Diego Sáez Mulero U137519
Amanda Pintado Lineros U137702

# Project 1- Pacman Search, 2020

The objective of the lab is to find the search algorithms necessary to perform the behavior of a simple Pacman using Berkely Pacman and determine heuristics derivations.

## Introduction

We can find a Pacman Project with all the specifications and the files we need to complete. We had to fill some algorithms in search.py and searchAgents.py, that intectue with our main class pacman.py.

## Search

This file stores some of the searching algorithms that we seen in theory classes. Following that, a possible solution that we find was:

1. **depthFirstSearch (DFS)**

```
def depthFirstSearch(problem):

    first = (problem.getStartState(), None, 1)#Estado inicial (5,5)
    s = util.Stack()                          #Creamos una pila s (util.Stack contine funcio
    s.push(first)                             #Agregamos origen a la pila s
    visit = []                                #Nodos visitados
    path = []                                 #Ruta a devolver
    adyac = {}                                #Nodos adyacentes

    while not s.isEmpty():
        v = s.pop()                           #sacamos un elemento de la pila s llamado v

        if not v[0] in visit:
            if problem.isGoalState(v[0]):     #Si es el objectivo delvolvemos la ruta

                while v[0] != first[0]:
                    path.append(v[1])         #Introducimos ruta de direcciones  en p
                    v = adyac[v]              #El nodo v ahora sera el padre

                path.reverse()
                return path                   #Devolvemos ruta al reves

            visit.append(v[0])
            successors = problem.getSuccessors(v[0])
            for suc in successors:            #recorremos los nodos sucesores
                if suc[0] not in visit:
                    s.push(suc)               #Anadimos nodos no visitados a la pila
                    adyac[suc] = v            #Anadimos el nodo a sus hijos

    return []
```

We use a Stack to save the newest nodes before the oldest. The aim is to find the deepest nodes and give a list of actions.

Diego Sáez Mulero U137519
Amanda Pintado Lineros U137702

## 2. breadthFirstSearch (BFS)

```python
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    #BFS es el mismo concepto que el DFS solo que cambia la Pila por la Cola, asi hara la

    first = (problem.getStartState(), None, 1)#Estado inicial
    s = util.Queue()                            #Creamos una cola s (util.Queue contine funci
    s.push(first)                               #Agregamos origen a la cola s
    visit = []                                  #Nodos visitados
    path = []                                   #Ruta a devolver
    adyac = {}                                  #Nodos adyacentes

    while not s.isEmpty():
        v = s.pop()                             #sacamos un elemento de la cola s llamado v

        if not v[0] in visit:
            if problem.isGoalState(v[0]):       #Si es el objectivo delvolvemos la ruta

                while v[0] != first[0]:
                    path.append(v[1])           #Introducimos ruta de direcciones  en p
                    v = adyac[v]                #El nodo v ahora sera el padre

                path.reverse()
                return path                     #Devolvemos ruta al reves

            visit.append(v[0])
            successors = problem.getSuccessors(v[0])
            for suc in successors:              #recorremos los nodos sucesores
                if suc[0] not in visit:
                    s.push(suc)                 #Anadimos nodos no visitados a la cola
                    adyac[suc] = v              #Anadimos el nodo a sus hijos

    return []
```

This algorithm is very similar to the previous one, the difference is that the visitor nodes in amplitude using a queue instead of a stack.

In practice, we has been created a queue where we will use the queue functions provided by the "util.py" file. The rest of the code is the same.

