CSP Solver - README

Overview

This is a hierarchical Constraint Satisfaction Problem (CSP) solver for three nested problems involving 13 integer variables {A, B, C, D, E, F, G, H, I, J, K, L, M} with domains in {1, ..., 120}.

Requirements

- Python 3.6 or higher
- No external libraries required (uses only Python standard library)

Installation

No installation needed. Simply download `csp_solver.py` to your working directory.

```
# Verify Python version
python --version  # or python3 --version
```

Usage

Basic Usage

Run the solver from command line:

```
# Solve Problem A (6 variables, 5 constraints)
python csp_solver.py A

# Solve Problem B (10 variables, 12 constraints)
python csp_solver.py B

# Solve Problem C (13 variables, 17 constraints)
python csp_solver.py C
```

Default Behavior

If no argument is provided, the solver defaults to Problem A:

```
python csp_solver.py
```

Using as a Module

You can also import and use the solver in your own Python code:

```
from csp_solver import solve_csp, CSPSolver

# Solve Problem A
solution, nva = solve_csp('A')

if solution:
    print(f"Solution found: {solution}")
    print(f"Variable assignments: {nva}")
else:
    print("No solution exists")

# Or use the class directly
solver = CSPSolver()
found = solver.solve_problem_b()
if found:
    print(f"Solution: {solver.solution}")
    print(f"NVA: {solver.nva}")
```

Output Format

Console Output

The solver prints:

1. Solution variables and their values
2. Constraint verification (✓ for satisfied, ✗ for violated)
3. Total number of variable assignments (nva)

Example:

```
==================================================
Problem A Results
==================================================
Solution found:
  A = 60
  B = 16
  C = 14
  D = 16
  E = 5
  F = 5

Verification:
  C1: ✓
  C2: ✓
  C3: ✓
  C4: ✓
  C5: ✓
  Overall: ✓ VALID

Number of variable assignments (nva): 154
==================================================
```

CSV Output

A solution is automatically saved to `csp_solution.csv` with format:

```
Variable,Value
A,60
B,16
C,14
D,16
E,5
F,5
```

Problem Definitions

Problem A (C1-C5)

Variables: A, B, C, D, E, F

- C1: $A = B^2 - C^2$
- C2: $C + E > B$
- C3: $D = B^2 - 4*A$
- C4: $(B-C)^2 = E*F*B - 396$
- C5: $C + D + E + F < 125$

Problem B (C1-C12)

Variables: A, B, C, D, E, F, G, H, I, J

- C1-C5: Same as Problem A
- C6: $(G+I)^3 - 4 = (H-A)^2$
- C7: $C*E*F + 40 = (H-F-I)*(I+G)$
- C8: $(C+I)^2 = B*E*(I+3)$
- C9: $G + I < E + 3$
- C10: $D + H > 180$
- C11: $J < D + E + F$
- C12: $J > H + E + F + G + I$

Problem C (C1-C17)

Variables: A, B, C, D, E, F, G, H, I, J, K, L, M

- C1-C12: Same as Problem B
- C13: $K*L*M = B*(K+5)$
- C14: $F^3 = K^2*(L-29) + 25$
- C15: $H*M^2 = L*G - 3$
- C16: $J + M = (L-15)*(E+G)$
- C17: $K^3 = (J-4)*(L-20)$

Algorithm Details

Search Strategy

1. **Algebraic Elimination**: 6 variables (A, D, F, H, L, M) are computed directly from other variables
2. **Constraint Propagation**: Tight bounds reduce search domains significantly
3. **Hierarchical Search**: Problems B and C build upon valid solutions from lower levels
4. **Early Termination**: Stops at first valid solution

Optimization Techniques

- Divisibility filtering (C4, C13, C17)
- Perfect square testing (C6)
- Tight domain bounds from inequality constraints
- Early constraint checking to prune invalid branches

Variable Assignment Counter (nva)

The `nva` counter tracks every time a variable is:

- Initially assigned a value
- Changed to a different value during search

This includes:

- Loop iterations (B, C, E, G, I, J, K)
- Computed variables (A, D, F, H, L, M)

The counter properly accumulates across hierarchical levels.

