Mini-Project 2 {COMP 472}

Artem Chernigel {40115241}

1. Imports

Import all the necessary libraries, such as: **numpy** (to work with arrays), **time** (to record the runtime), **collections** (to count lengths of vehicles in a board configuration), and Workbook from **openpyxl** (to save spreadsheet with all the metrics).

In [1]:
```
import numpy as np
import time
import collections
from openpyxl import Workbook
```

2. Define Classes

This section will explain the classes that were created and used in order to efficiently solve the puzzles and record the performance of each algorithm.

2.1 Node

Node class defines a single configuration of a board. It represents a single node in a complete game tree.
Variables:

state -- string board configuration of length 36
children -- array of children of the node
fuel -- dictionary of available fuel of vehicles
cost -- int cost from the start to the node
path -- string path from the start to the node
heuristic_value -- int heuristic value of the node
parent -- Node parent of the node
lengths -- dictionary lengths of the vehicles of the node

Functions:

addChild(self, node)
-- add node to the array of children
setFuel(self, car, fuel)
-- set fuel of the given car to fuel
getFuel(self, car)
-- get fuel of the given car
getLength(self, car)
-- get length of the given car
inList(self, given_list)
-- check whether self.state is in the given list

In [2]:
```
class Node:
    def __init__(self, state):
        self.state = state
        self.children = np.array([])
        self.fuel = {"A" : 100,
                     "B" : 100,
                     "C" : 100,
                     "D" : 100,
                     "E" : 100,
                     "F" : 100,
                     "G" : 100,
                     "H" : 100,
                     "I" : 100,
                     "J" : 100,
                     "K" : 100,
                     "L" : 100,
                     "M" : 100,
                     "N" : 100,
                     "O" : 100,
                     "P" : 100,
                     "Q" : 100,
                     "R" : 100,
                     "S" : 100,
                     "T" : 100,
                     "U" : 100,
                     "V" : 100,
                     "W" : 100,
                     "X" : 100,
                     "Y" : 100,
                     "Z" : 100}
        self.cost = 0
        self.path = ""
        self.heuristic_value = 0
        self.parent = None
        self.lengths = collections.Counter(self.state)
    def addChild(self, node):
        self.children = np.append(self.children, node)
    def setFuel(self, car, fuel):
        self.fuel[car] = fuel
    def getFuel(self, car):
        return int(self.fuel[car])
    def getLength(self, car):
        return self.lengths[car]
    def inList(self, given_list):
        for i in range(0, len(given_list)):
            if(self.state == given_list[i].state):
                return True
        return False
```

2.2 PrioQueue

PrioQueue class defines a priority queue of nodes. On the background the class utilizes a numpy array, and inserts the node in a correct place depending on the heuristic value of the node. The priority queue is sorted in the ascending order, meaning that the node of the minimum heuristic

value will be placed at the index 0. The PrioQueue is utilized for general_solve method of the Algorithm class, which is explained in the next section.

Variables:

queue -- array of nodes sorted as an ascending priority queue

Functions:

insert(self, node)

-- insert the node in the priority queue based on the heuristic value of the given node

pop(self)

-- pop the node which has the minimum heuristic value in the priority queue

-- return the node that was removed from the priority queue

In [3]:
```
class PrioQueue(Node):
    def __init__(self):
        self.queue = np.array([])
    def insert(self, node):
        if(len(self.queue) == 0):
            self.queue = np.append(self.queue, node)
        elif(node.heuristic_value <= self.queue[0].heuristic_value):
            self.queue = np.insert(self.queue, 0, node)
        elif(node.heuristic_value >= self.queue[-1].heuristic_value):
            self.queue = np.append(self.queue, node)
        else:
            for i in range(0, len(self.queue) - 1):
                if(node.heuristic_value > self.queue[i].heuristic_value and node.
                    self.queue = np.insert(self.queue, i + 1, node)
                    break
    def pop(self):
        first, self.queue = self.queue[0], self.queue[1:]
        return first
```

2.3 Algorithm

Algorithm class is a superclass to all algorithms executed in this project, that is: **UCS**, **GBFS**, and **A**. It defines general methods that are used in each algorithm, therefore, all consecutive classes of algorithms will be subclasses of Algorithm.

