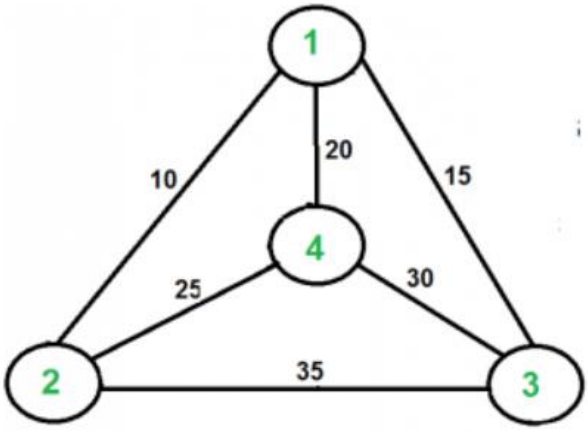
HC.py

Problema del vendedor ambulante (TSP): dado un conjunto de ciudades y distancias entre cada par de ciudades, el problema es encontrar la ruta más corta posible que visite cada ciudad exactamente una vez y regrese al punto de partida.

Tenga en cuenta la diferencia entre el ciclo hamiltoniano y el TSP. El problema del ciclo hamiltoniano es averiguar si existe un recorrido que visite cada ciudad exactamente una vez. Aquí sabemos que existe el Hamiltonian Tour (porque el gráfico está completo) y, de hecho, existen muchos de estos tours, el problema es encontrar un ciclo hamiltoniano de peso mínimo.

Por ejemplo, considere el gráfico que se muestra en la figura del lado derecho. Un recorrido de TSP en el gráfico es 1-2-4-3-1. El costo del tour es 10 + 25 + 30 + 15 que es 80.

El problema es un famoso problema NP-hard. No existe una solución conocida en tiempo polinómico para este problema.



Ejemplos:

```
Salida del gráfico dado:
Peso mínimo Ciclo Hamiltoniano:
10 + 25 + 30 + 15: = 80
```

En esta publicación, se analiza la implementación de una solución simple.

1. Considere la ciudad 1 como punto inicial y final. Dado que la ruta es cíclica, podemos considerar cualquier punto como punto de partida.
2. ¡Genere todo (n-1)! permutaciones de ciudades.
3. Calcule el costo de cada permutación y realice un seguimiento del costo mínimo de permutación.
4. Devuelve la permutación con costo mínimo.

```python
# Python3 program to implement traveling salesman
# problem using naive approach.
from sys import maxsize
from itertools import permutations
V = 4

# implementation of traveling Salesman Problem
def travellingSalesmanProblem(graph, s):

    # store all vertex apart from source vertex
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)

    # store minimum weight Hamiltonian Cycle
    min_path = maxsize
    next_permutation=permutations(vertex)
    for i in next_permutation:

        # store current Path weight(cost)
        current_pathweight = 0

        # compute current path weight
        k = s
        for j in i:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][s]

        # update minimum
        min_path = min(min_path, current_pathweight)

    return min_path


# Driver Code
if __name__ == "__main__":

    # matrix representation of graph
    graph = [[0, 10, 15, 20], [10, 0, 35, 25],
             [15, 35, 0, 30], [20, 25, 30, 0]]
    s = 0
    print(travellingSalesmanProblem(graph, s))
```

ALGORITMOS CAPITULO 5 TEORIA CODIGO EJEMPLOS

The alpha–beta search algorithm

- La poda alfa-beta es una versión modificada del algoritmo minimax. Es una técnica de optimización del algoritmo minimax.
- Como hemos visto en el algoritmo de búsqueda minimax, el número de estados de juego que tiene que examinar son exponenciales en la profundidad del árbol. Ya que no podemos eliminar el exponente, pero podemos cortarlo a la mitad. Por lo tanto, existe una técnica mediante la cual, sin verificar cada nodo del árbol del juego, podemos calcular la decisión correcta de minimax, y esta técnica se llama poda . Esto implica dos parámetros de umbral Alfa y beta para una expansión futura, por lo que se denomina poda alfa-beta . También se le llama algoritmo Alfa-Beta .
- La poda alfa-beta se puede aplicar a cualquier profundidad de un árbol y, a veces, no solo poda las hojas del árbol sino también todo el subárbol.
- Los dos parámetros se pueden definir como:
    - Alfa: la mejor opción (valor más alto) que hemos encontrado hasta ahora en cualquier punto a lo largo del camino de Maximizer. El valor inicial de alfa es -∞ .
    - Beta: la mejor opción (el valor más bajo) que hemos encontrado hasta ahora en cualquier punto de la ruta de Minimizer. El valor inicial de beta es + ∞ .
- La poda Alfa-beta a un algoritmo minimax estándar devuelve el mismo movimiento que el algoritmo estándar, pero elimina todos los nodos que realmente no afectan la decisión final, pero hacen que el algoritmo sea lento. Por lo tanto, al podar estos nodos, el algoritmo se acelera.

