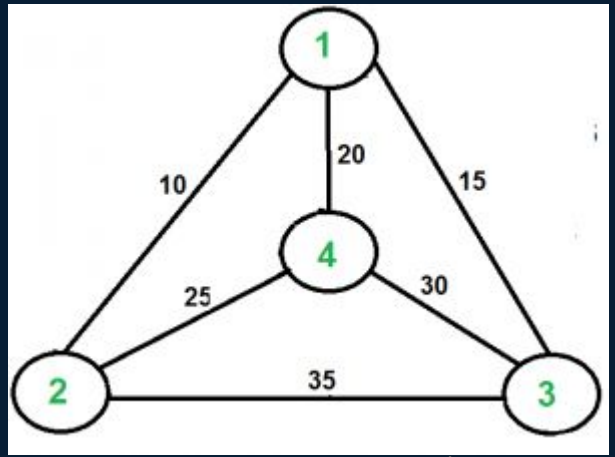
Poda y Ramificación

Problema del viajero

Problema del viajero (TCP)

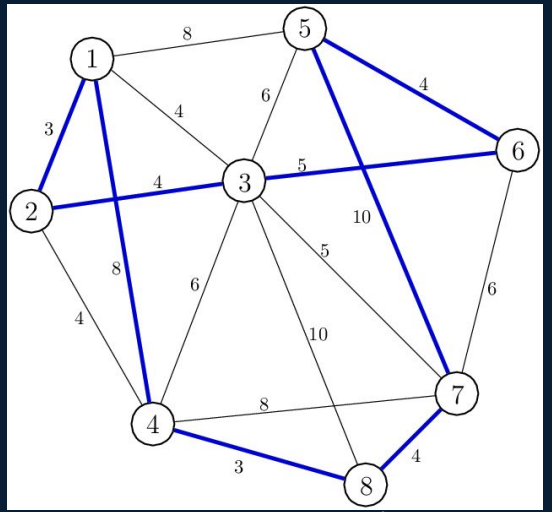
Dado un conjunto de ciudades y distancia entre cada par de ciudades, el problema es encontrar la ruta más corta posible que visite cada ciudad exactamente una vez y regrese al punto de partida.

El problema es un famoso problema NP-duro. No existe una solución conocida en tiempo polinómico para este problema.



Diferencia entre un ciclo Hamiltoniano

El problema del ciclo Hamiltoniano es encontrar si existe un recorrido que visita cada ciudad exactamente una vez. Aquí sabemos que un Tour Hamiltoniano existe (porque el grafo está completo) y de hecho, existen muchos de estos tours, el problema es encontrar un ciclo Hamiltoniano de peso mínimo.



Implementación Naive

1.- Considere la ciudad 1 como el punto de partida y fin.
Dado que la ruta es cíclica, podemos considerar cualquier punto como punto de partida.

2.- Generar todo (n-1)! permutaciones de ciudades.

3.- Calcule el costo de cada permutación y realice un seguimiento de la permutación del coste mínimo.
Devuelva la permutación con un coste mínimo.

```python
def travellingSalesmanProblem(graph, s, V):
    # almacenar todos los vértices aparte del vértice de origen
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)

    # almacenamos el peso mínimo del ciclo Hamiltoniano
    min_path = maxsize
    next_permutation=permutations(vertex)
    for i in next_permutation:
        # almacenamos el peso de la ruta actual (costo)
        current_pathweight = 0

        # calculamos el peso de la ruta actual
        k = s
        for j in i:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][s]

        # actualizamos el mínimo
        min_path = min(min_path, current_pathweight)

    return min_path
```

Complejidad del tiempo: (n!)

Ramificación y Poda (Branch y Bound)

Tenga en cuenta que el coste a través de un nodo incluye dos costes:

1) Costo de llegar al nodo desde la raíz (cuando llegamos a un nodo, tenemos este costo calculado)
2) Costo de alcanzar una respuesta desde el nodo actual a una hoja (calculamos un límite en este costo para decidir si ignorar el subárbol con este nodo o no).

