UADE

Facultad de Ingeniería y Ciencias Exactas

Inteligencia Artificial

Trabajo Práctico 2:

Algoritmos de búsqueda

Profesor:

Christian Parkinson

Integrantes - Grupo 6		
Apellido y Nombre	**Legajo**	**Carrera**
Breuer, Andrés	1120248	Ing. en Informatica
Cabezas, Alexander	1086279	Ing. en Informatica
Cappellano, María	1056140	Ing. en Informatica
Caneva, Matheo	1129004	Ing. en Informatica
Facón, Nicolas	1133988	Ing. en Informatica

Índice

Introducción

El siguiente trabajo está enfocado en los algoritmos de búsquedas que se utilizan para encontrar caminos o posibles soluciones a problemas complejos.
Los algoritmos de búsqueda son como mapas mentales que nos guían desde un punto de inicio “S” hasta un destino “G” en un grafo. Estos algoritmos exploran diferentes caminos desde el punto de inicio, expandiendo opciones a medida que avanzan hacia el destino. Una vez que alcanzan el destino, siguen el camino de vuelta para reconstruir la ruta exacta. En este trabajo práctico, lo vamos a utilizar para encontrar la mejor manera para que el Pacman se mueva en un laberinto y alcance su objetivo, como comer todos los puntos.

Partimos del Pacman encontrado en el siguiente repositorio de Github
https://github.com/thiadeliria/Pacman

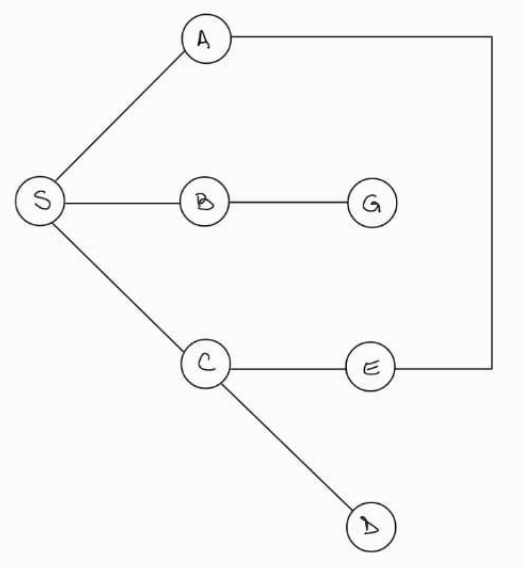
El repositorio ni bien lo clonas tiene faltantes de paréntesis en funciones print y excepciones, las cuales debimos corregir para poder compilarlo.

Además, creamos un archivo launch.json para poder debuggear el código y tener distintos argumentos de línea de comandos para los distintos tipos de algoritmos.

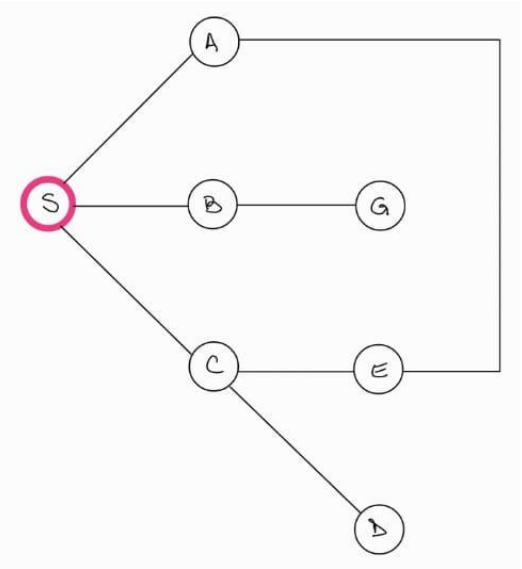
Búsqueda en Profundidad

La búsqueda en profundidad es un algoritmo que se utiliza para explorar y encontrar soluciones en estructuras de datos como grafos o árboles. Comienza desde un nodo inicial y luego se mueve hacia abajo, explorando todos los caminos posibles lo más lejos posible antes de retroceder.
En resumen, la búsqueda en profundidad es un proceso de exploración recursiva que sigue un camino hasta el nivel más bajo posible antes de retroceder y probar otro camino, priorizando la profundidad sobre la amplitud en la exploración de un grafo o estructura similar.

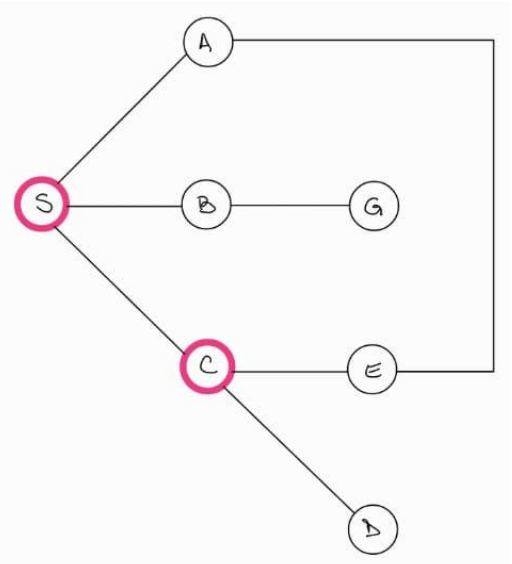
Los pasos que realiza la búsqueda en profundidad son:



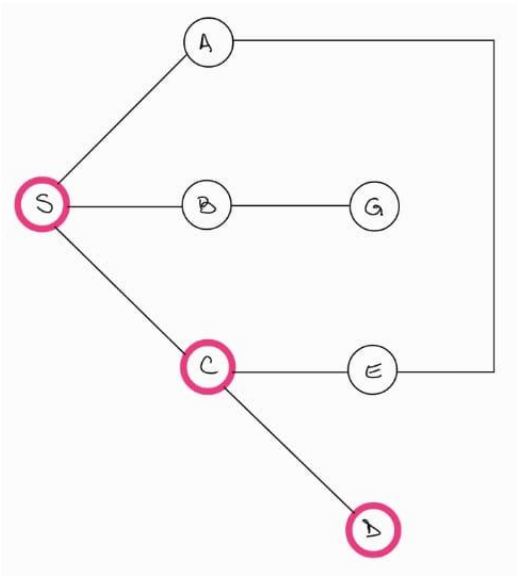
1- Se selecciona el nodo de inicio para comenzar la búsqueda. Se verifica si es el nodo objetivo que se está buscando. Si es el objetivo, el proceso de búsqueda finaliza acá. Si no es el objetivo, continúa al siguiente paso.



