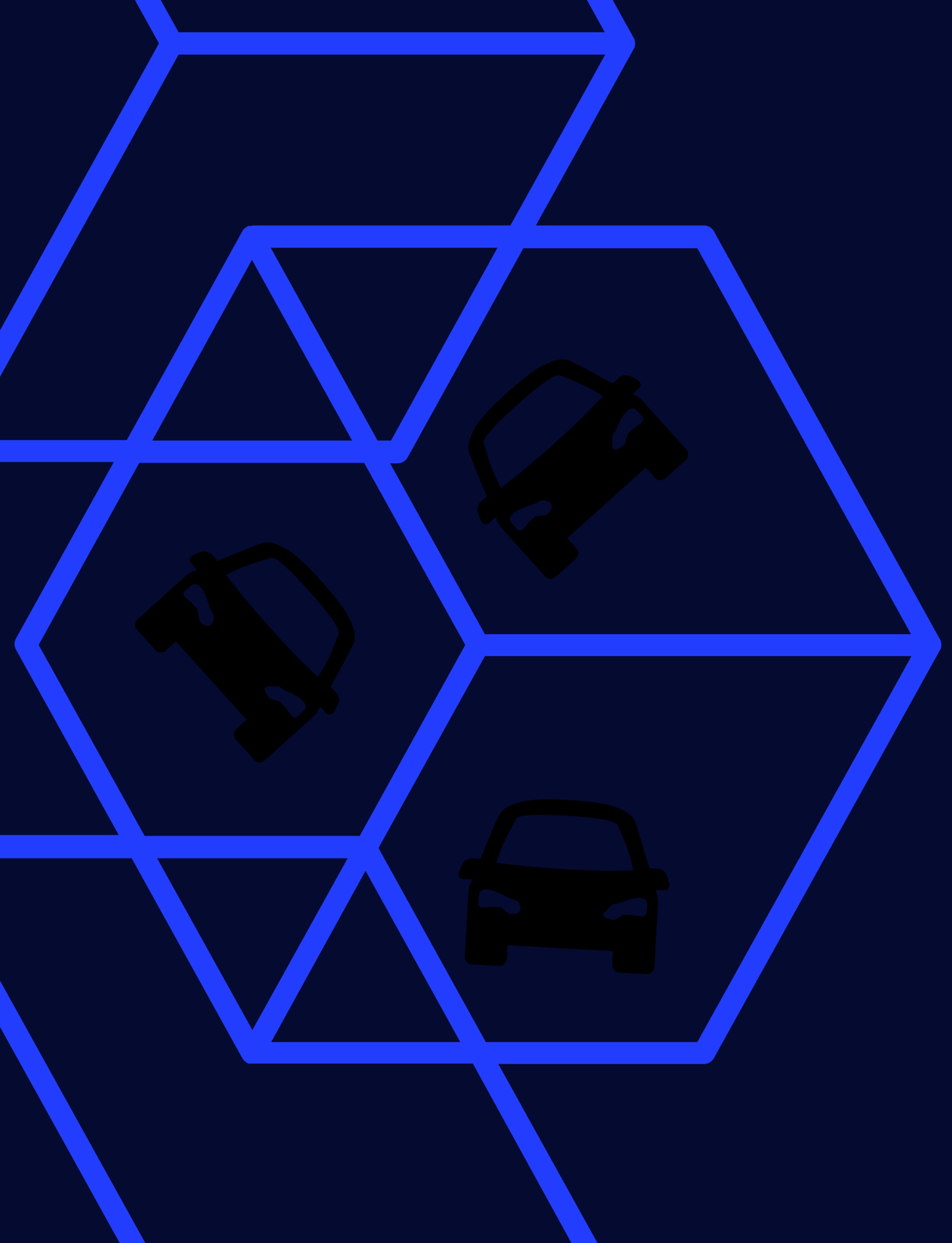
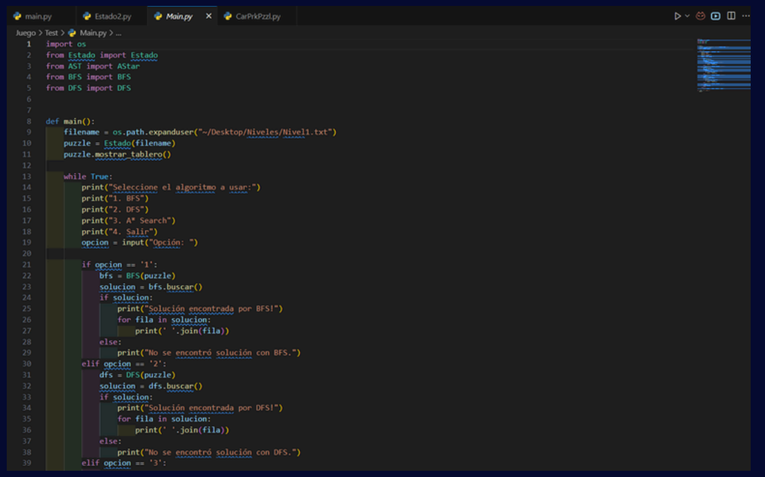


CAR PARK PUZZLE

INTEGRANTES

- Christian Enmanuel Navarro Rojas - **2023-1152**
- Guillermo Rotestan Polonia - **2022-2011**
- Michael Alexander Espinosa Batista - **2023-0927**

INTRODUCCIÓN AL PROYECTO



ALGORITMO BFS

BFS es un algoritmo de búsqueda utilizado para explorar todos los nodos de un grafo (o árbol) comenzando desde un nodo inicial y explorando todos los nodos vecinos en el nivel actual antes de pasar a los nodos del siguiente nivel. Este algoritmo es especialmente útil para encontrar la ruta más corta en grafos no ponderados.

