

# Postulación empleo Reverso

Postulante: Felipe Gutiérrez Benítez

Fecha entrega: 24/01/2021

## Introduccion

La prueba de la postulación se define en buscar la solución a un juego, en la cual una persona debe mover la caja dentro de una bodega (tamaño M x N). Donde los elementos pueden ser murallas, piso, persona y caja. Siendo el objetivo mover C a P cumpliendo ciertas normas.

P	=	Persona
C	=	Caja
D	=	Destino
.	=	Piso
#	=	Pared

Normas:

- Solo existe una caja y una persona
- Solo están permitidos 4 movimientos: Arriba, Abajo, Izquierda y Derecha
- La caja se puede mover solo si en una casilla adyacente se encuentra C
- El jugador no puede posicionarse sobre la caja

Diccionario de movimientos:

P derecha	=	p-r
P izquierda	=	p-l
P arriba	=	p-u
P abajo	=	p-d
C derecha	=	c-r
C izquierda	=	c-l
C arriba	=	c-u
C abajo	=	c-d

Input:

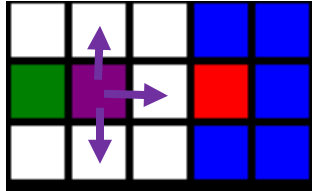
```
grid = [ ["#", "#", "#", "#", "#"],  
        ["#", "#", "#", "D", "#"],  
        ["#", "#", "#", ".", "#"],  
        ["#", "#", "#", ".", "#"],  
        ["#", "P", ".", "C", "#"],  
        ["#", "#", "#", "#", "#]]
```





Output:

```
> search_path(grid)  
> ["p-r", "c-u", "p-r", "c-u", "p-u", "c-u"]
```

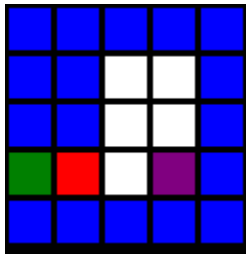
## Observaciones personales e interpretaciones.





1. La caja solo puede ir hacia adelante y laterales desde la perspectiva de P



	P		C
	#		D

2. Cuando la caja llegue a destino termina el juego
3. Al no especificarse si **P** puede pasar sobre **D**. Se considero posible que lo hiciera



	P		C
	#		D







4. Una vez P alcanza a C. Todos los movimientos de P se vuelven una seguidilla del movimiento de C.

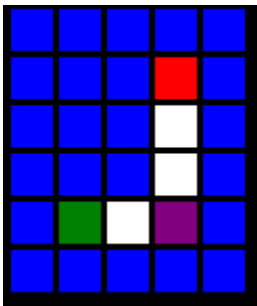
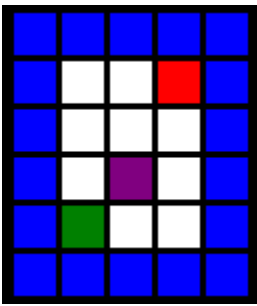
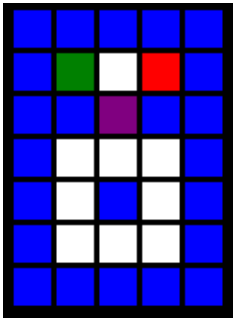
## Software utilizado

- Sistema operativo Windows 10
- Visual studio code
- Lenguaje de programacion: Python 3.9.1
  - astroid==2.4.2
  - colorama==0.4.4
  - isort==5.7.0
  - lazy-object-proxy==1.4.3
  - mccabe==0.6.1
  - pylint==2.6.0
  - six==1.15.0
  - toml==0.10.2
  - wrapt==1.12.1

## Explicación solución propuesta.

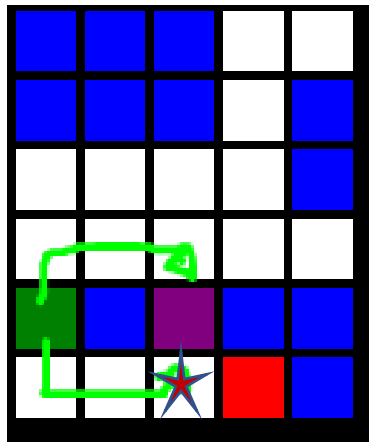
El script busca una de las rutas más cortas para que persona (P) pueda llevar la caja (C) hacia el destino(D). Para ello vamos a definir la bodega como un laberinto y se dividirá en tres posibles tipos de laberintos (Aclaración: los nombres fueron inventados según sus características para este ejercicio, se desconoce si realmente existe un nombre para cada uno).