## 3. uniformCostSearch (UCS)

```python
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"

    #El objectivo es encontrar una ruta al nodo de destino que tenga el coste acumulativo mas bu


    first = ((problem.getStartState(), None, 0), 0)  #Estado inicial, costo inicial 0

    s = util.PriorityQueue()                        #Creamos una cola de prioridades s
    s.push(first,0)                                 #Agregamos origen a la cola s y prioridad

    visit = []                                      #Nodos visitados
    path = []                                       #Ruta a devolver
    adyac = {}


    while not s.isEmpty():

        v, p = s.pop()                        #sacamos un elemento de la cola y su prioridad p

        if not v[0] in visit:
            if problem.isGoalState(v[0]):   #Si es el objectivo delvolvemos la ruta

                while v[0] != first[0][0]:
                    path.append(v[1])       #Introducimos ruta de direcciones  en p
                    v = adyac[v]            #El nodo v ahora sera el padre

                path.reverse()
                return path                 #Devolvemos ruta al reves

            visit.append(v[0])
            successors = problem.getSuccessors(v[0])
            for suc in successors:          #recorremos los nodos sucesores
                if suc[0] not in visit:

                    cost= suc[2] + p        #Coste sucesor + coste nodo padre
                    s.push((suc,cost),cost) #Anadimos nodos no visitados a la cola y su coste acu
                    adyac[suc] = v          #Anadimos el nodo a sus hijos

    return []
```

For this algorithm we have to find a path of cost for the deepest nodes, so the main challenge compared to the other algorithms is to store this value.

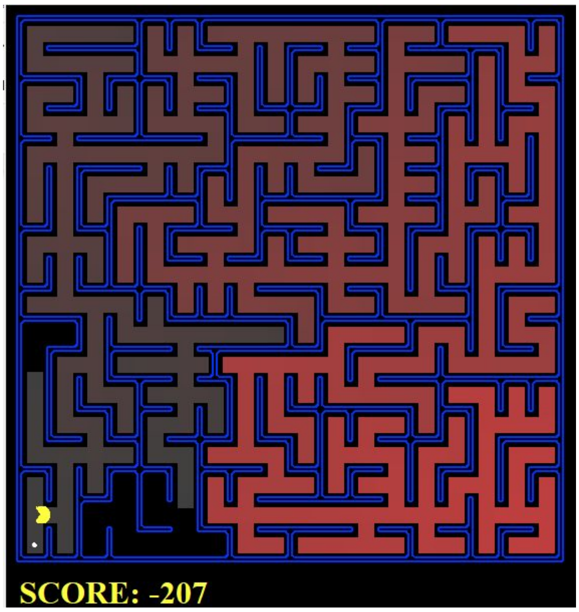
We use a queue that will contain the state and its priority. At the beginning, the priority will be 0 and as we go to the successor nodes, the cost will increase and add to the queue.

### 4. aStarSearch (A*)

```python
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"

    #El algoritmo tiene en cuenta el coste del camino recorrido y el coste de la heuristica (esti

    first = ((problem.getStartState(), None, 0), 0)  #Estado inicial, costo inicial 0

    s = util.PriorityQueue()                        #Creamos una cola de prioridades s
    s.push(first,0)                                 #Agregamos origen a la cola s y prioridad

    visit = []                                      #Nodos visitados
    path = []                                       #Ruta a devolver
    adyac = {}

    while not s.isEmpty():

        v, p = s.pop()                        #sacamos un elemento de la cola y su prioridad p

        if not v[0] in visit:
            if problem.isGoalState(v[0]):   #Si es el objectivo delvolvemos la ruta

                while v[0] != first[0][0]:
                    path.append(v[1])       #Introducimos ruta de direcciones  en p
                    v = adyac[v]            #El nodo v ahora sera el padre

                path.reverse()
                return path                 #Devolvemos ruta al reves

            visit.append(v[0])
            successors = problem.getSuccessors(v[0])
            for suc in successors:          #recorremos los nodos sucesores
                if suc[0] not in visit:
                    cost= suc[2] + p        #Coste sucesor + coste nodo padre
                    s.push((suc,cost),cost + heuristic(suc[0],problem)) #Anadimos nodos no visitc
                    adyac[suc] = v          #Anadimos el nodo a sus hijos

    return []
```

As with the uniformCostSearch algorithm, the main issue is to store not only the cost of the actions as well as to store the value of the heuristic function.

