Juan Carlos Calderón García A01625696

Actividad Integradora (Multiagentes)

Estrategia:

La estrategia que implementé implica que el agente tome una caja como home para acomodar cajas subsecuentes sobre la misma caja hasta que haya apilado 5 cajas, tras esto busca una nueva caja para hacer su home y repetir el proceso.

Esto lo logré dividiendo las acciones del agente en 2 funciones search y sendHome

Las variables más importantes dentro del agente son:

```
#Si tiene una caja o no
    self.holding = False
#Si regresé en X o no
    self.returnX = False
#Si regresé en Y o no
    self.returnY = False
#La posición de home
    self.home = None
```

Cada step decide que función elegir, se va a search si no tiene un home o si no está tomando algo, sino debería que tanto estar sosteniendo algo como tener un home y se mueve a findHome

```
if not self.holding or self.home == None:
      self.search()
    else:
      self.findHome()
```

Search es donde se utiliza la mayoría del código del autómata con estrategia aleatoria, tomando una celda de sus vecinos para analizar y decidir que hacer sobre ella, la diferencia está en que primero va a revisar todas las celdas para asegurarse de que no tenga cajas en su vecindad y solo si no hay cajas se va a mover a la primera posición válida que haya encontrado

```
def search(self):
    neighbors = self.model.grid.get_neighborhood(self.pos,
moore=False, include_center=False)
    acting = False
    chX, chY = (-1, -1)
    first = (-1, -1)

    #Repite mientras no haya actuado y tenga vecinos para revisar
```

```
    while not acting and neighbors:
      #Selecciona una ubicación al azar para revisar y la elimina de
la lista de posibles
      chX, chY = self.random.choice(neighbors)
      neighbors = [x for x in neighbors if x != (chX, chY)]

      #Si la celda está vacía hace el resto de comprobaciones
      if self.model.grid.is_cell_empty((chX, chY)):
        #Guardo la primera posición seleccionada si no he guardado una
antes y está vacía
        if first == (-1, -1) and self.model.boxes[chX][chY] == 0:
          first = (chX, chY)
        #Si no ha guardado una caja para hacer de home y home está
vacío la guarda como home
        if self.model.boxes[chX][chY] > 0 and
self.model.boxes[chX][chY] < 5 and self.home == None:
          self.home = (chX, chY)
        #Si hay una caja diferente a home la toma y no está
sosteniendo nada toma una caja
        elif self.model.boxes[chX][chY] > 0 and
self.model.boxes[chX][chY] < 5 and (chX, chY) != self.home and not
self.holding:
          self.model.boxes[chX][chY] -= 1
          self.holding = True
          acting = True
        #Si no puede tomar ninguna caja pone una posición en chX y chY
        elif not neighbors:
          chX, chY = first

    #chX y chY son mayores o iguales a 0 cuando tienen una coordenada
válida guardada y no se hizo ninguna acción
    if chX >= 0 and chY >= 0 and self.model.grid.is_cell_empty((chX,
chY)) and not acting and (chX, chY) != self.home:
      self.model.grid.move_agent(self, (chX, chY))
      self.steps_moving += 1
```

La lógica de findHome permite que el autómata pueda navegar hacia de regreso hacia la misma, lo primero que hace es decidir si se va a mover en dirección a X o Y dependiendo de cuál dirección es la más lejana una vez decidida la dirección pasa a moveDir un valor de 1 o -1 dependiendo de la dirección hacia la que se va a mover

```
def findHome(self):
    py,px = self.pos
    hy,hx = self.home
```

```
    distX = hx - px
    distY = hy - py

    if abs(distX) >= abs(distY):
      if distY == 0:
        distY = 1
      if distX == 0:
        self.moveDir(0, 0, int(abs(distY) / distY))
      else:
        self.moveDir(int(abs(distX) / distX), 0, int(abs(distY) /
distY))
    elif abs(distX) < abs(distY):
      if distX == 0:
        distX = 1
      self.moveDir(0, int(abs(distY) / distY), int(abs(distX) /
distX))
```

