Logic and Constraint Programming Report
Color Maze Puzzle Solver

Faculdade de Engenharia da Universidade do Porto - Programação em Lógica com Restrições 2024

José Ramos
up202005460@edu.fe.up.pt
Contribution: 50%

Guilherme Pereira
up202007375@edu.fe.up.pt
Contribution: 50%

***Abstract*—Development of a Color Maze puzzle solver using constraint programming.**

***Index Terms*—PLR, Prolog, Python, Constraint Programming, Color Maze, Puzzle**

I. Introduction

This report is prepared in the context of Logic and Constraint Programming, focusing on a specific problem: finding a path from the bottom left to the top right of a grid that passes through an equal number of squares of each (non-white) color.

The core challenge of this problem involves navigating a grid with multiple colored squares, ensuring that the path adheres to the following constraints:

1) Start at the bottom-left corner of the grid.
2) End at the top-right corner.
3) Pass through an equal number of squares of each non-white color.

Below is an example of a solved puzzle:

[Insert Image of Solved Grid Here]

Several factors influenced the selection of this problem for study:

1) **Scalability**: The problem is inherently scalable in terms of grid size and the number of colors used. This flexibility allows for the creation of puzzles of varying difficulty levels, accommodating different skill levels and computational complexities.
2) **Complexity and Challenge**: The problem presents a significant challenge that is engaging and thought-provoking. It requires a combination of logical reasoning and strategic planning, making it an excellent candidate for demonstrating the principles of constraint programming.
3) **Ease of Understanding**: Despite its complexity, the problem is easy to understand and visualize. The clear and straightforward rules make it accessible to a broad audience, from beginners in logic programming to advanced practitioners.

II. Solver

A. SICSTUS Solver

This section details the data structures and logic for representing and navigating a maze within the context of the solver developed in SICStus Prolog.

1) Maze Representation: We represent the maze as a list $Maze = [color1, color2, ...]$, where each number corresponds to a unique color.

2) Maze Properties: In order to simplify access to some properties of the Maze we defined the following variables

- Size: represents the length of the Maze list;
- N: the *side* of the maze;
- Start: defines the index of the starting node of the maze path;
- Finish: defines the index of the last node of the maze path;

```
% Calculate Maze Size and Side Lenght
length(Maze, Size),
N is round(sqrt(Size)),

% Define the start and finish nodes
Start is Size - N + 1,
Finish is N,
```

3) Path Constraints: **Path**: A subcircuit defined using the *subcircuit/2* constraint, ensuring it's a valid sequence of positions within the maze. This Path will have the domain from 1 to the Size of the Maze.

4) Neighbors Function: This code snippet defines a predicate neighbors(Position, Neighbor, N).

```
% Check if two nodes are neighbors
neighbor(Position, Neighbor, N) :-
  Diff #= abs((Position-1) mod N -
      (Neighbor-1) mod N),
  Neighbor #= Position - N #\/ % Up
  (Neighbor #= Position + N #/\ Position mod N
      #> 1) #\/ % Down
  (Neighbor #= Position + 1 #/\ Diff #= 1) #\/
      % Right
  (Neighbor #= Position - 1 #/\ Diff #= 1) #\/
      % Left
  Neighbor #= Position. % Self
```

It takes a Position within the maze and the Maze side (N) as input and adds constraints related to the possible Neighbor of the related position. The ***Position mod N > 1*** constraint is a minor optimization add that forbids considering the neighbor of the row below when Position is in the leftmost or rightmost column, as choosing that neighbor will always lead to impossible pathways. The ***Diff = 1*** condition prevents scenarios where there are no *Right* or *Left* neighbors due to a change in a row of the Maze. The *Self* constraint is present in the particularities of the *subcircuit/2* constraint, specifically that Position isn't part of the Path.

5) Reduced Maze: We created a NewMaze variable that is a copy of the original Maze, with one small difference, the positions that don't belong to the Path had it's value altered to 0 in order to be ignored when counting the colors. This is because path finding process only considers positions, not colors.

```
% Removes unuseful colors from the maze
filter_maze(_, [], _, _) :- !.
filter_maze(Path, [H|T], Maze, NewMaze) :-
  element(Position, Path, H),
  element(Position, Maze, Color),
  element(Position, NewMaze, NewColor),
  Position #= H #<=> Bool, !,
  if_then_else(Bool, 0, Color, NewColor),
  filter_maze(Path, T, Maze, NewMaze).
```

