AI Capstone Homework 1 Report

0716040 江睿哲

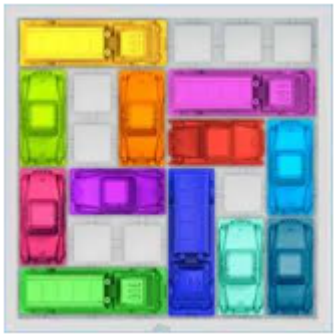
Abstract

This is a report of my rush hour puzzle AI solver, where I implement five searching algorithms: BFS, DFS, IDS, A*, IDA* to solve the game.

Introduction

Rush Hour is a sliding block puzzle invented by Nob Yoshigahara in the 1970s. The game board is a 6x6 grid with cars on it. Each car has its own postion, length, orientation. The final goal of the game is to move the red car out the grid(to the right). Each problem will have different initial grid which comes with different difficulties.

Goal



In this project my goal is to solve the rush hour puzzle with serveral searching algorithms, including some uninformed search such like Breadth First Search, Depth First Search, Iterative Deepening Search, and some informed search such like A* Search, Iterative Deepening A* Search. For uninformed search, I will simply solve the puzzle, while in informed search, I will test some heuristic to find the best one. Last but not least, I will compare these algorithms in different metrics and analyze their pros and cons.

Approach

I finished all the coding with python. All of my code is in the **Appendix** section of this report and also available on my github.
I simply implement the rush hour puzzle with two classes. Car, where I store some basic information of a car such like id, position, length, orientation, and define some method such like how to move them and how to get their occupied area. Gameboard, where I read the input and put all the cars on. For each car, I don't put only one car on the board but several same cars on its occupied area, which can make the search easier.

```
def add_car(self, car):
    if car.orientataion == 'horizontal':
        y = car.start_location[0]
        for x in range(car.start_location[1], car.start_location[1] + car.length):
            self.grid[y][x] = car
    else:
        x = car.start_location[1]
        for y in range(car.start_location[0], car.start_location[0] + car.length):
            self.grid[y][x] = car
```

For all the searching algorithms, I implement them in an another class, solver. I simply implement the frontier as a *list*, and the explored set as a *set* for doing graph search. For IDS and IDA*, the explored set is implement by a *dictionary*, where I can keep track of the minimum depth and score.

Heuristics

For A* and IDA*, heuristics is needed. In addition of the heuristic function which the teacher offered, I also design two more heuristic functions.

Blocking Heuristic

The number of cars directly blocking the way of the red car to the exit.

Distance Heuristic

The distance of the red car from the exit.

Hybrid Heuristic

Its a hybrid version of the above two heuristcs, that is, the number of cars directly blocking the way of the red car to the exit plus the distance of the red car from the exit.

Results

Heuristics

As I mentioned above, in addition of blocking heuristics, I design two more heuristic functions. To tell which heuristic fuction is the best, I tested them on A* and sum up the number of expanded nodes after running several input files:

Total number of expanded nodes after running A* on L01.txt L02.txt L03.txt L04.txt:

Blocking Heuristic	Distance Heuristic	Hybrid Heuristic
3208	7254	2459

It shows that the **hybrid heuristic** is the best since it has the least number of expand nodes. So I choose **hybrid heuristic** for A* and IDA*.

Searching

To compare the five searching algorithms, I used four metrics: number of steps to solution, number of expanded nodes, peak memory usage and total execution time. For number of steps, it's the depth of the search. For number of expanded nodes, it's simply the length of the explored set. As for tracing mememory usage and execution time, I used python standard library *tracemalloc* and *time*. I found that using *tracemalloc* will slow down the program. However, I'm only comparing the relative speed of the five algorithm, so it's fine to put them all in the program.
These are the results running on different input files:

Report of Problem L02.txt:

	BFS	DFS	IDS	A*	IDA*
Number of steps	14	2465	14	14	14
Number of expanded nodes	3390	10473	2953	1645	1645
Peak memory usage(MB)	4.1	171.73	0.98	3.29	3.23
Execution time(sec)	34.24	33.5	569.08	25.52	76.01

Report of Problem L10.txt:

	BFS	DFS	IDS	A*	IDA*
Number of steps	32	491	32	32	32
Number of expanded nodes	2310	1850	2067	1744	1744
Peak memory usage(MB)	2.0	16.48	1.26	2.29	2.26
Execution time(sec)	24.79	6.69	1758.94	28.0	366.68

