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
Diego Montoya

4 de octubre 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. Ejercicios

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 Breadth 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

2.4. A* Search

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 manhattanHeuristic(state, problem):
    """
    Heuristica de Manhattan.

    Args:
        state (tuple): Coordenadas del estado
        problem (SearchProblem): Problema de busqueda
    Returns:
        int: Distancia de Manhattan al objetivo
    """
    return util.manhattanDistance(state, problem.goal)

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 2.5: Implementación final del A*

Comentarios

2.5. Corners Problem: Representación

Descripción

Primera implementación

```
class CornersProblem(search.SearchProblem):
  """
  This search problem finds paths through all four corners of a layout.

  You must select a suitable state space and successor function
  """

  def __init__(self, startingGameState):
      """
      Stores the walls, pacman's starting position and corners.
      """
      self.walls = startingGameState.getWalls()
      self.startingPosition = startingGameState.getPacmanPosition()
      top, right = self.walls.height - 2, self.walls.width - 2
      self.corners = ((1, 1), (1, top), (right, 1), (right, top))
      for corner in self.corners:
          if not startingGameState.hasFood(*corner):
              print('Warning: no food in corner ' + str(corner))
      self._expanded = 0  # DO NOT CHANGE; Number of search nodes expanded
      # Please add any code here which you would like to use
      # in initializing the problem

  def getStartState(self):
      return self.startingPosition

  def isGoalState(self, state):
      """
      Returns whether this search state is a goal state of the problem.
      """
      if state in self.corners:
          self.explored.add(state)
      if len(self.explored) == 4:
          return True
      return False

  def getSuccessors(self, state):
      """
      Returns successor states, the actions they require, and a cost of 1.

        As noted in search.py:
          For a given state, this should return a list of triples, (successor,
          action, stepCost), where 'successor' is a successor to the current
          state, 'action' is the action required to get there, and 'stepCost'
          is the incremental cost of expanding to that successor
      """

      successors = []
      for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
          x, y = state
          dx, dy = Actions.directionToVector(action)
          nextx, nexty = int(x + dx), int(y + dy)
          if not self.walls[nextx][nexty]:
              nextState = (nextx, nexty)
              cost = self.costFn(nextState)
              successors.append((nextState, action, cost))

      self._expanded += 1  # DO NOT CHANGE
      return successors

  def getCostOfActions(self, actions):
      """
      Returns the cost of a particular sequence of actions.  If those actions
      include an illegal move, return 999999.  This is implemented for you.
      """
```

```
        if actions is None: return 999999
        x, y = self.startingPosition
        for action in actions:
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]: return 999999
        return len(actions)

```

Listing 2.6: Implementación inicial del problema de las esquinas

Implementación Final

```
class CornersProblem(search.SearchProblem):
  """
  This search problem finds paths through all four corners of a layout.

  You must select a suitable state space and successor function
  """

  def __init__(self, startingGameState):
      """
      Stores the walls, pacman's starting position and corners.
      """
      self.walls = startingGameState.getWalls()
      self.startingPosition = startingGameState.getPacmanPosition()
      top, right = self.walls.height - 2, self.walls.width - 2
      self.corners = ((1, 1), (1, top), (right, 1), (right, top))
      for corner in self.corners:
          if not startingGameState.hasFood(*corner):
              print('Warning: no food in corner ' + str(corner))
      self._expanded = 0  # DO NOT CHANGE; Number of search nodes expanded
      # Please add any code here which you would like to use
      # in initializing the problem

  def getStartState(self):
      """
      Returns the start state, including the initial position and the visited corners.
      """
      # The start state now includes an empty set of visited corners
      return (self.startingPosition, ())

  def isGoalState(self, state):
      """
      Returns whether this search state is a goal state of the problem.
      """
      # Unpack the state
      position, visitedCorners = state

      # Check if we have visited all four corners
      return len(visitedCorners) == 4

  def getSuccessors(self, state):
      """
      Returns successor states, the actions they require, and a cost of 1.
      """
      successors = []
      # Unpack the current state
      currentPosition, visitedCorners = state

      for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
          x, y = currentPosition
          dx, dy = Actions.directionToVector(action)
          nextx, nexty = int(x + dx), int(y + dy)

          # Check if the next position is a wall
          if not self.walls[nextx][nexty]:
              nextPosition = (nextx, nexty)
              # Check if we have reached a new corner
```

```
57                 newVisitedCorners = list(visitedCorners)
58                 if nextPosition in self.corners and nextPosition not in visitedCorners:
59                     newVisitedCorners.append(nextPosition)
60
61                 # Create a new state with updated corner list
62                 nextState = (nextPosition, tuple(newVisitedCorners))
63                 cost = 1  # Step cost is always 1
64                 successors.append((nextState, action, cost))
65
66         self._expanded += 1  # DO NOT CHANGE
67         return successors
68
69     def getCostOfActions(self, actions):
70         """
71         Returns the cost of a particular sequence of actions.  If those actions
72         include an illegal move, return 999999.  This is implemented for you.
73         """
74         if actions is None: return 999999
75         x, y = self.startingPosition
76         for action in actions:
77             dx, dy = Actions.directionToVector(action)
78             x, y = int(x + dx), int(y + dy)
79             if self.walls[x][y]: return 999999
80         return len(actions)
81
```

Listing 2.7: Implementación final del problema de las esquinas

Comentarios

2.6. Corners Problem: Heurística

Descripción

Primera implementación

```

```

Listing 2.8: Implementación inicial de la heurística del problema de las esquinas