Variables:

root -- node which represents the root of the game tree *Functions*:

calculateHeuristicValue(self, node)

-- calculate heuristic value of the given node (returns immediately since a superclass)

solve(self)

-- will encapsulate general_solve method (returns immediately since a superclass)

generate_next_moves(self, node)

-- move every vehicle in any possible direction by any possible steps

-- return array of children (every possible move) of the given node

is_solution(self, node)

-- verify wether given node is a solution (solution if pathway for A is clear)

-- return false if node is not a solution, return true otherwise

general_solve(self, node)

-- main brain of the algorithm, which utilizes open & closed PrioQueues to traverse

the game tree, and generate_next_moves method to generate next board configurations

-- return parameters of the solution

In [4]:

```
class Algorithm:
    def __init__(self, root):
        self.root = root
    def calculateHeuristicValue(self, node):
        return
    def solve(self):
        return
    def generate_next_moves(self, node):
        node.state = np.array([*node.state]).reshape(6, 6)
        children = np.array([])
        for i in range(0, len(node.state)):
            for j in range(0, len(node.state)):
                horizontal_orientation = True
                if(has_vehicle(node.state, i, j)):
                    below_upper_boundary = i - 1 >= 0
                    higher_lower_boundary = i + 1 < len(node.state)

                    if((below_upper_boundary and node.state[i - 1][j] == node.sta
                      or (higher_lower_boundary and node.state[i + 1][j] == node.
                        horizontal_orientation = False
                    if(horizontal_orientation):
                        internal_j = j + 1
                        # movement to the right
                        while(internal_j < len(node.state) and not has_vehicle(nc
                            if(node.getFuel(node.state[i][j]) > 0):
                                next_state_matrix = np.empty_like(node.state)
                                next_state_matrix[:] = node.state
                                move_vehicle_counter = node.getLength(node.state[
                                temp_j_moving = internal_j
                                temp_j_removing = j
                                moving_char = node.state[i][j]
                                if(i == 2 and internal_j == 5 and node.state[i][j
                                    moving_char = '.'
                                while(move_vehicle_counter > 0):
                                    next_state_matrix[i][temp_j_removing] = "."
                                    next_state_matrix[i][temp_j_moving] = moving_
                                    temp_j_moving -= 1
                                    temp_j_removing -= 1
                                    move_vehicle_counter -= 1
                                next_state = Node("".join(str(x) for x in next_st
                                next_state.cost = node.cost + 1
                                next_state.heuristic_value = self.calculateHeuris
                                next_state.path = node.path + "\n " + str(node.st
                                next_state.fuel = node.fuel.copy()
                                next_state.setFuel(node.state[i][j], node.getFuel
                                children = np.append(children, next_state)
                            internal_j += 1
                        # movement to the left
                        internal_j = j - 1
                        while(internal_j >= 0 and not has_vehicle(node.state, i,
                            if(node.getFuel(node.state[i][j]) > 0):
                                next_state_matrix = np.empty_like(node.state)
                                next_state_matrix[:] = node.state
                                move_vehicle_counter = node.getLength(node.state[
                                temp_j_moving = internal_j
                                temp_j_removing = j
```

```
                    moving_char = node.state[i][j]
                    if(i == 2 and internal_j == 5 and node.state[i][j
                        moving_char = '.'
                    while(move_vehicle_counter > 0):
                        next_state_matrix[i][temp_j_removing] = "."
                        next_state_matrix[i][temp_j_moving] = moving_
                        temp_j_moving += 1
                        temp_j_removing += 1
                        move_vehicle_counter -= 1
                    next_state = Node("".join(str(x) for x in next_st
                    next_state.cost = node.cost + 1
                    next_state.heuristic_value = self.calculateHeuris
                    next_state.path = node.path + "\n " + str(node.st
                    next_state.fuel = node.fuel.copy()
                    next_state.setFuel(node.state[i][j], node.getFuel
                    children = np.append(children, next_state)
                internal_j -= 1
        else:
            internal_i = i + 1
            # movement downards
            while(internal_i < len(node.state) and not has_vehicle(no
                if(node.getFuel(node.state[i][j]) > 0):
                    next_state_matrix = np.empty_like(node.state)
                    next_state_matrix[:] = node.state
                    move_vehicle_counter = node.getLength(node.state[
                    temp_i_moving = internal_i
                    temp_i_removing = i
                    moving_char = node.state[i][j]
                    while(move_vehicle_counter > 0):
                        next_state_matrix[temp_i_removing][j] = "."
                        next_state_matrix[temp_i_moving][j] = moving_
                        temp_i_moving -= 1
                        temp_i_removing -= 1
                        move_vehicle_counter -= 1
                    next_state = Node("".join(str(x) for x in next_st
                    next_state.cost = node.cost + 1
                    next_state.heuristic_value = self.calculateHeuris
                    next_state.path = node.path + "\n " + str(node.st
                    next_state.fuel = node.fuel.copy()
                    next_state.setFuel(node.state[i][j], node.getFuel
                    children = np.append(children, next_state)
                internal_i += 1
            # movement upwards
            internal_i = i - 1
            while(internal_i >= 0 and not has_vehicle(node.state, int
                if(node.getFuel(node.state[i][j]) > 0):
                    next_state_matrix = np.empty_like(node.state)
                    next_state_matrix[:] = node.state
                    move_vehicle_counter = node.getLength(node.state[
                    temp_i_moving = internal_i
                    temp_i_removing = i
                    moving_char = node.state[i][j]
                    while(move_vehicle_counter > 0):
                        next_state_matrix[temp_i_removing][j] = "."
                        next_state_matrix[temp_i_moving][j] = moving_
                        temp_i_moving += 1
                        temp_i_removing += 1
```

```
                                move_vehicle_counter -= 1
                            next_state = Node("".join(str(x) for x in next_st
                            next_state.cost = node.cost + 1
                            next_state.heuristic_value = self.calculateHeuris
                            next_state.path = node.path + "\n " + str(node.st
                            next_state.fuel = node.fuel.copy()
                            next_state.setFuel(node.state[i][j], node.getFuel
                            children = np.append(children, next_state)
                        internal_i -= 1
        node.state = "".join(str(x) for x in node.state.flatten())
        return children
    def is_solution(self, node):
        node.state = np.array([*node.state]).reshape(6, 6)
        internal_j = np.where(node.state[2] == 'A')[0][-1] + 1
        if(internal_j != 6):
            for j in range(internal_j, len(node.state)):
                if(node.state[2][j] != '.'):
                    node.state = "".join(str(x) for x in node.state.flatten())
                    return False
        solution_matrix = np.empty_like(node.state)
        solution_matrix[:] = node.state
        solution_matrix[2][np.where(node.state[2] == 'A')[0][0]] = '.'
        solution_matrix[2][np.where(node.state[2] == 'A')[0][-1]] = '.'
        move_vehicle_counter = node.getLength('A')
        j_moving = 5
        while(move_vehicle_counter > 0):
            solution_matrix[2][j_moving] = 'A'
            j_moving -= 1
            move_vehicle_counter -= 1
        solution_matrix = "".join(str(x) for x in solution_matrix.flatten())
        solution_state = Node(solution_matrix)
        solution_state.heuristic_value = self.calculateHeuristicValue(solution_st
        solution_state.parent = node
        solution_state.path = node.path + "\n " + "A right " + str((5 - internal_
        solution_state.fuel = node.fuel.copy()
        solution_state.setFuel('A', node.getFuel('A') - (6 - internal_j))
        node.addChild(solution_state)
        return True
    def general_solve(self, node):
        opened = PrioQueue()
        opened.insert(node)
        start = time.time()
        num_of_steps = 0
        closed = np.array([])

        while True:
            num_of_steps += 1

            if(num_of_steps % 2000 == 0):
                print(f"Steps performed: {num_of_steps}")
            if(len(opened.queue) == 0):
                return np.array([False, closed, num_of_steps, time.time() - start
                break

            selected_node = opened.pop()

            if(self.is_solution(selected_node)):
```

```
                return np.array([selected_node.children[0], len(selected_node.chi
            closed = np.append(closed, selected_node)
            new_nodes = self.generate_next_moves(selected_node)

            if(len(new_nodes) > 0):
                for new_node in new_nodes:
                    if(not new_node.inList(closed) and not new_node.inList(opened
                        new_node.parent = selected_node
                        opened.insert(new_node)
                    elif(new_node.inList(opened.queue)):
                        old_node, index = getOldNode(opened.queue, new_node)
                        if(new_node.heuristic_value < old_node.heuristic_value):
                            new_node.parent = selected_node
                            opened.queue = np.delete(opened.queue, index)
                            opened.insert(new_node)
```

