Informe programación 3 proyecto Kakuro por Emmanuel Galvis Morales

```python
Dom=set(range(1,10))
# print(Dom)

Idcols = "ABCDEFGHI"

import itertools as it

Varkeys = list(it.product(Dom,Idcols))
# print(Varkeys)

# convertir lista de tuplas a strings

strVarkeys= [f"{key[1]}{key[0]}" for key in Varkeys]

# print(strVarkeys)

VarDoms={key:Dom.copy() for key in strVarkeys}

# print(VarDoms)



# tableros disponibles, se debe comentar todos menos el que se desea utilizar


# tablero="ProgIIIG1-Act08-KK5GGRJS-Board.txt"
# tablero="ProgIIIG1-Act08-KK5DDPQF-Board.txt"
tablero="ProgIIIG1-Act08-KK5VMPMA-Board.txt"


with open(tablero,"r") as archivo:
    for clave in VarDoms:
        linea = archivo.readline().strip() #0-4 & - &
        if linea=="-":
            VarDoms[clave] = "Negro"
        elif linea.isalnum():
            VarDoms[clave] = linea
```

Inicialización de los dominios y del tablero, establece 3 diferentes tableros de dificultad muy difícil, con archivos locales .txt, de 81 líneas equivalentes a las casillas del kakuro, estos tienen 81 líneas con diferentes dominios asignados en Sudokumania, se hace la distinción entre las celdas Negras que no tienes valores para asignar, y las celdas de pistas que tienen dos valores separados por y de la forma 0y30, donde el primer valor es la sumatoria de fila, y el segundo valor es la sumatoria de columnas.

```python
def defColsConstraints(VarDoms, IdCols):
    constraints = []
    for col in IdCols:
        bloque = []
        suma = None
        for row in range(1, 10):
            clave = f"{col}{row}"
            valor = VarDoms[clave]
            if isinstance(valor, set):  # blanca
                bloque.append(clave)
            else:  # negra o sumador
                if bloque:
                    # Revisar celda anterior para buscar la suma
                    fila_suma = row - len(bloque) - 1
                    if fila_suma >= 1:
                        clave_suma = f"{col}{fila_suma}"
                        valor_suma = VarDoms.get(clave_suma, "")
                        if isinstance(valor_suma, str) and "y" in valor_suma:
                            try:
                                _, suma_str = valor_suma.split("y")
                                suma = int(suma_str)
                                if suma > 0:
                                    constraints.append([bloque.copy(), suma])
                            except:
                                pass  # formato inválido, se ignora
                    bloque = []
        if bloque:
            fila_suma = 9 - len(bloque)
            if fila_suma >= 1:
                clave_suma = f"{col}{fila_suma}"
                valor_suma = VarDoms.get(clave_suma, "")
                if isinstance(valor_suma, str) and "y" in valor_suma:
                    try:
                        _, suma_str = valor_suma.split("y")
                        suma = int(suma_str)
                        if suma > 0:
                            constraints.append([bloque.copy(), suma])
                    except:
                        pass
            bloque = []
    return constraints
```

Definición de la lista de listas de las restricciones por columnas y filas, permite recorrer la lista de variables por restricciones, y definir las consistencias, la lista Constraints debe tener múltiples posibles listas con las variables que hacen referencia a una columna o fila, se tiene en cuenta que una columna o fila pueden tener más de una lista de restricciones con sus respectivas sumatorias de la forma, [[[A1,A2,A3],30],[[B3,B4],10],[[],],...

```python
def defRowsConstraints(VarDoms, IdCols):
    constraints = []
    for row in range(1, 10):
        bloque = []
        suma = None
        for col_index, col in enumerate(IdCols):
            clave = f"{col}{row}"
            valor = VarDoms[clave]
            if isinstance(valor, set):  # celda blanca
                bloque.append(clave)
            else:  # celda negra o sumador
                if bloque:
                    # Revisar celda anterior (a la izquierda) para la suma
                    col_suma_idx = col_index - len(bloque) - 1
                    if col_suma_idx >= 0:
                        col_suma = IdCols[col_suma_idx]
                        clave_suma = f"{col_suma}{row}"
                        valor_suma = VarDoms.get(clave_suma, "")
                        if isinstance(valor_suma, str) and "y" in valor_suma:
                            try:
                                suma_str, _ = valor_suma.split("y")
                                suma = int(suma_str)
                                if suma > 0:
                                    constraints.append([bloque.copy(), suma])
                            except:
                                pass  # formato inválido
                    bloque = []
        if bloque:
            col_suma_idx = 8 - len(bloque)
            if col_suma_idx >= 0:
                col_suma = IdCols[col_suma_idx]
                clave_suma = f"{col_suma}{row}"
                valor_suma = VarDoms.get(clave_suma, "")
                if isinstance(valor_suma, str) and "y" in valor_suma:
                    try:
                        suma_str, _ = valor_suma.split("y")
                        suma = int(suma_str)
                        if suma > 0:
                            constraints.append([bloque.copy(), suma])
                    except:
                        pass
            bloque = []
    return constraints

Constraints = defColsConstraints(VarDoms, Idcols) + defRowsConstraints(VarDoms, Idcols)
# print(Constraints)
```

