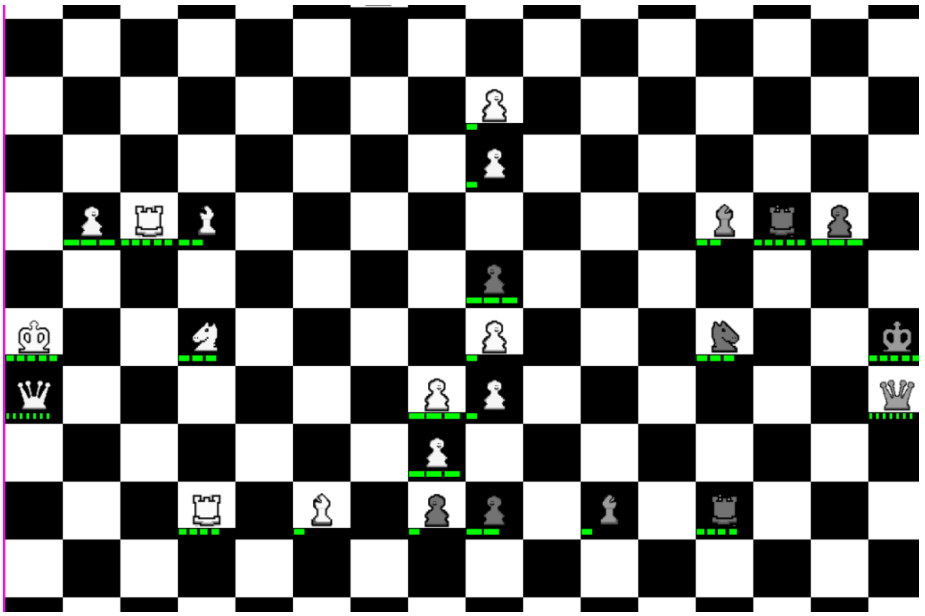
Documentación: ChessWars

1.- Introducción:

El propósito del juego “ChessWars” es poder visualizar entornos complejos de decisión autónomos, utilizando las estructuras de datos aprendidos durante el segundo parcial de la materia “Estructura de Datos y Algoritmos II”, utilizando el entorno gráfico de pygame.



Alcance:

El sistema permite desarrollar dos colonias de ajedrez enemigas, cuyo propósito sería sobreponerse a la otra, y rigiéndose bajo las reglas de ajedrez, tipos de movimientos, capturas, etc.

Audiencia Prevista:

La audiencia prevista es a todo el interesado acerca de sistemas autónomos o que esté llevando la materia en curso

Definiciones / acrónimos:

- Pygame: Librería para entornos gráficos de python
- Nodo: Estructura para almacenar información
- Queue / Heap: estructuras para manejar datos
- Autómata: no es necesario estar presentes en el juego para que este se desarrolle

2.- Visión general:

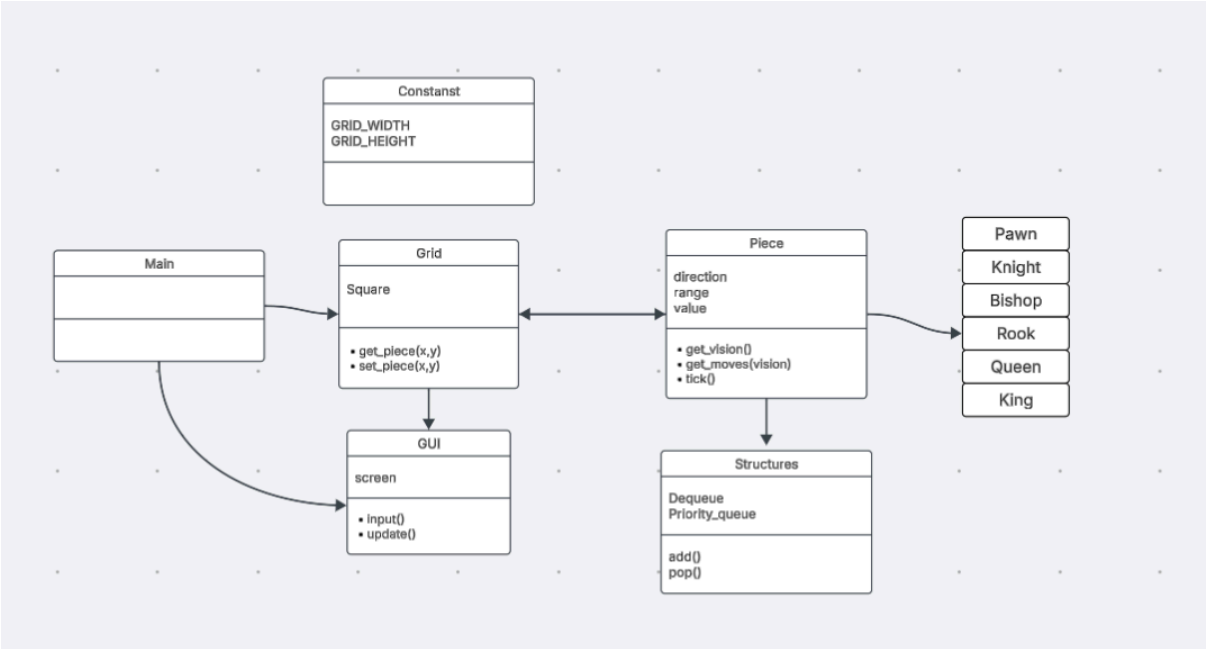
- Módulos del sistema, proposito, entradas y salidas:

Módulo	Propósito	Entradas	Salidas
Pieces	Definir piezas, y características generales que todas deben tener, poder manejar todas las piezas como un conjunto y no de forma individual	Posición, tipo de pieza	Decisión de la pieza a donde moverse
Grid	Trabajar el tablero y ubicación de las piezas, intermediario entre pieces y pygame para el renderizado y buena interacción entre ellas	Pieza nueva o con posición a actualizar	Ubicación actual de la pieza, borrar pieza
GUI	Interfaz con el que el usuario podrá observar las decisiones de las piezas y además podrá interferir sobre qué tendencias tendrán las mismas	Estado actual del tablero	Entradas del jugador, acciones o dinámicas
Main	Programa principal para llamar al GUI y al grid		Llamadas API
Constants	Almacenar valores fijos del sistema que varios módulos ocuparan	N/A	Valores accesibles
Structures	Estructuras de datos para manejar la información del sistema, queues y heaps	Datos	El mayor elemento, el primero en cola etc.

- Dependencias y funciones principales:

Módulo	Dependencias	Funciones principales
Pieces	constants	• get_vision() • get_moves(vision) • tick()
Grid	Piece	• get_piece(x,y) • set_piece(x,y)
GUI	pygame, Grid	• input() • update()

- Diagrama de componentes y flujo de datos:



El flujo de datos mantiene la estructura de:

- Input GUI
- Actualización grid
- Obtención de información de Piece
- Pice llama y actualiza heaps
- Respuesta de Piece
- Cambio en Grid
- Salida en GUI

3.- Clases Principales:

- Piece
    - Atributos:
        - Queue de equipo
        - posición
        - Estado
        - Nivel
        - Equipo
    - Métodos:
        - valid (position)
            - regresa True si la coordenada indexada esta en el mapa
        - delete (self)

                - Elimina a la pieza de la deque de su equipo
            - get_vision(self)
                - Regresa las casillas a las cuales la pieza tiene acceso a su información, (movimientos + adyacentes)
            - get_moves(vision_other_pieze)
                - Dado una pieza y un campo de visión a cuales movimientos puede moverse la pieza, es decir retornara movimientos que incluso sean imposibles pero que bajo la perspectiva de la visión si lo sean (La vision no vea si algo interviene)
            - tick()
                - Genera una decicion de la pieza a la cual se le llama la función
    - Grid
        - Atributos:
            - Square
            - Screen
        - Métodos:
            - get_piece(x, y)
                - Regresa None si no hay ninguna pieza sobre esa parte del tablero y la pieza si la hay
            - Set_piece(x, y)
                - Coloca una pieza si es posible en la coordenada la cual se le ingresa
            - Clear(x,y)
                - Remueve la pieza en una casilla especifica
            - Update()
                - Actualiza constantemente el juego para simular escenarios
                -