	P		C
	#		D

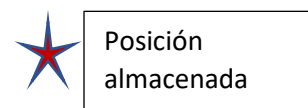
Laberinto 1: Lineal	Laberinto 2: Doble o múltiple	Laberinto 3: Tapa de botella
		
Este laberinto solo tiene una solución y los movimientos son fijos y secuenciales	En este hay a lo menos dos rutas para que P llegue a C y para que C llegue a D	El más complicado de resolver. Ya que P bloquea a C. Se necesita buscar una ruta alternativa para que pueda llegar C a D. Se necesita que pueda dar una vuelta en circulo por la ruta alternativa

**Laberinto 1 y laberinto 2** se pueden resolver de la siguiente manera (se entregará la explicación con un ejemplo de laberinto con doble alternativa pero siguen la misma lógica):

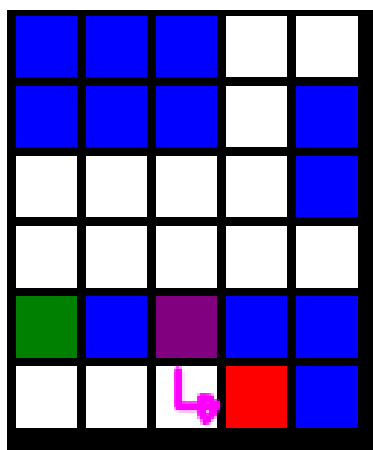
1. Primero se buscan todas las rutas posibles y se define una de la ruta más corta entre P a C y se almacena la posición aledaña de C (se utiliza en la explicación 3)



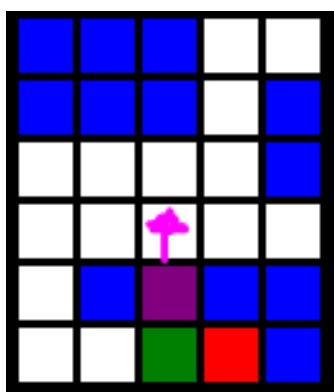
	P		C
	#		D



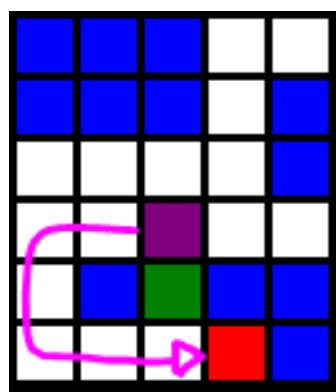
2. Segundo, se busca la ruta más corta entre C a D.



3. Tercero, se busca ruta más corta entre C y D. Marcando de manera temporal la posición de P. Dado que en solo en el primer movimiento P estaría posicionado ahí.

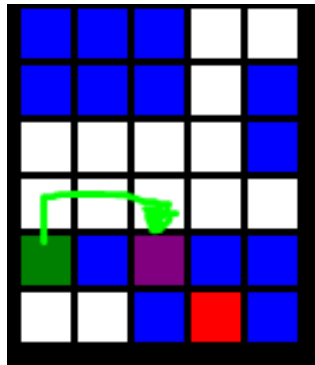


*Primer Ciclo*

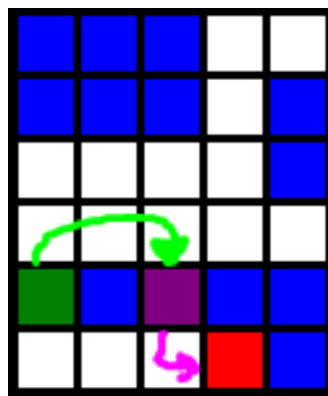


*Segundo Ciclo*

4. Cuarto, Se compara la ruta ideal de C con la ruta forzada de C gracias a P. Si resulta ser más corta la ruta ideal de C. Se fuerza a crear una ruta nueva para P