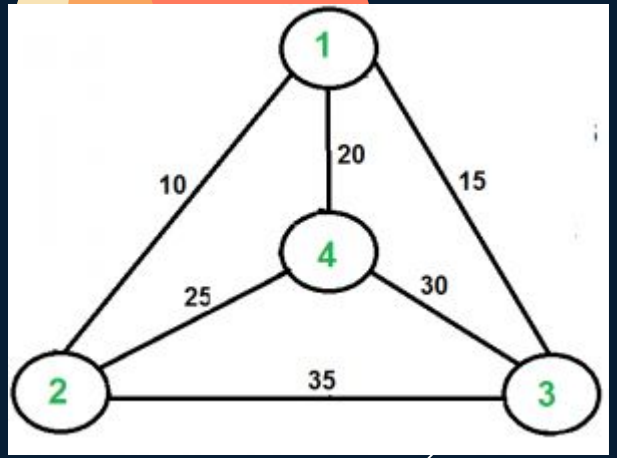
- En los casos de un problema de maximización, un límite superior nos indica la solución máxima posible si seguimos el nodo dado.
- En los casos de un problema de minimización, un límite inferior nos indica la solución mínima posible si seguimos el nodo dado.

Ramificación y Poda (Branch y Bound)

En la ramificación y poda, la parte difícil es averiguar una manera de calcular un límite en la mejor solución posible. A continuación se muestra una idea que se usa para calcular los límites para el problema del viajero. El costo de cualquier tour se puede escribir como se indica a continuación.

```
Costo de un recorrido T = (1/2) * ∑ (Suma del costo de dos aristas adyacente a u y en el
                                    recorrido T)
                                    donde u ∈ V

Para cada vértice u, si consideramos dos aristas que lo atraviesan en T, y sumamos sus costos.
La suma total de todos los vértices sería el doble del costo del tour T (Hemos considerado
cada arista dos veces.)

(Suma de dos aristas del recorrido adyacentes a u) >= (suma del peso mínimo de dos aristas
                                                       adyacentes a u)

Costo de cualquier tour >= 1/2) * ∑ (Suma del costo mínimo de dos aristas de peso adyacentes a u)
                                    donde u ∈ V
```

Por ejemplo, considere el gráfico mostrado anteriormente. A continuación se muestra el coste mínimo de dos aristas adyacentes a cada nodo.




```
Nodo    Aristas de menor costo    Costo total
0           (0, 1), (0, 2)            25
1           (0, 1), (1, 3)            35
2           (0, 2), (2, 3)            45
3           (0, 3), (1, 3)            45

Por lo tanto, un límite inferior en el costo de
cualquier recorrido =
                    1/2 (25 + 35 + 45 + 45)
                    = 75
```

Ahora tenemos una idea sobre el cálculo del límite inferior. Veamos cómo se aplica el árbol de búsqueda del espacio de estados.

Comenzamos enumerando todos los nodos posibles (preferiblemente en orden lexicográfico)

1. **El nodo raíz**:
   Sin pérdida de generalidad, asumimos que comenzamos en el vértice "0" para el cual se ha calculado el límite inferior anteriormente.

Tratando con el nivel 2:
   El siguiente nivel enumera todos los vértices posibles a los que podemos ir (teniendo en cuenta que en cualquier camino un vértice tiene que ocurrir solo una vez) que son, 1, 2, 3 ... n (tenga en cuenta que el grafo está completo) .

   Considere que estamos calculando para el vértice 1. Dado que pasamos de 0 a 1, nuestro recorrido ahora ha incluido la arista 0-1. Esto nos permite realizar los cambios necesarios en el límite inferior de la raíz.

```
Límite inferior para el vértice 1 =

    Límite inferior anterior - ((costo de arista mínimo de 0 +
                                 coste de arista mínimo de 1) /
2)
                              + (costo de arista 0-1)
```

¿Cómo funciona?

Para incluir la arista 0-1, agregamos el costo de la arista 0-1, y restamos un peso de la arista de tal manera que el límite inferior permanece lo más apretado posible que sería la suma de las aristas mínimas de 0 y 1 dividido por 2.
Claramente, la arista restada no puede ser más pequeña que esta.