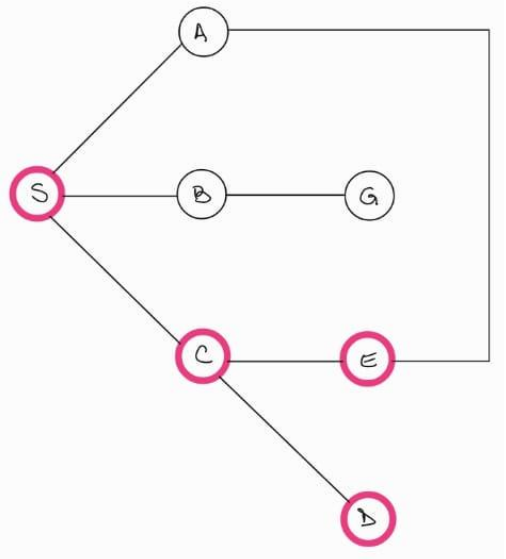
2- Examina los nodos conectados al nodo inicial. Escoge un nodo adyacente y avanza hacia él.

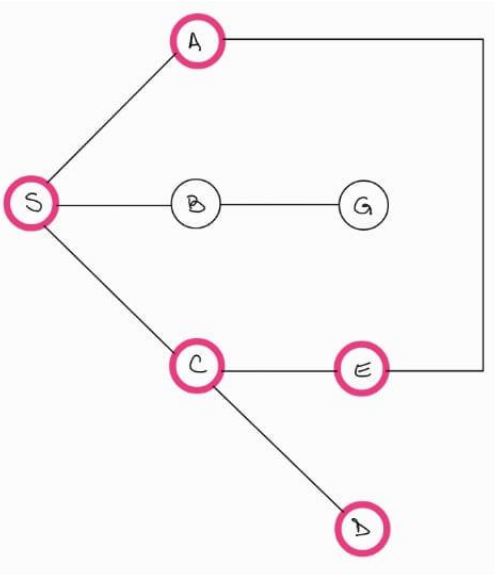


3- Una vez está en el nodo subyacente, repite los pasos 1 y 2 para este nuevo nodo. Explora recursivamente hacia abajo por cada rama del grafo o estructura de datos, priorizando los nodos más recientes visitados.

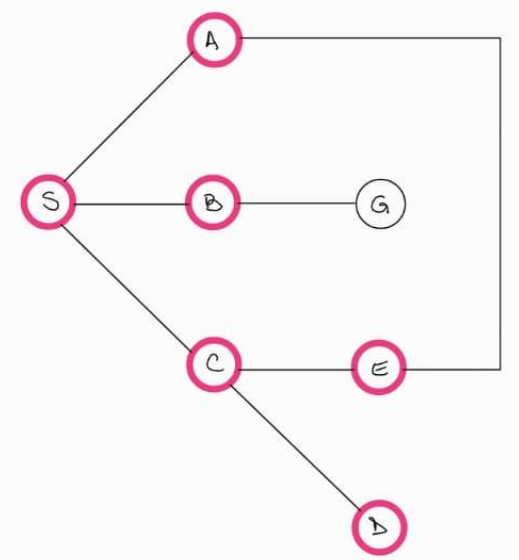


4- Cuando alcanza un punto donde no haya más nodos adyacentes por explorar, retrocede al nodo anterior y busca otros nodos aún no visitados desde allí.

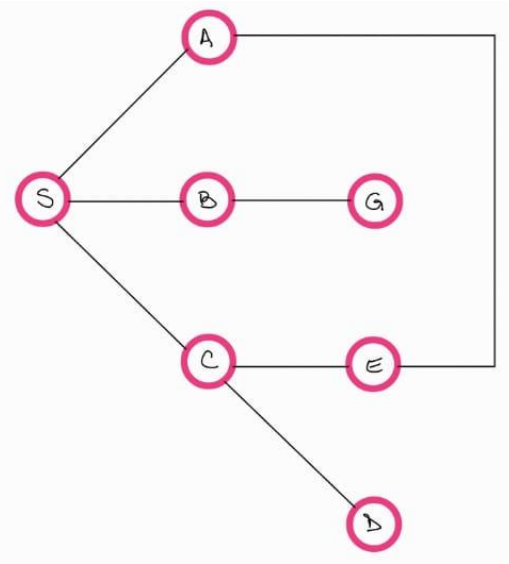




5- Continúa este ciclo de exploración y retroceso hasta encontrar el nodo objetivo o haber explorado todo el grafo.



6- Si encuentra el nodo objetivo, sigue el camino de vuelta desde ese nodo hasta el nodo inicial para determinar la solución real.



Solución: S - C - D - E - A - B - G

Búsqueda en Profundidad - Implementación

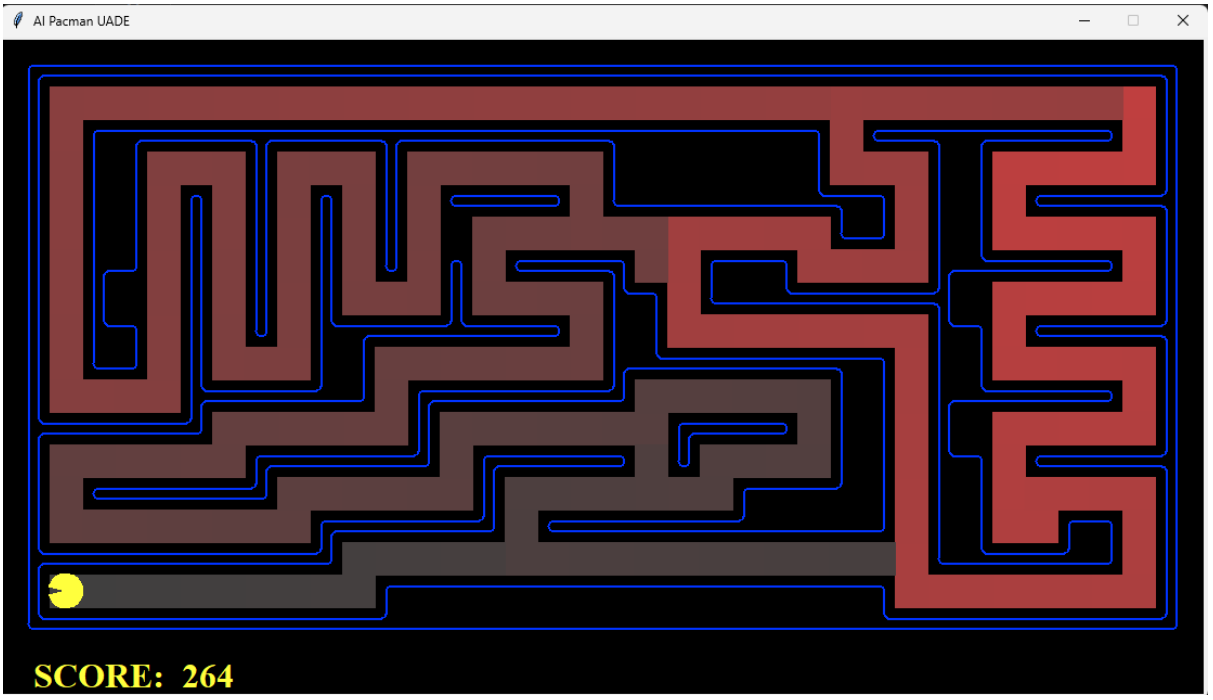
En la implementación, primero se ve el estado inicial, luego está el recorrido y finalmente la reconstrucción del camino.

