rush_hour

December 26, 2024

1 INF421 - Unblock me

Elliot THOREL & Hubert LEROUX

TODO

- regarder si la table de hachage est uniformément rempli est bien remplie grâce à max_items_in_one_bucket
- mise en valeur des résultats
- regarder si on n'a pas trop de lignes de code vis à vis de ce qui est attendu -> faire un choix
- uniformiser français / anglais
- faire les questions d'études de complexité
- supprimer seen et se servir seulement de antécédent

1.1 Introduction

[1]:
```
import os
import queue
import heapq
import time
from matplotlib import pyplot as plt

examples_dir = os.path.join("./ExRushHour", "ExRushHour")
```

1.2 Setting up the game

Le jeu est encodé selon le format suivant :

```
6
8
1 h 2 2 3
2 h 2 1 1
3 h 2 5 5
4 h 3 3 6
5 v 3 6 1
```

Question 1 - Check if the configuration is valid and initializes a rush hour game.

To check if the game is valid, we will put the each car on a board one after another. If all the cars can fit, then it is a valide configuration. Therefore we will create during the process a board that

will be useful later. Void cases will be set to 0. Additionnaly, we will put our cars in a dictionnary list.

[2]:
```
def get_input():
    """
    Get the input from console
    :return: the input
    @param n: the size of the grid
    @param nb_cars: the number of cars
    @param cars: the list of cars (dict list)
    """
    n = int(input())
    nb_cars = int(input())
    cars = []
    for _ in range(nb_cars):
        line = input().split()
        cars.append({"id_car" : int(line[0]), "orientation" : line[1], "length":
 ↪int(line[2]), "x_topleft" : int(line[3])-1, "y_topleft" : int(line[4])-1})

    return n, cars # pas besoin de renvoyer nb_cars, on peut le calculer avec␣
 ↪len(cars)


def get_input_from_file(file):
    """
    Get the input from console
    :return: the input
    @param n: the size of the grid
    @param nb_cars: the number of cars
    @param cars: the list of cars (dict list)
    """
    with open(file, 'r') as f:
        n = int(f.readline().strip())
        nb_cars = int(f.readline().strip())
        cars = []
        for _ in range(nb_cars):
            line = f.readline().strip().split()
            cars.append({"id_car" : int(line[0]), "orientation" : line[1],␣
 ↪"length":int(line[2]), "x_topleft" : int(line[3])-1, "y_topleft" :␣
 ↪int(line[4])-1})

    return n, cars
```

[3]:
```
def is_point_in_grid(point, n):
    """
    Check if a point is in the grid
    """
```

```
    return point[0] >= 0 and point[0] < n and point[1] >= 0 and point[1] < n

def set_game(n, cars):
    """
    Set the game
    :param n: the size of the grid
    :param nb_cars: the number of cars
    :param cars: the list of cars (dict list)
    :return: the grid
    """
    def add_car(n, grid, car):
        """
        Add a car to the grid
        :param grid: the grid
        :param car: the car to add
        :return: True if the car can be added, False otherwise
        """
        if not is_point_in_grid((car["x_topleft"], car["y_topleft"]), n):
            return False # le premier point n'est même pas dans la grille, pas␣
 ↪besoin d'aller plus loin

        # On différencie en fonction de l'orientation de la voiture
        if car["orientation"] == "h": # horizontal
            if  not is_point_in_grid((car["x_topleft"] + car["length"] - 1,␣
 ↪car["y_topleft"]), n):
                return False # la voiture ne rentrera pas
            for i in range(car["length"]):
                if grid[car["y_topleft"]][car["x_topleft"] + i] != 0: # there␣
 ↪is already a car
                    return False
                grid[car["y_topleft"]][car["x_topleft"] + i] = car["id_car"]

        else: # the car is vertical
            if  not is_point_in_grid((car["x_topleft"], car["y_topleft"] +␣
 ↪car["length"] - 1), n):
                return False
            for i in range(car["length"]):
                if grid[car["y_topleft"] + i][car["x_topleft"]] != 0: # there␣
 ↪is already a car
                    return False
                grid[car["y_topleft"] + i][car["x_topleft"]] = car["id_car"]
        return True # the car was set correctly


    grid = [[0 for _ in range(n)] for _ in range(n)]
    for car in cars:
        possible = add_car(n, grid, car)
```

```
        if not possible:
            return False, None
    return True, grid


# TEST
set_game(*get_input_from_file(os.path.join(examples_dir, "GameP01.txt")))
```

[3]:
```
(True,
 [[3, 3, 0, 0, 0, 8],
  [2, 0, 0, 4, 0, 8],
  [2, 1, 1, 4, 0, 8],
  [2, 0, 0, 4, 0, 0],
  [6, 0, 0, 0, 7, 7],
  [6, 0, 5, 5, 5, 0]])
```

Question 2 - Affichage de la grille

[4]:
```
def show_grid(grid):
    """
    Show the grid
    :param grid: the grid
    """
    max_id_car = max(max(line) for line in grid)
    nb_digits = len(str(max_id_car))
    for line in grid:
        print('|' + '|'.join('%0*d' % (nb_digits, cell) if cell != 0 else␣
 ↪'_'*nb_digits for cell in line) + '|')

# TEST
n, cars = get_input_from_file(os.path.join(examples_dir, "GameP01.txt"))
possible, grid = set_game(n, cars)
if possible:
    show_grid(grid)
```

```
|3|3|_|_|_|8|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|2|_|_|4|_|_|
|6|_|_|_|7|7|
|6|_|5|5|5|_|
```

1.3 Solving the game

1.3.1 Computing the length of a shortest solution

Question 3 - Brute force algorithm

Nous pour trouver la solution, nous allons parcourir bêtement l'ensemble des solutions à l'aide d'un parcours en largeur (BFS). Nous aurons donc besoin d'une file de priorité (FIFO) et d'un set

(implémenté sous forme de table de hachage) pour marqué les configurations déjà vues.