Tratando con otros niveles:

A medida que pasamos al siguiente nivel, volvemos a enumerar todos los vértices posibles. Para el caso anterior va más allá después de 1, echamos un vistazo para 2, 3, 4, ... n.
Considere el límite inferior para 2 a medida que nos movemos de 1 a 1, incluimos la arista 1-2 al recorrido y alteramos el nuevo límite inferior para este nodo.

```
Límite inferior (2) =

      Límite inferior anterior - ((segundo costo mínimo de la
                                  arista 1 + coste de arista mínimo de 2) / 2)

                               + coste de borde 1-2)
```

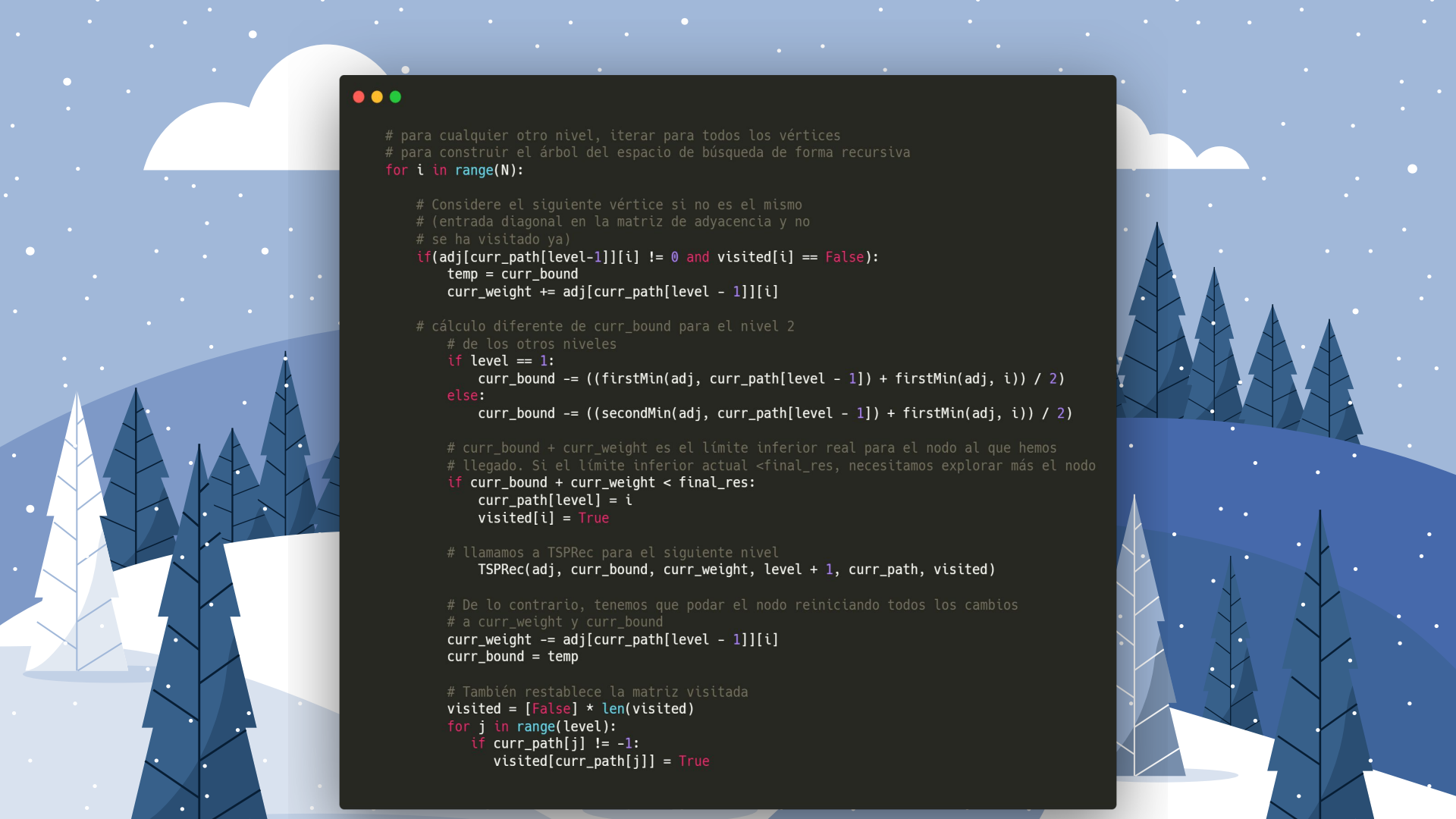
Algoritmo

```python
def firstMin(adj, i):

    """
    Función para encontrar el costo de
    arista mínimo que tiene un final en
    el vértice i
    """

    min = maxsize
    for k in range(N):
        if adj[i][k] < min and i != k:
            min = adj[i][k]

    return min
```

```python
def secondMin(adj, i):
    """
    Función para encontrar el segundo costo de arista
    mínimo que tiene un final en el vértice 'i'
    """

    first, second = maxsize, maxsize
    for j in range(N):
        if i == j:
            continue
        if adj[i][j] <= first:
            second = first
            first = adj[i][j]

        elif(adj[i][j] <= second and adj[i][j] != first):
            second = adj[i][j]

    return second
```

