Universidad Nacional Autónoma de México

Facultad de ingeniería

Documentación – Código
Entregable

Proyecto: Origin Space

Realizado por:

- José Ángel Olmedo Guevara

Objetivo:

Implementación de un algoritmo de búsqueda que pueda ser usado en la planificación de rutas

Introducción:

La navegación autónoma se define como la capacidad de un sistema (robot, automóvil, dron) capaz de moverse en un ambiente (conocido o no) sin la intervención humana, esto a fin de reconocer un terreno inexplorado o bien llegar a un objetivo previamente establecido. Para poder hacer de la navegación autónoma una realidad, implementamos algoritmos de búsqueda, en este caso propondremos dos algoritmos, A* y búsqueda primera en profundidad (DFS). Cabe mencionar que emplearemos una estructura de datos conocida como grafos para simular una ruta conocida.

DFS: Su principal enfoque radica en explorar a profundidad una rama tanto como sea posible antes de retroceder, a continuación, una explicación más detallada:

1. Partimos de un nodo inicial de nuestro grafo
2. Exploración profunda: Seleccionamos una rama y profundizamos en ella, para cada nodo:
   2.1 Visitamos el nodo, lo marcamos como visitado
   2.2 Explorar vecinos no visitados: Seleccionamos un vecino no visitado de nuestro nodo actual y repetimos el proceso recursivamente.
   2.3 Retroceso: Cuando el nodo actual no tiene más vecinos no visitados, el algoritmo retrocede a la siguiente rama para explorar otros caminos.
3. El algoritmo finaliza cuando se haya encontrado el nodo objetivo.

A*: Es bastante utilizado para encontrar el camino más corto entre un nodo inicial y un nodo final u objetivo, específicamente para grafos ponderados. La condiciones del algoritmo A* establecen que siempre se debe de buscar el camino con el menor costo, para ello se emplea la siguiente relación matemática: $f(n)=g(n) + h(n)$.

g(n) representa el costo real del camino desde el nodo inicial hasta el nodo objetivo.

h(n) es la heurística, debe de ser admisible, es decir que el costo no debe ser mayor al camino real g(n). Este algoritmo siempre devolverá el camino de menor costo.

Componentes de A:*

- Función de evaluación: $f(n)=g(n)+h(n)$:
  - f(n): Costo total estimado desde el nodo inicial hasta el nodo destino pasando por el nodo presente n.
  - g(n): Costo real desde el nodo inicial hasta el nodo destino pasando por el nodo presente n.
  - h(n): Heurística, no debe de sobreestimar el costo real del camino para que el algoritmo encuentre la solución óptima

- Lista abierta y cerrada:
    - Lista abierta: Contiene todos los nodos descubiertos pero que aún no han sido explorados, en un inicio solo tiene el nodo de partida.
    - Lista cerrada: Contiene los nodos explorados y evaluados.

Pasos de A:*

1. Agregar el nodo de inicio a la lista abierta con g (costo real) igual a 0 y una heurística h estimada desde el nodo inicial hasta el nodo destino.

2. Inicializar la lista cerrada como vacía.

3. While lista abierta != [void]:
    - Encontrar el nodo n en la lista abierta con menor costo total f(n).
    - Si n es el nodo destino, la solución ha sido encontrada y debemos reconstruir el camino desde el nodo de inicio hasta el nodo destino.
    - Mover n (nodo presente) de la lista abierta a la cerrada.
    - Para cada nodo vecino m de n:
        - Si m está en la lista cerrada, ignorar (pass).
        - Si m no está en la lista abierta, agregar m a la lista abierta y establecer su costo real (g) y heurística (h) correspondiente.
        - Si m ya está en la lista abierta y la nueva ruta de n hasta m tiene menor costo real (g), actualizar g y el nodo padre de m.

4. Si la lista abierta está vacía (ningún nodo por recorrer) y el nodo objetivo no ha sido alcanzado no existe solución.

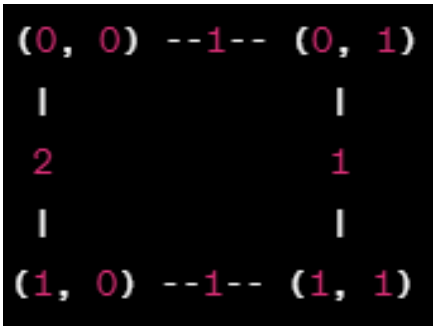
Resultados:

```python
import heapq #Importamos biblioteca para manejar colas (estructura de dato)

def a_estrella_busqueda(grafo, inicio, destino):
  '''Función que permite realizar una búsqueda dentro de un grafo  dado un
  nodo de inicio y uno de destino empleando el algoritmo A*'''
  no_explorado = []  # Creamos una lista de los nodos que aún no han sido explorados
  explorados = set()  # Creamos un conjunto de de los nodos ya visitados
  camino = {}  # Diccionario que nos dirá los nodos que recorrimos

    # Iniciando el recorrio con el primer nodo
  heapq.heappush(no_explorado, (0, inicio)) #Agregar un nodo a la cola, la tupla representa '0' el costo de inicio y 'start' el nodo de inicio.
  camino[inicio] = None #El diccionario en un inicio no tendrá un camino recorrido por lo que lo pondremos vacío con el nodo de inicio.
  distancia = {nodo: float('inf') for nodo in grafo}
  '''Generamos un diccionario en donde todas las claves sean todos los nodos existentes, incializamos todos
  los valores en infinito, posteriormente lo modificaremos dependiendo de las distancias de cada nodo.'''

  distancia[inicio] = 0 #La distancia del nodo de inicio será cero.

  while no_explorado: #Ciclo para recorrer los nodos sin explorar
      costo_actual, nodo_actual = heapq.heappop(no_explorado)
      '''De mi estructura de datos, extraemos el primer nodo no explorado,
      extraemos el coste actual y cual es ese nodo'''

      if nodo_actual == destino: #Si el nodo actual es el nodo al que queremos llegar
          return formar_camino(camino, inicio, destino) #Reconstruimos el camino FIN DEL CICLO

      explorados.add(nodo_actual) #Agregamos al conjunto de nodos explorados al nodo actual
```

