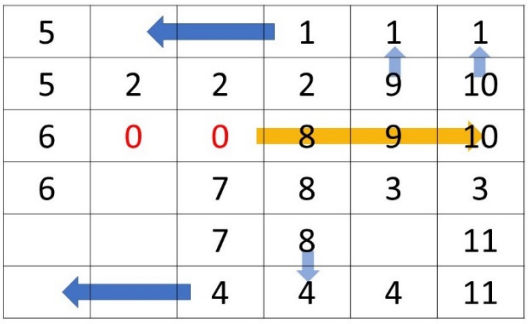
AI Capstone Programming Assignment #1 Report

0716206 陳昱丞

首先，先定義: 任何一台車移動一格稱為走一步。以下分析皆用下圖一作為起始狀態，而此狀態最少可在走 10 步的情況下得到解答。走法就如下圖二所示:

5			1	1	1
5	2	2	2	9	10
6	0	0	8	9	10
6		7	8	3	3
		7	8		11
		4	4	4	11

圖一: 數字即為車子之 index



圖二

接下來，下表顯示五個演算法之結果:

	BFS	DFS	IDS	A*	IDA*
Steps	10	185	10	10	10
Searched states	321	186	7177480	303	349070
Exe. times (sec)	0.125	0.074	662.818	0.139	35.512
Memory used	321	186	9	303	9

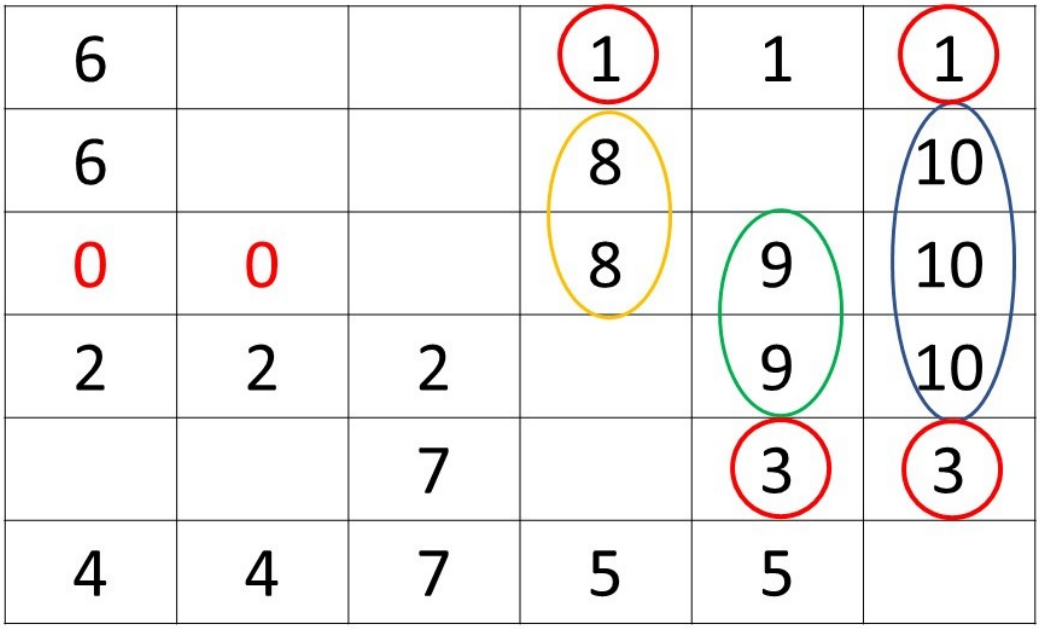
上表中的 memory used 是指記錄(走過)狀態的 dictionary 的大小。由於 DFS、IDS、IDA*具有回溯性質，dictionary 的大小會增加也會減少，因此只記錄 dictionary 最大的時候之大小。

初步分析:

上述結果可以看到，除了 DFS 外，其他演算法皆能找到 optimal solution。而 DFS 找不到是因為沒有限制深度的 DFS 會不斷的往「下」去尋找，也就是深度優先的性質，就無法維持搜尋樹的廣度。而搜尋過的狀態數量和執行時間基本上是成正比的，A*的執行時間較 BFS 稍久是因為要進行每個 states (nodes) 的 heuristic function 計算。在這種短步數的樣本中，BFS、DFS、A*的執行時間基本上相差無幾。最後，在記憶體消耗的部分，具有 graph search 性質的 BFS 和 A*因為要記錄所有走過的 state，記憶體消耗量會較大；而 tree search 性質的 IDS 和 IDA*受到回溯影響，雖然實際搜尋過的 state 數量驚人，但 dictionary 卻不太需要儲存 state，大多數 state 在存進去不久後就又被回溯刪掉了。而此例的 DFS 沒有經過回溯，因此搜尋過的 state 和儲存的 state 數量相等。