Algoritmo

```python
def TSPRec(adj, curr_bound, curr_weight, level, curr_path, visited):

    """
    función que toma como argumentos:

    - curr_bound -> límite inferior del nodo raíz
    - curr_weight -> almacena el peso de la ruta hasta ahora
    - level -> nivel actual mientras se mueve  en el árbol
              del espacio de búsqueda
    - curr_path [] -> donde se almacena la solución que luego
                      se copiaría a final_path

    """
    global final_res

    # El caso base es cuando hemos alcanzado el nivel N, lo que
    # significa que hemos cubierto todos los nodos una vez.
    if level == N:

    # compruebe si hay una arista desde el último vértice
    # en la ruta de regreso al primer vértice
        if adj[curr_path[level - 1]][curr_path[0]] != 0:

        # curr_res tiene el peso total de la solución
            # que obtuvimos
            curr_res = curr_weight + adj[curr_path[level - 1]][curr_path[0]]
            if curr_res < final_res:
                copyToFinal(curr_path)
                final_res = curr_res
        return
```



```
    # para cualquier otro nivel, iterar para todos los vértices
    # para construir el árbol del espacio de búsqueda de forma recursiva
    for i in range(N):

        # Considere el siguiente vértice si no es el mismo
        # (entrada diagonal en la matriz de adyacencia y no
        # se ha visitado ya)
        if(adj[curr_path[level-1]][i] != 0 and visited[i] == False):
            temp = curr_bound
            curr_weight += adj[curr_path[level - 1]][i]

        # cálculo diferente de curr_bound para el nivel 2
            # de los otros niveles
            if level == 1:
                curr_bound -= ((firstMin(adj, curr_path[level - 1]) + firstMin(adj, i)) / 2)
            else:
                curr_bound -= ((secondMin(adj, curr_path[level - 1]) + firstMin(adj, i)) / 2)

            # curr_bound + curr_weight es el límite inferior real para el nodo al que hemos
            # llegado. Si el límite inferior actual <final_res, necesitamos explorar más el nodo
            if curr_bound + curr_weight < final_res:
                curr_path[level] = i
                visited[i] = True

            # llamamos a TSPRec para el siguiente nivel
                TSPRec(adj, curr_bound, curr_weight, level + 1, curr_path, visited)

            # De lo contrario, tenemos que podar el nodo reiniciando todos los cambios
            # a curr_weight y curr_bound
            curr_weight -= adj[curr_path[level - 1]][i]
            curr_bound = temp

            # También restablece la matriz visitada
            visited = [False] * len(visited)
            for j in range(level):
               if curr_path[j] != -1:
                  visited[curr_path[j]] = True
```

Algoritmo

```
def TSP(adj):
    """
    Esta función configura final_path
    """

    # Calcule el límite inferior inicial para el nodo raíz
    # usando la fórmula 1/2 * (suma del primer min + segundo min) para todas las aritas.
    # También inicialice curr_path y la matriz visitada
    curr_bound = 0
    curr_path = [-1] * (N + 1)
    visited = [False] * N

    # Calcular el límite inicial
    for i in range(N):
        curr_bound += (firstMin(adj, i) + secondMin(adj, i))

    # Redondear el límite (la ramificación) inferior a un
    # número entero
    curr_bound = math.ceil(curr_bound / 2)

    # Comenzamos en el vértice 1, por lo que el primer vértice en curr_path [] es 0
    visited[0] = True
    curr_path[0] = 0

    # Llamamos a TSPRec para curr_weight igual a 0 y nivel 1
    TSPRec(adj, curr_bound, 0, 1, curr_path, visited)
```

Complejidad del tiempo:

La complejidad del peor caso de Branch and Bound sigue siendo la misma que la de la fuerza bruta claramente porque en el peor de los casos, es posible que nunca tengamos la oportunidad de podar un nodo.

