Artificial Intelligence Capstone Hw1

Goal of the experiment

- 透過不同的演算法來解決 **Rush Hour Puzzle**。
- 觀察每個演算法在不同 Level 下所帶來的效能差距，以及所需要花費的計算資源。
- 透過實驗結果來分析各種可能會造成差距的原因，藉以深入了解搜尋演算法。

Project Overview

透過 Python 來實作 BFS、DFS、IDS、A* 以及 IDA* 這五個演算法的 Graph-search。

- Global Variables：
  - Exist_State：透過 **dictionary** 記錄已經出現過的 state。
- Class：
  - Car：用來記錄單台車子的資訊，包含 index、location、length，以及 location。
  - Board_State：用來記錄 Board 當下的狀態，包含每台車子的資訊、Board 還有哪些位置為空格，以及 Board 的 Key (用來當作 Exist_State 的 Index)。
  - PriorityQueue：透過 heapq (heap queue) 來實作 Priority Queue 的功能，包括 push、pop、top、empty，以及 size。用於 A* algorithm 中，以找出在 Queue 中 cost 最小 (也就是 priority 最高) 的 State。
- Functions：
  - ReadFile：將關卡資訊經過處理後，轉換成 Initial State 回傳。
  - main：根據所輸入的參數，來決定要執行的關卡以及演算法，並印出數據及解法。
  - BFS：透過 **Queue** 來實作 BFS algorithm。
  - DFS：透過 **Stack** 來實作 DFS algorithm。(若為單純的 DFS，則會將 depth 限制設為 -1；若為 IDS 所呼叫的 DFS，則會設定 depth 限制為所需)
  - IDS：重複利用 DFS 這個 function 來進行搜索，並傳入深度限制。
  - A*：透過 **Priority Queue** 來實作 A* algorithm。
  - IDA*：與 IDS 作法類似，但是將深度限制改為 cost 限制 (depth + Heuristic function)。而限制的更新方式為上一輪的 state 中，cost 超過限制的最小值。
- Solution 的尋找方式：透過 Predecessor 在 Exist State 中找出移動的方式。

Experiments and Results

The number of explored nodes during search

- 目標：分析在不同演算法以及難度時，Exist States 的總數

	BFS	DFS	IDS	A*	IDA*
L01	1072	713	1065	1069	1060
L11	848	555	834	849	833
L21	261	253	260	260	258
L31	3954	2646	3897	3952	3938
L40	3029	2554	2792	3081	2925

- 結果分析：
  - 從上方表格發現關卡難度的增長，並不代表 explored nodes 的數量就會增加，但是這五種演算法 explored nodes 的增減都有相同的趨勢。這代表 explored nodes 的數量跟每個關卡的 **maximum branching factor** 可能相關（也有可能是受到 depth 的影響）。
  - DFS 在大多數的 Level 下，explored nodes 的數量皆較少。因為 BFS、IDS 及 IDA* 都能找到最佳解，會盡可能去**遍歷所有最短路徑**，增加 explored nodes 的數量。而 DFS 的 explored nodes 數量取決於 maximum branching factor 及 solution depth，且不需要擴張每一個點，因此 explored nodes 的數量較少。
  - A* 在這次實驗中沒有帶來太大的效益，最主要的原因為 **blocking heuristic** 的估計值大多遠小於實際到目標頂點所需要的距離且 Rush Hour Puzzle 為 **unit step costs** 的問題，雖然可以得到最佳解，但是所需要探訪的深度上升，計算的節點量也因此增加。
  - 綜合上述，倘若想要在結果上減少 explored nodes 所花費的空間的話，可以選擇 DFS 來搜尋。或者優化 **heuristic function**，使得預測的距離更接近實際上抵達 goal state 的距離，藉此可以大幅減少 A* 需要探訪的深度與節點數。

The actual time of execution during search and the number of solution steps

- 目標：分析在不同演算法以及難度時，所需要執行的時間 (seconds) 與解答步數
	BFS	DFS	IDS	A*	IDA*
L01	0.61s/16步	0.107s/183步	13.814s/16步	0.597s/16步	13.390s/16步
L11	0.37s/56步	0.095s/186步	89.766s/56步	0.361s/56步	87.974s/56步
L21	0.085s/49步	0.054s/100步	12.961s/49步	0.086s/49步	12.309s/49步
L31	2.39s/69步	0.608s/799步	624.046s/69步	2.391s/69步	607.881s/69步
L40	2.21s/81步	0.830s/597步	497.054s/81步	2.265s/81步	475.921s/81步