Algorithm 1: Brute force algorithm

```
Data: n, nb_cars, cars
Result: True if the red car can exit, False otherwise
file ← FIFO([cars]) ;
seen ← ∅ ;
while file ≠ ∅ do
    cars ← file.pop() ;
    if Winning_configuration(cars) then
        return True ;
    end
    if cars ∉ seen then
        seen.add(car) ;
        for next_car ∈ car.next_possible_moves() do
            if cars ∉ seen then
                file.push(next_car) ;
            end
        end
    end
end
return False ;
```

Question 4 - Get next moves

Pour déterminer les états accessibles, on peut utiliser : - soit la liste des voitures - soit la grille

L'avantage de la grille est qu'il est plus facile de voir directement les cases accessibles. L'avantage de la liste de voitures est qu'il est plus facile de la modifier. Nous allons stocker les listes de voitures dans la FIFO, mais à chaque fois nous créerons la grille de toute façon pour vérifier que la configuration est réglementaire.

Pour chaque voiture, pour chaque direction, pour chaque delta possible dans cette direction on ajoute la configuration. Pour savoir si la configuration est possible on utilise `set_game` en avançant jusqu'à l'échec. C'est assez gourmand puisque qu'on reteste tout au lieu de ne vérifier qu'une seule case de la grille. Mais de toute façon si on voulait justement utiliser ces grilles il faudrait les recréer à chaque noeud d'un arbre. On est obligé de faire des copies ici puisque qu'on fait un parcours en largeur. Un parcours en profondeur nous aurait permis de travailler sur la même grille et gagner ainsi beaucoup d'espace. Mais on n'aurait pas trouvé forcément le chemin le plus court.

ETUDE DE LA COMPLEXITE

Pistes d'amélioration : - on n'a pas besoin de conserver toutes les informations sur les voitures, seules les coordonnées top-left suffisent pour suivre l'évolution suffisent. Cependant cette amélioration ne fait que changer la constante de la complexité.

[5]:
```python
def copy_cars(cars):
    """
    Copy the cars
    :param cars: the cars
    :return: the copied cars
    """
    # return copy.deepcopy(cars)
    return [car.copy() for car in cars]
```

[6]:
```python
def get_next_moves(n, cars):
    """
    Get the next moves
    :param n: the size of the grid
    :param nb_cars: the number of cars
    :param cars: the list of cars (dict list)
    :param grid: the grid
    :return: the list of next moves
    """
    def get_next_moves_car(n, cars, id_car):
        """
        Get the next moves for a car
        :param grid: the grid
        :param car: the car
        :param id_car: the id of the car
        :return: the list of next moves
        """
        def get_next_moves_car_in_one_direction(n, cars_copy, id_car,␣
↪direction):
            """
            Get the next moves for a car in one direction
            :param grid: the grid
            :param car: the car
            :param id_car: the id of the car
            :param direction: the direction
            :return: the list of next moves
            """
            next_moves_car_one_direction = []

            delta_x, delta_y = direction
            possible = True
            i = 0
            while possible:
                i += 1
                cars_copy[id_car]["x_topleft"] += delta_x
                cars_copy[id_car]["y_topleft"] += delta_y
                possible, _ = set_game(n, cars_copy)
                if possible:
```

```
                    next_moves_car_one_direction.append(copy_cars(cars_copy))

            return next_moves_car_one_direction

        next_moves_car = []

        if cars[id_car]["orientation"] == "h": # horizontal
            next_moves_car.extend(get_next_moves_car_in_one_direction(n,␣
 ↪copy_cars(cars), id_car, (1, 0)))
            next_moves_car.extend(get_next_moves_car_in_one_direction(n,␣
 ↪copy_cars(cars), id_car, (-1, 0)))
        else: # vertical
            next_moves_car.extend(get_next_moves_car_in_one_direction(n,␣
 ↪copy_cars(cars), id_car, (0, 1)))
            next_moves_car.extend(get_next_moves_car_in_one_direction(n,␣
 ↪copy_cars(cars), id_car, (0, -1)))

        return next_moves_car

    next_moves = []
    for car in cars:
        id_car = car["id_car"]-1
        next_moves.extend(get_next_moves_car(n, cars, id_car))

    return next_moves

# TEST
n, cars = get_input_from_file(os.path.join(examples_dir, "GameP01.txt"))
print("== CONFIGURATION INITIALE ==")
_, grid = set_game(n, cars)
show_grid(grid)
print()
for move in get_next_moves(n, cars):
    _, grid = set_game(n, move)
    show_grid(grid)
    print()
```

```
== CONFIGURATION INITIALE ==
|3|3|_|_|_|8|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|2|_|_|4|_|_|
|6|_|_|_|7|7|
|6|_|5|5|5|_|

|_|3|3|_|_|8|
|2|_|_|4|_|8|
```

```
|2|1|1|4|_|8|
|2|_|_|4|_|_|
|6|_|_|_|7|7|
|6|_|5|5|5|_|

|_|_|3|3|_|8|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|2|_|_|4|_|_|
|6|_|_|_|7|7|
|6|_|5|5|5|_|

|_|_|_|3|3|8|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|2|_|_|4|_|_|
|6|_|_|_|7|7|
|6|_|5|5|5|_|

|3|3|_|_|_|8|
|2|_|_|_|_|8|
|2|1|1|4|_|8|
|2|_|_|4|_|_|
|6|_|_|4|7|7|
|6|_|5|5|5|_|

|3|3|_|4|_|8|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|2|_|_|_|_|_|
|6|_|_|_|7|7|
|6|_|5|5|5|_|

|3|3|_|_|_|8|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|2|_|_|4|_|_|
|6|_|_|_|7|7|
|6|_|_|5|5|5|

|3|3|_|_|_|8|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|2|_|_|4|_|_|
|6|_|_|_|7|7|
|6|5|5|5|_|_|

|3|3|_|_|_|8|
```

```
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|2|_|_|4|_|_|
|6|_|_|7|7|_|
|6|_|5|5|5|_|

|3|3|_|_|_|8|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|2|_|_|4|_|_|
|6|_|7|7|_|_|
|6|_|5|5|5|_|

|3|3|_|_|_|8|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|2|_|_|4|_|_|
|6|7|7|_|_|_|
|6|_|5|5|5|_|

|3|3|_|_|_|_|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|2|_|_|4|_|8|
|6|_|_|_|7|7|
|6|_|5|5|5|_|
```

