VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

Introduction to Artificial Intelligent

Assignment 1

BLOXORZ & WATER SORT PROBLEMs

Student: Lê Anh Vũ - 1915972

HO CHI MINH CITY, NOVEMBER 2022

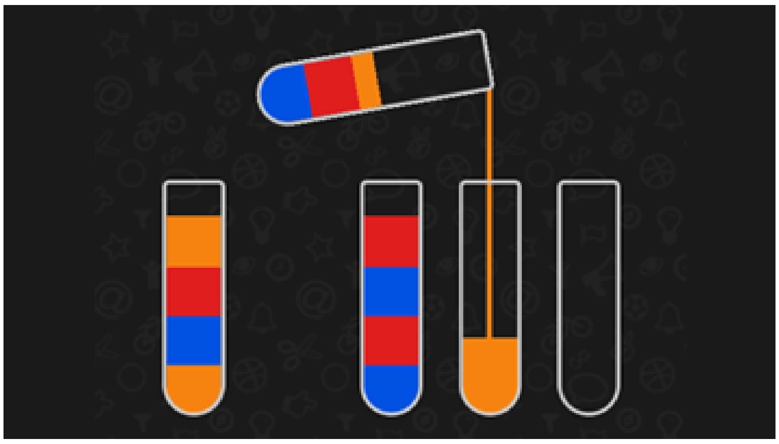
Contents

1 Water Sort

1.1 Rule of game

The WATER SORT is one of popular puzzle game. We are given a set of bottles filled with colored units water, and some empty bottles. This puzzles allow us to move colored water from a bottle to another destination bottle.

The goal of WATER SORT is sort the water by color in each of bottles.



The rule game is very simple: (0) Each bottle work like a stack, that is, we can pick up the top water in the bottle. (1) We can move the top water in bottle to another which has same color with in the top or the is being empty.

Notice that in case the destination bottle has enough capacity, all of water has to same color will be moved. In another hand, a part of the water will be moved up to limit of the destination bottle if the destination bottle does not have enough margin.

1.2 Formulating problem

1.2.1 Predefinition

Before we go to next section, we define and clarify some points that:

- Each color in WATER SORT is mapped by a number (BLUE = 1, YELLOW = 2, ...) counted from zero. *Example*: If we have a set three colors in game, the set mapped $\{0, 1, 2\}$.
- The state is formulated by 2D matrix, each element in which is a single array describe a bottle. An array for each bottle work like a stack with maximum length is 4 unit and each element is a number that was converted from color unit water. The unit water in the bottom of the bottle is element has index 0 in the array.
  Example: If we have bottle consist of two colors order from the bottom to the top is [BLUE, YELLOW, BLUE]. The corresponding array is [1,2,1].
  Example: The empty bottle is corresponding the empty array.
- Incase the number of bottles is $n = 2$, we need one more empty bottle to solve problem, the remaining cases with $n > 2$ we need 2 empty bottles to solve and the number of bottle now is $n + 2$.

Example: If there are 4 colors water, so we need 2 more empty bottles and there are 6 single array in matrix

- The goal state is set of permutations the admissible 2D matrix which there are n single array in matrix was filled by one color water and 1 or 2 empty bottles in. So the number of admissible states are $(n+1)!$ (if $n=2$) or $(n+2)!$.
  Example: If there are 3 colors water, one of admissible 2D matrix is

$$M = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 3 & 3 & 3 & 3 \\ \_ & \_ & \_ & \_ \\ 2 & 2 & 2 & 2 \\ \_ & \_ & \_ & \_ \end{bmatrix}$$

so the set of goal state is set of permutations of M. There are 5! goal states in state space.