- 結果分析：
  - 除了 DFS 沒辦法找到最佳解外，其餘四者皆可找到最短步數解。

- 皆使用 Graph-search，所以倘若地圖有解的情況下，DFS 一定能走到終點。
- 因為問題為 unit step costs，因此 IDS 一定能找到最佳解。
- 發現 A* 總是能找到最佳解，可能原因為我們所使用的 heuristic function 預測的距離會小於目前節點到目標節點的實際距離。
- 從上述五個 cases 可以發現，不一定步數較多，所花費的時間就較長，因為 maximum branching factor 也會影響速度。
- 在速度上，可以發現儘管 DFS 的步數總是多於其他四種演算法，但執行速度卻是最快的，原因為 DFS 的目標是找到一條可行解就好，不需要去優化步數，因此可以大幅減少需要搜索的範圍，藉此能夠降低許多需要執行的時間。
- 理論上來說，若 heuristic function 估計的值較接近實際上的距離的話，A* 的速度應該會快於 BFS。然而在這次實驗上，可以發現兩者的速度沒有太大的差異。因此可以推論 heuristic function 沒有優化太多 A* 的效能，使得 A* 的探訪深度與 BFS 差不多。
- IDS 與 IDA* 兩者所花費的時間遠大於其他三者，而且當 depth 上升時，所花費的時間會大幅上升。這是因為這兩種演算法必須重複跑整個圖很多次，且若當次的限制尚未到達最佳解的最短距離時，就必須遍歷所有在當前限制內的所有路徑，導致執行時間大幅上升。此外，由於這次的問題為 **unit step cost** 且 heuristic function 估計的不好，使得 IDA* 無法有效的提升限制，因此速度才會與 IDS 差異不大。
- 綜合上述，倘若問題不要求最佳解的話，則 DFS 會是最好的選擇。然而若想同時要求速度及最佳解，BFS 會是最簡易的演算法，因為 A* 需要去找到適當的 heuristic function 才能大幅優化，但是往往很難找到合適的 function 去估計。

The maximum number of nodes kept in the memory during search

- 目標：分析在不同演算法以及難度時，所需要在記憶體（Stack, Queue, 或 Priority Queue）中保存最多幾個點
-

	BFS	DFS	IDS	A*	IDA*
L01	159	529	72	161	72
L11	45	321	96	61	92
L21	12	101	57	13	56
L31	175	1585	143	207	142
L40	194	1417	176	238	176

- 結果分析：
  - 理論上來說，DFS 需要保存的點會比 BFS 及 A* 少，因為不需要去記錄所有可能性。然而由於這次是透過 stack 來實作 DFS，在每往更深一層前進時，便會記錄許多不太會被使用到的點，導致 DFS 需要暫時儲存的點數量遠大於 BFS 及 A*。
  - BFS 及 A* 由於會盡可能擴展每個點的所有可能，因此會把所有能抵達的點都記錄在 Queue 中，所以從上方表格可以發現，BFS 及 A* 所暫時記錄的點數量大多都比 IDS 及 IDA* 多。

- IDS 結合了 BFS 及 DFS 的優點。在其他實驗中發現，IDS 在不同難度下都能跟 BFS 一樣找到最佳解；而在大多數的 case 中，所需要暫時記錄的 node 數量會小於 BFS。因為 IDS 不用記錄當前深度所有可能的 nodes，且 stack 中的 nodes 有很大的機率被拿出來使用，因此不會產生 stack 實作 DFS 的副作用。
- IDA* 則優化了 A* 的空間複雜度，其原因與 IDS 相同，只需要記錄部分的點，且不會產生 stack 實作 DFS 的副作用。
- 綜合上述，若想要同時達到最佳解及耗費較少的 momory 時，可以透過 IDS 及 IDA* 來完成。

The effectiveness of generating nodes (expanded nodes)

- 目標：分析在不同演算法以及難度時，generated nodes 的使用效率（計算的方式為從 stack/queue/priority queue 中 pop 的次數除以 push 的次數）
-

	BFS	DFS	IDS	A*	IDA*
L01	0.9935 (1065/1072)	0.2581 (184/713)	0.9899 (3924/3964)	0.9888 (1057/1069)	0.9894 (3836/3877)
L11	0.9882 (838/848)	0.4216 (234/555)	0.9935 (9636/9699)	0.9859 (837/849)	0.9933 (9436/6500)
L21	0.9923 (259/261)	0.6008 (152/253)	0.9877 (1935/1959)	0.9885 (257/260)	0.9867 (1856/1881)
L31	0.9863 (3900/3954)	0.4010 (1061/2646)	0.9994 (50849/50881)	0.9838 (3888/3952)	0.9993 (50438/50471)
L40	0.9591 (2905/3029)	0.4452 (1137/2554)	0.9967 (27791/27883)	0.9679 (2982/3081)	0.9968 (28676/28769)

- 結果分析：
  - 除了 DFS 之外，其他四種演算法的使用效率皆很高。
  - BFS 的 nodes 在 queue 中的優先序會隨著時間上升，所以被 pop 出來的機率高。使用效率說明了 BFS 盡可能探索每個點的可能性，藉此來找到最佳解。
  - 由於使用 recursion 實作 DFS 會導致拜訪的太深，使 memory 的容量不足以記錄 **activation records**，因此使用 stack 來實作。尋找路徑的過程中，會把該 node 所有可以前往的 nodes 放入 stack 中，並在下一輪取出最上方的 nodes 來實現以深度為優先的 DFS。但這樣的作法會導致許多點被放入後，卻不會被使用到（因為優先序一直下降），導致最終 DFS 的 nodes 使用效率很低。
  - A* 是以 Priority Queue 實作，會根據 Priority 的大小來取出 nodes。然而因為 heuristic function 沒辦法有效估計距離的緣故，導致 A* 的行為與 BFS 差異不大，因此 push 和 pop 次數皆差異不大，使兩者的使用效率相當接近。

- IDS 及 IDA* 雖然是以 DFS 為基底來實作，卻不會產生 DFS 的使用效率低的問題，相反地，使用效率幾乎穩定在 98% 以上。
  - 使用效率比 DFS 高：在搜尋最佳解時，他們需要盡可能去延伸探訪過的 nodes 的路徑，因此儘管是用 stack 實作，但是 nodes 被再次拿出來的機率卻比 DFS 高出許多，所以效率比 DFS 高上許多。
  - 使用效率達 98 % 以上：以 DFS 為基底，探訪的過程中沒辦法確保 state 是以最短路徑抵達的，因此若 state 可以透過更短的路徑抵達，都要再次放入 stack 中，以找到最佳解。從上述得知，因為這兩個演算法都會重複放入同樣的 state，push 跟 pop 次數便會上升，使用效率也會因此較為穩定。