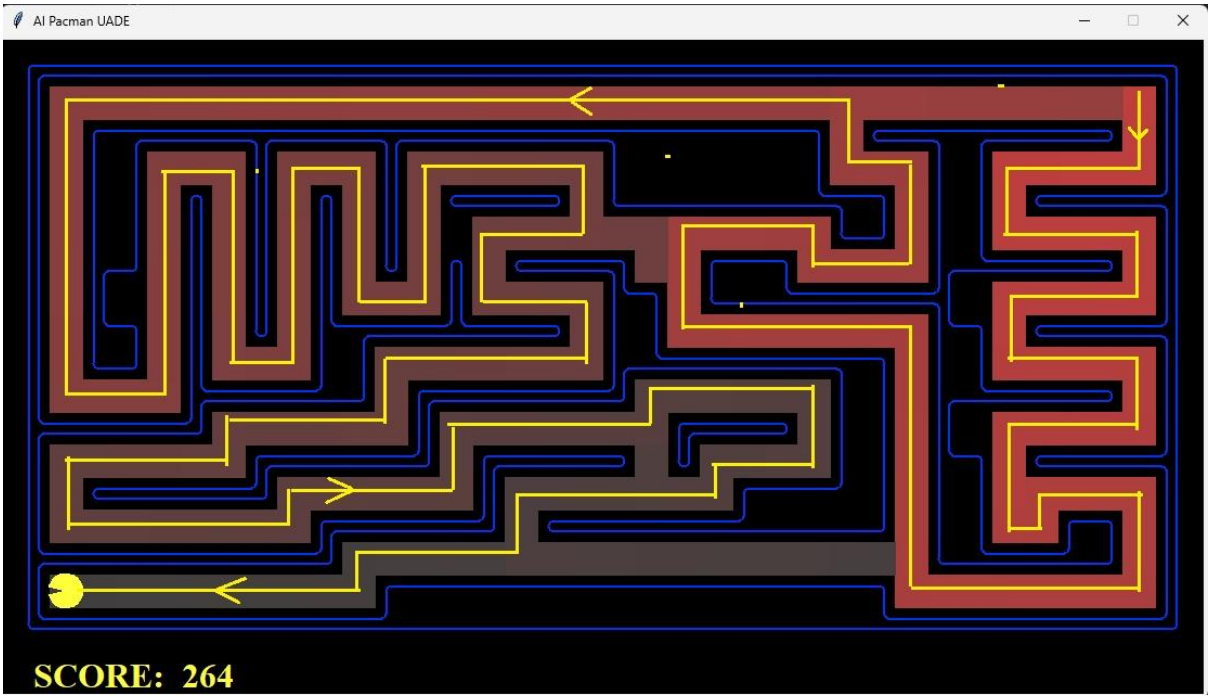
```
def depthFirstSearch(problem):
    # Genero una función recursiva que realice búsqueda en profundidad
    def recursiveDFS(currentState, directions, exploredNodes):
        if currentState in exploredNodes:
            return None  # Ya usé este nodo, no lo repito.

        exploredNodes.append(currentState)

        if problem.isGoalState(currentState):
            return directions

        successors = problem.getSuccessors(currentState)

        for nextCoords, nextDirection, nextCost in successors:
            result = recursiveDFS(nextCoords, directions +
[nextDirection], exploredNodes)

            if result is not None:
                return result

        return None
```

```
    startState = problem.getStartState()
    exploredNodes = []

    return recursiveDFS(startState, [], exploredNodes)
```

La configuración del launch.json utilizada fue:

```
{
            "name": "DFS - Búsqueda en profundidad",
            "type": "python",
            "request": "launch",
            "program": "pacman.py",
            "console": "integratedTerminal",
            "args": ["-l", "mediumMaze", "-p", "SearchAgent", "-a",
"fn=dfs"],
        },
```


AI Pacman UADE
SCORE: 264


AI Pacman UADE
SCORE: 264

Búsqueda en Anchura

La búsqueda en anchura es un algoritmo utilizado para explorar y encontrar soluciones en estructuras de datos como grafos o árboles. Comienza desde un nodo inicial, explora todos los nodos vecinos del nodo actual antes de pasar a los nodos más lejanos.
En resumen, la búsqueda en anchura explora gradualmente todos los nodos en niveles sucesivos a partir del nodo inicial, visitando primero todos los nodos a una distancia de un paso, luego de dos pasos, y así sucesivamente, hasta encontrar el nodo objetivo o haber explorado todos los nodos en el grafo.

Los pasos que realiza la búsqueda en anchura son:

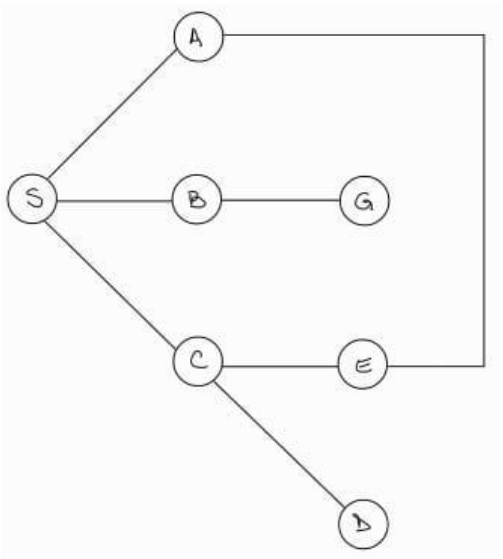
1- Se selecciona el nodo de inicio para comenzar la búsqueda. Se verifica si es el nodo objetivo que se está buscando. Si es el objetivo, el proceso de búsqueda finaliza acá. Si no es el objetivo, continúa al siguiente paso.

