



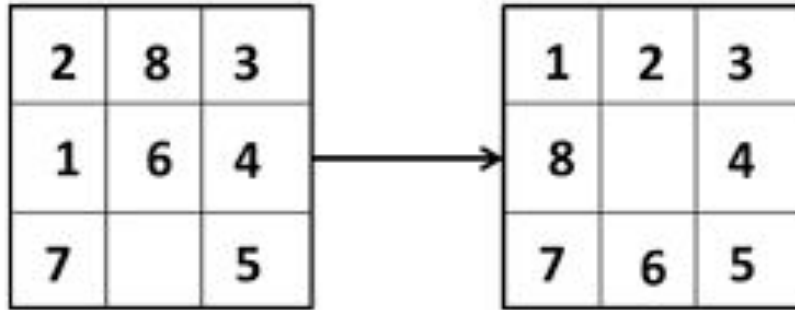
# SIA - TP1

## Métodos de Búsqueda

Grupo 1

Alberto Bendayan  
Tobias Ves Losada  
Cristian Tepedino  
Luca Bloise

# Ejercicio 1 - “8-puzzle”



Initial State

Goal State



# Estructura de estado y acciones

- Como estado, se tendrá una matriz de 3x3 de enteros, con los números del 1 al 8 representando las respectivas piezas del puzzle y el número 0 indicando el espacio vacío.
- El espacio de acciones está conformado por 4 direcciones: arriba, abajo, izquierda y derecha. Se describe por la ubicación de la pieza relativa al espacio vacío en el tablero previo a ejecutar la acción.
- En cuanto a los costos de las acciones, consideramos que cualquier movimiento realizado es un costo idéntico. Dado que no definimos ninguna meta-acción que encapsule a más de un movimiento, decidimos que el costo de toda acción sea 1.

$$S = \{s \in \mathbb{N}^{3 \times 3} \mid s \text{ contiene a los números del 0 al 8} \}$$



# Primera Heurística

## Cantidad de piezas fuera de lugar

- La primera heurística que consideramos es aquella que estima basándose en la cantidad de piezas que estén “fuera de lugar”. Es decir, aquellas en las que su posición en el estado actual no coincida con la posición de la pieza en el tablero solución.
- Si hay  $N$  piezas fuera de lugar, en el mejor de los casos se necesitan  $N$  acciones para llegar a la solución. En este caso la función heurística valdrá también  $N$ .
- En cualquier otro caso el acomodar las piezas tomaría más de un movimiento, lo que significa que el costo de acomodarlas es mayor que el valor de la heurística.
- Dado que para todo estado la estimación de la heurística es menor al costo de la solución óptima, la heurística es admisible.



## Segunda Heurística

### Suma de distancias Manhattan

- Se calcula mediante la suma de las distancias Manhattan entre la posición actual de las fichas y su posición en la solución.
- Para cada pieza se cumple  $D(\text{pieza}, \text{solución}) = N$ . Se necesitan al menos  $N$  acciones para llevar esa pieza a su lugar en la solución
- Si extendemos esto a todos las fichas y soluciones del tablero, podemos comprobar la admisibilidad:

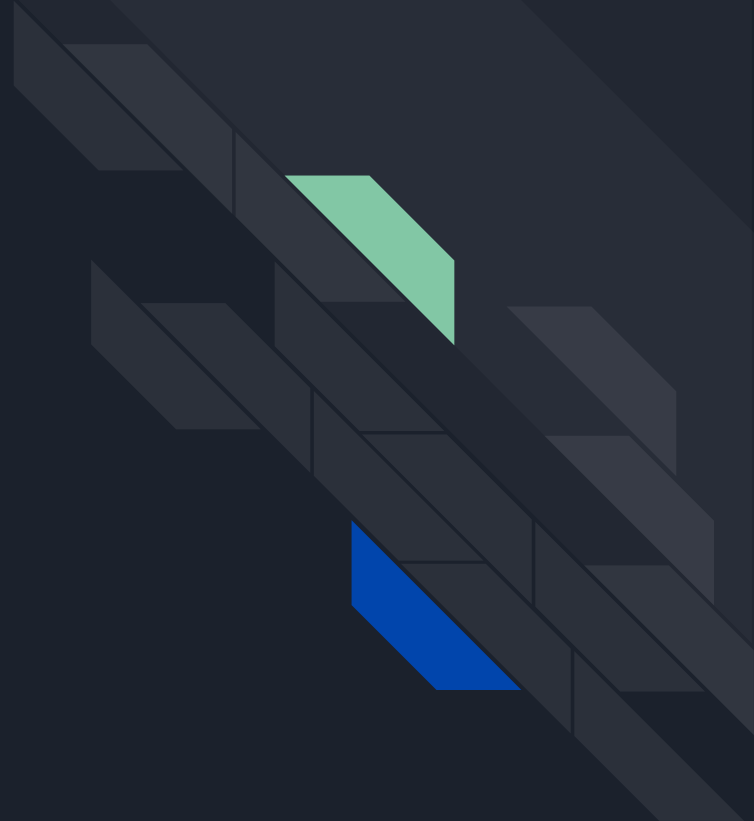
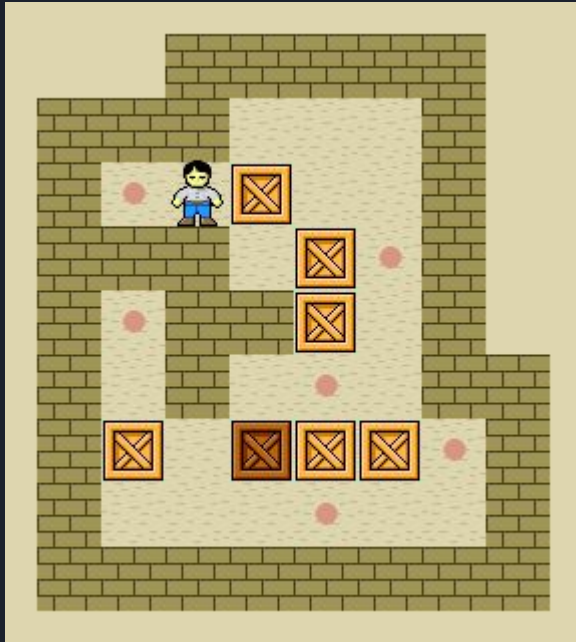
$$h_2(e) = \sum_{i=1}^8 d_1(\text{ficha}_i, \text{solución}_i) \leq |\text{camino desde } e \text{ hasta solución óptima}| = \text{goal}_g(e)$$



# Método de búsqueda y heurística a utilizar A\* con distancia Manhattan

- Dado que tenemos una heurística admisible, las mejores opciones para realizar la búsqueda son A\* e Iterative Deepening A\*.
- Como el tablero del 8-puzzle es pequeño, y hay como máximo 4 acciones por estado, el uso de memoria no sería lo suficientemente alto como para ameritar usar IDA\*. Por ende, optamos por A\*.
- Debido a que la heurística de la distancia Manhattan domina a la de la cantidad de fichas fuera de lugar, optamos por usarla en el algoritmo de búsqueda.