Question 5 - Implementation of the set

Python gives a general implementation with the `set` library. We will implement our own here. We will keep it simple. We will use a fixed size table of hashing.

First let's have a upper bound for the number of possible configuration in order to have an idea of the size of the table. Each car is blocked in his column or lign. Therefore, there are at most `n` possible configurations for each car (we do not bother with its length). So that gives us an upper bound $\mathcal{O}(n^{\texttt{nb\_cars}})$ which his very large : with ten cars and 10 by 10 grid, it's 10^{10}

Faisons plutôt une estimation du nombre d'états que nous comptons explorer. Disons un million, ou plutôt un nombre premier P proche.

Maintenant, il faut réfléchir à une fonction de hachage efficace. C'est-à-dire qu'elle soit injective au maximum et que ses images soient bien réparties.

Pour cela il y a une idée assez simple. Notons p_i le i-ème nombre premier et d_i la distance de cette voiture à la frontière gauche (si elle est horizontale) en haut (si elle est verticale).

$$\texttt{hash} = \prod_{i}^{\texttt{nb\_cars}} p_i^{d_i+1} \quad [P]$$

Pour obtenir les `nb_cars` premiers nombres premiers on peut implémenter le crible d'Erathosthene. Mais ici cela revient à écraser une mouche avec une bombe atomique. Le nombre de voitures n'est pas le facteur limitant dans le problème.

[7]:
```python
def generate_primes(n):
    """
    Generate the first n prime numbers
    :param n: the number of prime numbers to generate
    :return: a list of the first n prime numbers
    """
    primes = []
    candidate = 2
    while len(primes) < n:
        is_prime = True
        for prime in primes:
            if candidate % prime == 0:
                is_prime = False
                break
        if is_prime:
            primes.append(candidate)
        candidate += 1
    return primes

# Example usage
n_primes = generate_primes(10)
print(n_primes)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

[8]:
```python
def distance_top_left(car):
    """
    Get the distance from the top left corner
    :param car: the car
    :return: the distance
    """
    if car["orientation"] == "h":
        return car["x_topleft"]
    else:
        return car["y_topleft"]
```

[9]:
```python
def fast_exp_mod(base, exp, mod):
    """
    Compute (base^exp) % mod using fast exponentiation
    :param base: the base
    :param exp: the exponent
    :param mod: the modulus
    :return: (base^exp) % mod
    """
```

```
    result = 1
    base = base % mod
    while exp > 0:
        if exp % 2 == 1:  # If exp is odd, multiply base with result
            result = (result * base) % mod
        exp = exp >> 1  # Divide exp by 2
        base = (base * base) % mod  # Square the base
    return result

# Example usage
print(fast_exp_mod(2, 10, 1000))  # Output: 24
```

24

[10]:
```
class My_hash_table:
    def __init__(self, nb_cars=10,size=1000003):
        self.size = size
        self.nb_items = 0
        self.max_items_in_one_bucket = 0
        self.primes = generate_primes(nb_cars)
        self.table = [None for _ in range(size)]

    def __len__(self):
        return self.nb_items

    def hash(self, key):
        h = 1
        for car in key:
            d = distance_top_left(car)
            h = (h * fast_exp_mod(self.primes[car["id_car"]-1], d+1, self.
 ↪size)) % self.size
        return h

    def get(self, key):
        h = self.hash(key)
        if self.table[h] is None:
            return None
        for k,v in self.table[h]:
            if key == k:
                return v
        return None

    def replace(self, key, value):
        h = self.hash(key)
        if self.table[h] is None:
            return False
        for i, (k,v) in enumerate(self.table[h]):
```

```
            if key == k:
                self.table[h][i] = (key, value)
                return True
        return False

    def add(self, key, value):
        h = self.hash(key)
        if self.table[h] is None:
            self.table[h] = [(key, value)]
        else:
            self.table[h].append((key, value))
        self.max_items_in_one_bucket = max(self.max_items_in_one_bucket,␣
 ↪len(self.table[h]))
        self.nb_items += 1

    def contains(self, key, value):
        h = self.hash(key)
        if self.table[h] is None:
            return False
        for k,v in self.table[h]:
            if (key, value) == (k, v):
                return True
        return False

    def show_table(self):
        for data in self.table:
            if data is not None:
                print(data)

# TEST
ht = My_hash_table()
ht.add([{"id_car": 1, "orientation": "h", "length": 2, "x_topleft": 0,␣
 ↪"y_topleft": 0}], [{"id_car": 1, "orientation": "h", "length": 2,␣
 ↪"x_topleft": 0, "y_topleft": 0}])
ht.add([{"id_car": 1, "orientation": "h", "length": 2, "x_topleft": 0,␣
 ↪"y_topleft": 0}, {"id_car": 2, "orientation": "h", "length": 2, "x_topleft":␣
 ↪0, "y_topleft": 0}],
          [{"id_car": 1, "orientation": "h", "length": 2, "x_topleft": 0,␣
 ↪"y_topleft": 0}, {"id_car": 2, "orientation": "h", "length": 2, "x_topleft":␣
 ↪0, "y_topleft": 0}])
ht.show_table()
print(ht.contains([{"id_car": 1, "orientation": "h", "length": 2, "x_topleft":␣
 ↪0, "y_topleft": 0}], [{"id_car": 1, "orientation": "h", "length": 2,␣
 ↪"x_topleft": 0, "y_topleft": 0}]))  # Output: True
```