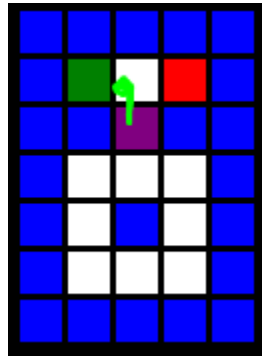


El resultado final de este ejemplo seria:

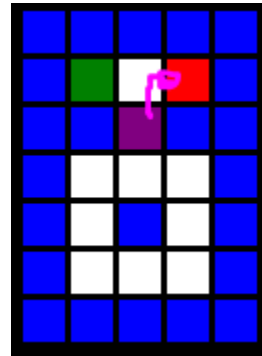


**Laberinto 3** : Para resolver laberintos del tipo 3 es necesario que en el camino alternativo se pueda realizar un ciclo sin pasar por el bloque anterior visitado despues de un turno.

1. Primero se comprueba que exista una ruta entre P y C. Y que también exista una ruta entre C y D

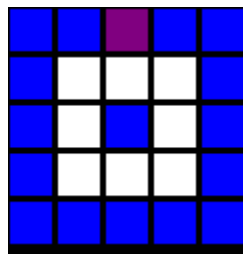


*Ruta P-C*

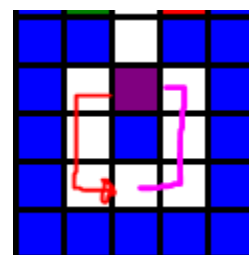
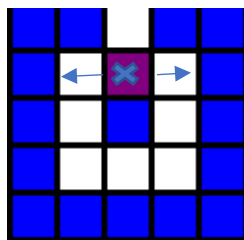
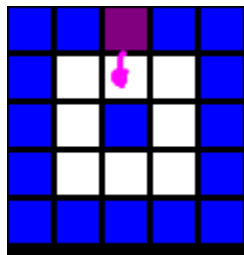


*Ruta C-D*

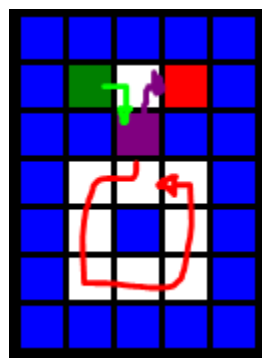
2. Luego se bloquea el lado por el cual llega P disminuyendo el laberinto a la siguiente ruta alternativa.



3. Se busca que C pueda llegar a su punto de inicio sin volver sobre su paso de manera inmediata, hasta que se encuentre una posición visitada

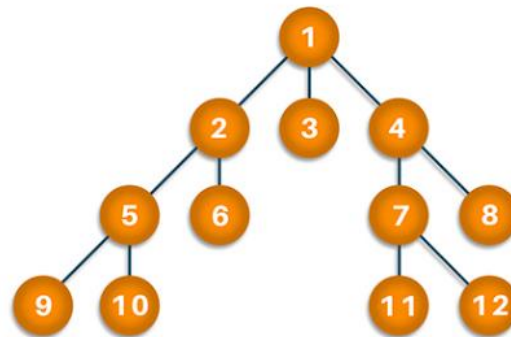


4. Se suman las rutas de (P-C) + (Ruta\_alterna)+ (C-D)



¿Cómo se encuentra la ruta más corta en el laberinto 1 y laberinto 2?

Se utilizo el algoritmo BFS el cual es uno de los más eficientes para encontrar rutas en laberintos que lleguen a un destino deseado.