6) Color Validation: We use the *count/4* predicate to ensure all colors appear an equal number of times in the original Maze. This validates the path structure.

```
% Count the number of colors in the maze
% and ensures they are the same amount (K)
count_colors(_,0,_) :- !.
count_colors(Maze,NumColors, K) :-
  count(NumColors, Maze, #=, K),
  NumColors1 is NumColors - 1,
  count_colors(Maze,NumColors1,K).
```

7) Pathfinding: We employ a search algorithm (not shown here) to find solutions within the defined path constraints. We ensure the search yields only one solution, indicating a single valid path through the maze.

```
length(Solutions, 1), % Only one solution
findall(Path, labeling([ffc, bisect, down],
    Path), Solutions).
```

B. DOCPLEX Solver

```
from docplex.cp.model import CpoModel
import time
def solve_maze():
  mdl = CpoModel('Maze Solver')
  start_time = time.time()
```

The script imports the CpoModel class from the docplex.cp.model module, which is used to create a constraint programming model. It also imports the time module to measure the execution time. **CpoModel('Maze Solver')** creates a new constraint programming model named "Maze Solver", and start_time = time.time() is the start of the timer that will later be used to measure execution time.

```
max_number = max(maze)
size = len(maze)
```

These lines calculate the maximum number of colors in the maze and the size of the maze.

```
finish = round(size ** 0.5) - 1
start = size - finish - 1
```

These lines calculate the start and finish nodes of the maze.

```
path = mdl.integer_var_list(size, 0, size -
    1, "path")
```

This line creates a list of integer variables that represent the path through the maze.

```
mdl.add(path[finish] == start)
mdl.add(mdl.sub_circuit(path))
```

These lines add constraints to the model. The first constraint ensures that the path starts at the start node and the second constraint ensures that the path forms a subcircuit.

```
for i in range(size):
  if i != finish:
    position = path[i]
    N = finish + 1
    diff = (position) % N
    mdl.add((position == i - N) | # Up
        ((position == i + N) & (position % N >
            1)) | # Down
        ((position == i + 1) & (diff > 0)) | #
            Right
        ((position == i - 1) & (diff < N - 1))
            | # Left
        (position == i)) # Self
```

This loop adds constraints to the model for each node in the maze. The constraints ensure that each node in the path is connected to its neighbors.

```
new_maze = mdl.integer_var_list(size, 0, size
    - 1, "new_path")
```

This line creates a new list of integer variables that represent the new maze after the path has been found.

```
for i in range(size):
  position = path[i]
  mdl.add(mdl.if_then(position == i,
      new_maze[i] == 0))
  mdl.add(mdl.if_then(position != i,
      new_maze[i] == maze[i]))
```

This loop adds constraints to the model for each node in the new maze. The constraints ensure that each node in the new maze is either 0 (if it's not part of the path) or the same

as the corresponding node in the original maze (if it's part of the path).

```
count_colors =
    mdl.integer_var_list(max_number, 0, size
    - 1, "count_colors")
for i in range(1, max_number+1):
  mdl.add(count_colors[i-1] ==
      mdl.count(new_maze, i))
```

This code creates a list of integer variables representing the count of each color in the new maze and adds constraints to the model to ensure that the counts are correct.

```
for i in range(1, max_number):
  mdl.add(count_colors[i] == count_colors[0])
```

This loop adds constraints to the model to ensure that all colors in the new maze have the same count.

```
solution = mdl.solve()

if solution:
  end_time = time.time()
  print("Execution time: ", end_time -
      start_time, "seconds")
  print([solution.get_value(path[i]) for i in
      range(size)])

  return solution.get_solve_status()
else:
  return "No solution found"
```

mdl.solve() solves the model and returns the solution. If a solution was found, it prints the execution time and the path through the maze, and it returns the solve status. If no solution was found, it returns "No solution found".

C. Solver Results

To test this model we decided to use 7 mazes that vary both in size and complexity. Maze 1 with size 4 and 2 colors. Maze 2 with size 4 and 3 colors. Maze 3 with size 5 and 3 colors. Maze 4 with size 5 and 4 colors. Maze 5 with size 5 and 5 colors. Maze 6 with size 5 and 6 colors. Maze 7 with size 6 and 4 colors.