Se aplica la función anterior para filas también y luego se juntan en la lista Constraints.

```python
def ConsistenceKakuro(Constraints, VarDoms):
    anyChange = False
    for constraint in Constraints:
        vars_bloque, suma = constraint
        asignadas = {v: next(iter(VarDoms[v])) for v in vars_bloque if len(VarDoms[v]) == 1}
        no_asignadas = [v for v in vars_bloque if len(VarDoms[v]) > 1]

        # 1. Eliminar valores asignados en el mismo bloque (como Sudoku)
        for v_asig, val in asignadas.items():
            for v in no_asignadas:
                oldDom = VarDoms[v].copy()
                VarDoms[v].discard(val)
                if VarDoms[v] != oldDom:
                    anyChange = True

        # 2. Generar combinaciones válidas para las no asignadas
        doms = [VarDoms[v] for v in no_asignadas]
        posibles_combinaciones = []
        for comb in it.product(*doms):
            if len(set(comb)) == len(comb):  # sin repeticiones
                total = sum(comb) + sum(asignadas.values())
                if total == suma:
                    posibles_combinaciones.append(comb)

        # 3. Recortar dominios según combinaciones válidas
        for i, v in enumerate(no_asignadas):
            posibles_valores = set(comb[i] for comb in posibles_combinaciones)
            oldDom = VarDoms[v].copy()
            VarDoms[v].intersection_update(posibles_valores)
            if VarDoms[v] != oldDom:
                anyChange = True

    return anyChange
```

función que representa las tres consistencias básicas del kakuro, la consistencia básica para no repetir valores en columnas y filas; la consistencia que permite buscar dominios iguales entre dos o más recuadros del kakuro que elimina esos dominios de los demás recuadros ya que no hay posibilidad de que esos valores sean asignados en otro punto y consistencia que busca dominios que contengan valores no repetidos en las columnas o filas; la implementación de estas tres consistencias permite resolver tableros de dificultad muy difícil de sudokumania, se implementa un algoritmo de búsqueda básico, para mostrar la diferencia en tiempo de ejecución, utilizando la función de consistencia, y la que no.

```python
def mostrar_tablero(VarDoms):
    for i in range(1, 10):
        fila = ""
        for c in Idcols:
            val = VarDoms[f"{c}{i}"]
            fila += str(next(iter(val))) if len(val) == 1 else "."
            fila += " "
            if c in "CF": fila += "| "
        print(fila)
        if i in [3, 6]: print("-" * 21)
```

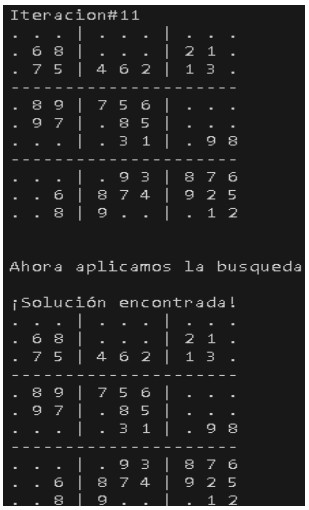
```python
def es_valido(var, valor, asignacion, Constraints):
    # Crear una copia temporal de la asignación
    temp_asignacion = asignacion.copy()
    temp_asignacion[var] = valor

    for constraint in Constraints:
        vars_bloque, suma = constraint
        valores = []
        for v in vars_bloque:
            if v in temp_asignacion:
                valores.append(temp_asignacion[v])
            else:
                valores.append(None)

        asignados = [v for v in valores if v is not None]

        # Verificar duplicados
        if len(asignados) != len(set(asignados)):
            return False

        # Verificar suma si todas están asignadas
        if None not in valores:
            if sum(asignados) != suma:
                return False

        # Verificar que no sobrepasemos la suma parcial
        if sum(asignados) > suma:
            return False

    return True

def backtracking(VarDoms, Constraints):
    def bt(asignacion):
        if len(asignacion) == len([v for v in VarDoms if isinstance(VarDoms[v], set)]):
            return asignacion  # ¡Todas las variables asignadas!

        # Seleccionamos la siguiente variable no asignada
        for var in VarDoms:
            if isinstance(VarDoms[var], set) and var not in asignacion:
                for valor in VarDoms[var]:
                    if es_valido(var, valor, asignacion, Constraints):
                        asignacion[var] = valor
                        result = bt(asignacion)
                        if result:
                            return result
                        del asignacion[var]  # backtrack
                return None  # No hay valor válido para esta variable
        return None  # No hay más variables
```

