University of Cyprus

Computer Science Department

Second Assignment: Solving the Bingo Constraints

MAI611 AI Fundamentals

Group Members:

Ahsan Fayyaz
fayyaz.ahsan@ucy.ac.cy

Hafiz Muhammad Soban Khan
khan.hafiz-muhammad-soban@ucy.ac.cy

Instructor:

Dr.Elpida Keravnou

28th November 2023

Solving the Bingo Constraints

Introduction:

In this assignment, the Bingo game is implemented which solves several constraints. The task at hand involves solving a constraint satisfaction problem (CSP) related to the construction of templates for Bingo cards. The goal is to generate unique Bingo card templates that adhere to specific constraints, such as the distribution of numbers across rows and columns.

Game Overview:

A player holds a collection of six 3 x 9 cards, resulting in an 18 x 9 block. Within this block, 72 positions remain vacant while the remaining 90 positions are assigned the numbers 1 to 90. The assignment's constraints follow that each number from 1 to 90 should appear precisely once in the constructed Bingo cards posing an intriguing challenge for the depth-first search-based solution.

Problem Decomposition:

The task is decomposed into two main sub-problems: constructing the template and filling it with actual numbers. The critical aspect is the construction of the template, which forms a CSP. The assignment focuses on the first sub-problem, emphasizing randomness to generate different templates on each program run.

Constructing Templates for Bingo Cards as a CSP

For the CSP, there are 162 binary variables (18 x 9). A value of 0 represents a blank space, and 1 represents a space to be filled. The solution involves depth-first search with backtracking. The order of unbound variables can be row-wise, column-wise, or randomly selected. When generating successor states, the two possible values (0 and 1) for the selected unbound variable should be randomly ordered.

Code Structure:

This code is generating a bingo card by constructing it in 3 parts:

1. **template_part1** - This sets up an initial 18x9 grid of -1s to mark empty cells
2. **template_part2** - This populates the 18x9 grid with 1s to indicate cells that should be filled, based on the following constraints:

   - **Constraints**:
     - Each number from 1 to 90 appears exactly once.
     - Each row must contain exactly 5 numbers.
     - Columns have specific number ranges and counts i.e., 1st column has 9 number, last column has 11 numbers and remaining columns have 10 numbers each.
   - It uses a CSP search to place the 1s randomly based on satisfying all constraints.
3. **template_part3** - This combines the template from Part 2 with actual bingo numbers, which are assigned randomly to the marked positions from Part 2.

Challenges and Lessons:

We faced several challenges while implementing the game which are stated below:

- The Bingo card template construction problem involves a considerable search space with a large number of variables (162 binary variables for an 18 x 9 grid).
- DFS is a powerful search algorithm, proved computationally expensive for large search spaces, potentially leading to extensive backtracking and increased runtime.
- Stochastic Local Search (SLS) with random sampling is used for constraint satisfaction problems, providing a more flexible and diverse exploration of the solution space compared to the deterministic nature of DFS.
- SLS can be more straightforward to implement, particularly when incorporating random sampling at various stages of the solution process.
- It facilitates experimentation with different aspects of the search strategy, providing a more agile approach for tuning parameters and exploring alternative methods.

Conclusion:

The implementation aims to address the unique challenges posed by constructing Bingo card templates, emphasizing the use of CSP and SLS. The randomization aspects ensure diverse templates are generated with each program execution, enhancing the variety and uniqueness of Bingo cards. The decision to use Stochastic Local Search over DFS was made by considering the problem's complexity, constraint satisfaction requirements, the need for diverse solutions, practical considerations, and the benefits offered by stochastic methods in handling large search spaces.