Report of Problem L20.txt:

	BFS	DFS	IDS	A*	IDA*
Number of steps	18	264	18	18	18
Number of expanded nodes	2393	1111	1915	922	922
Peak memory usage(MB)	3.27	6.52	0.74	2.36	2.37
Execution time(sec)	19.37	5.72	252.11	10.67	34.15

Report of Problem L31.txt:

	BFS	DFS	IDS	A*	IDA*
Number of steps	69	816	69	69	69
Number of expanded nodes	3998	2586	3886	3837	3837
Peak memory usage(MB)	2.17	24.88	1.66	2.75	2.74
Execution time(sec)	38.88	9.88	16142.48	65.3	1659.65

The comparision between each algorithm is mentioned in the **Observation and Conclusion** section.

Observation and Conclusion

1. DFS is the fastest algorithm (O(b^m)), but the solution isn't optimal.
2. BFS has the optimal solution but it use a lot of memory (O(b^d)).
3. IDS surely can use less memory (O(bd)) while getting the optimal solution, but it costs a huge amount of time (O(b^d)).
4. IDA* costs more time to run but only has a similar performance compared to A*, which is a little bit weird.
5. With appropriate heuristics, searching can become easier since the number of expanded nodes is less while also getting the optimal solution.

Discussion

1. How to implement the explored set if you want to do graph search? Is the extra time and space worth it?
   Ans: I only did graph search in this project, and the approach is simply having a explored set to get track of the same states.

2. Can you improve on the provided heuristic function, or devise new ones? If you do, be sure to make comparisons of how they affect the evaluation metrics.
   Ans: I design two more heuristic function, distance heuristic and hybrid heuristic, and it comes out that hybrid heuristc has the best performance since it have the least number of expanded nodes(see **Results-Heuristics** section).

What I Learned

1. How to implement several searching algorithms in games like rush hour puzzle.
2. How to analyze these algorithms with different metrics.

Future Works

1. Implement GUI interface.
2. Implement automatic puzzle generator with different difficulties.
3. Improve IDS and IDA* speed.
4. Find out why IDA* costs more time than A* but only has a similar preformance with A*.

Reference

[1] https://en.wikipedia.org/wiki/Rush_Hour_(puzzle)
[2] https://github.com/CirXe0N/RushHourSolver

Appendix

All of the codes are written in python 3.8, and they are availible on my github.

