Universidad del País Vasco

Euskal Herriko Unibertsitatea

Técnicas de Inteligencia Artificial

Ingeniería Informática de Gestión y Sistemas de Información

Practica 1
Problemas de Búsqueda

Autor(es):
Xabier Gabiña

30 de septiembre de 2024

Índice general

1. Introducción

En el marco de la asignatura de Técnicas de Inteligencia Artificial, se nos ha propuesto implementar y analizar diversos algoritmos de búsqueda aplicados al contexto de un proyecto académico desarrollado por la Universidad de Berkeley, basado en el clásico juego Pacman. El objetivo principal de esta práctica es profundizar en el funcionamiento de diferentes estrategias de búsqueda, estudiando su eficiencia y comportamiento en diferentes escenarios.

Los algoritmos de búsqueda son fundamentales en el campo de la inteligencia artificial, ya que permiten encontrar soluciones óptimas o satisfactorias en problemas complejos. En esta práctica, nos enfocaremos en tres tipos de algoritmos de búsqueda no informados: Depth First Search (DFS), Breadth First Search (BFS) y Uniform Cost Search (UCS). Además, exploraremos un algoritmo de búsqueda informado: A*. Cada uno de estos algoritmos tiene sus propias características y aplicaciones, y su estudio nos permitirá comprender mejor sus ventajas y limitaciones.

A lo largo de este documento, se presentarán las implementaciones de cada uno de estos algoritmos, junto con una descripción detallada de su funcionamiento y análisis de su rendimiento. Se incluirán ejemplos prácticos y se discutirán los resultados obtenidos en diferentes escenarios de búsqueda. El objetivo es proporcionar una visión completa y comprensiva de cómo estos algoritmos pueden ser aplicados en la resolución de problemas de búsqueda en inteligencia artificial.

2. Algoritmos no informados

2.1. DFS - Depth First Search

Descripción

DFS o Depth First Search es un algoritmo de búsqueda no informado que se basa en la exploración de todos los nodos de un grafo siguiendo una rama hasta llegar a un nodo hoja, para después retroceder y explorar otra rama. Este algoritmo se implementa mediante una pila, en la que se van almacenando los nodos a visitar.
Su coste en tiempo es de $O(b^m)$, donde b es el factor de ramificación y m es la profundidad máxima del árbol.
Su coste en espacio es de O(bm), donde b es el factor de ramificación y m es la profundidad máxima del árbol.

Primera implementación

```
def depthFirstSearch(problem):
"""
Implementación del algoritmo de búsqueda en profundidad.

Args:
    problem (SearchProblem): Problema de búsqueda
Returns:
    list: Lista de acciones para llegar al objetivo
"""
  stack = [problem] # Pila para almacenar los nodos a visitar
  visited = set()     # Conjunto para almacenar los nodos visitados
  path = []           # Lista para almacenar el camino al nodo objetivo

  while stack:    # Mientras haya elementos en el stack
      nodo_actual = stack.pop()   # Sacar el último elemento de la pila
      if nodo_actual in visited:  # Si el nodo actual ya ha sido visitado
          continue
      visited.add(nodo_actual)   # Marcar el nodo actual como visitado
      path.append(nodo_actual.contenido)  # Añadir el nodo actual al camino
      if nodo_actual.isGoalState():  # Si el nodo actual es el objetivo
          return path
      for hijo in reversed(nodo_actual.getSuccesor()): # Añadir los hijos del nodo actual a la
    pila
          stack.append(hijo)

```

Listing 2.1: Implementación final del DFS

Implementación Final

```python
def depthFirstSearch(problem):
"""
Implementación del algoritmo de búsqueda en profundidad.

Args:
    problem (SearchProblem): Problema de búsqueda
Returns:
    list: Lista de acciones para llegar al objetivo
"""
stack = util.Stack()  # Añadir el nodo inicial a la pila
stack.push([problem.getStartState(), []])
visited = set()  # Conjunto para almacenar los nodos visitados

while not stack.isEmpty():  # Mientras haya elementos en el stack
    nodo_actual = stack.pop()  # Sacar el último elemento de la pila
    if problem.isGoalState(nodo_actual[0]):  # Si el nodo actual es el objetivo
        return nodo_actual[1]  # Devolver el camino
    if nodo_actual[0] not in visited:
        visited.add(nodo_actual[0])
        for estado, accion, costo in reversed(problem.getSuccessors(nodo_actual[0])):
            camino = nodo_actual[1] + [accion]
            stack.push([estado, camino])

```

Listing 2.2: Implementación final del DFS

Comentarios

2.2. BFS - Breadth First Search

Descripción

BFS o Breath First Search es un algoritmo de búsqueda no informado que se basa en la exploración de todos los nodos de un grafo nivel por nivel. Este algoritmo se implementa mediante una cola en la que se van almacenando los nodos que se deben visitar, manteniendo un orden de llegada basado en los niveles.
Su coste en tiempo es de $O(b^d)$, donde b es el factor de ramificación y d es la profundidad del nodo más cercano del árbol y su coste en espacio es de $O(b^d)$, ya que debe almacenar todos los nodos del nivel actual antes de pasar al siguiente.

