# Algoritmos de búsqueda

## Práctica de Laboratorio

Ing. Jose Eduardo Laruta Espejo

3 de marzo de 2020

Universidad La Salle

# Contenido i

# Introducción

# 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

# Introducción

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.

# Introducción

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).

```
$ python pacman.py --layout testMaze --pacman GoWestAgent
$ 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:

```
$ 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 grafos[1].

```
1  def busqueda_grafo(problema, frontera) return solucion
2      cerrados <- conjunto vacio
3      frontera <- insertar(nodo(estado_inicial))
4      loop:
5          if fontera esta vacia: return falla
6          nodo <- REMOVER_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
```

[1] La 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:

```
$ python pacman.py -l tinyMaze -p SearchAgent
$ python pacman.py -l mediumMaze -p SearchAgent
$ 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`:

```
$ python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
$ 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`. Suy algoritmo debería comportarse adecuadamente en los siguientes 3 mapas[2]:

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

[2]Los 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`)

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

# Visitar Esquinas: Representación

# Ejercicio 5(3 puntos): Visitar esquinas

Se procederá a formular un nuevo problema diseñando la función heurística.

En ciertos mapas, existen 4 puntos de comida, uno en cada esquina. El nuevo problema de búsqueda consiste en encontrar el camino más corto a través del mapa que visite las 4 esquinas.

**Nota:** Asegúrese de haber completado el ejercicio 2 antes.

Implemente el problema `CornersProblem` en el archivo `searchAgents`. Se necesita elegir una representación de estado que codifique la información necesaria para detectar las 4 esquinas y cuándo han sido visitadas. El agente debería poder resolver:

```
1 $ python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,
    prob=CornersProblem
2 $ python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs
    ,prob=CornersProblem
```

# Ejercicio 5(3 puntos): Visitar esquinas

Para recibir la nota completa se debe definir una representación abstracta del estado que no codifique información irrelevante al problema (posición de fantasmas, comida extra, etc.). En particular, no use `GameState` como estado en la búsqueda, el algoritmo resultante será demasiado lento.

**Pista:** Las únicas partes del estado del juego que necesita referenciar en su implementación son la posición de Pacman y la ubicación de las 4 esquinas.

# Visitar Esquinas: Heurística

# Ejercicio 6(3 puntos): Visitar esquinas (Heurística)

Implemente una heurística no trivial consistente y admisible para `CornersProblem` en la función `cornersHeuristic`.

Pruebe el funcionamiento con:

```
$ python pacman.py -l mediumCorners -p AStarCornersAgent -z
    0.5
```

# Comer todo: Heurística

# Ejercicio 7(4 puntos): Comer todo

Para este problema ya se cuenta con una definición en `FoodSearchProblem` in `searchAgents.py`. Una solución se define como el camino que recolecte toda la comida disponible en el mapa. Si ha implementado correctamente A*, el siguiente test debería resultar exitoso:

```
$ python pacman.py -l testSearch -p AStarFoodSearchAgent
```

**Nota:** Asegúrese de haber completado exitosamente el ejercicio 4 antes de implementar el ejercicio 7.

# Ejercicio 7(4 puntos): Comer todo

Implemente una función heurística en `foodheuristic` en el archivo `searchAgents.py` que cumpla con las condiciones de admisibilidad y consistencia para el problema `FoodSearchProblem`.

Pruebe la implementación con su agente usando:

```
$ python pacman.py -l trickySearch -p AStarFoodSearchAgent
```

**Nota:** La función heurística debe cumplir con los requisitos de consistencia y admisibilidad, caso contrario no se otorgarán puntos. Además, mientras menos iteraciones tome la búsqueda más puntos podrá obtener.

# Búsqueda subóptima

# Ejercicio 8(3 puntos): Búsqueda subóptima

A veces, incluso con A* y una buena heurística, encontrar el camino óptimo para recoger toda la comida es complicado. En estos casos se requiere un camino razonablemente bueno de manera rápida. Se deberá implementar un agente que siempre vaya a la comida más cercana.

`ClosestDotSearchAgent` está implementada en `searchAgents.py`, pero le falta una función clave para encontrar el camino a la comida más cercana.

Implemente la función `findPathToClosestDot` en `searchAgents.py`.

```
$ python pacman.py -l bigSearch -p ClosestDotSearchAgent -z
    .5
```

**Nota:** Debe completar `AnyFoodSearchProblem` con el test objetivo y luego resolver el problema con una función de búsqueda apropiada. La solución debería ser bastante corta.