2.4 UCS

UCS class is a subclass of Algorithm class, which defines the logic of the Uniform Cost Search. All the methods of the Algorithm are inherited, therefore, only the ones that were placeholders (calculateHeuristicValue & solve) have to be defined. Albeit the fact that UCS does not have a heuristic function, for the purposes of better structurization of the code the function returns the cost of reaching given node from the root.

Variables:

Functions:

calculateHeuristicValue(self, node)

-- return the cost of reaching given node from the root

solve(self)

-- call general_solve of the superclass in order to solve the puzzle

In [5]:
```
class UCS(Algorithm):
    def __init__(self, root):
        super().__init__(root)
    def calculateHeuristicValue(self, node):
        return node.cost
    def solve(self):
        return self.general_solve(self.root)
```

2.5 GBFS

GBFS class is a subclass of Algorithm class, which defines the logic of the Greedy Best-First Search. All the methods of the Algorithm are inherited, therefore, only the ones that were placeholders (calculateHeuristicValue & solve) have to be defined. The heuristic function is passed through the constructor of the class, therefore, calculateHeuristicValue simply calls the passed function in order to calculate the heuristic value. Solve method stays the same with the exception of a necessity to calculate the heuristic value of the root itself.

Variables:

heuristic -- function that calculates the heuristic value of a node

Functions:

calculateHeuristicValue(self, node)

-- call heuristic function passed beforehand to the constructor to calculate and return heuristic value

solve(self)

-- call general_solve of the superclass in order to solve the puzzle; calculate heuristic value of the root beforehand

In [6]:
```
class GBF(Algorithm):
    def __init__(self, root, heuristic):
        self.heuristic = heuristic
        super().__init__(root)
    def calculateHeuristicValue(self, node):
        return self.heuristic(node)
    def solve(self):
        self.root.heuristic_value = self.calculateHeuristicValue(self.root)
        return self.general_solve(self.root)
```

2.6 A

A class is a subclass of Algorithm class, which defines the logic of the A Search. All the methods of the Algorithm are inherited, therefore, only the ones that were placeholders (calculateHeuristicValue & solve) have to be defined. The heuristic function is passed through the constructor of the class, therefore, calculateHeuristicValue simply calls the passed function in order to calculate the heuristic value, however, also adds the cost of the node, which is a difference between the GBFS class. Solve method stays the same with the exception of a necessity to calculate the heuristic value of the root itself.

Variables:

Functions:

calculateHeuristicValue(self, node)

-- call heuristic function passed beforehand to the constructor to calculate and return heuristic value added to the node cost

solve(self)

-- call general_solve of the superclass in order to solve the puzzle

In [7]:
```
class A(Algorithm):
    def __init__(self, root, heuristic):
        self.heuristic = heuristic
        super().__init__(root)
    def calculateHeuristicValue(self, node):
        return node.cost + self.heuristic(node)
    def solve(self):
        self.root.heuristic_value = self.calculateHeuristicValue(self.root)
        return self.general_solve(self.root)
```

2.7 SolutionManager

SolutionManager class is a class that encapsulates the previously-defined algorithms, initializes puzzles by reading an input file, solves the puzzles and records the solutions and searches of each algorithm.

Variables:

file_name

-- string input file name

roots

-- roots of puzzles that were extracted from the input file

solutions

-- array of solutions to the puzzles

solution_path_length

-- int solution path length

search_path_lentgh

-- int search path length

search_path

-- string search path

solution_path

-- string solution path

solution_runtime

-- float solution runtime

solution_counter

-- int index of the puzzle that is currently being solved

solution_found

-- bool indicates whether solution was found or not after execution of algorithm

sheet_row_counter

-- int index of the row of the spreadsheet to fill

workbook

-- workbook with which a spreadsheet will be created and saved afterwards

sheet

-- spreadsheet which will have all the statistics with regards to each puzzle and algorithm

Functions:

initialize(self)

-- read input file and add each puzzle to roots array

check_and_set_params(self, solution_params)

-- check parameters returned by solve method of algorithms and set parameters accordingly

solve_and_out(self)

-- solve each puzzle in roots by every algorithm created and record each solution

admissibility_check(self)

-- create admissibility checker for each heuristic and verify the admissibility

record_solution(self, method)

