**DPML project 1**

Tudor Cosmin Oanea, ICA/2

### **1. Introduction**

In this project, I implemented a Python-based solver for Constraint Satisfaction Problems (CSPs). CSPs are mathematical problems defined by three key elements: variables, domains, and constraints. Each problem involves a set of variables, each of which can take on a value from a specific domain. Constraints define the rules or relationships that restrict the possible values that the variables can simultaneously take. The goal of a CSP solver is to find values for all variables such that all constraints are satisfied.

CSPs appear in many real-world scenarios, from scheduling tasks to solving puzzles, and their structured nature makes them an excellent candidate for algorithmic solutions. In my implementation, I focused on creating a generalized solver that can adapt to any CSP. By simply specifying the variables, domains, and constraints of a problem, the solver can find a solution (or determine that no solution exists).

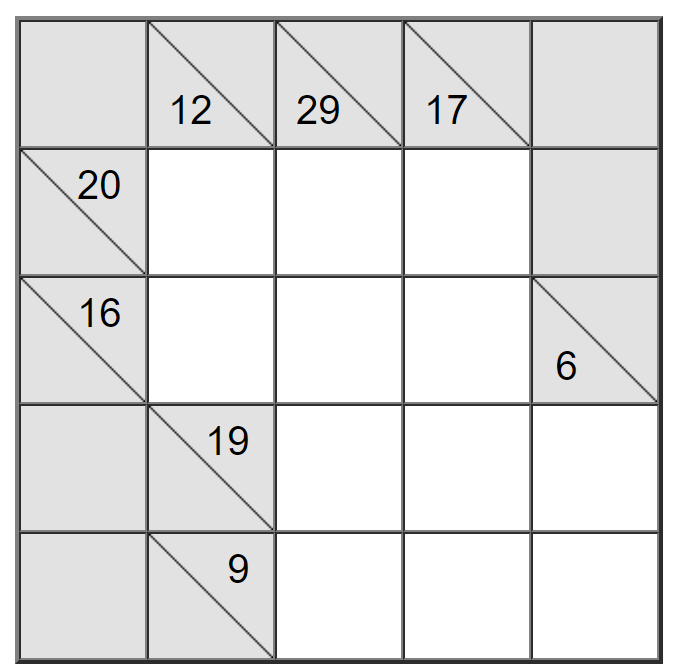
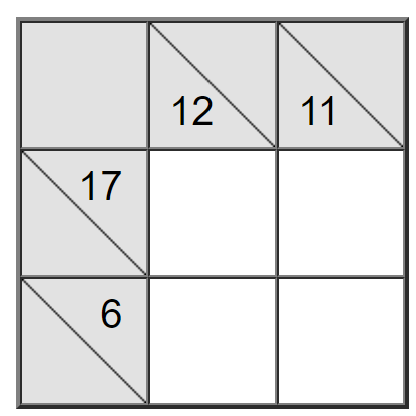
To demonstrate the versatility of the solver, I selected two distinct types of CSPs: **Kakuro**, a grid-based numerical puzzle, and a **Logic Equations Solver**, where relationships between variables are solved through equations. These examples showcase the power of the solver to handle both combinatorial and logical constraints.

### **2. Selected Problems**

To illustrate the functionality and versatility of my CSP solver, I selected two distinct types of problems: **Kakuro** and **Logic Equations**. These problems are fundamentally different in structure and constraint types, providing a diverse testbed for the solver. Below, I will describe each problem in detail, focusing on their rules and how they are framed as CSPs.

#### **2.1 Kakuro**

Kakuro is a numerical puzzle that combines elements of Sudoku and crossword puzzles. The puzzle consists of a grid where some cells are shaded and others are left blank. Shaded cells contain numbers that act as clues, specifying the target sum for the adjacent blank cells in either a row or a column.

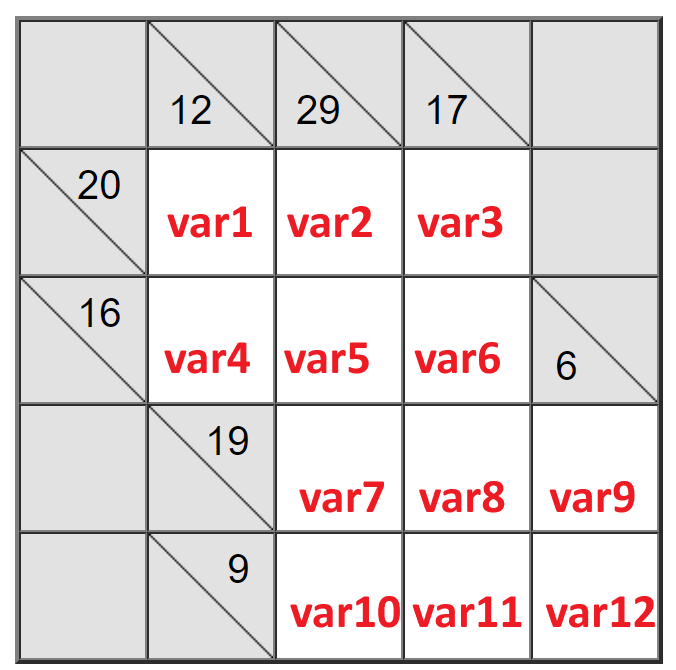
 

The objective of Kakuro is to fill in the blank cells with digits from 1 to 9 so that:

1. The sum of the numbers in a group of connected blank cells matches the clue in the corresponding shaded cell.
2. No digit repeats within a single group contributing to a specific sum.