-Herencias y polimorfismos:

- Piece:
    - Pawn
    - Knight
    - Bishop
    - Rook
    - Queen
    - King

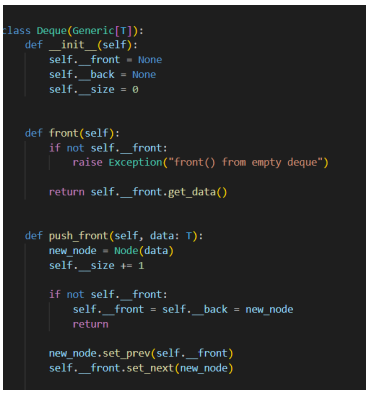
Todas las anteriores piezas heredan de Piece, y usan sus métodos como get_vision(), get_moves(), etc. pero cada una tiene datos que la hacen distinta a los demás como:

- Tipo de movimiento
- Preferencias de pieza
- Diferentes aumentos de nivel
- Rango de visibilidad
- Propiedades únicas como coronar o comer en diagonal del peon.

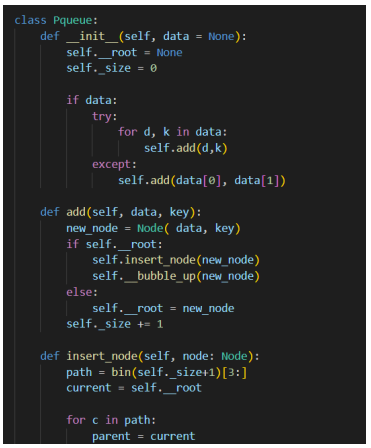
Interfaz de cada módulo:

Módulo	Métodos	Descripción	Retorno
Pieces	Tick()	Función que indica que se debe generar un movimiento, piece procesa la información y mueve la pieza según el algoritmo de decisión	Void
Grid	get_piece(x,y) set_piece(self, x,y) clear(x,y)	Funciones para trabajar con las piezas en grid, preguntar que hay, colocar, borrar etc.	-Piece - Void - Void
GUI	input() outpu()	Obtiene la información dada por el usuari, orden ejecutada u posición del mouse Muestra los datos de la pieza principal (king)	-lista de valores -void

4.- Estructura de datos y algoritmos:

Estructura	Función en el programa	Operaciones	Complejidades
Dequeue	Mantener el orden de cola de los movimientos: 	front() push_front() remove()	Operaciones como append y pop_back() tienen una complejidad de O(1) gracias a su estructura de lista enlazada y el método remove tiene una complejidad de O(n)

```
class Deque(Generic[T]):
    def __init__(self):
        self.__front = None
        self.__back = None
        self.__size = 0


    def front(self):
        if not self.__front:
            raise Exception("front() from empty deque")

        return self.__front.get_data()


    def push_front(self, data: T):
        new_node = Node(data)
        self.__size += 1

        if not self.__front:
            self.__front = self.__back = new_node
            return

        new_node.set_prev(self.__front)
        self.__front.set_next(new_node)
```

Heap	Ordena y cambia valores a celdas para encontrar la mejor en el algoritmo de decisión. 	add() change_priority() top()	Todas estas operaciones gracias a la estructura de arbol binario se consiguen en una complejidad de O(log n)

```
class Pqueue:
    def __init__(self, data = None):
        self.__root = None
        self._size = 0

        if data:
            try:
                for d, k in data:
                    self.add(d,k)
            except:
                self.add(data[0], data[1])

    def add(self, data, key):
        new_node = Node( data, key)
        if self.__root:
            self.insert_node(new_node)
            self.__bubble_up(new_node)
        else:
            self.__root = new_node
        self._size += 1

    def insert_node(self, node: Node):
        path = bin(self._size+1)[3:]
        current = self.__root

        for c in path:
            parent = current
```

Algoritmo de decisión:
El algoritmo de decisión de cada pieza se basa en los siguientes parámetros:

- Visión de la pieza
- Movimientos de la pieza
- Alejamiento del origen
- Movimientos enemigos en mi visión
- Movimientos aliados en mi visión
- Valor de las piezas
- Iniciativa
- Descanso de estamina