Pseudocódigo para poda alfa-beta:

```
1.  unción minimax (nodo, profundidad, alfa, beta, maximizingPlayer) es
2.  si la  profundidad == 0  o el nodo es un nodo terminal, entonces
3.  devolver la  evaluación estática del nodo
4.
5.  si  MaximizingPlayer entonces       // para Maximizer Player
6.     maxEva = -infinito
7.     para  cada hijo del nodo  hacer
8.     eva = minimax (niño, profundidad- 1 , alfa, beta, falso)
9.    maxEva = max (maxEva, eva)
10.   alpha = max (alfa, maxEva)
11.    si  beta <= alfa
12.  romper
13.  return  maxEva
14.
15. else // para el reproductor minimizador
16.    minEva = + infinito
17.    para  cada hijo del nodo  hacer
18.    eva = minimax (niño, profundidad- 1 , alfa, beta,  verdadero )
19.    minEva = min (minEva, eva)
20.    beta = min (beta, eva)
21.     si  beta <= alfa
22.   romper
23.  volver  minEva
```

EJEMPLO

ABETA.py

```
# Python3 program to demonstrate
# working of Alpha-Beta Pruning

# Initial values of Aplha and Beta
MAX, MIN = 1000, -1000

# Returns optimal value for current player
#(Initially called for root and maximizer)
def minimax(depth, nodeIndex, maximizingPlayer,
            values, alpha, beta):

    # Terminating condition. i.e
    # leaf node is reached
    if depth == 3:
        return values[nodeIndex]

    if maximizingPlayer:

        best = MIN

        # Recur for left and right children
        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex * 2 + i,
                          False, values, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)

            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best

    else:
        best = MAX

        # Recur for left and
        # right children
        for i in range(0, 2):

            val = minimax(depth + 1, nodeIndex * 2 + i,
```

```
                            True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)

            # Alpha Beta Pruning
            if beta <= alpha:
                break

        return best

# Driver Code
if __name__ == "__main__":

    values = [3, 5, 6, 9, 1, 2, 0, -1]
    print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))

# This code is contributed by Rituraj Jain
```

Provoca como salida el valor optimo 3

Monte Carlo tree search algorithm (MCTS).

Monte Carlo Tree Search (MCTS) es una técnica de búsqueda en el campo de la Inteligencia Artificial (IA). Es un algoritmo de búsqueda probabilístico y heurístico que combina las implementaciones clásicas de búsqueda de árbol junto con los principios de aprendizaje automático del aprendizaje por refuerzo.

En la búsqueda de árbol, siempre existe la posibilidad de que la mejor acción actual no sea realmente la acción más óptima. En tales casos, el algoritmo MCTS se vuelve útil ya que continúa evaluando otras alternativas periódicamente durante la fase de aprendizaje ejecutándolas, en lugar de la estrategia óptima percibida actual. Esto se conoce como " ***compensación de exploración-explotación*** ". Explota las acciones y estrategias que se encuentran mejores hasta ahora, pero también debe continuar explorando el espacio local de decisiones alternativas y descubrir si podrían reemplazar las mejores actuales.

La exploración ayuda a explorar y descubrir las partes inexploradas del árbol, lo que podría resultar en encontrar un camino más óptimo. En otras palabras, podemos decir que la exploración expande la amplitud del árbol más que su profundidad. La exploración puede ser útil para asegurar que MCTS no pase por alto ningún camino potencialmente mejor. Pero rápidamente se vuelve ineficaz en situaciones con una gran cantidad de pasos o repeticiones. Para evitarlo, se equilibra con la explotación. La explotación se ciñe a un único camino que tiene el mayor valor estimado. Este es un enfoque codicioso y extenderá la profundidad del árbol más que su ancho. En palabras simples Por esta característica, MCTS se vuelve particularmente útil para tomar decisiones óptimas en problemas de Inteligencia Artificial (IA).

Algoritmo de búsqueda de árbol de Monte Carlo (MCTS):

en MCTS, los nodos son los componentes básicos del árbol de búsqueda. Estos nodos se forman en función del resultado de una serie de simulaciones. El proceso de búsqueda de árboles de Monte Carlo se puede dividir en cuatro pasos distintos, a saber, selección, expansión, simulación y retropropagación. Cada uno de estos pasos se explica en detalle a continuación:

- **Selección:** en este proceso, el algoritmo MCTS atraviesa el árbol actual desde el nodo raíz utilizando una estrategia específica. La estrategia utiliza una función de evaluación para seleccionar de manera óptima los nodos con el valor estimado más alto. MCTS utiliza la fórmula del límite de confianza superior (UCB) aplicada a los árboles como estrategia en el proceso de selección para atravesar el árbol. Equilibra la compensación de exploración-explotación. Durante el recorrido del árbol, se selecciona un nodo en función de algunos parámetros que devuelven el valor máximo. Los parámetros se caracterizan por la fórmula que se usa típicamente para este propósito que se da a continuación.