**Framing Kakuro as a CSP**:

* **Variables**: Each blank cell in the grid is a variable.
* **Domains**: Each variable (blank cell) can take a value from 1 to 9.
* **Constraints**:
  + The sum of the values of variables in a group must equal the target sum of the corresponding shaded cell.
  + Values within a group must be unique (no duplicates).



Kakuro challenges the solver to efficiently handle combinatorial constraints and explore multiple valid combinations until a correct solution is found.

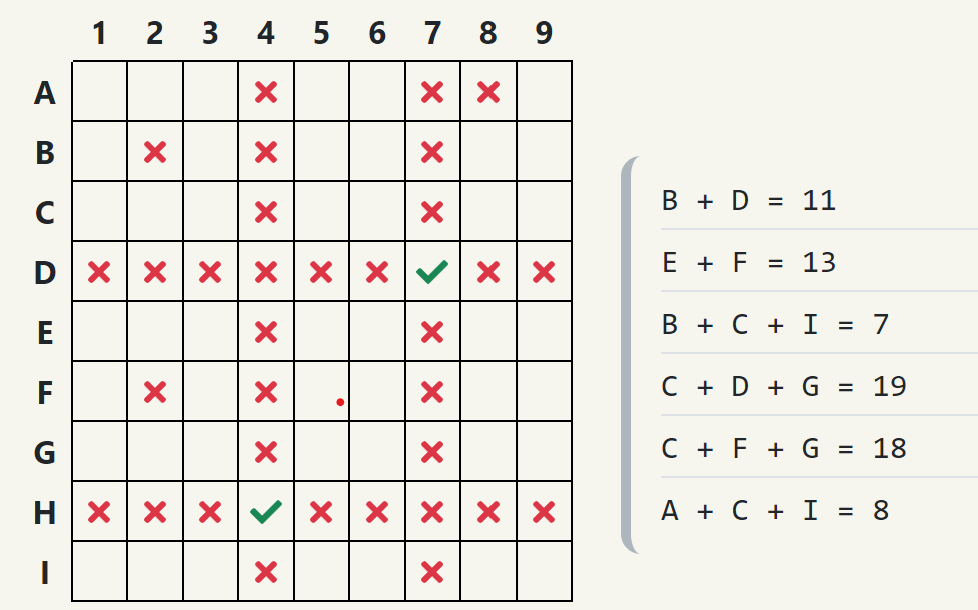
### **2.2 Logic Equations**

The Logic Equations problem is a grid-based puzzle that revolves around solving mathematical relationships between variables. Each variable represents a letter (e.g., A, B, C, etc.) and must satisfy a set of constraints in the form of equations or inequalities. The grid visually represents possible values for the variables, where a checkmark indicates a valid assignment and a cross marks invalid options.

For example, consider the following equations:

* B + D = 1
* E + F = 13
* B + C + I = 7
* C + D + G = 19
* C + F + G = 18
* A + C + I = 8

The goal is to assign values to variables A,B,C,D,E,F,G,H,IA, B, C, D, E, F, G, H, IA,B,C,D,E,F,G,H,I such that all equations hold true.



**Framing Logic Equations as a CSP**:

* **Variables**: Each variable (e.g., A,B,C,…) represents an unknown in the equations.
* **Domains**: Each variable has a specific range of possible integer values (e.g., 1 to 9).
* **Constraints**: The constraints are defined by the given equations. For example:
  + The sum of B and D must equal 11.
  + The sum of C, F and G must equal 18.
  + Every variable can only take on a single value.

Logic Equations provide a great example of how CSPs can model and solve puzzles with arithmetic relationships. The inclusion of multiple equations ensures that the solver must carefully navigate interdependent constraints to find the correct solution.

### **3. Code Implementation of the CSP Class**

The **CSP** class serves as the foundation for solving any Constraint Satisfaction Problem. It is designed to be general and flexible, allowing it to adapt to various types of problems by defining variables, domains, and constraints. Below, I will provide an explanation of its structure and functionality, supported by code snippets.

The constructor initializes the CSP class by defining the three core components of a CSP: variables, domains, and constraints. It also sets a time limit for backtracking operations.

| class CSP:  def \_\_init\_\_(self, variables, domains, constraints):  self.variables = variables  # variables: list of possible variables [var1, var2, var3, ...]   self.domains = domains  # domains = dictionary of pairs var : list of possible values  # {  # var1: [1, 2, 3, ...],  # var2: [1, 2, 3, ...],  # ...  # }   self.constraints = constraints  # constraints = list of functions (each ones checks a constraint based on the variables and their values)   self.BKT\_TIME\_LIMIT = 10 |
| --- |

* **Variables**: A list of identifiers representing the problem's unknowns (e.g. A, B, C… for Logic Equations).
* **Domains**: A dictionary mapping each variable to its possible values. For example, in Kakuro, a variable representing a blank cell might have a domain of [1, 2, 3…].
* **Constraints**: A list of constraint-checking functions. These functions evaluate whether a set of variable assignments satisfies the problem's rules. For example, a function might ensure that the sum of variables matches a specific target.

### **4. Search Methods**

Constraint Satisfaction Problems can be solved using various search strategies. In this chapter, I describe the **baseline solution**, implemented as the **backtracking search algorithm**, which serves as the foundation for solving CSPs. This method is both simple and effective, iteratively assigning values to variables and backtracking when constraints are violated.

**4.1 Backtracking Solution (Baseline Solution)**

The backtracking algorithm is a recursive depth-first search method designed to explore the solution space of a CSP efficiently. It assigns values to variables one at a time and backtracks whenever a constraint is violated. This method ensures that no invalid partial solutions are further expanded, thus reducing unnecessary computations.