How to run code

Download all the codes and files on my github and run:

```
python3 Rush_Hour_Puzzle.py
```

```python
# car.py
class Car():
    def __init__(self, id, y, x, length, orientataion):
        self.id = id
        self.start_location = [y, x]
        self.length = length
        self.orientataion = orientataion
    def move_forward(self):
        if self.orientataion == 'horizontal':
            self.start_location[1] += 1
        else:
            self.start_location[0] += 1
    def move_backward(self):
        if self.orientataion == 'horizontal':
            self.start_location[1] -= 1
        else:
            self.start_location[0] -= 1
    def get_id(self):
        return self.id
    def get_start_location(self):
        return self.start_location
    def get_end_location(self):
        if self.orientataion == 'horizontal':
            return [self.start_location[0], self.start_location[1] + self.length - 1]
        else:
            return [self.start_location[0] + self.length - 1, self.start_location[1]]
    def get_ocuupied_locations(self):
        self.occupied_location = []
        if self.orientataion == 'horizontal':
            y = self.start_location[0]
            for x in range(self.start_location[1], self.start_location[1] + self.length
                self.occupied_location.append([y, x])
        else:
            x = self.start_location[1]
            for y in range(self.start_location[0], self.start_location[0] + self.length
                self.occupied_location.append([y, x])
        return self.occupied_location
    def get_orientation(self):
        return self.orientataion
    def __repr__(self):
        return self.id
```

```
#gameboard.py
from car import Car
class GameBoard():
    def __init__(self, height, width):
        self.grid = []
        self.height = height
        self.width = width
        self.generate_grid()
        self.cars = {}
    def generate_grid(self):
        for i in range(self.height):
            self.grid.append([])
            for j in range(self.width):
                self.grid[i].append('.')
    def read_board(self, filename):
        f = open(filename, 'r')
        content = f.readlines()
        f.close()
        data = []
        for line in content:
            data.append(line.split())
        for i in range(len(data)):
            orientataion = ''
            if data[i][4] == '1':
                orientataion = 'horizontal'
            else:
                orientataion = 'vertical'
            car = Car(data[i][0], int(data[i][1]), int(data[i][2]), int(data[i][3]), or
            self.add_car(car)
    def add_car(self, car):
        if car.orientataion == 'horizontal':
            y = car.start_location[0]
            for x in range(car.start_location[1], car.start_location[1] + car.length):
                self.grid[y][x] = car
        else:
            x = car.start_location[1]
            for y in range(car.start_location[0], car.start_location[0] + car.length):
                self.grid[y][x] = car
    def get_height(self):
        return self.height
    def get_width(self):
        return self.width
    def get_grid(self):
        return self.grid
    def __str__(self):
        string = 'Game Board:\n'
        for i in range(self.height):
            for j in range(self.width):
                string += '{:^3s}'.format((str(self.grid[i][j])))
            string += '\n'
        return string
```

```
#solver.py
import copy
import tracemalloc
import time
class Solver():
    def __init__(self, board):
        self.game_board = board
        self.solotion = ''
    def BFS(self):
        tracemalloc.start()
        start_time = time.time()
        self.solotion = 'Solution <car_index, new_row, new_column>: \n'
        grid = self.game_board.get_grid()
        explored = set()
        frontier = [[[], grid, 0]]
        explored.add(hash(str(grid)))
        while len(frontier)>0:
            moves, grid, step_num = frontier.pop(0)
            if self.is_solved(grid):
                for step in moves:
                    self.solotion += '< '
                    self.solotion += str(step[0])
                    self.solotion += ', '
                    self.solotion += str(step[1][0])
                    self.solotion += ', '
                    self.solotion += str(step[1][1])
                    self.solotion += ' >'
                    self.solotion += '\n'
                current, peak = tracemalloc.get_traced_memory()
                tracemalloc.stop()
                return [self.solotion, len(explored), peak/10**6, time.time()-start_time
            for new_moves, new_grid in self.get_states(grid):
                if hash(str(new_grid)) not in explored:
                    frontier.append([moves + new_moves, new_grid, step_num + 1])
                    explored.add(hash(str(new_grid)))
        return 'No Solution'
    def DFS(self):
        tracemalloc.start()
        start_time = time.time()
        self.solotion = 'Solution <car_index, new_row, new_column>: \n'
        grid = self.game_board.get_grid()
        explored = set()
        frontier = [[[], grid, 0]]
        explored.add(hash(str(grid)))
        while len(frontier)>0:
            moves, grid, step_num = frontier[-1]
            if self.is_solved(grid):
                for step in moves:
                    self.solotion += '< '
                    self.solotion += str(step[0])
                    self.solotion += ', '
```

```
                self.solotion += str(step[1][0])
