



INTELIGENCIA ARTIFICIAL II

TRABAJO PRÁCTICO N°1



GRUPO 2

Contenido



1 Algoritmo A*

2 Algoritmo A* Multiagente

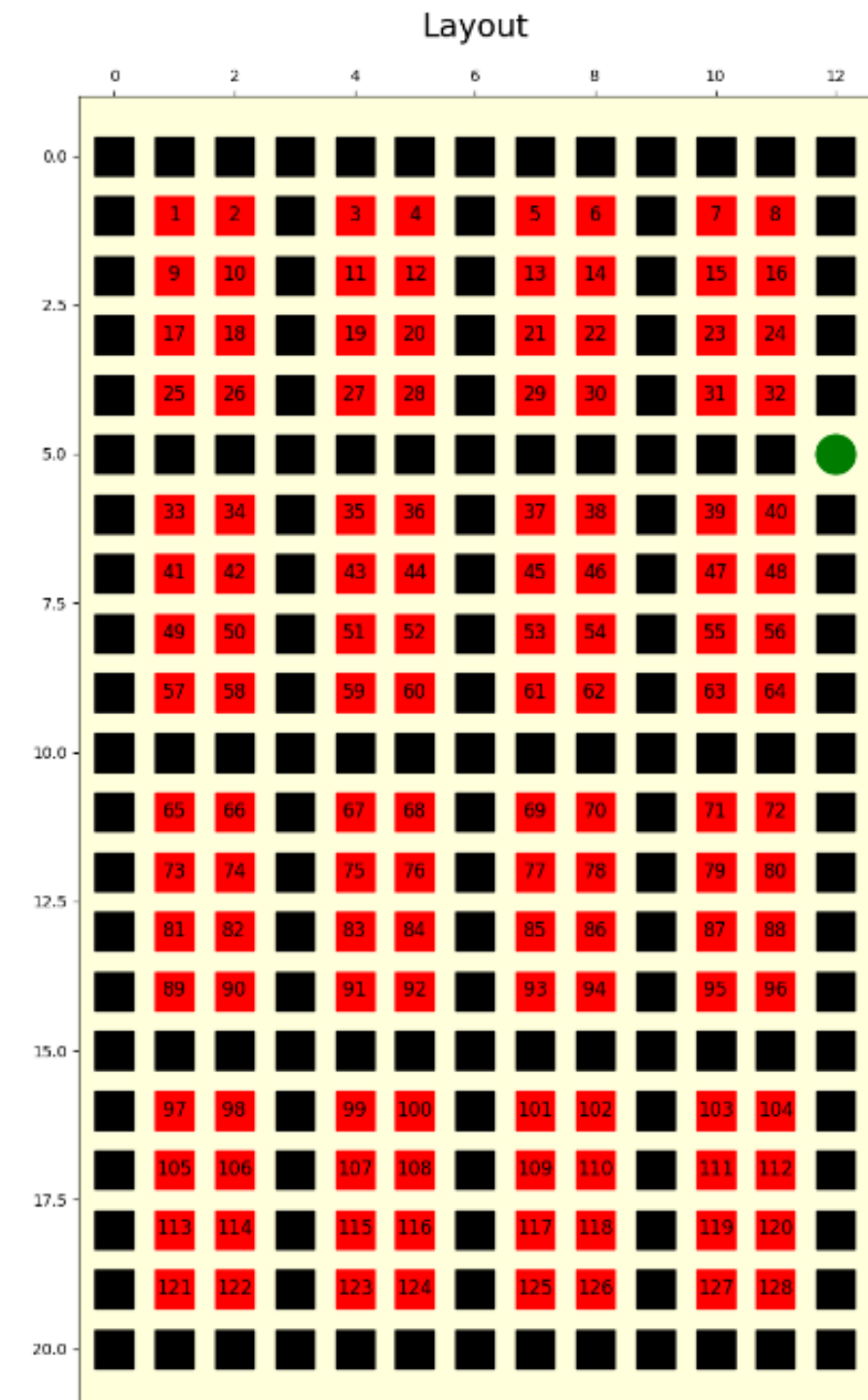
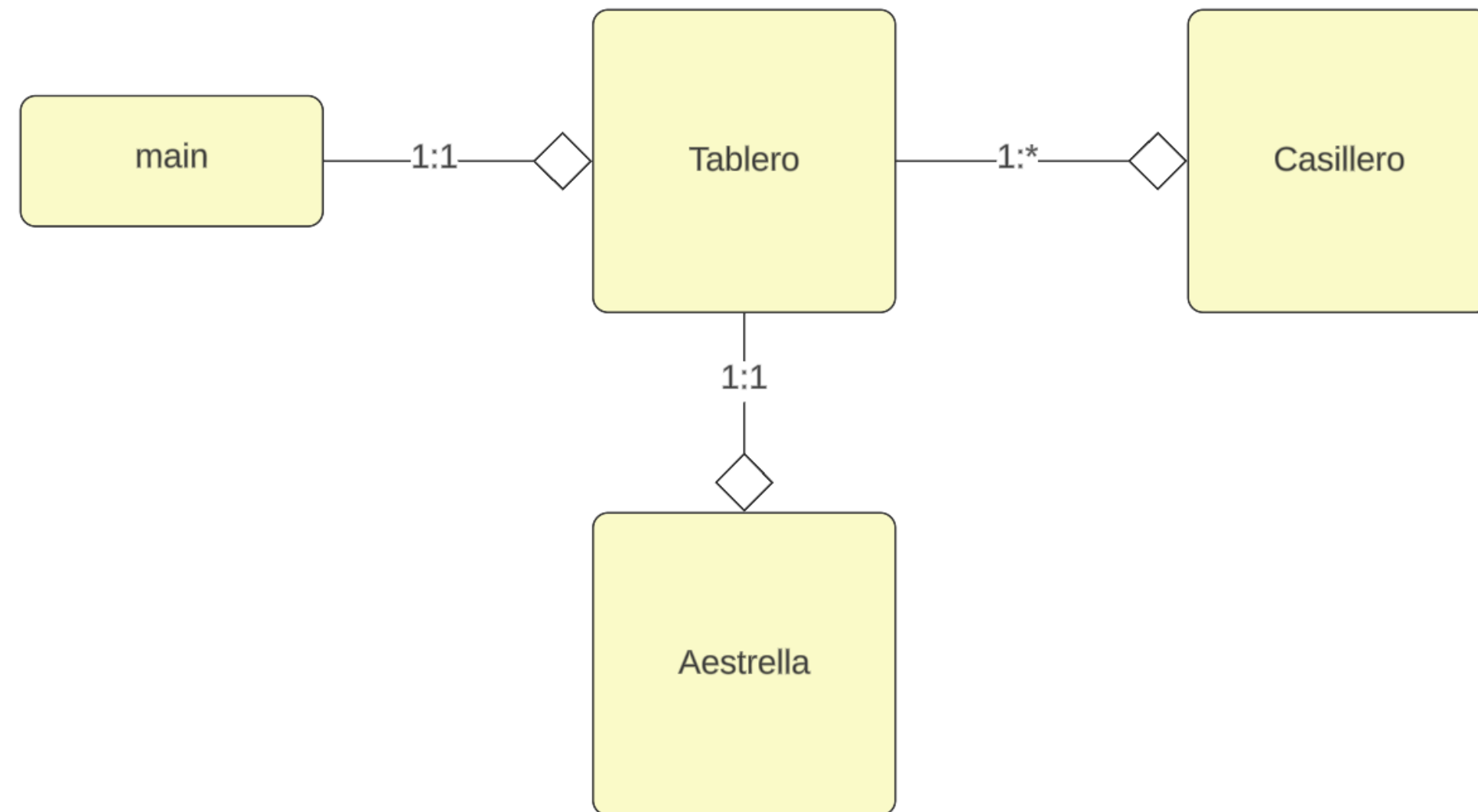
3 Recocido Simulado

4 Algoritmo Genético

EJERCICIO 1: Algoritmo A*



Ejercicio 1 - Diagrama de bloques



Ejercicio 1 - Generalidades

- **Menú** tipo CLI.
- Generación del **tablero** de forma matricial con condicionales para definir pasillos.
- La matriz tablero está compuesta de objetos de la clase **Casillero**
- La clase casillero tiene atributos para establecer vecinos, tipo (pasillo/estantería), ubicación y un alias asociado al número de la estantería para un trabajo más cómodo.
- El **agente** se genera de forma aleatoria, su posición es conocida por **tablero**.
- Generación de **objetivos** de forma aleatoria o manual.
- Impresión usando impresiones recursivas con **matplotlib** e información contenida en **tablero** y en cada **casillero** para dar estilos.