```
[([{'id_car': 1, 'orientation': 'h', 'length': 2, 'x_topleft': 0, 'y_topleft':
0}], [{'id_car': 1, 'orientation': 'h', 'length': 2, 'x_topleft': 0,
```

```
'y_topleft': 0}])]
[([{'id_car': 1, 'orientation': 'h', 'length': 2, 'x_topleft': 0, 'y_topleft':
0}, {'id_car': 2, 'orientation': 'h', 'length': 2, 'x_topleft': 0, 'y_topleft':
0}], [{'id_car': 1, 'orientation': 'h', 'length': 2, 'x_topleft': 0,
'y_topleft': 0}, {'id_car': 2, 'orientation': 'h', 'length': 2, 'x_topleft': 0,
'y_topleft': 0}])]
True
```

[11]:
```python
class My_set(My_hash_table):
    def add(self, key):
        super().add(key, key)

    def contains(self, key):
        return super().contains(key, key)


# TEST
s = My_set()
s.add([{"id_car": 1, "orientation": "h", "length": 2, "x_topleft": 0,␣
 ↪"y_topleft": 0}])
s.add([{"id_car": 1, "orientation": "h", "length": 2, "x_topleft": 0,␣
 ↪"y_topleft": 0}, {"id_car": 2, "orientation": "h", "length": 2, "x_topleft":␣
 ↪0, "y_topleft": 0}])
print(s.contains([{"id_car": 1, "orientation": "h", "length": 2, "x_topleft":␣
 ↪0, "y_topleft": 0}]))  # Output: True
```

```
True
```

Question 6 - Implémentation

Il faut avant tout faire une fonction déterminant si la position est gagnante.

[12]:
```python
def is_winning(n, cars):
    return cars[0]["x_topleft"] == n-cars[0]["length"]
```

[13]:
```python
def brute_foce_BFS(n, cars):
    """
    Brute force BFS
    """
    q = queue.Queue()
    q.put((0, cars))

    seen = My_set(len(cars))

    while q:
        depth, cars = q.get()

        if seen.contains(cars): # si déjà traité, ...
            continue # ... on passe à la suite
```

```
        seen.add(cars)

        if is_winning(n, cars):
            print(f"Nb de noeuds explorés : {seen.nb_items:,}")
            return depth

        for next_cars in get_next_moves(n, cars):
            if not seen.contains(next_cars):
                q.put((depth+1, next_cars))

    return -1


# TEST
n, cars = get_input_from_file(os.path.join(examples_dir, "GameP40.txt"))
print(brute_foce_BFS(n, cars))
```

```
Nb de noeuds explorés : 3,025
51
```

1.3.2 Reconstruction of the solution

Question 7

Il faut ajouter une deuxième base de donnée conservant l'antécédent de chaque solution. C'est ici que la structure clef-valeur de notre table de hachage va être utilisé, juqu'alors on utilisait cette structure simplement comme un set et on enregistrait la même chose pour la clef et la valeur. Là la clef va être la configuration actuelle et la valeur celle antécédente.

[14]:
```
def deballe_solution(antecedent, cars):
        solution = []
        while cars is not None:
            solution.append(cars)
            cars = antecedent.get(cars)
        return solution[::-1]
```

[15]:
```
def brute_foce_BFS_with_solution(n, cars):
    """
    Brute force BFS
    """

    q = queue.Queue()
    q.put((0, cars))

    seen = My_set(len(cars))
    antecedent = My_hash_table(len(cars))
    antecedent.add(cars, None) # Celle de départ n'a pas d'antécédant
```

```
    while q:
        depth, cars = q.get()

        if seen.contains(cars): # si déjà traité, ...
            continue # ... on passe à la suite

        seen.add(cars)

        if is_winning(n, cars):
            return len(seen), depth, deballe_solution(antecedent, cars)

        for next_cars in get_next_moves(n, cars):
            if not seen.contains(next_cars):
                antecedent.add(next_cars, cars)
                q.put((depth+1, next_cars))

    return -1, None

# TEST
n, cars = get_input_from_file(os.path.join(examples_dir, "GameP01.txt"))
_, min_dist, solution = brute_foce_BFS_with_solution(n, cars)
print(f"Nombre de coups minimum : {min_dist}")
for cars in solution:
    print()
    show_grid(set_game(n, cars)[1])
```

```
Nombre de coups minimum : 8

|3|3|_|_|_|8|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|2|_|_|4|_|_|
|6|_|_|_|7|7|
|6|_|5|5|5|_|

|_|3|3|_|_|8|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|2|_|_|4|_|_|
|6|_|_|_|7|7|
|6|_|5|5|5|_|

|2|3|3|_|_|8|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|_|_|_|4|_|_|
|6|_|_|_|7|7|
```