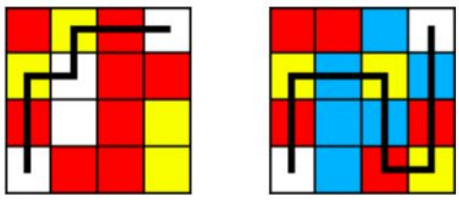

Fig. 1. Maze 1 and Maze2

As we expected the more complex the puzzle the more time it takes to solve, except for Prolog when the path passes in the middle, because of bisect in the labeling, we get much better times than those who do not pass in the middle. Overall the Prolog times aren't bad, but for maze 7 it was so complex that

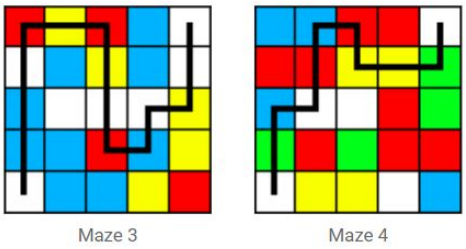

Fig. 2. Maze 3 and Maze 4

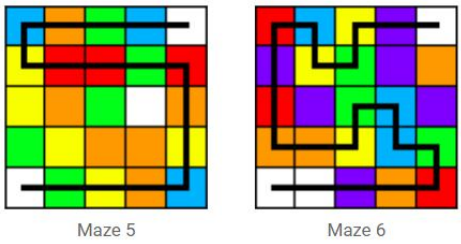

Fig. 3. Maze 5 and Maze 6

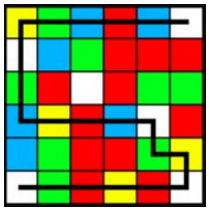

Fig. 4. Maze 7

MAZE	Prolog time	Docplex time
1	0.15	0.09
2	0.07	0.09
3	3.75	0.11
4	14.45	0.11
5	10.78	0.15
6	3.85	0.08
7	?	1.9

TABLE I
MAZE COMPLETION TIMES IN SECONDS

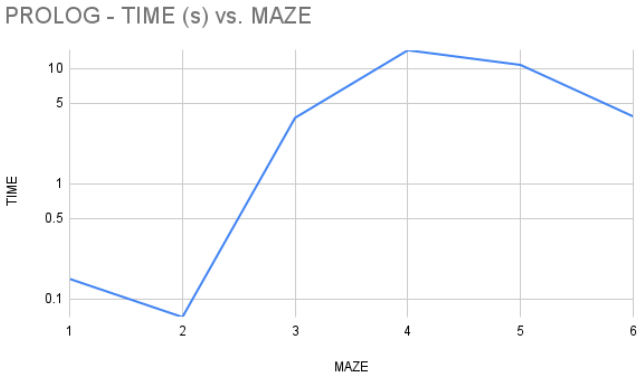

Fig. 5. PROLOG - TIME (s) vs. MAZE

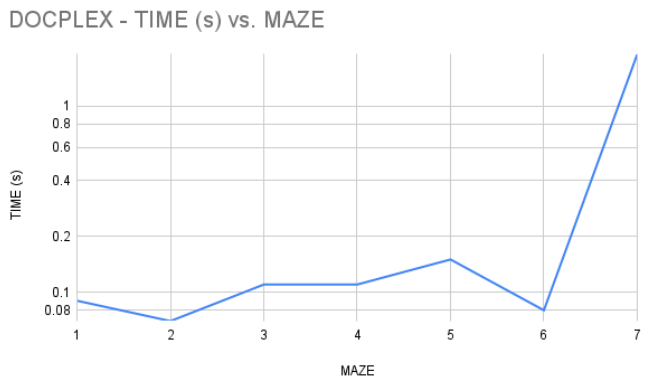

Fig. 6. DOCPLEX - TIME (s) vs. MAZE

the solver would take so much time that we couldn't measure. On the other hand Dockplex times were much better and is obviously the better model at solving the problem, being that it was able to solve maze 7 in 1.9 seconds as opposed to Prolog that couldn't even solve it in a few hours.

III. Generator

The generator is bit simpler than the solver, although its construction is sub-optimal.

We start by asking the size of the side of the maze and the number of colors. After that, just like the solver, we get the start and finish node positions and put them to 0. To prevent the generator to create, puzzles with a lot of zeros, we limit the amount of zeros the maze can have, which has to be ¡ Size of the side + 2, that way our generator will have more quality results. Also, to make the maze to spec, we make sure to have at least one of each color in the maze.

When all of this is done, the puzzle has been generated, but we still need to check if the solution is unique. We do this by running the solver and checking if there is more than one solution. This is where our generator needs a bit of work; by doing this we are forced to use two labeling, which are affecting results.