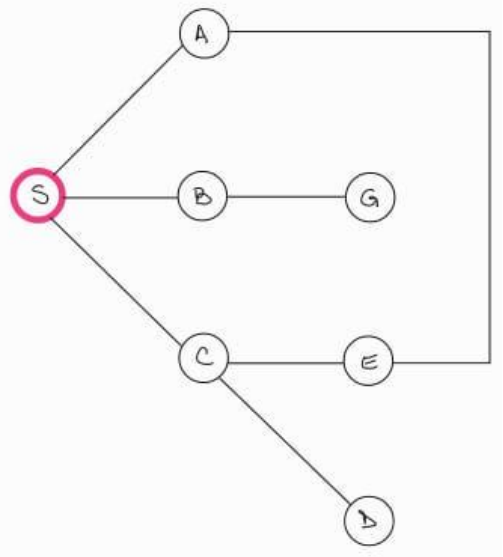
2- Visita todos los nodos adyacentes al nodo inicial. A medida que se visita cada nodo, se lo va marcando como visitado para evitar volver a visitarlo y evitar ciclos infinitos.

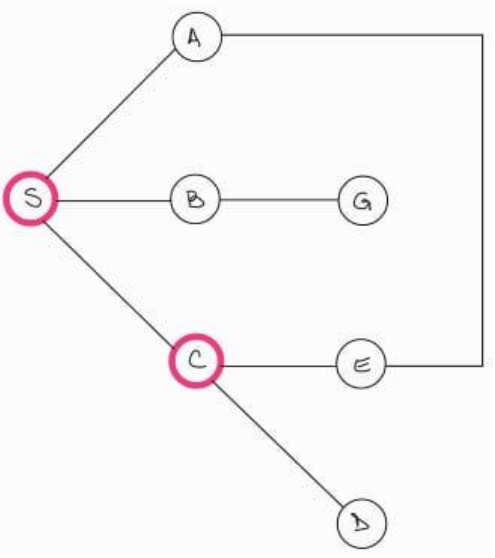
3- Una vez que haya visitado los nodos vecinos del nodo inicial, pasa a explorar los nodos adyacentes a esos nodos visitados. En este momento se exploran los nodos que están a una distancia de dos pasos desde el nodo inicial.

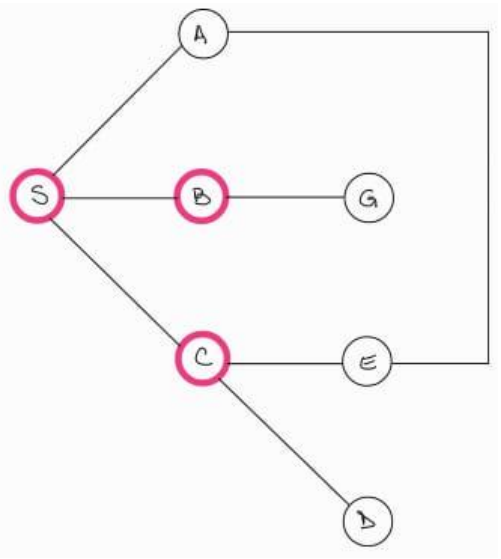
4- Continúa este proceso, explorando los nodos a distancias cada vez mayores desde el nodo inicial, nivel por nivel. Primero fueron todos los nodos a una distancia de un paso, luego a una distancia de dos pasos, luego de tres y así sucesivamente.

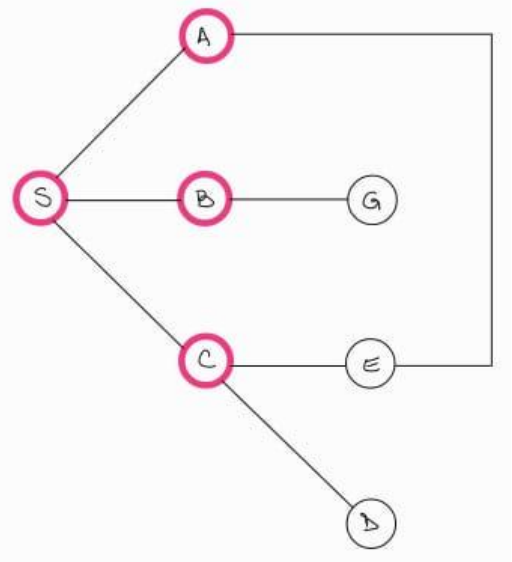
5- Una vez se encuentra el nodo objetivo, sigue el camino de vuelta desde ese nodo hasta el nodo inicial para determinar la solución o el camino más corto.

