Búsqueda en el juego de Pacman

Arian Gallardo, Luis Carranza and Patricio Avila
20153227, 20151110, 20151191

Pontificia Universidad Católica del Perú
Facultad de Ciencias e Ingeniería
Aplicaciones de Ciencias de la Computación

Abstract

This paper explains the most known ways for an agent to reach a certain goal on a certain problem with well-defined states. In this case, the Pacman game is taken as the model in which we are going to define some agents in order to reach the win state, optimizing both costs and memory. Moreover, those agents are going to work based on five graph-search algorithms: Depth First Search, Breadth First Search, Iterative Deepening Search, Bidirectional Search and A* search, which are going to be explained with more details in this document.

Introducción

En este informe se presentarán los resultados de la implementación y ejecución de los diferentes agentes de búsqueda implementados para el juego "Pacman" provisto por el repositorio del curso CS188 de UC Berkeley. El objetivo principal es realizar la contrastación de las medidas de desempeño que dichos agentes poseen, para luego realizar posibles conclusiones y decisiones sobre ello. En primer lugar, se explicará brevemente el desafío planteado sobre el juego. Luego, se explicaran las funciones de búsqueda implementadas para el funcionamiento de los agentes. Luego, se explicará de qué manera se representaron los agentes en los problemas planteados, principalmente en el problema "CornersProblem". Además, se dará una breve explicación de las heurísticas usadas para la implementación de uno de los algoritmos de búsqueda implementados para dichos agentes. Finalmente, se presentarán los resultados de desempeño en tablas y gráficas que nos permitan realizar un mayor análisis de las diferencias que presentan cada una de las técnicas planteadas en este informe.

El objetivo de los agentes de búsqueda implementados es el de encontrar el estado final de Pacman de manera eficaz, esto es, procurar que $Pacman$ logre comer todos los Pac-dots y regrese a su posición inicial. La eficiencia de los agentes variará de forma sustancial según el método de búsqueda implementado. En la Tabla 1 se muestran las medidas de desempeño con las que se evaluará la eficiencia de los agentes en el presente informe.

Medida de desempeño	Tipo de medida
Cantidad de Nodos Visitados	Tiempo
Cantidad de Nodos en Memoria	Memoria
Costo total de la solución	Optimalidad

Table 1: Medidas de desempeño para los agentes

Estructura de Estados para CornersProblem

Este desafio plantea habilitar un agente de búsqueda para que encuentre una ruta mínima que comience en la posición inicial del Pacman, visite las cuatro esquinas del layout y regrese a la posición inicial. Para esto, se definió un espacio de estados cuyos estados tienen la siguiente estructura:

$$(posición, estadoEsquinas) \tag{1}$$

donde **posición** es una tupla de las coordenadas x, y donde se encuentra el Pacman,

y **estadoEsquinas** es el estado en el que se encuentran cada una de las cuatro esquinas, como una tupla de cuatro numeros que pueden ser 0 (cuando la esquina no está visitada) o 1 (cuando la esquina ya ha sido visitada).

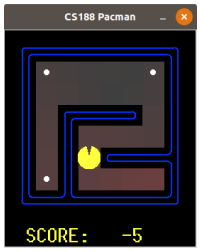

Figure 1: Pacman en estado ((3,1), (0,0,1,0))

Las esquinas del layout se encuentran en las posiciones

(1,1), (1, top), (right, 1), (right, top), con sus indices del 0 al 3 respectivamente. Por ejemplo:

$$((4,2),(0,0,1,0)) \tag{2}$$

significaría que Pacman se encuentra en la posicion $(4,2)$ del layout y la solo tercera esquina ya ha sido visitada.

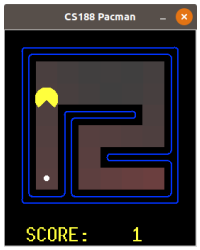

Figure 2: Pacman en estado ((1,4), (0,1,1,1))

Algoritmos de búsqueda

Para la búsqueda del estado final de Pacman, se usaron los siguientes algoritmos de búsqueda:

1. Depth First Search o Búsqueda en Profundidad:
Depth First Search (DFS) es un algoritmo de búsqueda en grafos que sirve para recorrer un grafo finito y no dirigido. Se basa en una búsqueda que expande el primer nodo vecino que encuentre dentro de sus vecinos que no han sido visitados. Se puede implementar con una estructura LIFO (una pila, por ejemplo). Este método asegura que se llegará al nodo solución, mas no de la manera óptima en función a costo.

