William Stewart

1927721

Professor Eick

Task 2 Report

Strategies and Overview)

To train model the CSP I used integer values A-M, a domain form {1-120}, and constrictions C1-C17. And to solve it, I used backtracking, and made it efficient with two ideals:

1. Static variable elimination (pre-analysis):

From C1, C3, C4 I remove A, D, and F from the search and compute them deterministically from B, C, and E respectively.

- $A = B^2 - C^2$
- $D = 4 \cdot C^2 - 3 \cdot B^2$ (derived by substituting A into C3)
- $F = ((B - C)^2 + 396) / (E \cdot B)$ (must be integer and 1..120)

This shrinks the search tree—fewer guessed variables—and preserves correctness.

2. Early pruning (lightweight propagation / forward -checking):

As soon as partial values exist, I check easy consequences (divisibility, ranges, perfect squares, inequality feasibility). If a partial assignment cannot possibly extend to a full solution, I prune immediately. This slashes the number of value attempts (nva). I stop at the first solution, record the solution and nva, and write a CSV.

Psuedocode)

```
INPUT: problem ∈ {A,B,C}; domain D = {1..120}
```

```
BUILD:
  if A: base_vars = [B, C, E]
  if B: base_vars = [B, C, E, G, H, I, J]
  if C: base_vars = [B, C, E, G, H, I, J, K, L, M]

FUNCTION derive_A(B, C): return B^2 - C^2
FUNCTION derive_D(B, C): return 4*C^2 - 3*B^2
FUNCTION derive_F(B, C, E):
  num = (B - C)^2 + 396
  den = E * B
  if den == 0 or num % den != 0: return None
  F = num / den
  if 1 ≤ F ≤ 120: return F else None

FUNCTION feasible_partial(assign, problem):
  // Fast checks with available variables only
  if B,C known:
    A = derive_A(B,C); D = derive_D(B,C)
    if A or D out of domain: return False
  if B,C,E known:
    F = derive_F(B,C,E); if F is None: return False
    if C + E ≤ B: return False
  if C,I known:  // C8 divisibility
    if (C+I)^2 not divisible by (I+3): return False
    if also B,E known and (C+I)^2 ≠ B*E*(I+3): return False
  if G,I known:  // C6 perfect square test
    if (G+I)^3 - 4 not a perfect square: return False
  // C9, C10 feasibility (bounds) and, for B/C, J-interval feasibility from C11/C12
  // For C: check feasibility/equalities for C13..C17 when inputs present
  return True

FUNCTION check_full_A(assign):
  compute A,D,F from B,C,E; verify C1..C5 strictly
  if ok: store A,D,F into assign; return True else False

FUNCTION check_full_B(assign):
  if not check_full_A(assign): return False
  verify C6..C12 with current values; return True/False

FUNCTION check_full_C(assign):
  if not check_full_B(assign): return False
  verify C13..C17; return True/False

BACKTRACK(idx, assign):
  if idx == len(base_vars):
    return check_full_{problem}(copy(assign)) ? assign : None
  var = base_vars[idx]
  for val in D:
    nva++
    assign[var] = val
    if feasible_partial(assign, problem):
      sol = BACKTRACK(idx+1, assign)
      if sol != None: return sol
    remove var from assign
  return None


MAIN:
  build base_vars for problem
  solution = BACKTRACK(0, {})
  write CSV(solution, nva)
  output solution or "no solution", and nva
```

Explanation)

It starts off by searching over the base variables and reconstructs the eliminated variables (A, D, and F) algebraically. This preserves the solution set and reduces branching. After it begins the backtracking segment, where the solver assigns base variables one by one from 1 - 120. After each assignment, it runs *feasible_partial* which performs quick checks that don't require full information: range checks, divisibility, perfect-square conditions, etc. Lastly, it does a leaf check, that once the base variables are assigned, computes A, D, and F and verifies all the constraints for the chosen problem. And checks if they all pass, if so, the solution is found and the solver stops.

Runtime Reduction)

There were a couple of strategies I implemented to lower the nva count,

1. Variable elimination (A, D, F): this removes the selected variables from the search entirely and instead are computed once per leaf, rather than being guessed and this put a huge dent in reduction to the ova count.