##### **How It Works**

1. **Initialization**: Start with an empty state where no variables have been assigned values.
2. **Variable Assignment**:
   * Select an unassigned variable.
   * Try assigning a value from the variable's domain.
3. **Constraint Checking**: After assigning a value, check if all constraints are satisfied.
4. **Recursive Search**: If the partial assignment is valid, recursively attempt to assign values to the remaining variables.
5. **Backtracking**:
   * If no valid assignment is found for a variable, backtrack by removing the last assignment and trying the next value.
6. **Termination**:
   * If all variables are assigned values that satisfy the constraints, a solution is found.
   * If the time limit is exceeded or no solution exists, terminate the search.

##### **Implementation**

Below is the implementation of the **backtracking\_solution** method:

| def backtracking\_solution(self, state={}, start\_time=None):  if start\_time is None:  start\_time = time.time() # Initialize the start time if not already set   # Check if the time limit has been exceeded  if time.time() - start\_time > self.BKT\_TIME\_LIMIT:  return None   # If assignment is complete, return the state as a solution  if self.is\_complete(state):  # Check if the complete state satisfies all constraints  if self.check\_constraints(state):  return state  else:  return None   # Select the next variable to assign  for var in self.variables:  if var not in state:  break   # Try each value in the domain for the selected variable  for value in self.domains[var]:  # Create a new assignment for the variable  new\_state = state.copy()  new\_state[var] = value   # Proceed with backtracking  result = self.backtracking\_solution(new\_state, start\_time)  if result:  return result   # If no value leads to a solution, backtrack  return None |
| --- |

The **backtracking solution** relies on several key supporting functions to ensure its operation. The is\_complete(state) function checks whether all variables have been assigned values, signaling the completion of the current state. The check\_constraints(state) function verifies that all assigned variables satisfy the defined constraints, ensuring the validity of the solution at every step. Additionally, the select\_unassigned\_variable(state) function selects the next variable to be assigned from the list of variables, facilitating the recursive process of backtracking.

This algorithm has notable strengths, including completeness and correctness. It guarantees a solution if one exists within the specified time limit and ensures that all explored states adhere to the problem's constraints. However, it also has limitations. The baseline implementation lacks heuristics or advanced pruning techniques, which makes it inefficient for large or complex problems. Moreover, its time sensitivity means that it may fail to find a solution within the given BKT\_TIME\_LIMIT for particularly challenging scenarios. Despite these limitations, the backtracking solution provides a solid foundation for solving CSPs and can be extended with more advanced optimizations.

**4.2 Chronological Backtracking Solution**

The **chronological\_backtracking\_solution** method refines the baseline backtracking approach by allowing partial assignments to pass preliminary constraint checks, enabling the search to proceed without prematurely rejecting potentially valid solutions. The key difference lies in how constraints are evaluated during the search, with partial states being handled gracefully. Below is the implementation of this method:

| def chronological\_backtracking\_solution(self, state={}):  # If assignment is complete, return the state as a solution  if self.is\_complete(state):  return state    # Select an unassigned variable  var = self.select\_unassigned\_variable(state)   # Try each value in the domain for the selected variable  for value in self.domains[str(var)]:  # Create a new assignment for the variable  new\_state = state.copy()  new\_state[str(var)] = value   # Check if the current assignment satisfies all constraints  if self.check\_chronological\_constraints(new\_state):  result = self.chronological\_backtracking\_solution(new\_state)  if result:  return result   # If no value leads to a solution, backtrack  return None |
| --- |

In this implementation, the method starts by checking if all variables have been assigned values using the is\_complete function. If so, the current state is returned as a solution. If the state is incomplete, an unassigned variable is selected using select\_unassigned\_variable. For each value in the selected variable's domain, the algorithm creates a new state with the current variable-value pair added to the assignment.

The check\_chronological\_constraints function is crucial here. It iterates over all constraints and evaluates them against the current state. If a constraint involves unassigned variables, it handles the partial assignment by catching KeyError exceptions and continuing without rejecting the state. This allows the algorithm to delay evaluating certain constraints until all involved variables have been assigned values. If the partial state satisfies all constraints, the method recursively calls itself to continue the search with the updated state. If a valid solution is found in any recursive call, it is immediately returned. If no valid assignment exists for the current variable, the algorithm backtracks to explore other values, eventually returning None if no solution is possible.

This method improves upon the baseline by enabling flexibility in handling constraints during partial assignments, making it suitable for problems where certain constraints cannot be fully evaluated until later in the search. While it does not fundamentally address the inefficiencies of the depth-first search approach, it offers a more robust solution for CSPs with interdependent constraints.

### **4.3 Hill Climbing Solution**

The **hill\_climbing\_solution** method implements a local search algorithm to solve CSPs by iteratively improving an initial random state. This heuristic approach focuses on minimizing the number of conflicts in the current state until a solution is found or a local optimum is reached. Below is the implementation of the hill climbing method:

| def hill\_climbing\_solution(self, max\_restarts=10000):  for restart in range(max\_restarts):  # Generate a random initial state  current\_state = self.generate\_initial\_state()  current\_score = self.conflict\_score(current\_state)   while True:  if current\_score == 0:  # Found a solution  return current\_state   # Try to improve the state by modifying one variable  best\_neighbor = None  best\_score = float('inf')   # Search for the best option in adjacent neighbors  for var in self.variables:  for value in self.domains[var]:  if value != current\_state[var]:  # Create a new state with the updated variable  neighbor = current\_state.copy()  neighbor[var] = value   # Evaluate the neighbor's conflict score  neighbor\_score = self.conflict\_score(neighbor)   # Keep track of the best neighbor  if neighbor\_score < best\_score:  best\_neighbor = neighbor  best\_score = neighbor\_score   # Update the current state if a better neighbor was found  if best\_score < current\_score:  current\_state = best\_neighbor  current\_score = best\_score  else:  # No improvement possible, terminate this hill-climbing attempt  break   # Restart if no solution was found in this run  # print(f"Restarting... (attempt {restart + 1})")   # If all restarts fail, return None (shouldn't happen if the problem is solvable)  return None |
| --- |