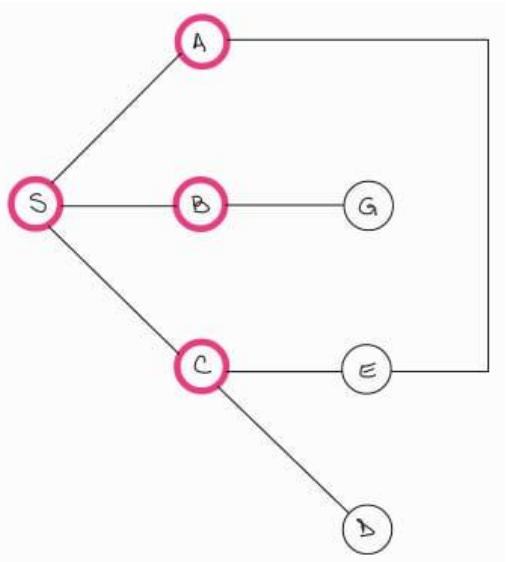

A
S
B
G
C
E
D

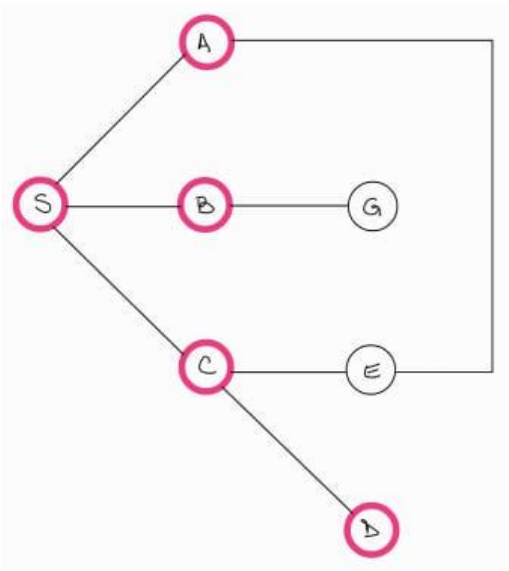

A
S
B
G
C
E
D

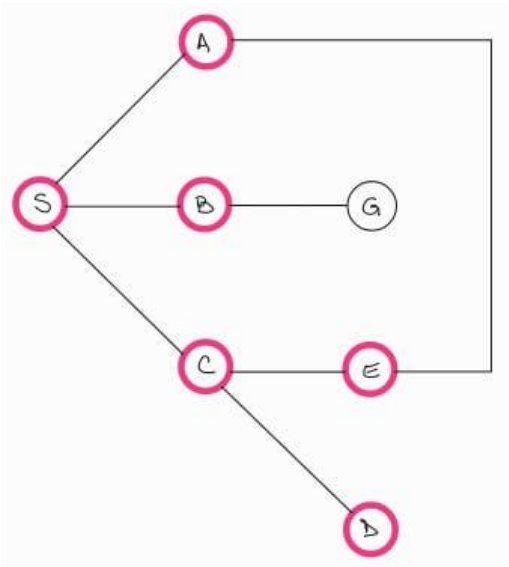

A
S
B
G
C
E
D

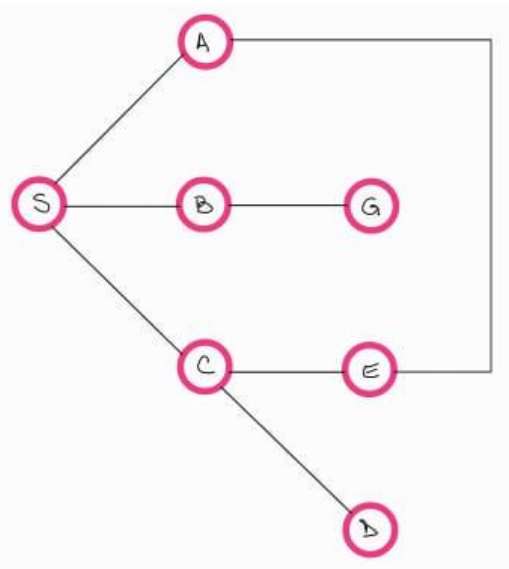

A
S
B
G
C
E
D


A
S
B
G
C
E
D


A
S
B
G
C
E
D


A
S
B
G
C
E
D


A
S
B
G
C
E
D


A
S
B
G
C
E
D

Búsqueda en Anchura - Implementación

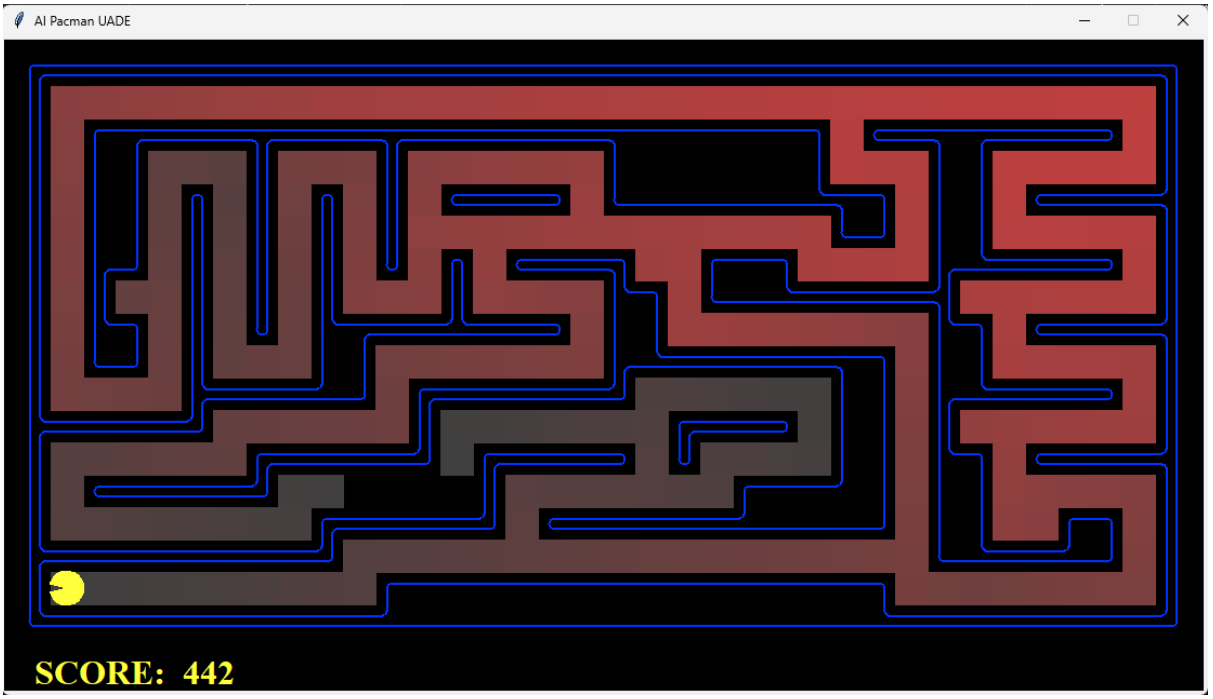
En la implementación, primero se puede apreciar el estado inicial, luego está el recorrido y finalmente la reconstrucción del camino.

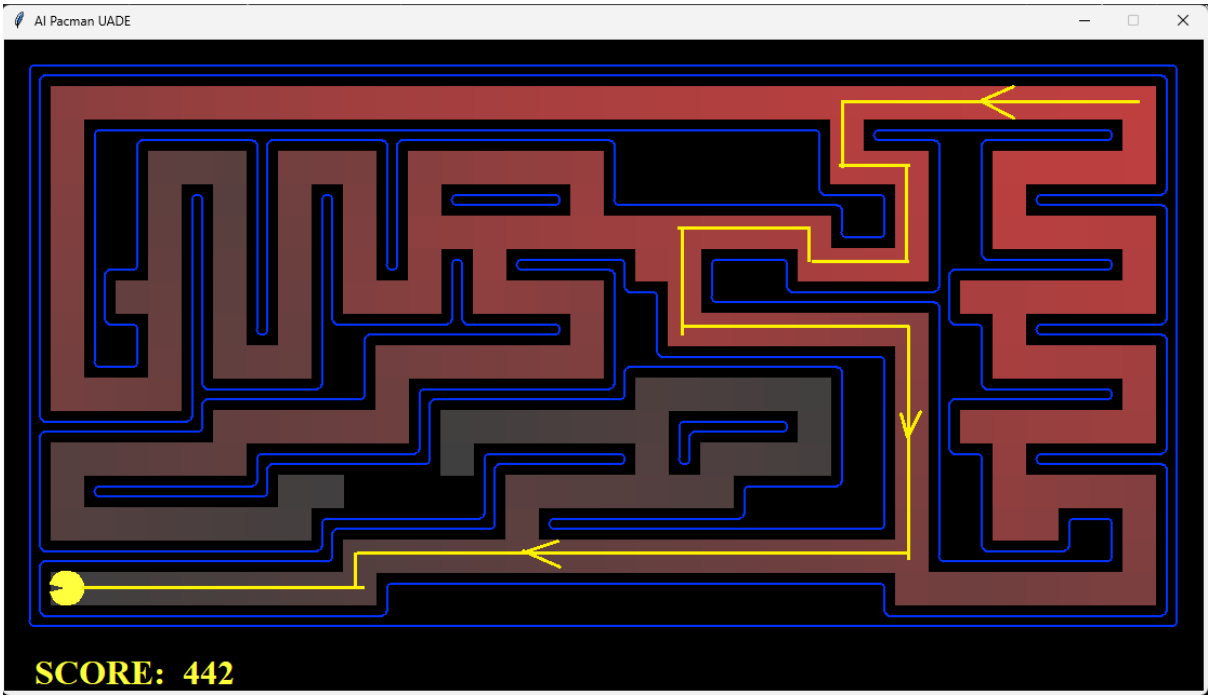
```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    def recursiveBFS(nodes, exploredNodes):
        if len(nodes) == 0:
            return None  # Primero reviso tener nodos para evaluar

        currentState, directions = nodes.pop(0)

        if currentState in exploredNodes:
            return recursiveBFS(nodes, exploredNodes) # Ya evalué este
nodo

        exploredNodes.append(currentState)

        if problem.isGoalState(currentState):
            return directions

        successors = problem.getSuccessors(currentState)

        for nextCoords, nextDirection, nextCost in successors:
            newNode = (nextCoords, directions + [nextDirection])

            nodes.append(newNode)

        return recursiveBFS(nodes, exploredNodes)

    startState = problem.getStartState()
    nodes = []
    exploredNodes = []

    nodes.append((startState, []))

    return recursiveBFS(nodes, exploredNodes)
```