Outcomes:

```
Part 1: Initial Template
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1
```

```
Part 2: Populated Template with Constraints
1 1 1 1 1 0 0 0 0
1 0 0 1 1 0 0 1 1
0 0 0 0 1 1 0 1 1
1 1 1 1 0 0 0 0 1
0 1 1 1 0 0 1 0 1
0 0 0 1 0 1 1 0 1
0 1 1 1 0 0 1 0 1
1 1 1 0 0 1 0 1 0
0 1 0 0 0 0 1 0 0
0 1 1 1 0 0 1 1 0
1 0 1 1 1 0 0 0 1
1 1 0 1 1 0 0 0 1
0 0 1 0 1 1 0 1 1
1 0 1 0 0 1 1 0 0
0 1 0 0 1 1 1 0 1
1 0 0 1 0 0 1 0 0
0 0 0 0 1 1 0 1 0
1 1 1 0 1 0 0 0 1
```

```
Part 3: Combined Template and Numbers
 2 17 29 37 40  0  0  0  0
 1  0  0 34 44  0  0 73 82
 0  0  0  0 47 52  0 78 85
 3 14 20 33  0  0  0  0 81
 0 11 21 38  0  0 65  0 90
 0  0  0 30  0 54 63  0 87
 0 10 22 32  0  0 60  0 88
 8 15 26  0  0 58  0 79  0
 0 12  0  0  0  0 69  0  0
 0 18 23 35  0  0 64 71  0
 4  0 28 36 45  0  0  0 89
 9 13  0 31 48  0  0  0 84
 0  0 24  0 43 51  0 72 86
 7  0 25  0  0 59 62  0  0
 0 16  0  0 41 57 66  0 80
 5  0  0 39  0  0 67  0  0
 0  0  0  0 42 50  0 70  0
 6 19 27  0 46  0  0  0 83
```

The code for above implementation is in the Appendix I and code implemented with DFS is in Appendix II.

```
#BINGO Using SLS
import random

class BingoCard:
    def __init__(self):
        self.template_part1 = [[-1 for _ in range(9)] for _ in
range(18)]
        self.template_part2 = [[0 for _ in range(9)] for _ in
range(18)]
        self.template_part3 = [[0 for _ in range(9)] for _ in
range(18)]
        self.populate_template_part1()
        self.populate_template_part2()
        self.populate_numbers_part2()
        self.combine_part2_and_part3()

    def populate_template_part1(self):
        #This will involve setting -1s and 1s in the template
based on constraints
        pass

    def populate_template_part2(self):
        #Constraint: Each number from 1 to 90 appears exactly
once
        numbers = list(range(1, 91))
        random.shuffle(numbers)

        #Constraint: Each row contains five numbers
        #Constraint: Column-wise number distribution
        column_numbers = {
            0: numbers[:9],  #First column: 9 numbers
            8: numbers[9:20],  #Last column: 11 numbers
        }
        current_index = 20
        for i in range(1, 8):  #Columns 2nd to the penultimate
            column_numbers[i] =
numbers[current_index:current_index + 10]
            current_index += 10

        #Populate the template with 1s for filled positions
        for col, nums in column_numbers.items():
            rows = random.sample(range(18), len(nums))
            for row in rows:
                #Ensure each row has exactly 5 ones
                while sum(self.template_part2[row]) >= 5:
                    row = random.choice(rows)
                self.template_part2[row][col] = 1
```

```
    def populate_numbers_part2(self):
        #Implement the code to populate numbers in Part 2
        pass

    def combine_part2_and_part3(self):
        #Assign numbers to each column based on their domain
        for col in range(9):
            if col == 0:
                numbers = random.sample(range(1, 10),
9)  #Numbers for the first column
            elif col == 8:
                numbers = random.sample(range(80, 91),
11)  #Numbers for the last column
            else:
                start = col * 10
                numbers = random.sample(range(start, start + 10),
10)  #Numbers for middle columns

            idx = 0
            for row in range(18):
                if self.template_part2[row][col] == 1:
                    self.template_part3[row][col] = numbers[idx]
                    idx += 1

    def display(self):
        print("Part 1: Initial Template")
        for row in self.template_part1:
            print(" ".join(str(cell).rjust(2, ' ') for cell in
row))

        print("\nPart 2: Populated Template with Constraints")
        for row in self.template_part2:
            print(" ".join(str(cell) for cell in row))

        print("\nPart 3: Combined Template and Numbers")
        for row in self.template_part3:
            print(" ".join(str(cell).rjust(2, ' ') for cell in
row))

if __name__ == "__main__":
    bingo_card = BingoCard()
    bingo_card.display()
```