Ejercicio 1 - Código A*

Iniciación

```
def __init__(self, pinicial, pfinal, tablero):  
  
    self.pinicial = pinicial  
    self.pactual = pinicial  
    self.pfinal = pfinal  
    self.tablero = tablero  
    self.pfinalVecinosCoords = [vecino.getCoords() for vecino in tablero[pfinal[0]][pfinal[1]].getVecinos()]  
    self.camino = []  
    self.encontrarCamino()
```

Test Objetivo

```
def testObjetivo(self):  
  
    if self.pactual in self.pfinalVecinosCoords:  
        return True  
  
    return False
```


Bucle (1)

```
def encontrarCamino(self):

    visitados = [{"punto": self.pinicial, "padre": None, "costo":0}]

    pila_explorar = []

    while not self.testObjetivo():

        vecinos = self.tablero[self.pactual[0]][self.pactual[1]].getVecinos()

        for vecino in vecinos:

            if vecino.getTipo() == "estante":
                continue

            if vecino.getCoords() in [visitado["punto"] for visitado in visitados]:
                continue

            else:

                heuristica = abs(vecino.getCoords()[0] - self.pfinal[0]) + abs(vecino.getCoords()[1]
                                                                              - self.pfinal[1])
                #costo = abs(vecino.getCoords()[0] - self.pinicial[0]) + abs(vecino.getCoords()[1]
                                                                              # - self.pinicial[1])

                costo = visitados[-1]["costo"] + 1
```

Bucle (2)

```
        pila_explorar.append({"posicion": vecino.getCoords(), "vecino": vecino,
                              "f_eval": heuristica + costo, "padre": self.pactual})

    pila_explorar = sorted(pila_explorar, key=lambda k: k["f_eval"])

    padre = pila_explorar[0]["padre"]

    self.pactual = pila_explorar[0]["vecino"].getCoords()
    pila_explorar.pop(0)

    visitados.append({"punto": self.pactual, "padre": padre})

self.camino = []

self.camino.append(visitados[-1]["punto"])
self.camino.insert(0, visitados[-1]["padre"])

while self.camino[0] is not None:

    for visitado in visitados:

        if visitado["punto"] == self.camino[0]:
            self.camino.insert(0, visitado["padre"])
            break

    self.camino.pop(0)
```

Reconstrucción hacia
atrás del camino

Ejercicio 1 - Posibles mejoras

- Para la **reconstrucción hacia atrás** del camino utilizamos un arreglo con diccionarios de elementos visitados, en ellos se tienen las asociaciones padre-hijo. La complejidad de búsqueda es lineal **$O(n)$** siendo n el número de diccionarios.
- El **método de impresión** requiere reimprimir toda la matriz evaluando todas las condiciones para dar estilos en cada paso lo que no solo tiene una demanda computacional sino que, al sobrecargarse las ordenes de impresión, puede presentarse un error.
- En cada ejecución del programa el **tablero se regenera** lo que introduce una carga innecesaria que podría ser evitada almacenando la información respecto a este y a sus casilleros. **Solución:** podría usarse otra librería para la visualización, como pygame.
- La evaluación de **casilleros ya visitados** es satisfactoria para el tamaño de pasillo dado, en contextos diferentes podría ocasionar problemas por suponer que ya se ha visitado un casillero cuando en realidad se está recorriendo por un nuevo camino.

```

if option == 1:
    tablero.objetivoA1

elif option == 2:
    obj = int(input("
    tablero.objetivoMa

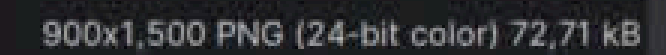
elif option == 3:
    tablero.recorrer()

else:
    print("Opción no v

nt("Saliendo del progr
urn

