Introducción

Kakuro es un juego de lógica que combina aspectos de los crucigramas con operaciones aritméticas. El objetivo es llenar celdas blancas con números del 1 al 9, de manera que sumen el valor objetivo establecido por las pistas dadas y sin repetir valores en cada grupo. En este informe se presenta un enfoque basado en **Programación por Restricciones (CSP)** para resolver tableros de Kakuro de dificultad "Muy difícil".

La propuesta se implementa en **Python**, empleando técnicas como inferencia por consistencia, consistencia de arco (**AC-3**) y búsqueda con retroceso (backtracking). Se parte de una lectura de tablero desde un archivo plano, procesando la información en estructuras que permiten identificar grupos y restricciones del juego.

Planteamiento del problema

Cada celda blanca es modelada como una **variable**, con un **dominio** inicial de valores enteros del 1 al 9. Las celdas negras contienen las **restricciones de suma** que afectan grupos de celdas adyacentes horizontal o verticalmente. Estas restricciones también implican que los valores dentro de un grupo no pueden repetirse.

El objetivo es encontrar una asignación consistente para todas las variables, de modo que se cumplan todas las restricciones impuestas por las pistas y la lógica del juego.

Metodología

La actividad exige que se resuelvan 3 tableros de dificultad "muy dificil". La resolución del problema se dividió en las siguientes etapas:

1. Lectura y representación de datos

Se lee el tablero desde un archivo plano identificando las celdas blancas, negras y las que contienen pistas de suma. Estas se representan en un diccionario de variables en las que como dominio se les asigna 0 a las casillas negras que no son pistas, se le asignas los números del 1 al 9 a las casillas blancas, representando los posibles valores que puede tomar. Finalmente, las celdas negras que tienen valores como `" "` son los que contienen las pistas, pudiendo ser `"0 R"`, `"C 0"` o `"C R"`, con C y F siendo las pistas para la fila y la columa, son guardados completamente como un string.

2. Construcción de restricciones

Se generan las restricciones horizontales y verticales, excluyendo las celdas negras, siendo una lista de listas que contienen los valores por fila y columna que no deben repetirse. Luego, se agrupan secuencias consecutivas para formar los grupos de suma. Cada grupo queda representado como una lista de variables asociada a una suma total, cuyo primer elemento es la casilla con la pista, el segundo es una lista con las casillas que esta pista afecta y en el tercero se generan todas las permutaciones posibles que respetan la suma y la no repetición.

3. Modelado de vecinos

Cada variable se relaciona con las restricciones de suma a las que pertenece, lo cual permite evaluar combinaciones válidas durante la inferencia.

4. Inferencia por restricciones cruzadas

Para reducir dominios desde el inicio, se implementa un filtro de dominios válido mediante comparación cruzada con las combinaciones permitidas de los grupos de suma, eliminando valores incompatibles con las restricciones locales. Haciendo que el dominio de cada casilla blanca pueda cumplir con las dos sumas a las que pertenece

5. Consistencia de arco (AC-3)

Se aplica el algoritmo AC-3 para asegurar consistencia binaria entre variables de un mismo grupo, eliminando valores inconsistentes y reforzando la podadura de dominios antes de aplicar búsqueda.

6. Búsqueda con retroceso

Si luego de la inferencia aún hay variables no asignadas, se aplica un algoritmo de backtracking con selección heurística de variables con menor dominio. En cada paso se realiza nueva inferencia con AC-3 para mantener consistencia.

Implementación

Librerías

```
import itertools as it # Funciones de combinaciones y permutaciones
from collections import defaultdict # Para inicializar diccionarios en el que el
valor va a ser un elemento por defecto
from collections import deque # Para usar colas doblemente terminadas
import copy # Para hacer un copiado de estructuras más complejas
```

Lectura de datos

```
# Dominio de valores posibles para las celdas (1 a 9)
dom = set(range(1, 10))
# Identificadores de columnas del tablero
idCols = "ABCDEFGHI"

# Producto cartesiano de números y columnas para crear claves
keys = list(it.product(dom, idCols))
# Claves en formato "LetraNúmero" (ej: "A1")
strKeys = [f"{key[1]}{key[0]}" for key in keys]

# Diccionario con dominio completo para cada celda inicialmente
vars = {key: dom.copy() for key in strKeys}

# Lectura del archivo con el código del tablero y valores iniciales por celda
with open('c:/Juan UTP/Programación III/python/board3.txt', 'r') as f:
  code = f.readline().strip()  # Código identificador del tablero
  for key in vars.keys():
    valor = f.readline().strip()
```

```
    # Si la celda es negra o bloqueada se asigna {0}
    if "0" in valor:
      vars[key] = {int(0)}
    # Si la celda contiene pista o restricción se asigna el string con espacio
    if " " in valor:
      vars[key] = {valor}
```

