

Algoritmos de búsqueda

Práctica de Laboratorio

Ing. Jose Eduardo Laruta Espejo

2 de marzo de 2020

Universidad La Salle

1. Introducción

Bienvenido a pacman!

2. DFS

3. BFS

4. UCS

5. A*

6. Visitar Esquinas: Representación

7. Visitar Esquinas: Heurística

8. Comer todo: Heurística

9. Búsqueda subóptima

Introducción

Este laboratorio tiene como objetivo el de implementar los algoritmos de búsqueda de árboles implementados en un entorno de un videojuego.

Pacman debe encontrar el camino a través de distintos laberintos para llegar a un lugar en particular y recolectar comida eficientemente.

El archivo con lo necesario se encuentra en este link

Al igual que en la anterior práctica de laboratorio, esta práctica cuenta con un programa autoevaluador para comprobar la efectividad y correcta implementación de los algoritmos.

Las prácticas consisten en implementar algunas funciones en los siguientes archivos de código fuente:

- `search.py`: Archivo donde se deben implementar todos los algoritmos de búsqueda.
- `searchAgents.py`: Archivo donde se deben implementar detalles acerca de los agentes.

Por su parte, resultará útil estudiar los siguientes archivos (no editar):

- `util.py`: Estructuras de datos útiles para implementar los algoritmos.
- `pacman.py`: Archivo que describe el estado del juego de pacman.
- `game.py`: Contiene la lógica de todo el mundo de Pacman.

Bienvenido a Pacman

Luego de descargar el fichero comprimido, descomprimir e ingresar a la carpeta correspondiente. Se puede correr el juego usando: `python pacman.py`.

El agente más simple implementado en `searchAgents.py` se llama `GoWestAgent` y siempre se dirige hacia el oeste (un agente reflejo trivial).

```
1 $ python pacman.py --layout testMaze --pacman GoWestAgent
2 $ python pacman.py --layout tinyMaze --pacman GoWestAgent
```

La opción `-l` o `--layout` se puede usar para seleccionar el mapa.

Nota. Todos los comandos usados se pueden encontrar en el archivo `commands.txt`

DFS

Ejercicio 1(3 puntos): Encontrando un punto fijo usando DFS

En el archivo `searchAgents.py` se puede encontrar un agente (`SearchAgent`) completamente implementado que planifica un camino a través del mapa y ejecuta el plan paso a paso. Los algoritmos para encontrar este plan no han sido implementados.

Primero, pruebe que el agente funciona correctamente con el siguiente comando:

```
1 $ python pacman.py -l tinyMaze -p SearchAgent -a fn=
    tinyMazeSearch
```

El anterior comando le dice al `SearchAgent` que use `tinyMazeSearch` como su algoritmo de búsqueda que esta implementado en `search.py`.

Ejercicio 1(3 puntos): Pseudo código

A continuación se muestra pseudo código para implementar un algoritmo de búsqueda de grafos¹.

```
1 def busqueda_grafo(problema, frontera) return solucion
2   cerrados <- conjunto vacio
3   frontera <- insertar(nodo(estado_inicial))
4   loop:
5     if frontera esta vacia: return falla
6     nodo <- REMOVE_DE_FRONTERA(frontera)
7     if ES_OBJETIVO(problema, estado(nodo)):
8       retornar camino
9     if estado(nodo) no esta en cerrados:
10      AGREGAR estado(nodo) a cerrados
11      for nodos_hijos en EXPANDIR(nodo):
12        frontera <- INSERTAR(nodo_hijo, frontera)
13      end
14   end
```

¹La búsqueda de grafos, a diferencia de la búsqueda de árboles, ignora los nodos de los estados anteriormente visitados.

Ejercicio 1(3 puntos): DFS

Implemente el algoritmos de búsqueda por profundidad DFS en la función `depthFirstSearch` en el archivo `search.py` con las siguientes consideraciones:

- La función debe retornar una **lista de acciones** que llevarán al agente desde el inicio al objetivo, las acciones deben ser válidas.
- Asegúrese de usar las estructuras de datos proveídas en `util.py`.

Ejercicio 1(3 puntos): DFS

Para probar el funcionamiento correcto de su algoritmo use los siguientes comandos con distintos mapas:

```
1 $ python pacman.py -l tinyMaze -p SearchAgent
2 $ python pacman.py -l mediumMaze -p SearchAgent
3 $ python pacman.py -l bigMaze -z .5 -p SearchAgent
```

Se podrá visualizar una capa de color rojo sobre el mapa luego de realizar la búsqueda con el orden en el que cada nodo fue explorado (rojo claro significa expansión más pronta).