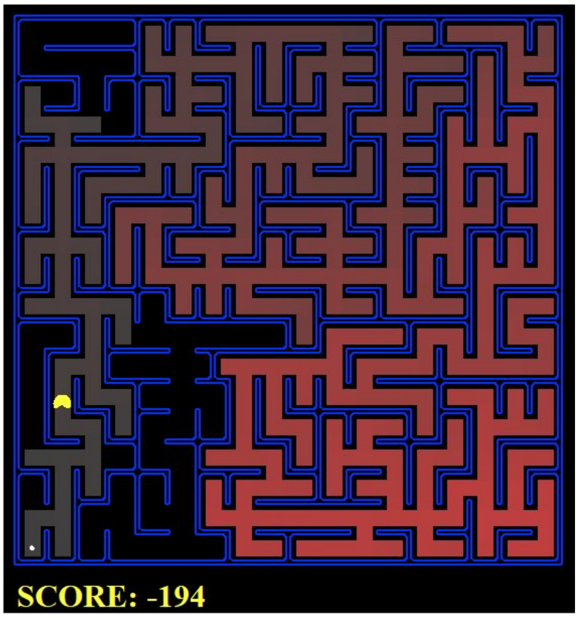
**Uniform Cost Search (UCS)**




Search nodes expanded: 620

```
[SearchAgent] using function ucs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 seconds
Search nodes expanded: 620
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

**A* Search**




Search nodes expanded: 549

```
[SearchAgent] using function astar and heuristic manhattanHeuristic
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 seconds
Search nodes expanded: 549
Pacman emerges victorious! Score: 300
Average Score: 300.0
Scores:        300.0
Win Rate:      1/1 (1.00)
Record:        Win
```

## SearchAgents

This file stores all the agents that can be controlled on the program. SearchAgent finds a route using algorithms of search and returns actions to follow that path.

## 5. Finding All the corners

Our goal now is to find the shortest path between 4 corners, whether with food or not. To do this, we implement the class *CornersProblem* that is responsible for finding the path of the 4 corners. The algorithm used in this section is the BFS of the search.

Diego Sáez Mulero U137519
Amanda Pintado Lineros U137702

Our implementation looks like:

### a. startState getter

```python
    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state
        space)
        """
        "*** YOUR CODE HERE ***"
        posIni = self.startingPosition #Posicion inicial
        posCorners = self.corners      #Posiciones de las esquinas del juego

        return posIni,posCorners

        util.raiseNotDefined()
```

Make a return of the start state of the class.

### b. isGoalState

```python
    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
        "*** YOUR CODE HERE ***"

        if not state[1]:
            return True
        return False

        util.raiseNotDefined()
```

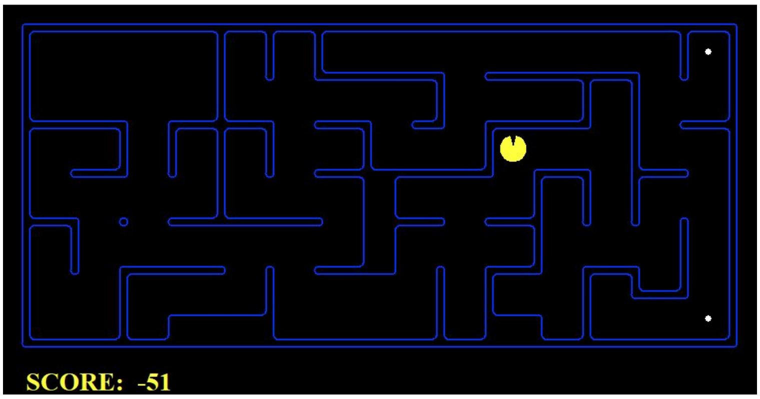
To figure out if the the state inserted as a parameter is a goal

### c. getSuccessors