- $$S_i = x_i + C\sqrt{\frac{\ln(t)}{n_i}}$$

  dónde;

- S_i = valor de un nodo i
  x_i = media empírica de un nodo i
  C = una constante
  t = número total de simulaciones

  Al atravesar un árbol durante el proceso de selección, el nodo hijo que devuelva el mayor valor de la ecuación anterior será el que se seleccione. Durante el recorrido, una vez que se encuentra un nodo hijo que también es un nodo hoja, el MCTS salta al paso de expansión.

- **Expansión:** en este proceso, se agrega un nuevo nodo hijo al árbol a ese nodo que se alcanzó de manera óptima durante el proceso de selección.
- **Simulación:** En este proceso, se realiza una simulación eligiendo movimientos o estrategias hasta lograr un resultado o estado predefinido.
- **Retropropagación:** después de determinar el valor del nodo recién agregado, el árbol restante debe actualizarse. Entonces, se realiza el proceso de retropropagación, donde se retropropaga desde el nuevo nodo al nodo raíz. Durante el proceso, se incrementa el número de simulación almacenada en cada nodo. Además, si la simulación del nuevo nodo da como resultado una victoria, también se incrementa el número de ganancias.

P. CODIGO ALGORITMO MONTE CARLO

```
# main function for the Monte Carlo Tree Search
def monte_carlo_tree_search(root):

    while resources_left(time, computational power):
        leaf = traverse(root)
        simulation_result = rollout(leaf)
        backpropagate(leaf, simulation_result)

    return best_child(root)

# function for node traversal
def traverse(node):
    while fully_expanded(node):
        node = best_uct(node)

    # in case no children are present / node is terminal
    return pick_univisted(node.children) or node

# function for the result of the simulation
def rollout(node):
    while non_terminal(node):
        node = rollout_policy(node)
    return result(node)

# function for randomly selecting a child node
def rollout_policy(node):
    return pick_random(node.children)

# function for backpropagation
def backpropagate(node, result):
    if is_root(node) return
    node.stats = update_stats(node, result)
    backpropagate(node.parent)

# function for selecting the best child
# node with highest number of visits
def best_child(node):
    pick child with highest number of visits
```

Ventajas de la búsqueda de árboles de Monte Carlo:

1. MCTS es un algoritmo simple de implementar.
2. Monte Carlo Tree Search es un algoritmo heurístico. MCTS puede operar de manera efectiva sin ningún conocimiento en el dominio en particular, aparte de las reglas y condiciones finales, y puede encontrar sus propios movimientos y aprender de ellos jugando playouts aleatorios.
3. El MCTS se puede guardar en cualquier estado intermedio y ese estado se puede utilizar en casos de uso futuros cuando sea necesario.
4. MCTS admite la expansión asimétrica del árbol de búsqueda en función de las circunstancias en las que está funcionando.

Desventajas de la búsqueda de árboles de Monte Carlo:

1. A medida que el crecimiento del árbol se acelera después de algunas iteraciones, requiere una gran cantidad de memoria.
2. Hay un pequeño problema de confiabilidad con Monte Carlo Tree Search. En ciertos escenarios, puede haber una sola rama o ruta, que podría conducir a una pérdida contra la oposición cuando se implemente para esos juegos por turnos. Esto se debe principalmente a la gran cantidad de combinaciones y es posible que cada uno de los nodos no se visite el número suficiente de veces para comprender su resultado o resultado a largo plazo.
3. El algoritmo MCTS necesita una gran cantidad de iteraciones para poder decidir efectivamente la ruta más eficiente. Entonces, hay un pequeño problema de velocidad allí.