Algoritmo de búsqueda básico de back tracking cronológico, visto en clase, utilizamos una función de validación es_valido(), que permite verificar si un valor asignado en una celda cumple con las restricciones básicas del kakuro, verificar duplicados, si la suma de los datos asignados den el valor exacto del sumador, y que los valores no sobrepasen el valor del sumador, en la función de backtracking(), iteramos recursivamente hasta encontrar la solución valida, sin las ventajas de búsqueda en velocidad de otros algoritmos, esto debido a mostrar la diferencia en velocidad de operación cuando se utiliza consistencia en este programa específico.

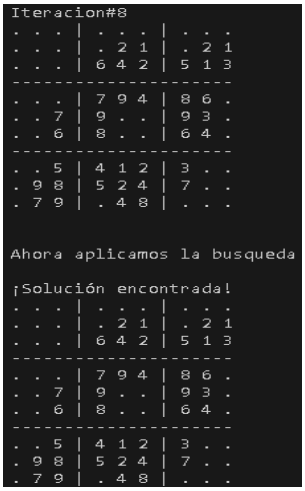
```python
anyChange = True
iteration = 1
while anyChange:
    print(f"Iteracion#{iteration}")
    iteration += 1
    anyChange = False
    mostrar_tablero(VarDoms)
    print("\n")
    for constraint in Constraints:
        if ConsistenceKakuro([constraint], VarDoms):
            anyChange = True

print("Ahora aplicamos la busqueda\n")
solution = backtracking(VarDoms, Constraints)
if solution:
    for var in VarDoms:
        if isinstance(VarDoms[var], set) and var in solution:
            VarDoms[var] = {solution[var]}
    print("¡Solución encontrada!")
else:
    print("No se encontró solución.")

mostrar_tablero(VarDoms)
```

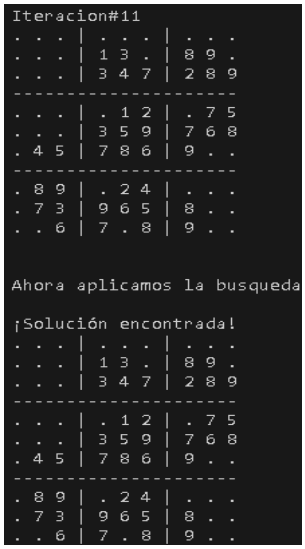
Ciclo de iteración para llamar a las diferentes funciones de consistencia hasta encontrar la solución o el CSP reducido, y posteriormente llamar la función de búsqueda si esta es requerida.

Resultados:



Podemos apreciar que utilizando tres tableros kakuro de dificultad muy dificil de sudokumania, el programa itera entre 8 a 11 iteraciones para encontrar la solución del tablero, en este caso la búsqueda no es necesaria, en el caso de que el kakuro tuviera mas de 81 celdas el espacio de posibilidades haría necesario utilizar búsqueda obligatoriamente, en la ejecución de video podemos apreciar una diferencia en tiempo sustancial que muestra lo eficiente de aplicar consistencias en este tipo de problemas.





Conclusiones:

La solución de problemas utilizando CSP, es una estrategia muy útil para minimizar al máximo el espacio de búsqueda para problemas cuyas posibles combinaciones pueden generar un espacio de búsqueda demasiado grande para ser computado por fuerza bruta, a su vez hay diferentes implementaciones de algoritmos de búsqueda que mejoran la eficiencia de este proceso en caso de que la minimización por consistencias no nos permita hallar la solución, en ejercicios como el kakuro podemos apreciar de manera intuitiva y visual como cada uno de los diferentes pasos en el código reduce sustancialmente la cantidad de casillas y dominios a resolver, lo que para un humano puede tardar horas, con las correctas estrategias una computadora puede hallar soluciones en milésimas de segundo con las restricciones, y mejoras adecuadas.