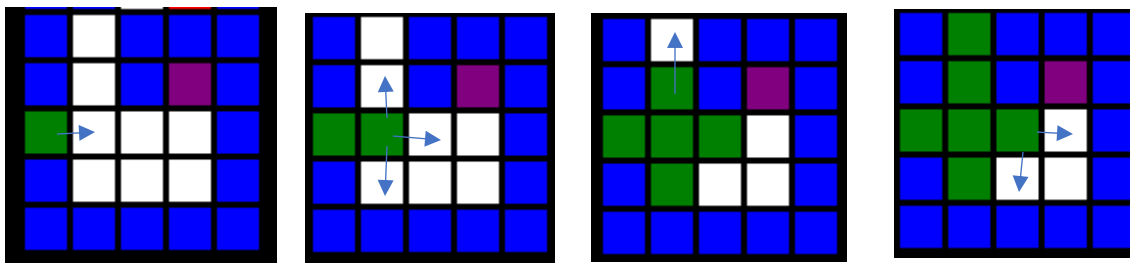






**BREATH FIRST SEARCH**

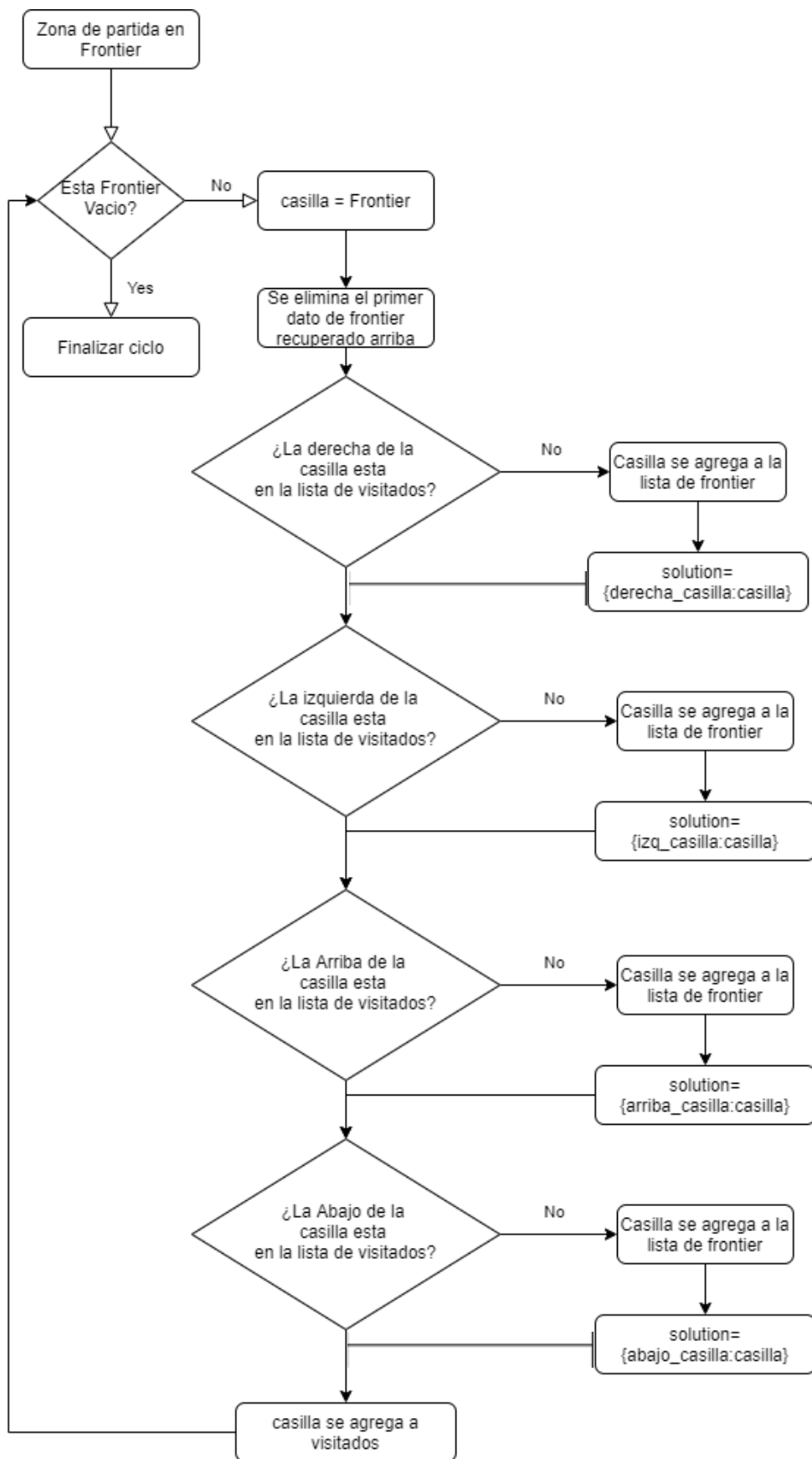
En el script presentado se define la función crearLaberinto(). La idea de esta función es traducir la matriz ingresada como grid para así sacar la posion inicial, la posicion final en cordenadas. Y también sacar el camino en un arreglo.