## Ejercicio 2 - “Sokoban”





# Estructura de estado y acciones

- El estado de un nivel de Sokoban cuenta con una parte inmutable (las paredes y objetivos) y una parte mutable (el jugador y las cajas).
- Cada uno de estos elementos se guarda como un par (x, y) indicando la posición del elemento.

```
class GameState:
    def __init__(self, player, boxes, walls, targets, path_from_start: Directions):
        self.player = player
        self.boxes = frozenset(boxes)
        self.walls = walls
        self.targets = targets
        self.path = copy(path_from_start)
```

- Las acciones posibles son los movimientos en cada dirección {Arriba, Abajo, Izquierda, Derecha}. Si en un estado el jugador tiene de un lado una pared, o una caja con una pared u otra caja detrás, no podrá moverse en esa dirección en ese estado.
- El costo de cada acción es uniformemente 1






# Métodos de búsqueda

Para la implementación, se creó una función que implementa el método de búsqueda general y recibe una colección que hace de frontera y determina el orden de extracción de los nodos según el método específico. En particular:

- BFS - Se usa una Queue
- DFS - Se usa un Stack
- IDDFS - Se usa una PriorityQueue, ordenada por nivel de iteración primero, y por el camino más largo para desempatar
- Greedy - Se usa una PriorityQueue, ordenada por el valor de la heurística usada evaluada en el estado
- A\* - Se usa una PriorityQueue, ordenada por la suma entre el costo acumulado y la heurística, y usando la heurística para desempatar



# Heurísticas

## Por distancia de las cajas

- Misplaced Boxes:

Devuelve la cantidad de cajas que no están ubicadas en un objetivo.

Es admisible, dado que en un movimiento solo se puede, como máximo, mover una caja, y cada caja que no esté en un objetivo requiere al menos un movimiento para llegar.

- Manhattan Distance Sum:

Devuelve la suma de la mínima distancia Manhattan entre cada caja y su objetivo más cercano.

Es admisible, ya que si una caja está a una distancia  $N$  del objetivo más cercano, se requieren al menos  $N$  movimientos para llevarla hasta dicho objetivo.

En el mejor de los casos, la cantidad de movimientos sería igual a la suma de distancias



# Heurísticas

## Por distancia de las cajas

- Nearest Box:

Devuelve la distancia Manhattan del personaje a la caja más cercana que no esté en un objetivo.

Es admisible, puesto que para mover una caja el personaje debe estar adyacente a ella, y en el mejor caso en que solo quede una caja no ubicada a distancia  $n$ , se requieren al menos  $n$  movimientos para llegar a la caja (y luego más para llevarla al objetivo)

- Walled Distance Sum:

Calcula el mínimo camino de una caja hacia el objetivo más cercano, sin atravesar paredes, luego, devuelve la suma de longitudes de estos caminos para cada caja.

Es admisible, ya que se requieren al menos la cantidad de movimientos dado por el camino calculado para poder llegar al objetivo, por lo tanto la suma de las distancias de todas las cajas será menor a la cantidad de movimientos necesarios.



# Heurísticas

## Situaciones de deadlock

Todas estas heurísticas devuelven infinito en caso de encontrar un deadlock, puesto que un camino que la contenga no tendrá solución alguna.

- Not Cornered:

Busca deadlocks causados por cajas que quedan en esquinas sin un objetivo.

Como el juego solo permite mover cajas empujandolas, el dejar una caja en una esquina de paredes significa que esa caja no puede volver a ser movida.

- No Square Blocks:

Busca deadlocks causados por cuadrados de 4 cajas o cajas y paredes.

Dado que no se puede empujar una caja que tenga detrás a otra caja, el agrupar 4 cajas impide que ninguna de estas pueda volver a ser empujada, por lo que, si no están en un objetivo, es imposible ganar a partir de ese estado.



# Heurísticas

## Situaciones de deadlock

Todas estas heurísticas devuelven infinito en caso de encontrar un deadlock, puesto que un camino que la contenga no tendrá solución alguna.

- Not Wall Stuck:

Busca deadlocks causados por una caja adherida a una fila o columna de paredes, cuando no hay un objetivo adyacente a dicha fila o columna que pueda alcanzar.

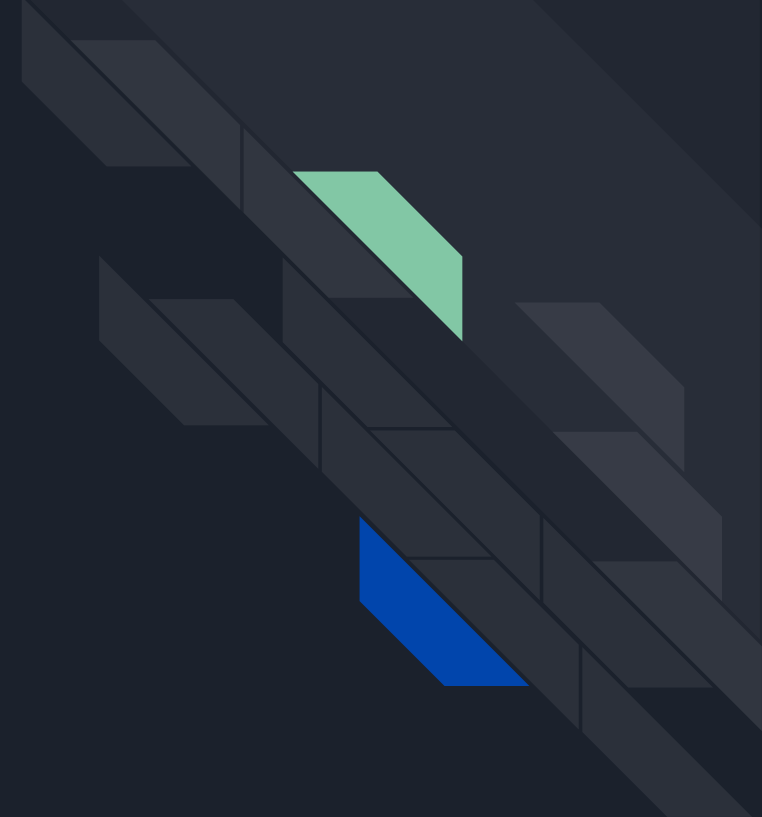
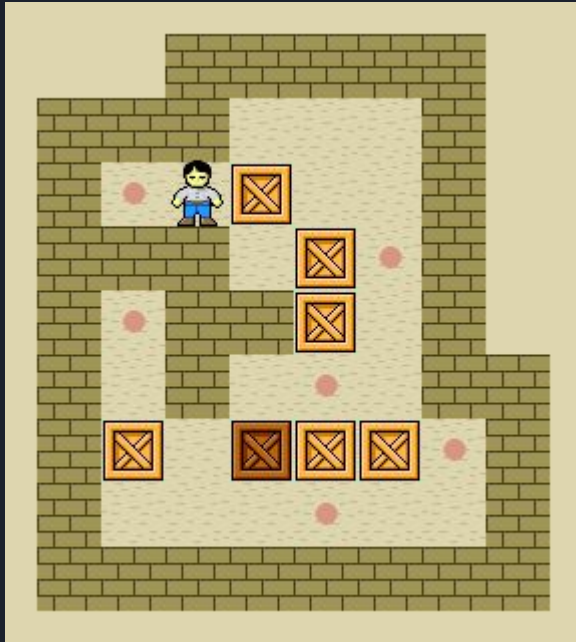
El criterio de búsqueda implica buscar, si una caja es adyacente a una pared de un lado, si existe una forma, aunque lleve múltiples movimientos, de empujar la caja en el eje x (si el lado es izquierda o derecha) o en el eje y (si es arriba o abajo) o si tiene un objetivo en sus casilleros alcanzables. Caso contrario, la caja nunca podrá despegarse de ese grupo de paredes, y el nivel no podrá completarse.