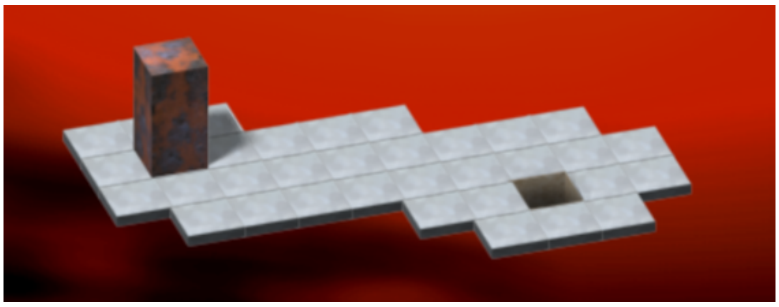
1.2.2 Water Sort definition

- STATES: State of n bottles, described by 2D matrix with n arrays, which each array determined units water in it.
- INITIAL STATE: The initialization of state, 2D matrix which array filled up by 4 colored units water.
- LEGAL MOVES: Source array (length>0) i in matrix can move a number of units water to destination array (length<4) j in condition:
  - Source array i is non-empty array.
  - Destination j array is non-full array.
  - Destination array j is empty or the top of element in source array j has same value with the top of element in destination array.
- GOAL STATES: A set of n permutation of admissible matrices (the admissible matrix is the matrix which all arrays is empty or fulled by 4 unique value).
- STEP COST: Each action/move will cost 1.

2 Bloxorz

2.1 Rule of game

Bloxorz is a 3-D block sliding puzzle game consists of a terrain that is built by 1×1 tile with a special shape and size, and a 1×1×2 size block. This game is a single agent path-finding problem that involves moving the block from its initial position using four directions (right, left, up, and down) and ensuring that its ends are always within the terrain boundary, until it falls into a 1×1 square hole in the terrain that represents our goal state. The block can be in three states, standing, lying horizontally, and lying vertically. When the block reaches the hole, it must be in standing state to fall in it.

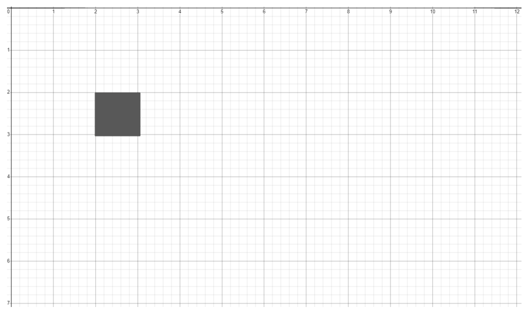


2.2 Formulating problem

2.2.1 Predefinition

Before we go to next section, we define and clarify some points that:

- The board is similar 2D coordinate space. The default space is always 2D coordinate.