Más constantes que dependen de cada pieza:

- value
- support
- attack
- recomended_x
- initiative
- restore

Primero en la heap se colocan todos los posibles movimientos de la pieza, y con un ligero peso en tendencia a acercarse a su posición x recomendada (que depende de la pieza y su estado)

```
cells = Pqueue()

for x,y in moves:
        # Weight of position
        cells.add((x, y),0)

        if self.team == 0 and pos_x <= self.recomended_x:
            cells.change_priority((x,y), 10 * abs((x - pos_x)) + abs(y -pos_y))
        elif self.team == 1 and pos_x >= self.recomended_x:
            cells.change_priority((x,y), 10 * abs((x - pos_x)) + abs(y -pos_y))
        else:
            cells.change_priority((x,y), 5 * abs((x - pos_x)) + abs(y -pos_y))
```

Despues se recorre la vision y de las piezas encontradas se calcula cuales atacan y cuales defienden (enemigas o amigas)

```
for x,y in vision:
    piece = grid.get_piece((x ,y))
    if piece:
        for u,v in piece.get_moves(vision,0):
            if (u,v) in moves:
                # Weight of incoming attacks or support of other pieces
                if piece.team != self.team:
                    cells.change_priority((u,v), self.attacked)
                else:
                    cells.change_priority((u,v), self.support)
```

Después de ello se calculan cuales piezas son comidas en el movimiento y esa casilla gana valor, y también cuales se pueden atacar después de dos movimientos (iniciativa):

```
for x,y in moves:

    piece = grid.get_piece((x,y))

    #Direct attack pieces
    if piece:
        cells.change_priority((x,y), piece.value + 5)

    self.position = pygame.Vector2(x,y)
    moves2 = self.get_moves(vision, attack = False)

    for u,v in moves2:
        if (u,v) == (x,y):
            continue
        if (u,v):
            piece = grid.get_piece((u, v))
            if piece:
                cells.change_priority((x ,y), self.initiative)
```

Finalmente se agrega un extra a la casilla actual (quedarse quieto) si es necesario recuperar estamina:

```
if self.stamina <= self.max_stamina// 2 :
    cells.change_priority((pos_x ,pos_y), self.restore)
```

Una vez calculado y sumado todos estos valores en la heap, el máximo (el top de la heap) será nuestra elección.

5.- Integración GUI - logica

- GUI: muestra al usuario las posiciónes de las piezas en el tablero y botones de comando
- Información obtenida por el GUI es enviada a main y de ahi distribuida a clases como:
- Grid, recibe parametros del mouse para saber su ubicación, si presiona un boton, genera un cambio global que es afectado a Piece
- Piece genera las respuestas a dichos cambios y se los informa a Grid
- Grid retorna la información obtenida al GUI para que despliegue el resultado.

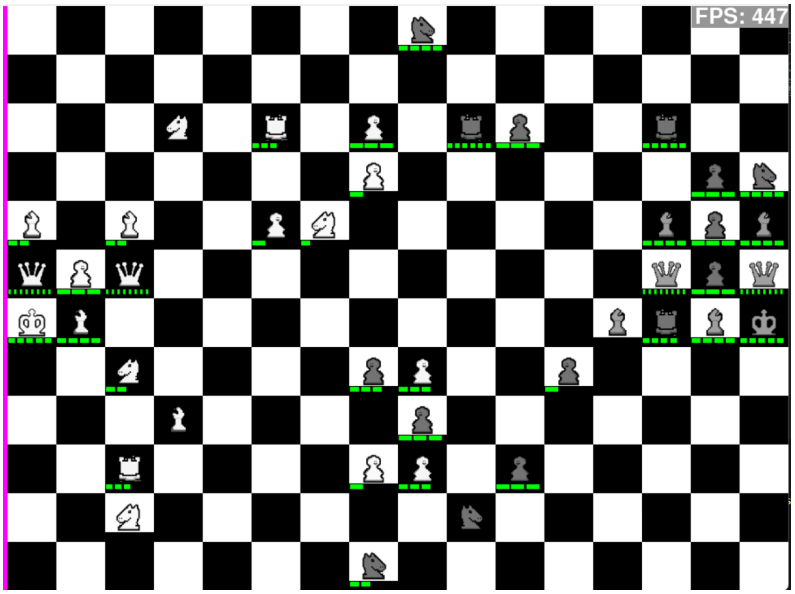
6.- Requisitos funcionales:
Se necesita de un environnement con la extensión de python de pygame, además de eso, ram, o memoria no es exigente.