```
|6|_|5|5|5|_|

|2|3|3|_|_|8|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|6|_|_|4|_|_|
|6|_|_|_|7|7|
|_|_|5|5|5|_|

|2|3|3|_|_|8|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|6|_|_|4|_|_|
|6|_|_|_|7|7|
|5|5|5|_|_|_|

|2|3|3|_|_|8|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|6|_|_|4|_|_|
|6|7|7|_|_|_|
|5|5|5|_|_|_|

|2|3|3|_|_|8|
|2|_|_|_|_|8|
|2|1|1|_|_|8|
|6|_|_|4|_|_|
|6|7|7|4|_|_|
|5|5|5|4|_|_|

|2|3|3|_|_|_|
|2|_|_|_|_|_|
|2|1|1|_|_|_|
|6|_|_|4|_|8|
|6|7|7|4|_|8|
|5|5|5|4|_|8|

|2|3|3|_|_|_|
|2|_|_|_|_|_|
|2|_|_|_|1|1|
|6|_|_|4|_|8|
|6|7|7|4|_|8|
|5|5|5|4|_|8|
```

1.4 Approach based on heuristics

1.4.1 First steps with heuristics

Nous allons utiliser des tas pour gérer plus simplement les priorités. L'implémentation est assez facile avec des arbres binaires, mais on va utiliser ici celle de python.

Avec une heuristique constante égale à 0, cela revient à utiliser une LIFO si l'ajout est à la racine et une FIFO sinon.

Implémentation des tas

Voici un lien sur l'implémentation d'un tas de priorité.

Cependant, nous ne pouvons pas l'utiliser directement puisqu'il faut une clef de comparaison pour les données, dont nos configurations. Voici une proposition pour enrober les fonctions.

[16]:
```
class My_Cars(tuple):
    def __lt__(self, other):
        return self[0] < other[0] or (self[0] == other[0] and self[1] <␣
 ↪other[1])
```

On ajoute quelques modifications : - Cette fois-ci, si on voit à nouveau une configuration, peut-être que celle-ci a été atteinte en moins de mouvements. Il faut donc changer l'antécédent si c'est le cas - on pourrait utiliser qu'une seule base de données au lieu de seen et antécédent. Elles sont redondantes.

[17]:
```
def deballe_solution2(antecedent, cars):
        solution = []
        while cars is not None:
            solution.append(cars)
            data = antecedent.get(cars)
            if data is None:
                break
            _, cars = data
        return solution[::-1]
```

[18]:
```
def heuristic_BFS(n, cars, heuristic = lambda x, y : 0):
    """
    Brute force BFS
    """
    q = [My_Cars((heuristic(n, cars), 0, cars))]
    heapq.heapify(q)

    seen = My_set(len(cars))
    antecedent = My_hash_table(len(cars))
    antecedent.add(cars, None) # Celle de départ n'a pas d'antécédant

    while q:

        h, dist, cars = heapq.heappop(q)
```

```
        if seen.contains(cars): # si déjà traité, ...
            #! on peut regarder si on est arrivé plus vite à cette configuration
            data = antecedent.get(cars)
            if data is not None and dist < data[0]:
                antecedent.replace(cars, (dist, antecedent.get(cars)[1]))
            continue # ... on passe à la suite

        seen.add(cars) # on le marque

        if is_winning(n, cars):
            solution = deballe_solution2(antecedent, cars)
            return len(seen), len(solution), solution

        for next_cars in get_next_moves(n, cars):
            if not seen.contains(next_cars):
                antecedent.add(next_cars, (dist+1, cars))
                heapq.heappush(q, My_Cars((heuristic(n, next_cars), dist+1,␣
 ↪next_cars)))

    return -1, None

# TEST
n, cars = get_input_from_file(os.path.join(examples_dir, "GameP01.txt"))
nb_explored, min_dist, solution = heuristic_BFS(n, cars)
print(f"Nombre de noeuds explorés : {nb_explored:,}")
print(f"Nombre de coups minimum : {min_dist}")
for cars in solution:
    print()
    show_grid(set_game(n, cars)[1])
```

```
Nombre de noeuds explorés : 1,062
Nombre de coups minimum : 9

|3|3|_|_|_|8|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|2|_|_|4|_|_|
|6|_|_|_|7|7|
|6|_|5|5|5|_|

|_|3|3|_|_|8|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|2|_|_|4|_|_|
|6|_|_|_|7|7|
|6|_|5|5|5|_|
```

```
|2|3|3|_|_|8|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|_|_|_|4|_|_|
|6|_|_|_|7|7|
|6|_|5|5|5|_|

|2|3|3|_|_|8|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|_|_|_|4|_|_|
|6|7|7|_|_|_|
|6|_|5|5|5|_|

|2|3|3|_|_|8|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|6|_|_|4|_|_|
|6|7|7|_|_|_|
|_|_|5|5|5|_|

|2|3|3|_|_|8|
|2|_|_|4|_|8|
|2|1|1|4|_|8|
|6|_|_|4|_|_|
|6|7|7|_|_|_|
|5|5|5|_|_|_|

|2|3|3|_|_|8|
|2|_|_|_|_|8|
|2|1|1|_|_|8|
|6|_|_|4|_|_|
|6|7|7|4|_|_|
|5|5|5|4|_|_|

|2|3|3|_|_|_|
|2|_|_|_|_|_|
|2|1|1|_|_|_|
|6|_|_|4|_|8|
|6|7|7|4|_|8|
|5|5|5|4|_|8|

|2|3|3|_|_|_|
|2|_|_|_|_|_|
|2|_|_|_|1|1|
|6|_|_|4|_|8|
|6|7|7|4|_|8|
|5|5|5|4|_|8|
```

NB. Voir l'implémentation de A* pour quelques corrections. On devrait avoir le même résultat…. avec une heuristique constante nulle.

[19]:
```
def distance_to_goal(n, cars):
    return n-cars[0]["length"]-cars[0]["x_topleft"]
```

[20]:
```
# TEST
n, cars = get_input_from_file(os.path.join(examples_dir, "GameP01.txt"))
nb_explored, min_dist, solution = heuristic_BFS(n, cars, distance_to_goal)
print(f"Nombre de noeuds explorés : {nb_explored:,}")
print(f"Nombre de coups minimum : {min_dist}")
```