Troubleshooting

No Solution Found

If the solver reports "No solution exists":

- Verify constraint definitions are correct
- Check that variable domains are [1, 120]
- Try running with debug prints (modify source code)

Slow Performance

Expected nva values:

- Problem A: < 10,000
- Problem B: < 100,000
- Problem C: < 1,000,000

If much higher:

- Ensure Python 3.6+ is being used
- Check for inadvertent nested loops
- Verify constraint filters are working

Import Errors

If you get import errors when using as a module:

```
# Make sure csp_solver.py is in your Python path
import sys
sys.path.append('/path/to/csp_solver_directory')
from csp_solver import solve_csp
```

CSV File Issues

If CSV file is not generated:

- Check write permissions in current directory
- Ensure solution was found (file only created on success)
- Manually specify path: modify `save_to_csv()` call

Code Structure

```
csp_solver.py
├── CSPSolver class
│   ├── __init__(): Initialize solver and nva counter
│   ├── solve_problem_a(): Solve Problem A (C1-C5)
│   ├── solve_problem_b(): Solve Problem B (C1-C12)
│   ├── solve_problem_c(): Solve Problem C (C1-C17)
│   ├── verify_solution(): Verify all constraints for given problem
│   ├── save_to_csv(): Export solution to CSV file
│   └── print_results(): Display results and verification
└── solve_csp(): Main interface function
```

Extending the Solver

Adding New Problems

To add Problem D with additional variables and constraints:

```
def solve_problem_d(self):
    """Solve Problem D with constraints C1-C20"""
```

```
    # Copy solve_problem_c structure
    for each valid (A-M) from Problem C:
        for N in range(1, 121):
            self.nva += 1
            # Add constraint checks for C18-C20
            # Compute dependent variables
            # Return solution if valid
    return False
```

Then update:

1. `verify_solution()` to include Problem D constraints
2. `solve_csp()` to handle problem='D'

Modifying Constraints

To change a constraint (example: modify C4):

1. Update the check in `solve_problem_a()`:

```
# Old: target = (B - C)**2 + 396
# New: target = (B - C)**2 + 500
```

2. Update verification in `verify_solution()`:

```
# Old: 'C4': (B - C)**2 == E*F*B - 396,
# New: 'C4': (B - C)**2 == E*F*B - 500,
```

Changing Variable Domains

To change from [1, 120] to [1, 200]:

```
# At top of file, add:
MIN_VALUE = 1
MAX_VALUE = 200

# Then replace all range(1, 121) with:
range(MIN_VALUE, MAX_VALUE + 1)
```

Performance Tips

For Faster Execution

1. Use PyPy instead of CPython: `pypy3 csp_solver.py A`
2. Run on faster hardware (CPU speed matters for this algorithm)
3. Profile with: `python -m cProfile csp_solver.py A`

For Finding Multiple Solutions

Modify the solver to not return immediately:

```
solutions = []
# Inside innermost loop, instead of return True:
solutions.append({...})
if len(solutions) >= max_solutions:
    return True
```

For Debugging

Add debug prints:

```
# At key decision points
if DEBUG:
    print(f"Testing B={B}, C={C}, A={A}, D={D}")
```

Testing

Unit Tests

Create a test file `test_csp_solver.py`:

```
from csp_solver import CSPSolver

def test_problem_a():
    solver = CSPSolver()
    found = solver.solve_problem_a()
    assert found, "Should find a solution for Problem A"

    # Verify solution
    sol = solver.solution
    A, B, C, D, E, F = sol['A'], sol['B'], sol['C'], sol['D'], sol['E'],
sol['F']

    assert A == B**2 - C**2, "C1 violated"
    assert C + E > B, "C2 violated"
    assert D == B**2 - 4*A, "C3 violated"
    assert (B - C)**2 == E*F*B - 396, "C4 violated"
    assert C + D + E + F < 125, "C5 violated"

    print("Problem A tests passed!")

def test_problem_b():
    solver = CSPSolver()
    found = solver.solve_problem_b()
    assert found, "Should find a solution for Problem B"
    all_valid, checks = solver.verify_solution(solver.solution, 'B')
    assert all_valid, "Solution should satisfy all constraints"
    print("Problem B tests passed!")

def test_problem_c():
    solver = CSPSolver()
    found = solver.solve_problem_c()
```

```
    assert found, "Should find a solution for Problem C"
    all_valid, checks = solver.verify_solution(solver.solution, 'C')
    assert all_valid, "Solution should satisfy all constraints"
    print("Problem C tests passed!")

if __name__ == "__main__":
    test_problem_a()
    test_problem_b()
    test_problem_c()
    print("\nAll tests passed!")
```