```python
    def getSuccessors(self, state):

        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
            # Add a successor state to the successor list if the action is legal
            # Here's a code snippet for figuring out whether a new position hits a wall:
            #   x,y = currentPosition
            #   dx, dy = Actions.directionToVector(action)
            #   nextx, nexty = int(x + dx), int(y + dy)
            #   hitsWall = self.walls[nextx][nexty]

            "*** YOUR CODE HERE ***"


            x,y = state[0]                                  #Estado actual
            dx, dy = Actions.directionToVector(action)      #Direccion de la accion
            nextx, nexty = int(x + dx), int(y + dy)         #Nueva posicion
            hitsWall = self.walls[nextx][nexty]
            if not hitsWall:                                #Miramos que la siguiente pos. no s

                food = filter(lambda a: a!= (nextx,nexty), state[1]) #Filtramos y devolvemos la
                successors.append((((nextx,nexty),food),action,1))

        self._expanded += 1 # DO NOT CHANGE
        return successors
```

Where we must find the successors of a state, the actions and the cost.

Diego Sáez Mulero U137519
Amanda Pintado Lineros U137702




## 6. Corners Heuristic

```
def cornersHeuristic(state, problem):

    corners = problem.corners # These are the corner coordinates
    walls = problem.walls # These are the walls of the maze, as a Grid (game.py)

    "*** YOUR CODE HERE ***"

    posState = state[0]
    food = state[1]
    h = 0
    while food:                                             #Mientras haya comida

        near=999999
        pos=0

        for f in food:                                      #Para cada pos. donde hay
            dist = util.manhattanDistance(posState, f)      #Calculamos la distacia (
            if dist < near:
                near = dist
                pos = f

        h = h + near                                        #Incrementamos el coste p
        posState = pos                                      #Posicion comida
        food = filter(lambda el: el != posState, food)      #filtaramos y eliminamos

    return h #Devuleve el coste del camino mas corto
```

Returns a number that will always be lower from the current state to the goal. The returned heuristic will always be admissible.

## 7. Eating All The dots

```python
def foodHeuristic(state, problem):

    position, foodGrid = state

    "*** YOUR CODE HERE ***"

    heuristic = 0
    foodList = foodGrid.asList() #Lista de coordenadas food

    if not foodList:
        return 0

    maxDist = 0
    for f in foodList: #para cada pos. de la comida, cojemos la distancia mas c
        dist = mazeDistance(position, f, problem.startingGameState)#Distancia e
        if dist > maxDist:
            maxDist = dist #Actualizamos si la distancia de esta food es mayor

    return maxDist #Devolvemos distancia de la comida mas lejana
```

To find the further capsule, using List and a heuristic value.

## 8. Suboptimal Search

### a. findPathToClosesDot

```python
    def findPathToClosestDot(self, gameState):
        """
        Returns a path (a list of actions) to the closest dot, starting from
        gameState.
        """
        # Here are some useful elements of the startState
        startPosition = gameState.getPacmanPosition()
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)


        "*** YOUR CODE HERE ***"

        return search.aStarSearch(problem)
        util.raiseNotDefined()
```

Using our previous aStarSearch algorithm, we return a path to find the closest dot (or capsule).

A* is capable of finding the solution having to expand slightly less nodes than BFS.

Used why findPathToClosestDot, is necessary to implement a class called AnyFoodSearchProblem to find out paths to any food, as the name can suggest.

### b. isGoalState

```
    def isGoalState(self, state):
        """
        The state is Pacman's position. Fill this in with a goal test that will
        complete the problem definition.
        """
        x,y = state

        "*** YOUR CODE HERE ***"

        food = self.food.asList()

        return (x, y) in food
        util.raiseNotDefined()
```

Similar to the implementation for the previous class.

### Conclusions and issues

For search.py file we have to implement algorithms that we see briefly in previous theory sessions.