This method begins by generating a random initial state using the generate\_initial\_state method, which assigns random values to all variables from their respective domains. The current state's quality is measured by the conflict\_score function, which counts the number of violated constraints. If the initial state has no conflicts (current\_score == 0), it is immediately returned as the solution. If the state is not conflict-free, the algorithm explores "neighboring" states, where a single variable's value is changed. For each neighboring state, the conflict score is calculated, and the state with the lowest score is chosen as the next state. This process continues iteratively, improving the state until no better neighbors can be found (indicating a local optimum) or the conflict score reaches zero (indicating a solution). To avoid being trapped in local optima, the algorithm supports multiple restarts, where it generates a new random initial state and repeats the process. The number of allowed restarts is specified by the max\_restarts parameter. If no solution is found after exhausting all restarts, the method returns None, though this is unlikely for solvable problems.

The hill climbing solution is particularly suited for large CSPs where traditional backtracking methods may be inefficient, as it focuses on improving a single solution iteratively rather than exploring the entire search space.

### **5. Problem Reduction Methods**

Problem reduction methods are preprocessing techniques designed to simplify Constraint Satisfaction Problems (CSPs) before attempting to solve them. These methods aim to reduce the search space by eliminating inconsistent values or reducing complex constraints into simpler forms, making the solving process more efficient. Two widely used problem reduction techniques are **arc consistency** and **path consistency**, which systematically enforce consistency among variables in the problem. These techniques do not solve the CSP outright but ensure that the problem is easier to solve with search algorithms.

#### **5.1 Arc Consistency**

Arc consistency is a method that ensures every value in the domain of a variable is compatible with at least one value in the domain of every other variable it is constrained with. If a value in a variable’s domain has no compatible value in a neighboring variable’s domain, it is removed from the domain. The implementation of arc consistency in this CSP class revolves around the revise\_domain function. This function iterates over the values in the domain of a variable and checks if each value is consistent with at least one value in the domain of a connected variable. If no such consistent value exists, the inconsistent value is removed from the domain. The arc\_consistency method uses a queue-based approach to ensure that all arcs are processed iteratively until no further changes occur. If a domain becomes empty during this process, it indicates that the CSP is unsolvable.

| def revise\_domain(self, x, y):  # Revise the domain of variable x to ensure arc-consistency with variable y.  # Removes values from the domain of x that have no compatible value in the domain of y.  # Returns True if a value is removed; otherwise, False.  revised = False  x\_domain = self.domains[x]  y\_domain = self.domains[y]   # Iterate over initial x\_domain (and make changes to the real domain in real time)  for value\_x in list(x\_domain):  consistent = False   # Check all values in y's domain for compatibility with value\_x  for value\_y in y\_domain:  if self.satisfies\_constraints(x, value\_x, y, value\_y):  consistent = True  break # At least one valid value\_y is found   # If no consistent value\_y exists, remove value\_x from x\_domain  if not consistent:  x\_domain.remove(value\_x)  revised = True   return revised |
| --- |

| def arc\_consistency(self):  # Implements the arc consistency algorithm.   # Step 1: Initialize the queue with all arcs  queue = [(x, y) for x in self.variables for y in self.variables if x != y]  # print(queue)   # Step 2: Process the queue until no changes  while queue:  x, y = queue.pop(0) # Dequeue an arc  if self.revise\_domain(x, y): # Revise the domain of x  if not self.domains[x]: # If the domain of x is empty, CSP is unsolvable  return False  # If x's domain is reduced, add all related arcs back to the queue  for z in self.variables:  if z != x and z != y:  queue.append((z, x))   return True |
| --- |

Arc consistency reduces the search space by eliminating values that would inevitably lead to constraint violations. However, it is not a complete solution and serves as a preprocessing step that simplifies the CSP.

#### **5.2 Path Consistency**

Path consistency extends the idea of arc consistency to triplets of variables. It ensures that for every pair of variables X and Y, the constraints involving a third variable Zare also satisfied. This is achieved by verifying that for every value of X, there exists a consistent pair of values for Y and Z that satisfies all constraints.

The path\_consistency method implements this by iterating through all triplets of variables and performing a process called relation composition. The revise\_path\_domain function ensures that each value in X’s domain is consistent with some valid combination of values in the domains of Y and Z. If no valid combination exists, the value is removed from X’s domain. The algorithm continues until no further changes are detected or an empty domain is encountered, indicating that the CSP is unsolvable.

| def revise\_path\_domain(self, x, y, z):  # Revise the domain of variable x to ensure path-consistency with y and z.  # Uses relation composition to remove values from x's domain that cannot participate  # in valid triplets with y and z.  # Returns True if the domain of x is modified; otherwise, False.  revised = False # Track if x's domain is modified  x\_domain = self.domains[x]  y\_domain = self.domains[y]  z\_domain = self.domains[z]   # Iterate over a copy of x's domain to allow safe modification  for value\_x in list(x\_domain):  valid = False # Track if value\_x is consistent in any triplet   # Check all combinations of y and z values  for value\_y in y\_domain:  for value\_z in z\_domain:  # Check if (x=value\_x, y=value\_y, z=value\_z) satisfies all relevant constraints  if self.satisfies\_path\_constraints(x, value\_x, y, value\_y, z, value\_z):  valid = True  break # Stop checking further if a valid triplet is found  if valid:  break   # If no valid triplet exists, remove value\_x from x's domain  if not valid:  x\_domain.remove(value\_x)  revised = True   return revised |
| --- |