```

```
if __name__ == '__main__':
```

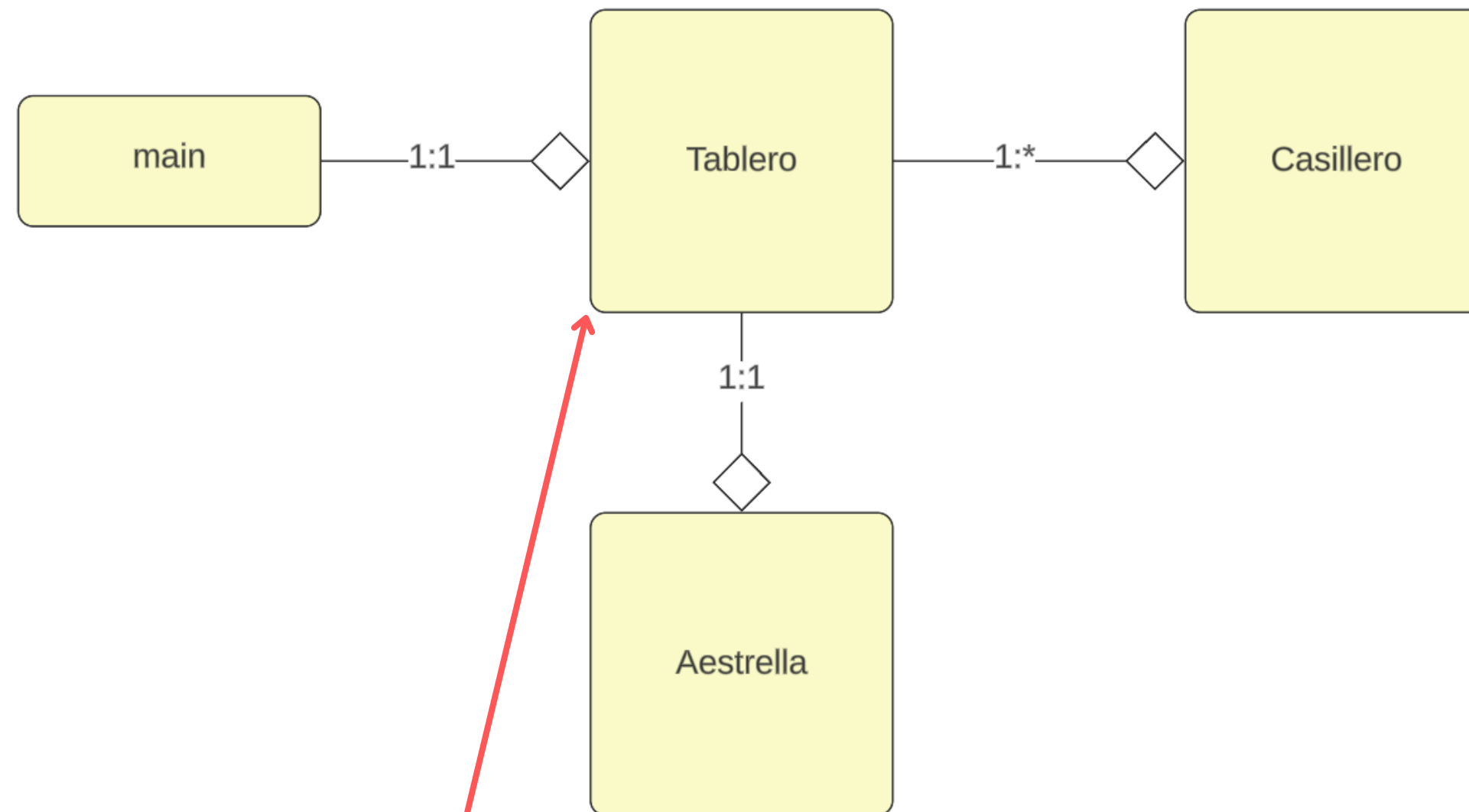


EJERCICIO 2: Algoritmo A*

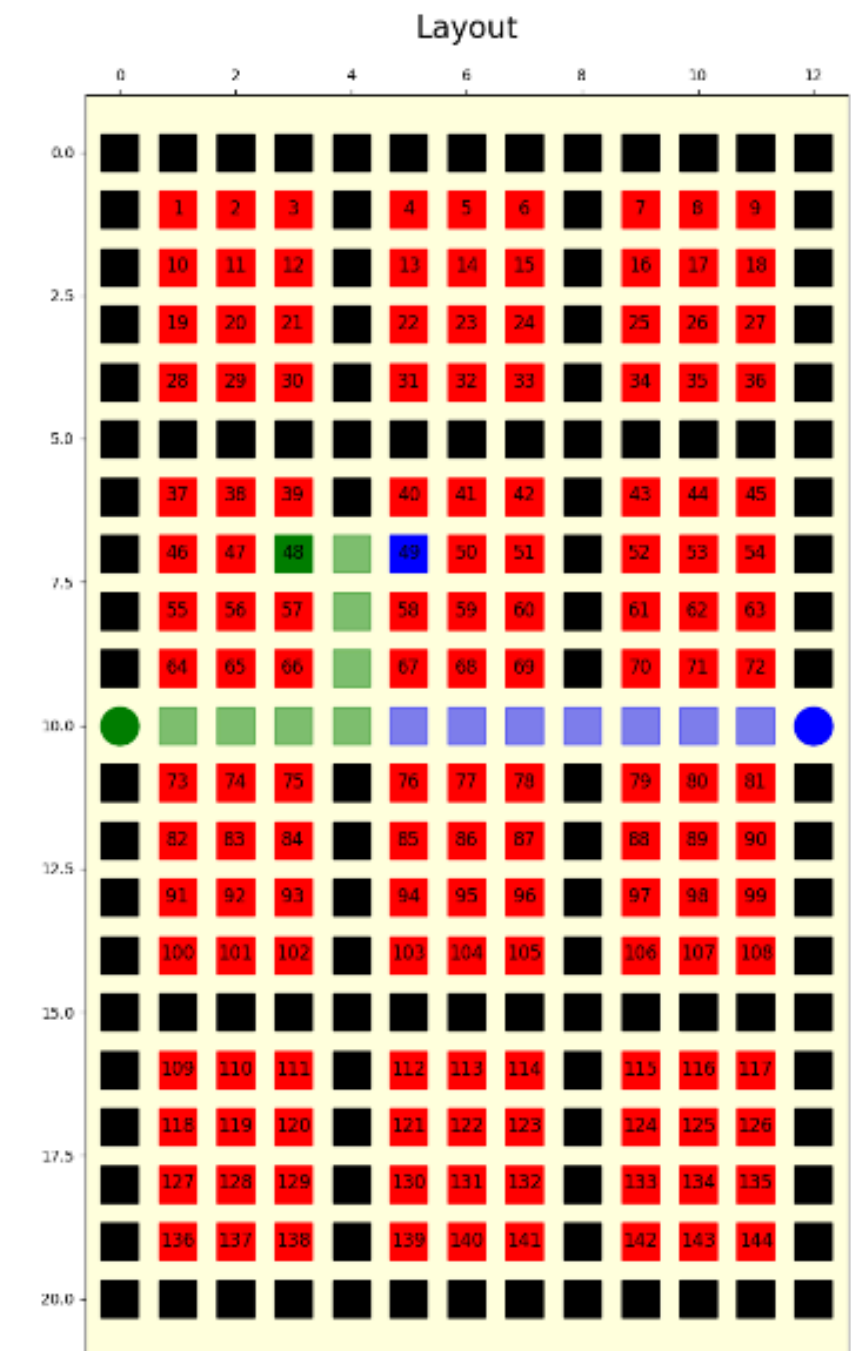
Multiagente



Ejercicio 2 - Diagrama de bloques



Incorporación de funcionalidades



Ejercicio 2 - Generalidades

- La mayor parte del código es análogo al ejercicio anterior, con algunas modificaciones.
- Cada **agente** comienza siempre en un mismo punto, ambos enfrentados en los bordes del pasillo que se encuentra en la mitad del tablero.
- Los **agentes** van avanzando en turnos alternados hasta llegar al objetivo (A-B-A-B-...).
- Cada agente evalúa diferentes condiciones antes de avanzar por la ruta planeada (modificaciones del algoritmo A*).

Ejercicio 2 - Generalidades



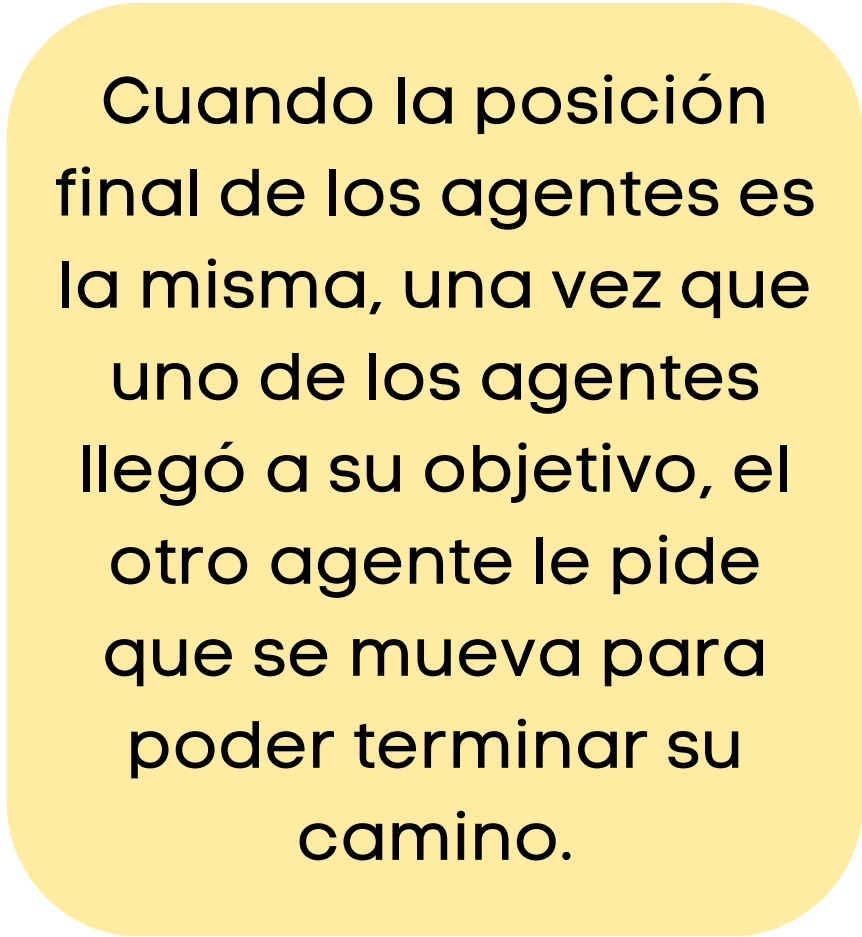
Agente A

Antes de avanzar, evalúa si la próxima casilla es la casilla en la que se encuentra el Agente B. En caso de serlo, se ejecuta nuevamente A^* desde la posición actual, pero enviando un nuevo argumento, que es una restricción en la casilla donde está el Agente B. Se descarta esa opción como vecino.



Agente B

Si la próxima casilla en el camino del Agente B es la posición actual del Agente A, el Agente B se queda quieto ("pasa su turno") hasta que A se mueva como se describió anteriormente.



Cuando la posición final de los agentes es la misma, una vez que uno de los agentes llegó a su objetivo, el otro agente le pide que se mueva para poder terminar su camino.

Ejercicio 2 - Código

Menú: en el modo manual, se piden dos coordenadas objetivo, una para cada agente

```
Inicianado el programa...
=====
[0] Salir
[1] Generar objetivo aleatorio
[2] Ingresar objetivo manualmente
[3] Recorrer
Ingrese una opción: 2
=====
Ingrese el alias del primer objetivo: 23
Ingrese el alias del segundo objetivo: 64
Objetivos ingresados: [3, 10] - [9, 11]
=====
```

Verificación que realiza el Agente A antes de avanzar

```
if self.caminoA:
```

```
    if self.caminoA[0] == self.agenteB and self.agenteB in self.vecinosObjB:
```

```
        print(f"{RED}Solicitando al agenteB liberar el espacio...{RESET}")
```

```
        for vecino in self.tablero[self.agenteB[0]][self.agenteB[1]].getVecinos():
```

```
            if vecino.getTipo() == "pasillo" and self.agenteA != vecino.getCoords():
```

```
                self.caminoB.append(vecino.getCoords())
```

```
    elif self.caminoA[0] == self.agenteB:
```

```
        print(f"{RED}Colisión detectada{RESET}")
```

```
        a_estrella_A = Aestrella(self.agenteA, self.coordsObjA, self.tablero, restricciones=[self.agenteB])
```

```
        self.caminoA = a_estrella_A.getCamino()
```

```
        print(f"{GREEN}Camino reculado para A.{RESET}")
```

```
        print(f"{GREEN}Camino A: {RESET}{self.caminoA}")
```

```
    else:
```

```
        self.agenteA = self.caminoA.pop(0)
```

```
        self.plotearTablero()
```

```
        time.sleep(2)
```