2. Early arithmetic filters:

   - C4 makes F an exact fraction; if the numerator isn't divisible by E·B or $F \notin \{1..120\}$, prune early.
   - C8 enforces (C+I)^2 divisible by (I+3), and with B,E known enforces equality; eliminates most (C,I).
   - C6 requires (G+I)^3 − 4 to be a perfect square; kills most (G,I) pairs quickly.

3. Inequality feasibility:

For C9, C10, and the J window from C11/C12, it checks whether there exists any value between 1 - 120 that could still work. If not, pruning happens.

4. Forcing a stop after the first solution: The program is forced to stop once the first solution is found, stoping any extra searching to happen beyond the first valid assignment.

Combined, these drop the count of nva in a dramatic fashion compared to just regular backtracking.

Mathematical Pre-Analysis)

1. C1 + C3, Eliminates A and D:

   $A = B^2 - C^2$. Substitute into C3.

   $D = B^2 - 4A = B^2 - 4(B^2 - C^2) = 4C^2 - 3B^2$.

   Now A and D are functions of B and C; the are forced to fall in between 1 - 120, giving early range tests.

2. C4 ⇒ eliminate F and add divisibility:

   $(B - C)^2 = E \cdot F \cdot B - 396 \rightarrow E \cdot F \cdot B = (B - C)^2 + 396 \rightarrow$

   $F = ((B - C)^2 + 396)/(E \cdot B)$ must be an integer in 1..120.

   That's both a divisibility filter and a range filter, used immediately when B, C, E are set.

3. C6 perfect-square condition:

   $(G + I)^3 - 4$ must be a perfect square. Pre-checking this keeps us from touching H or A until a (G,I) pair is plausible.

4. C8 factor/ratio structure:

   $(C + I)^2 = B \cdot E \cdot (I + 3)$. If B,E unknown, I still enforce divisibility by (I+3). When B,E are known, it's a strict equality—acts like a factorization gate.

5. Bound checks from inequalities:

   C9: $G + I < E + 3 \Rightarrow$ even with partials, you can prove impossibility if min/max don't fit 1..120. C10: $D + H > 180 \Rightarrow$ with $D \in [1..120]$, this implies lower bounds on H (or D) that must be feasible.

All of these are computed before the program starts to commit to deeper choices, which lowers the number of assignments by a ton.

Hierarchical Structure)

This program uses the structure by implementing shared constraints, including all of A's constraints into Problem B; Problem C contains both all of A's and B's Constants inside. In addition, there is a reuse of code in that there is 3 full checks implemented: check_full_A, check_full_B, check_full_C. B calls A's checker internally; C calls B's (and hence A's). This mirrors the hierarchy and avoids duplicating logic. Lastly, there is consistent elimination in that the same A/D/F elimination is used in all three problems; higher problems just add their extra constraints on top.

My solver does not precompute or reuse solutions from Problem A when solving Problem B, nor reuse Problem B for Problem C. Instead, each problem is solved independently: when solving B, the program enforces all of A's constraints internally, and when solving C it enforces both A's and B's constraints. Because of this design, the number of variable assignments (nva) is reported separately for each run.

This means the value of nva shown for Problem B already includes the cost of checking all of A's constraints, and the value for Problem C already includes the costs of A and B. Therefore, it would be incorrect to add nva_A + nva_B + nva_C. Each reported nva is self-contained.

Program Reusability)

My program was designed with separation of concerns in mind. At its core is a small, problem-agnostic backtracking engine that handles the search process and counts variable assignments (nva). All of the actual CSP logic is isolated in small, pure functions such as check_full_A/B/C and feasible_partial, which can be swapped out for new sets of constraints without changing the engine. The solver also uses pluggable base variables: for each problem the list of variables to assign is built separately, allowing me to change which variables are searched and which are eliminated algebraically without touching the backtracking logic. In addition, the reusable filters I implemented—like divisibility checks, perfect-square tests, and feasibility bounds—are general patterns that can be adapted to other arithmetic CSPs. Finally, the program outputs solutions through a CSV writer and exposes a command-line interface, so any similar CSP with integer variables in the same domain can be solved simply by providing a new set of constraints and, if needed, new elimination formulas.

AI Credit)

I used ChatGPT to help with the implementation of the different pre-analysis methods, to reduce the nva count as the program runs. I also used it to help implement the initial version of the program to build off of, and lastly I had it help me ensure that the report contains everything needed by the rubric.