-- create search and solution files for the current solution and method

In [8]:
```
class SolutionManager:
    def __init__(self, file_name):
        self.file_name = file_name
        self.roots = np.array([])
        self.solutions = np.array([])
        self.solution_path_length = 0
        self.search_path_lentgh = 0
        self.search_path = np.array([])
        self.solution_path = ""
        self.solution_runtime = 0
        self.solution_counter = 0
        self.solution_found = True
        self.sheet_row_counter = 2
        self.workbook = Workbook()
        self.sheet = self.workbook.active
    def initialize(self):
        self.sheet["A1"] = "Puzzle Number"
        self.sheet["B1"] = "Algorithm"
        self.sheet["C1"] = "Heuristic"
        self.sheet["D1"] = "Length of the Solution"
        self.sheet["E1"] = "Length of the Search Path"
        self.sheet["F1"] = "Execution Time (in seconds)"
        with open (self.file_name) as f:
            for line in f.readlines():
                if line[0] != "\n" and line[0] != "#" and line != "":
                    line_arr = line.split(" ")
                    root = Node(line_arr[0][0:36])
                    for i in range (1, len(line_arr)):
                        if(line_arr[i] != "\n"):
                            root.setFuel(line_arr[i][0], line_arr[i][1])
                    self.roots = np.append(self.roots, root)
                    self.solutions = np.append(self.solutions, root)
    def check_and_set_params(self, solution_params):
        if(solution_params[0] == False):
            self.solution_found = False
            self.search_path = solution_params[1]
            self.search_path_length = solution_params[2]
            self.solution_runtime = solution_params[3]
            return
        else:
            self.solution_found = True
            self.solutions[self.solution_counter - 1] = solution_params[0]
            self.solution_path_length = solution_params[1]
            self.search_path_length = solution_params[2]
            self.solution_path = solution_params[3]
            self.solution_runtime = solution_params[4]
            self.search_path = solution_params[5]
            return
    def solve_and_out(self):
        for root in self.roots:
            self.solution_counter += 1
            print("\n-----------UCS-----------")
            ucs_alg = UCS(root)
            self.check_and_set_params(ucs_alg.solve())
            self.record_solution("ucs")
            print("UCS executed puzzle #" + str(self.solution_counter))
```

```
            print("\n----------GBF[h1]----------")
            gbf_alg_h1 = GBF(root, h1)
            self.check_and_set_params(gbf_alg_h1.solve())
            self.record_solution("gbfs-h1")
            print("GBF_h1 executed puzzle #" + str(self.solution_counter))
            print("\n----------GBF[h2]----------")
            gbf_alg_h2 = GBF(root, h2)
            self.check_and_set_params(gbf_alg_h2.solve())
            self.record_solution("gbfs-h2")
            print("GBF_h2 executed puzzle #" + str(self.solution_counter))
            print("\n----------GBF[h3]----------")
            gbf_alg_h3 = GBF(root, h3)
            self.check_and_set_params(gbf_alg_h3.solve())
            self.record_solution("gbfs-h3")
            print("GBF_h3 executed puzzle #" + str(self.solution_counter))
            print("\n----------GBF[h4]----------")
            gbf_alg_h4 = GBF(root, h4)
            self.check_and_set_params(gbf_alg_h4.solve())
            self.record_solution("gbfs-h4")
            print("GBF_h4 executed puzzle #" + str(self.solution_counter))
            print("\n----------A[h1]----------")
            a_alg_h1 = A(root, h1)
            self.check_and_set_params(a_alg_h1.solve())
            self.record_solution("a-h1")
            print("A_h1 executed puzzle #" + str(self.solution_counter))
            print("\n----------A[h2]----------")
            a_alg_h2 = A(root, h2)
            self.check_and_set_params(a_alg_h2.solve())
            self.record_solution("a-h2")
            print("A_h2 executed puzzle #" + str(self.solution_counter))
            print("\n----------A[h3]----------")
            a_alg_h3 = A(root, h3)
            self.check_and_set_params(a_alg_h3.solve())
            self.record_solution("a-h3")
            print("A_h3 executed puzzle #" + str(self.solution_counter))
            print("\n----------A[h4]----------")
            a_alg_h4 = A(root, h4)
            self.check_and_set_params(a_alg_h4.solve())
            self.record_solution("a-h4")
            print("A_h4 executed puzzle #" + str(self.solution_counter))
        self.workbook.save(filename="results_table.xlsx")
    def admissibility_check(self):
        index = 0
