Week_4_Documentation

Week 4

ChatGPT was used to clean up the code, make it more readable by adding comments, and by presenting the results in a clear fashion.

Problem 1: Game of 21

Two players take turns removing 1, 2, or 3 objects from a pile starting with 21 objects. The player who takes the last object wins.

We want to know: Can the first player guarantee a win, and if so, what's the winning strategy?

We use Z3 to figure out which positions are winning and which are losing by setting up logical constraints.

Overall strategy

1. We work backwards by starting from the end of the game (i.e., when we have 0 objects to remove).
2. A winning position is one such that there exists a move from this position that forces the opponent into a losing position.
3. A position is losing if, no matter what move you make, all moves lead to a winning position for the opponent.
4. We let Z3 solve these constraints to classify all positions from 0 to 21.

Explanation of the code

refer to file Week4_problem1_Game_of_21.py alongside this explanation.

Step 1:

`Winning = [Bool(f'W_{i}') for i in range(22)]`

We create 22 boolean variables (one for each position 0 through 21):

- `Winning[i] = True` means "position i is a winning position for whoever's turn it is"
- `Winning[i] = False` means "position i is a losing position"

Step 2:

```
s.add(Not(Winning[0]))
```

When there are 0 objects left, the game is already over—the previous player took the last object and won. So if it's "your turn" at position 0, you've already lost (there's nothing to take). Therefore, **position 0 is a losing position**.

Step 3:

For every position i, we encode the following rule - "Position i is winning iff there exists at least one move that puts your opponent in a losing position."

```
for i in range(1, 22):
  possible_moves = []

  if i >= 1:
     possible_moves.append(Not(Winning[i - 1]))
  if i >= 2:
     possible_moves.append(Not(Winning[i - 2]))
  if i >= 3:
     possible_moves.append(Not(Winning[i - 3]))

  s.add(Winning[i] == Or(possible_moves))
```

So consider some position i. From position i, you can remove

- 1 object and end up at position (i-1)
- 2 objects and end up at position (i-2)
- 3 objects and end up at position (i-3).

We want to move towards a losing position (for we want the opponent to lose!).

- `Not(Winning[i-1])` means "position i-1 is losing"
- `Or(possible_moves)` means "at least one of these moves leads to a losing position"

Step4:

```
if s.check() == sat:
  m = s.model()
```

Z3 take in all these constraints and finds a solution. This will tell us which positions are winning and which are loosing.

Step5:

```
first_player_wins = m.evaluate(Winning[21])
```

We check position 21. If it is a winning position, then that means the first player can guarantee/force a win.

```
for move in [1, 2, 3]:
  if not m.evaluate(Winning[21 - move]):
      print(f"First player should remove {move} object(s)")
```

To find the optimal first move, we check which moves from position 21 lead to a losing position. For example, if we start at position 21 and remove one object then the opponent is at position 20. If position 20 is a losing position, then removing 1 object is optimal.

Solving these constraints in Z3 reveal that the losing positions are multiples of 4.

- **Losing positions:** 0, 4, 8, 12, 16, 20 (multiples of 4)
- **Winning positions:** The remaining positions.

Thus the winning strategy is to always leave the opponent at a multiple of 4.

Problem 2: Non-linear constraint solving

We want to solve a system of equations by finding *all* integer solutions. We shall use an iterative approach. Here's the high-level idea -

1. Set up x and y as integer variables and add contraints to Z3.
2. Ask Z3 to find a solution.
3. Once a solution is found, block it and search again.
4. Repeat this procedure until no more solutions exist.

This process gives us all the integer solutions.

Explanation of the code

refer to file Week4_problem2_non_lin_constraint_solving.py alongside this explanation.

Step 1: This is just setting up the integer variables and adding the constraints.

```
# Create integer variables
  x = Int('x')
  y = Int('y')

  # Create solver
  s = Solver()

  # Add constraints
  s.add(x * x + y * y == 25)  # x² + y² = 25
  s.add(x + y == 7)           # x + y = 7
  s.add(x > 0)                # x > 0
  s.add(y > 0)                # y > 0
```

Step 2: Finding the first solution

```
  while s.check() == sat:
     m = s.model()
     x_val = m[x].as_long()
     y_val = m[y].as_long()
```

This will output a solution, something like x=3 and y=4.

Step 3:

```
     # Block this solution to find others
     s.add(Or(x != x_val, y != y_val))
```

This blocks the solution we found and finds other solutions.

Step 4: The while loop continues -
find solution - block solution - search again.

Step 5:

We substitute the solutions to check whether they really are correct.

`print(f"x² + y² = {x_val}² + {y_val}² = {x_val**2} + {y_val**2} = {x_val**2 + y_val**2} ✓")`

Problem 3: Invariant Synthesis

We have been given a loop and we want to use Z3 to sythesize a loop invariant for this loop.

Here’s the high-level idea -

1. Setup the coefficients a,b,c and add the constraints. The constraints would be
   a. Initialization - constraint must hold initially
   b. Preservation - invariant must be preserved by the loop
   c. inavariant must be non-trivial (0=0 is not very helpful).
2. Let Z3 solve for a,b, and c.
3. verify the invariant generated.

Explanation of the code

refer to file Week4_problem3_invariants.py alongside this explanation.

Step 1:

Setting up the unknowns and states of the loop -

```
# Coefficients
a = Int('a')
b = Int('b')
c = Int('c')

# Variables
i = Int('i')
```

```
    sum_var = Int('sum')
    i_next = Int('i_next')
    sum_next = Int('sum_next')
    n = Int('n')

    s = Solver()
```

Step 2:

Adding the initiation constraint

s.add(c <= 0)

Step 3:

Adding the Preservation constraint.

```
    s.add(i_next == i + 1)
    s.add(sum_next == sum_var + i)
```

This defines what the loop body does.

```
    for test_i in range(5):
        test_sum = sum(range(test_i))  # sum = 0+1+2+...+(i-1)
        test_i_next = test_i + 1
        test_sum_next = test_sum + test_i

        # If we're in the loop (i < n), say n = 10
        test_n = 10
        if test_i < test_n:
            # Invariant before
            inv_before = a * test_i + b * test_sum + c <= 0
            # Invariant after
            inv_after = a * test_i_next + b * test_sum_next + c <= 0
            # Add implication
            s.add(Implies(inv_before, inv_after))
```

Here we are using test cases to guide synthesis. The `Implies(inv_before, inv_after)` means: "If the invariant held before this iteration, it must hold after."

It’s difficult for Z3 to handle forall quantifiers, so instead we check whether an invariant works for the first 5 or 10 iterations, if it does, then its likely to be a good candidate.

Step 4:

This is just saying that the invariant must not be trivial -

```
# no non-trivial invariant
s.add(Or(a != 0, b != 0))
print("\n[3] Non-triviality: At least one coefficient must be non-zero")
```

Step 5:

Here we bound the search space. For we don’t want just any solution, we want a practical solution.

```
# Bound the coefficients for practical solutions
s.add(a >= -10, a <= 10)
s.add(b >= -10, b <= 10)
s.add(c >= -100, c <= 100)
```

Step 6:

Z3 searches

```
# Find solution
if s.check() == sat:
    m = s.model()
    a_val = m[a].as_long()
    b_val = m[b].as_long()
    c_val = m[c].as_long()
```

Step 7:

Here Z3 verifies the invariants -

```
def verify_invariant(a, b, c, n_vals=[5, 10, 20], verbose=True):
    """
    Verify that the invariant a*i + b*sum + c <= 0 holds throughout loop exec
ution
    """
```