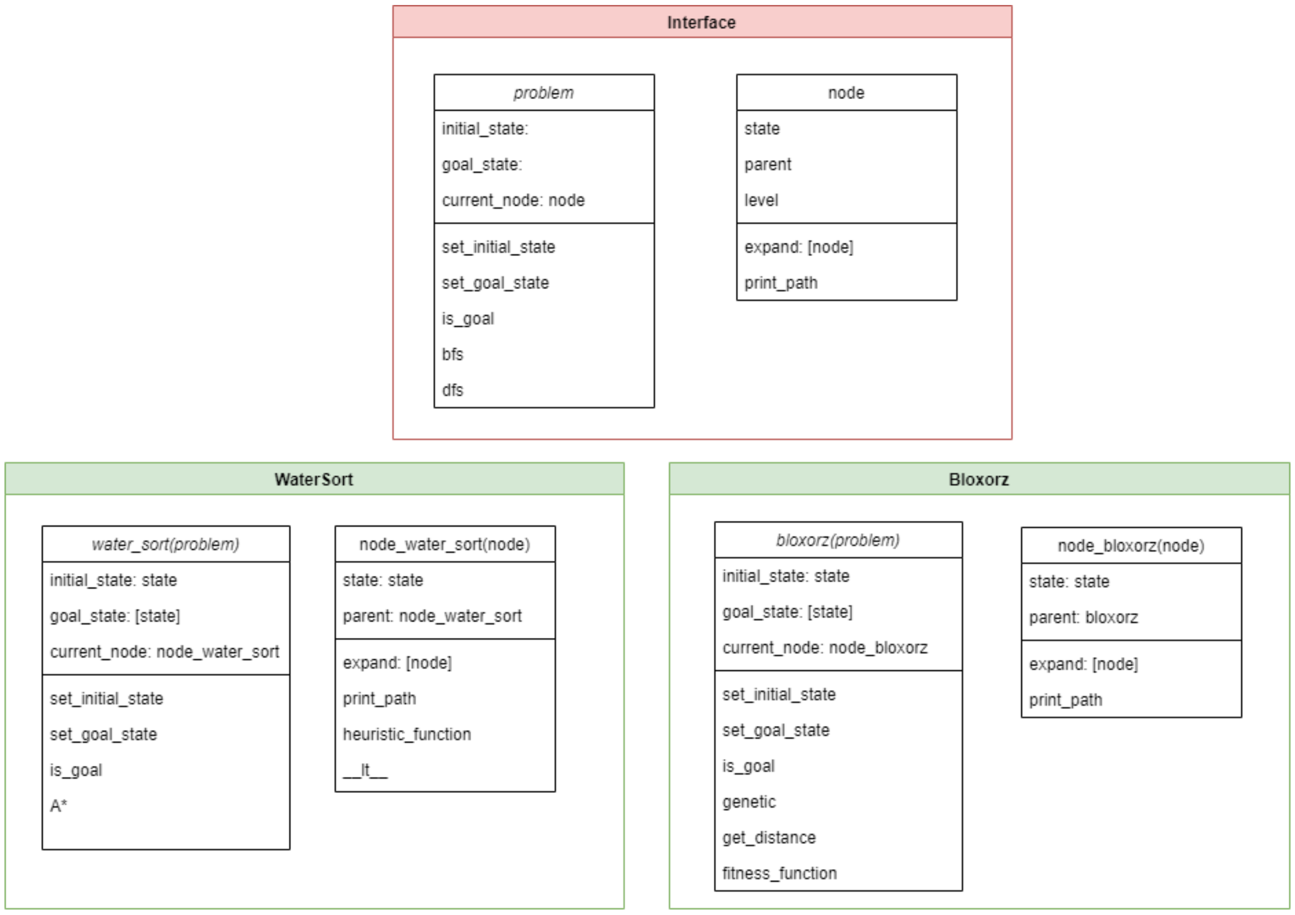
- When block is in vertical, x obtained one value and y obtained one value. When block is in horizontal either x can be obtained 2 values and y can be obtained 1 value or x 1 value and y 2 values.
- The map is 2D 15x15 matrix. Some values of each element in the map we should be mindful:
  - 'E' represent an empty cell.
  - 'C' represent the initialization of the block.
  - 'D' represent the destination cell.
  - '' represent the cell that we can step on.

2.2.2 Bloxorz formulation

- STATES: Coordinates of block on the board, in vertical or horizontal.
- INITIAL STATE: The map is 2D 15x15 matrix, Block on the map in a random position which can be either in vertical or horizontal orientation.
- GOAL STATE: After applying action A to the block in state S, we check if the block is at point G in the matrix, in a vertical position. If yes, we have reached the goal. If not, we did not reach the goal.
- LEGAL MOVES: Block can be move left, right, up, down. So, we have 4 different actions. With the any action, there are 2 possibilities: Block can fall down from the board or block will be on the board either in vertical or horizontal orientation.
- STEP COST: Each action/move will cost 1.

3 Searching Algorithm and Implementation

3.1 Data structure WaterSort and Bloxorz



3.2 State

3.2.1 WaterSort

State in WaterSort is an matrix store state in each bottle.
Example: [[], [2, 2], [3, 2, 3, 2], [1, 1, 1, 1], [3, 3]]

3.2.2 Bloxorz

State in Bloxorz is a dictionary in Python with 2 elements are 'current_position' and 'empty_cells'. Current position store direction and values x and y that block is stepping on. Empty cell store list coordinate of empty cells.
Example:

'current_position': 'direction': 'vertical', 'coordinate_y': [7],'coordinate_x': [2],
'empty_cell': [(0, 0), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 9), (0, 10), (0, 11), (0, 12), (0, 13), (0, 14), (1, 0), (1, 1)
, (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (1, 12), (1, 13), (1, 14)]

3.3 Breadth first search for WaterSort and Bloxorz

3.3.1 Breadth first search

```
    def bfs(self):
        print('Breadth first search')
        counter = 0
        if self.is_goal(self.current_node.state):
            return True
        frontier = queue.Queue()
        frontier.put(self.current_node)
        reached = [self.current_node.state]
        while not frontier.empty():
            self.current_node = frontier.get()
            for child in self.current_node.expand():
                counter += 1
                if self.is_goal(child.state):
                    child.print_path()
                    print(counter)
                    return True
                if child.state not in reached:
                    reached.append(child.state)
                    frontier.put(child)
        return False
```