        for root in self.roots:
            print("\n----------GBF[h1]----------")
            gbf_h1_admissibility = AdmissibilityChecker(root, h1)
            gbf_h1_admissibility.solve()
            if(gbf_h1_admissibility.admissible):
                print("ADMISSIBLE")
            else:
                print("__NOT__ ADMISSIBLE")
            print("\n----------GBF[h2]----------")
            gbf_h2_admissibility = AdmissibilityChecker(root, h2)
            gbf_h2_admissibility.solve()
            if(gbf_h2_admissibility.admissible):
                print("ADMISSIBLE")
            else:
```

```
            print("__NOT__ ADMISSIBLE")
        print("\n----------GBF[h3]----------")
        gbf_h3_admissibility = AdmissibilityChecker(root, h3)
        gbf_h3_admissibility.solve()
        if(gbf_h3_admissibility.admissible):
            print("ADMISSIBLE")
        else:
            print("__NOT__ ADMISSIBLE")
        print("\n----------GBF[h4]----------")
        gbf_h4_admissibility = AdmissibilityChecker(root, h4)
        gbf_h4_admissibility.solve()
        if(gbf_h4_admissibility.admissible):
            print("ADMISSIBLE")
        else:
            print("__NOT__ ADMISSIBLE")
        index += 1
def record_solution(self, method):
    self.sheet["A" + str(self.sheet_row_counter)] = self.solution_counter
    self.sheet["B" + str(self.sheet_row_counter)] = method.split("-")[0].uppe
    if(method.__contains__("ucs")):
        self.sheet["C" + str(self.sheet_row_counter)] = "NA"
    else:
        self.sheet["C" + str(self.sheet_row_counter)] = method.split("-")[1]
    self.sheet["E" + str(self.sheet_row_counter)] = self.search_path_length
    self.sheet["F" + str(self.sheet_row_counter)] = self.solution_runtime
    with open("Output/" + method + "-search-" + str(self.solution_counter) +
        for node in self.search_path:
            if(method.__contains__("ucs")):
                f.write(str(node.cost) + " " + str(node.cost) + " 0")
            elif(method.__contains__("gbf")):
                f.write(str(node.heuristic_value) + " 0 " + str(node.heuristi
            elif(method.__contains__("a")):
                f.write(str(node.cost + node.heuristic_value) + " " + str(noc
            f.write(" " + node.state + " ")
            for car, fuel in node.fuel.items():
                if(fuel != 100):
                    f.write(car + str(fuel) + " ")
            f.write("\n")

    with open("Output/" + method + "-sol-" + str(self.solution_counter) + ".t
        f.write("Initial board configuration: " + "".join(str(x) for x in sel
        f.write("\n")
        f.write(str(np.array([*self.roots[self.solution_counter - 1].state]).
        f.write("\n")
        #f.write("Car fuel available: " + str(self.roots[self.solution_counte
        f.write("Car fuel available: ")
        fuel_set = set(self.solutions[self.solution_counter - 1].state)
        for elem in fuel_set:
            if(elem != "."):
                f.write(elem + ":" + str(self.roots[self.solution_counter - 1
        f.write("\n\n")
        if(not self.solution_found):
            f.write("Sorry, could not solve the puzzle as specified\n")
            f.write("Error: no solution found\n")
            f.write("\n")
        f.write("Runtime: " + str(self.solution_runtime) + "\n")
        if(not self.solution_found):
```

```
                self.sheet["D" + str(self.sheet_row_counter)] = "NA"
                self.sheet_row_counter += 1
                self.solution_found = True
                return
            self.sheet["D" + str(self.sheet_row_counter)] = self.solution_path_le
            self.sheet_row_counter += 1
            f.write("Search path length: " + str(self.search_path_length) + "\n")
            f.write("Solution path length: " + str(self.solution_path_length) + "
            f.write("Solution path: " + self.solution_path.replace("\n", "|") + "
            f.write("\n")
            temp_node = Node(self.roots[self.solution_counter - 1].state)
            for single_move in self.solution_path.split("\n"):
                if(single_move != ""):
                    car_dir_step = [x for x in single_move.split(" ") if x != ""]
                    f.write(single_move + "\t\t" + move(temp_node, car_dir_step[0
                    for car, fuel in temp_node.fuel.items():
                        if(fuel != 100):
                            f.write(car + str(fuel) + " ")
                    f.write("\n")
            f.write("\n")
            for car, fuel in self.solutions[self.solution_counter - 1].fuel.items
                if(fuel != 100):
                    f.write(car + str(fuel) + " ")
            f.write("\n" + str(np.array([*self.solutions[self.solution_counter -
        self.solution_found = True
        return
```