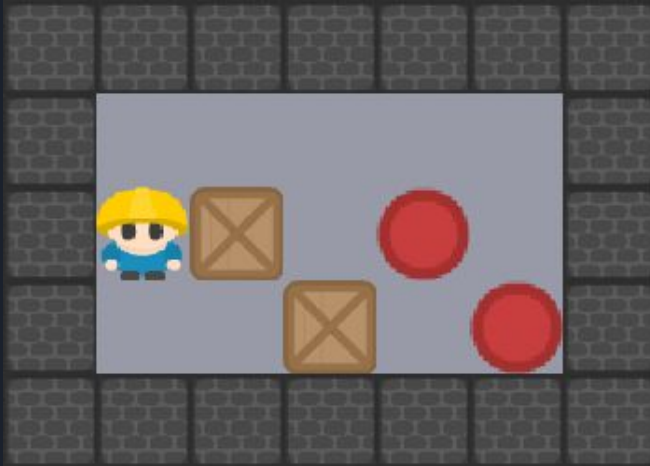
# Heurísticas Combinaciones

- Avoid Deadlocks
- Manhattan Distance Sum (Avoid Deadlocks)
- Nearest Box (Avoid Deadlocks)
- Walled Distance Sum (Avoid Deadlocks)

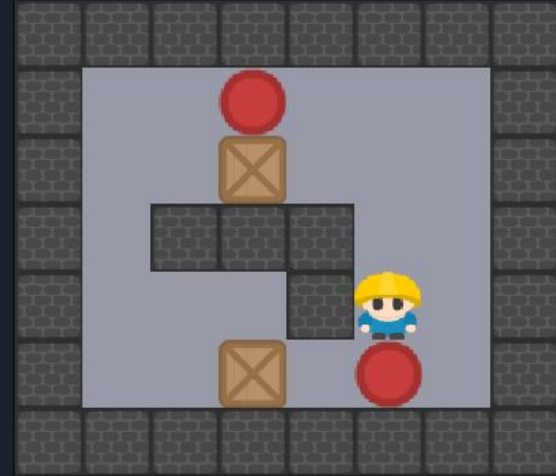
# Análisis de resultados



# Niveles utilizados



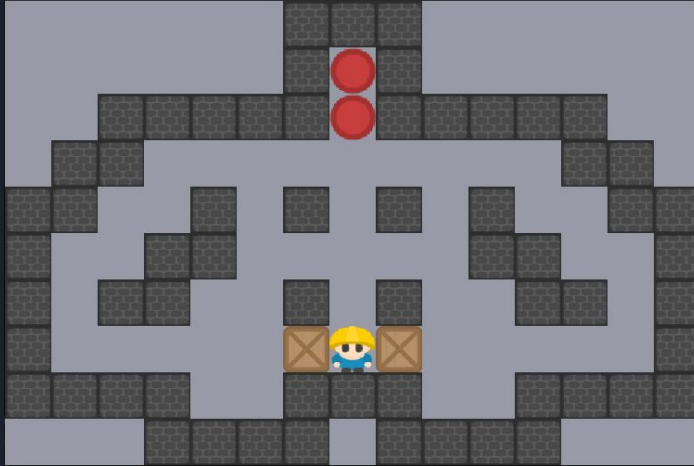
Nivel 1



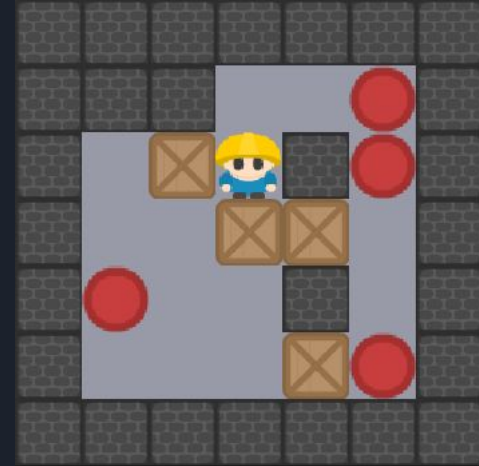
Nivel 2



## Niveles utilizados

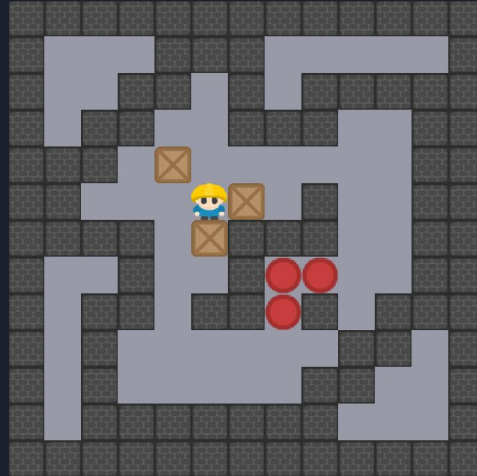


Nivel 3



Nivel 4

# Niveles utilizados



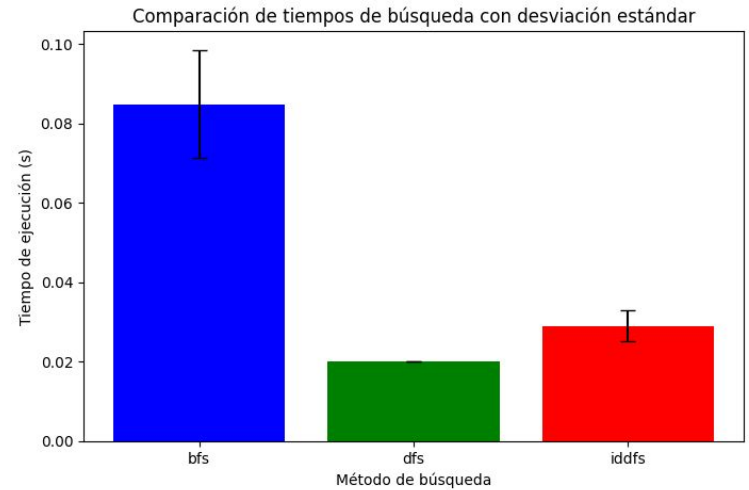
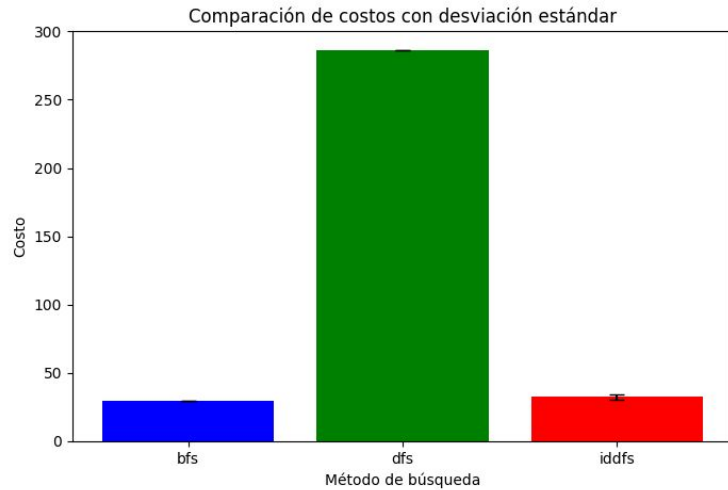
Nivel 5