| def path\_consistency(self):  # Implements the naive path consistency algorithm  # Ensures that for every triplet of variables (X, Y, Z), their constraints are consistent.  # Returns True if the CSP is path-consistent, False if an empty domain is encountered.    # Get the list of variables  variables = self.variables   # Iterate until no changes occur  while True:  changes = False # Track if any domain or constraint is modified   # Iterate over all triplets of variables (X, Y, Z)  for x in variables:  for y in variables:  for z in variables:  # Skip triplets where variables are not distinct  if x == y or y == z or x == z:  continue   # Perform relation composition and update the domain of X  if self.revise\_path\_domain(x, y, z):  changes = True   # If a domain becomes empty, the CSP is unsolvable  if not self.domains[x]:  return False   # If no changes occurred, stop  if not changes:  break   return True |
| --- |

Path consistency is particularly useful for problems where constraints involve more than two variables. It provides a deeper level of preprocessing by considering the interactions between triplets of variables, making the CSP easier to solve.

Both arc consistency and path consistency serve as valuable preprocessing techniques for reducing the complexity of CSPs. Arc consistency simplifies the problem by ensuring that values in a variable's domain are compatible with neighboring variables. Path consistency takes this further by ensuring consistency across triplets of variables. While these techniques do not guarantee a solution, they significantly reduce the search space, making subsequent solving methods more efficient and practical for large or complex problems.

### **6. Comparison of CSP Solving Methods**

To evaluate the performance of the implemented CSP solving methods, I conducted a systematic comparison using a Python-based benchmarking framework. This comparison involved running the three different algorithms, backtracking\_solution, chronological\_backtracking\_solution, and hill\_climbing\_solution on a common CSP instance under varying preprocessing conditions. These conditions included no preprocessing, arc consistency, and path consistency. The goal was to assess the impact of these algorithms and preprocessing techniques on solution time and success rates.

#### **6.1 Experimental Setup**

The comparison was executed using the compare\_csp\_algorithms function, which automates the benchmarking process for different CSP solving methods. Each method was tested with a timeout of 9 seconds to ensure fair performance evaluation and avoid infinite computations for unsolvable instances. The function used multithreading to manage timeouts and capture results effectively.

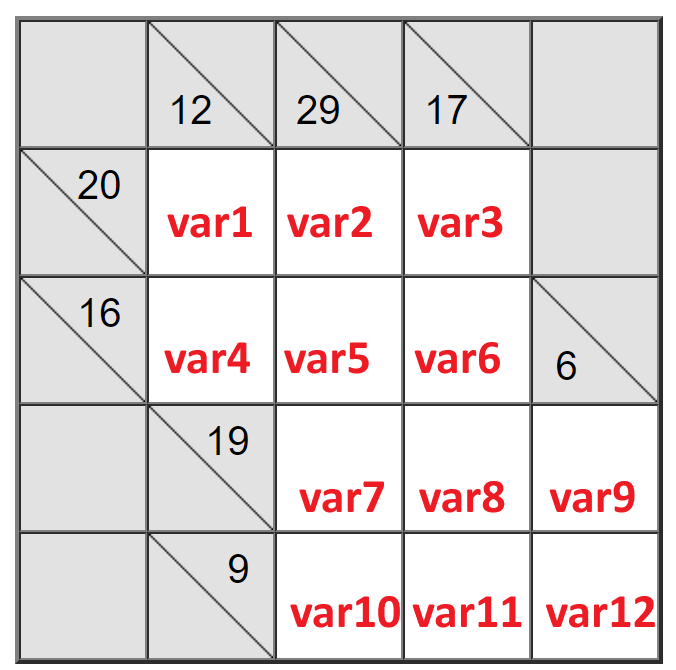
The CSP solving methods were executed under three scenarios:

1. No Preprocessing: The algorithms were run on the initial CSP setup without any preprocessing.
2. Arc Consistency: Arc consistency was applied to the CSP to reduce the domains of variables by eliminating inconsistent values before running the algorithms.
3. Path Consistency: Path consistency was applied to further simplify the problem by enforcing consistency among triplets of variables.

### **6.2. Mapping the Kakuro Game to the CSP Class**

The Kakuro puzzle in the provided image was mapped to the CSP class by defining the grid's structure in terms of variables, domains, and constraints. This process involved translating the numerical and uniqueness rules of Kakuro into constraints that the CSP solver can understand and solve systematically.

Each blank cell in the Kakuro grid was assigned a unique variable identifier (var1, var2, ..., var12) corresponding to its position in the puzzle. These variables represent the unknown values that need to be determined. The domain for each variable was set to [1,2,3,...,9], as the puzzle only allows numbers between 1 and 9 to be placed in each cell.



#### **Constraints**

To frame the Kakuro puzzle as a CSP, constraints were defined to enforce both the sum rules and uniqueness rules.

##### 1. Horizontal Sum Constraints

Each row in the grid has a shaded cell with a number indicating the target sum of the variables in that row. For example:

* Row 1: var1+var2+var3=20
* Row 2: var4+var5+var6=16

These constraints were implemented as functions that validate whether the assigned values satisfy the target sum for each row.

##### 2. Vertical Sum Constraints

The vertical shaded cells in the grid also specify the target sum for each column. For example:

* Column 1: var1+var4=12
* Column 2: var2+var5+var7+var10=29

These constraints were implemented similarly to the horizontal constraints, ensuring the sum rule is enforced for each column.