3.3.2 Depth first search

```
    def dfs(self):
        print('Depth first search')
        counter = 0
        if self.is_goal(self.current_node.state):
            return True
        frontier = queue.LifoQueue()
        frontier.put(self.current_node)
        reached = [self.current_node.state]
        while not frontier.empty():
            self.current_node = frontier.get()
            for child in self.current_node.expand():
                counter += 1
                if self.is_goal(child.state):
                    child.print_path()
                    print(counter)
                    return True
                if child.state not in reached:
                    reached.append(child.state)
                    frontier.put(child)
        return False
```

3.4 A* for WaterSort

There is an important question to solve problem with A* Algorithm - What is heuristic function for Water-Sort?

Notice that, the smaller number of differences about color between 2 units water in all bottles, the smaller number of steps we need move to reach the goal. Thus, we can design a heuristic function with above idea.

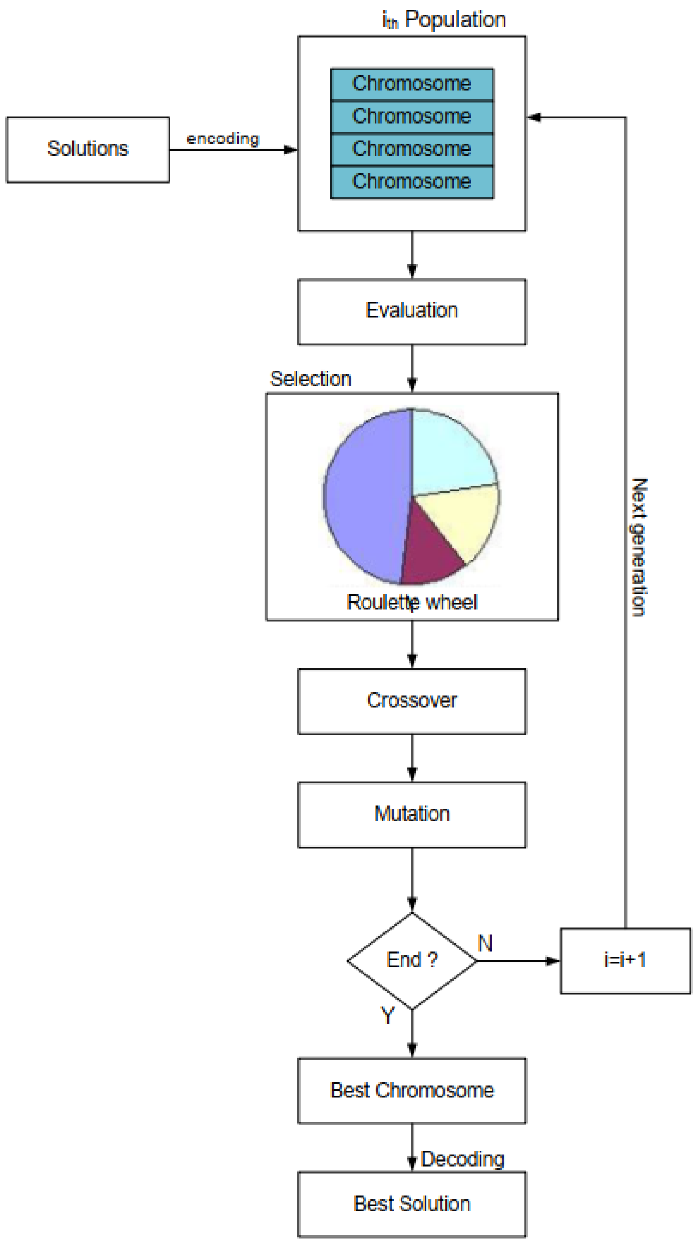
```
    def heuristic_function(self):
        evaluation = 0
        for i in range(len(self.state)):
            if len(self.state[i]) < 2:
                continue
            for j in range(len(self.state[i]) - 1):
                if self.state[i][j] != self.state[i][j + 1]:
                    evaluation += 1
        return evaluation
```