Caso: cuando el Agente B se encuentra bloqueando el objetivo de A

Encuentra una casilla de tipo "pasillo" y lo agrega al camino de B

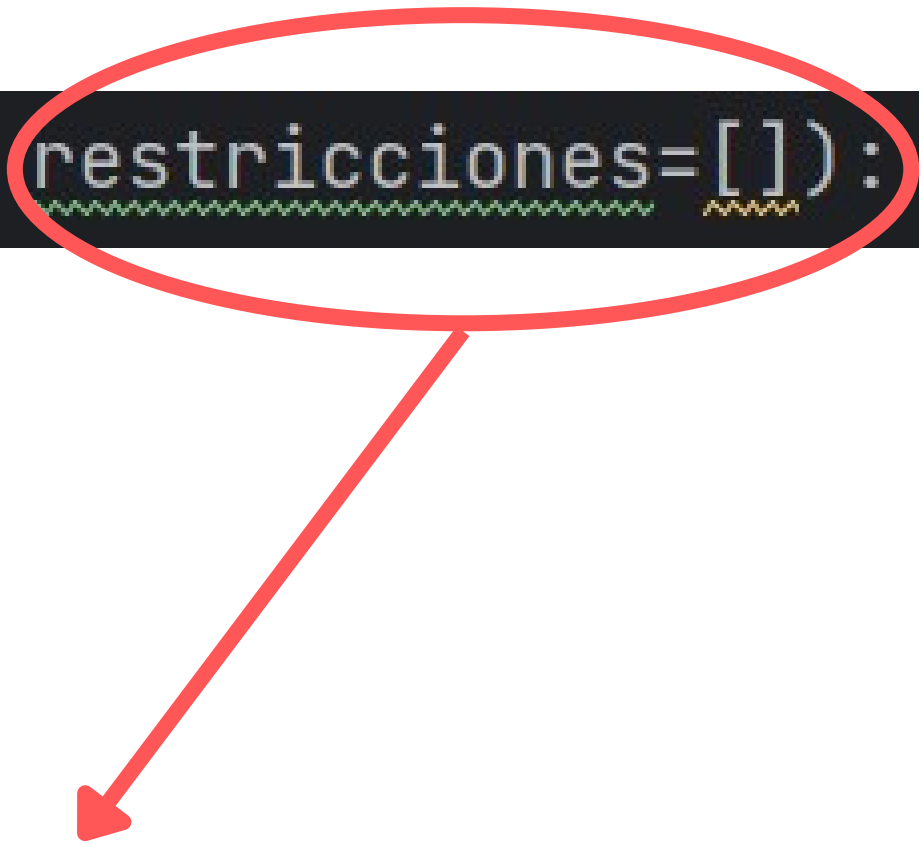
Caso: cuando el Agente B se encuentra bloqueando el camino de A

Envío de restricciones a A* para calcular la ruta óptima de nuevo

Procedimiento "normal"

Incorporación de argumento en el algoritmo A*

```
def __init__(self, pinicial, pfinal, tablero, restricciones=[]):
```



Se incluyen las coordenadas de la posición actual del Agente B, si está bloqueando la ruta planeada de A

Verificación que realiza el Agente B antes de avanzar

```
if self.caminoB:

    if self.caminoB[0] == self.agenteA and self.agenteA in self.vecinosObjA:

        print(f"{RED}Solicitando al agenteA liberar el espacio...{RESET}")

        for vecino in self.tablero[self.agenteA[0]][self.agenteA[1]].getVecinos():
            if vecino.getTipo() == "pasillo" and self.agenteB != vecino.getCoords():
                self.caminoA.append(vecino.getCoords())

    elif self.caminoB[0] == self.agenteA:

        self.agenteB = self.agenteB

    else:

        self.agenteB = self.caminoB.pop(0)
        self.plotearTablero()
        time.sleep(2)
```

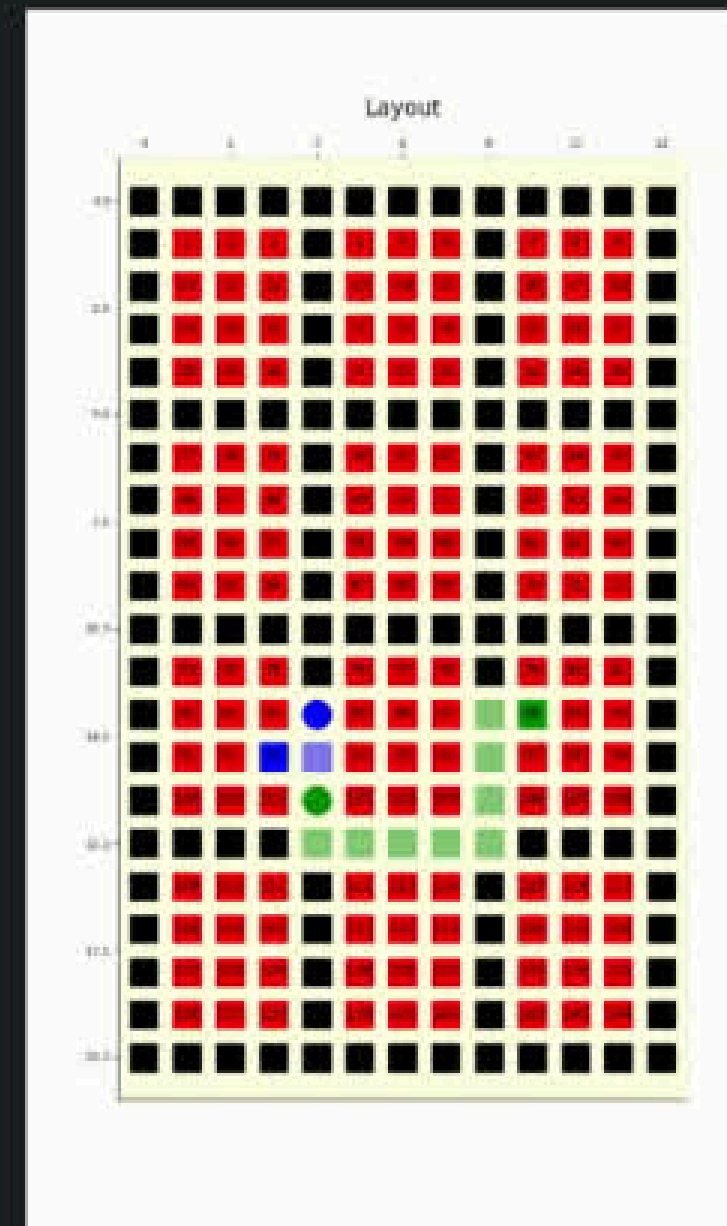
Caso: cuando el Agente A se encuentra bloqueando el camino de B

El resto es análogo al caso del Agente A

Ejercicio 2 - Posibles mejoras

- Cuando los agentes se encuentran “de frente” en un mismo pasillo, uno de ellos toma un camino no conveniente para dejar pasar al otro (en este caso, el Agente A, que recalcula su ruta). En vez que hacer esto, sería mejor que uno de ellos se mueva, deje pasar al otro y luego siga avanzando por la ruta planeada, que es la de menor costo.
- En el ejercicio se plantea que los agentes cuentan con sensores que pueden indicarle al robot que el otro se encuentra en frente de él para evitar colisiones. Se podría contar con un sensor de mayor alcance (programarlo) para lograr la solución anterior.

Colisión en camino



900x1,500 PNG (24-bit color) 75,16 kB

Run  main 

Colisión detectada.

Camino reculado para A.