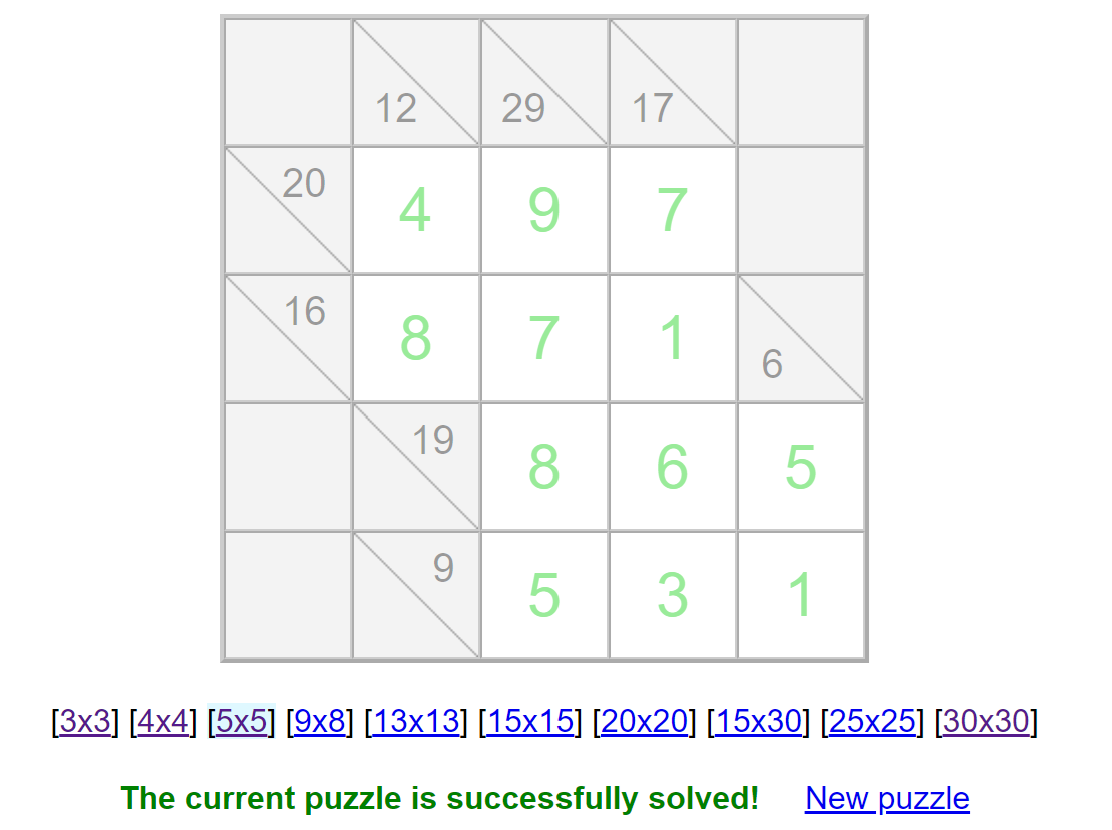
##### 3. Uniqueness Constraints

Kakuro also enforces that no number can repeat within a group (row or column) contributing to a sum. These constraints ensure that the values assigned to variables in a group are distinct.

For example, in the first row:

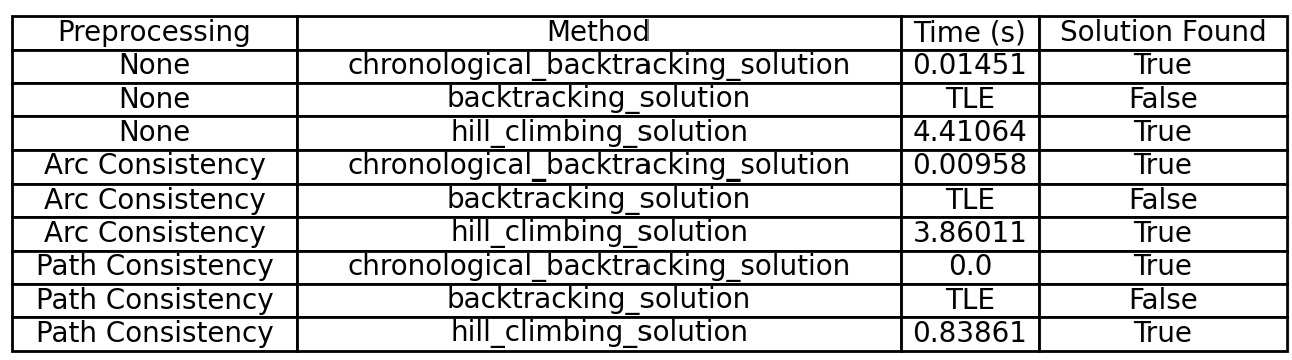
* var1,var2,var3 must all be unique.

The CSP solver successfully solved the Kakuro puzzle, producing the solution shown in the second image. Each cell in the grid was assigned a value such that all sum and uniqueness constraints were satisfied. This demonstrates the flexibility of the CSP class in solving complex, real-world puzzles like Kakuro by systematically exploring valid assignments.



The comparison table below summarizes the performance of three CSP solving methods on a 5x5 Kakuro puzzle. Each method was tested under three preprocessing conditions: no preprocessing, arc consistency, and path consistency. The results highlight the time taken by each method to find a solution (or timeout if a solution was not found) and whether the solution was successfully found.