Funciones de construcción

```
# Definición de restricciones de columnas: obtiene grupos de celdas blancas sin
bloqueos ni pistas
def DefColsConstraints(idCols, dom):
  colsconstraints = []
  for id in idCols:
    constraintsVars = [f"{id}{i}" for i in dom]
    # Filtrar celdas bloqueadas o con pista
    for key in list(constraintsVars):
      if 0 in vars.get(key, []) or any(" " in str(s) for s in vars.get(key, [])):
        constraintsVars.remove(key)
    colsconstraints.append(constraintsVars)
  return colsconstraints

# Definición de restricciones de filas: obtiene grupos de celdas blancas sin
bloqueos ni pistas
def DefRowsConstraints(idCols, dom):
  rowsconstraints = []
  for i in dom:
    constraintsVars = [f"{id}{i}" for id in idCols]
    # Filtrar celdas bloqueadas o con pista
    for key in list(constraintsVars):
      if 0 in vars.get(key, []) or any(" " in str(s) for s in vars.get(key, [])):
        constraintsVars.remove(key)
    rowsconstraints.append(constraintsVars)
  return rowsconstraints

# Ajusta los grupos de columnas para que sean secuencias contiguas
def FixColsConstraints(lists):
  result = []
  for sublist in lists:
    actualGroup = [sublist[0]]
    for i in range(1, len(sublist)):
      previous = sublist[i - 1]
      actual = sublist[i]
      # Verifica continuidad en la columna y filas consecutivas
      if previous[0] == actual[0] and int(actual[1:]) == int(previous[1:]) + 1:
        actualGroup.append(actual)
      else:
        result.append(actualGroup)
        actualGroup = [actual]
    result.append(actualGroup)
  return result
```

```
# Ajusta los grupos de filas para que sean secuencias contiguas
def FixRowsConstraints(lists):
  result = []
  for sublist in lists:
    actualGroup = [sublist[0]]
    for i in range(1, len(sublist)):
      previous = sublist[i - 1]
      actual = sublist[i]
      # Verifica continuidad en la fila y columnas consecutivas
      if int(previous[1]) == int(actual[1]) and ord(actual[0]) == ord(previous[0])
+ 1:
        actualGroup.append(actual)
      else:
        result.append(actualGroup)
        actualGroup = [actual]
    result.append(actualGroup)
  return result

# Genera combinaciones de m números cuya suma es n (para las restricciones de
suma)
def sumCombinations(n, m):
    numeros = range(1, 10)
    return [c for c in it.permutations(numeros, m) if sum(c) == n]

# Define las restricciones de suma para columnas basándose en las pistas
def DefSumColsConstraints(colsConstraintsDifference):
  sumConstraints = []
  for constraint in colsConstraintsDifference:
    key = constraint[0]
    # La celda que contiene la pista (suma objetivo) está justo arriba del grupo
    element = key[0] + str(int(key[1]) - 1)
    string = str(next(iter(vars[element])))
    n = int(string.split()[0])  # Suma objetivo para la columna
    combinations = sumCombinations(n, len(constraint))
    constraintData = [element, constraint, combinations]
    sumConstraints.append(constraintData)
  return sumConstraints

# Define las restricciones de suma para filas basándose en las pistas
def DefSumRowsConstraints(rowsConstraintsDifference):
  sumConstraints = []
  for constraint in rowsConstraintsDifference:
    key = constraint[0]
    # La celda que contiene la pista (suma objetivo) está a la izquierda del grupo
    element = str(chr(ord(key[0]) - 1)) + key[1]
    string = str(next(iter(vars[element])))
    n = int(string.split()[1])  # Suma objetivo para la fila
    combinations = sumCombinations(n, len(constraint))
    constraintData = [element, constraint, combinations]
    sumConstraints.append(constraintData)
  return sumConstraints
```