EJEMPLO

MCTS.py

```
"""
A minimal implementation of Monte Carlo tree search (MCTS) in Python 3
Luke Harold Miles, July 2019, Public Domain Dedication
See also https://en.wikipedia.org/wiki/Monte_Carlo_tree_search
https://gist.github.com/qpwo/c538c6f73727e254fdc7fab81024f6e1
"""
from abc import ABC, abstractmethod
from collections import defaultdict
import math


class MCTS:
    "Monte Carlo tree searcher. First rollout the tree then choose a move."

    def __init__(self, exploration_weight=1):
        self.Q = defaultdict(int)  # total reward of each node
        self.N = defaultdict(int)  # total visit count for each node
        self.children = dict()  # children of each node
        self.exploration_weight = exploration_weight

    def choose(self, node):
        "Choose the best successor of node. (Choose a move in the game)"
        if node.is_terminal():
            raise RuntimeError(f"choose called on terminal node {node}")

        if node not in self.children:
            return node.find_random_child()

        def score(n):
            if self.N[n] == 0:
                return float("-inf")  # avoid unseen moves
            return self.Q[n] / self.N[n]  # average reward

        return max(self.children[node], key=score)

    def do_rollout(self, node):
        "Make the tree one layer better. (Train for one iteration.)"
        path = self._select(node)
        leaf = path[-1]
        self._expand(leaf)
        reward = self._simulate(leaf)
        self._backpropagate(path, reward)
```

```
    def _select(self, node):
        "Find an unexplored descendent of `node`"
        path = []
        while True:
            path.append(node)
            if node not in self.children or not self.children[node]:
                # node is either unexplored or terminal
                return path
            unexplored = self.children[node] - self.children.keys()
            if unexplored:
                n = unexplored.pop()
                path.append(n)
                return path
            node = self._uct_select(node)  # descend a layer deeper

    def _expand(self, node):
        "Update the `children` dict with the children of `node`"
        if node in self.children:
            return  # already expanded
        self.children[node] = node.find_children()

    def _simulate(self, node):
        "Returns the reward for a random simulation (to completion) of `node
`"
        invert_reward = True
        while True:
            if node.is_terminal():
                reward = node.reward()
                return 1 - reward if invert_reward else reward
            node = node.find_random_child()
            invert_reward = not invert_reward

    def _backpropagate(self, path, reward):
        "Send the reward back up to the ancestors of the leaf"
        for node in reversed(path):
            self.N[node] += 1
            self.Q[node] += reward
            reward = 1 - reward  # 1 for me is 0 for my enemy, and vice vers
a

    def _uct_select(self, node):
        "Select a child of node, balancing exploration & exploitation"

        # All children of node should already be expanded:
```

```
        assert all(n in self.children for n in self.children[node])

        log_N_vertex = math.log(self.N[node])

        def uct(n):
            "Upper confidence bound for trees"
            return self.Q[n] / self.N[n] + self.exploration_weight * math.sq
rt(
                log_N_vertex / self.N[n]
            )

        return max(self.children[node], key=uct)


class Node(ABC):
    """
    A representation of a single board state.
    MCTS works by constructing a tree of these Nodes.
    Could be e.g. a chess or checkers board state.
    """

    @abstractmethod
    def find_children(self):
        "All possible successors of this board state"
        return set()

    @abstractmethod
    def find_random_child(self):
        "Random successor of this board state (for more efficient simulation
)"
        return None

    @abstractmethod
    def is_terminal(self):
        "Returns True if the node has no children"
        return True

    @abstractmethod
    def reward(self):
        "Assumes `self` is terminal node. 1=win, 0=loss, .5=tie, etc"
        return 0

    @abstractmethod
    def __hash__(self):
        "Nodes must be hashable"
```

```
        return 123456789

    @abstractmethod
    def __eq__(node1, node2):
        "Nodes must be comparable"
        return True
tictactoe.py
"""
An example implementation of the abstract Node class for use in MCTS
If you run this file then you can play against the computer.
A tic-tac-toe board is represented as a tuple of 9 values, each either None,
True, or False, respectively meaning 'empty', 'X', and 'O'.
The board is indexed by row:
0 1 2
3 4 5
6 7 8
For example, this game board
O - X
O X -
X - -
corrresponds to this tuple:
(False, None, True, False, True, None, True, None, None)
"""

from collections import namedtuple
from random import choice
from monte_carlo_tree_search import MCTS, Node

_TTTB = namedtuple("TicTacToeBoard", "tup turn winner terminal")

# Inheriting from a namedtuple is convenient because it makes the class
# immutable and predefines __init__, __repr__, __hash__, __eq__, and others
class TicTacToeBoard(_TTTB, Node):
    def find_children(board):
        if board.terminal:  # If the game is finished then no moves can be m
ade
            return set()
        # Otherwise, you can make a move in each of the empty spots
        return {
            board.make_move(i) for i, value in enumerate(board.tup) if value
 is None
        }

    def find_random_child(board):
        if board.terminal:
```

```
            return None  # If the game is finished then no moves can be made
        empty_spots = [i for i, value in enumerate(board.tup) if value is No
ne]
        return board.make_move(choice(empty_spots))

    def reward(board):
        if not board.terminal:
            raise RuntimeError(f"reward called on nonterminal board {board}"
)
        if board.winner is board.turn:
            # It's your turn and you've already won. Should be impossible.
            raise RuntimeError(f"reward called on unreachable board {board}"
)
        if board.turn is (not board.winner):
            return 0  # Your opponent has just won. Bad.
        if board.winner is None:
            return 0.5  # Board is a tie
        # The winner is neither True, False, nor None
        raise RuntimeError(f"board has unknown winner type {board.winner}")

    def is_terminal(board):
        return board.terminal

    def make_move(board, index):
        tup = board.tup[:index] + (board.turn,) + board.tup[index + 1 :]
        turn = not board.turn
        winner = _find_winner(tup)
        is_terminal = (winner is not None) or not any(v is None for v in tup
)
        return TicTacToeBoard(tup, turn, winner, is_terminal)

    def to_pretty_string(board):
        to_char = lambda v: ("X" if v is True else ("O" if v is False else "
 "))
        rows = [
            [to_char(board.tup[3 * row + col]) for col in range(3)] for row
in range(3)
        ]
        return (
            "\n  1 2 3\n"
            + "\n".join(str(i + 1) + " " + " ".join(row) for i, row in enume
rate(rows))
            + "\n"
        )
```

```
def play_game():
    tree = MCTS()
    board = new_tic_tac_toe_board()
    print(board.to_pretty_string())
    while True:
        row_col = input("enter row,col: ")
        row, col = map(int, row_col.split(","))
        index = 3 * (row - 1) + (col - 1)
        if board.tup[index] is not None:
            raise RuntimeError("Invalid move")
        board = board.make_move(index)
        print(board.to_pretty_string())
        if board.terminal:
            break
        # You can train as you go, or only at the beginning.
        # Here, we train as we go, doing fifty rollouts each turn.
        for _ in range(50):
            tree.do_rollout(board)
        board = tree.choose(board)
        print(board.to_pretty_string())
        if board.terminal:
            break


def _winning_combos():
    for start in range(0, 9, 3):  # three in a row
        yield (start, start + 1, start + 2)
    for start in range(3):  # three in a column
        yield (start, start + 3, start + 6)
    yield (0, 4, 8)  # down-right diagonal
    yield (2, 4, 6)  # down-left diagonal


def _find_winner(tup):
    "Returns None if no winner, True if X wins, False if O wins"
    for i1, i2, i3 in _winning_combos():
        v1, v2, v3 = tup[i1], tup[i2], tup[i3]
        if False is v1 is v2 is v3:
            return False
        if True is v1 is v2 is v3:
            return True
    return None


def new_tic_tac_toe_board():
```

```
    return TicTacToeBoard(tup=(None,) * 9, turn=True, winner=None, terminal=
False)


if __name__ == "__main__":
    play_game()
```