# Métodos no informados

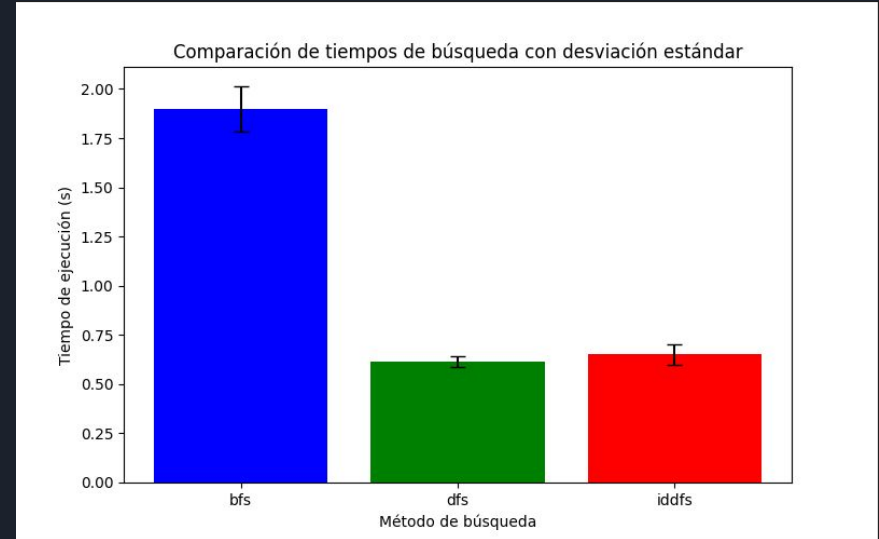
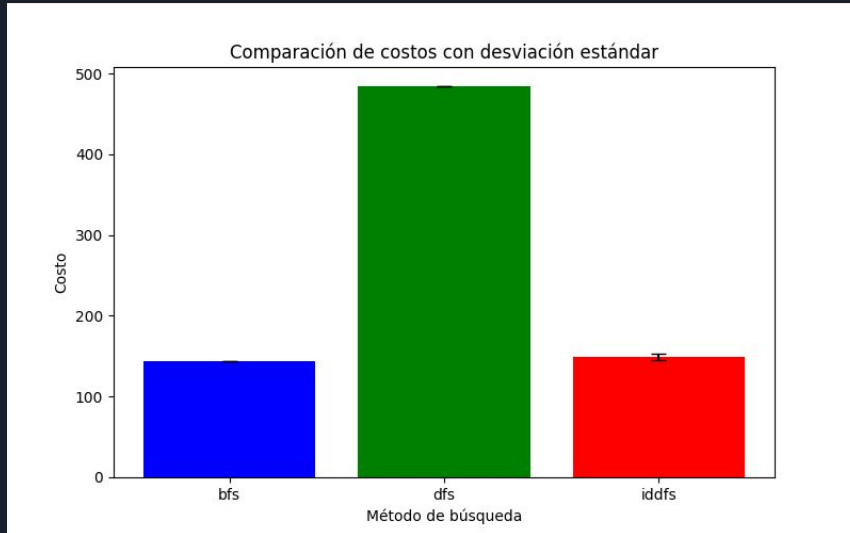
- BFS
- DFS
- IDDFS (depth 5 para casos fijos)

## Nivel 3 - Tiempo vs Costo



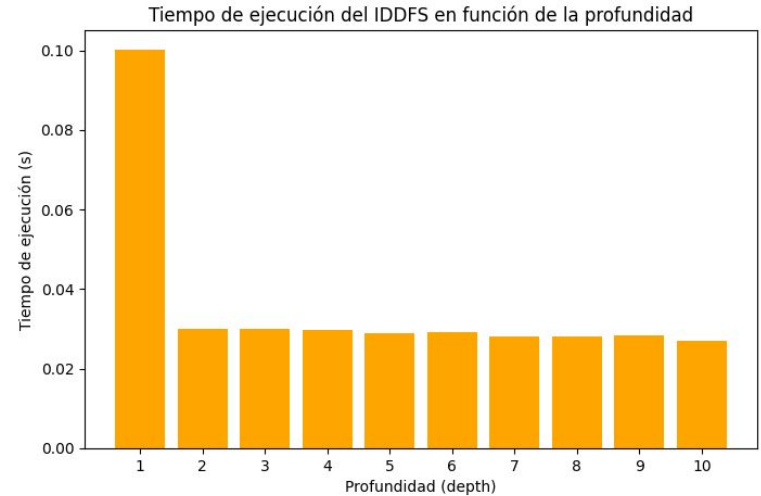
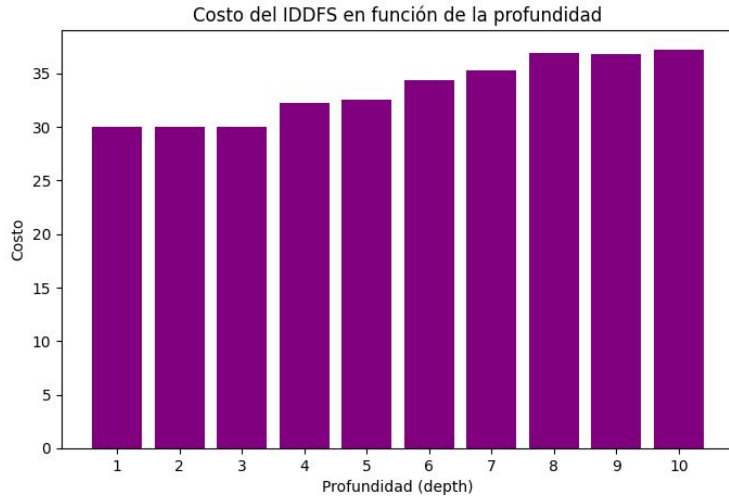
Se realizaron 1000 iteraciones por nivel y por método