Implementación Final

```
def breadthFirstSearch(problem):
"""
Implementacion del algoritmo de busqueda en anchura.

Args:
    problem (SearchProblem): Problema de busqueda
Returns:
    list: Lista de acciones para llegar al objetivo
"""
queue = util.Queue()  # Añadir el nodo inicial a la cola
queue.push([problem.getStartState(), []])
visited = set()     # Conjunto para almacenar los nodos visitados

while not queue.isEmpty():    # Mientras haya elementos en la cola
    nodo_actual = queue.pop()   # Sacar el primer elemento de la cola
    if problem.isGoalState(nodo_actual[0]):  # Si el nodo actual es el objetivo
        return nodo_actual[1]  # Devolver el camino
    if nodo_actual[0] not in visited:
        visited.add(nodo_actual[0])
        for estado, accion, costo in problem.getSuccessors(nodo_actual[0]): # Añadir los hijos del
     nodo actual a la cola
            camino = nodo_actual[1] + [accion]
            queue.push([estado, camino])

```

Listing 2.3: Implementación final del BFS

Comentarios

2.3. UCS - Uniform Cost Search

Descripción

UCS o Uniform Cost Search es un algoritmo de búsqueda no informada que expande los nodos en función del costo acumulado. En contrario al BFS que prioriza la profundidad de los nodos el UCS utiliza la cola de prioridad para ordenar los nodos dependiendo su costo acumulado y expande primero el nodo con menor costo, obteniendo así una solución óptima en términos de costo.
El coste en tiempo del UCS es de $O(b^{1+\frac{C^*}{\varepsilon}})$, donde b es el factor de ramificación, C* es el costo de la solución óptima y ε es el menor costo de transición entre nodos. Su coste de espacio es así mismo $O(b^{1+\frac{C^*}{\varepsilon}})$ ya que debe de almacenar todos los nodos en la cola de prioridad hasta encontrar la solución óptima

Implementación Final

```
def uniformCostSearch(problem):
"""
Implementacion del algoritmo de busqueda de coste uniforme.

Args:
    problem (SearchProblem): Problema de busqueda
Returns:
    list: Lista de acciones para llegar al objetivo
"""
queue = util.PriorityQueue()  # Añadir el nodo inicial a el heap
queue.push([problem.getStartState(), [], 0], 0)
visited = set()     # Conjunto para almacenar los nodos visitados

while not queue.isEmpty():    # Mientras haya elementos en el stack
    nodo_actual = queue.pop()   # Sacar el último elemento de la pila
    if problem.isGoalState(nodo_actual[0]):  # Si el nodo actual es el objetivo
        return nodo_actual[1]  # Devolver el camino
    if nodo_actual[0] not in visited:
        visited.add(nodo_actual[0])
        for estado, accion, costo in problem.getSuccessors(nodo_actual[0]): # Añadir los hijos del
     nodo actual a la pila
            camino = nodo_actual[1] + [accion]
            queue.push([estado, camino, nodo_actual[2] + costo], nodo_actual[2] + costo)

```

Listing 2.4: Implementación final del UCS

Comentarios

3. Algoritmos informados

3.1. A*

Descripción

El A* es un algoritmo de búsqueda informada que utiliza las ventajas del UCS y Greedy Best-First Search, utilizando una función heurística para guiar la búsqueda hacia el objetivo de una manera más eficiente. A* expande los nodos en mediante una función de costo total. Donde f(n) = g(n) + h(n), g(n) siendo el costo acumulado desde el nodo inicial hasta n y h(n) es una estimación heurística del costo n hasta el objetivo. Este algoritmo se implementa mediante una cola de prioridad , donde los nodos se ordenan según su valor f(n) y se expande primero el nodo con menor valor f(n).
Su coste en tiempo es de $O(b^d)$, donde b es el factor de ramificación y d es la profundidad de la solución óptima, sin embargo el tiempo puede reducirse si la heurística es eficiente. Así mismo su coste de espacio es de $O(b^d)$, ya que se debe de almacenar todos los nodos generados en la cola de prioridad para asegurar la solución óptima.

Implementación Final

```
def aStarSearch(problem, heuristic=nullHeuristic):
"""
Implementacion del algoritmo de busqueda A*.

Args:
    problem (SearchProblem): Problema de busqueda
    heuristic (function): Heuristica para el problema
Returns:
    list: Lista de acciones para llegar al objetivo
"""
queue = util.PriorityQueue()  # Añadir el nodo inicial a el heap
queue.push([problem.getStartState(), [], 0], 0)
visited = set()     # Conjunto para almacenar los nodos visitados

while not queue.isEmpty():    # Mientras haya elementos en el stack
    nodo_actual = queue.pop()   # Sacar el último elemento de la pila
    if problem.isGoalState(nodo_actual[0]):  # Si el nodo actual es el objetivo
        return nodo_actual[1]  # Devolver el camino
    if nodo_actual[0] not in visited:
        visited.add(nodo_actual[0])
        for estado, accion, costo in problem.getSuccessors(nodo_actual[0]): # Añadir los hijos del
     nodo actual a la pila
            camino = nodo_actual[1] + [accion]
            queue.push([estado, camino, nodo_actual[2] + costo], nodo_actual[2] + costo +
    heuristic(estado, problem))

```

Listing 3.1: Implementación final del A*

Comentarios