                self.solotion += ', '
                self.solotion += str(step[1][1])
                self.solotion += ' >'
                self.solotion += '\n'
            current, peak = tracemalloc.get_traced_memory()
            tracemalloc.stop()
            return [self.solotion, len(explored), peak/10**6, time.time()-start_time
        states = self.get_states(grid)
        flag = True
        for new_moves, new_grid in states:
            if hash(str(new_grid)) not in explored:
                frontier.append([moves + new_moves, new_grid, step_num + 1])
                explored.add(hash(str(new_grid)))
                flag = False
        if flag:
            del frontier[-1]
    return 'No Solution'
def IDS(self):
    tracemalloc.start()
    start_time = time.time()
    max_depth = 0
    while True:
        self.solotion = 'Solution <car_index, new_row, new_column>: \n'
        grid = self.game_board.get_grid()
        explored = {}
        frontier = [[[], grid, 0, 0]]
        explored[hash(str(grid))] = 0
        while len(frontier) > 0:
            moves, grid , depth, step_num = frontier[-1]
            if self.is_solved(grid):
                for step in moves:
                    self.solotion += '< '
                    self.solotion += str(step[0])
                    self.solotion += ', '
                    self.solotion += str(step[1][0])
                    self.solotion += ', '
                    self.solotion += str(step[1][1])
                    self.solotion += ' >'
                    self.solotion += '\n'
                current, peak = tracemalloc.get_traced_memory()
                tracemalloc.stop()
                return [self.solotion, len(explored), peak/10**6, time.time()-start_
            if depth > max_depth:
                del frontier[-1]
                continue
            states = self.get_states(grid)
            flag = True
            for new_moves, new_grid in states:
                if hash(str(new_grid)) not in explored:
                    frontier.append([moves + new_moves, new_grid, (depth + 1), step_
```

```
                    explored[hash(str(new_grid))] = depth+1
                    flag = False
                elif depth+1 < explored[hash(str(new_grid))]:
                    frontier.append([moves + new_moves, new_grid, (depth + 1), step_
                    explored[hash(str(new_grid))] = depth+1
                    flag = False
            if flag:
                del frontier[-1]
        max_depth += 1
    return 'No Solution'
def A_star(self):
    tracemalloc.start()
    start_time = time.time()
    self.solotion = 'Solution <car_index, new_row, new_column>: \n'
    grid = self.game_board.get_grid()
    h_score = self.hybrid_heuristic_score(grid)
    f_score = 0 + h_score
    explored = set()
    frontier = [[[], grid, f_score, 0]]
    explored.add(hash(str(grid)))
    while len(frontier) > 0:
        index = self.min_score_index(frontier)
        moves, grid, f_score, step_num = frontier[index]
        if self.is_solved(grid):
            for step in moves:
                self.solotion += '< '
                self.solotion += str(step[0])
                self.solotion += ', '
                self.solotion += str(step[1][0])
                self.solotion += ', '
                self.solotion += str(step[1][1])
                self.solotion += ' >'
                self.solotion += '\n'
            current, peak = tracemalloc.get_traced_memory()
            tracemalloc.stop()
            return [self.solotion, len(explored), peak/10**6, time.time()-start_time
        states = self.get_states(grid)
        flag = True
        for new_moves, new_grid in states:
            new_h_score = self.hybrid_heuristic_score(new_grid)
            new_g_score = step_num + 1
            new_f_score = new_g_score + new_h_score
            if hash(str(new_grid)) not in explored:
                frontier.append([moves + new_moves, new_grid, new_f_score, step_num
                explored.add(hash(str(new_grid)))
                flag = False
        if flag:
            del frontier[index]
    return "No Solution"
def IDA_star(self):
    tracemalloc.start()
```

```
    start_time = time.time()
    max_score = 0
    while True:
        self.solotion = 'Solution <car_index, new_row, new_column>: \n'
        grid = self.game_board.get_grid()
        h_score = self.hybrid_heuristic_score(grid)
        f_score = 0 + h_score
        explored = {}
        frontier = [[[], grid, f_score, 0]]
        explored[hash(str(grid))] = f_score
        while len(frontier) > 0:
            index = self.min_score_index(frontier)
            moves, grid, f_score, step_num = frontier[index]
            if self.is_solved(grid):
                for step in moves:
                    self.solotion += '< '
                    self.solotion += str(step[0])
                    self.solotion += ', '
                    self.solotion += str(step[1][0])
                    self.solotion += ', '
                    self.solotion += str(step[1][1])
                    self.solotion += ' >'
                    self.solotion += '\n'
                current, peak = tracemalloc.get_traced_memory()
                tracemalloc.stop()
                return [self.solotion, len(explored), peak/10**6, time.time()-start_
            if f_score > max_score:
                del frontier[index]
                continue
            states = self.get_states(grid)
            flag = True
            for new_moves, new_grid in states:
                new_h_score = self.hybrid_heuristic_score(new_grid)
                new_g_score = step_num + 1
                new_f_score = new_g_score + new_h_score
                if hash(str(new_grid)) not in explored:
                    frontier.append([moves + new_moves, new_grid, new_f_score, step_
                    explored[hash(str(new_grid))] = new_f_score
                    flag = False
                elif new_f_score < explored[hash(str(new_grid))]:
                    frontier.append([moves + new_moves, new_grid, new_f_score, step_
                    explored[hash(str(new_grid))] = new_f_score
                    flag = False
            if flag:
                del frontier[index]
        max_score += 1
    return 'No Solution'
def get_states(self, grid):
    states = []
    for i in range(self.game_board.get_height()):
        for j in range(self.game_board.get_width()):
```

```
            car = grid[i][j]
            if car != '.':
                for direction in ['Forward', 'Backward']:
                    if self.is_movable(car, direction, grid):
                        new_grid = copy.deepcopy(grid)
                        new_car = copy.deepcopy(grid[i][j])
                        if direction == 'Forward':
                            new_car.move_forward()
                        if direction == 'Backward':
                            new_car.move_backward()
                        old_locations = car.get_ocuupied_locations()
                        new_locations = new_car.get_ocuupied_locations()
                        new_grid = self.update_grid(new_grid, new_car, old_locations
                        states.append([[[car, new_car.get_start_location()]], new_gr
    return states
def is_movable(self, car, direction, grid):
    if car.get_orientation() == 'horizontal':
        if direction == 'Forward':
            end = car.get_end_location()
            x = end[1] + 1
            y = end[0]
            if x < self.game_board.get_width() and grid[y][x] == '.':
                return True
            else:
                return False
        if direction == 'Backward':
            start = car.get_start_location()
            x = start[1] - 1
            y = start[0]
            if x > -1 and grid[y][x] == '.':
                return True
            else:
                return False
    if car.get_orientation() == 'vertical':
        if direction == 'Forward':
            end = car.get_end_location()
            x = end[1]
            y = end[0] + 1
            if y < self.game_board.get_height() and grid[y][x] == '.':
                return True
            else:
                return False
        if direction == 'Backward':
            start = car.get_start_location()
            x = start[1]
            y = start[0] - 1
            if y > -1 and grid[y][x] == '.':
                return True
            else:
                return False
def update_grid(self, grid, car, old_locations, new_locations):
```

```
    for i in old_locations:
        grid[i[0]][i[1]] = '.'
    for i in new_locations:
        grid[i[0]][i[1]] = car
    return grid
def is_solved(self, grid):
    if grid[2][4] != '.' and grid[2][5] != '.':
        if grid[2][4].id == '0' and grid[2][5].id == '0':
            return True
    else:
        return False
def blocking_heuristic_score(self, grid):
    i = 0
    counter = 0
    while i < len(grid):
        if grid[2][i] != '.' and grid[2][i].get_id() == '0':
            i += 1
            break
        i += 1
    while i < len(grid):
        if grid[2][i] != '.':
            counter += 1
        i += 1
    return counter
def distance_heuristic_score(self, grid):
    i = 0
    for i in range(len(grid)):
        if grid[2][i] != '.' and grid[2][i].get_id() == '0':
            break
    return len(grid) - i - 2
def hybrid_heuristic_score(self, grid):
    return self.blocking_heuristic_score(grid) + self.distance_heuristic_score(grid
def min_score_index(self, frontier):
    min = 100
    index = -1
    for i in range(len(frontier)):
        if frontier[i][2] < min:
            min = frontier[i][2]
            index = i
    return index
```

```python
#Rush_Hour_Puzzle.py (main.py)
from gameboard import GameBoard
from solver import Solver
from prettytable import PrettyTable

algo = int(input('Please select an algorithm to solve the game\n(1 for BFS, 2 for DFS, 3

input_file = input('Please input the file name: ')

gameboard = GameBoard(6, 6)
gameboard.read_board('tests/' + input_file)

solver = Solver(gameboard)
if algo == 1:
    solution, node_num, memory_usage, execution_time, step_num = solver.BFS()
elif algo == 2:
    solution, node_num, memory_usage, execution_time, step_num = solver.DFS()
elif algo == 3:
    solution, node_num, memory_usage, execution_time, step_num = solver.IDS()
elif algo == 4:
    solution, node_num, memory_usage, execution_time, step_num = solver.A_star()
elif algo == 5:
    solution, node_num, memory_usage, execution_time, step_num = solver.IDA_star()

print('Problem: ' + input_file)
print(gameboard)
if algo == 1:
    print('Algorithm: BFS')
elif algo == 2:
    print('Algorithm: DFS')
elif algo == 3:
    print('Algorithm: IDS')
elif algo == 4:
    print('Algorithm: A*')
elif algo == 5:
    print('Algorithm: IDA*')

print(solution)
print('Number of steps: %d' % step_num)
print('Number of expanded nodes: %d' % node_num)
print('Peak memory usage: %.2f MB' % memory_usage)
print('Total execution time: %.2f sec' % execution_time)
```