ALGORITMOS CAPITULO 6 TEORIA CODIGO EJEMPLOS

The MIN-CONFLICTS local search algorithm

Un algoritmo de conflictos mínimos comienza con un estado inicial completo, este estado inicial se puede generar al azar o utilizando algún enfoque codicioso para acelerar el tiempo de procesamiento. El estado inicial probablemente tendrá conflictos (restricciones no cumplidas) y el algoritmo de conflictos mínimos intentará eliminar estos conflictos paso a paso, cambiando el valor de una variable en cada paso.

Un algoritmo de conflictos mínimos seleccionará una variable en conflicto al azar en cada paso y asignará el valor con el número mínimo de conflictos a esta variable. El algoritmo podría quedarse atascado en un mínimo local y no podría encontrar una solución, si no aplicamos algo de aleatoriedad. Debemos romper empates al azar y también incluir candidatos menos óptimos al azar para poder saltar de las mínimas locales.

Sodoku

EJEMPLO

MCL.py

Esta implementación se puede utilizar para resolver acertijos sodoku simples y difíciles, no se garantiza que este algoritmo encuentre una solución siempre. Estoy usando algo de aleatoriedad (var_ratey val_rate) para incluir variables que no están en conflicto e incluir valores que no son óptimos. El algoritmo puede resolver un rompecabezas muy rápido con algo de suerte.

```
import os
import copy
# Opciones que tiene el menú del sudoku
MENU = ["  a - Fijar valor de una celda. ",
    "  b - Borrar valor de una celda",
    "  c - Agregar opción a una celda.",
    "  d - Borrar opción de una celda.",
    "  e - Verificar factible.",
    "  f - Deshacer.",
    "  g - Mostrar opciones de una celda.",
    "  h - Salir.",""]

class Celda():
    """Una celda contiene un valor y un conjunto de opciones de valores
    para esta celda. Una celda vacía se representa como una celda cuyo
    valor es cero. Una celda puede ser fija, es decir, su valor no puede
     ser modificado una vez que es introducido. El invariante de la
     clase celda especifica que si una celda no está vacía entonces
     el conjunto de opciones debe estar vacío."""
    def __init__(self):
        self.__valor = 0
        self.__fija = False
```

```
        self.__opciones = []

    def fijar_valor(self, dato, fija = False):
        """Método para introducir un valor en una celda e indicar si
        el mismo es o no fijo. Si la celda ya es fija entonces no se
        puede modificar y por tanto devuelve False. En caso contrario
        devuelve True."""
        if not self.__fija:
            self.__valor = dato
            self.__fija = fija
            self.__opciones = []
            return True
        return False

    def borrar_valor(self):
        """Método que borra el contenido de una celda."""
        if not self.__fija:
            self.__valor = 0
            return True
        return False

    def valor(self):
        """Método que devuelve el valor que aloja la celda."""
        return self.__valor

    def valida(self):
        """Método que devuelve True si la celda no está vacía y
        False en caso contrario."""
        return self.__valor is not 0

    def fija(self):
        """Método que devuelve True si la celda es fija y False
        en caso contrario."""
        return self.__fija

    def nueva_opcion(self, dato):
        """Método que guarda una nueva opción en la celda. Si el dato ya
        existe en el conjunto de posibles opciones, entonces devuelve
        False sino devuelve True."""
        if not self.__fija and dato not in self.__opciones:
            self.__opciones.append(dato)
            return True
        return False

    def borrar_opcion(self, dato):
```

```
        """Método que borra una opción de la celda. Si el dato no existe
         en el conjunto de posibles opciones, entonces devuelve False
         sino devuelve True."""
        if not self.__fija and dato in self.__opciones:
            self.__opciones.remove(dato)
            return True
        return False

    def opciones(self):
        """Método que devuelve las opciones de la celda."""
        self.__opciones.sort()
        return self.__opciones

    def __str__(self):
        """Método llamado para crear una cadena de texto que represente
        el valor definitivo de la celda en caso de que sea válido, sino
        mostará un "_"."""
        return str(self.__valor) if self.__valor else "_"

class Sudoku():
    """Un sudoku contiene un tablero de celdas de tamaño 9×9 y una pila
     de jugadas. El tablero puede tener tanto celdas fijas como
     modificables. El invariante de la clase especifica que cada celda
     del tablero y cada elemento de la lista de opciones de la celda
     debe tener valores entre 0 y 9 en donde el cero representa una
     celda vacía. Además no puede haber en una misma fila, columna o
     bloque dos valores iguales, ni en la celda ni en la lista de
     opciones de la misma."""
    def __init__(self):
        self.__tablero = [[Celda() for x in range(9)] for y in range(9)]
        self.__jugadas = []
        try:
            os.system("python generador.py > tablero.txt")
            f = open("tablero.txt", "r")
        except:
            print "Error: Imposible cargar tablero de juego"
            sys.exit(1)
        f.readline()
        fila = 0
        while fila < 9:
            linea = f.readline().split()
            if linea != []:
                for i in range(9):
                    if linea[i] != "_":
                        self.__tablero[fila][i].fijar_valor\
```

```
                    (int(linea[i]),True)
            fila+= 1
        try: os.system("rm -f tablero.txt")
        finally: f.close()

    def fijar_valor(self, fila, columna, valor):
        """Método que fija el valor dado en la celda ubicada en la
         posición (fila,columna) del tablero. En caso de que la jugada
          sea valida, en cuyo caso se devolverá True, o False en caso
          contrario."""
        if self.es_factible(fila, columna, valor):
            jugada = []
            jugada.append((fila-1, columna-1,copy.deepcopy\
            (self.__tablero[fila-1][columna-1])))
            for i in range(9):
                if i!= columna-1 and valor in\
                 self.__tablero[fila-1][i].opciones():
                    jugada.append((fila-1, i,copy.deepcopy\
                    (self.__tablero[fila-1][i])))
                    self.__tablero[fila-1][i].borrar_opcion(valor)
                if i!= fila-1 and valor in\
                 self.__tablero[i][columna-1].opciones():
                    jugada.append((i, columna-1,copy.deepcopy\
                    (self.__tablero[i][columna-1])))
                    self.__tablero[i][columna-1].borrar_opcion(valor)
            f = 3 * ((fila-1)/3)
            c = 3 * ((columna-1)/3)
            for i in range(f,f+3):
                for j in range(c,c+3):
                    if i!= fila-1 and j!= columna-1 and valor in\
                     self.__tablero[i][j].opciones():
                        jugada.append((i, j,copy.deepcopy\
                        (self.__tablero[i][j])))
                        self.__tablero[i][j].borrar_opcion(valor)
            self.__jugadas.append(jugada)
            return self.__tablero[fila-1][columna-1].fijar_valor(valor)
        return False

    def borrar_valor(self, fila, columna):
        """Método que elimina el valor de la celda ubicada en la
        posición (fila,columna) del tablero."""
        if 0 < fila < 10 and 0 < columna < 10 and\
        self.__tablero[fila-1][columna-1].valida() and\
        not self.__tablero[fila-1][columna-1].fija():
            self.__jugadas.append([(fila-1, columna-1,copy.deepcopy\
```

```
            (self.__tablero[fila-1][columna-1]))])
            return self.__tablero[fila-1][columna-1].borrar_valor()
        return False

    def finalizado(self):
        """Método que devuelve True si el juego ha finalizado,
         es decir, todas las celdas contienen valores y éstos son
         válidos según las reglas del juego."""
        for i in range(9):
            for j in range(9):
                if not self.__tablero[i][j].valida():
                    return False
        return True

    def opciones(self, fila, columna):
        """Método que devuelve las opciones que ha dejado el jugador
         guardadas en la celda ubicada en la posición (fila,columna)
         del tablero."""
        if 0 < fila < 10 and 0 < columna < 10:
            return self.__tablero[fila-1][columna-1].opciones()

    def nueva_opcion(self, fila, columna, valor):
        """Método que guarda un nuevo valor en las opciones de la celda
         ubicada en la posición (fila,columna) del tablero. Si el dato
         ya existe en las opciones de la celda, entonces devuelve False
         sino devuelve True."""
        if not self.__tablero[fila-1][columna-1].valida() and\
        self.es_factible(fila, columna, valor) and valor not in\
         self.__tablero[fila-1][columna-1].opciones():
            self.__jugadas.append([(fila-1, columna-1,copy.deepcopy\
            (self.__tablero[fila-1][columna-1]))])
            return self.__tablero[fila-1][columna-1].nueva_opcion(valor)
        return False

    def borrar_opcion(self, fila, columna, valor):
        """Método que borra el valor indicado de las opciones de la
        celda ubicada en la posición (fila,columna) del tablero. Si el
        dato no existe en las opciones de la celda, entonces devuelve
        False sino devuelve True."""
        if 0 < fila < 10 and 0 < columna < 10 and valor in\
         self.__tablero[fila-1][columna-1].opciones():
            self.__jugadas.append([(fila-1, columna-1,copy.deepcopy\
            (self.__tablero[fila-1][columna-1]))])
            return self.__tablero[fila-1][columna-1].borrar_opcion\
            (valor)
```

```
        return False

    def es_factible(self, fila, columna, valor):
        """Método que devuelve True si el valor dado puede ubicarse en
        la celda ubicada en la posición (fila,columna) del tablero según
        las reglas del juego."""
        if 0 < fila < 10 and 0 < columna < 10 and 0 < valor < 10 and\
        not self.__tablero[fila-1][columna-1].fija():
            for i in range(9):
                if self.__tablero[fila-1][i].valor() == valor or\
                self.__tablero[i][columna-1].valor() == valor:
                    return False
