Inteligencia Artificial
Problemas de búsqueda

Alberto González Rosales
Grupo C-511

En este proyecto se brindan varios archivos en *python*, los cuales simulan el mundo de *Pacman*. En esta solución hay porciones de código incompletos, que tendrán que ser rellenados para lograr objetivos específicos como llegar a alguna posición particular dentro del laberinto o recolectar comida de forma eficiente. Las tareas asignadas y soluciones propuestas serán discutidas a continuación.

1. Preguntas 1 - 4: Algoritmos de Recorrido

El primer *problema de búsqueda* propuesto consistía en encontrar una posición específica dentro del laberinto y devolver la secuencia de acciones a realizar para llegar desde la posición de *Pacman* hasta el objetivo.

Para esto se brinda una clase llamada *SearchProblem*, la cual es abstracta y tiene la estructura clásica de un problema de búsqueda. Cuenta con métodos para devolver el estado inicial del problema y determinar si un estado es final. Además, tiene una función *getSuccessors* que devuelve, dado un estado, una lista de triplas con la siguiente información: el estado siguiente, la acción a realizar para llegar a ese estado y el costo de hacer esa acción.

Por tanto, la tarea a realizar es implementar la forma en la que se busca la posición objetivo en este grafo de estados.

1.1. Depth-First Search

El primer recorrido es *depth-first search* o *dfs*. Para simular el recorrido de *dfs* se tiene implementada en el módulo *utils.py* una clase *Stack*, la cual es una estructura de datos cuya política de inserción y extracción es de la forma *LIFO*, o sea, *Last In First Out*.

Se inicia el algoritmo insertando el estado inicial en nuestra estructura de datos. Luego, mientras no esté vacía, se extrae de la pila el estado correspondiente, si este estado es el objetivo, el algoritmo acaba, sino se revisa la lista de los sucesores del mismo y se insertan en la pila todos aquellos que no hayan sido visitados.

Como se pide además una lista de acciones para llegar al estado objetivo, se cuenta con un array *parent*, el cual tendrá para cada estado alcanzable en el recorrido cual fue el estado que lo descubrió y con que acción fue descubierto.

Se implementó además una función *getActions* encargada de reconstruir la secuencia de acciones; a este método se le pasa el estado objetivo y el array *parent* y este recorre todos los estados en el camino desde el objetivo hasta el estado inicial, conformando una lista de las acciones en el orden contrario al que se realizaron.

1.2. Breadth-First Search

El segundo recorrido es *breadth-first search* o *bfs*. Este recorrido, en términos de implementación es muy parecido al anterior. De hecho, solo cambia la estructura de datos con que se almacenan los estados. En este caso es una cola, la cual también está implementada en *utils.py*, cuya política de inserción y extracción es de la forma *FIFO*, *First In First Out*, y es lo que hace que el orden de visitar los estados sea diferente en ambos algoritmos.

Igualmente se comienza insertando el estado inicial en la cola. Se extrae un estado mientras no esté vacía. Se comprueba si es el objetivo y se termina en caso de que lo sea. Sino se revisan sus sucesores y se insertan en la cola si no han sido visitados. Este es el momento donde se asigna a cada estado cual es su correspondiente en el array *parent*.

Una vez terminado el recorrido se reconstruye la secuencia de acciones con el método *getActions*.

Este recorrido tiene la particularidad de que si los costos para transitar de un estado a otro son iguales, entonces encontrará el camino de costo mínimo desde el estado inicial al estado objetivo. Sin embargo puede no hacerlo en el caso más general que es cuando los costos no son todos iguales.

1.3. Uniform Cost Search

La búsqueda de costo uniforme se utiliza cuando se desea encontrar el camino de menor costo total a un estado particular. O sea, de todos los caminos que existen entre el estado inicial y el estado objetivo, aquel que minimice la suma de los costos de las aristas transitadas.

Se tendrá en todo momento un array *cost* que indica el menor costo descubierto entre el estado inicial y todos los demás. Al inicio del algoritmo se puede interpretar que el costo del estado inicial es cero, y el de todos los demás estados es *infinito*.