```
Nombre de noeuds explorés : 856
Nombre de coups minimum : 10
```

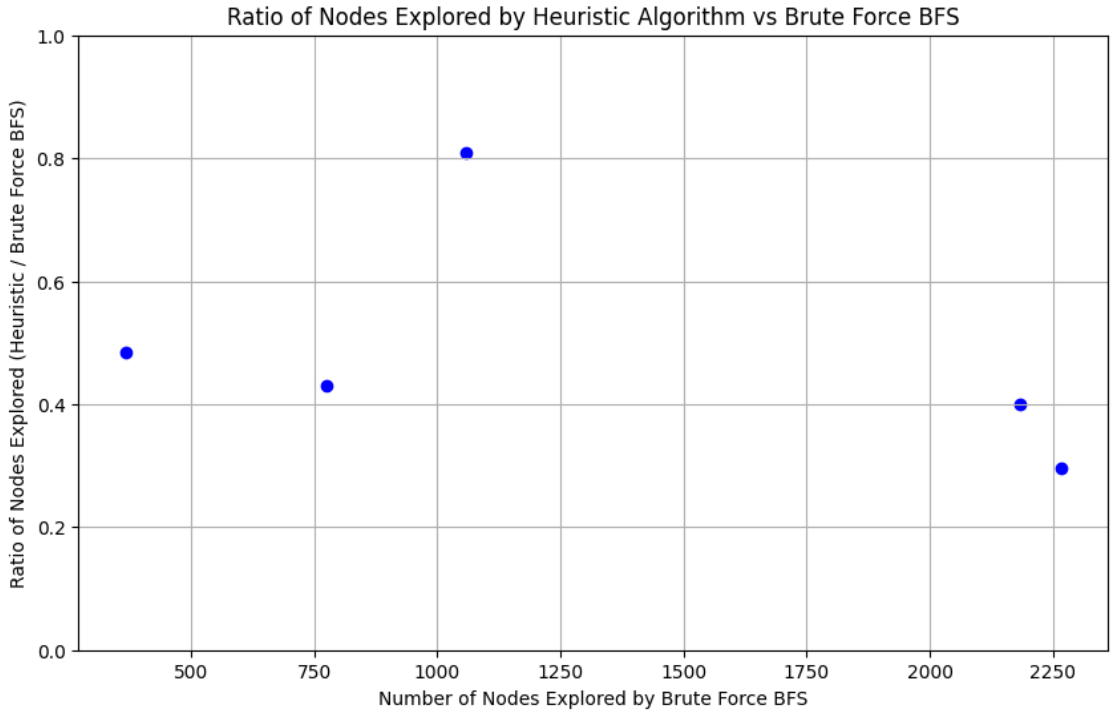
On remarque qu'on a exploré moins de configurations. Cependant, le chemin trouvé n'est pas optimal. Comparons maintenant les deux algorithmes sur l'ensemble du jeu de données.

[21]:
```
heuristic_data = []
brute_foce_BFS_data = []

number_games = 5 # 40 max
for i in range(1,number_games+1):
    print(f"{i:2d}", end=" ")
    start_time = time.time()
    n, cars = get_input_from_file(os.path.join(examples_dir, f"GameP{i:02}.
 ↪txt"))

    start_time_heuristic = time.time()
    nb_explored_h, min_dist_h, _ = heuristic_BFS(n, cars, distance_to_goal)
    heuristic_data.append((nb_explored_h, min_dist_h, n, time.
 ↪time()-start_time_heuristic))

    start_time_brute_force = time.time()
    nb_explored, min_dist, _ = brute_foce_BFS_with_solution(n, cars)
    brute_foce_BFS_data.append((nb_explored, min_dist, n, time.
 ↪time()-start_time_brute_force))

    print(f"{time.time()-start_time:5.2f} s - Done")
```

```
 1  1.27 s - Done
 2  2.11 s - Done
 3  0.61 s - Done
 4  0.40 s - Done
 5  2.64 s - Done
```

Affichons nos résultats sous une forme exploitable. On peut tracer par exemple le ratio du nombre de noeuds explorés grâce à l'algorithme heuristique contre celui du parcours en largeur en fonction du nombre de noeuds explorés par le parcours en largeur.

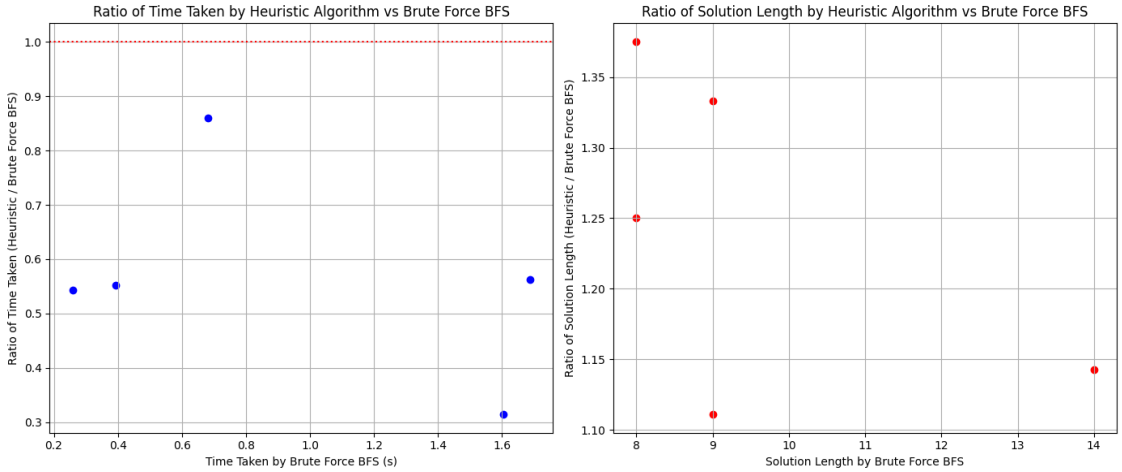
[22]:
```
# Extract data
brute_force_nodes = [data[0] for data in brute_foce_BFS_data]
heuristic_nodes = [data[0] for data in heuristic_data]

