Algoritmo "Primero el Mejor" (Best-First Search)

Descripción Teórica:

El algoritmo de búsqueda "Primero el Mejor" (Best-First Search, BFS) es una técnica de búsqueda informada que apunta a un enfoque heurístico para explorar grafos o árboles con el objetivo de encontrar una ruta óptima desde un nodo inicial hasta un nodo objetivo. Utiliza una función heurística que asigna un valor a cada nodo para estimar la distancia o costo restante hasta el objetivo. La idea principal es expandir siempre el nodo que parece más prometedor según funciones de evaluación.

Las funciones de evaluación son reglas específicas que adopta el modelo para buscar el mejor camino [f(n)]. Es una función de evaluación heurística que depende de la descripción dada a “n”, es decir, es la descripción de la meta a lograr, la información reunida hasta cierto punto de la búsqueda o cualquier conocimiento extra acerca del dominio del problema.

Cuando hablamos de costos dentro del BFS nos referimos al costo asociado a cada paso, priorizando siempre la opción más favorable. Los costos son determinados según el valor heurístico asignado, siendo el de valor más bajo el más prometedor para llegar a la meta del algoritmo.

Pasos del Algoritmo:

1. **Inicialización**:
    - Comienza con un nodo inicial (o conjunto de nodos si se trata de una búsqueda en múltiples puntos de partida).
    - Inserta el nodo inicial en una estructura de datos de tipo prioridad, puede ser una cola de prioridad o un heap, donde la prioridad está determinada por la función heurística.
2. **Exploración**:
    - Extrae el nodo con la menor puntuación heurística de la cola de prioridad.
    - Si este nodo es el objetivo, el algoritmo termina y se ha encontrado la ruta.
    - Si no, se expanden todos sus nodos vecinos (hijos), y cada vecino se añade a la cola de prioridad con su valor heurístico.
3. **Evaluación**:
    - Repite el proceso de extracción y expansión hasta que se encuentre el nodo objetivo o hasta que se agoten los nodos en la cola de prioridad (en cuyo caso no hay ruta).

Complejidad Temporal

La complejidad temporal del algoritmo depende principalmente de dos factores:

1. **Número de Nodos N**:
    - En el peor de los casos, el algoritmo podría visitar todos los nodos del grafo, especialmente si la heurística no es efectiva.
    - Cada nodo puede ser procesado una vez, lo que implica un costo básico de O(N) **[O(n) es el Tiempo Lineal (en proporción al número de elementos)]**
2. **Operaciones de la Cola de Prioridad**:
    - Usamos una cola de prioridad para seleccionar el siguiente nodo a explorar basado en la heurística.
    - Insertar y extraer elementos de la cola de prioridad (implementada con un heap) tiene un costo de O(logN) por operación. **[O(logN) es el Tiempo Logarítmico (tiempo de ejecución en proporción al tamaño de entrada del logaritmo)]**
    - Como cada nodo puede ser añadido y removido de la cola de prioridad una vez, el costo total es O(NlogN) **[O(NlogN) es el Tiempo Linealítmico (las operaciones de logN que ocurrirán "n" veces)]**
3. **Expansión de Nodos**:
    - Revisar los vecinos de cada nodo expandido tiene un costo total de O(E), donde E es el número de aristas.

Complejidad Temporal Total: O(NlogN+E)

Complejidad Espacial

La complejidad espacial está influenciada por:

1. **Cola de Prioridad**:
    - Puede contener hasta N nodos en el peor de los casos.
    - Espacio requerido: O(N)
2. **Rastreo del Camino**:
    - Utilizamos un diccionario para mantener el camino desde el nodo inicial hasta el nodo objetivo.
    - Espacio requerido: O(N)
3. **Heurística**:
    - Guardamos una estimación para cada nodo.
    - Espacio requerido: O(N)

Complejidad Espacial Total: O(N)

Ventajas y Desventajas

Ventajas

1. **Eficiencia con Buenas Heurísticas**:
    - Si la heurística es efectiva, el algoritmo puede encontrar la solución rápidamente al expandir menos nodos.
2. **Simple de Implementar**:
    - Comparado con algoritmos más complejos, "Primero el Mejor" es relativamente sencillo de implementar y comprender.
3. **Apropiado para Espacios de Estados Grandes**:
    - Puede ser más eficiente en espacios de búsqueda grandes cuando el objetivo está cerca según la heurística.

Desventajas

1. **No Garantiza Optimalidad**:
    - Puede no encontrar la ruta más corta si la heurística no es consistente o admisible, ya que no considera el costo acumulado.
2. **Dependencia de la Heurística**:
    - Su eficiencia depende en gran medida de la calidad de la heurística. Una heurística pobre puede llevar a la exploración innecesaria de muchos nodos.
3. **Posible Consumo Alto de Memoria**:
    - En grafos grandes, la cola de prioridad y las estructuras de rastreo pueden consumir una cantidad significativa de memoria.
4. **No Asegura Completitud**:
    - En algunos casos, puede no encontrar una solución aunque exista, especialmente si la heurística no es adecuada o hay ciclos en el grafo.

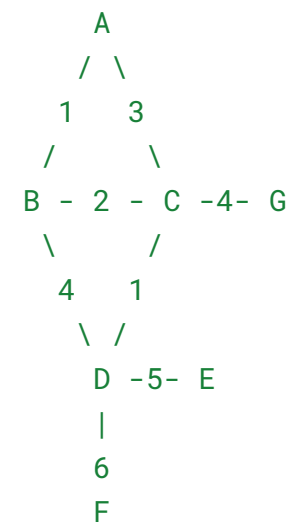
Aplicaciones:

- Navegación en mapas y juegos.
- Búsqueda de caminos en robots.
- Resolución de puzzles y problemas de optimización.