2.8 BONUS

AdmissibilityChecker class was created as a bonus to verify using code whether a heuristic is admissible or not. Mind that the checker takes quite a while to verify the admissibility of the search path is long enough, since every call to the calculateHeuristicValue, a lowest_cost solution from the current node configuration is calculated, which, evidently, takes quite some time. Neverhteless, the class is still a subclass of GBF class, therefore, inherits all the necessary functions. The only function to redefine is calculateHeuristicValue, which checks the admissibility of the heuristic itself.

Variables:

admissible -- boolean that represents if heuristic is admissible or not

Functions:

calculateHeuristicValue(self, node)

-- call heuristic function passed beforehand to the constructor to calculate and return heuristic value added to the node cost. Beforehand solve given node with UCS and check whether heuristic value is still admissible

In [9]:
```
class AdmissibilityChecker(GBF):
    def __init__(self, root, heuristic):
        self.admissible = True
        super().__init__(root, heuristic)
    def calculateHeuristicValue(self, node):
        heuristic_val = self.heuristic(node)
        ucs = UCS(node)
        sol_path_length = ucs.solve()[1]
        if(heuristic_val > sol_path_length):
            self.admissible = False
        return heuristic_val
```

3. Define Functions

This section will explain the functions that were created and used in order to efficiently solve the puzzles and better structurize the code.