Colisión por coincidencia de posición final

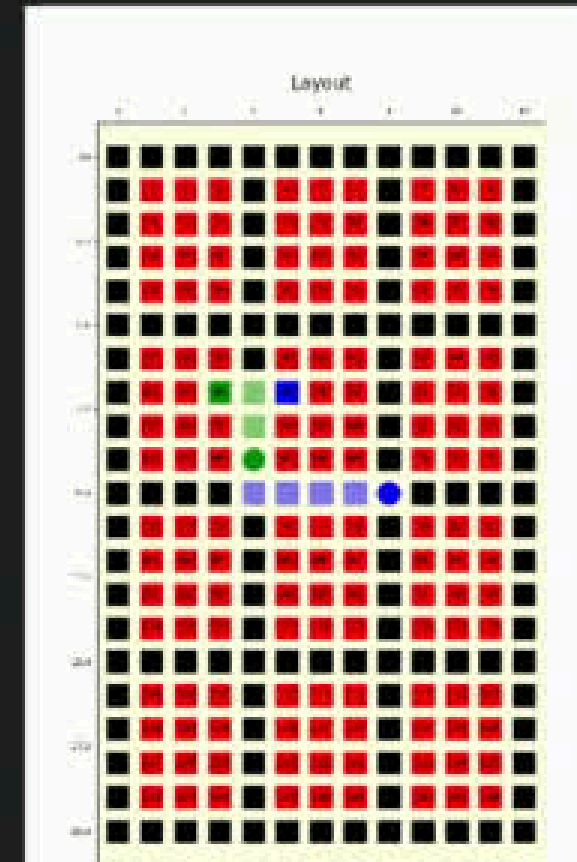
```
ción: ")
print("=====")

try:
    option = int(option)

except ValueError:
    print(f"{RED}Opción no válida{RESET}")
    continue

if option == 1:
    tablero.objetivoAleatorio()

elif option == 2:
```



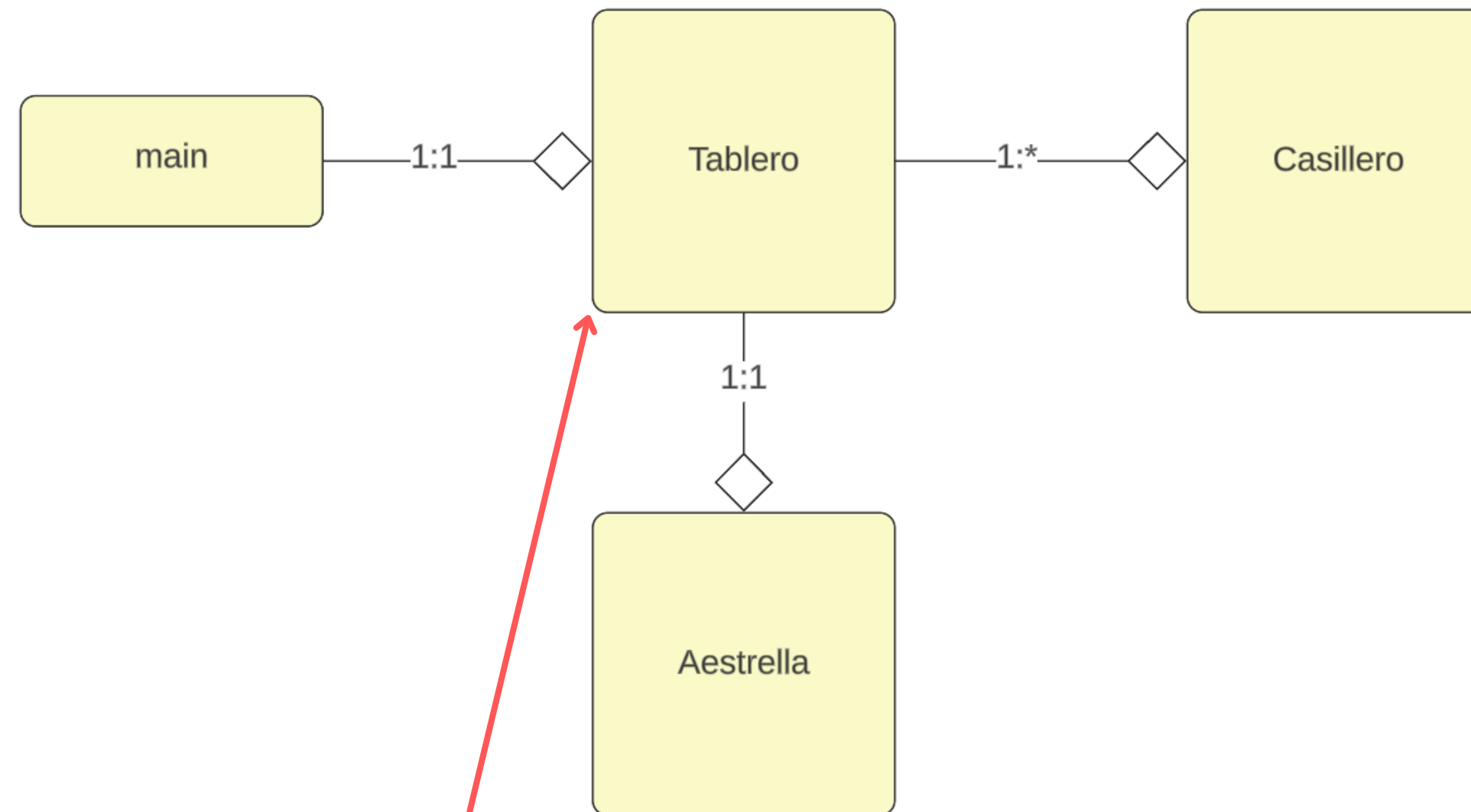
900x1,500 PNG (24-bit color) 74,68 kB

```
[3] Reconner
Ingrese una opción: 3
=====
Reconriendo...
Camino encontrado para A. Imprimiendo camino.
Camino A: [[10, 0], [10, 1], [10, 2], [10, 3], [10, 4], [9, 4], [8, 4], [7, 4]]
Camino encontrado para B. Imprimiendo camino.
Camino B: [[10, 12], [10, 11], [10, 10], [10, 9], [10, 8], [10, 7], [10, 6], [10, 5], [10, 4], [9, 4], [8, 4], [7, 4]]
```

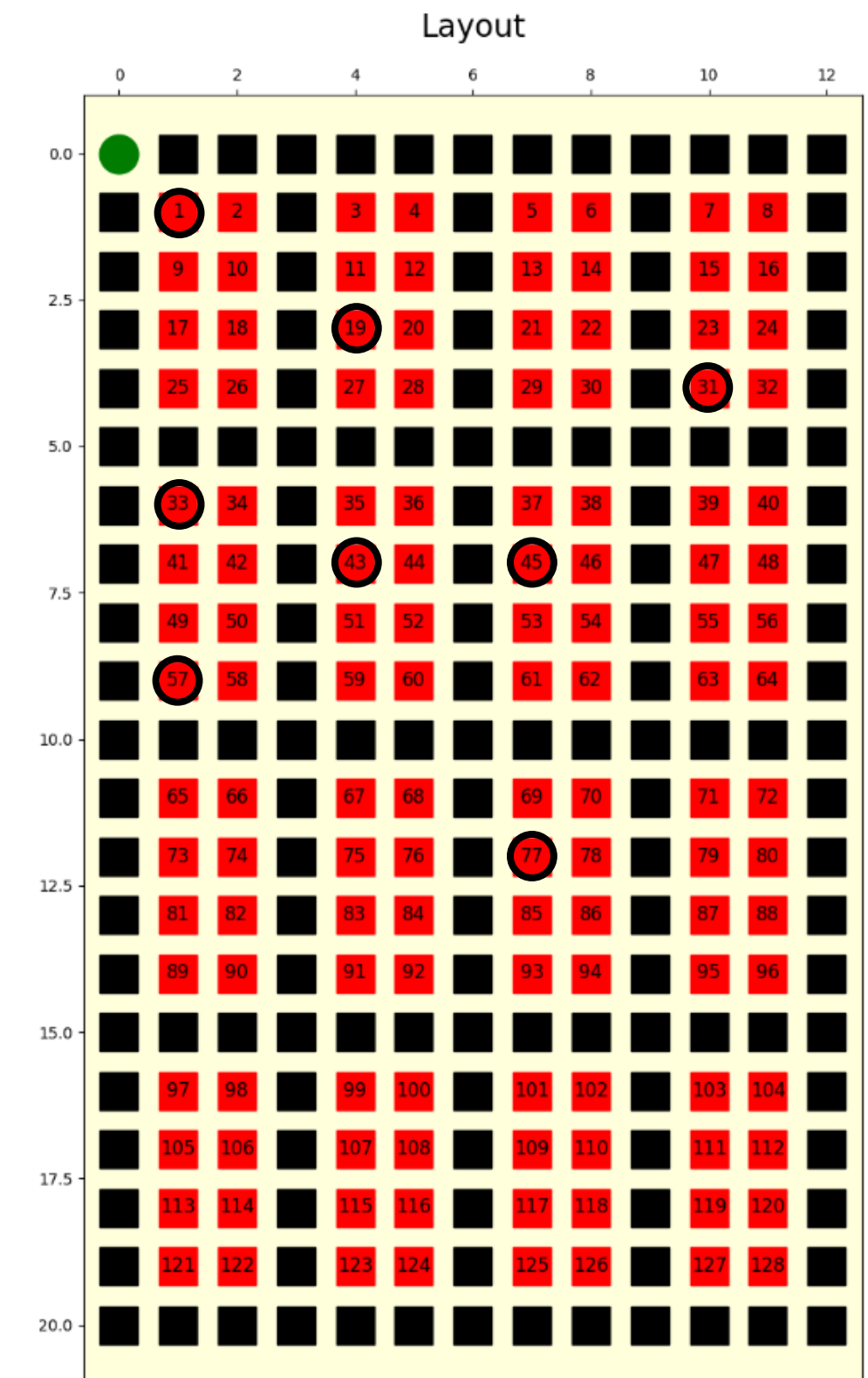
EJERCICIO 3: Recocido simulado para optimización de recogida de múltiples pedidos