Implement the explored set by graph search

- 一開始先試著實作 Tree-search，由於沒有紀錄 Exist State，BFS 要多跑更多 State，而 DFS 會卡在無窮迴圈中（將某台車子右移後，又將那台車子左移，重複循環）。
- 因此我改用 Graph-search 來實作這五個演算法，多花一些空間紀錄 Exist State，並花時間搜索 State 是否存在於 Exist State 中。
- 我認為所花費的時間及空間是值得的。因為紀錄 Exist State 的話，可以使我們避免去展開一顆很大的樹（當它已經被遍歷過的話），如此以來所花費的時間差距不會太大，而且我是透過 **dictionary** 的方式來記錄 Exist State，其背後的實作方法為 **hashmap**，因此查找的速度也很快。
- 在空間上，確實比 Tree-search 多花空間紀錄，但是藉此可以減少額外的探訪及避免無窮迴圈，在不擔憂記憶體不足的情況下，我認為這樣的 trade-off 是值得的。

Things I learned from this project

- 在這次的 Project 中，實際透過五種演算法去解決一個遊戲的問題。雖然我們學過很多演算法，但都沒有將其實作，導致對演算法的效能及可行性不甚了解。然而這次的作業讓我能深度的去了解如何實踐這些演算法，並分析每個演算法的優缺點。藉此讓自己往後在遇到問題時，可以較正確的將合適的演算法運用於問題上。
- 此外，在這次實作演算法前，我便花了許多時間去解析 Rush Hour Puzzle，並設計合適的 class 來描述這個問題。在這個過程中，我學到了如何去將一個實際上會出現的問題，轉換成程式能解決的問題，並設計出相對應的 model 來模擬遊戲的進行。

Remaining questions and Ideas of future investigation

- 如何去優化 heuristic function：由於 heuristic function 會大幅影響 A* 的效能，若能找出能更適當估計的 function，將可以把 A* 的優點展現出來。因此未來若有機會的話，要更深入的去探討如何針對一個問題去設計出合適的 heuristic function。
- 如何去實作 automatic puzzle generator：對於一個問題，如何去產生所有可能的測資是相當重要的，畢竟可能會有我們沒有辦法考慮的情況，因此我認為這部分是比較可惜的，沒辦法有系統地去測試自己的解法。在未來，希望能夠完善測試系統，確保自己的程式能在任何情況下完善的運作。