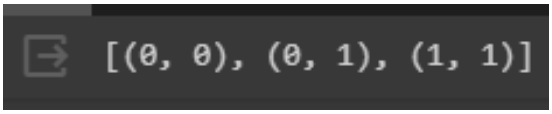
```python
      for nodo_vecino, coste in grafo[nodo_actual].items(): #Ciclo para iterar en cada uno de los nodos vecinos y su coste en el grafo
          if nodo_vecino in explorados: #Si el nodo vecino se encuentra en la lista de nodos  ya explorados
              continue #Decidimos continuar

          distancia_tentativa = distancia[nodo_actual] + coste #Calculamos la distancia acumulada desde el nodo de inicio hasta el nodo actual.

          if distancia_tentativa < distancia[nodo_vecino]:
            '''Verifica si la distancia tentativa es menor que la distancia acumulada desde el nodo de inicio hasta el nodo vecino'''
            distancia[nodo_vecino] = distancia_tentativa
            '''Si la distancia tentativa es menor, actualiza la distancia acumulada conocida para el vecino'''
            funcion_a_estrella = distancia_tentativa + heuristica(nodo_vecino, destino)
            '''Obtenemos f(x) para sumando la distancia acumulada tentativa y una heurística que estima la distancia restante hasta el destino'''
            heapq.heappush(no_explorado, (funcion_a_estrella, nodo_vecino))
            '''Agrega el otro vecino que aún no ha sido explorado a la cola (lista) con la nuevo coste calculado (función_a_estrella)'''
            camino[nodo_vecino] = nodo_actual #Actualizamos el camino recorrido dado el nodo actual

  return None  # Si nada de lo anterior funciona, significa que no se encontró ruta

def formar_camino(camino, nodo_inicial, destino): #Función para devolver el camino de los nodos que visitamos para llegar al destino.
  '''Toma como argumentos el diccionario de los nodos recorridos, el nodo de inicio y el nodo destino'''
  nodo_actual = destino #Inicializamos el nodo actual como el nodo destino (ya llegamos al destino)
  ruta = [] #Lista vacía que almacenará los nodos que forman el camino.

  while nodo_actual is not None:  #Ciclo que iterará hasta que el nodo actual este vacío
    ruta.insert(0, nodo_actual) #Insertamos en la primera posición de la lista el nodo actual.
    nodo_actual = camino[nodo_actual]
    ''' Actualiza el nodo actual con el valor almacenado en el diccionario 'camino' para el nodo actual y así recorreremos los nodos visitados'''

  return ruta #Regresamos la ruta
```

```python
def heuristica(nodo, destino):
  #La heurística en este caso será la distancia auclidana entre las posiciones de ambos nodos.
  return abs(nodo[0] - destino[0]) + abs(nodo[1] - destino [1])

# Ejemplo de uso
grafo = { #Este grafo es un diccionario en donde cada clave es el grafo y los valores son los los nodos vecinos con sus respectivos pesos
    (0, 0): {(0, 1): 1, (1, 0): 2},
    (0, 1): {(0, 0): 1, (1, 1): 1},
    (1, 0): {(0, 0): 1, (1, 1): 1},
    (1, 1): {(0, 1): 1, (1, 0): 1}
}

inicio = (0, 0) #El nodo de inicio será el (0,0)
destino = (1, 1) #El nodo de destino será el (1,1)

prueba = a_estrella_busqueda(grafo, inicio, destino) #Probamos el algoritmo
print(prueba) #Imprimimos la prueba
```

Visualización del grafo:



Camino:

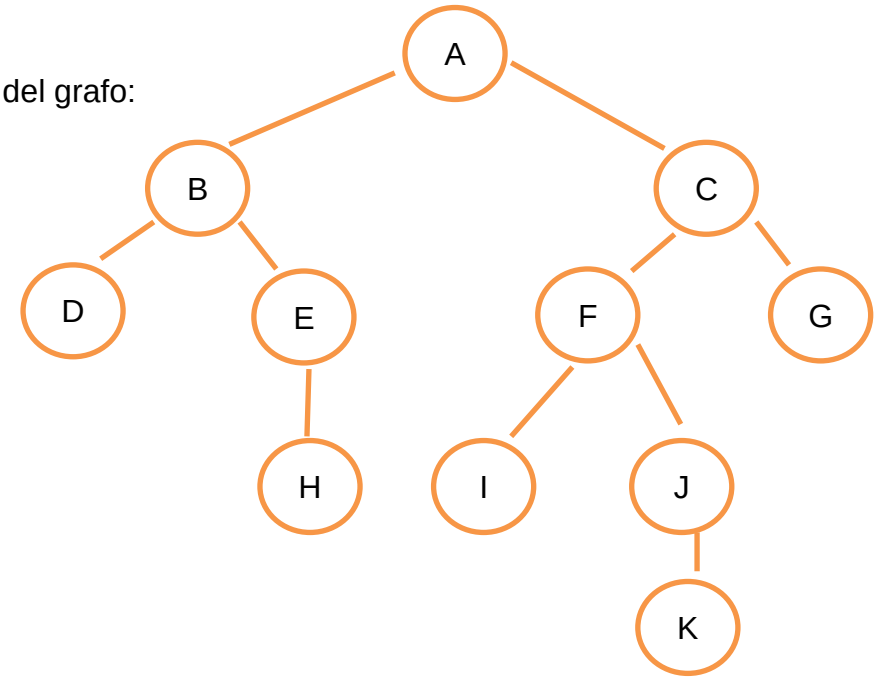


DFS

```python
def dfs(grafo, inicio, destino): #Función para implementar el algoritmo DFS
  pila = [(inicio, [inicio])]  # Pila de nodos por visitar y camino seguido
  '''Haremos uso de una estructura de datos conocida como pila, recibe tuplas,
  el primer elemento contendra un nodo y el segundo será el camino hacia ese nodo. '''
  visitados = set()  # Conjunto de nodos visitados, no se pueded modificar.

  while pila: #Bucle para iterar mientras haya nodos en la pila.
    (nodo_actual, camino) = pila.pop() #Desapilamos el nodo actual y el camino hasta a ese nodo

    if nodo_actual in visitados: #Si ese nodo ya ha sido visitado
      continue #Saltamos a la siguiente iteración

    visitados.add(nodo_actual) #Sino, añadimos al conjunto de nodos visitados el nodo actual

    if nodo_actual == destino: #Si el nodo actual es el nodo objetivo al que queremos llegar
      return camino #Regresaremos el camino, es decir, todos los nodos por los que pasamos (claves del diccionario)

    '''Aquí comienza la exploración de mis nodos vecinos'''
    for nodo_vecino in grafo[nodo_actual]: #Para cada nodo vecino (valor) de mi grafo en el nodo actual
      '''Ciclo para recorrer mis nodos vecinos'''
      pila.append((nodo_vecino, camino + [nodo_vecino]))
      #Añadiremos a la pila, el nodo vecino y camino para llegar a él desde mi nodo actual

  return None  #Sino hay alguna ruta no devolveremos nada.
```

```python
'''El grafo será un diccionario en donde las claves son el nodo y los valores los nodos vecinos'''
grafo = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F', 'G'],
    'D': ['B'],
    'E': ['B', 'H'],
    'F': ['C', 'I', 'J'],
    'G': ['C'],
    'H': ['E'],
    'J': ['K']
}

nodo_de_inicio = 'A' #Nodo de inicio presente
nodo_objetivo = 'K' #Nodo objetivo o final

resultado = dfs(grafo, nodo_de_inicio, nodo_objetivo) #Probamos el algoritmo
print(resultado) #Mostramos el resultado
```

Visualización del grafo:



Camino:

```
['A', 'C', 'F', 'J', 'K']
```

Conclusiones

El objetivo propuesto fue cumplido de manera satisfactoria ya que implementamos exitosamente no solo uno, sino dos algoritmos de búsqueda que pueden ser utilizados en entornos reales dado un ambiente conocido. El algoritmo A* puede ser un poco más útil ya que encuentra el camino con menor heurística y peso, eso es fundamental ya que se puede ver a el camino con menor gasto de energía y no necesariamente el más corto, ya que el robot Rover puede enfrentarse a superficies irregulares la cual le hagan gastar mayor energía y esto puede ser contraproducente, pues se puede quedar sin batería a mitad del camino o bien, quedar atascado en una superficie irregular, por otra parte el algoritmo DFS puede ser más útil si lo que se desea es explorar un terreno nuevo o explorar y llegar a un objetivo, siempre y cuando no sea un grafo ponderado.

Bibliografía:

- Rusell, S., & Norvig, P. (2016). Artificial Intelligence: A modern aproach (3.a ed.). Pearson.
- https://www.ecured.cu/Algoritmo_de_B%C3%BAsqueda_Heur%C3%ADstica_A*
- Coto, E. (2003). Algoritmos básicos de grafos. Lecturas en Ciencias de computación. ISSN 1316, 6239.