Luego se ejecuta la funcion definida como search() la cual utiliza el principio de BFS almacenando en primera instancia un frontier el cual permite recorrer por fila el arbol a medida que vaya existiendo algo por recorrer en cualquiera de las 4 direcciones se va agregando al frontier esa dirección y las ya visitadas se van eliminando de frontier. También se utiliza dentro de esta funcion una lista que guarda las zonas visitadas y un diccionario llamado Solución [esencial para encontrar la ruta final más corta]

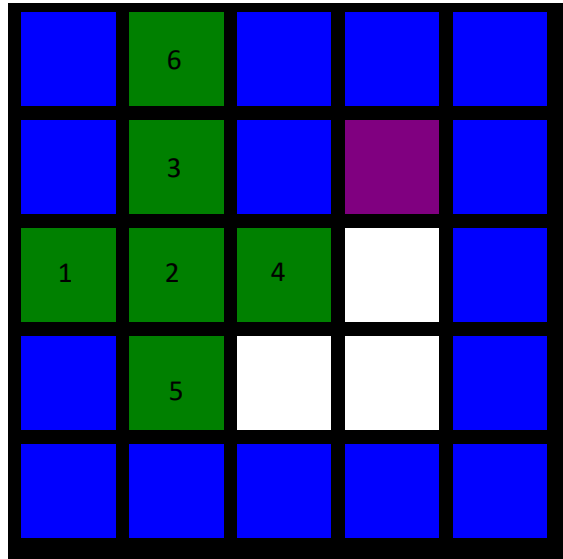
*Explicación visual y diagrama de flujo:*



	C		Zona visitada
	#		Posible camino



Para definir cual es la ruta más corta tenemos la función backroute() el cual lee el diccionario creado con el nombre de variable “solucion”



El formato del diccionario es el siguiente: solucion {Casilla\_Actual:Casilla\_Previa}

Casilla_Actual {significante}	Casilla_previa {significado}
1	1
2	1
3	2
4	2
5	2
6	3

Ejemplo secuencia del 4:

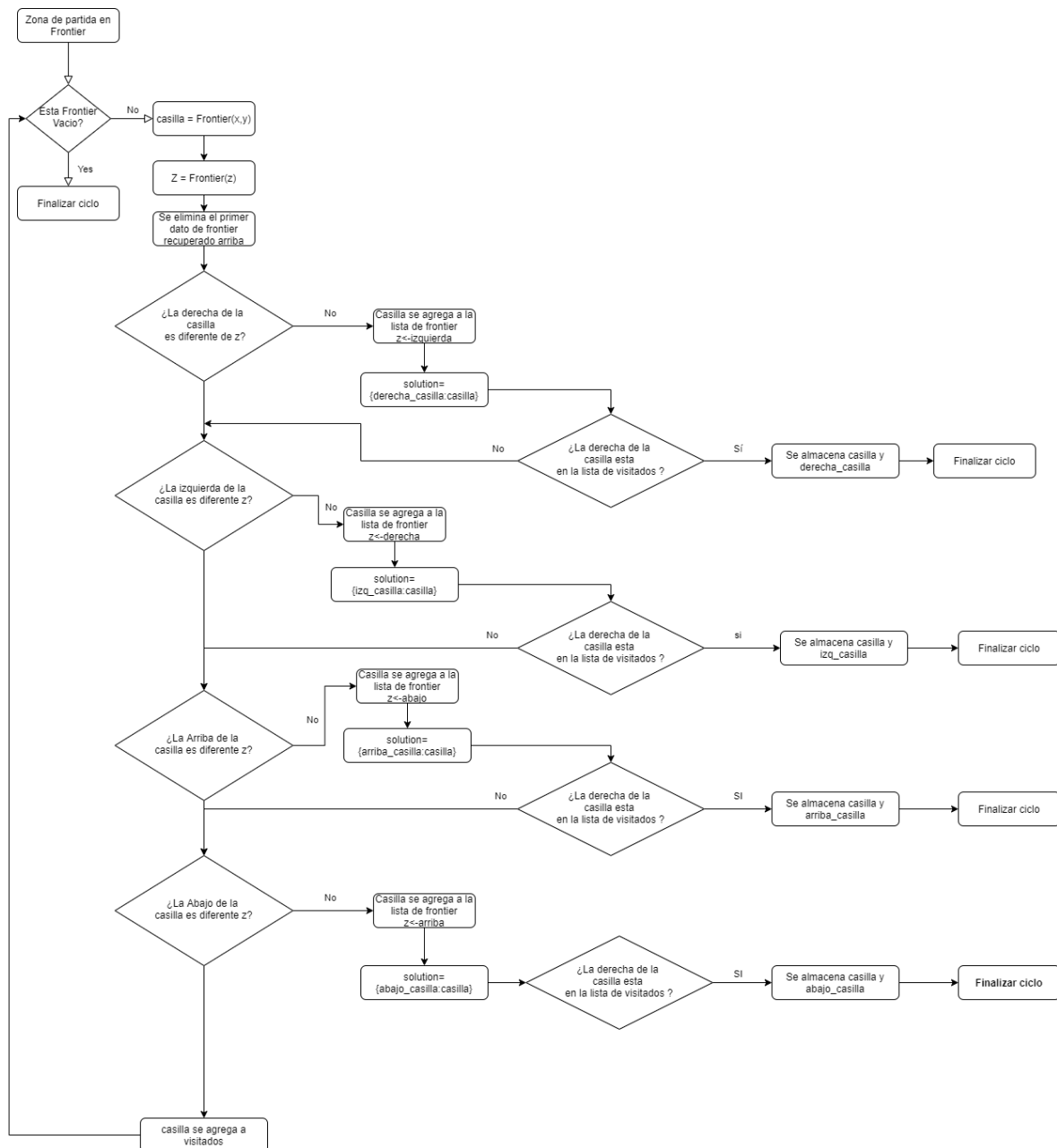
El numero 4 proviene de la casilla 2 y casilla 2 viene de la casilla 1

Si la secuencia se hace con la posición final se saca cual fue la ruta más corta.

## ¿Cómo se encuentra la ruta más corta en el laberinto 1 y laberinto 2?

En base para buscar la ruta entre P-C y C-D ideales. Se utiliza la misma forma utilizada en la explicación anterior. Pero para buscar la ruta alternativa se utiliza una funcion similar a search utilizando una variables extra, las cual almacena de donde viene la casilla para que este no pueda volver sobre sus paso anterior y solo seguir hacia adelante.

### Explicacion diagrama de flujo



Ejemplos de laberintos resueltos por el script entregado:

```
grid = [ ["#", "#", "#", "#", "#"],
          ["#", "#", "#", "D", "#"],
          ["#", "#", "#", ".", "#"],
          ["#", "#", "#", ".", "#"],
          ["#", "P", ".", "C", "#"],
          ["#", "#", "#", "#", "#"]]
```

```
grid = [ ["#", "#", "#", "#", "#"],
          ["#", "#", "#", "#", "#"],
          ["#", "#", ".", ".", "#"],
          ["#", "#", ".", ".", "#"],
          ["P", "D", ".", "C", "#"],
          ["#", "#", "#", "#", "#"]]
```

```
grid= [ [".", ".", ".", "#", "#"],
          ["P", "C", ".", "D", "#"],
          [".", ".", ".", "#", "#"]]
```

```
grid= [ ["#", "#", "#", "D", "."],
          ["P", ".", ".", ".", "#"],
          ["#", "#", "C", "#", "#"],
          ["#", ".", ".", ".", "#"],
          ["#", ".", ".", ".", "#"],
          ["#", ".", ".", ".", "#"]]
```