El algoritmo inicia insertando en algún tipo de estructura de datos el estado inicial. Ahora, mientras queden estados insertados, se extrae el que actualmente tiene el menor costo desde el estado inicial. Se recorren los sucesores de este estado y se comprueba para cada uno de ellos si se puede *relajar*, o sea, si puede mejorar el costo que tiene actualmente. Para eso se hace la comprobación $cost[currentState] + edgeCost < cost[currentSuccessor]$, si esto se cumple entonces se actualiza el costo de ese estado sucesor y se inserta en la estructura.

Para realizar este algoritmo de forma eficiente se cuenta en el módulo *utils.py* con una clase *PriorityQueue*. Esta estructura asigna a cada estado actualmente contenido en ella una prioridad, y cada vez que se le pide extraer un estado devolverá aquel con la menor prioridad. Como se quiere en todo momento extraer el más cercano al inicio, la prioridad para cada estado será el menor costo encontrado para ese estado hasta el momento.

Solo queda saber como actualizar el array *parent* para luego reconstruir la secuencia de acciones. Para esto hay que notar que cada vez que se pueda *relajar* un estado es porque se encontró un camino de menor costo desde el estado inicial hasta ese estado, el cual, potencialmente, es parte del camino de costo mínimo hasta el estado objetivo. Por tanto se le asigna como padre el estado que provocó su *relajación*.

1.4. A*

El algoritmo $A*$ se utiliza cuando a un problema de búsqueda se le asigna una heurística. Una heurística es una función que dado un estado devuelve un valor asociado al mismo, que da una medida de cuan *distante* se encuentra de un estado final. Un ejemplo de heurística es la *nula*, que devuelve el valor cero independientemente de cual sea el estado que evalúe.

Digamos ahora que a cada estado le corresponde un valor $h(u)$, el valor de la heurística evaluada en ese estado. Definamos como $g(u)$ el menor costo para ir desde el estado inicial al estado u utilizando el algoritmo de *Búsqueda de Costo Uniforme*.

Sea $f(u) = g(u) + h(u)$, entonces el algoritmo $A*$ se implementa igual que el *UCS* con la diferencia de que la prioridad de cada estado es su valor de f.

1.5. Generalidades y particularidades

En principio cada algoritmo para recorrer el grafo de estado tiene sus particularidades, pero son bastante parecidos. Todos terminan cuando sale de la estructura de datos algún estado final, todos inician insertando el estado inicial en la estructura y todos expanden cada estado con una función que indica cuales son los sucesores del mismo.

Las diferencias radican principalmente en la estructura de datos que utiliza cada uno ya que las políticas de inserción y extracción difieren, logrando así cada uno ajustarse mejor a su propósito.

2. Preguntas 5 - 6: Encontrando las esquinas

El problema que se plantea a continuación es dado un laberinto encontrar el camino más corto que pase por las cuatro esquinas del mismo. O sea, *Pacman* comienza en una posición y tiene que hacer un recorrido donde toque cada esquina al menos una vez. Además este recorrido debe ser de longitud mínima.

2.1. Caracterizando el problema

Para este problema de búsqueda se pide definir los estados que caracterizan el problema, así como implementar las funciones *getStartState*, *isGoalState* y *getSuccessors*.

La representación de estado utilizada fue un par (x, y) que representa la posición actual donde se encuentra *Pacman* y un vector *booleano* de tamaño 4, que indica para cada esquina si ya se visitó o no.

El estado inicial consiste en la posición inicial de *Pacman*, información que se tiene cuando se inicializa el juego, y un vector *booleano* con todos sus elementos tomando valor *False*, ya que ninguna esquina ha sido visitada.

Para determinar si un estado es final basta con comprobar si alguno de sus valores tiene valor *False*, sino significa que todas las esquinas han sido visitadas y, por tanto, terminamos.

Los sucesores de un estado serán unas ternas de la forma *nextState*, *action*, *cost*. El costo para transitar de un estado a otro es 1 en este problema. Las acciones pertenecen al conjunto (*UP*, *DOWN*, *LEFT*, *RIGHT*). El estado al que se transita dado que se realiza determinada acción es de la forma (*nextPosition*, *mask*), donde *nextPosition* es la nueva posición de *Pacman* y *mask* es el vector booleano actualizado si la nueva posición es una de las esquinas.