Funciones de consistencia

```
# Propagación: reduce dominios en función de las combinaciones válidas que
coinciden con vecinos
def DomainCrossing(neighbors, varDoms):
  for cell, clues in neighbors.items():
    for clue in clues:
      group = next((g for g in sumConstraints if g[0] == clue), None)
      if group is None:
        continue
      groupCells = group[1]
      combinations = group[2]
      if cell not in groupCells:
        continue

      i = groupCells.index(cell)
      possible_values = set()
      for comb in combinations:
        valid = True
        for j, var in enumerate(groupCells):
          if var == cell:
            continue
          if comb[j] not in varDoms[var]:
            valid = False
            break
        if valid:
          possible_values.add(comb[i])
      varDoms[cell] = varDoms[cell].intersection(possible_values)
  return varDoms

# En cada grupo, si una celda tiene valor fijo, eliminar ese valor de dominios de
las demás (restricción de diferencia)
def ConsistenceDifference(constraints, varDoms):
  for constraint in constraints:
    for var in constraint:
      if len(varDoms[var]) == 1:
        for varAux in constraint:
          if varAux != var:
            varDoms[varAux].discard(list(varDoms[var])[0])
  return varDoms
```

Funciones auxiliares

```
# Construye el diccionario de vecinos (pistas a las que pertenece cada celda
blanca)
def Neighbors(sumConstraints):
  neighbors = {var: [] for var in vars}
  # Lista de todas las celdas blancas (no bloqueadas ni pistas)
  whiteCells = list(set([clave for sublista in differenceConstraints for clave in
sublista]))
```

```
  neighbors = {k: v for k, v in neighbors.items() if k in whiteCells}
  # Para cada grupo de suma, añadir la pista como vecino de las celdas
  for group in sumConstraints:
    groupName = group[0]
    groupCells = group[1]
    for cell in groupCells:
      if cell in whiteCells and groupName not in neighbors[cell]:
        neighbors[cell].append(groupName)
  return neighbors

# Imprime el tablero de Kakuro con el estado actual de variables
def PrintBoard(vars):
  print("Kakuro Code: ", code, "\n")
  print(" ".join("  ABCDEFGHI"))
  print(" ".join("  ---------"))
  for i in range(1, 10):
    row = []
    for c in "ABCDEFGHI":
      cell = f"{c}{i}"
      val = vars[cell]
      if val == {0}:
        row.append("0")      # Celda negra
      elif any(" " in str(v) for v in val):
        row.append("X")      # Celda con pista
      elif len(val) == 1:
        row.append(str(list(val)[0]))  # Celda con valor único asignado
      else:
        row.append(".")      # Celda sin valor fijo aún
    print(i, "|", " ".join(row))

# Obtener arcos para AC-3: pares de variables en el mismo grupo de suma
def GetArcs(sumConstraints):
  arcs = []
  for group in sumConstraints:
    cells = group[1]
    for xi in cells:
      for xj in cells:
        if xi != xj:
          arcs.append((xi, xj))
  return arcs
```

Algoritmo AC-3

```
# Revisar consistencia entre xi y xj; eliminar valores inconsistentes de dominio
de xi
def Revise(xi, xj, domains, sumConstraints):
  revised = False
  to_remove = set()

  relevant_groups = [g for g in sumConstraints if xi in g[1] and xj in g[1]]
```

```
    for vi in domains[xi]:
      valid = False
      for group in relevant_groups:
        cells = group[1]
        combos = group[2]
        idx_xi = cells.index(xi)
        idx_xj = cells.index(xj)

        for combo in combos:
          if combo[idx_xi] == vi and combo[idx_xj] in domains[xj]:
            compatible = True
            for k, cell in enumerate(cells):
              if cell == xi or cell == xj:
                continue
              if combo[k] not in domains[cell]:
                compatible = False
                break
            if compatible:
              valid = True
              break
        if valid:
          break
      if not valid:
        to_remove.add(vi)
        revised = True

    if revised:
        domains[xi] -= to_remove

    return revised

  # Algoritmo AC-3 para propagación de restricciones
  def AC3(domains, sumConstraints):
    queue = deque(GetArcs(sumConstraints))
    while queue:
      xi, xj = queue.popleft()
      if Revise(xi, xj, domains, sumConstraints):
        if not domains[xi]:
          return False
        for group in sumConstraints:
          cells = group[1]
          if xi in cells:
            for xk in cells:
              if xk != xi and xk != xj:
                queue.append((xk, xi))
    return domains

  # Inferencia combinada: cruces de dominio + diferencia + AC-3, iterada hasta
  estabilizar dominios
  def InferWithAC3(vars, sumConstraints, differenceConstraints):
    varDoms = copy.deepcopy(vars)
    while True:
      old = copy.deepcopy(varDoms)
      varDoms = DomainCrossing(neighbors, varDoms)
```

```
    varDoms = ConsistenceDifference(differenceConstraints, varDoms)
    varDoms = AC3(varDoms, sumConstraints)
    if varDoms == old:
      break
  return varDoms
```