Ejemplo: Encontrar el Camino en un Grafo Simple

Supongamos que tenemos un grafo pequeño representando ciudades conectadas por caminos, y queremos encontrar la ruta desde la ciudad **A** hasta la ciudad **G**.

Este sería el grafo con nodos (ciudades) y aristas (caminos con costos asociados):



- **Nodos**: A, B, C, D, E, F, G
- **Aristas y Costos**:
    - A-B: 1
    - A-C: 3
    - B-C: 2
    - B-D: 4
    - C-D: 1
    - C-G: 4
    - D-E: 5
    - D-F: 6

Heurística

Para este ejemplo, usaremos una heurística simple que representa una estimación de la distancia en línea recta desde cada nodo hasta el nodo objetivo **G**. Esta heurística es:

- A: 5
- B: 4
- C: 2
- D: 6
- E: 3
- F: 7
- G: 0

La heurística debe ser menor o igual al costo real para asegurar que guíe la búsqueda de manera efectiva hacia el objetivo.

Ejecución Paso a Paso

Paso 1: Inicialización

- Cola de prioridad: (5,′A′)(5, 'A')(5,′A′)
- Nodo actual: **A**
- Caminos explorados: []

Paso 2: Exploración del Nodo A

- Extraemos **A** de la cola (heurística 5).
- Expansión de A:
    - Vecino B con costo 1, heurística 4.
    - Vecino C con costo 3, heurística 2.
- Añadimos B y C a la cola:
    - Cola de prioridad: (4,′B′),(2,′C′)(4, 'B'), (2, 'C')(4,′B′),(2,′C′)
- Caminos explorados: [A]

Paso 3: Exploración del Nodo C

- Extraemos **C** de la cola (heurística 2).
- Expansión de C:
    - Vecino B con costo 2, heurística 4 (ya en la cola, no lo añadimos de nuevo).
    - Vecino D con costo 1, heurística 6.
    - Vecino G con costo 4, heurística 0.
- Añadimos D y G a la cola:
    - Cola de prioridad: (4,′B′),(6,′D′),(0,′G′)(4, 'B'), (6, 'D'), (0, 'G')(4,′B′),(6,′D′),(0,′G′)
- Caminos explorados: [A, C]

Paso 4: Exploración del Nodo G

- Extraemos **G** de la cola (heurística 0).
- **G** es nuestro nodo objetivo, por lo que hemos encontrado el camino.
- Camino encontrado: A -> C -> G
- Caminos explorados: [A, C, G]

Implementación en Python

Para ilustrar este proceso en código, la implementación es la siguiente:

```
import heapq

# Definir el grafo
graph = {
    'A': [('B', 1), ('C', 3)],
    'B': [('A', 1), ('C', 2), ('D', 4)],
    'C': [('A', 3), ('B', 2), ('D', 1), ('G', 4)],
    'D': [('B', 4), ('C', 1), ('E', 5), ('F', 6)],
    'E': [('D', 5)],
    'F': [('D', 6)],
    'G': [('C', 4)]
}

# Definir la heurística
heuristic = {
    'A': 5,
    'B': 4,
    'C': 2,
    'D': 6,
    'E': 3,
    'F': 7,
    'G': 0
}

def best_first_search(graph, start, goal, heuristic):
    # Cola de prioridad
    priority_queue = []
    heapq.heappush(priority_queue, (heuristic[start], start))

    # Conjunto de nodos explorados
    explored = set()

    # Diccionario para rastrear el camino
    came_from = {start: None}

    while priority_queue:
        # Extraer el nodo con la menor heurística
        _, current = heapq.heappop(priority_queue)

        if current == goal:
            # Reconstruir el camino
```

```
            path = []
            while current is not None:
                path.append(current)
                current = came_from[current]
            return path[::-1]  # Devolver el camino en el orden
correcto

        explored.add(current)

        # Expandir los vecinos
        for neighbor, cost in graph[current]:
            if neighbor not in explored:
                heapq.heappush(priority_queue, (heuristic[neighbor],
neighbor))
                if neighbor not in came_from:
                    came_from[neighbor] = current

    return None  # Si no se encuentra un camino

# Ejemplo de uso
start = 'A'
goal = 'G'
path = best_first_search(graph, start, goal, heuristic)
print(f"Ruta desde {start} hasta {goal}: {path}")
```

Explicación del Código

1. **Definición del Grafo y Heurística**:
   - `graph`: Diccionario que define las conexiones entre nodos y sus costos.
   - `heuristic`: Diccionario que asigna una heurística (estimación de la distancia) a cada nodo.
2. **Función `best_first_search`**:
   - Inicializa una cola de prioridad con el nodo inicial.
   - Utiliza un conjunto `explored` para rastrear los nodos ya visitados.
   - Utiliza un diccionario `came_from` para rastrear el camino desde el inicio hasta el objetivo.
3. **Bucle Principal**:
   - Extrae el nodo con la menor heurística de la cola de prioridad.
   - Si es el nodo objetivo, reconstruye y devuelve el camino.
   - Si no, expande los vecinos y los añade a la cola de prioridad si no han sido explorados.

4. **Resultados**:
    - La función devuelve la ruta desde el nodo inicial hasta el nodo objetivo.

Resultados Obtenidos

Ejecutando el código, obtenemos la ruta desde **A** hasta **G** basada en la heurística.