Ejercicio 3 - Diagrama de bloques



Incorporación de funcionalidades



Ejercicio 3 - Generalidades

- El **agente** parte siempre desde la misma posición de carga/descarga
- Al inicio de cada ejecución se **procesa el archivo pedidos.txt** y se lo almacena como un arreglo de diccionarios en formato .json. Estos diccionarios relacionan el número de orden con el listado de pedidos.
- El usuario simplemente introduce el **número de la orden** a optimizar y recoger.
- Se limita la cantidad de pedidos dentro de una orden a **16**.

Ejercicio 3 - Código

Regeneración de pedidos.txt -> pedidos.json

```
datos = file.read()

allPedidos = []
pedidos = []

counter = 0

for line in datos.split("\n"):

    if "Order" in line:

        counter = 0
        numero_orden = int(line.replace(__old: "Order ", __new: ""))
        pedidos = []

    elif line == "\n" or line == "" or line == " " or line == " \n":

        pedidos.append(int(line.replace(__old: "P", __new: "")))

with open("./data/pedidos.json", "w") as file:
```

Interpretación de entrada de usuario (número de la orden)

```
def tomarPedido(self, pedido):  
  
    try:  
        pedido = int(pedido)  
  
    except ValueError:  
        print(f"{RED}El pedido debe ser un número entero{RESET}")  
        return  
  
    with open("./data/pedidos.json", "r") as file:  
        datos = json.load(file)  
  
    ok = 0  
  
    for orden in datos:  
        if orden["orden"] == pedido:  
            ok = 1  
  
    if not ok:  
        print(f"{RED}El pedido no existe{RESET}")  
        return  
  
    self.pedido = datos[int(pedido) - 1]["pedidos"]  
  
    self.reordenarPedido()  
  
    pass
```


Recocido simulado para optimización del orden de los pedidos (1)

```
def reordenarPedido(self):  
  
    self.coordsObj = []  
    self.plotearTablero()  
  
    T0 = 50 # No mejoró por subirla  
    Tf = 0.1 # No mejoró por disminuir  
    alpha = 0.90 # 0.9 es la mejor relacion costo-tiempo  
    T = T0  
    L = 15 # 15 es la mejor relación costo-tiempo.  
           # Probar L=10 o L=5 da costos mayores, L=20 es muy costoso en tiempo  
  
    start_time = time.time()  
  
    costo_actual, caminos = self.costoPlan()  
  
    mejor_orden = copy.deepcopy(self.pedido)  
    mejores_caminos = copy.deepcopy(caminos)  
  
    counter = 0
```

Recocido simulado para optimización del orden de los pedidos (2)

```
while Tf <= T:

    for i in range(1, L):

        counter += 1

        random.shuffle(self.pedido)

        costo_nuevo, caminos = self.costoPlan()

        delta = costo_nuevo - costo_actual

        if delta < 0:

            mejor_orden = copy.deepcopy(self.pedido)
            mejores_caminos = copy.deepcopy(caminos)
            costo_actual = costo_nuevo

        elif random.random() <= np.exp(-delta/T):

            mejor_orden = copy.deepcopy(self.pedido)
            mejores_caminos = copy.deepcopy(caminos)
            costo_actual = costo_nuevo

    T = alpha * T
```

```
self.pedido = copy.deepcopy(mejor_orden)

end_time = time.time()

print(f"=====")
print(f"{GREEN}Pedido exitosamente optimizado luego de {RESET}{counter}{GREEN} iteraciones.{RESET}")
print(f"{GREEN}Plan: {RESET}{self.pedido}")
print(f"{GREEN}Costo del plan: {RESET}{costo_actual}")
print(f"{GREEN}Tiempo de ejecución: {RESET}{end_time - start_time}")
print(f"{GREEN}Imprimiendo caminos...{RESET}")

for i in range(len(caminos)):

    self.camino = mejores_caminos[i]
    self.plotearTablero()
    input("(Enter)")
    self.agente = self.camino[-1]
    self.camino = []
    self.plotearTablero()

self.camino = []
self.plotearTablero()
```

Función de cálculo del costo del plan

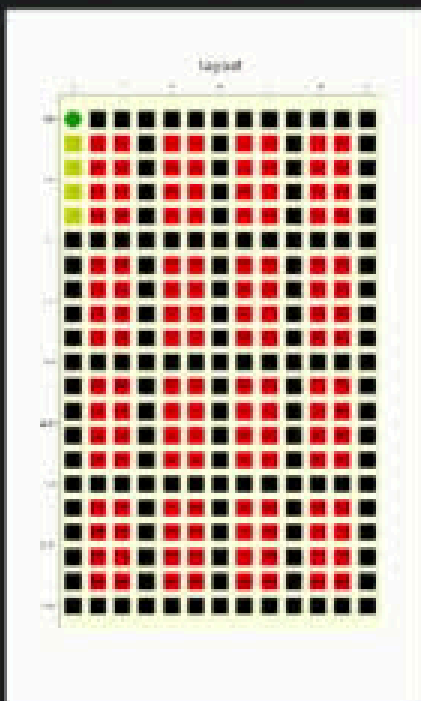
```
def costoPlan(self):  
  
    costo = 0  
  
    actual = self.agente  
    objetivo_actual = None  
  
    caminos = []  
  
    for i in range(self.filas):  
        for j in range(self.columnas):  
            if self.tablero[i][j].getAlias() == self.pedido[0]:  
                objetivo_actual = [i, j]  
  
        a_estrella = Aestrella(actual, objetivo_actual, self.tablero)  
        costo += a_estrella.getCosto()  
        caminos.append(a_estrella.getCamino())  
  
    for i in range(len(self.pedido)):  
  
        if i == 0:  
            continue  
  
        actual = caminos[-1][-1]  
        objetivo_actual = None
```

```
        for j in range(self.filas):  
            for k in range(self.columnas):  
                if self.tablero[j][k].getAlias() == self.pedido[i]:  
                    objetivo_actual = [j, k]  
  
        a_estrella = Aestrella(actual, objetivo_actual, self.tablero)  
        costo += a_estrella.getCosto()  
        caminos.append(a_estrella.getCamino())  
  
        #print(f"{GREEN}--> Plan a evaluar: {RESET}{self.pedido}")  
        #print(f"{GREEN}--> Calculo del costo del plan: {RESET}{costo}")  
  
    return [costo, caminos]
```

Ejercicio 3 - Posibles mejoras

- Aplicar conocimientos del contexto del problema para una resolución con menor costo computacional o más óptima.