            f = 3 * ((fila-1)/3)
            c = 3 * ((columna-1)/3)
            for i in range(f,f+3):
                for j in range(c,c+3):
                    if i!= fila-1 and j!= columna -1 and\
                    self.__tablero[i][j].valor() == valor:
                        return False
            return True
        return False

    def undo(self):
        """Método que deshace la ultima jugada. En el caso en que no
        haya más jugadas para deshacer la función devuelve False sino
        devuelve True."""
        if self.__jugadas != []:
            jugada = self.__jugadas.pop()
            for tupla in jugada:
                self.__tablero[tupla[0]][tupla[1]] = tupla[2]
            return True
        return False

    def __str__(self):
        """Método llamado para crear una cadena de texto que represente
    el tablero del sudoku."""
        sudoku = 10*" "+ "Tablero" + 10* " "+"Opciones\n"
        for i in range(9):
            linea = str(i+1)
            for j in range(9):
                if j%3 == 0:
                    linea+= " "
                linea+= "|"+str(self.__tablero[i][j])
                if j==2 or j==5 or j==8:
                    linea+="|"
```

```
            linea+=MENU[i]
            if((i+1)%3 == 0):
                linea+="\n"
            linea+="\n"
            sudoku+=linea
            linea = str(i+1)
        sudoku+= "   1 2 3   4 5 6   7 8 9\n"
        return sudoku

def Entrada(devolver_valor=True):
    """Función que facilita la lectura de los parámetros fila, columna y
     valor (este último opcional) desde el teclado."""
    fila=columna=valor=0
    while fila<1 or fila>9:
        try: fila = int(raw_input\
            ("Introduce el número de fila (entre 1 y 9): "))
        except: continue
    while columna<1 or columna>9:
        try: columna = int(raw_input\
            ("Introduce el número de columna (entre 1 y 9): "))
        except: continue
    while devolver_valor and valor<1 or valor>9:
        try: valor = int(raw_input\
        ("Introduce el valor (entre 1 y 9): "))
        except: continue
    if devolver_valor: return fila, columna, valor
    else: return fila, columna

def main():
    sudoku = Sudoku()
    opcion = ""
    os.system("clear")
    mensaje = "Sudoku en Python: \n"\
    +"Interfaz de juego desarrollada por César Aguilera,\n"\
    +"Tablero generado gracias al"\
    + """ "Sudoku Generator and Solver" """\
    + "de David Bau.\n"""
    while(not sudoku.finalizado() and opcion!= "h"):
        print mensaje
        print sudoku
        opcion=raw_input("\nIntroduce una opción: ")
        if(opcion=="a"):
            fila, columna , valor = Entrada()
            if sudoku.fijar_valor(fila, columna, valor):
                mensaje = "Valor de la celda ("+ str(fila) + ","\
```

```
                + str(columna) + ") fijado correctamente."
            else: mensaje = "Imposible fijar valor. Revise la jugada."
        elif opcion =="b":
            fila, columna = Entrada(False)
            if sudoku.borrar_valor(fila,columna):
                mensaje = "Valor de la celda ("+ str(fila) + ","\
                + str(columna) + ") borrado correctamente."
            else: mensaje = "Imposible borrar valor. Revise la jugada."
        elif opcion == "c":
            fila, columna, valor = Entrada()
            if sudoku.nueva_opcion(fila,columna, valor):
                mensaje = "Opción de la celda ("+ str(fila) + ","\
                + str(columna) + ") agregada correctamente."
            else: mensaje = "Imposible agregar opción. Revise la jugada."
        elif opcion == "d":
            fila, columna, valor = Entrada()
            if sudoku.borrar_opcion(fila, columna, valor):
                mensaje = "Opción de la celda ("+ str(fila) + ","\
                + str(columna) + ") borrada correctamente."
            else: mensaje = "Imposible borrar opción. Revise la jugada."
        elif opcion == "e":
            fila, columna, valor = Entrada()
            if sudoku.es_factible(fila,columna,valor):
                mensaje = "SI es factible introducir el valor "\
                + str(valor) +" en la celda ("+ str(fila) + ","\
                + str(columna) +")."
            else: mensaje = "NO es factible introducir el valor "\
                + str(valor) + " en la celda ("+ str(fila) + ","\
                + str(columna) + ")."
        elif opcion == "f":
            if sudoku.undo(): mensaje ="Jugada deshecha."
            else: mensaje = "Imposible deshacer la jugada."
        elif opcion == "g":
            fila, columna = Entrada(False)
            if sudoku.opciones(fila,columna) != []:
                 mensaje ="La celda (" + str(fila) + ","\
                + str(columna) + ") tiene como opciones: "\
                + str(sudoku.opciones(fila,columna))
            else: mensaje="La celda (" + str(fila) + ","\
                + str(columna) + ") no tiene ninguna opción."
        else: mensaje = "Introduzca una opción correcta."
        os.system("clear")
    if opcion == "h": print "¡Adiós!"
    else: print "Sudoku completado. Enhorabuena."
    return 0
```

```
if __name__ == '__main__':
    main()
```