# Calculate ratio
ratios = [h / b for h, b in zip(heuristic_nodes, brute_force_nodes)]

# Plot
plt.figure(figsize=(10, 6))
plt.scatter(brute_force_nodes, ratios, marker='o', linestyle='-', color='b')
plt.xlabel('Number of Nodes Explored by Brute Force BFS')
plt.ylabel('Ratio of Nodes Explored (Heuristic / Brute Force BFS)')
plt.ylim(0, 1)
plt.title('Ratio of Nodes Explored by Heuristic Algorithm vs Brute Force BFS')
plt.grid(True)
plt.show()
```



Comme on peut le voir tous les ratios sont bien inférieurs à 1, parfois il explore jusqu'à cinq fois moins de nœuds, mais parfois c'est simplement 20% de moins. Nous n'arrivons pas bien à discerner sur ce graphique dans quel cas l'heuristique est vraiment bonne. Cela doit dépendre de l'arrangement du jeu.

Tentons d'autres graphiques.

[23]:
```
# Extract data
brute_force_times = [data[3] for data in brute_foce_BFS_data]
heuristic_times = [data[3] for data in heuristic_data]
brute_force_lengths = [data[1] for data in brute_foce_BFS_data]
heuristic_lengths = [data[1] for data in heuristic_data]

# Calculate ratios
time_ratios = [h / b for h, b in zip(heuristic_times, brute_force_times)]
length_ratios = [h / b for h, b in zip(heuristic_lengths, brute_force_lengths)]

# Plot time ratios
plt.figure(figsize=(14, 6))

plt.subplot(1, 2, 1)
plt.scatter(brute_force_times, time_ratios, marker='o', linestyle='-',␣
 ↪color='b')
plt.xlabel('Time Taken by Brute Force BFS (s)')
plt.ylabel('Ratio of Time Taken (Heuristic / Brute Force BFS)')
plt.title('Ratio of Time Taken by Heuristic Algorithm vs Brute Force BFS')
plt.axhline(y=1, color='r', linestyle=':')
plt.grid(True)

# Plot length ratios
plt.subplot(1, 2, 2)
plt.scatter(brute_force_lengths, length_ratios, marker='o', linestyle='-',␣
 ↪color='r')
plt.xlabel('Solution Length by Brute Force BFS')
plt.ylabel('Ratio of Solution Length (Heuristic / Brute Force BFS)')
plt.title('Ratio of Solution Length by Heuristic Algorithm vs Brute Force BFS')
# plt.ylim(0,1)
plt.grid(True)

plt.tight_layout()
plt.show()
```



On voit bien à gauche que l'algorithme heuristique est plus rapide en temps également, malgré son implémentation avec un tas. De plus on voit à droite que la solution n'est presque jamais la meilleure, mais ne s'en éloigne pas trop non plus (30% en plus maximum à partir de 20).

1.4.2 New heuristics

Implémentons une fonction pour collecter les données de différentes heuristiques sur les données d'entraînement.

[24]:
```
def get_data_from_different_heuristic(heuristics_list, nb_games=40):
    data = [[] for i in range(len(heuristics_list))]

    for i in range(1,nb_games+1):
        print(f"{i:2d}", end=" ")
        start_time = time.time()
        n, cars = get_input_from_file(os.path.join(examples_dir, f"GameP{i:02}.
↪txt"))

        for i, h in enumerate(heuristics_list):
            start_time_heuristic = time.time()
            nb_explored, min_dist, _ = heuristic_BFS(n, cars, h)

            data[i].append({"nb_explored": nb_explored, "min_dist": min_dist,␣
↪"n": n, "time": time.time()-start_time_heuristic})

        print(f"{time.time()-start_time:5.2f} s - Done")

    return data
```

Tentons une nouvelle heuristique : le nombre de véhicules bloquant le chemin de notre voiture.

[25]:
```
def number_cars_blocking(n, cars):
    """
    Get the number of cars blocking the path of the main car
    Rq :  toutes les voitures bloquantes sont verticales, sinon le jeut n'est pas résoluble.
    """
    return sum(1 for car in cars[1:] if car["y_topleft"] <= cars[0]["y_topleft"]
               and cars[0]["y_topleft"] < car["y_topleft"] + car["length"]
               and cars[0]["x_topleft"]<car["x_topleft"]+car["length"] and car["x_topleft"]<cars[0]["x_topleft"]+cars[0]["length"])

# TEST
n, cars = get_input_from_file(os.path.join(examples_dir, "GameP01.txt"))
print(number_cars_blocking(n, cars))  # Output: 1
```

