

Sistemas Inteligentes

Proyecto 01:

**“Algoritmos de Búsquedas”**

Kennia Jackeline Sánchez Castillo A00517129

Profe. Luis Ricardo Peña Llamas

Monterrey, Nuevo León

Abril 02, 2024

**Objetivo**

Aplicar los diferentes algoritmos vistos en las clases de Algoritmos de Búsquedas:

* Búsquedas Sin Información.
* Búsquedas Con Información.

**Introducción**

Los algoritmos de búsqueda se pueden clasificar en dos categorías principales: búsqueda sin información y búsqueda con información.

Búsqueda sin información: Estos algoritmos no utilizan ninguna información adicional sobre el problema más allá de la definición de las reglas y restricciones del mismo. No tienen conocimiento sobre la ubicación del objetivo o la estructura del espacio de búsqueda. Estos algoritmos exploran el espacio de búsqueda de manera sistemática sin hacer suposiciones adicionales sobre la naturaleza del problema.

Ejemplos de algoritmos son los siguientes:

* Búsqueda en profundidad (DFS - Depth-First Search).
* Búsqueda en amplitud (BFS - Breadth-First Search).

Búsqueda con información (informada): Estos algoritmos utilizan información adicional sobre el problema para guiar la búsqueda hacia la solución de manera más eficiente. Esta información suele ser una función heurística que proporciona una estimación del costo para alcanzar el objetivo desde cualquier estado dado. Esta función heurística puede ser admisible (nunca sobreestima el costo real) o no admisible.

Ejemplo de algoritmo es:

* Búsqueda A\* (A\* Search).

**Desarrollo**

*Problema 1*

Question 2: Breadth First Search

Implement the breadth-first search (BFS) algorithm in the breadthFirstSearch function in search.py. Again, write a graph search algorithm that avoids expanding any already visited states. Test your code the same way you did for depth-first search.

python3 pacman.py -l mediumMaze -p SearchAgent -a fn=bfs

python3 pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5

Código Implementado

def breadthFirstSearch(problem):

"""Search the shallowest nodes in the search tree first."""

start\_state = problem.getStartState()

if problem.isGoalState(start\_state):

return []

queue = util.Queue()

queue.push((start\_state, []))

visited = set()

while not queue.isEmpty():

curr\_state, actions = queue.pop()

if curr\_state in visited:

continue

visited.add(curr\_state)

if problem.isGoalState(curr\_state):

return actions

successors = problem.getSuccessors(curr\_state)

for next\_state, action, \_ in successors:

if next\_state not in visited:

queue.push((next\_state, actions + [action]))

return []



Explicación Código

start\_state = problem.getStartState()

if problem.isGoalState(start\_state):

return []

Obtiene el estado inicial del problema y verifica si este estado es el estado objetivo. Si lo es, devuelve una lista vacía porque ya hemos alcanzado el objetivo.

queue = util.Queue()

queue.push((start\_state, [])) # Tupla de estado y acciones

Crea una cola (queue) para almacenar los estados a explorar. Inserta en la cola una tupla que contiene el estado inicial y una lista vacía de acciones (ya que aún no se ha realizado ninguna acción).

visited = set()

Crea un conjunto (visited) para llevar un registro de los estados que ya han sido visitados para evitar ciclos.

while not queue.isEmpty():

curr\_state, actions = queue.pop()

Inicia un bucle que se ejecuta mientras la cola no esté vacía. Extrae un estado de la cola junto con las acciones realizadas hasta ese estado.

if curr\_state in visited:

continue

visited.add(curr\_state)

Si el estado actual ya ha sido visitado, se ignora y se pasa al siguiente estado en la cola.

if problem.isGoalState(curr\_state):

return actions

Verifica si el estado actual es el estado objetivo. Si lo es, devuelve las acciones realizadas hasta ese punto.

successors = problem.getSuccessors(curr\_state)

for next\_state, action, \_ in successors:

if next\_state not in visited:

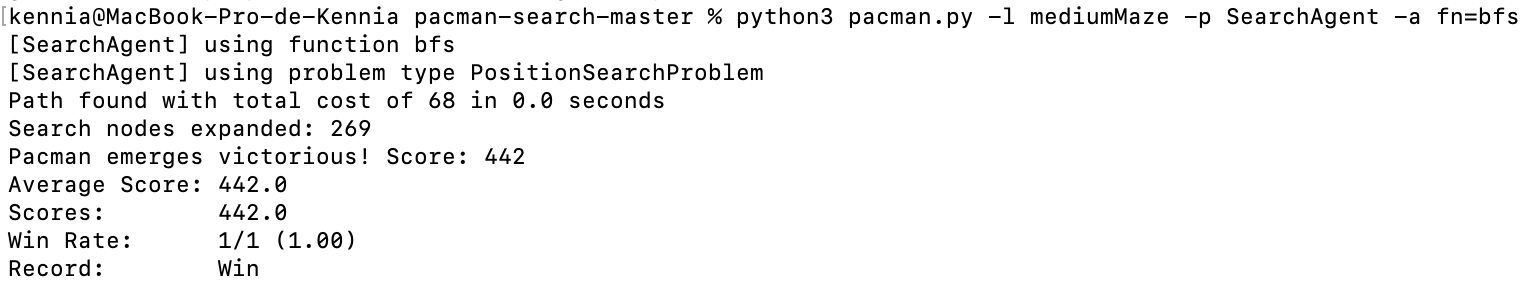
queue.push((next\_state, actions + [action]))

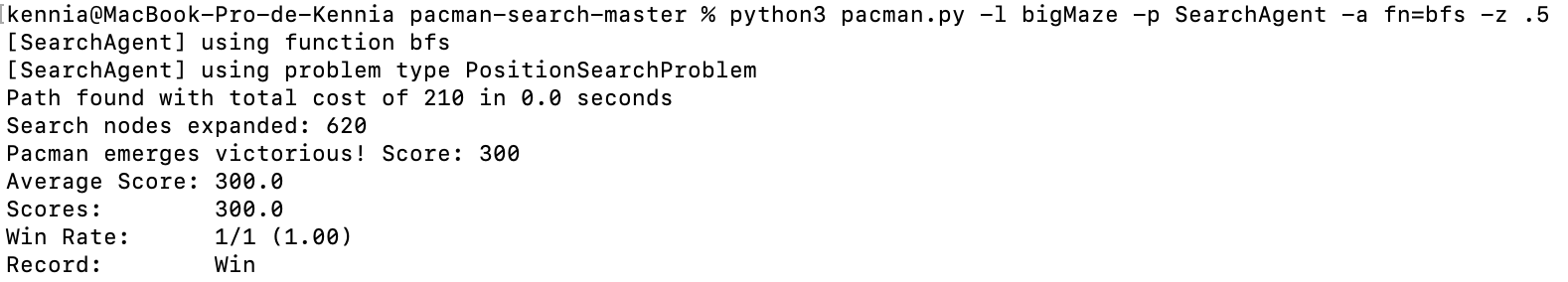
Obtiene los sucesores del estado actual y los explora. Para cada sucesor no visitado, agrega el sucesor a la cola junto con las acciones realizadas hasta ese punto más la nueva acción.

return []

Si no se encuentra una solución (es decir, la cola se vacía sin encontrar el estado objetivo), devuelve una lista vacía.

Evidencia





*Problema 2*

Question 4: A\* search

Implement A\* graph search in the empty function aStarSearch in search.py. A\* takes a heuristic function as an argument. Heuristics take two arguments: a state in the search problem (the main argument), and the problem itself (for reference information). The nullHeuristic heuristic function in search.py is a trivial example.

python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic

Código Implementado

def aStarSearch(problem, heuristic=nullHeuristic):

"""Search the node that has the lowest combined cost and heuristic first."""