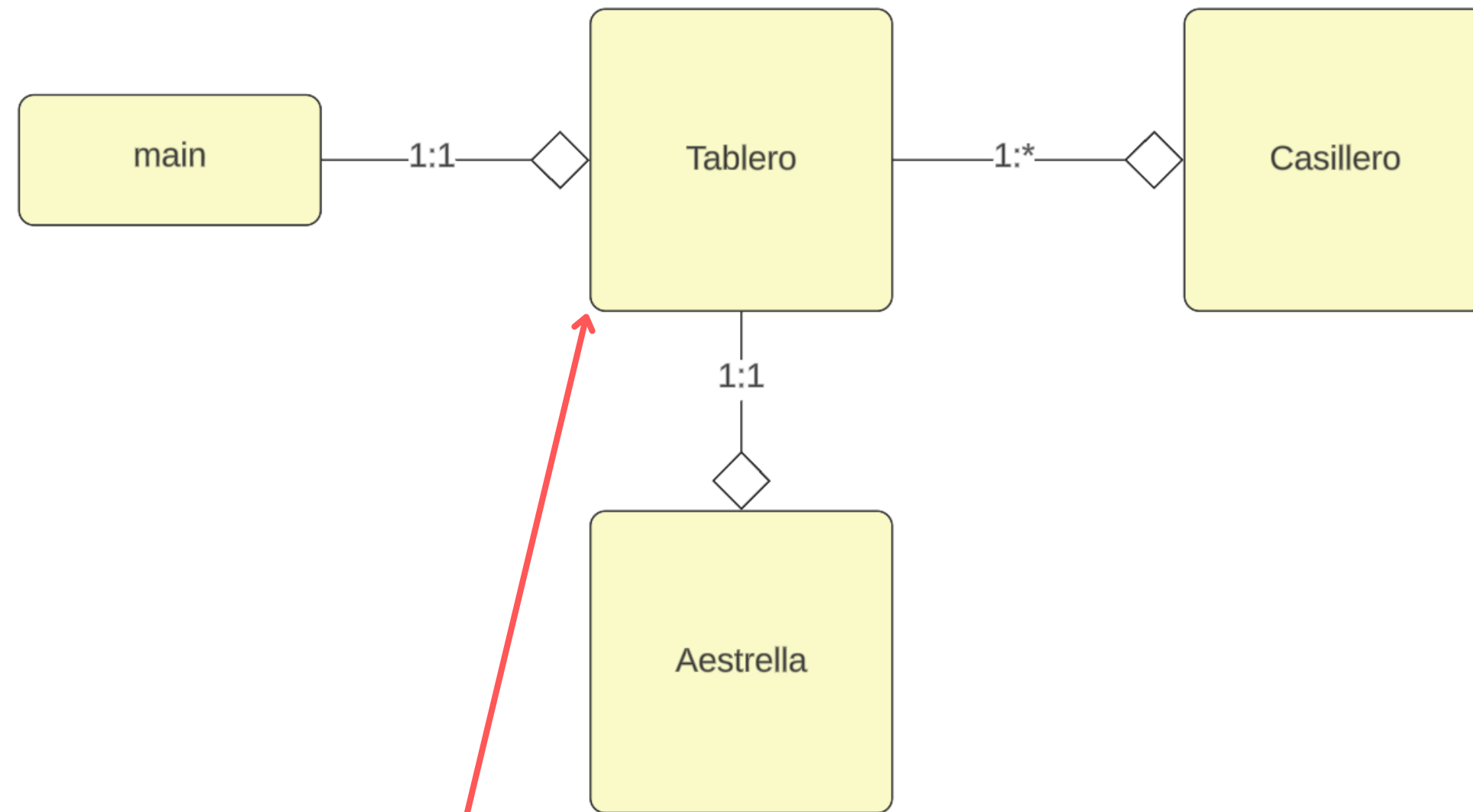
900x1,500 PNG (24-bit color) 71,33 kB



EJERCICIO 4: Algoritmo genético para ordenar estanterías de manera eficiente



Ejercicio 4 - Diagrama de bloques

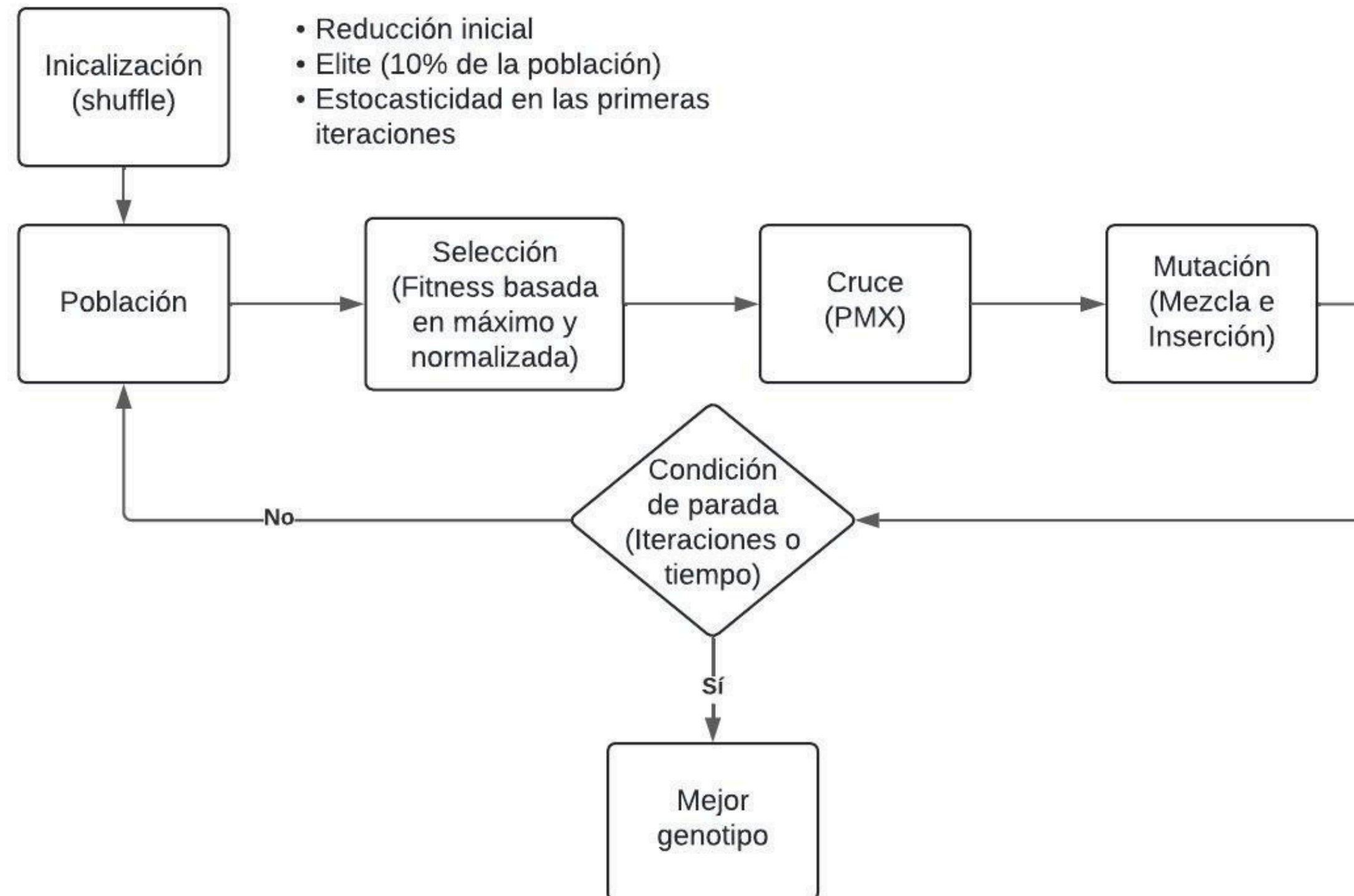


Incorporación de funcionalidades

Ejercicio 4 - Generalidades

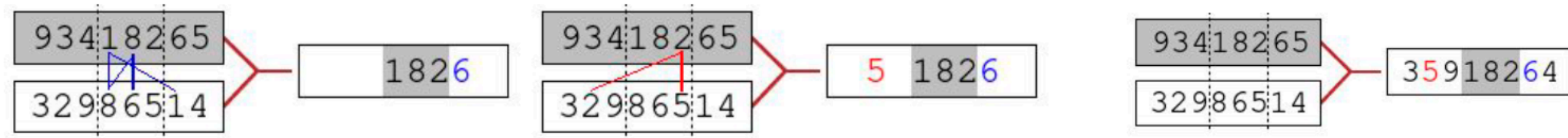
- El agente inicia y termina en la misma posición.
- Se toman varios pedidos a partir de los cuales se va a intentar **optimizar la distribución** de estanterías.
- Las estanterías se optimizan para la **media de los pedidos**, los pedidos atípicos serán muy costosos.
- Hay una relación de compromiso entre el tamaño de la población y la cantidad de generaciones.
- Trabajamos con un **genoma reducido**, por cuestiones de procesamiento

Ejercicio 4 - Diagrama de flujo



Ejercicio 4 - Código

- Crossover mediante:
 - PMX



Ejercicio 4 - Código

- Función de fitness

```
for genotipo in genotipos:

    print(f"{GREEN}--> Evaluando genotipo: {RESET}{genotipo}")

    self.reasignarEstantes(genotipo)

    costo_dist = 0

    with open(pedidos_json_path, "r") as file:
        datos = json.load(file)

    for orden in datos:

        self.pedido = orden["pedidos"]

        costo_dist += self.reordenarPedido(printMessage=False)

    fitness_vec.append(costo_dist)
    print(f"{GREEN}--> Costo del genotipo: {RESET}{costo_dist}")

fit_max = max(fitness_vec)
prom_norm = 0

for i in range(len(fitness_vec)):
    prom_norm += fit_max - fitness_vec[i]

for i in range(len(fitness_vec)):
    fitness_vec[i] = (fit_max - fitness_vec[i])/prom_norm
    fitness_vec[i] = round(fitness_vec[i], 4)

return fitness_vec
```