## Nivel 4 - Tiempo vs Costo



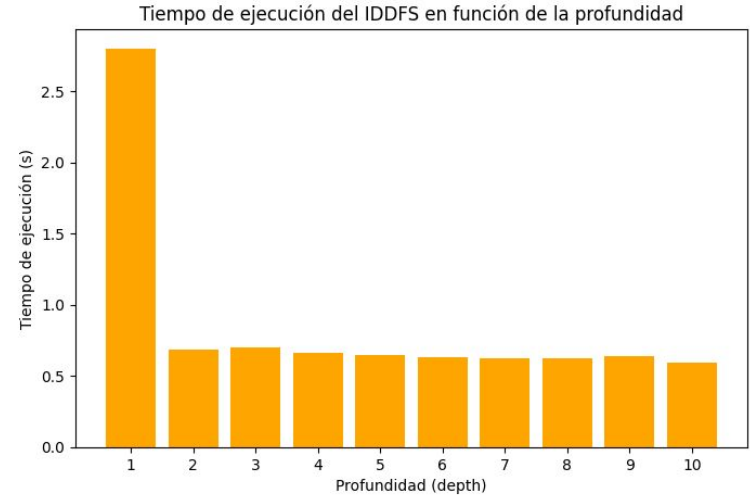
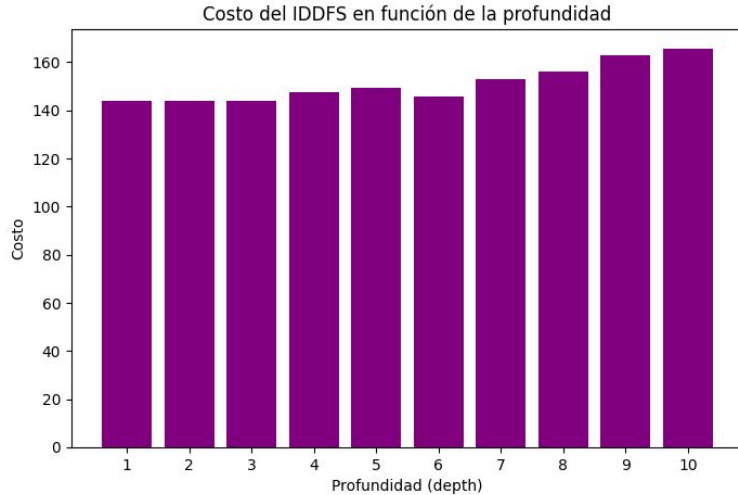
Se realizaron 1000 iteraciones por nivel y por método

## Nivel 3 - IDDFS(depth)-Tiempo vs Costo



Se realizaron 100 iteraciones por cada depth entre 1 y 10

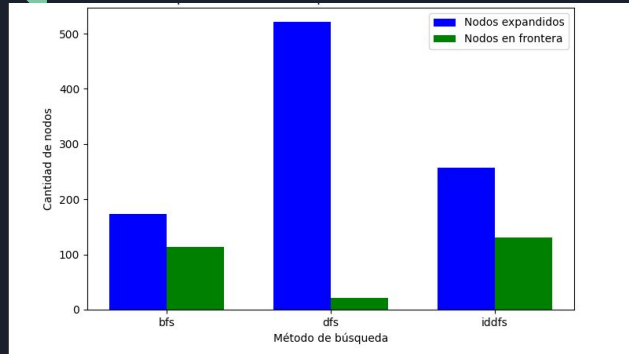
## Nivel 4 - IDDFS(depth)-Tiempo vs Costo



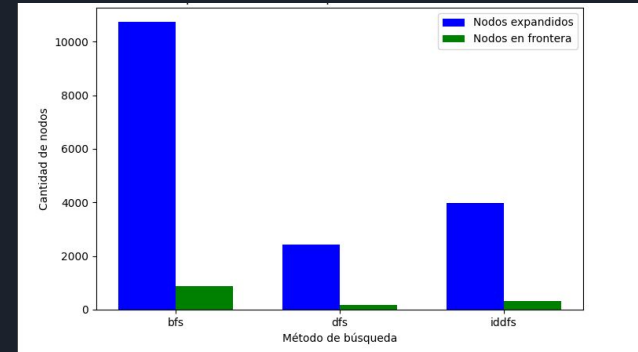
Se realizaron 100 iteraciones por cada depth entre 1 y 10

# Nodos expandidos - Nodos frontera

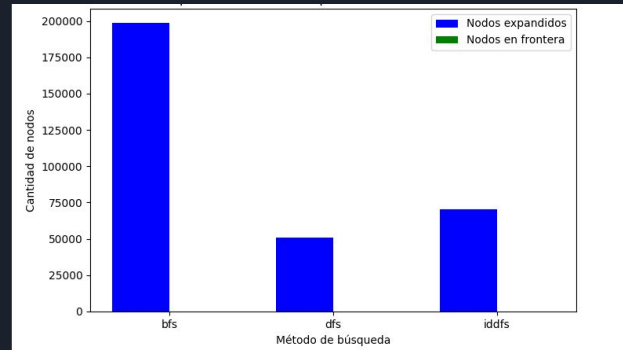
## Nivel 1



## Nivel 2



## Nivel 5



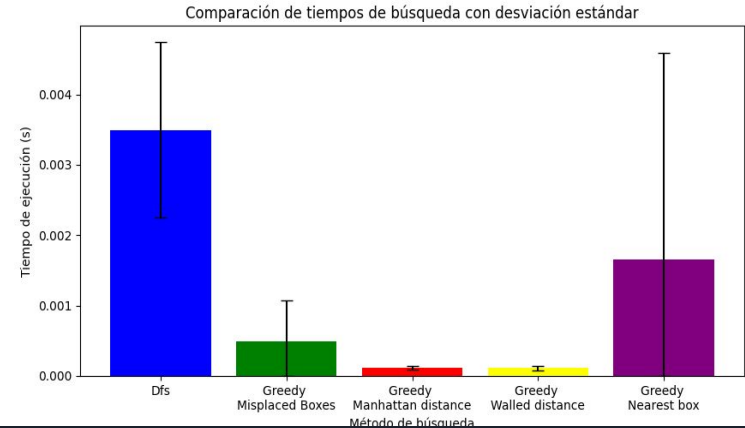
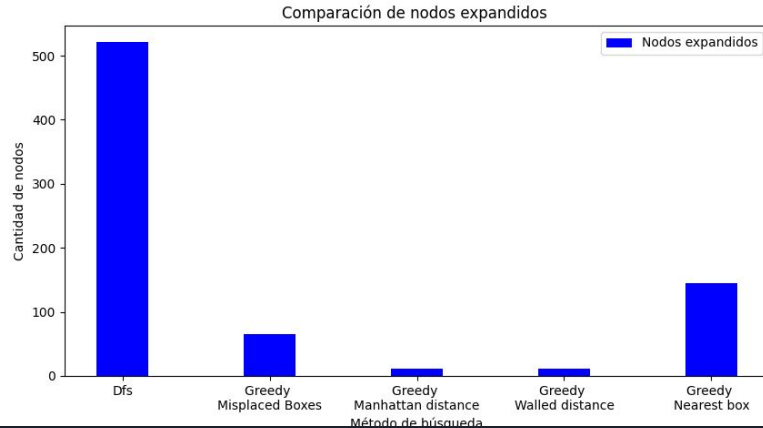