"\*\*\* YOUR CODE HERE \*\*\*"

start\_state = problem.getStartState()

if problem.isGoalState(start\_state):

return []

priority\_queue = util.PriorityQueue()

priority\_queue.push((start\_state, [], 0), heuristic(start\_state, problem))

visited = set()

while not priority\_queue.isEmpty():

curr\_state, actions, cost = priority\_queue.pop()

if curr\_state in visited:

continue

visited.add(curr\_state)

if problem.isGoalState(curr\_state):

return actions

successors = problem.getSuccessors(curr\_state)

for next\_state, action, step\_cost in successors:

if next\_state not in visited:

next\_actions = actions + [action]

next\_cost = cost + step\_cost

priority\_queue.push((next\_state, next\_actions, next\_cost), next\_cost + heuristic(next\_state, problem))

return []



Explicación Código

start\_state = problem.getStartState()

if problem.isGoalState(start\_state):

return []

Se obtiene el estado inicial del problema. Si este estado es el objetivo, se devuelve una lista vacía porque ya hemos alcanzado el objetivo.

priority\_queue = util.PriorityQueue()

priority\_queue.push((start\_state, [], 0), heuristic(start\_state, problem))

Se crea una cola de prioridad para almacenar los estados a explorar. Se inserta en la cola una tupla que contiene el estado inicial, una lista vacía de acciones (porque aún no se han realizado acciones) y un costo acumulado inicial de 0, junto con la estimación heurística del costo restante para llegar al objetivo desde el estado inicial.

visited = set()

Se crea un conjunto para llevar un registro de los estados que ya han sido visitados para evitar ciclos.

while not priority\_queue.isEmpty():

curr\_state, actions, cost = priority\_queue.pop()

Se inicia un bucle que se ejecuta mientras la cola de prioridad no esté vacía. Se extrae un estado de la cola junto con las acciones realizadas hasta ese estado y el costo acumulado hasta ese estado.

if curr\_state in visited:

continue

visited.add(curr\_state)

Si el estado actual ya ha sido visitado, se ignora y se pasa al siguiente estado en la cola.

if problem.isGoalState(curr\_state):

return actions

Se verifica si el estado actual es el estado objetivo. Si lo es, se devuelve la lista de acciones que llevan hasta este estado.

successors = problem.getSuccessors(curr\_state)

for next\_state, action, step\_cost in successors:

if next\_state not in visited:

next\_actions = actions + [action]

next\_cost = cost + step\_cost

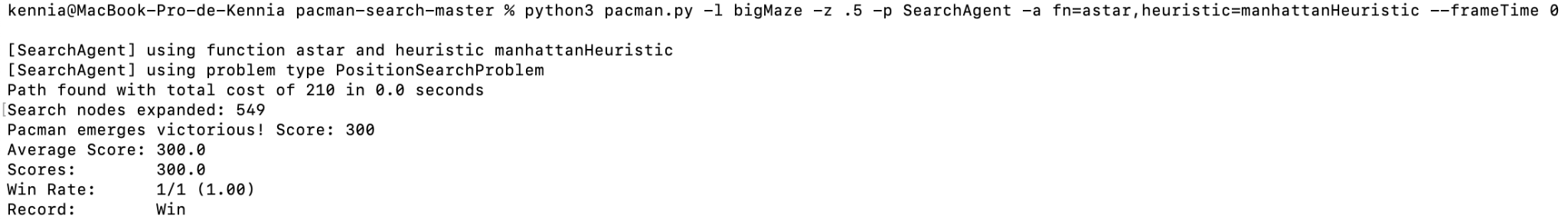
priority\_queue.push((next\_state, next\_actions, next\_cost), next\_cost + heuristic(next\_state, problem))

Se obtienen los sucesores del estado actual y se exploran. Para cada sucesor no visitado, se calcula el costo acumulado para alcanzarlo y se agrega a la cola de prioridad junto con las acciones realizadas hasta ese punto y la estimación heurística del costo restante.

return []

Si no se encuentra una solución (es decir, la cola de prioridad se vacía sin encontrar el estado objetivo), se devuelve una lista vacía.

Evidencia



**Conclusión**

En resumen, hemos explorado dos algoritmos de búsqueda comunes en inteligencia artificial: búsqueda en amplitud (BFS) y búsqueda A\* (A Star).

Ambos algoritmos tienen sus aplicaciones y se eligen según las características del problema en cuestión. BFS es útil cuando no hay información adicional sobre el espacio de búsqueda, mientras que A\* es eficiente cuando se dispone de una buena heurística que puede guiar la búsqueda hacia la solución de manera más efectiva.