A. SICSTUS Generator

```
generate_maze(N, NumColors, Maze):-
  reset_timer,!,
```

This is the start of the generate_maze predicate. It takes three arguments: N (the size of the maze), NumColors (the number of colors in the maze), and Maze (the list representing the maze). The reset_timer,! is used to reset the timer for performance measurement.

```
Size is N * N,
length(Maze, Size),
domain(Maze, 0, NumColors),
```

These lines calculate the total size of the maze (N * N), ensure that the Maze list has the correct length, and restrict the domain of the elements in Maze to be between 0 and NumColors.

```
Start is Size - N + 1,
Finish is N,
element(Start, Maze, 0),
element(Finish, Maze, 0),
```

These lines calculate the indices of the start and finish nodes in the maze and ensure that they are both 0.

```
ZeroCount #= N + 2,
count(0, Maze, #<, ZeroCount),
```

These lines calculate the maximum number of zeros allowed in the maze (N + 2) and add a constraint to ensure that the number of zeros in the maze is less than this maximum.

```
generate_count_colors(Maze, NumColors), !,

generate_count_colors(_,0) :- !.
generate_count_colors(Maze,NumColors) :-
  count(NumColors, Maze, #>, 0),
  NumColors1 #= NumColors - 1,
  generate_count_colors(Maze,NumColors1).
```

Generate_count_colors predicate to add constraints related to the count of each color in the maze. It adds a constraint to ensure that the count of NumColors in the maze is greater than 0, decrements NumColors, and calls itself recursively.If NumColors is 0, it does nothing and succeeds immediately.

```
  labeling([], Maze),
  generate_solve_maze(Maze),nl,print_time('Time:
      '),fd_statistics.
```

These lines find a solution to the maze, solve the maze, print the time taken, and print some statistics. The generate_solve_maze is basically the same as having the solve_maze predicate, but it checks if there is more than one solution.

```
length(Solutions, 1),
findall(Path, labeling([ffc, bisect, down],
    Path), Solutions),
member(Path, Solutions),
write_path(Path, NewMaze, Start, Finish).
```

This is the only difference from the solve_maze predicate. Here we limit the numbers of solutions to 1, this way we know that there is one and only one solution.

B. DOCPLEX Generator

```
def create_maze(N, NumColors):
  start_time = time.time()
  mdl = CpoModel()
```

This function create_maze takes two parameters: N (the size of the maze) and NumColors (the number of colors). It starts by recording the current time and creating a new CPO model.

```
Size = N * N
```

```
Maze = mdl.integer_var_list(Size, 0,
    NumColors, "Maze")
```

The Size is calculated as N squared, representing the total number of cells in the maze. Maze is a list of integer variables, each representing a cell in the maze. The value of each cell can range from 0 to NumColors.

```
Start = Size - N
Finish = N - 1
mdl.add(Maze[Start] == 0)
mdl.add(Maze[Finish] == 0)
```

The Start and Finish positions are defined. Constraints are added to the model to ensure that the Start and Finish cells are both 0 (representing the color).

```
mdl.add(mdl.count(Maze, 0) < N + 2)
```

A constraint is added to ensure that there are fewer than N + 2 cells with the color 0 in the maze.

```
for color in range(NumColors, 0, -1):
  mdl.add(mdl.count(Maze, color) >= 1)
```

A loop is used to add a constraint for each color from NumColors to 1, ensuring that each color appears at least once in the maze.

```
found = 0

while True:
  solution: CpoSolveResult =
      mdl.solve(log_output=None)
```

A while loop is used to find solutions to the model. The solve method is called to solve the model, and the result is stored in solution.

```
if solution:
  maze_solution = [solution.get_value(Maze[i])
      for i in range(Size)]
```

If a solution is found, the values of the cells in the solution are stored in maze_solution.

```
found, sol = solve_maze(maze_solution)
```

The solve_maze function is called with maze_solution as the argument. The result is stored in found and sol.

```
mdl.add(mdl.sum([Maze[i] != maze_solution[i]
    for i in range(Size)]) >= 1)
```

A constraint is added to the model to exclude the current solution, ensuring that the next solution found will be different.

```
else:
  break # No more solutions found
```

If no solution is found, the loop is broken and the function ends.

```
if found:
    end_time = time.time()
    print(solution)
    print("Execution time: ", end_time -
        start_time, "seconds")
    found = 0
    print("Maze: ", maze_solution)
    print("Solution: ", sol)
    user_input = input("Do you want to
        continue? (y/n): ")
    if user_input.lower() != 'y':
       break
    start_time = time.time()
```