Appendix

Code

- 由於以 HackMD 書寫報告，Code 在轉成 PDF 之後沒辦法用滾輪滑動，因此有盡量以換行符來表示換行的位置，使 Code 能較清楚明瞭。
- Code 位於下一頁。
- 而以下是 HackMD 的連結：https://hackmd.io/gZ9zNum0TVWnqNXG6U1FMg?view (https://hackmd.io/gZ9zNum0TVWnqNXG6U1FMg?view)

```
import argparse
import queue
import sys
import time
import copy
import heapq
import itertools

Search_Type = ["BFS", "DFS", "IDS", "A*", "IDA*"]
# index = Board(int), value = [predecessor, (idx, x_loc, y_loc), (depth/cost)]
Exist_State = {}
# use to make sure every entry in the priority queue is unique
counter = itertools.count()
# use to record the maximum number of nodes kept in memory
maxsize = 1
# use to record the number of pushing nodes into queue/stack/priority queue
push = 1
# use to record the number of popping nodes into queue/stack/priority queue
pop = 0

class Car:
    def __init__(self, idx, x_loc, y_loc, length, direction):
        self._idx = idx              # index of car
        self._x_loc = x_loc          # location of x
        self._y_loc = y_loc          # location of y
        self._length = length        # length of car
        self._direction = direction  # direction of car: 1 is horizontal;
                                     # 2 is vertical.

    @property
    def Index(self):
        return self._idx

    @property
    def X_loc(self):
        return self._x_loc

    @property
    def Y_loc(self):
        return self._y_loc

    @property
    def Length(self):
        return self._length
    @property
    def Direction(self):
        return self._direction

    @X_loc.setter
    def X_loc(self, value:int):
        self._x_loc = value

    @Y_loc.setter
    def Y_loc(self, value:int):
```

```
        self._y_loc = value

class Board_State:
    def __init__(self, cars, board):
        self._cars = cars    # list of Car
        self._board = board  # 6x6 board
        self._key = 0

    @property
    def Cars(self)->list:
        return self._cars

    @property
    def Board(self)->list:
        return self._board

    @property
    def Key(self)->int:
        return self._key

    def State_to_Key(self)->int:
        ret = 0
        for car in self._cars:
            ret = ret * 100 + car.X_loc * 10 + car.Y_loc
        return ret

    @Key.setter
    def Key(self, value:int):
        self._key = value

    def MoveLeft(self, idx, x_loc, y_loc, length):
        new_state = copy.deepcopy(self)
        new_state.Board[x_loc][y_loc-1] = True
        new_state.Board[x_loc][y_loc+length-1] = False
        new_state.Cars[idx].Y_loc = new_state.Cars[idx].Y_loc - 1
        new_state.Key = new_state.State_to_Key()
        return new_state

    def MoveRight(self, idx, x_loc, y_loc, length):
        new_state = copy.deepcopy(self)
        new_state.Board[x_loc][y_loc+length] = True
        new_state.Board[x_loc][y_loc] = False
        new_state.Cars[idx].Y_loc = new_state.Cars[idx].Y_loc + 1
        new_state.Key = new_state.State_to_Key()
        return new_state

    def MoveUp(self, idx, x_loc, y_loc, length):
        new_state = copy.deepcopy(self)
        new_state.Board[x_loc-1][y_loc] = True
        new_state.Board[x_loc+length-1][y_loc] = False
        new_state.Cars[idx].X_loc = new_state.Cars[idx].X_loc - 1
        new_state.Key = new_state.State_to_Key()
        return new_state

    def MoveDown(self, idx, x_loc, y_loc, length):
```

```
        new_state = copy.deepcopy(self)
        new_state.Board[x_loc+length][y_loc] = True
        new_state.Board[x_loc][y_loc] = False
        new_state.Cars[idx].X_loc = new_state.Cars[idx].X_loc + 1
        new_state.Key = new_state.State_to_Key()
        return new_state

class PriorityQueue:

    def __init__(self):
        self._queue = []
        self._size = 0

    def push(self, entry):
        heapq.heappush(self._queue, entry)
        self._size += 1

    def pop(self):
        try:
            ret = heapq.heappop(self._queue)
        except IndexError:
            print("Index Error happens because of trying popping an item \
            from an empty heap.")
        else:
            self._size -= 1
            return ret

    def top(self):
        try:
            return self._queue[0]
        except IndexError:
            print("Index Error happens because of trying accessing the top item \
            from an empty heap.")

    def empty(self)->bool:
        return (self._size == 0)

    def size(self)->int:
        return self._size

def ReadFile(FileName: str)->Board_State:
    cars = []
    board = [[False for i in range(6)] for j in range(6)]

    ### Store the information of the Beginning Board ###
    f = open(FileName)
    line = f.readline()
    while line:
        information = line.strip().split()
        new_car = Car(int(information[0]), int(information[1]), int(information[2]),
                      int(information[3]), int(information[4]))
        cars.append(new_car)
        if new_car.Direction == 1:
            for i in range(new_car.Y_loc, new_car.Y_loc+new_car.Length):
                board[new_car.X_loc][i] = True
```

```
        else:
            for i in range(new_car.X_loc, new_car.X_loc+new_car.Length):
                board[i][new_car.Y_loc] = True
        line = f.readline()
    f.close()

    return Board_State(cars, board)

def BFS(initial_state: Board_State)->int:

    # Initialize Data
    global maxsize, pop, push
    q = queue.Queue(maxsize=0) # maxsize <= 0: The size of the queue is not limited
    q.put(initial_state)
    termianl_code = -1
    Exist_State[initial_state.Key] = [-1, (-1, -1, -1)]

    # Traversal
    while(not q.empty()):

        maxsize = max(maxsize, q.qsize())
        current_state = q.get()
        current_cars, current_board, current_key = \
                    current_state.Cars, current_state.Board, current_state.Key
        pop += 1


        for i in range(len(current_cars)):
            idx, x_loc, y_loc, length, direction = current_cars[i].Index, \
            current_cars[i].X_loc, current_cars[i].Y_loc, current_cars[i].Length, \
            current_cars[i].Direction

            # Current car is Horizontal
            if direction == 1:

                # Check whether we can move current car to left
                if y_loc > 0 and not current_board[x_loc][y_loc-1]:
                    new_state = current_state.MoveLeft(idx, x_loc, y_loc, length)