Appendix II

```
#BINGO Using DFS
class Search:
    def __init__(self):
        self.init_node = None
        self.current_Node = None
        self.open = []
        self.closed = []
        self.successor_nodes = []
        self.goal_state = None

    def get_Goal(self):
        return self.goal_state

    def put_Goal(self, goal):
        self.goal_state = goal

    def get_current_node(self):
        return self.current_Node

    def run_Search(self, init_state, g_state, search_method):
        self.goal_state = g_state
        return self.run_Search(init_state, search_method)

    def run_Search(self, init_state, search_method):
        self.init_node = Search_Node(init_state, None)
        print("\nStarting Search")
        self.open = [self.init_node]
        self.closed = []
        cnum = 1
        while self.open:
            self.select_Node(search_method)
            if self.current_Node.goalP(self):
                return self.report_Success()
            self.expand(search_method)
            self.closed.append(self.current_Node)
            cnum += 1
        return "Search Fails"

    def expand(self, search_method):
        self.successor_nodes =
self.current_Node.get_Successors(self)
        self.vet_Successors(search_method)
        if search_method == "depth_first":
            self.open = self.successor_nodes + self.open
        else:
            self.open.extend(self.successor_nodes)
```

```
    def vet_Successors(self, search_method):
        vslis = []
        for snode in self.successor_nodes:
            if search_method in ["depth_first", "breadth_first",
"best_first"]:
                if snode not in self.closed and snode not in
self.open:
                    vslis.append(snode)
            elif search_method == "branch_and_bound":
                if snode not in self.closed and snode not in
self.open:
                    vslis.append(snode)
                elif snode in self.open:
                    i = self.open.index(snode)
                    if snode.best_path_cost() <
self.open[i].best_path_cost():
                        self.open.pop(i)
                        self.open.append(snode)
                        vslis.append(snode)
                elif snode in self.closed:
                    i = self.closed.index(snode)
                    if snode.best_path_cost() <
self.closed[i].best_path_cost():
                        self.closed.pop(i)
                        self.open.append(snode)
                        vslis.append(snode)
            elif search_method == "A_star":
                if snode not in self.closed and snode not in
self.open:
                    vslis.append(snode)
                elif snode in self.open:
                    i = self.open.index(snode)
                    if snode.evaluation_fn(self.goal_state) <
self.open[i].evaluation_fn(self.goal_state):
                        self.open.pop(i)
                        self.open.append(snode)
                        vslis.append(snode)
                elif snode in self.closed:
                    i = self.closed.index(snode)
                    if snode.evaluation_fn(self.goal_state) <
self.closed[i].evaluation_fn(self.goal_state):
                        self.closed.pop(i)
                        self.open.append(snode)
                        vslis.append(snode)
        self.successor_nodes = vslis

    def on_Closed(self, new_node):
        return new_node in self.closed
```

```
    def on_Open(self, new_node):
        return new_node in self.open

    def select_Node(self, search_method):
        if search_method == "depth_first":
            self.current_Node = self.open.pop(0)
        elif search_method == "breadth_first":
            self.current_Node = self.open.pop(0)
        elif search_method == "branch_and_bound":
            i, min_cost = 0, self.open[0].best_path_cost()
            for j in range(1, len(self.open)):
                if self.open[j].best_path_cost() < min_cost:
                    i = j
                    min_cost = self.open[j].best_path_cost()
            self.current_Node = self.open.pop(i)
        elif search_method == "best_first":
            i, min_diff = 0,
self.open[0].difference(self.goal_state)
            for j in range(1, len(self.open)):
                if self.open[j].difference(self.goal_state) <
min_diff:
                    i = j
                    min_diff =
self.open[j].difference(self.goal_state)
            self.current_Node = self.open.pop(i)
        elif search_method == "A_star":
            i, min_eval = 0,
self.open[0].evaluation_fn(self.goal_state)
            for j in range(1, len(self.open)):
                if self.open[j].evaluation_fn(self.goal_state) <
min_eval:
                    i = j
                    min_eval =