Use las funciones definidas en los comentarios para probar los datos que se usarán en el problema.

BFS

Ejercicio 2(3 puntos): BFS

Implemente búsqueda en anchura BFS en la función `breadthFirstSearch`. Para probar el funcionamiento de su algoritmo puede especificar el algoritmo usando la opción `-a fn=bfs`:

```
1 $ python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
2 $ python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
```

BFS encuentra la solución más corta?

Si su función ha sido implementada de forma genérica puede usarla también en el problema del rompecabezas de piezas deslizantes usando: `python eightpuzzle.py`.

UCS

Ejercicio 3(3 puntos): UCS

BFS es capaz de encontrar la solución más corta pero usualmente se debe considerar otros aspectos como el costo de las transiciones entre nodos. Considere los siguientes mapas: `mediumDottedMaze` y `mediumScaryMaze`.

Cambiando la función de costo, se puede aconsejar a Pacman a encontrar distintos caminos. Por ejemplo para evitar lugares peligrosos o dirigirse a áreas con más comida, un agente Pacman racional debería tomar en cuenta estas consideraciones.

Ejercicio 3(3 puntos): UCS

Implemente búsqueda de costo uniforme UCS en la función `uniformCostSearch`. Para probar el funcionamiento de su algoritmo puede especificar el algoritmo usando la opción `-a fn=ucs`. Su algoritmo debería comportarse adecuadamente en los siguientes 3 mapas²:

```
1 $ python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
2 $ python pacman.py -l mediumDottedMaze -p
   StayEastSearchAgent
3 $ python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

²Los agentes y las funciones de costo para los dos últimos mapas han sido implementadas y no debe modificar nada para correr las simulaciones.

A*



Ejercicio 4(3 puntos): A*

Implemente A* en la función `uniformCostSearch`. A* toma una función heurística como argumento. Las heurísticas tienen 2 argumentos: el estado del problema y el mismo problema.

Para un ejemplo, puede referirse a la función heurística trivial `nullHeuristic` en `search.py`.

Puede probar su implementación del A* en los mismos mapas de DFS y BFS usando la heurística de la *distancia de Manhattan* (implementación disponible en la función `manhattanHeuristic` en `searchAgents.py`)

```
1 $ python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=
    astar,heuristic=manhattanHeuristic
```

Visitar Esquinas: Representación

Visitar Esquinas: Heurística

Comer todo: Heurística

Búsqueda subóptima

Una vez descargado el archivo, descomprimir el contenido en alguna carpeta destinada a proyectos de programación. dentro de la carpeta encontrará varios archivos de python.

Para este Laboratorio, se tienen un conjunto de archivos que deberán editar:

- `addition.py`: ejercicio 1
- `buyLotsOfFruit.py`: ejercicio 2
- `shop.py`: ejercicio 3
- `shopSmart.py`: ejercicio 3

Una vez editados los archivos correspondientes a los ejercicios, deberá ejecutar el autoevaluador para comprobar la solución.

El archivo para evaluar los ejercicios se llama `autograder.py`

Para ejecutar la evaluación de los ejercicios:

```
1 $ python autograder.py
```

Los archivos restantes se pueden ignorar, no los toque.

Ejercicio 1 (ejemplo) i

Abrir el archivo `addition.py` y ver la definición de `add`:

```
1 def add(a, b):  
2     "Return the sum of a and b"  
3     "*** YOUR CODE HERE ***"  
4     return 0
```

Modificar de la siguiente manera:

```
1 def add(a, b):  
2     "Retorna la suma de a y b"  
3     print('a={}, b={}, retornando a+b={}'.format(a,b,a+b))  
4     return a+b
```

Correr a evaluacion:

```
1 $ python autograder.py -q q1
```

Ejercicio 2

Añadir una función llamada `buyLotsOfFruit(orderList)` al archivo `buyLotsOfFruit.py` que tome como parámetro una lista de tuplas `(fruta, kilo)` y retorne el costo de la lista. Si aparece una fruta en la lista que no este presente en `fruitPrices` se debe imprimir un mensaje de error y retornar `None`.

No cambie la variable `fruitPrices`

Ejercicio 3

Complete la función `shopSmart(orders,shops)` en el archivo `shopSmart.py` que recibe un `orderList` (similar al del anterior ejercicio) y una lista de objetos `FruitShop` y retorne el objeto `FruitShop` que posea el menor costo total. No cambie los nombres de archivos o de variables.

Compruebe el funcionamiento usando el archivo de evaluación `autograder.py`