If a solution is found and it is unique (as indicated by found being 1), the solution and the execution time are printed, and the user is asked whether they want to continue. If the user enters 'n', the loop is broken and the function ends. If the user enters 'y', the time is recorded, and the loop continues to find the next solution.

```
def solve_maze(maze):
...
  if solution:
      initial_solution =
          [solution.get_value(path[i]) for i in
          range(size)]

      # Add a constraint to exclude the initial
          solution
      mdl.add(mdl.sum([path[i] !=
          initial_solution[i] for i in
          range(size)]) >= 1)

      # Solve again to check for uniqueness
      new_solution = mdl.solve(log_output=None)
      if new_solution:
         return [0,initial_solution]
      else:
         return [1,initial_solution]
  else:
      return [0,[]]
```

The solve_maze is also very similar to the previous solve maze, but it checks if there are more than one solution. It does this by excluding the initial solution and trying to solve it again.

C. Generator Results

For the generator we measured the times it would take to generate one maze for the following sizes and number of colors. Maze 1 has size 4 and 2 colors. Maze 2 has size 4 and 3 colors. Maze 3 has size 4 and 4 colors. Maze 4 has size 5 and 2 colors.

MAZE	Prolog time	Docplex time
1	2.18	4.82
2	20.67	0.67
3	23.59	1.01
4	512.53	31.1

TABLE II
MAZE CREATION TIMES IN SECONDS

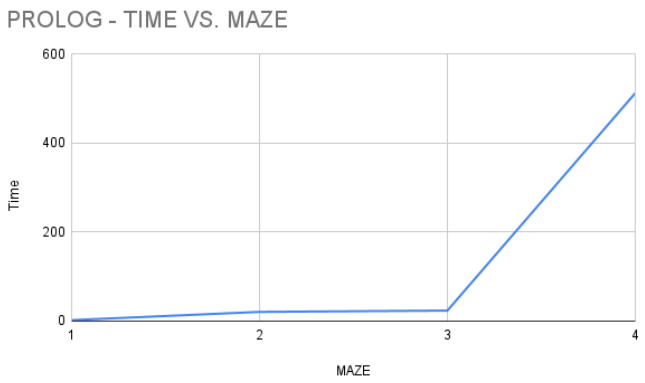

Fig. 7. PROLOG - TIME (s) vs. MAZE

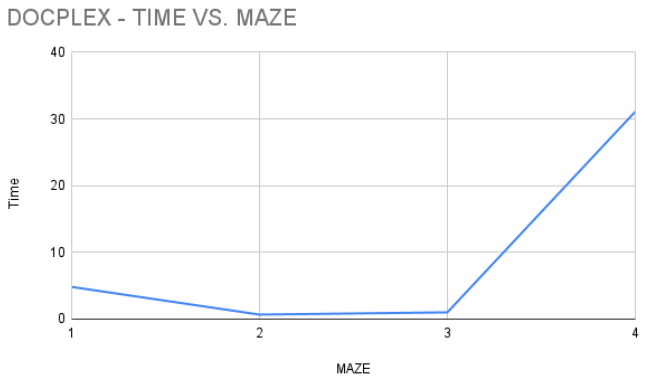

Fig. 8. DOCPLEX - TIME (s) vs. MAZE

Again as we expected times went up with complexity, but for the simpler maze in docplex, we had a worse time compared to other more complex ones, this maze was also the only one to have a worst time compared to prolog. Even thought this happened Docplex was still better than Prolog in all other cases.

IV. Conclusion

The SICStus Prolog solver demonstrated effectiveness in solving simpler mazes efficiently. However, its performance degraded noticeably with increasing maze complexity, particularly when paths did not pass through the center due to labeling strategies.

Conversely, the DOCPLEX solver consistently outperformed Prolog, especially on more complex mazes. DOCPLEX demonstrated superior scalability and efficiency, handling larger and more intricate mazes within reasonable time frames.

The maze generator implementations in both Prolog and DOCPLEX varied in their efficiency. Prolog's generator suffered from performance issues as maze size and complexity increased, whereas DOCPLEX maintained relatively stable performance across different maze configurations, showcasing its robustness in generating varied and complex puzzles.

Overall, while SICStus Prolog offered a straightforward approach to solving the Color Maze puzzle, DOCPLEX provided a more scalable and efficient solution, particularly suited for handling larger and more intricate puzzle configurations.

The insights gained from this study highlight the strengths and weaknesses of different constraint programming approaches in tackling puzzle-solving tasks, underscoring the importance of selecting the appropriate tool based on problem complexity and computational requirements.