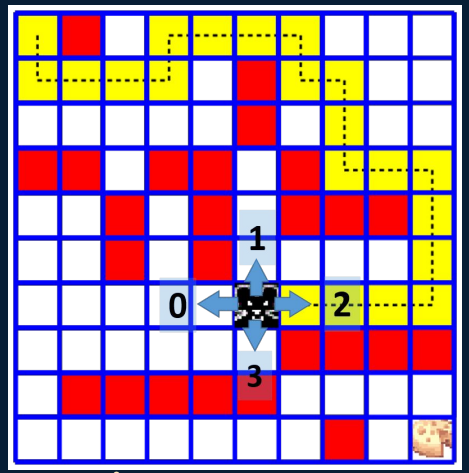
Considerando que, en la práctica, funciona muy bien dependiendo de las diferentes instancias del TSP. La complejidad también depende de la elección de la función delimitadora, ya que son ellos los que deciden cuántos nodos se podarán.

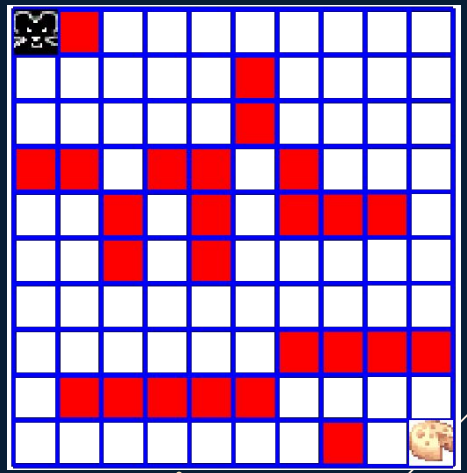
Backtracking

Problema del laberinto

Problema del laberinto

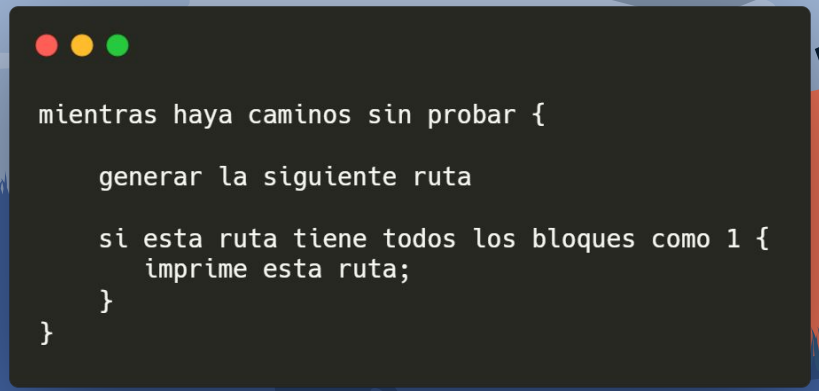
Considere la posibilidad de una rata colocada en (0, 0) en una matriz cuadrada **m[ ][ ]** de la orden n y tiene que llegar al destino en (n-1, n-1). La tarea consiste en encontrar una matriz ordenada de cadenas que denota todas las direcciones posibles que la rata puede tomar para llegar al destino en (n-1, n-1). Las direcciones en las que la rata puede moverse son **'U' (arriba)**, **'D' (abajo)**, **'L' (izquierda)**, **'R' (derecha)**.