Búsqueda con backtracking

```
# Verifica si el tablero está completamente asignado
def IsComplete(domains):
  return all(len(domains[v]) == 1 for v in domains if domains[v] != {0} and not
any(" " in str(s) for s in domains[v]))

# Selecciona la siguiente variable para asignar usando heurística MRV (mínimo
dominio)
def SelectUnassignedVariable(domains):
  candidates = [v for v in domains if len(domains[v]) > 1 and domains[v] != {0}
and not any(" " in str(s) for s in domains[v])]
  return min(candidates, key=lambda v: len(domains[v])) if candidates else None

# Algoritmo de búsqueda con retroceso usando inferencia AC-3 para asignar valores
def BacktrackingSearch(domains):
  if IsComplete(domains):
    return domains
  var = SelectUnassignedVariable(domains)
  if var is None:
    return domains
  for value in domains[var]:
    new_domains = copy.deepcopy(domains)
    new_domains[var] = {value}
    new_domains = InferWithAC3(new_domains, sumConstraints, differenceConstraints)
    if new_domains:
      result = BacktrackingSearch(new_domains)
      if result:
        return result
  return False
```

Ejecución

```
# Obtener las restricciones por columnas y filas
colsConstraints = DefColsConstraints(idCols, dom)
rowsConstraints = DefRowsConstraints(idCols, dom)

# Filtrar listas vacías (sin celdas blancas)
colsConstraintsDifference = [sublist for sublist in colsConstraints if sublist]
rowsConstraintsDifference = [sublist for sublist in rowsConstraints if sublist]

# Ajustar grupos para que sean contiguos
fixCols = FixColsConstraints(colsConstraintsDifference)
```

```
fixRows = FixRowsConstraints(rowsConstraintsDifference)

# Definir restricciones de suma para columnas y filas ajustadas
colsSumConstraints = DefSumColsConstraints(fixCols)
rowsSumConstraints = DefSumRowsConstraints(fixRows)

# Combinar todas las restricciones de suma
sumConstraints = colsSumConstraints + rowsSumConstraints
# Combinar las listas de grupos contiguos para restricciones de diferencia
differenceConstraints = fixCols + fixRows

# Diccionario para almacenar restricciones que afectan cada celda
cellConstraints = defaultdict(list)

# Construir vecinos según las restricciones de suma
neighbors = Neighbors(sumConstraints)

# Ejecutar inferencia inicial con AC-3
vars = InferWithAC3(vars, sumConstraints, differenceConstraints)

# Si no está completo, aplicar búsqueda con retroceso
if not IsComplete(vars):
  vars = BacktrackingSearch(vars)

# Imprimir el tablero final (resuelto o parcialmente resuelto)
PrintBoard(vars)
```

Pruebas realizadas

Se han resuelto tableros con estructuras diversas y múltiples restricciones de suma, simulando la dificultad “Muy difícil”. Los resultados se validaron visualmente y por revisión del cumplimiento de las reglas del juego.

- **Tablero 1:** ProgIIIG1-Act08-KK5IFHWH-Board.txt
- **Tablero 2:** ProgIIIG1-Act08-KK5BOEIP-Board.txt
- **Tablero 3:** ProgIIIG1-Act08-KK5TIIKZ-Board.txt

Cada tablero se almacenó en un archivo plano siguiendo el formato definido por el grupo. El sistema es capaz de detectar celdas negras, reconocer restricciones y resolver el tablero completo.

Discusión

El enfoque CSP resulta altamente efectivo para modelar y resolver instancias de Kakuro. Las estrategias de inferencia temprana como la **poda de dominios cruzados** y la aplicación de **consistencia de arco (AC-3)** permiten reducir significativamente la necesidad de retrocesos.

Las técnicas utilizadas evitan que el algoritmo explore combinaciones inválidas desde el inicio, y la aplicación de **heurísticas de selección de variable** mejora aún más la eficiencia de la búsqueda.

Este modelo también puede extenderse a tableros de mayor tamaño o adaptarse a variantes del juego.

Conclusiones

- Se logró modelar el juego de Kakuro como un problema CSP completo, utilizando Python y técnicas avanzadas de inferencia.
- La implementación fue capaz de resolver tableros de dificultad muy alta gracias a la combinación de filtrado de dominios, consistencia de arco y búsqueda con heurística.
- El enfoque es modular, reutilizable y adaptable a nuevos retos.
- Las estrategias utilizadas emulan el razonamiento humano y optimizan el uso de recursos computacionales.