Implementación Final

```python
def cornersHeuristic(state, problem):
  """
  A heuristic for the CornersProblem that you defined.

    state:   The current search state (currentPosition, visitedCorners)
    problem: The CornersProblem instance for this layout.
  """
  currentPosition, visitedCorners = state
  corners = problem.corners

  # Identificar las esquinas que aún no han sido visitadas
  unvisitedCorners = [corner for corner in corners if corner not in visitedCorners]

  # Si no hay esquinas por visitar, la heurística es 0
  if not unvisitedCorners:
      return 0

  # Calcular la distancia mínima utilizando la distancia Manhattan
  heuristic = 0
  current = currentPosition

  while unvisitedCorners:
      # Encontrar la esquina más cercana (distancia Manhattan)
      distances = [(util.manhattanDistance(current, corner), corner) for corner in
    unvisitedCorners]
      minDistance, closestCorner = min(distances)

      # Agregar la distancia mínima a la heurística y moverse a la siguiente esquina
      heuristic += minDistance
      current = closestCorner
      unvisitedCorners.remove(closestCorner)

  return heuristic

```

Listing 2.9: Implementación final de la heurística del problema de las esquinas

Comentarios

2.7. Eating All The Dots: Heurística

Descripción

Primera implementación

```

```

Listing 2.10: Implementación inicial de la heurística del problema de las esquinas

Implementación Final

```
def foodHeuristic(state, problem):
  """
  A heuristic for the FoodSearchProblem.

  state: (pacmanPosition, foodGrid)
  problem: The FoodSearchProblem instance.
  """
  position, foodGrid = state

  # Convert foodGrid to a list of food positions
  foodList = foodGrid.asList()

  # Si no hay comida restante, la heurística es 0
  if not foodList:
      return 0

  # Calcular la distancia de laberinto a cada punto de comida desde la posición actual de Pacman
  distances = [mazeDistance(position, food, problem.startingGameState) for food in foodList]

  # La heurística es la distancia máxima a cualquier punto de comida
  return max(distances)

```

Listing 2.11: Implementación final de la heurística del problema de las esquinas

Comentarios

2.8. Suboptimal Search

Descripción

Primera implementación

```
1
```

Listing 2.12: Implementación inicial del problema de las esquinas

Implementación Final

```
1
```

Listing 2.13: Implementación final del problema de las esquinas

Comentarios

3. Resultados

3.1. Casos de pruebas

3.2. Autograder

```
Starting on 10-1 at 17:38:32

Question q1
===========

*** PASS: test_cases/q1/graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases/q1/graph_bfs_vs_dfs.test
***     solution:               ['0:A->B', '0:B->D', '0:D->G']
***     expanded_states:        ['A', 'B', 'D']
*** PASS: test_cases/q1/graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases/q1/graph_manypaths.test
***     solution:               ['0:A->B1', '0:B1->C', '0:C->D', '0:D->E1', '0:E1->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'D', 'E1', 'F']
*** PASS: test_cases/q1/pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 246
***     nodes expanded:         269

### Question q1: 3/3 ###


Question q2
===========

*** PASS: test_cases/q2/graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases/q2/graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases/q2/graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases/q2/graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases/q2/pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269

### Question q2: 3/3 ###


Question q3
===========

*** PASS: test_cases/q3/graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases/q3/graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases/q3/graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases/q3/graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
```

```
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases/q3/ucs_0_graph.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases/q3/ucs_1_problemC.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269
*** PASS: test_cases/q3/ucs_2_problemE.test
***     pacman layout:          mediumMaze
***     solution length: 74
***     nodes expanded:         260
*** PASS: test_cases/q3/ucs_3_problemW.test
***     pacman layout:          mediumMaze
***     solution length: 152
***     nodes expanded:         173
*** PASS: test_cases/q3/ucs_4_testSearch.test
***     pacman layout:          testSearch
***     solution length: 7
***     nodes expanded:         14
*** PASS: test_cases/q3/ucs_5_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']

### Question q3: 3/3 ###


Question q4
===========

*** PASS: test_cases/q4/astar_0.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases/q4/astar_1_graph_heuristic.test
***     solution:               ['0', '0', '2']
***     expanded_states:        ['S', 'A', 'D', 'C']
*** PASS: test_cases/q4/astar_2_manhattan.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         221
*** PASS: test_cases/q4/astar_3_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases/q4/graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases/q4/graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

### Question q4: 3/3 ###


Question q5
===========

*** Method not implemented: getStartState at line 311 of searchAgents.py
*** FAIL: Terminated with a string exception.

### Question q5: 0/3 ###


Question q6
===========

*** Method not implemented: getStartState at line 311 of searchAgents.py
*** FAIL: Terminated with a string exception.

### Question q6: 0/3 ###


```

```
Question q7
===========

*** PASS: test_cases/q7/food_heuristic_1.test
*** FAIL: test_cases/q7/food_heuristic_10.test
*** Heuristic failed non-triviality test
*** Tests failed.

### Question q7: 0/4 ###


Question q8
===========

[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Warning: this does not look like a regular search maze
*** Method not implemented: findPathToClosestDot at line 511 of searchAgents.py
*** FAIL: Terminated with a string exception.

### Question q8: 0/3 ###


Finished at 17:38:32

Provisional grades
==================
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
Question q5: 0/3
Question q6: 0/3
Question q7: 0/4
Question q8: 0/3
------------------
Total: 12/25

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.

```