La configuración del launch.json utilizada fue:

```
{
            "name": "BFS - Breadth First",
            "type": "python",
```

```
            "request": "launch",
            "program": "pacman.py",
            "console": "integratedTerminal",
            "args": ["-l", "mediumMaze", "-p", "SearchAgent", "-a",
"fn=bfs"],
        },
```





Búsqueda con Costo Uniforme

La búsqueda con costo uniforme es un algoritmo que se utiliza para encontrar la ruta más económica desde un nodo inicio a un nodo objetivo en un grafo. A diferencia de otros algoritmos que priorizan la distancia en número de pasos, este algoritmo se centra en el costo real asociado con cada paso o movimiento entre nodos. Funciona expandiendo gradualmente hacia los nodos adyacentes desde el nodo inicial, seleccionando la ruta que tiene el costo acumulado más bajo hasta el momento. Esto asegura que se encuentre la ruta que tiene el menor costo total, teniendo en cuenta los valores asignados a cada paso o movimiento entre nodos en lugar de simplemente la cantidad de pasos.
En resumen, la búsqueda de costo uniforme prioriza la exploración de rutas que tienen un costo menos acumulado para llegar a cada nodo, asegurando que se encuentre la ruta más económica desde el nodo inicial hasta el nodo objetivo en un grado ponderado.

Los pasos que realiza la búsqueda con costo uniforme son:

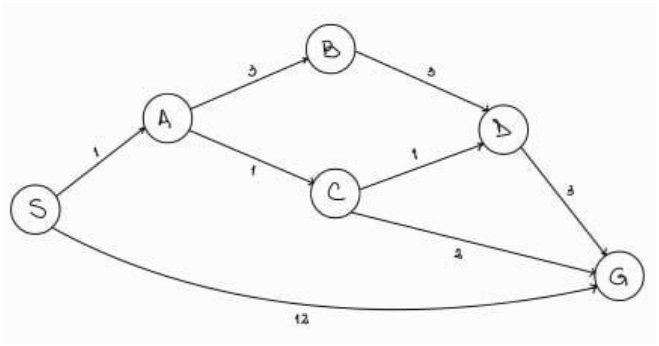
1- Se selecciona el nodo de inicio para comenzar la búsqueda. Se mantiene un registro de los nodos visitados y los costos asociados con llegar a cada nodo desde el nodo inicial. El costo inicial es 0, y el costo de los demás nodos se establece inicialmente en infinito o un valor alto para representar que aún no se ha calculado el costo.

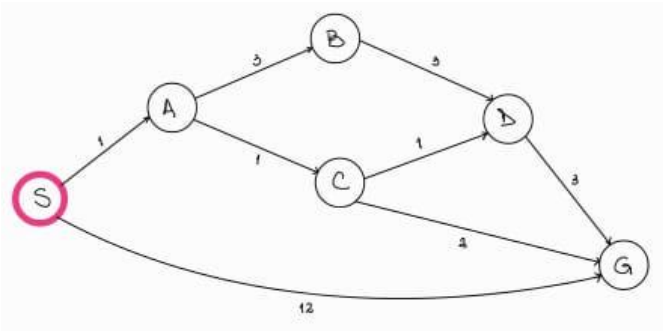
2- Se examinan los nodos adyacentes al nodo inicial y calcula el costo para alcanzar cada uno de ellos desde el nodo inicial. Actualiza los costos si se encuentra una ruta más económica para llegar a un nodo en comparación con la ruta previamente calculada.

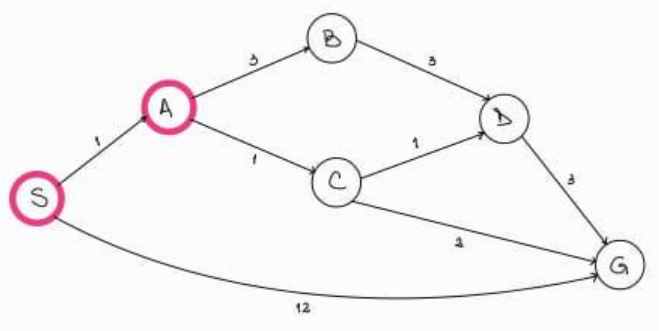
3- Se encuentra el nodo con el costo más bajo entre los nodos adyacentes no visitados. Este nodo será el siguiente en ser explorado.

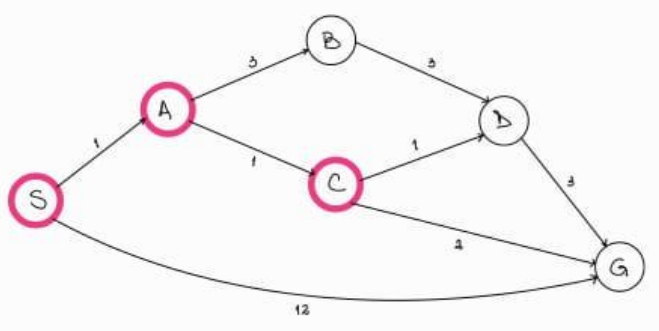
4- Continúa expandiendo hacia los nodos adyacentes del nodo con el costo más bajo y actualizando los costos hasta llegar al nodo objetivo o hasta que se hayan explorado todos los nodos alcanzables.