MoveDir tiene toda la lógica para decidir cómo se va a mover el agente al principio hay un if que da lugar a la lógica correspondiente a cada eje , a continuación está la lógica para uno de los ejes ya que ambos usan la misma lógica

```
if abs(x) > abs(y) and self.model.grid.is_cell_empty((py, px + x)):
      #Si la posición siguiente es home haz las revisiones para home
      if (py, px + x) == self.home:
          self.checkHome(py, px + x)
      else:
        #Si la posición está dentro del rango del espacio
        if (px + x) < self.model.grid.width and (px + x) >= 0:
          #Si la posición está disponible y no me he movido hacia
atrás me muevo a esa posición
          if self.model.boxes[py][px + x] == 0 and
self.model.grid.is_cell_empty((py, px + x)) and not self.returnX:
            self.model.grid.move_agent(self, (py, px + x))
            self.steps_moving += 1
          #Si la posición no estaba disponible me muevo en la
dirección de repuesto y si me había regresado muestro que ya me moví
hacia atrás
          elif (py + backup_mag) >= 0 and (py + backup_mag) <
self.model.grid.width and self.model.boxes[py + backup_mag][px] == 0
and self.model.grid.is_cell_empty((py + backup_mag, px)):
            if self.returnX:
              self.returnX = False
            self.model.grid.move_agent(self, (py + backup_mag, px))
            self.steps_moving += 1
```

```
          #Lo mismo que el anterior, pero en caso de que la dirección
inicial esté ocupada
          elif (py - backup_mag) >= 0 and (py - backup_mag) <
self.model.grid.width and self.model.boxes[py - backup_mag][px] == 0
and self.model.grid.is_cell_empty((py - backup_mag, px)):
            if self.returnX:
              self.returnX = False
            self.model.grid.move_agent(self, (py - backup_mag, px))
            self.steps_moving += 1
          #Si todo falló regreso una posición y marco que regresé para
la siguiente comprobación
          elif self.model.boxes[py][px + x] > 0 or not
self.model.grid.is_cell_empty((py, px + x)):
            if (px - x) >= 0 and (px - x) < self.model.grid.width and
self.model.grid.is_cell_empty((py, px - x)) and
self.model.boxes[py][px - x] == 0:
              self.model.grid.move_agent(self, (py, px - x))
              self.returnX = True
              self.steps_moving += 1
        else:
          #Si no me muevo una posición en el eje opuesto hacia adentro
del espacio
          if (py - backup_mag) >= 0 and (py - backup_mag) <
self.model.grid.width and self.model.boxes[py +- backup_mag][px] == 0
and self.model.grid.is_cell_empty((py - backup_mag, px)):
            self.model.grid.move_agent(self, (py - backup_mag, px))
          elif (py + backup_mag) >= 0 and (py + backup_mag) <
self.model.grid.width and self.model.boxes[py + backup_mag][px] == 0
and self.model.grid.is_cell_empty((py + backup_mag, px)):
            self.model.grid.move_agent(self, (py + backup_mag, px))
```

En caso de que la posición siguiente sea home va a añadir una caja a la pila y marcar que ya no tiene una caja

```
def checkHome(self, posY, posX):
    if (posY, posX) == self.home and self.holding:
      if self.model.boxes[posY][posX] < 5:
        self.model.boxes[posY][posX] += 1
        self.holding = False
        self.sorted += 1
```

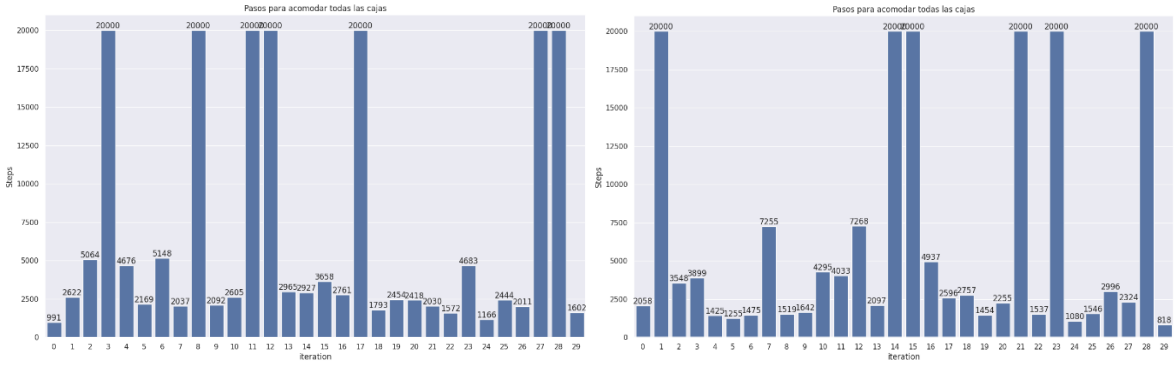
Y en cada paso va a revisar si tiene que reiniciar home para buscar un nuevo home o no

```
if self.home != None:
      homeX, homeY = self.home
```

```
        if self.model.boxes[homeX][homeY] >= 5:
            self.home = None
```

Comparación:

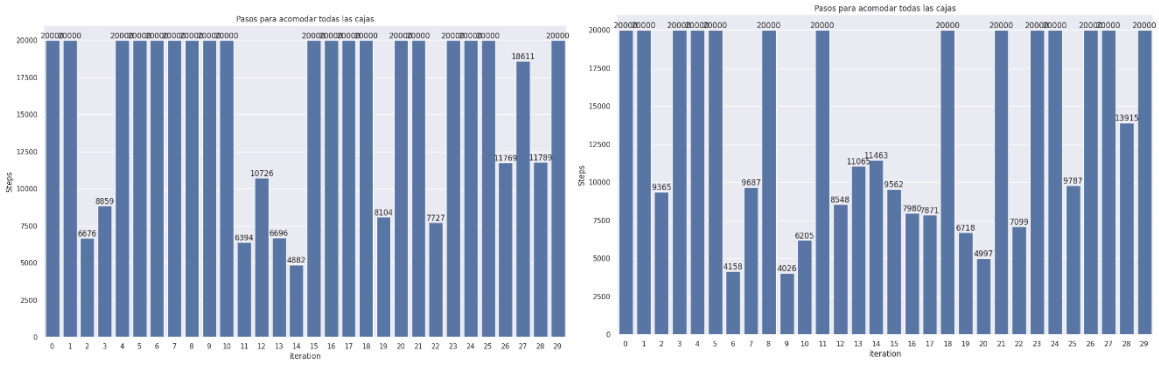
Con estrategia:



Promedio con no completados: 6164.3

Promedio sin no completados: 2337.4

Random:



Promedio con no completados: 14759.1

Promedio sin no completados: 8353.5

Con estas gráficas podemos ver que en general la solución con estrategia logra terminar en más casos y en promedio los casos que si se logran completar son más rápidos que los de las simulaciones aleatorias