Juego > Test > BFS.py > ...

```python
from queue import Queue

class BFS:
    def __init__(self, puzzle):
        self.puzzle = puzzle
        self.frontera = Queue()
        self.frontera.put(puzzle.tablero)
        self.explorado = set()

    def buscar(self):
        while not self.frontera.empty():
            estado = self.frontera.get()
            if self.puzzle.es_meta(estado):
                return estado
            self.explorado.add(self.estado_a_tupla(estado))
            for sucesor in self.puzzle.generar_vecinos(estado):
                if self.es_meta(sucesor) not in self.explorado:
                    self.frontera.put(sucesor)
        return None

    def estado_a_tupla(self, estado):
        return tuple(tuple(fila) for fila in estado)
```

ALGORITMO DFS

DFS es un algoritmo de búsqueda que comienza en el nodo raíz (o nodo inicial) y explora tan lejos como sea posible a lo largo de cada rama antes de retroceder. Es útil para recorrer todos los nodos de un grafo o para encontrar caminos en problemas específicos.

Juego > Test > DFS.py > DFS

```python
class DFS:
    def __init__(self, puzzle):
        self.puzzle = puzzle
        self.frontera = []
        self.frontera.append(puzzle.tablero)
        self.explorado = set()

    def buscar(self):
        while self.frontera:
            estado = self.frontera.pop()
            if self.puzzle.es_meta(estado):
                return estado
            self.explorado.add(self.estado_a_tupla(estado))
            for sucesor in reversed(self.puzzle.generar_vecinos(estado)):
                if self.es_meta(sucesor) not in self.explorado:
                    self.frontera.append(sucesor)
        return None
```

ALGORITMO AST

es un algoritmo de búsqueda heurística que encuentra el camino más corto desde un nodo inicial hasta un nodo objetivo en un grafo. Utiliza una función de costo combinada para evaluar los nodos, que incluye el costo real desde el nodo inicial hasta el nodo actual y una estimación heurística del costo desde el nodo actual hasta el objetivo.

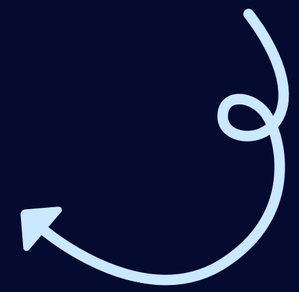
Juego > Test > AST.py > ...

```python
from queue import Queue, PriorityQueue

class AStar:
    def __init__(self, puzzle):
        self.puzzle = puzzle
        self.frontera = PriorityQueue()
        self.frontera.put((0, puzzle.tablero))
        self.explorado = set()

    def heuristica(self, estado):
        # Implementar una heurística simple
        for i, fila in enumerate(estado):
            for j, celda in enumerate(fila):
                if celda == 'A':
                    return abs(self.puzzle.meta[0] - i) + abs(self.puzzle.meta[1] - j)

    def buscar(self):
        while not self.frontera.empty():
            _, estado = self.frontera.get()
            if self.puzzle.es_meta(estado):
                return estado
            self.explorado.add(self.estado_a_tupla(estado))
            for sucesor in self.puzzle.generar_vecinos(estado):
                if self.es_meta(sucesor) not in self.explorado:
                    costo = self.heuristica(sucesor)
                    self.frontera.put((costo, sucesor))
        return None
```

Heurísticas

Las heurísticas son técnicas o métodos que ayudan a resolver problemas de manera eficiente cuando los métodos tradicionales son demasiado lentos o complejos.

Una heurística es una función que estima el costo o la distancia desde un punto en el espacio de búsqueda hasta el objetivo. Estas funciones no garantizan siempre una solución óptima, pero suelen producir soluciones suficientemente buenas en un tiempo razonable.



Conclusión del Proyecto

- **BFS:** Ideal para encontrar la solución más corta en espacios de búsqueda pequeños o moderados, pero su alto consumo de memoria lo hace ineficiente para grandes espacios.

- **DFS:** Bueno para problemas con soluciones cercanas al nodo raíz y para exploraciones profundas, pero no garantiza encontrar la solución más corta y puede ser ineficiente en términos de tiempo.

- **A*:** Generalmente, la mejor opción para el Car Parking Puzzle debido a su balance entre eficiencia y garantía de encontrar la solución óptima, siempre que se utilice una heurística adecuada.

¡GRACIAS!