Run with: `python test_csp_solver.py`

Examples

Example 1: Basic Usage

```
$ python csp_solver.py A

CSP Solver - Problem A
Solving constraint satisfaction problem...

==================================================
Problem A Results
==================================================
Solution found:
  A = 60
  B = 16
  C = 14
  D = 16
  E = 5
  F = 5

Verification:
  C1: ✓
  C2: ✓
  C3: ✓
  C4: ✓
  C5: ✓
  Overall: ✓ VALID

Number of variable assignments (nva): 154
==================================================

Solution saved to csp_solution.csv

Final solution:
{'A': 60, 'B': 16, 'C': 14, 'D': 16, 'E': 5, 'F': 5}
Total variable assignments: 154
```

Example 2: Programmatic Usage

```
#!/usr/bin/env python3
from csp_solver import solve_csp

# Solve all three problems
for problem in ['A', 'B', 'C']:
    print(f"\n{'='*60}")
    print(f"Solving Problem {problem}")
    print('='*60)

    solution, nva = solve_csp(problem)

    if solution:
        print(f"✓ Found solution with {nva} variable assignments")
        print(f"Variables: {list(solution.keys())}")
    else:
        print(f"✗ No solution found after {nva} variable assignments")
```

Example 3: Custom Analysis

```
from csp_solver import CSPSolver

# Solve and analyze
solver = CSPSolver()
found = solver.solve_problem_c()

if found:
    sol = solver.solution
    print(f"Solution efficiency: {solver.nva} assignments")

    # Analyze solution properties
    print(f"\nSolution statistics:")
    print(f"  Sum of all variables: {sum(sol.values())}")
    print(f"  Max variable value: {max(sol.values())}")
    print(f"  Min variable value: {min(sol.values())}")

    # Check specific relationships
    A, B, C = sol['A'], sol['B'], sol['C']
    print(f"\nKey relationships:")
    print(f"  B² - C² = {B**2 - C**2} (should equal A={A})")
    print(f"  -3B² + 4C² = {-3*B**2 + 4*C**2} (should equal D={sol['D']})")
```

FAQ

Q: Why does the solver stop at the first solution?
A: As specified in the requirements, the solver terminates upon finding the first valid solution to minimize nva.

Q: Can I find all solutions?
A: Yes, modify the code to collect solutions instead of returning immediately. Be aware this will increase nva significantly.

Q: What if no solution exists?
A: The solver will exhaust the search space and report "No solution exists" along with the total nva count.

Q: How is nva counted for hierarchical problems?
A: For Problem B, nva includes all assignments made while searching through Problem A's space plus assignments for G, H, I, J. Similarly for Problem C.

Q: Can I use this for non-integer domains?
A: The current implementation is optimized for integers. For floats, you'd need to modify the divisibility checks and perfect square tests.

Q: Is the solution deterministic?
A: Yes, the search order is deterministic, so running multiple times produces the same solution and nva value.

Support

For issues or questions:

1. Check this README thoroughly
2. Verify your Python version (3.6+)
3. Review the constraint definitions
4. Check the source code comments
5. Test with simpler problems first

License

This code is provided for educational purposes as part of a constraint satisfaction problem assignment.

Authors

- Primary implementation with AI assistance (Claude AI by Anthropic)
- Mathematical analysis and optimization strategies (Juan Chavez with assistance of Claude AI)
- Testing and verification by Juan Chavez

Version History

- **v1.0** (2025-10-04): Initial release with Problems A, B, C
    - Hierarchical solver implementation
    - Algebraic variable elimination
    - Constraint-based pruning

- CSV export functionality
- Comprehensive verification