3.1 h1()

h1 is a heuristic function that counts the number of blocking vehicles. It is passed to a GBFS/A algorithm constructor, which afterwards is executed each calculateHeuristicValue call.

In [10]:
```
def h1(node):
    node.state = np.array([*node.state]).reshape(6, 6)
    temp = node.state[2][np.where(node.state[2] == 'A')[0][-1] + 1 : 6]
    node.state = "".join(str(x) for x in node.state.flatten())
    return np.count_nonzero(np.unique(temp) != ".")
```

3.2 h2()

h1 is a heuristic function that counts the number of blocking positions. It is passed to a GBFS/A algorithm constructor, which afterwards is executed each calculateHeuristicValue call.

In [11]:
```
def h2(node):
    node.state = np.array([*node.state]).reshape(6, 6)
    temp = node.state[2][np.where(node.state[2] == 'A')[0][-1] + 1 : 6]
    node.state = "".join(str(x) for x in node.state.flatten())
    return np.count_nonzero(temp != ".")
```

3.3 h3()

h1 is a heuristic function that counts the number of blocking vehicles multiplied by a constant, which was chosen to be five. It is passed to a GBFS/A algorithm constructor, which afterwards is executed each calculateHeuristicValue call.

In [12]:
```
def h3(node):
    return h1(node) * 5
```

3.4 h4()

h1 is a heuristic function that counts the number of vehicles that cannot move. It is passed to a GBFS/A algorithm constructor, which afterwards is executed each calculateHeuristicValue call.

In [13]:
```
def h4(node):
    node.state = np.array([*node.state]).reshape(6, 6)
    vehicles_can_move = set()
    for i in range(0, len(node.state)):
        for j in range(0, len(node.state)):
            horizontal_orientation = True
            if(has_vehicle(node.state, i, j)):
                below_upper_boundary = i - 1 >= 0
                higher_lower_boundary = i + 1 < len(node.state)

                if((below_upper_boundary and node.state[i - 1][j] == node.state[i
                   or (higher_lower_boundary and node.state[i + 1][j] == node.sta
                    horizontal_orientation = False
                if(horizontal_orientation):
                    if(j - 1 >= 0 and not has_vehicle(node.state, i, j - 1)):
                        vehicles_can_move.add(node.state[i][j])
                    if(j + 1 < len(node.state) and not has_vehicle(node.state, i,
                        vehicles_can_move.add(node.state[i][j])
                else:
                    if(i - 1 >= 0 and not has_vehicle(node.state, i - 1, j)):
                        vehicles_can_move.add(node.state[i][j])
                    if(i + 1 < len(node.state) and not has_vehicle(node.state, i
                        vehicles_can_move.add(node.state[i][j])
    node.state = "".join(str(x) for x in node.state.flatten())
    return len(node.lengths) - 1 - len(vehicles_can_move)
```

3.5 has_vehicle()

has_vehicle is an auxiliary function that simply verifies whether there is a vehicle in a matrix at it given indexes i and j.

In [14]:
```
def has_vehicle(matrix, i, j):
    return matrix[i][j] != '.'
```

3.6 getOldNode()

getOldNode is an auxiliary function that returns a node and the index of it in the given_list if this node is exactly the same as the caller of the function; returns none otherwise.

In [15]:
```
def getOldNode(given_list, node):
    for i in range(0, len(given_list)):
        if(given_list[i].state == node.state):
            return given_list[i], i
    return None, None
```

3.7 move()

move is an auxiliary function that returns board configuration based on the current node configuration, car to be moved, a direction to be moved in, and a number of steps for the car to be moved.