                    # Check if the state can be put into Queue or not
                    if new_state.Key not in Exist_State:
                        q.put(new_state)
                        Exist_State[new_state.Key] = [current_key,
                        (idx, new_state.Cars[idx].X_loc, new_state.Cars[idx].Y_loc)]
                        push += 1

                # Check whether we can move current car to right
                if y_loc+length < 6 and not current_board[x_loc][y_loc+length]:
                    new_state = current_state.MoveRight(idx, x_loc, y_loc, length)

                    # Check if the state can be put into Queue or not
                    if new_state.Key not in Exist_State:
                        q.put(new_state)
                        Exist_State[new_state.Key] = [current_key,
                        (idx, new_state.Cars[idx].X_loc, new_state.Cars[idx].Y_loc)]
```

```
                        push += 1

                    maxsize = max(maxsize, q.qsize())

                    # Check whether we find the final state
                    if new_state.Cars[0].X_loc == 2 and new_state.Cars[0].Y_loc == 4:
                        termianl_code = new_state.Key
                        break

            # Current car is Vertical
            else:

                # Check whether we can move current car to Up
                if x_loc > 0 and not current_board[x_loc-1][y_loc]:
                    new_state = current_state.MoveUp(idx, x_loc, y_loc, length)

                    # Check if the state can be put into Queue or not
                    if new_state.Key not in Exist_State:
                        q.put(new_state)
                        Exist_State[new_state.Key] = [current_key,
                        (idx, new_state.Cars[idx].X_loc, new_state.Cars[idx].Y_loc)]
                        push += 1

                # Check whether we can move current car to Down #
                if x_loc+length < 6 and not current_board[x_loc+length][y_loc]:
                    new_state = current_state.MoveDown(idx, x_loc, y_loc, length)

                    # Check if the state can be put into Queue or not
                    if new_state.Key not in Exist_State:
                        q.put(new_state)
                        Exist_State[new_state.Key] = [current_key,
                        (idx, new_state.Cars[idx].X_loc, new_state.Cars[idx].Y_loc)]
                        push += 1

        # Find the final state
        if termianl_code != -1:
            break

    return termianl_code

def DFS(initial_state: Board_State, depth: int)->int:

    # Initialize Data
    global maxsize, push, pop
    stack = []
    stack.append([initial_state, 0])
    termianl_code = -1
    Exist_State[initial_state.Key] = [-1, (-1, -1, -1), 0]

    # Traversal
    while(len(stack) != 0):

        maxsize = max(maxsize, len(stack))
        top = stack.pop()
        current_state, current_depth = top[0], top[1]
```

```
        current_cars, current_board, current_key = \
                        current_state.Cars, current_state.Board, current_state.Key
        pop += 1

        for i in range(len(current_cars)-1, -1, -1):

            idx, x_loc, y_loc, length, direction = current_cars[i].Index, \
            current_cars[i].X_loc, current_cars[i].Y_loc, current_cars[i].Length, \
            current_cars[i].Direction

            # Current car is Horizontal
            if direction == 1:

                # Check whether we can move current car to left
                if y_loc > 0 and not current_board[x_loc][y_loc-1]:
                    new_state = current_state.MoveLeft(idx, x_loc, y_loc, length)

                    # Check if the state can be put into Stack or not
                    if  ((new_state.Key not in Exist_State) and \
                        (current_depth < depth or depth == -1)) or \
                        ((new_state.Key in Exist_State) and \
                        (current_depth+1 < Exist_State[new_state.Key][2]) and (\
                        depth != -1)):

                        stack.append([new_state, current_depth+1])
                        Exist_State[new_state.Key] = [current_key,
                        (idx, new_state.Cars[idx].X_loc, new_state.Cars[idx].Y_loc),
                        current_depth+1]
                        push += 1

                # Check whether we can move current car to Right
                if y_loc+length < 6 and not current_board[x_loc][y_loc+length]:
                    new_state = current_state.MoveRight(idx, x_loc, y_loc, length)

                    # Check if the state can be put into Stack or not
                    if  ((new_state.Key not in Exist_State) and \
                        (current_depth < depth or depth == -1)) or \
                        ((new_state.Key in Exist_State) and \
                        (current_depth+1 < Exist_State[new_state.Key][2]) and \
                        (depth != -1)):

                        stack.append([new_state, current_depth+1])
                        Exist_State[new_state.Key] = [current_key,
                        (idx, new_state.Cars[idx].X_loc, new_state.Cars[idx].Y_loc),
                        current_depth+1]
                        push += 1

                    maxsize = max(maxsize, len(stack))

                    # Find the terminal state
                    if new_state.Cars[0].X_loc == 2 and new_state.Cars[0].Y_loc == 4:
                        Exist_State[new_state.Key] = [current_key,
                        (idx, new_state.Cars[idx].X_loc, new_state.Cars[idx].Y_loc),
                        current_depth+1]
                        termianl_code = new_state.Key
```

```
                        break

                # Current car is Vertical
                else:

                    # Check whether we can move current car to Up
                    if x_loc > 0 and not current_board[x_loc-1][y_loc]:
                        new_state = current_state.MoveUp(idx, x_loc, y_loc, length)

                        # Check if the state can be put into Stack or not
                        if  ((new_state.Key not in Exist_State) and \
                            (current_depth < depth or depth == -1)) or \
                            ((new_state.Key in Exist_State) and \