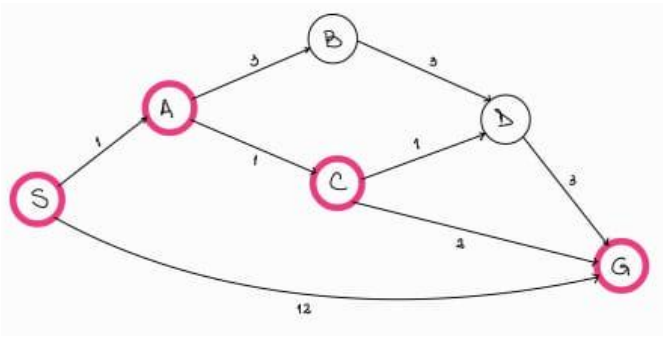
5- Una vez que se alcanza el nodo objetivo, se puede rastrear el camino más económico desde el nodo inicial hasta el nodo objetivo siguiendo los registros de los costos asociados con cada nodo visitado.

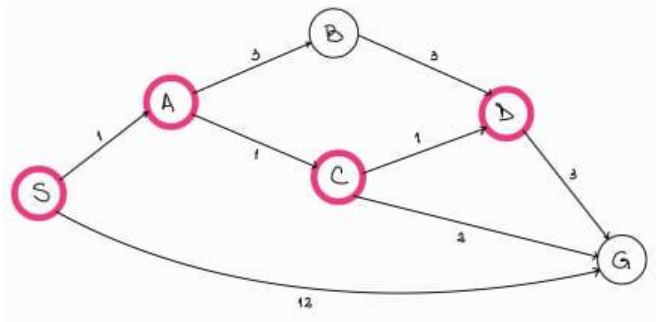

B
3
3
A
D
1
1
1
3
S
C
4
G
12

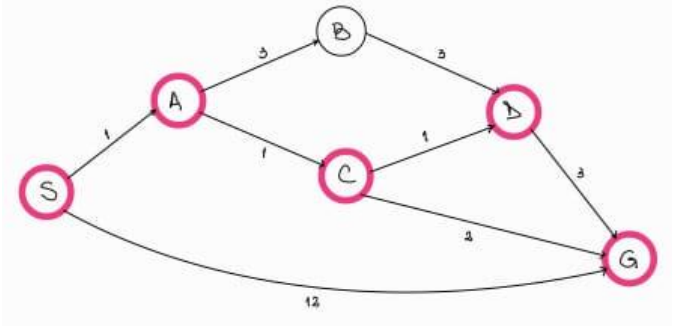

B
3
3
A
D
1
1
1
3
S
C
4
G
12


B
3
3
A
D
1
1
1
3
S
C
4
G
12


B
3
3
A
D
1
1
1
3
S
C
4
G
12







Solución: S - A - C - G

Búsqueda con Costo Uniforme- Implementación

En la implementación, primero se puede apreciar el estado inicial, luego está el recorrido y finalmente la reconstrucción del camino.

```
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    def recursiveUCS(nodes, exploredNodes):
        if len(nodes) == 0:
            return None  # Primero reviso tener nodos para evaluar

        nodes.sort(key=lambda x: x[2])  # Ordenar la lista por costo
        currentState, directions, currentCost = nodes.pop(0)
```

```
        if (currentState not in exploredNodes) or (currentCost <
exploredNodes[currentState]):
            exploredNodes[currentState] = currentCost

            if problem.isGoalState(currentState):
                return directions
            else:
                successors = problem.getSuccessors(currentState)

                for nextCoords, nextDirection, nextCost in successors:
                    newNode = (nextCoords, directions +
[nextDirection], currentCost + nextCost)

                    nodes.append(newNode)

                return recursiveUCS(nodes, exploredNodes)

        return recursiveUCS(nodes, exploredNodes)

    startState = problem.getStartState()
    nodes = [(startState, [], 0)]
    exploredNodes = {}

    return recursiveUCS(nodes, exploredNodes)
```

La configuración del launch.json utilizada fue:

```
{
            "name": "UCS - Costos uniformes",
            "type": "python",
            "request": "launch",
            "program": "pacman.py",
            "console": "integratedTerminal",
            "args": ["-l", "mediumMaze", "-p", "SearchAgent", "-a",
"fn=ucs"],
        },
```

Por el tamaño del mapa que podemos ver con nuestras pantallas, el recorrido es el mismo. Se aprecia en un mapa más grande que es un recorrido más óptimo que BFS, pero no podemos ver la ruta completa.

Búsqueda A*

La búsqueda A* es un algoritmo que encuentra el camino más corto entre un nodo de inicio y un nodo objetivo en un grafo. Utiliza dos tipos de información para decidir qué nodo explorar: el costo real desde el nodo inicial hasta el nodo objetivo (determinado costo G), y una estimación heurística del costo restante desde ese nodo hasta el destino (denominado costo H). A* evalúa y elige expandir los nodos siguiendo una función de evaluación que es la suma del costo G y el costo H. De esta manera, prioriza la exploración de los nodos que tienen una función de evaluación menor, lo que suele llevar a una búsqueda eficiente del camino más corto hacia el nodo objetivo.
En resumen, la búsqueda A* evalúa y selecciona nodos basándose en la combinación de un costo real y una estimación heurística, priorizando la exploración de los nodos más prometedores para encontrar la ruta más corta hacia el nodo objetivo en un grafo.

Los pasos que realiza la búsqueda A* son:

1- Selecciona el nodo de inicio y el nodo objetivo para comenzar la búsqueda. Inicializa dos conjuntos: uno para los nodos visitados y otro para los nodos por visitar.

2- Calcula dos costos para cada nodo: el costo real “G” desde el nodo inicial hasta ese nodo y una estimación heurística “H” del costo restante desde ese nodo hasta el nodo objetivo.