2.2. Heurísticas

Se probaron tres heurísticas consistentes para resolver este problema. A continuación se explicará y analizará cada una.

2.2.1. Cantidad de esquinas sin visitar

La primera heurística es bastante intuitiva, consiste en devolver para cada estado la cantidad de esquinas que aún no han sido visitadas.

Análisis de admisibilidad:

- $h >= 0$ ya que la cantidad de esquinas restantes siempre es un número entero en el rango $[0, 4]$.
- $h <= h*$: supongamos que la cantidad de esquinas sin visitar en el estado *state* es x, entonces en la solución óptima el valor de $h*$ tiene que ser mayor o igual que x ya que se necesitan al menos x pasos del *Pacman* para visitar las x esquinas restantes.

Análisis de consistencia:

- Sea la arista que existe desde el estado x al estado y, se cumple que $h(x) - h(y) <= cost(x- > y)$. En este problema el costo de transitar cada arista es 1. Por tanto se cumple que $h(x) - h(y) <= 1$ siempre, ya que cuando pasamos de un estado a otro estado sucesor puede ocurrir que ese nuevo estado sea una esquina o no. En el segundo caso los valores de h en esos estados son iguales y en el primero difieren en 1.

Esta heurística a pesar de ser consistente expandía una cantidad considerable de estados. En los casos de prueba con los que cuenta el proyecto mostraba que expandía alrededor de 2000 estados, lo cual solo otorgaba 1 punto de 3 posibles.

2.2.2. Mayor distancia Manhattan

Una heurística un poco más elaborada es devolver la distancia a la esquina que aún no ha sido visitada más lejana de la posición de *Pacman* si tenemos en cuenta la *distancia Manhattan*.

Análisis de admisibilidad:

- $h >= 0$ ya que la *distancia Manhattan* entre dos posiciones siempre es un número entero en el rango $[0, n + m]$, donde n y m son las dimensiones del laberinto.
- $h <= h*$: si la cantidad de esquinas sin visitar es x, el valor de la heurística en la solución óptima pasa por definir un orden en el que se visitarán las x esquinas y devolver la menor de las distancias recorridas visitando las esquinas en ese orden. Esto es equivalente a resolver el problema del viajante, se estaría determinando el orden óptimo en que se deben visitar las esquinas para minimizar el costo total. Ls solución propuesta h será la mayor de las *distancias Manhattan*, pero, en la solución óptima la distancia entre dos posiciones tiene que ser la distancia real entre ambas, teniendo en cuenta que en el laberinto existen obstáculos. Sería realizar un recorrido *bfs* y devolver la distancia desde la posición de partida hasta la final. Queda bastante claro entonces que la heurística h es menor o igual a $h*$ ya que la *distancia Manhattan* no tiene en cuenta el hecho de que puedan existir obstáculos en el laberinto y devuelve la distancia más corta entre dos posiciones si no hubiesen obstáculos en el camino.

Analicemos su consistencia:

- Sea la arista que existe desde el estado x al estado y, se cumple que $h(x) - h(y) <= cost(x- > y)$. En este problema el costo de transitar cada arista es 1. Por tanto se cumple que $h(x) - h(y) <= 1$ siempre, ya que cuando pasamos de un estado a otro estado sucesor la *distancia Manhattan* entre ambas posiciones hasta la esquina más alejada puede diferir a lo sumo en 1. Esto se debe a que dos estados adyacentes mantienen las misma coordenada X o la misma coordenada Y, y la coordenada que cambia difiere en 1 de la anterior.

Esta heurística expandía una cantidad de estados cercana a los 1160, suficiente para obtener el máximo de puntos en esta pregunta. Además, el algoritmo de búsqueda con esta heurística no sufre afectaciones en su complejidad temporal ya que el costo de calcular la misma es constante.

2.2.3. Mayor distancia real

La heurística probada con mejores resultados fue la de devolver la *distancia real* más grande desde la posición de *Pacman* hasta alguna de las esquinas. Cuando se dice *distancia real* se hace alusión a la distancia que es devuelta al realizar el algoritmo *bfs*.