```
1
```

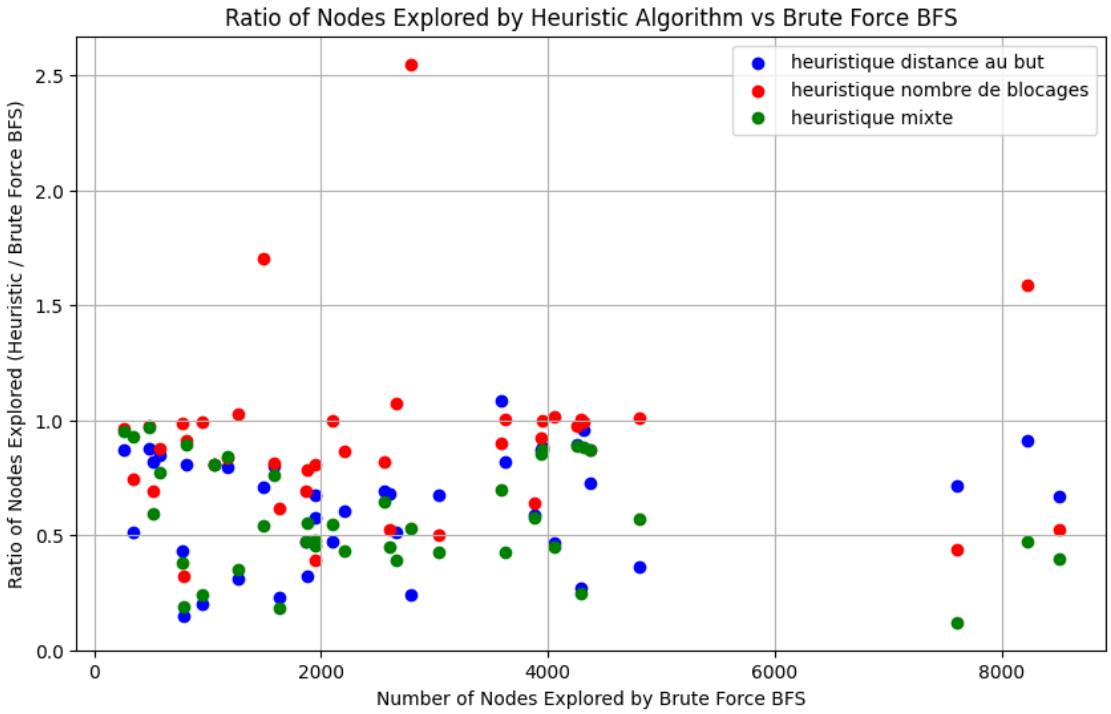
[26]:
```
data = get_data_from_different_heuristic([distance_to_goal,
                                          number_cars_blocking,
                                          lambda n, cars : distance_to_goal(n, cars) + 2*number_cars_blocking(n, cars),
                                          lambda x,y : 0],
                                          40)
```

```
 1  2.38 s - Done
 2  9.63 s - Done
 3  1.11 s - Done
 4  0.79 s - Done
 5  3.94 s - Done
 6  2.67 s - Done
 7  7.33 s - Done
 8  1.96 s - Done
 9  1.24 s - Done
10  3.83 s - Done
11  1.61 s - Done
12  2.01 s - Done
13 13.88 s - Done
14 30.97 s - Done
15  1.51 s - Done
16  6.24 s - Done
17  5.48 s - Done
18  3.22 s - Done
19  1.13 s - Done
20  3.94 s - Done
21  0.73 s - Done
22  8.13 s - Done
23  3.76 s - Done
24 11.09 s - Done
```

```
25 19.75 s - Done
26 10.88 s - Done
27  4.63 s - Done
28  3.70 s - Done
29 13.18 s - Done
30  2.87 s - Done
31 10.08 s - Done
32  1.70 s - Done
33  9.98 s - Done
34 12.06 s - Done
35 10.18 s - Done
36  5.44 s - Done
37  4.39 s - Done
38  7.66 s - Done
39  9.74 s - Done
40  6.01 s - Done
```

[27]:
```python
# Extract data
nb_explored_heuristic_distance = [d["nb_explored"] for d in data[0]]
nb_explored_heuristic_nb_blocking = [d["nb_explored"] for d in data[1]]
nb_explored_heuristic_mixte = [d["nb_explored"] for d in data[2]]
nb_explored_brute = [d["nb_explored"] for d in data[3]]

# Calculate ratio
ratios_distance = [h / b for h, b in zip(nb_explored_heuristic_distance,␣
 ↪nb_explored_brute)]
ratios_nb_blocking = [h / b for h, b in zip(nb_explored_heuristic_nb_blocking,␣
 ↪nb_explored_brute)]
ratios_mixte = [h / b for h, b in zip(nb_explored_heuristic_mixte,␣
 ↪nb_explored_brute)]



# Plot
plt.figure(figsize=(10, 6))
plt.scatter(nb_explored_brute, ratios_distance, marker='o', linestyle='-',␣
 ↪color='b', label="heuristique distance au but")
plt.scatter(nb_explored_brute, ratios_nb_blocking, marker='o', linestyle='-',␣
 ↪color='r', label="heuristique nombre de blocages")
plt.scatter(nb_explored_brute, ratios_mixte, marker='o', linestyle='-',␣
 ↪color='g', label="heuristique mixte")
plt.xlabel('Number of Nodes Explored by Brute Force BFS')
plt.ylabel('Ratio of Nodes Explored (Heuristic / Brute Force BFS)')
plt.title('Ratio of Nodes Explored by Heuristic Algorithm vs Brute Force BFS')
plt.grid(True)
plt.legend()
```

[27]:
```
<matplotlib.legend.Legend at 0x7d49c9e4a2a0>
```



L'heuristique du nombre de voitures bloquantes est clairement moins bonne. Mais l'heuristique mixte est encore meilleur que celle de la distance au but.

Table 1: Terminal set of an individual program in the population. B:Boolean, R:Real or Integer. The upper part of the table contains terminals used both in *Condition* and *Value* trees, while the lower part regards *Condition* trees only.

R=$BlockersLowerBound$	A lower bound on the number of moves required to remove blocking vehicles out of the red car's path
R=$GoalDistance$	Sum of all vehicles' distances to their locations in the deduced-goal board
R=$Hybrid$	Same as $GoalDistance$, but also add number of vehicles between each car and its designated location
R=$\{0.0, 0.1 \ldots, 1.0, 1, \ldots, 9\}$	Numeric terminals
B=$IsMoveToSecluded$	Did the last move taken position the vehicle at a location that no other vehicle can occupy?
B=$IsReleasingMove$	Did the last move made add new possible moves?
R=g	Distance from the initial board
R=$PhaseByDistance$	$g \div (g + DistanceToGoal)$
R=$PhaseByBlockers$	$g \div (g + BlockersLowerBound)$
R=$NumberOfSyblings$	The number of nodes expanded from the parent of the current node
R=$DifficultyLevel$	The difficulty level of the given problem, relative to other problems in the current problem set.