Ejercicio 4 - Código

- Mutación

Codiumate: Options | Test this method

```
def mezcla(self, hijo, cant_mut):  
    for _ in range(cant_mut):  
        idx1 = random.randint(0, len(hijo) - 1)  
        idx2 = random.randint(0, len(hijo) - 1)  
        hijo[idx1], hijo[idx2] = hijo[idx2], hijo[idx1]  
    return hijo
```

```
def intercambio(self, hijo):  
  
    idx1 = random.randint(0, len(hijo) - 1)  
    idx2 = random.randint(0, len(hijo) - 1)  
  
    hijo[idx1], hijo[idx2] = hijo[idx2], hijo[idx1]  
  
    return hijo
```

Codiumate: Options | Test this method

```
def insercion(self, hijo):  
  
    idx1 = random.randint(0, len(hijo) - 1)  
    idx2 = random.randint(0, len(hijo) - 1)  
  
    if idx1 == idx2:  
        return hijo  
  
    if idx1 > idx2:  
        idx1, idx2 = idx2, idx1  
  
    subsec = hijo[idx1:idx2+1]  
  
    newsec = [subsec[0], subsec[-1], *subsec[1:-1]]  
  
    hijo = [*hijo[:idx1], *newsec, *hijo[idx2+1:]]  
  
    return hijo
```


Ejercicio 4 - Código

- Evolución

```
for j in range(elite_size):
    nuevos_genotipos[j] = copy.deepcopy(genotipos[j])

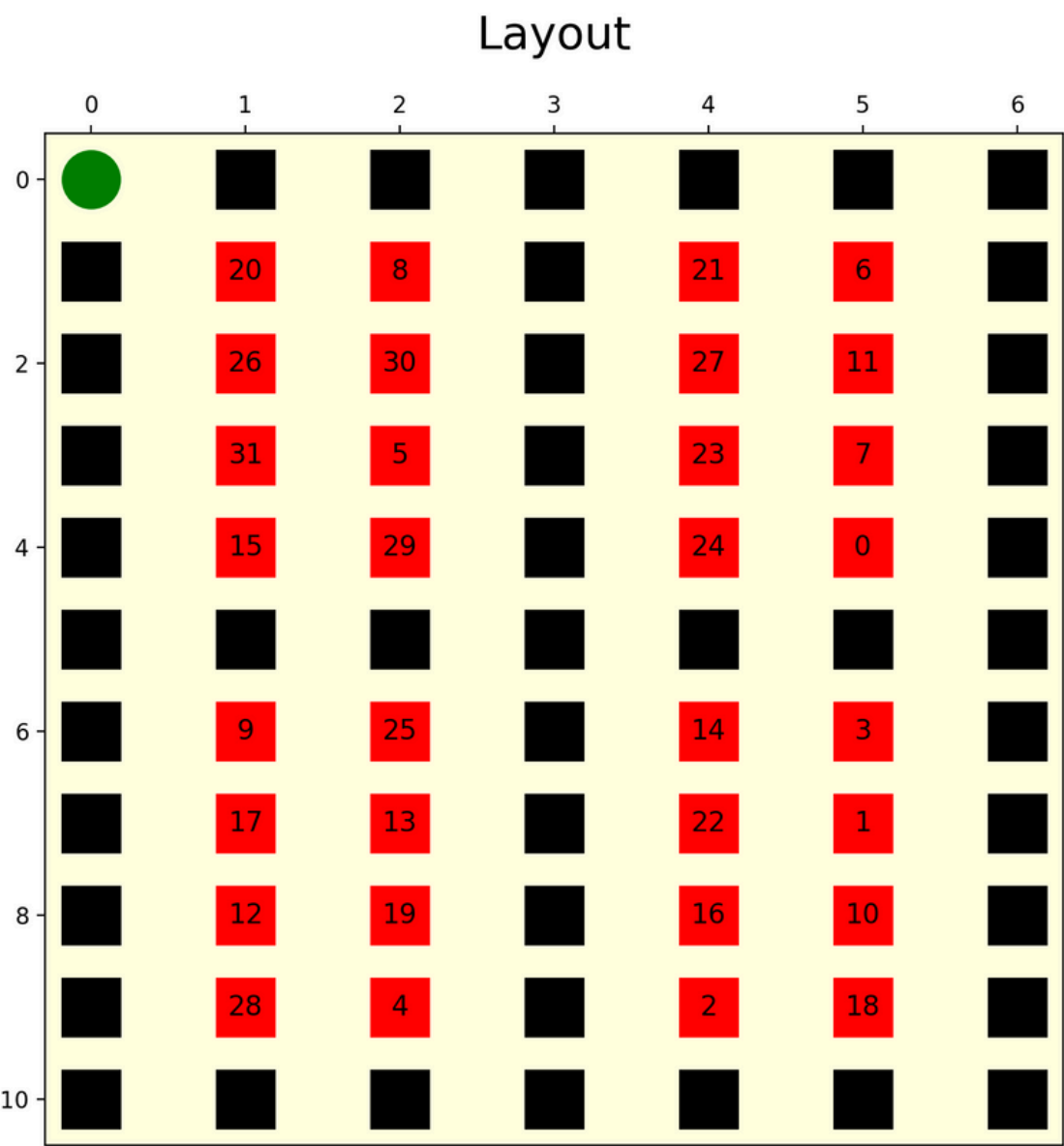
for j in range(0, N-elite_size):

    hijo1, hijo2 = self.PMX(genotipos[j], genotipos[j+1])

    if i < 5:
        hijo1 = self.mezcla(hijo1,24)
        hijo2 = self.mezcla(hijo2,24)
    elif i < 10:
        hijo1 = self.mezcla(hijo1,16)
        hijo2 = self.mezcla(hijo2,16)
    elif i < 15 and random.random() <= 0.8:
        hijo1 = self.mezcla(hijo1,8)
        hijo2 = self.mezcla(hijo2,8)
        hijo1 = self.insercion(hijo1)
        hijo2 = self.insercion(hijo2)
    elif i < 20 and random.random() <= 0.6:
        hijo1 = self.mezcla(hijo1,6)
        hijo2 = self.mezcla(hijo2,6)
        hijo1 = self.insercion(hijo1)
        hijo2 = self.insercion(hijo2)
    elif i < 25 and random.random() <= 0.4:
        hijo1 = self.mezcla(hijo1,4)
        hijo2 = self.mezcla(hijo2,4)
    elif i < 30 and random.random() <= 0.2:
        hijo1 = self.intercambio(hijo1)
        hijo2 = self.intercambio(hijo2)
```

Ejercicio 4 - Resultado

```
Fitness de la población generada en la iteración 19: [0.0365, 0.0356, 0.0361, 0.034, 0.0336, 0.0236, 0.0278, 0.0274, 0.0327, 0.0303, 0.0195, 0.0294, 0.0236, 0.0278, 0.0269, 0.0282, 0.0211, 0.0211, 0.0211, 0.0245, 0.0186, 0.0203, 0.0232, 0.0128, 0.0153, 0.0195, 0.0236, 0.0017, 0.0149, 0.0162, 0.0162, 0.0162, 0.0236, 0.0195, 0.017, 0.0245, 0.0149, 0.0286, 0.0099, 0.0, 0.017, 0.0075, 0.0128, 0.0079, 0.0116, 0.0128, 0.0137, 0.0037, 0.0141, 0.0017]
=====
Optimización finalizada
Tiempo de ejecución: 770.9584090709686
Genotipo final: [20, 8, 21, 6, 26, 30, 27, 11, 31, 5, 23, 7, 15, 29, 24, 0, 9, 25, 14, 3, 17, 13, 22, 1, 12, 19, 16, 10, 28, 4, 2, 18]
```



Ejercicio 4 - Posibles mejoras

- Mejorar el **eficiencia y eficacia** del código.
- Reducir los tiempos de ejecución.
- Encontrar punto de equilibrio entre los hiperparámetros.
- Mejor elección de padres e implementación óptima de crossosover y mutaciones.

¡Gracias!

Ignacio Martín Duci

Tomás Valentín Guevara Gaspari

Martín Cazabán

Martina Roby Culasso

Johannes Groß