                            (current_depth+1 < Exist_State[new_state.Key][2]) and \
                            (depth != -1)):

                            stack.append([new_state, current_depth+1])
                            Exist_State[new_state.Key] = [current_key,
                            (idx, new_state.Cars[idx].X_loc, new_state.Cars[idx].Y_loc),
                            current_depth+1]
                            push += 1

                    # Check whether we can move current car to Down
                    if x_loc+length < 6 and not current_board[x_loc+length][y_loc]:
                        new_state = current_state.MoveDown(idx, x_loc, y_loc, length)

                        # Check if the state can be put into Stack or not
                        if  ((new_state.Key not in Exist_State) and \
                            (current_depth < depth or depth == -1)) or \
                            ((new_state.Key in Exist_State) and \
                            (current_depth+1 < Exist_State[new_state.Key][2]) and \
                            (depth != -1)):

                            stack.append([new_state, current_depth+1])
                            Exist_State[new_state.Key] = [current_key,
                            (idx, new_state.Cars[idx].X_loc, new_state.Cars[idx].Y_loc),
                            current_depth+1]
                            push += 1

        # Find the final state
        if termianl_code != -1:
            break

    return termianl_code

def IDS(initial_state: Board_State)->int:

    global maxsize, push, pop
    depth = 0
    # try to find answer with depth_limit = 0
    terminal_state = DFS(initial_state, depth)

    # terminal state == -1 implies that we still haven't found the goal
    while(terminal_state == -1):
        maxsize = 1
```

```
            push = 1
            pop = 0
            # increase depth_limit by 1
            depth += 1
            Exist_State.clear()
            # try to find answer with new depth_limit
            terminal_state = DFS(initial_state, depth)

        return terminal_state

    def A_star(initial_state: Board_State)->int:

        # blocking heuristic
        def Heuristic(third_row: list, start: int)->int:
            cnt = 0
            for i in range(start, 6):
                if third_row[i] == 0:
                    cnt += 1
            return cnt

        # check if we can push the new state to priority queue
        def checker(current_state: Board_State, new_state: Board_State, \
                    current_depth: int, idx: int):
            global push
            new_cost = current_depth + 1 + Heuristic(new_state.Board[2],
                        new_state.Cars[0].Y_loc + new_state.Cars[0].Length)

            # 若尚未被發現或已經被發現但是 cost 更低的話，可以放入 Priority Queue
            if (new_state.Key not in Exist_State) or \
                (new_state.Key in Exist_State and \
                    new_cost < Exist_State[new_state.Key][2]):

                Entry = [new_cost, next(counter), new_state, current_depth+1]
                OpenList.push(Entry)
                Exist_State[new_state.Key] = [current_state.Key,
                (idx, new_state.Cars[idx].X_loc, new_state.Cars[idx].Y_loc), new_cost]
                push += 1

            return

        global maxsize, pop
        OpenList = PriorityQueue()
        Entry = [Heuristic(initial_state.Board[2],
        initial_state.Cars[0].Y_loc + initial_state.Cars[0].Length),
        next(counter), initial_state, 0]
        OpenList.push(Entry)
        terminal_code = -1
        Exist_State[initial_state.Key] = [-1, (-1, -1, -1),
        Heuristic(initial_state.Board[2],
                    initial_state.Cars[0].Y_loc + initial_state.Cars[0].Length)]

        # Traversal
        while(not OpenList.empty()):

            maxsize = max(maxsize, OpenList.size())
```

```
        item = OpenList.pop()
        priorty, current_state, current_depth = item[0], item[2], item[3]
        current_cars, current_board = current_state.Cars, current_state.Board

        # check if the node is in closed list
        if priorty > Exist_State[current_state.Key][2]:
            continue
        pop += 1

        for i in range(len(current_cars)):
            idx, x_loc, y_loc, length, direction = current_cars[i].Index, \
            current_cars[i].X_loc, current_cars[i].Y_loc, current_cars[i].Length, \
            current_cars[i].Direction

            # Current car is Horizontal
            if direction == 1:

                # Check whether we can move current car to left
                if y_loc > 0 and not current_board[x_loc][y_loc-1]:
                    new_state = current_state.MoveLeft(idx, x_loc, y_loc, length)
                    checker(current_state, new_state, current_depth, idx)

                # Check whether we can move current car to Right
                if y_loc+length < 6 and not current_board[x_loc][y_loc+length]:
                    new_state = current_state.MoveRight(idx, x_loc, y_loc, length)
                    checker(current_state, new_state, current_depth, idx)

                    maxsize = max(maxsize, OpenList.size())

                    # Find the terminal state
                    if new_state.Cars[0].X_loc == 2 and new_state.Cars[0].Y_loc == 4:
                        terminal_code = new_state.Key
                        break

            # Current car is Vertical
            else:

                # Check whether we can move current car to Up
                if x_loc > 0 and not current_board[x_loc-1][y_loc]:
                    new_state = current_state.MoveUp(idx, x_loc, y_loc, length)
                    checker(current_state, new_state, current_depth, idx)

                # Check whether we can move current car to Down
                if x_loc+length < 6 and not current_board[x_loc+length][y_loc]:
                    new_state = current_state.MoveDown(idx, x_loc, y_loc, length)
                    checker(current_state, new_state, current_depth, idx)

        # Find the final state
        if terminal_code != -1:
            break

    return terminal_code

def DFS_with_heuristic(initial_state: Board_State, limit_cost: int)->int:
```

```
# blocking heuristic
def Heuristic(third_row: list, start: int)->int:
    cnt = 0