7.- Pruebas y métricas

Las estructuras de datos, fueron probadas a casos limite y siguieron funcionando

```
def main():
    test = Deque()
    test.push_back(5)
    test.pop_back()
    test.push_front(5)
    test.remove(5)


    test.push_back(10)
    test.pop_back()
    test.push_front(10)
    test.remove(10)

if __name__ == "__main__":
    main()
```

El juego se cargó a 50 piezas y siguio marcando una buena taza de fps.



Los movimientos y visión de cada pieza fueron obtenidos y deseados como se quería:

```
def main():
    from bishop import Bishop
    prueba = Bishop(0,0,1)
    print(prueba.get_moves())

if __name__ == "main":
    main()
```

Fps: estables

8.- Decisiones de diseño y riesgos

Decidir hacer un ajedrez y no un mundo abierto:

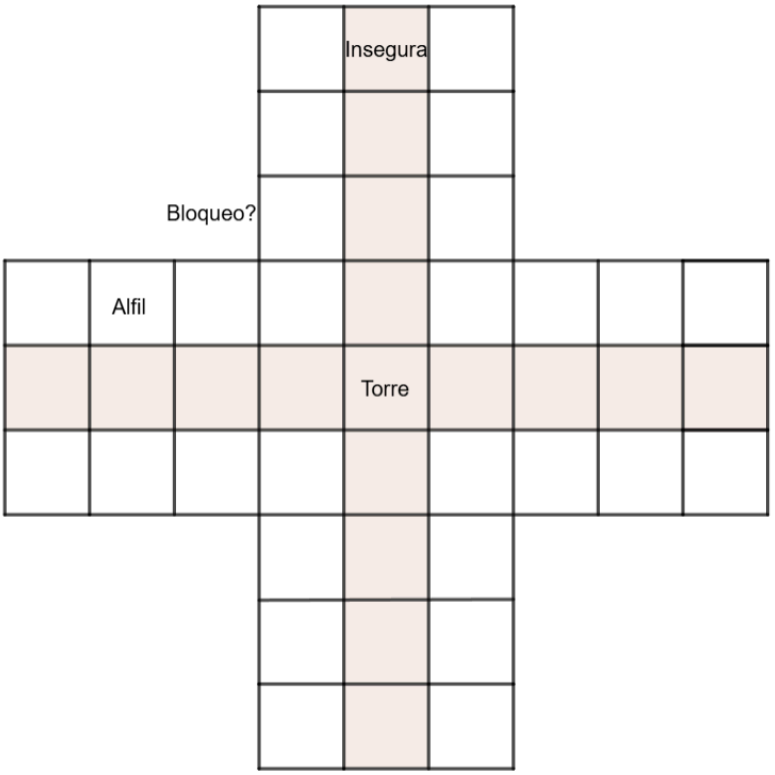
Esa decisión consideramos un riesgo y ganancia pues creemos que es algo bastante creativo he entretenido pero el mundo tan pequeño hace parecer que las piezas no gozaban de libertad, o que fueran iguales. Aquí comenzó uno de los mayores riesgos de como hacer las piezas únicas, distintas y a su vez como hacer que cumplan en los demás aspectos de recursos, etc.

El mundo limitado puede parecer que restringe mucho a los personajes, pero vimos que lo que hace importante al personaje no es la exploración si no la relación con otras piezas, de ahí decidimos explotar esto.

Eliminación de animaciones, todo lo demás de implementar ejecutar representó una gran carga para desarrollar este proyecto que decidimos dejar atrás las animaciones.

9.- Apéndices

-Evolución futura: Agregar más piezas, objetos, eventos, etc.
-Diagrama de visualización de la toma de decisiones.



Aunque es posible que el alfin este bloqueado y ataque a la casilla, la torre no lo sabe y la considera como insegura.