self.open[j].evaluation_fn(self.goal_state)
            self.current_Node = self.open.pop(i)

    def report_Success(self):
        n = self.current_Node
        buf = [n.__str__()]
        plen = 1
        while n.get_parent() is not None:
            buf.insert(0, "\n")
            n = n.get_parent()
            buf.insert(0, n.__str__())
            plen += 1
        print("===============================")
        print("Search Succeeds")
```

```
        print(f"Efficiency {(float(plen) / (len(self.closed) +
1))}")
        print(f"Nodes visited: {len(self.closed) + 1}")
        print("Solution Path")
        return ''.join(buf)

    def pos_Open(self, new_node):
        for i in range(len(self.open)):
            if new_node.same_State(self.open[i]):
                return i
        return -1

    def pos_Closed(self, new_node):
        for i in range(len(self.closed)):
            if new_node.same_State(self.closed[i]):
                return i
        return -1

class Search_Node:
    def __init__(self, s, p=None, c=1):
        self.state = s
        self.parent = p
        self.cost = c

    def get_State(self):
        return self.state

    def get_parent(self):
        return self.parent

    def put_parent(self, p):
        self.parent = p

    def get_cost(self):
        return self.cost

    def put_cost(self, c):
        self.cost = c

    def goalP(self, searcher):
        return self.state.goalP(searcher)

    def difference(self, goal):
        return self.state.difference(goal)

    def best_path_cost(self):
        cos = 0
        n = self
```

```
        while n.parent is not None:
            cos += n.cost
            n = n.parent
        return cos

    def evaluation_fn(self, goal):
        return self.difference(goal) + self.best_path_cost()

    def get_Successors(self, searcher):
        slis = self.state.get_Successors(searcher)
        nlis = []
        for suc_state in slis:
            n = Search_Node(suc_state,
searcher.get_current_node(),
suc_state.cost_from(searcher.get_current_node().get_State()))
            nlis.append(n)
        return nlis

    def same_State(self, n2):
        return self.state.same_State(n2.get_State())

    def __str__(self):
        return f"Node with state {str(self.state)}"

from abc import ABC, abstractmethod

class Search_State(ABC):
    @abstractmethod
    def goalP(self, searcher):
        pass

    @abstractmethod
    def get_Successors(self, searcher):
        pass

    @abstractmethod
    def same_State(self, n2):
        pass

    @abstractmethod
    def cost_from(self, from_state):
        pass

    @abstractmethod
    def difference(self, goal):
        pass

class BingoSearch(Search):
```

```
    def __init__(self, f):
        self.file = f

    def get_file(self):
        return self.file

class Run_Bingo_Search:
    @staticmethod
    def main(args):
        search_method = args[0] if args else "depth_first"
        searcher = BingoSearch(search_method)
        init_state = Bingo_State()
        res = searcher.run_Search(init_state, search_method)
        print(res)

class Bingo_State(Search_State):
    def __init__(self, template=None):
        if template is None:
            self.template = [[-1 for _ in range(9)] for _ in
range(18)]
        else:
            self.template = template
    def __str__(self):
        board = "\n".join(" ".join(str(cell) for cell in row) for
row in self.template)
        return board
    def goalP(self, searcher):
        return self.is_complete() and
self.satisfies_global_constraints()

    def get_Successors(self, searcher):
        successors = []
        unassigned_spot = self.find_unassigned_spot()
        if unassigned_spot is None:
            return successors

        row, col = unassigned_spot
        for value in [0, 1]:
            new_template = [r[:] for r in self.template]
            new_template[row][col] = value
            if self.is_locally_consistent(new_template, row,
col):
                successors.append(Bingo_State(new_template))

        return successors

    def same_State(self, other_state):
        return self.template == other_state.template
```

```
    def cost_from(self, from_state):
        return 1

    def difference(self, goal):
        return 0

    def is_complete(self):
        return all(all(cell != -1 for cell in row) for row in
self.template)

    def satisfies_global_constraints(self):
        return True

    def find_unassigned_spot(self):
        for i in range(18):
            for j in range(9):
                if self.template[i][j] == -1:
                    return i, j
        return None

    def is_locally_consistent(self, template, row, col):
        return True
if __name__ == "__main__":
    Run_Bingo_Search.main(["depth_first"])
```