Análisis de admisibilidad:

- $h >= 0$ ya que la *distancia real* entre dos posiciones siempre es un número entero en el rango $[0, n * m]$, donde n y m son las dimensiones del laberinto.

- $h <= h*$: como se enunció en la demostración de admisibilidad de la heurística anterior, la solución óptima de este problema es la menor suma de *distancias reales* entre posiciones consecutivas dado que determinamos un orden en que visitar las esquinas. Cuando en un estado quedan tanto 0 como 1 esquina por visitar, el valor h será igual al de $h*$. Cuando queda más de una esquina el valor de h será menor que el de $h*$.

Analicemos su consistencia:

- Sea la arista que existe desde el estado x al estado y, se cumple que $h(x) - h(y) <= cost(x- > y)$. En este problema el costo de transitar cada arista es 1. Por tanto se cumple que $h(x) - h(y) <= 1$ siempre, ya que cuando pasamos de un estado a otro estado sucesor la *distancia real* entre ambas posiciones hasta la esquina más alejada puede diferir a lo sumo en 1. Esto se debe a que dos estados adyacentes mantienen las misma coordenada X o la misma coordenada Y, y la coordenada que cambia difiere en 1 de la anterior.

Con esta heurística la cantidad de estados expandidos eran aproximadamente 800. La complejidad temporal del algoritmo de búsqueda se afecta un poco ya que para calcular el valor de la heurística en un estado es necesario realizar un recorrido *bfs*.

3. Pregunta 7: Comiendo todos los puntos

El siguiente problema de búsqueda consistía en encontrar una secuencia de acciones que permitiera a *Pacman* comerse todos los puntos que se encuentran en el laberinto con el menor costo posible. La idea de esta pregunta también era diseñar una heurística que se ajustara al problema y que expandiera la menor cantidad de estados posibles.

Las heurísticas que se analizarán a continuación son las vistas previamente con la excepción de que ahora no se revisan solo las cuatro esquinas sino toda posición que tenga un punto que *Pacman* se puede comer; así que solo se mostrarán los resultados obtenidos con cada una de ellas ya que las demostraciones de consistencia y admisibilidad son equivalentes.

3.1. Mayor distancia Manhattan

La cantidad de estados expandidos con esta heurística es aproximadamente 9000, lo que garantizaba un total de 3 puntos en esta pregunta, lo cual es aceptable pero aún mejorable.

3.2. Mayor distancia real

Con la heurística de la mayor *distancia real*, a pesar de que se demora considerablemente más que la anterior, se expanden solamente una cantidad cercana a los 4000 estados. Para obtener el máximo de puntuación en esta pregunta era necesario que se expandieran a lo sumo 7000.

4. Pregunta 8: Búsqueda subóptima

Esta pregunta enuncia la necesidad de encontrar a veces una solución que sea lo suficientemente buena aunque no sea la óptima. El problema radica en que la solución óptima de algunos problemas es demasiado costosa de computar en términos de tiempo o memoria.

Para el problema de comer todos los puntos en el laberinto se pide implementar un agente que siempre vaya al punto más cercano. Esta estrategia *greedy* no siempre encuentra la solución óptima pero es bastante rápida y devuelve una solución suficientemente buena.

El proyecto brinda una implementación incompleta de una agente. Solo falta completar una función que recibe un estado y devuelve una lista de acciones que llevan de ese estado al punto más cercano. Lo interesante es que hasta este punto en el proyecto, si se implementaron correctamente los algoritmos de búsqueda, se tiene todo lo necesario para completar esta función sin mucho trabajo.

Podemos interpretar este problema como un problema de encontrar una posición en el laberinto, o sea, cuando antes nos interesaban las esquinas ahora nos interesan las posiciones donde existe un punto que *Pacman* se puede comer. Así que nuestros estados finales serán aquellos donde la matriz *food* en la posición (x, y) tenga valor *True*.

Una vez definido nuestro problema de búsqueda solo resta encontrar una secuencia de acciones con los algoritmos *A**, *Uniform Cost Search* o *BFS*.