With heuristic function for each node, we can compare estimated cost from 2 states to reach a goal node.

```
    def astar(self):
        print('A* Search')
        if self.is_goal(self.current_node.state):
            return True
        counter = 0
        frontier = queue.PriorityQueue()
        frontier.put(self.current_node)
        reached = [self.current_node.state]
        while not frontier.empty():
            self.current_node = frontier.get()
            for child in self.current_node.expand():
                counter += 1
                if self.is_goal(child.state):
                    child.print_path()
                    print(counter)
                    return True
                if child.state not in reached:
                    reached.append(child.state)
                    frontier.put(child)
        return False
```

3.5 Genetic Algorithm

3.5.1 Flow chart of Genetic Algorithm



3.5.2 Chromosome and Initialize population

The chromosome is an array store each action(LEFT, RIGHT, UP, DOWN) to reach the goal.
The length of chromosome estimated by sum of delta(x) - delta(y)
For example: If we have the coordinate initial state is (2,3) and goal state is (7,5), so the length is $\sum = (7-2)+(5-3) = 7$
We generate 6 chromosome for initial population.

```python
    length_of_DNA = int(self.get_distance())
    original_element = ['LEFT', 'RIGHT', 'UP', 'DOWN']

    """Initialization ---------------------------------------"""
    population = []
    for i in range(6):
        population.append(random.choices(original_element, k=length_of_DNA))
```

Example for one chromosome: ['RIGHT', 'RIGHT', 'LEFT', 'LEFT', 'DOWN', 'RIGHT', 'UP', 'UP', 'DOWN', 'DOWN', 'DOWN', 'RIGHT', 'LEFT'],

3.5.3 Fitness function and evaluation population

1. **Fitness function**
   There are two factors affect to evaluate the node, that are **direction** and **distance from node to goal**
   For this problem the objective is minimizing the value of function
   $f(x) = 0.5$ **if** $x = x_g$ **and** $y = y_g$ **and** $direction! = vertical$
   $f(x) = 0$ **if** $x = x_g$ **and** $y = y_g$ **and** $direction = vertical$
   $f(x) = \sum(distance_x + distance_y)$ in remaining cases.

   x_g is x goal, y_g is y goal, $distance_x$
   is distance from x of current node to x of goal node, y the same.

```python
        if distance_x == 0 or distance_y == 0:
            if node_run.state.get('current_position').get('direction') !=
            'vertical':
                result = 0.5
            else:
                result = 0
            return result
        else:
            return distance
```

2. **Evaluation population**

```python
        """Evaluation -------------------------------------------"""
        value = list(map(self.fitness_function, population))
```

3.5.4 Selection

This step is very important because we will select fittest chromosomes are selected from the population for subsequent operations.

Based on the fitness values, more suitable chromosomes who have possibilities of producing low values of fitness function (because the value of our objective function needs to be 0) are selected and allowed to survive in succeeding generations. Some chromosomes are discarded to be unsuitable to produce low fitness values.

In this problem, I use ‘roulette wheel method’. **All of step described in lines of code below**

```python
        """Selection -------------------------------------------"""
        fitness_value = list(map(lambda x: 1 / (1 + x), value))

        total = sum(fitness_value)

        probability = list(map(lambda x: x / total, fitness_value))

        cumulative_probability = []
        for i in range(6):
            if i == 0:
                cumulative_probability.append(probability[i])
            else:
                cumulative_probability.append(probability[i] + cumulative_probability[i
                - 1])

        roulette_wheel = []
        for i in range(6):
            roulette_wheel.append(random.random())

        new_population = []
        for i in range(6):
            if roulette_wheel[i] < cumulative_probability[0]:
                new_population.append(population[0])
            else:
                for j in range(6):
                    if roulette_wheel[i] > cumulative_probability[j]:
                        new_population.append(population[j + 1])
                        break
```

3.5.5 Cross-over

Some chromosomes in population will mate through process called crossover thus producing new chromosomes named offspring which its genes composition are the combination of their parent.