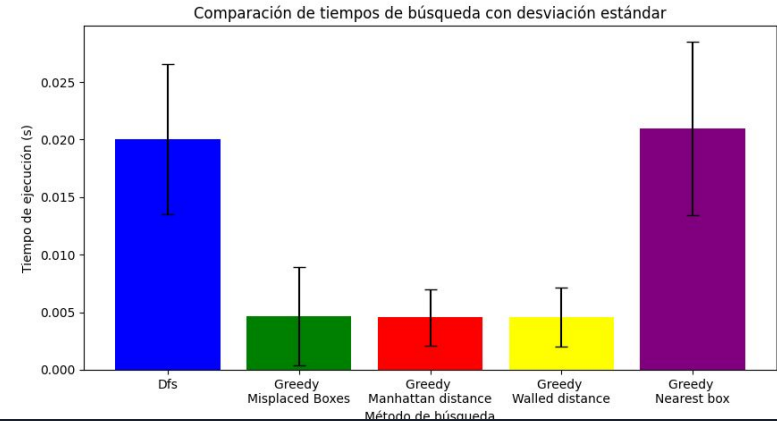
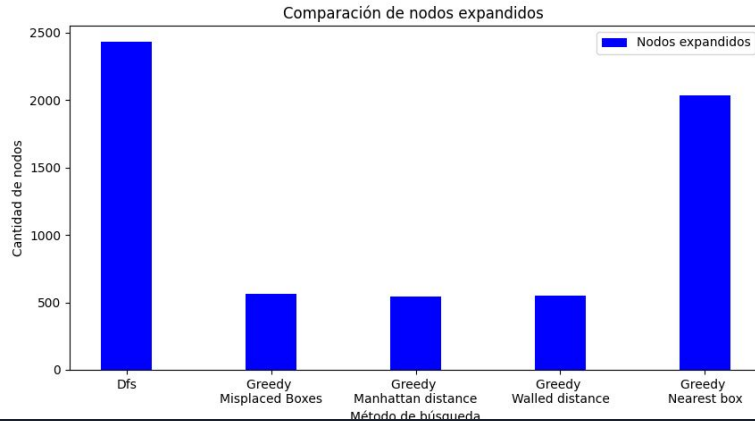
# Métodos informados

- Greedy
- $A^*$

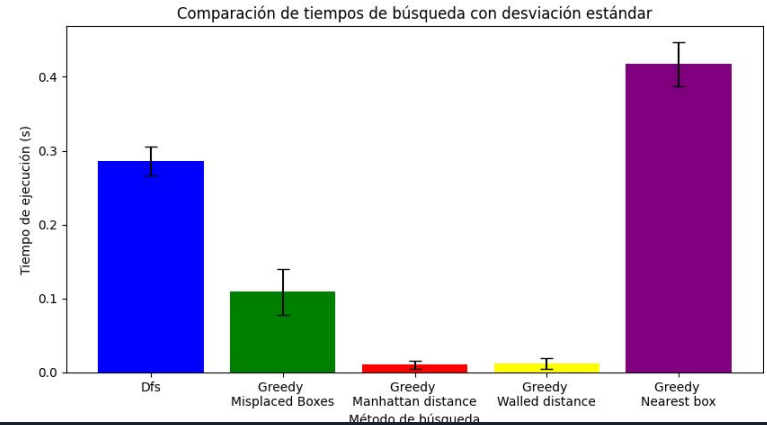
# Greedy - Comparación de heurísticas - Nivel 1



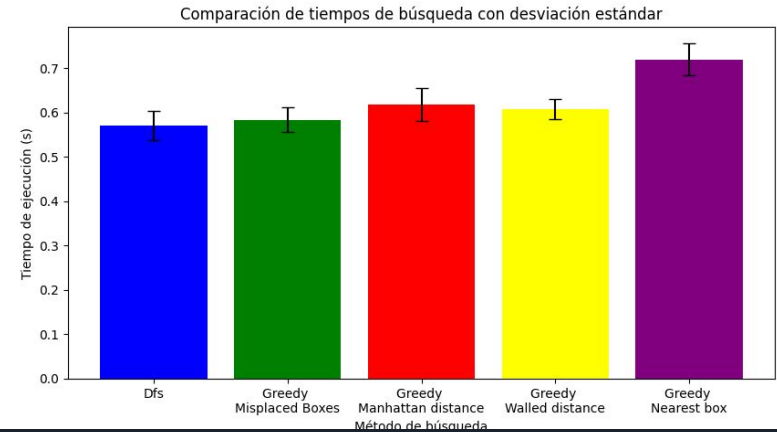
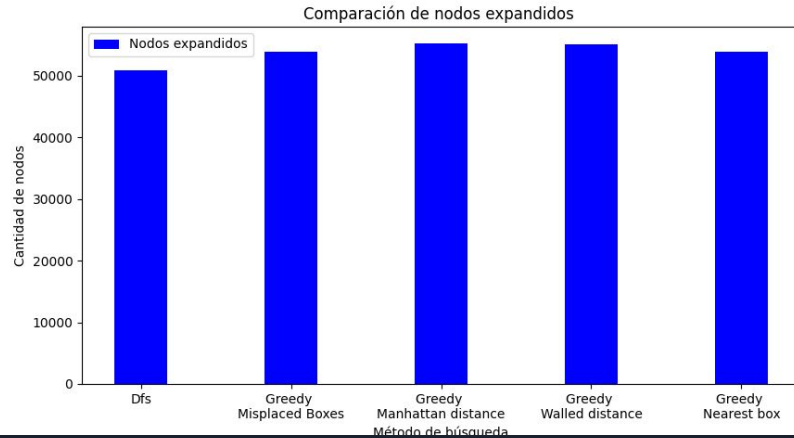
# Greedy - Comparación de heurísticas - Nivel 2



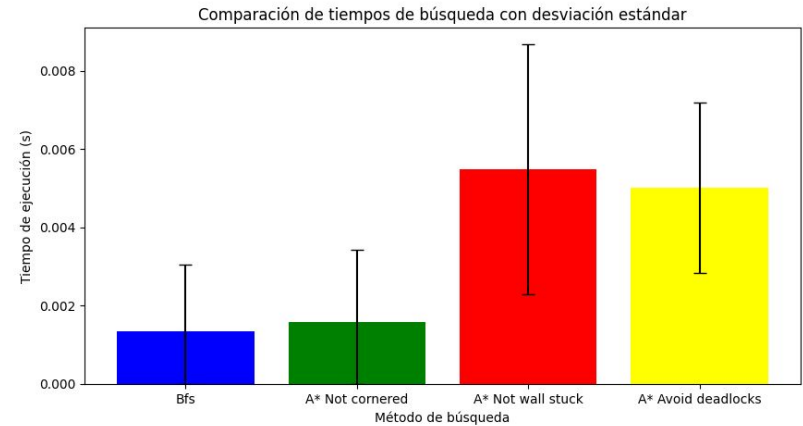
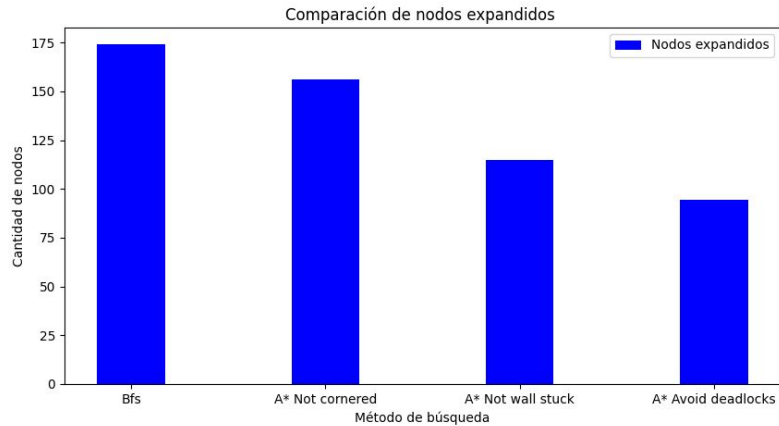
# Greedy - Comparación de heurísticas - Nivel 3