SIMPLE BACKTRACKING ALGORITHM

El retroceso es una técnica algorítmica para resolver problemas de forma recursiva al tratar de construir una solución de forma incremental, una pieza a la vez, eliminando aquellas soluciones que no satisfacen las limitaciones del problema en cualquier momento (por tiempo, aquí, se hace referencia a el tiempo transcurrido hasta alcanzar cualquier nivel del árbol de búsqueda).

Por ejemplo, considere el problema de resolución de SudoKo, intentamos completar los dígitos uno por uno. Siempre que encontremos que el dígito actual no puede llevar a una solución, lo eliminamos (retrocedemos) e intentamos el siguiente dígito. Esto es mejor que un enfoque ingenuo (generar todas las combinaciones posibles de dígitos y luego probar cada combinación una por una), ya que elimina un conjunto de permutaciones cada vez que retrocede.

3		6	5		8	4		
5	2							
	8	7					3	1
		3		1			8	
9			8	6	3			5
	5			9		6		
1	3					2	5	
							7	4
		5	2		6	3		

El retroceso es una forma de recursividad. Pero implica elegir la única opción entre todas las posibilidades. Comenzamos eligiendo una opción y retrocedemos de ella, si llegamos a un estado en el que concluimos que esta opción específica no da la solución requerida. Repetimos estos pasos recorriendo cada opción disponible hasta que obtengamos la solución deseada.

A continuación se muestra un ejemplo de cómo encontrar todo el orden posible de arreglos de un conjunto de letras dado. Cuando elegimos un par, aplicamos retroceso para verificar si ese par exacto ya se ha creado o no. Si aún no se ha creado, el par se agrega a la lista de respuestas; de lo contrario, se ignora.

```
def permute(list, s):
    if list == 1:
        return s
    else:
        return [ y + x
                 for y in permute(1, s)
                 for x in permute(list - 1, s)
                 ]

print(permute(1, ["a","b","c"]))
print(permute(2, ["a","b","c"]))
```

Cuando se ejecuta el código anterior, produce el siguiente resultado:

```
['a', 'b', 'c']
['aa', 'ab', 'ac', 'ba', 'bb', 'bc', 'ca', 'cb', 'cc']
```

EJEMPLO

SBA.py

```
def permutation(list, start, end):
    '''This prints all the permutations of a given list
       it takes the list,the starting and ending indices as input'''
    if (start == end):
        print list
    else:
        for i in range(start, end + 1):
            list[start], list[i] = list[i], list[start]  # The swapping
            permutation(list, start + 1, end)
            list[start], list[i] = list[i], list[start]  # Backtracking
permutation([1, 2, 3], 0, 2)  # The first index of a list is zero
```

SALIDA

```
[1, 2, 3]
[1, 3, 2]
[2, 1, 3]
[2, 3, 1]
[3, 2, 1]
[3, 1, 2]
```