The performance analysis of the Kakuro problem solving methods demonstrates clear distinctions in efficiency and effectiveness across different approaches and preprocessing techniques:



1. Chronological Backtracking:
   * Without preprocessing, chronological backtracking solved the problem quickly, taking only 0.01451 seconds.
   * Preprocessing further improved its performance. With arc consistency, the runtime reduced to 0.00958 seconds, and with path consistency, it achieved the best performance of 0.0 seconds.
   * This highlights its robustness and adaptability to preprocessing.
2. Backtracking Solution:
   * Basic backtracking consistently failed to solve the problem within the time limit (TLE) across all conditions, indicating its inefficiency for problems of this complexity.
   * Even with arc or path consistency preprocessing, it was unable to produce a solution, underscoring the limitations of this method for large search spaces.
3. Hill-Climbing Solution:
   * Hill-climbing was significantly slower without preprocessing, taking 4.41064 seconds to solve the problem.
   * However, preprocessing had a substantial positive impact. With arc consistency, the runtime dropped to 3.86011 seconds, and with path consistency, it achieved a much faster runtime of 0.83861 seconds.
   * While still slower than chronological backtracking, it demonstrated notable improvements when preprocessing was applied.

### Key Observations:

* Path consistency preprocessing provided the most significant performance gains across all methods, especially for hill-climbing and chronological backtracking.
* Chronological backtracking proved to be the most efficient and reliable method, consistently solving the puzzle in negligible time, particularly when preprocessing was applied.
* Basic backtracking, even with preprocessing, was not suitable for this problem due to its inability to converge within the set time limit.
* Hill-climbing, while less efficient than chronological backtracking, benefited greatly from preprocessing and could be a viable alternative with further optimization.

This analysis emphasizes the importance of choosing an appropriate solving method and leveraging preprocessing techniques to enhance solver efficiency for constraint satisfaction problems like Kakuro.

### **6.3. Mapping the Logic Expression Solver to the CSP Class**

The logic puzzle was systematically mapped to the CSP class by defining its structure in terms of variables, domains, and constraints. Each variable corresponds to a letter in the grid, uniquely identified (e.g., A, B, C). The domain of each variable was set to the range [1, 2, ..., 9], representing the possible integer values the variables can take.

The constraints reflect the rules and relationships of the logic puzzle:

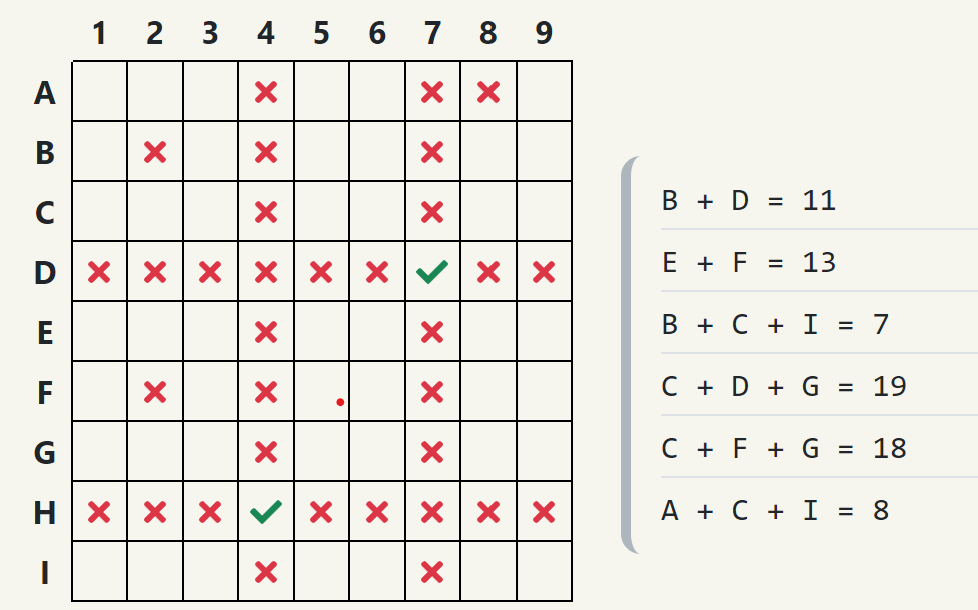
1. Equation Constraints

These numerical constraints enforce the relationships between the variables based on the provided equations.  
Each constraint was implemented as a function that evaluates the state of variable assignments and returns True if the equation is satisfied.

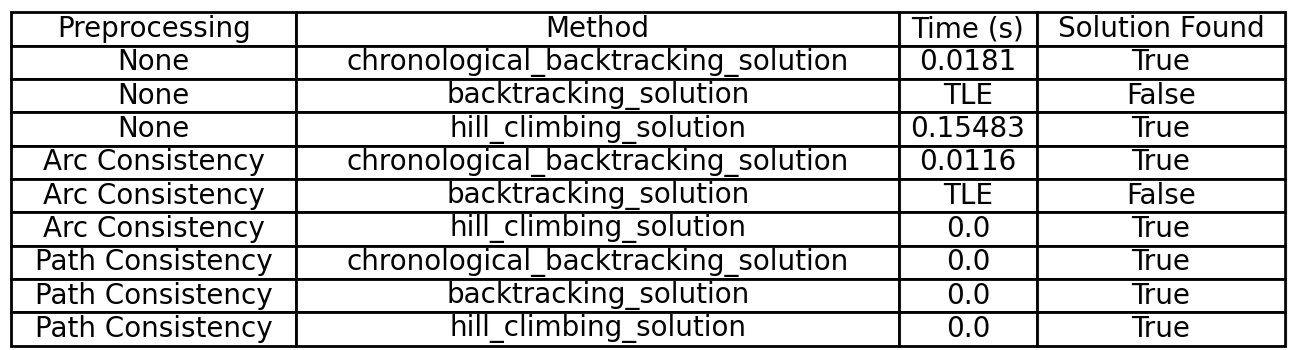
1. Uniqueness Constraints

To enforce the rule that no two variables can share the same value, pairwise uniqueness constraints were generated for all possible variable pairs. For any pair of variables (X,Y), the constraint ensures X≠Y.