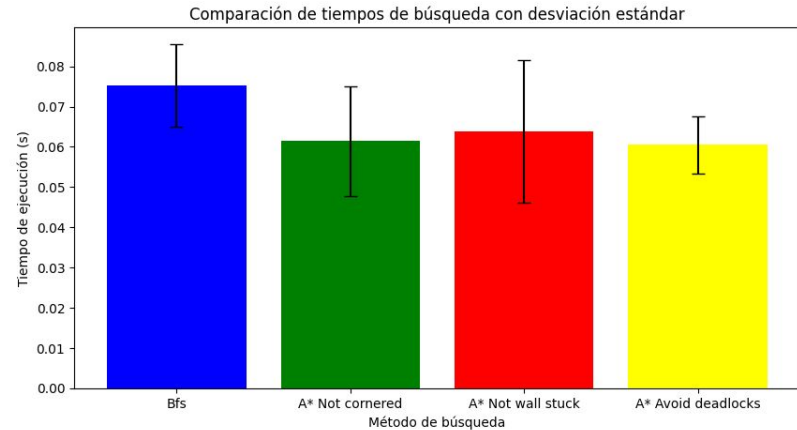
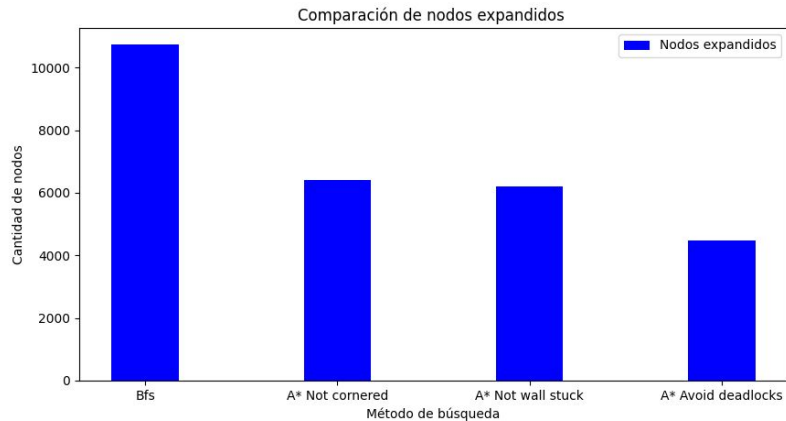
# Greedy - Comparación de heurísticas - Nivel 4



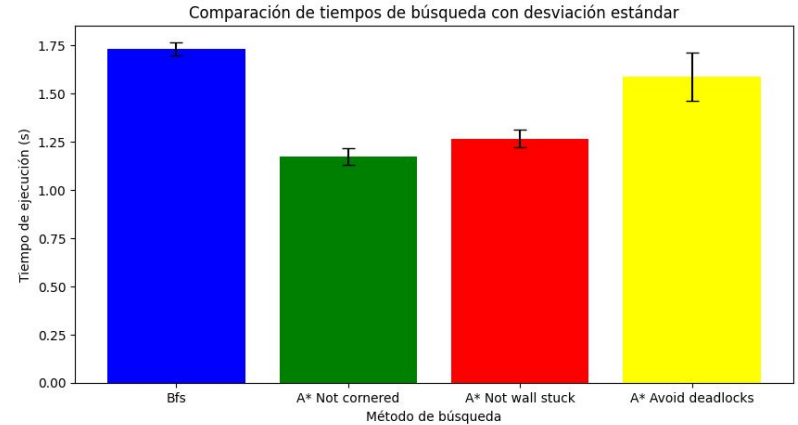
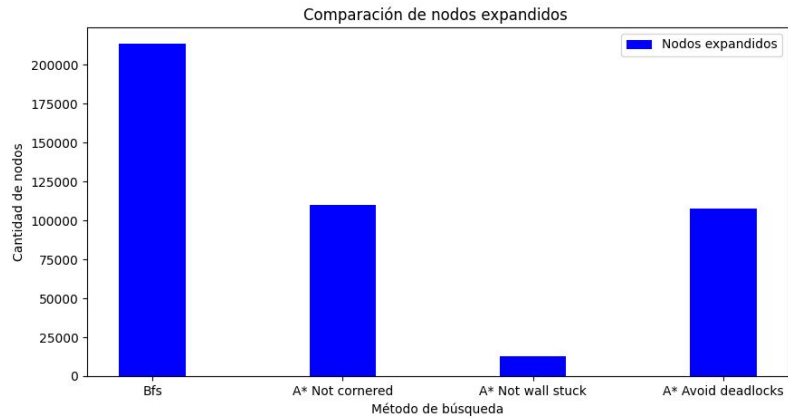
# A\* - Comparación de heurísticas - Nivel 1



# A\* - Comparación de heurísticas - Nivel 2

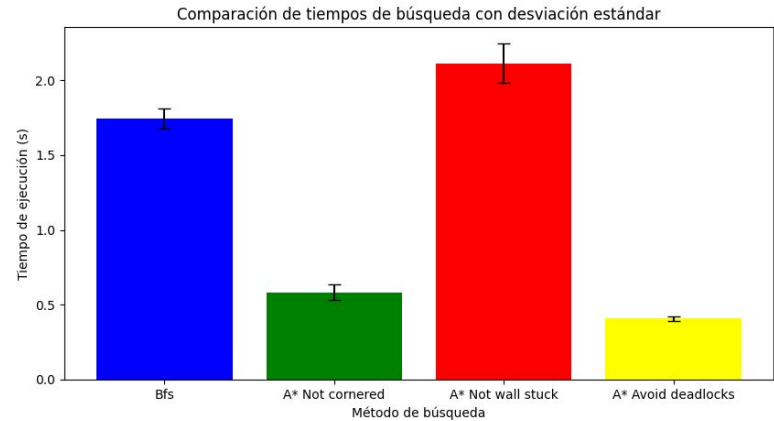
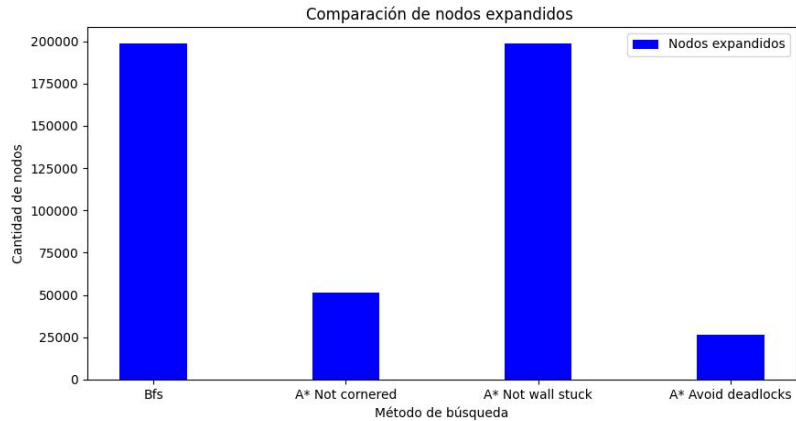