In [16]:

```
def move(node, car, direction, num_of_steps):
    node.state = np.array([*node.state]).reshape(6, 6)
    i_of_car = 0
    if direction == "right":
        for i in range(0, len(node.state)):
            if(len(np.where(node.state[i] == car)[0]) != 0):
                j_of_car = np.where(node.state[i] == car)[0][-1]
                i_of_car = i
                break
        step_counter = 0
        next_state_matrix = np.empty_like(node.state)
        next_state_matrix[:] = node.state
        move_vehicle_counter = node.getLength(car)
        temp_j_moving = j_of_car + int(num_of_steps)
        temp_j_removing = j_of_car
        moving_char = car
        if(i_of_car == 2 and j_of_car + int(num_of_steps) == 5 and car != 'A'):
            moving_char = '.'
        while(move_vehicle_counter > 0):
            next_state_matrix[i_of_car][temp_j_removing] = "."
            next_state_matrix[i_of_car][temp_j_moving] = moving_char
            temp_j_moving -= 1
            temp_j_removing -= 1
            move_vehicle_counter -= 1
        node.state = next_state_matrix
        return str(node.getFuel(car) - int(num_of_steps)) + " " + "".join(str(x)
    elif direction == "left":
        for i in range(0, len(node.state)):
            if(len(np.where(node.state[i] == car)[0]) != 0):
                j_of_car = np.where(node.state[i] == car)[0][0]
                i_of_car = i
                break
        step_counter = 0
        next_state_matrix = np.empty_like(node.state)
        next_state_matrix[:] = node.state
        move_vehicle_counter = node.getLength(car)
        temp_j_moving = j_of_car - int(num_of_steps)
        temp_j_removing = j_of_car
        moving_char = car
        if(i_of_car == 2 and j_of_car + int(num_of_steps) == 5 and car != 'A'):
            moving_char = '.'
        while(move_vehicle_counter > 0):
            next_state_matrix[i_of_car][temp_j_removing] = "."
            next_state_matrix[i_of_car][temp_j_moving] = moving_char
            temp_j_moving += 1
            temp_j_removing += 1
            move_vehicle_counter -= 1
        node.state = next_state_matrix
        return str(node.getFuel(car) - int(num_of_steps)) + " " + "".join(str(x)
    elif direction == "down":
        for i in range(0, len(node.state)):
            if(len(np.where(node.state[i] == car)[0]) != 0):
                j_of_car = np.where(node.state[i] == car)[0][0]
                i_of_car = i
        step_counter = 0
        next_state_matrix = np.empty_like(node.state)
```

```
        next_state_matrix[:] = node.state
        move_vehicle_counter = node.getLength(car)
        temp_i_moving = i_of_car + int(num_of_steps)
        temp_i_removing = i_of_car
        moving_char = car
        while(move_vehicle_counter > 0):
            next_state_matrix[temp_i_removing][j_of_car] = "."
            next_state_matrix[temp_i_moving][j_of_car] = moving_char
            temp_i_moving -= 1
            temp_i_removing -= 1
            move_vehicle_counter -= 1
        node.state = next_state_matrix
        return str(node.getFuel(car) - int(num_of_steps)) + " " + "".join(str(x)
    elif direction == "up":
        for i in range(0, len(node.state)):
            if(len(np.where(node.state[i] == car)[0]) != 0):
                j_of_car = np.where(node.state[i] == car)[0][0]
                i_of_car = i
                break
        step_counter = 0
        next_state_matrix = np.empty_like(node.state)
        next_state_matrix[:] = node.state
        move_vehicle_counter = node.getLength(car)
        temp_i_moving = i_of_car - int(num_of_steps)
        temp_i_removing = i_of_car
        moving_char = car
        while(move_vehicle_counter > 0):
            next_state_matrix[temp_i_removing][j_of_car] = "."
            next_state_matrix[temp_i_moving][j_of_car] = moving_char
            temp_i_moving += 1
            temp_i_removing += 1
            move_vehicle_counter -= 1
        node.state = next_state_matrix
        return str(node.getFuel(car) - int(num_of_steps)) + " " + "".join(str(x)
```

4. Solve Puzzles!

This section will create a solution manager previously-described in order to solve the puzzles from an input file.

4.1 SolutionManager

We simply create a solution manager with an input file, initialize the manager, and call solve_and_out method. That way, each algorithm will solve each puzzle in the input file, search and solution files will be created accordingly, as well as a spreadsheet with all the data will be created.

In [17]:
```
solution_manager = SolutionManager("input.txt")
solution_manager.initialize()
solution_manager.solve_and_out()
```

```
------------A[h1]------------
Steps performed: 2000
Steps performed: 4000
A_h1 executed puzzle #50

------------A[h2]------------
Steps performed: 2000
Steps performed: 4000
A_h2 executed puzzle #50

------------A[h3]------------
Steps performed: 2000
A_h3 executed puzzle #50

------------A[h4]------------
Steps performed: 2000
Steps performed: 4000
A_h4 executed puzzle #50
```

4.2 BONUS

We can check the admissibility of each heuristic, however, it will take quite some time to do that since UCS solve is called each calculateHeuristicValue execution.

In [ ]:
```
solution_manager.admissibility_check()
```