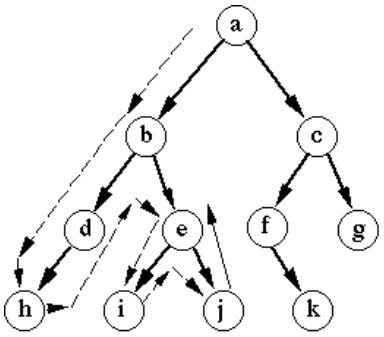

Figure 3: Depth First Search

2. Breadth First Search o Búsqueda en Anchura:
Breadth First Search (BFS) es un algoritmo de búsqueda en grafos que, al igual que DFS, sirve para recorrer un grafo finito y no dirigido, el cual se basa en una búsqueda "por niveles", o por "orden de llegada" apoyándose en la estructura Cola FIFO. Para un grafo con costos iguales en cada camino, esta búsqueda encontrará la solución óptima en función a su costo.

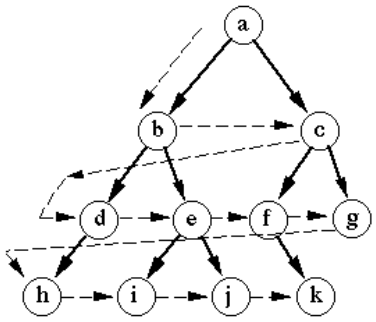

Figure 4: Breadth First Search

3. Iterative Deepening Search:
Iterative Deepening Search (IDS) es un algoritmo que búsqueda que se basa en DFS, incrementando la profundidad iterativamente con el fin de encontrar la mínima profundidad con la cual se puede alcanzar el nodo objetivo. Para un grafo con costos iguales en cada camino, esta búsqueda encontrará la solución óptima en función a su costo.

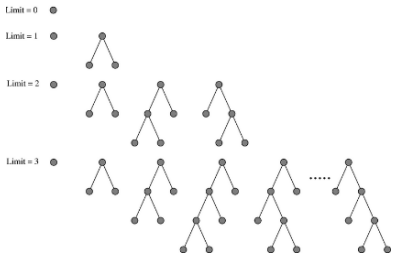

Figure 5: Iterative Deepening Search

4. Bidirectional Search:
Bidirectional Search (BDS) es un algoritmo que consiste en dos algoritmos BFS ejecutados en el estado inicial y en el estado final. Cada BFS guardar una cola de nodos a explorar y el Bidirectional Search terminar de buscar cuando el nodo actualmente a explorar sea el estado final o se encuentre en la cola de estados a explorar del otro BFS.

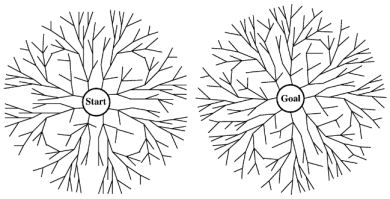

Figure 6: Bidirectional Search Example

5. A* Search:

A star Search (A*) es un algoritmo basado en la búsqueda de costos uniformes las cuales estan sujetas a una heurística dada. La heurística del problema debe ser consistente y admisible siguiendo las restricciones que el problema plantea. El objetivo principal de A* es encontrar el nodo objetivo con la menor cantidad de nodos expandidos, y, por consiguiente, el menor costo en memoria.

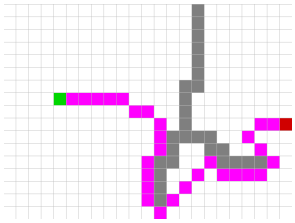

Figure 7: A* Search en una grilla

Heurística CornersHeuristic:

La heurística planteada para A* en nuestra implementación está basada en la distancia de Manhattan:

$$M(A(x1, y1), B(x2, y2)) = \|x1 - x2\| + \|y1 - y2\| \quad (3)$$

Nuestra heurística se define de la siguiente manera:

$$h(s) = \sum_{n=1}^{4} M(x, c[n]) * s(c[n]) + M(x, start)$$

donde $c[n]$ es la posición de la n-ésima esquina, $s(x)$ es el estado de la esquina x (0 si ya ha sido visitada y 1 si no ha sido visitada) y $start$ es la posición inicial de Pacman. En otras palabras, nuestra heurística calcula la suma de las distancias Manhattan desde la posición actual hacia todas las esquinas aún no visitadas, más la suma de la posición actual hacia la posición inicial del problema.