The problem setup is illustrated with a grid that defines the relationships between the variables. The equations and constraints are shown alongside the grid for clarity. After solving the puzzle, the solution grid displays the values assigned to each variable, ensuring that all equations and uniqueness rules are satisfied.

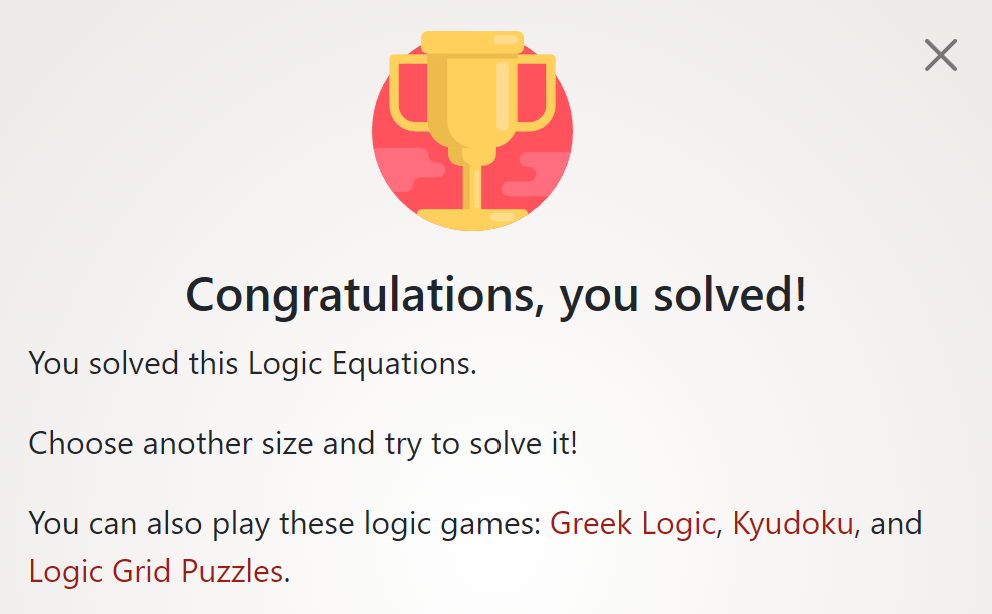
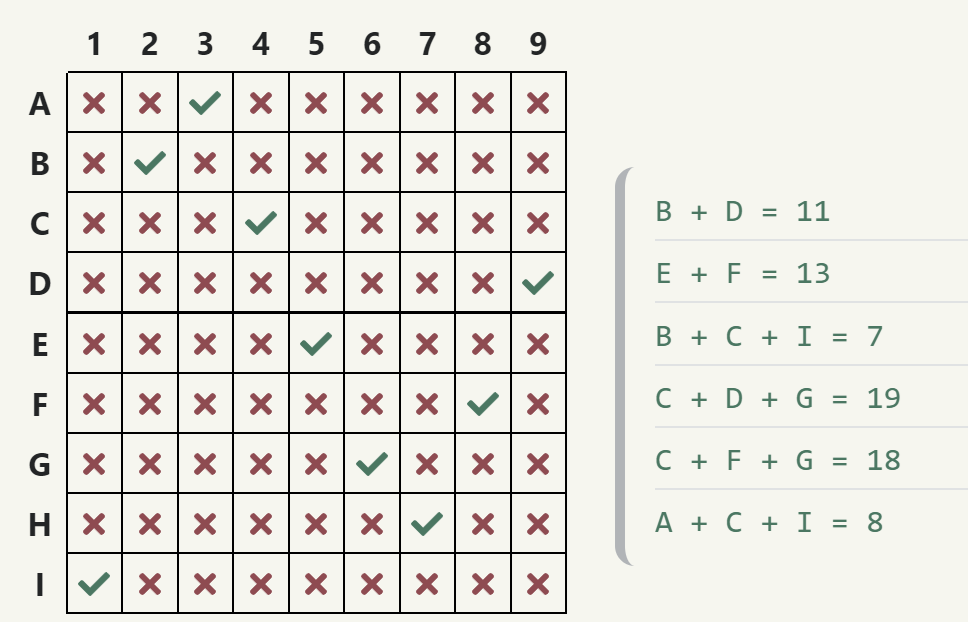


#### **Results**



The performance comparison table summarizes the results of three solving algorithms: Backtracking, Chronological Backtracking, and Hill-Climbing, under varying preprocessing conditions. The key findings from the comparison are as follows:

1. Chronological Backtracking  
   This method consistently outperformed both basic backtracking and hill-climbing when no preprocessing was applied. It proved to be more efficient at exploring the search space systematically.
2. Arc Consistency Preprocessing  
   Applying arc consistency preprocessing significantly reduced the runtime for both chronological backtracking and hill-climbing. Basic backtracking, however, continued to fail to find a solution within the time limit.
3. Path Consistency Preprocessing  
   Path consistency achieved the best overall results, enabling both chronological backtracking and hill-climbing to solve the problem in negligible time. Hill-climbing, in particular, showed substantial improvements with this preprocessing method.



This example demonstrates the effectiveness of the CSP class in solving logic puzzles by defining variables, domains, and constraints clearly. The combination of efficient solving methods and preprocessing techniques highlights the flexibility and scalability of the CSP framework for tackling complex problems. The final solution and performance results underline the advantages of preprocessing in reducing computational overhead and improving solver performance.