    for i in range(start, 6):
        if third_row[i] == 0:
            cnt += 1
    return cnt

# check if we can push the new state to stack
def checker(current_state: Board_State, new_state: Board_State, \
            current_depth: int, idx: int):
    nonlocal next_limit_cost
    global push
    new_cost = current_depth + 1 + Heuristic(new_state.Board[2],
    new_state.Cars[0].Y_loc + new_state.Cars[0].Length)

    # 若 new state 尚未被發現過且 cost 小於 limit cost，可以放入 stack
    # 若 new state 已經被發現過且新的 cost 小於 Exist state 所記錄的 cost 的話，
    # 可以放入 stack
    if ((new_state.Key not in Exist_State) and new_cost <= limit_cost) or \
        ((new_state.Key in Exist_State) and \
        new_cost < Exist_State[new_state.Key][2]):
        stack.append([new_state, current_depth+1])
        Exist_State[new_state.Key] = [current_key,
        (idx, new_state.Cars[idx].X_loc, new_state.Cars[idx].Y_loc), new_cost]
        push += 1

    # update the next cost limit
    if new_cost > limit_cost:
        next_limit_cost = min(new_cost, next_limit_cost)

    return

global maxsize, pop
next_limit_cost = 2147483647 # use to find the next cost limit
stack = []
stack.append([initial_state, 0])
terminal_code = -1
Exist_State[initial_state.Key] = [-1, (-1, -1, -1),
Heuristic(initial_state.Board[2],
initial_state.Cars[0].Y_loc + initial_state.Cars[0].Length)]

# Traversal
while(len(stack) != 0):

    maxsize = max(maxsize, len(stack))
    top = stack.pop()
    current_state, current_depth = top[0], top[1]
    current_cars, current_board, current_key = current_state.Cars, \
                            current_state.Board, current_state.Key
    pop += 1


    for i in range(len(current_cars)-1, -1, -1):
```

```
            idx, x_loc, y_loc, length, direction = current_cars[i].Index, \
            current_cars[i].X_loc, current_cars[i].Y_loc, current_cars[i].Length, \
            current_cars[i].Direction

            # Current car is Horizontal
            if direction == 1:

                # Check whether we can move current car to left
                if y_loc > 0 and not current_board[x_loc][y_loc-1]:
                    new_state = current_state.MoveLeft(idx, x_loc, y_loc, length)
                    checker(current_state, new_state, current_depth, idx)

                # Check whether we can move current car to Righ
                if y_loc+length < 6 and not current_board[x_loc][y_loc+length]:
                    new_state = current_state.MoveRight(idx, x_loc, y_loc, length)
                    checker(current_state, new_state, current_depth, idx)

                    maxsize = max(maxsize, len(stack))

                    # Find the terminal state
                    if new_state.Cars[0].X_loc == 2 and new_state.Cars[0].Y_loc == 4:
                        terminal_code = new_state.Key
                        break

            # Current car is Vertical
            else:

                # Check whether we can move current car to Up
                if x_loc > 0 and not current_board[x_loc-1][y_loc]:
                    new_state = current_state.MoveUp(idx, x_loc, y_loc, length)
                    checker(current_state, new_state,
                            current_depth, idx)

                # Check whether we can move current car to Down
                if x_loc+length < 6 and not current_board[x_loc+length][y_loc]:
                    new_state = current_state.MoveDown(idx, x_loc, y_loc, length)
                    checker(current_state, new_state,
                            current_depth, idx)

        # Find the final state
        if terminal_code != -1:
            break

    if terminal_code == -1:
        terminal_code *= next_limit_cost

    return terminal_code

def IDA_star(initial_state: Board_State)->int:
    global maxsize, push, pop
    limit_cost = 1
    # try to find answer with cost_limit = 1
    terminal_state = DFS_with_heuristic(initial_state, limit_cost)

    # terminal state == -1 implies that we still haven't found the goal
```

```
        while(terminal_state <= 0):
            maxsize = 1
            push = 1
            pop = 0
            # update the new cost_limit
            limit_cost = -1 * terminal_state
            Exist_State.clear()
            # try to find answer with new cost_limit
            terminal_state = DFS_with_heuristic(initial_state, limit_cost)

        return terminal_state

    def main(args):
        global counter, maxsize, push, pop
        print("Searching Algorithm:", Search_Type[args.Type-1], "\nLevel:",
        args.FileName[:len(args.FileName)-4])

        initial_state = ReadFile(args.FileName)
        initial_state.Key = initial_state.State_to_Key()
        index_num = 0 # 用來儲存 final state
        Exist_State.clear()
        counter = itertools.count()
        maxsize = 1
        push = 1
        pop = 0

        begin = time.time()
        if args.Type == 1:
            index_num = BFS(initial_state)
        elif args.Type == 2:
            index_num = DFS(initial_state, -1)
        elif args.Type == 3:
            index_num = IDS(initial_state)
        elif args.Type == 4:
            index_num = A_star(initial_state)
        elif args.Type == 5:
            index_num = IDA_star(initial_state)
        finish = time.time()
        print("Spending Time:", round(finish - begin, 3), "s")
        print("Number of Expanding Nodes:", len(Exist_State))
        print("Maximum number of nodes kept in memory:", maxsize)
        print("# of pop / # of push:", pop/push, "(", pop, "/", push, ")")

        # save solution by visiting predecessor
        Solution = []
        while index_num != -1:
            Solution.append(Exist_State[index_num][1])
            index_num = Exist_State[index_num][0]
        Solution.reverse()
        Solution.pop(0)

        # print solution
        print("Solution Step:", len(Solution), "steps")
        for i in range(0, len(Solution)):
            print("step", i+1, ": [",  Solution[i][0], ",", Solution[i][1],
```

```
        ",", Solution[i][2], "]")

if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('Type', type = int,
    help = 'Enter type of searching\n1.BFS2.DFS 3.IDS 4.A* 5.IDA*)\n')
    parser.add_argument('FileName', type = str, help = 'Enter a file name\n')
    args = parser.parse_args()
    main(args)
```