In this problem, I use ‘one-cut point method’. **All of step described in lines of code below**

```python
        parent_index = random.sample(range(0, 5), 3)
        position_interchange = random.sample(range(0, length_of_DNA - 1), 3)
        for i in range(3):
            if i == 0:
                new_population[parent_index[0]] =
                new_population[parent_index[0]][:position_interchange[0]] + \
                new_population[parent_index[1]][position_interchange[0]:]
            if i == 1:
                new_population[parent_index[1]] =
                new_population[parent_index[1]][:position_interchange[1]] + \
                new_population[parent_index[2]][position_interchange[1]:]
            if i == 2:
                new_population[parent_index[2]] =
                new_population[parent_index[2]][:position_interchange[2]] + \
                new_population[parent_index[0]][position_interchange[2]:]
```

3.5.6 Mutation

Number of chromosomes that have mutations in a population is determined by the mutation_rate parameter. Mutation process is done by replacing the gen at random position with a new value.

In this problem, I use 'one-cut point method'. **All of step described in lines of code below**

```
    mutation_rate = 0.1
    total_gen = 6 * length_of_DNA
    num_gen_exchange = round(mutation_rate * total_gen)
    mutation_gen = random.sample(range(0, 6 * length_of_DNA - 1), num_gen_exchange)

    for i in mutation_gen:
        n = int(i / length_of_DNA)
        m = i - n * length_of_DNA
        new_population[n][m] = random.choice(original_element)
```

3.5.7 Reevaluate new population

Number of chromosomes that have mutations in a population is determined by the mutation_rate parameter. Mutation process is done by replacing the gen at random position with a new value.

In this problem, I use 'one-cut point method'. **All of step described in lines of code below**

```
    mutation_rate = 0.1
    total_gen = 6 * length_of_DNA
    num_gen_exchange = round(mutation_rate * total_gen)
    mutation_gen = random.sample(range(0, 6 * length_of_DNA - 1), num_gen_exchange)

    for i in mutation_gen:
        n = int(i / length_of_DNA)
        m = i - n * length_of_DNA
        new_population[n][m] = random.choice(original_element)
```

4 Results

4.1 WaterSort

4.1.1 Testcase 1

```
initial_state1 = [
    [1, 4, 4, 5],
    [5, 3, 2, 4],
    [4, 5, 1, 3],
    [3, 5, 1, 2],
    [1, 2, 3, 2]
]
```

	DFS	BFS	A*
Num of nodes to reach goal	246	28	56
Num of node traversals	766	99729	131

4.1.2 Testcase 2

```
initial_state2 = [
    [5, 2, 3, 1],
    [3, 1, 2, 1],
    [3, 2, 3, 2],
    [4, 5, 4, 1],
    [4, 4, 5, 5]
]
```

	DFS	BFS	A*
Num of nodes to reach goal	590	28	40
Num of node traversals	1983	89989	100

4.1.3 Compare Algorithm

Number of nodes need checked to find the goal: $A* < DFS < BFS$
Number of nodes in the route to reach the goal: $BFS < A* < DFS$
Elapsed time $A* < DFS < BFS$

4.2 Bloxorz

4.2.1 Testcase 1

BFS & DFS

	DFS	BFS
Num of nodes to reach goal	46	9
Num of node traversals	158	381

Genetic Algorithm

	Population random 1	Population random 2	Population random 3
Number of generations	641	273	1245

4.2.2 Testcase 2

BFS & DFS

	DFS	BFS
Num of nodes to reach goal	50	9
Num of node traversals	185	369

Genetic Algorithm

	Population random 1	Population random 2	Population random 3
Number of generations	189	1065	2018

4.2.3 Compare Algorithm

Number of nodes need checked to find the goal: $DFS < BFS$
Number of nodes in the route to reach the goal: $BFS < DFS$
Elapsed time $BFS < Genetic < DFS$

References

[1] Denny Hermawanto. Genetic Algorithm for Solving Simple Mathematical Equality Problem, Link.

[2] Ph.D. Niranjan Pramanik. Genetic Algorithm—explained step by step with example, Link.

[3] Lamiaa A. Elrefaeia b * Tahani Q. Alhassana, Shefaa S. Omara. Game of Bloxorz Solving Agent Using Informed and Uninformed Search Strategies , Link.