Algoritmo Naive

Descubre todos los caminos posibles y luego encontrar la distancia mínima o el camino de saltos mínimo que debe tomar la rata para llegar al destino. Este procedimiento es de naturaleza compleja y se está utilizando mucha memoria para almacenar todas las rutas posibles y luego verificar la mejor ruta (cumple con las restricciones) posible en ella.



```
mientras haya caminos sin probar {

    generar la siguiente ruta

    si esta ruta tiene todos los bloques como 1 {
       imprime esta ruta;
    }
}
```

Backtracking

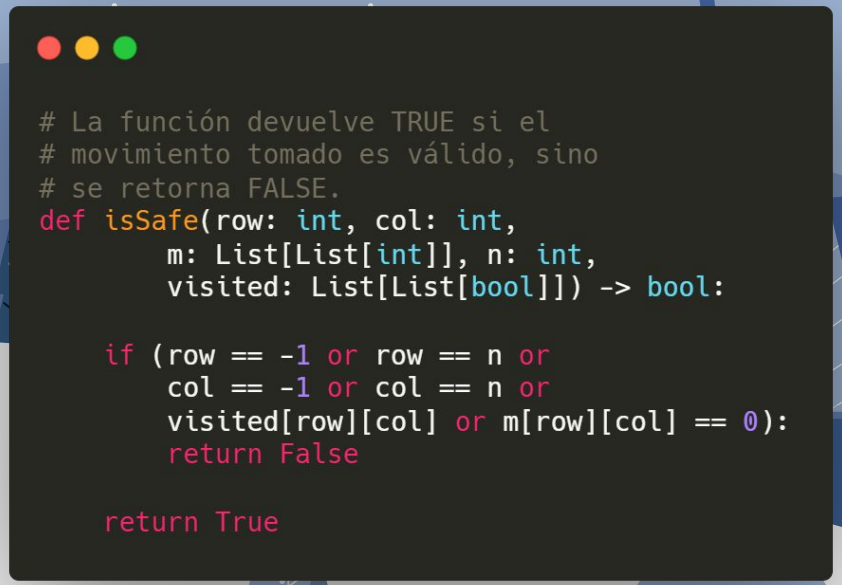
1. Comience desde el índice inicial (es decir **(0,0)**) y busque los movimientos válidos a través de las celdas adyacentes en el orden **Abajo**↣**Izquierda**↣**Derecha**↣**Arriba** (para obtener los trazados ordenados) en la cuadrícula.

2. Si el movimiento es posible, muévase a esa celda mientras almacena el carácter correspondiente al **mover(D, L, R, U)** y vuelva a empezar a buscar el movimiento válido hasta que se alcance el último índice (es decir, **(n-1,n-1)**).

3. Además, siga marcando las celdas como visitadas y cuando recorramos todas las rutas posibles desde esa celda, desmarque esa celda para otras rutas diferentes y elimine el carácter de la ruta formada.

4. A medida que se alcanza el último índice de la cuadrícula (abajo a la derecha), almacene la ruta de acceso atravesada.

Algoritmo



```python
# La función devuelve TRUE si el
# movimiento tomado es válido, sino
# se retorna FALSE.
def isSafe(row: int, col: int,
           m: List[List[int]], n: int,
           visited: List[List[bool]]) -> bool:

    if (row == -1 or row == n or
        col == -1 or col == n or
        visited[row][col] or m[row][col] == 0):
        return False

    return True
```

Algoritmo

```python
# Función para imprimir todas las posibles
# rutas desde (0, 0) hasta (n-1, n-1).
def printPathUtil(row: int, col: int,
                  m: List[List[int]],
                  n: int, path: str,
                  possiblePaths: List[str],
                  visited: List[List[bool]]) -> None:

    # Verificamos la posición inicial
    # (es decir. (0, 0)) para comenzar las rutas
    if (row == -1 or row == n or
        col == -1 or col == n or
        visited[row][col] or m[row][col] == 0):
        return

    # Si llega a la última celda (n-1, n-1)
    # entonces se almacena la ruta y retornamos
    if (row == n - 1 and col == n - 1):
        possiblePaths.append(path)
        return

    # Marcamos la celda como visitada
    visited[row][col] = True
```