# A\* - Comparación de heurísticas - Nivel 3



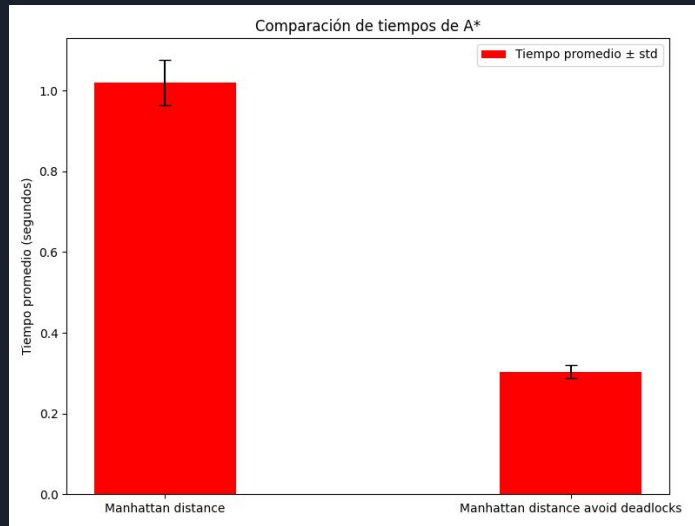


# A\* - Comparación de heurísticas - Nivel 4

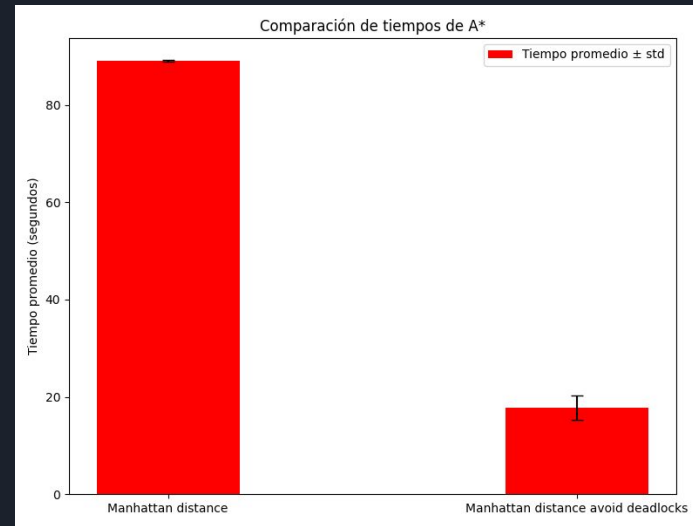


# Manhattan distance avoid deadlocks vs Manhattan distance

Nivel 4

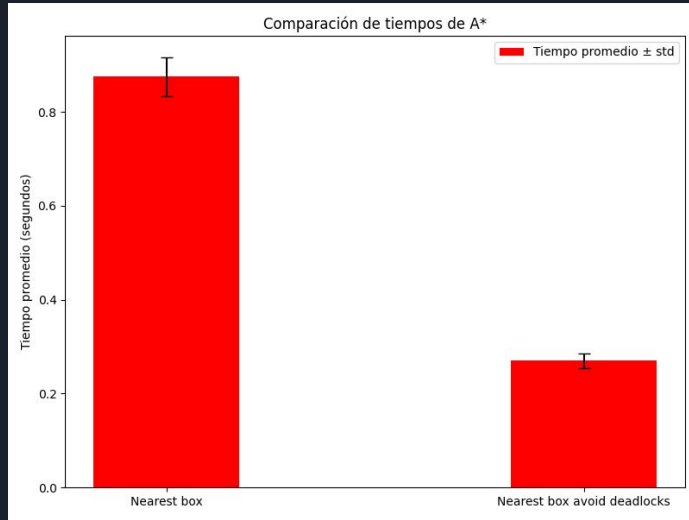


Nivel 5

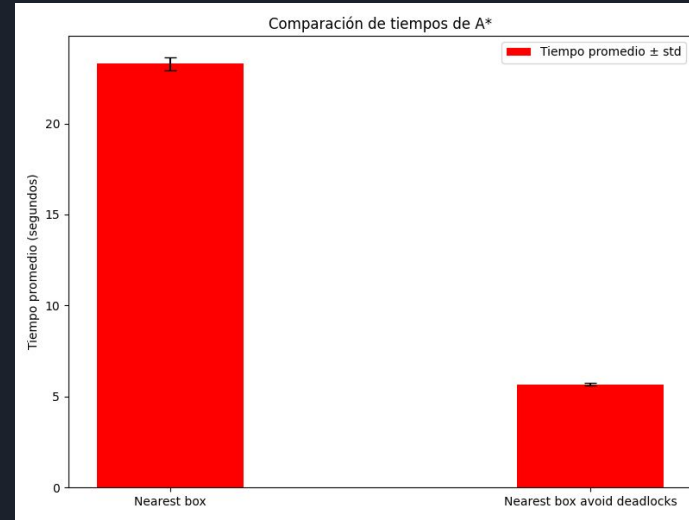


# Nearest box avoid deadlocks vs Nearest box

Nivel 4

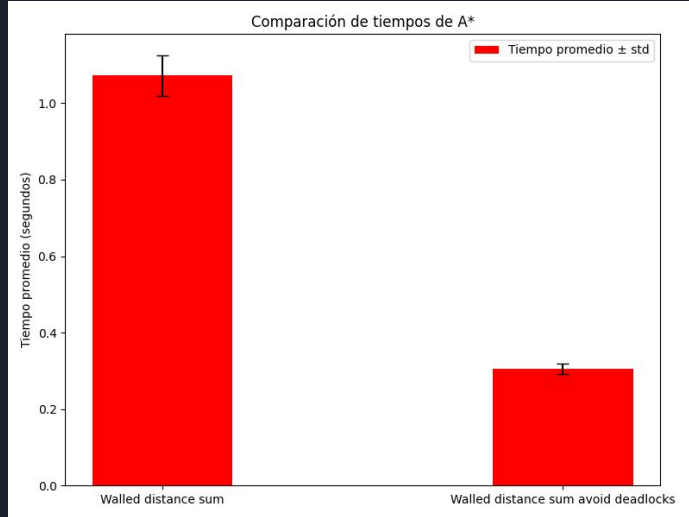


Nivel 5

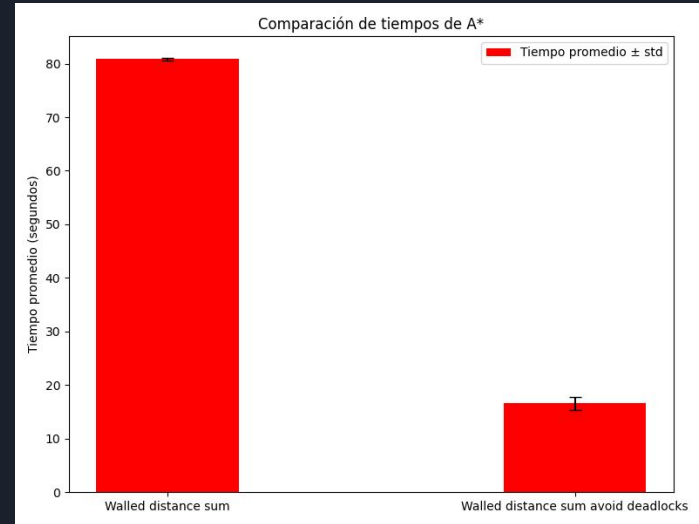


# Walled distance sum avoid deadlocks vs Walled distance sum

Nivel 4



Nivel 5





MUCHAS GRACIAS!!!

