|  |  |
| --- | --- |
|  | **UNIVERSIDAD ARGENTINA DE LA EMPRESA**  **Departamento de Tecnología Informática**  3.4.096 - INTELIGENCIA ARTIFICIAL  **Docentes: Ponzoni, Nelson** |

TP 2 – Búsqueda

Los algoritmos de búsqueda son utilizados para encontrar caminos o posibles soluciones a problemas realmente complejos. Existen diferentes alternativas y situaciones donde pueden ser aplicados. En el presente Trabajo Práctico implementaremos algunas en el proceso de un video juego, el Pacman.

# Objetivo

* Aplicar los algoritmos de búsqueda
* Entender la estructura de programación simple
* Comprender la complejidad del problema a resolver

# Algoritmos de Búsqueda

Debe implementar en el código los algoritmos de búsqueda informada

* *Búsqueda en profundidad*
* *Búsqueda en Anchura*
* *Búsqueda Costo Uniforme*
* *Búsqueda A\**

# Plazo

Según lo indicado en el cronograma. Webcampus.

# Tareas

El equipo debe realizar las siguientes actividades. En todos los casos, cada miembro del equipo debe poder justificar las respuestas.

1. Completar las secciones del código correspondiente a cada algoritmo
2. Presentar el código funcional

### Como jugar

Escribir el comando py pacman.py para ejecutar el juego.

### Búsqueda en profundidad (DFS)

Este algoritmo de búsqueda explora primero los nodos más profundos y los considera como parte de un camino parcial que puede expandirse. Para esto se utiliza una estructura de pila (LIFO) llamada *contenedor* donde el último nodo que entra debe ser el primero en salir, esto permite explorar primero los nodos más profundos y retroceder hasta la raíz. Así se construye un camino explorando una celda vecina, luego explorando la celda siguiente, luego la celda siguiente, y así sucesivamente.

def depthFirstSearch(problem):

contenedor = util.Stack()

inicio = problem.getStartState()

nodoInicial = (inicio, [])

contenedor.push(nodoInicial)

visitados = set()

while not contenedor.isEmpty():

nodo = contenedor.pop()

visitados.add(nodo[0])

if problem.isGoalState(nodo[0]): #si encuentra solución

return nodo[1]

sucesores = problem.getSuccessors(nodo[0])

for item in sucesores:

if item[0] in visitados:

continue

contenedor.push((item[0], nodo[1] + [item[1]]))

return None

Para probar el algoritmo generado se debe ejecutar:

py pacman.py -l tinyMaze -p SearchAgent -a fn=dfs

### Búsqueda en Anchura (BFS)

Este algoritmo de búsqueda explora primero los nodos menos profundos y los considera como parte de un camino parcial que puede expandirse. En este caso, se utiliza una estructura de cola (FIFO) donde el primer nodo ingresado es el primero en salir y permite explorar primero los nodos vecinos.

Ejemplo: Se explora todas las celdas vecinas en cada ruta que están a un paso de distancia. Dado que ninguna es la celda objetivo, se prueba las celdas a dos pasos de distancia, y si aún no hay celda objetivo intenta a tres pasos de distancia, y así sucesivamente.

def breadthFirstSearch(problem):

contenedor = util.Queue()

inicio = problem.getStartState()

nodoInicial = (inicio, [])

contenedor.push(nodoInicial)

visitados = set()

visitados.add(inicio)

while not contenedor.isEmpty():

nodo = contenedor.pop()

if problem.isGoalState(nodo[0]):

return nodo[1]

successors = problem.getSuccessors(nodo[0])

for item in successors:

if item[0] in visitados:

continue

visitados.add(item[0])

contenedor.push((item[0], nodo[1] + [item[1]]))

return None

Para probar el algoritmo generado se debe ejecutar:

py pacman.py -l tinyMaze -p SearchAgent -a fn=bfs

### Búsqueda Costo Uniforme

En este algoritmo de búsqueda se utiliza una cola con prioridad que tiene en cuenta el menor costo de la ruta acumulada, el orden por visitar y al expandir los nodos producirá una mejor ruta más corta y una menor cantidad de nodos expandidos.

def uniformCostSearch(problem):

contenedor = util.PriorityQueue()

inicio = problem.getStartState()

nodoInicial = (inicio, [], 0)

contenedor.push(nodoInicial, 0)

visitados = set()

while not contenedor.isEmpty():

nodo = contenedor.pop()

if problem.isGoalState(nodo[0]):

return nodo[1]

if nodo[0] not in visitados:

visitados.add(nodo[0])

for sucesores in problem.getSuccessors(nodo[0]):

if sucesores[0] not in visitados:

cost = nodo[2] + sucesores[2]

contenedor.push((sucesores[0], nodo[1] + [sucesores[1]], cost), cost)

return None

Para probar el algoritmo generado se debe ejecutar:

py pacman.py -l mediumMaze -p SearchAgent -a fn=ucs

## Búsqueda A\*

En este algoritmo además de ordenar el orden de visitar y expandir los nodos en función del costo del camino acumulado del nodo actual, tendrá en cuenta una heurística para aproximar mejor la distancia y alcanzar el estado objetivo.

def aStarSearch(problem, heuristic=nullHeuristic):

contenedor = util.PriorityQueue()

inicio = problem.getStartState()

nodoInicial = (inicio, [], 0)

contenedor.push(nodoInicial, 0)

visitados = set()

while not contenedor.isEmpty():

nodo = contenedor.pop()

if problem.isGoalState(nodo[0]):

return nodo[1]

if nodo[0] not in visitados:

visitados.add(nodo[0])

for sucesores in problem.getSuccessors(nodo[0]):

if sucesores[0] not in visitados:

costo = nodo[2] + sucesores[2]

costoTotal = costo + heuristic(sucesores[0], problem)

contenedor.push((sucesores[0], nodo[1] + [sucesores[1]], costo), costoTotal)

return None

Para probar el algoritmo usando la Distancia Manhattan como heurística se debe ejecutar:

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