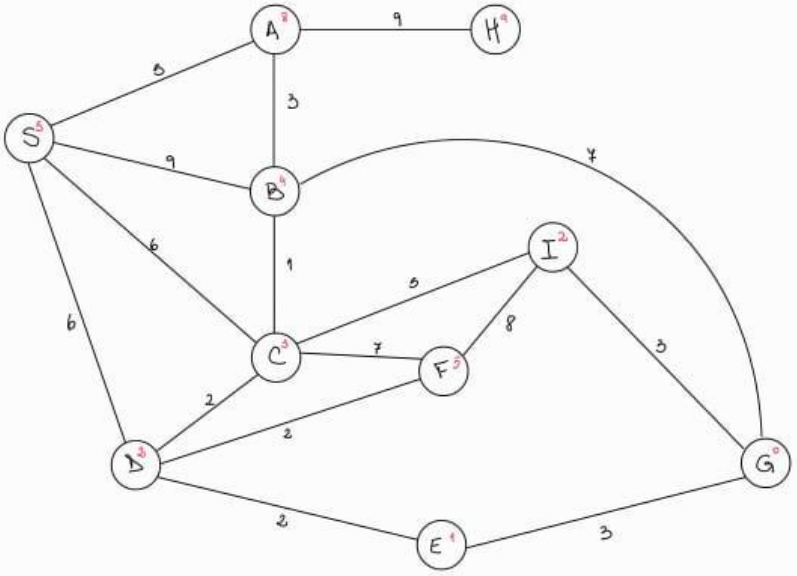
3- Para cada nodo, calcula la función de evaluación f, que es la suma del costo real “G” y el costo heurístico “H”. F = G + H

4- Elige el nodo con el menor valor de la función de evaluación f para ser expandido. Este nodo es considerado como el nodo más prometedor para llegar al objetivo.

5- Examina los nodos adyacentes al nodo seleccionado. Calcula y actualiza los valores de “G” y “F” para estos nodos, teniendo en cuenta su costo real y su estimación heurística.

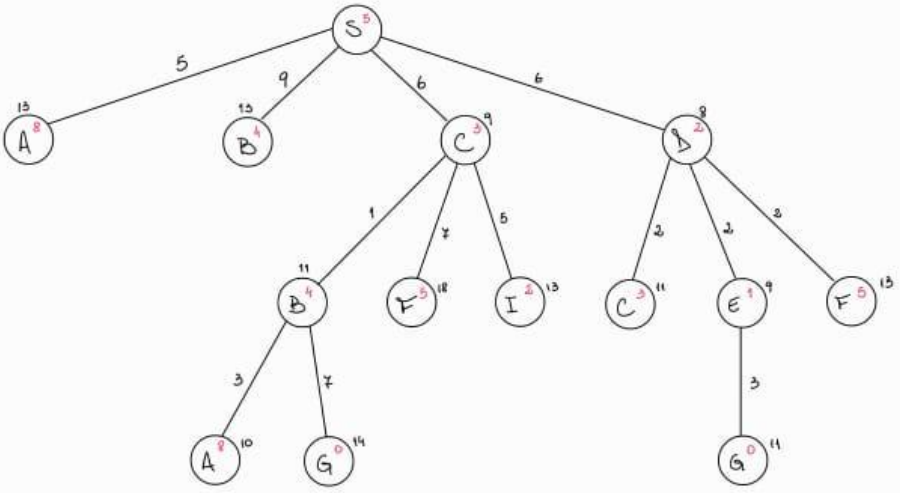
6- Agrega el nodo actual a los nodos visitados y continúa el proceso seleccionando el siguiente nodo a expandir, basándose en la función de evaluación “F” más baja entre los nodos por visitar.

7- Una vez que el nodo objetivo se coloca en el conjunto de nodos visitados, se ha encontrado la ruta óptima desde el nodo inicial al nodo objetivo. Se puede reconstruir esta ruta siguiendo los padres de los nodos desde el nodo objetivo hasta el nodo inicial.

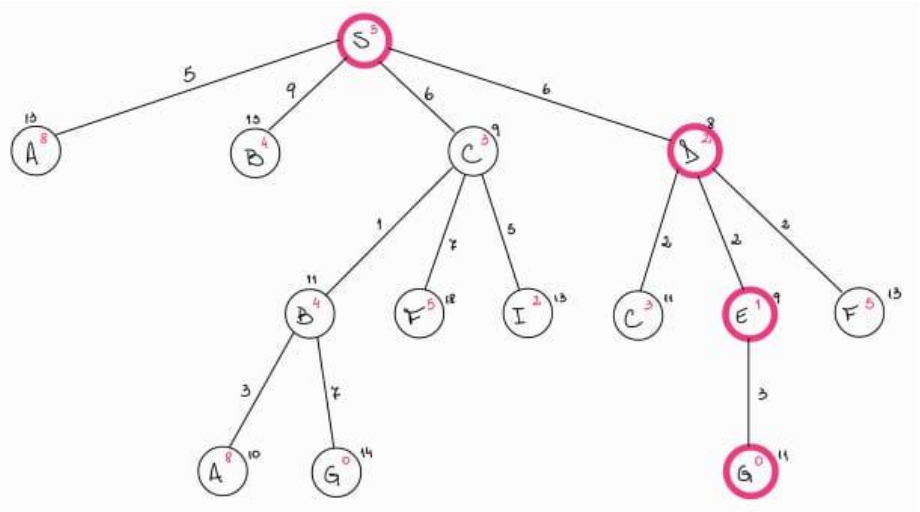


A* → $f(n) = g(n) + h(n)$ → ADMISIBLES

- $f(n)$: A* score
- $g(n)$: Costo para llegar a n
- $h(n)$: ↳ costo estimado para ir de n al destino



VISITADOS: $S^5, D^8, C^9, E^9, B^{11}$



Solución: S - D - E - G

Búsqueda A*- Implementación

En la implementación, primero se puede apreciar el estado inicial, luego está el recorrido y finalmente la reconstrucción del camino.

```
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic
first."""
    def recursiveAStar(nodes, exploredNodes):
        if len(nodes) == 0:
            return None  # Primero reviso tener nodos para evaluar

        nodes.sort(key=lambda x: x[2] + heuristic(x[0], problem))  #
Ordenar la lista por costo + heurística
        currentState, directions, currentCost = nodes.pop(0)

        if currentState in exploredNodes and currentCost >=
exploredNodes[currentState]:
            return recursiveAStar(nodes, exploredNodes)

        exploredNodes[currentState] = currentCost

        if problem.isGoalState(currentState):
            return directions

        successors = problem.getSuccessors(currentState)

        for nextCoords, nextDirection, nextCost in successors:
```

```
            newNode = (nextCoords, directions + [nextDirection],
currentCost + nextCost)

            nodes.append(newNode)

        return recursiveAStar(nodes, exploredNodes)

    startState = problem.getStartState()
    nodes = [(startState, [], 0)]
    exploredNodes = {}

    return recursiveAStar(nodes, exploredNodes)
```

La configuración del launch.json utilizada fue:

```
{
            "name": "A*",
            "type": "python",
            "request": "launch",
            "program": "pacman.py",
            "console": "integratedTerminal",
            "args": ["-l", "mediumMaze", "-p", "SearchAgent", "-a",
"fn=astar"],
        }
```

Igual que como explicamos para UCS, por el tamaño del mapa que podemos ver con nuestras pantallas, el recorrido es el mismo que en BFS y UCS. Se aprecia en un mapa más grande que es un recorrido más óptimo que BFS, pero no podemos ver la ruta completa.