The result of all levels

- The number of explored nodes during search

	BFS	DFS	IDS	A*	IDA*
L01	1072	713	1065	1069	1060
L02	2857	6921	2281	3047	2323
L03	824	619	804	815	782
L04	420	93	390	427	373
L10	2189	749	1933	2126	1908
L11	848	555	834	849	833
L20	2024	1888	1641	1803	1575
L21	261	253	260	260	258
L22	4799	2685	4441	4700	4221
L23	2773	1700	2574	2654	2503
L24	4267	3998	4201	4258	4194
L25	8867	5594	8804	8843	8781
L26	4849	3720	4824	4846	4824
L27	2831	1606	2602	2842	2573
L28	2116	1870	1885	2176	1996
L29	4313	3463	4299	4314	4310
L30	1170	922	1168	1170	1165
L31	3954	2646	3897	3952	3938
L40	3029	2554	2792	3081	2925

- The actual time of execution during search and the number of solution steps

	BFS	DFS	IDS	A*	IDA*
L01	0.61s/16步	0.107s/183步	13.814s/16步	0.597s/16步	13.390s/16
L02	2.23s/14步	1.310s/1463步	20.871s/14步	2.088s/14步	23.525s/14
L03	0.33s/33步	0.088s/197步	24.378s/33步	0.320s/33步	22.877s/33
L04	0.17s/22步	0.013s/40步	3.007s/22步	0.167s/22步	2.583s/22步
L10	1.49s/32步	0.148s/214步	77.488s/32步	1.404s/32步	81.794s/32
L11	0.37s/56步	0.095s/186步	89.766s/56步	0.361s/56步	87.974s/56
L20	1.02s/18步	0.337s/524步	9.04s/18步	0.849s/18步	8.289s/18步
L21	0.085s/49步	0.054s/100步	12.961s/49步	0.086s/49步	12.309s/49
L22	3.76s/46步	0.562s/725步	161.883s/46步	3.522s/46步	149.876s/4
L23	1.44s/49步	0.324s/550步	59.565s/49步	1.317s/49步	58.034s/49
L24	2.97s/50步	1.045s/981步	1070.761s/50步	2.901s/50步	1048.847s/
L25	8.07s/52步	1.396s/1465步	1003.657s/52步	7.726s/52步	1059.064s/
L26	3.72s/49步	1.397s/678步	553.702s/49步	3.618s/49步	577.465s/4
L27	1.46s/57步	0.291s/511步	148.645s/57步	1.46s/57步	148.386s/5
L28	1.45s/51步	0.523s/494步	66.419s/51步	1.47s/51步	65.671s/51
L29	3.25s/54步	0.882s/922步	817.586s/54步	3.165s/54步	832.218s/5
L30	0.63s/55步	0.221s/284步	74.276s/55步	0.618s/55步	80.441s/55
L31	2.39s/69步	0.608s/799步	624.046s/69步	2.391s/69步	607.881s/6
L40	2.21s/81步	0.830s/597步	497.054s/81步	2.265s/81步	475.921s/8

- The maximum number of nodes kept in the memory during search

	BFS	DFS	IDS	A*	IDA*
L01	159	529	72	161	72
L02	381	5448	70	591	72
L03	67	382	65	103	65
L04	45	52	26	59	24
L10	170	528	53	222	54
L11	45	321	96	61	92
L20	351	1343	38	478	36
L21	12	101	57	13	56
L22	389	1922	92	509	89
L23	185	1075	56	245	55
L24	381	2358	198	426	195
L25	593	3961	118	750	120
L26	304	2066	133	430	136
L27	123	1080	97	164	97
L28	154	1117	69	166	69
L29	214	2206	186	298	188
L30	60	560	86	76	86
L31	175	1585	143	207	142
L40	194	1417	176	238	176

- The effectiveness of generating nodes (expanded nodes)

	BFS	DFS	IDS	A*	IDA*
L01	0.9935 (1065/1072)	0.2581 (184/713)	0.9899 (3924/3964)	0.9888 (1057/1069)	0.9894 (3836/3877)
L02	0.8768 (2505/2857)	0.2128 (1473/6921)	0.9906 (4725/4770)	0.8467 (2580/3047)	0.9911 (5114/5160)
L03	0.9769 (805/824)	0.3829 (237/619)	0.9970 (5063/5078)	0.9448 (770/815)	0.9967 (4873/4889)
L04	0.9667 (406/420)	0.4409 (41/93)	0.9847 (1091/1108)	0.9578 (409/427)	0.9839 (976/992)
L10	0.9251 (2025/2189)	0.2951 (221/749)	0.9969 (11076/11110)	0.9239 (1966/2128)	0.9969 (11138/11173)
L11	0.9882 (838/848)	0.4216 (234/555)	0.9935 (9636/9699)	0.9859 (837/849)	0.9933 (9436/6500)
L20	0.8271 (1674/2024)	0.2887 (545/1888)	0.9916 (3794/3826)	0.7831 (1412/1803)	0.9908 (3545/3578)
L21	0.9923 (259/261)	0.6008 (152/253)	0.9877 (1935/1959)	0.9885 (257/260)	0.9867 (1856/1881)
L22	0.9596 (4605/4799)	0.2842 (763/2685)	0.9980 (21086/21129)	0.9534 (4481/4700)	0.9978 (19835/19879)
L23	0.9373 (2599/2773)	0.3676 (625/1700)	0.9972 (12717/12753)	0.9318 (2473/2654)	0.9970 (12280/12317)
L24	0.9852 (4204/4267)	0.4102 (1640/3998)	0.9994 (67154/67197)	0.9864 (4200/4258)	0.9993 (65725/65769)
L25	0.9939 (8813/8867)	0.2919 (1633/5594)	0.9991 (78497/78566)	0.9917 (8770/8843)	0.9991 (80178/80248)
L26	0.9953 (4826/4849)	0.5027 (1870/3720)	0.9983 (47175/47255)	0.9932 (4813/4846)	0.9983 (47923/48003)
L27	0.9583 (2713/2831)	0.3275 (526/1606)	0.9974 (20051/20104)	0.9546 (2713/2842)	0.9973 (19688/19742)
L28	0.9319 (1972/2116)	0.4027 (753/1870)	0.9951 (9366/9412)	0.9370 (2039/2176)	0.9950 (9413/9460)
L29	0.9961 (4296/4313)	0.3690 (1278/3463)	0.9987 (54841/54914)	0.9963 (4298/4314)	0.9987 (55454/55528)

	BFS	DFS	IDS	A*	IDA*
L30	0.9983 (1168/1170)	0.4479 (413/922)	0.9931 (9281/9345)	0.9966 (1166/1170)	0.9931 (9369/9434)
L31	0.9863 (3900/3954)	0.4010 (1061/2646)	0.9994 (50849/50881)	0.9838 (3888/3952)	0.9993 (50438/50471)
L40	0.9591 (2905/3029)	0.4452 (1137/2554)	0.9967 (27791/27883)	0.9679 (2982/3081)	0.9968 (28676/28769)