Se prueba la admisibilidad de la heurística con la siguiente gráfica de Costo Real vs Costo Estimado:

Una vez implementadas todas las funciones de búsqueda y sus respectivos agentes, se tienen los siguientes resultados:

Análisis de los resultados:

En cuanto a la **menor cantidad de nodos visitados**, DFS lleva la ventaja en ambos casos. Esto se debe a la manera de realizar la búsqueda de dicha función: expande siempre el primer nodo encontrado en los vecinos y continúa buscando hasta los niveles más profundos del grafo en primera instancia, encontrando el nodo objetivo sin necesidad de expandir nodos que se encuentren en un nivel menor.

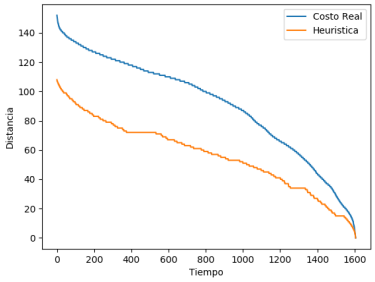

Figure 8: Costo Real y Costo Estimado (Heurística)

		DFS	BFS	IDS	BS	A* (cornersHeuristic)
mediumCorners	# nodos visitados	456	3031	52051	1878	1606
	# nodos en memoria	56	0	56	43	67
	Costo de la Solucion	246	152	339	152	152
bigCorners	# nodos visitados	823	10581	127212	6636	2469
	# nodos en memoria	131	14	131	149	73
	Costo de la Solucion	426	214	426	214	216

Figure 9: Comparación de diferentes metodos de búsqueda

En cuanto a la **menor cantidad de nodos en la memoria** (nodos que se quedan en la frontera al final de la ejecución), BFS lleva la ventaja en ambos casos. La causa de esto es que BFS siempre buscará explorar los nodos a medida que vayan expandiéndose, reduciendo así la cantidad de nodos que se quedan en la frontera en cada nivel. Esto también se puede notar en la cantidad de nodos visitados, la cual es relativamente alta con respecto a los demás métodos de búsqueda.

En cuánto al **menor costo de la solución**, BFS y A* llevan la delantera en ambos casos, con una ligera desventaja para A*. Como el grafo posee costos uniformes (cualquier movimiento cuesta 1) BFS será capaz de encontrar la solución óptima. Entonces, el costo hallado por BFS es el menor posible. A* difiere en 2 unidades en el segundo caso, sin embargo, con respecto a los nodos visitados, lleva una gran ventaja sobre BFS. Si bien la solución de A* no es óptima, se reduce el tiempo de ejecución de manera sustancial.

Agente de búsqueda codiciosa

En adición al agente planteado previamente, se implementó un agente de búsqueda que construye codiciosamente la solución al problema $CornersProblem$. Este agente siempre búsca la posición más cercana "objetivo", es decir, la esquina más cercana o la posición inicial (en caso todas las esquinas ya hayan sido visitadas). Los resultados son los siguientes:

Costos de las soluciones:

- mediumCorners:152
- bigCorners: 214

Pregunta: Vale la pena hacer dicha búsqueda codiciosa en los layouts testados?
Respuesta: No, pues al haber solo 4 puntos a los cuales llegar es muy probable que las distancias sumadas por la búsqueda codiciosa sea similar al de la búsqueda realizada por el agente inicial. Esto cambia cuando la cantidad de puntos aumenta: ir a los puntos más cercanos puede no ser óptimo en comparación a la solución con BFS.

Pregunta: La solución entregada será siempre la de menor costo?
Respuesta: No siempre. En una búsqueda con solución óptima, se debe tener trazada una solución en base a los posibles estados siguientes que el problema pueda generar, y también de los estados anteriores que el problema ha generado (así como en el algoritmo de Dijkstra). En mediumCorners y bigCorners en particular, se encontrará la solución óptima, pero esto no siempre será así.

References

-

CS-STACKEXCHANGE:

https://cs.stackexchange.com/questions/60674/why-how-are-bfs-nodes-goal-tested-when-they-are-generated

Brian Logan: Iterative Deepening Search:

http://www.cs.nott.ac.uk/ pszbsl/G52APT/slides/09-Iterative-deepening.pdf

Steven M. Lavalle: Planning Algorithms:

http://planning.cs.uiuc.edu/node50.html