Algoritmo

```
    # Probamos las 4 direcciones (abajo (D), izquierda (l),
    # derecha (R), arriba (U)) en el orden dado para obtener
    # las rutas en orden lexicográfico

    # Comprobamos si el movimiento hacia abajo es válido
    if (isSafe(row + 1, col, m, n, visited)):
        path += 'D'
        printPathUtil(row + 1, col, m, n,
                      path, possiblePaths, visited)
        path = path[:-1]

    # Comprobamos si el movimiento hacia la izquierda es válido
    if (isSafe(row, col - 1, m, n, visited)):
        path += 'L'
        printPathUtil(row, col - 1, m, n,
                      path, possiblePaths, visited)
        path = path[:-1]

    # Comprobamos si el movimiento hacia la derecha es válido
    if (isSafe(row, col + 1, m, n, visited)):
        path += 'R'
        printPathUtil(row, col + 1, m, n,
                      path, possiblePaths, visited)
        path = path[:-1]

    # Comprobamos si el movimiento hacia arriba es válido
    if (isSafe(row - 1, col, m, n, visited)):
        path += 'U'
        printPathUtil(row - 1, col, m, n,
                      path, possiblePaths, visited)
        path = path[:-1]

    # Marcamos la celda como no visitada para
    # otras posibles rutas
    visited[row][col] = False
```

Algoritmo

```
# función para almacenar e imprimir
# todas las rutas válidas
def printPaths(m: List[List[int]], n: int, showStringPath: bool):

    # almacenamos todas las posibles rutas
    possiblePaths = []
    path = ""
    visited = [[False for _ in range(MAX)] for _ in range(n)]

    # llamamaos a la función de utilidad para
    # encontrar las rutas válidas
    printPathUtil(0, 0, m, n, path, possiblePaths, visited)
    print("-----------------------------------------", end="\n")
    for i, p in enumerate(possiblePaths):
       print(f"\nMostrando solución {i+1}: ", end="\n\n")

       printMatrix(stringPath=p, n=n)
       print("-----------------------------------------", end="\n")

    if showStringPath:
        print("\nMostrando todas las cadenas las rutas: \n")
        for i in range(len(possiblePaths)):
            print(possiblePaths[i], end = " ")
    print("\n\n")
```

```
def printMatrix(stringPath, n):
    matrix = np.zeros(shape=(n, n))

    i, j = 0, 0
    matrix[i][j] = 1

    for letter in stringPath:
        if letter == 'L':
            j -= 1
            matrix[i][j] = 1

        if letter == 'R':
            j += 1
            matrix[i][j] = 1

        if letter == 'U':
            i -= 1
            matrix[i][j] = 1

        if letter == 'D':
            i += 1
            matrix[i][j] = 1

    print(matrix)
```

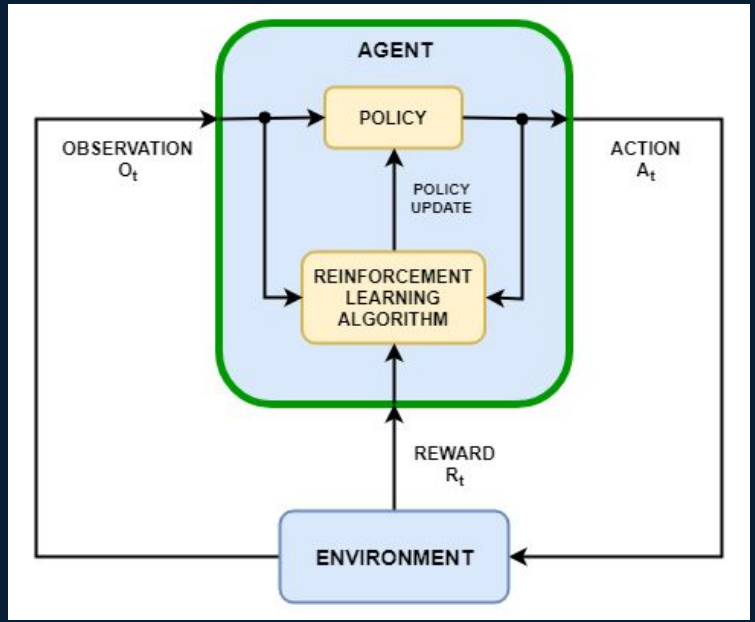
Análisis de complejidad:

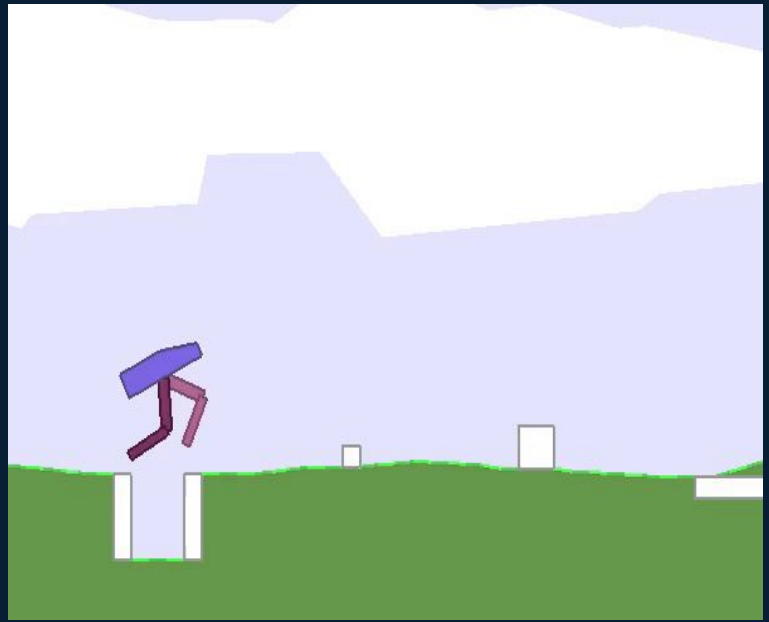
- **Complejidad del tiempo**:
  Como hay **N^2** celdas desde cada celda, hay 3 celdas vecinas no visitadas.
  Entonces, la complejidad del tiempo **O(3^(N^2))**.

- **Espacio auxiliar**:
  Como puede haber al menos **3^(N^2)** celdas en la respuesta, la complejidad del espacio es **O(3^(N^2))**.

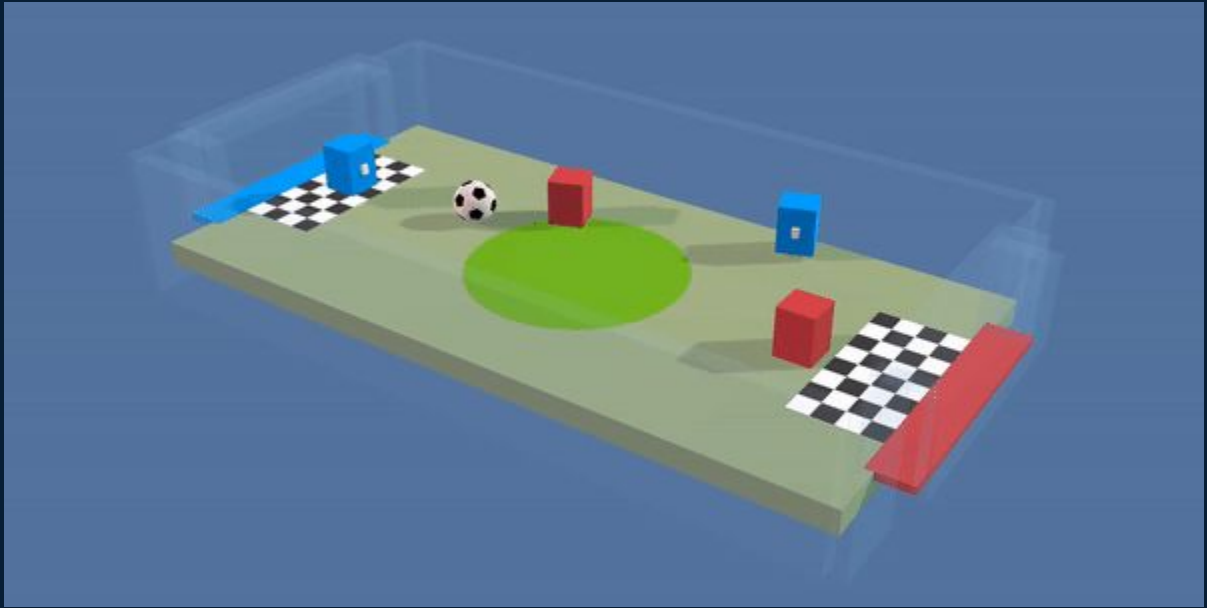
Aprendizaje Reforzado

Posibles soluciones a problemas intratables






Unity ML-Agents Workflow
Create Environment
Train Agents
Embed Agents



DEMO

Gym

Gym is a toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Pinball.

View documentation ›
View on GitHub ›

RandomAgent on SpaceInvaders-v0