Heuristic function 討論:

原本的 heuristic function 是用 blocking heuristic，也就是計算擋在紅車前的車輛數。現在設計一個 advanced_heuristic function。假設擋住紅車的車有兩輛，分別是 x1、x2，則定義 advanced_h 函數為 2＋x1 的靈活度 ＋x2 的靈活度。若車的上下都被擋住，則靈活度為 2；若被擋住一邊，則靈活度為 1；若上下都沒被擋住，則靈活度為 0。也就是說，靈活度的值越小，代表越靈活。以下圖(取自 L02.txt)說明:



圖中，擋住紅車的有車 8、9、10，8 號車上方不能走、下方可以，靈活度為 1；9 號上方能走，下方不能，靈活度為 1；10 號上下都不能走，靈活度為 2。因此，此 state 的 advanced_h 就是 3+1+1+2=7。而我們要選擇 advanced_h 值最小的作為優先遍歷的節點，代表擋住紅車的車最容易被移開。

以下為 BFS 和 A* with advanced_h 的比較數據:

	BFS			A* with advanced_h		
	steps	times	memory	steps	times	memory
Example	10	0.135	321	10	0.055	137
L01	16	0.354	1076	16	2.061	1072
L02	14	1.134	3249	14	6.802	1952
L03	33	0.201	833	33	0.619	819
L04	22	0.116	430	22	0.291	405
L10	32	0.932	2333	32	2.075	1847

由上表可以看出，這個 heuristic function 總體來說並不是對所有 case 都有效，因此取差距較為明顯的 example (圖一) 和 L02 作分析。

	Blocking heuristic		Advanced heuristic	
	times	memory	times	memory
Example (10 steps)	0.139	303	0.055	137
L02 (14 steps)	2.369	2586	6.802	1952

可以看出，Advanced heuristic 記憶體的使用量較少，但對時間來說卻不一定有幫助。隨著 memory 越大，計算每個 state 的 heuristic 值和選出最小 heuristic 的 state 的時間會逐漸將 advanced heuristic 的優勢壓過去。總結來說，這個 advanced heuristic 對特定的 case 會特別有效，但 blocking heuristic 更能在穩定的時間內給出結果。

Automatic puzzle generator:

要做出一個 automatic puzzle generator，我們需要先給定一個 final state，也就是結果狀態，紅車在 row=2, column=4 的狀態。接著用 DFS 的手法，每次隨機選一台可移動的車進行隨機的移動，且必須把每個遍歷過的 state 都存起來，並保證之前走到過的 state 不會再被走到。若沒有車有路可以走或深度 (步數) 達到預期，就得到目前狀態的 puzzle 並 return。

Explored set 討論:

在實作上，我們可以將 state 轉成 string 並利用 python 的 list 或 dictionary 去儲存 graph search 搜尋過的所有狀態，並且避免之後搜尋到相同的狀態。這樣可能會遇到的問題是 dictionary 的大小成長得太快，超出電腦記憶體的負荷量。但若以執行效率而言，儲存 explored set 是必須的。舉一個 7 步解的例子，儲存 explored set 的 BFS 只需要零點幾秒；但若沒有儲存 explored set，僅保障當下算出來的 state 不會和上個 state 相同，這樣的 BFS 要花到 9 秒多。可見效率上有著相當大的差距。另外，在這個 6*6 的 Rush Hour 問題中，就算解的步數高達五六十步，explored set 的大小也大多小於 10000 個 state，這對現代的電腦來說是絕對可以負荷的，檢查 state 有沒有在 set 裡也花不了太多時間。因此我認為，對於這個問題而言，儲存 explored set 是值得且必要的。

Observations and Interpretations:

BFS:

BFS 在五個演算法中表現得非常穩定，基本上對於一百步以內的測資都能在五秒內完成計算。雖然消耗的空間量在五個演算法中是最多的，但如上所述，都在電腦可以儲存的範圍內。我認為，BFS 是解答這個問題的最佳演算法。

DFS:

如在初步分析中所述，DFS 是五個演算法中唯一無法給出最佳解的，因為其深度優先的性質。然而，在短步數的測資中，它的效率卻有可能相當高。因

為它快速的選擇分支往下走，不用進行什麼複雜的計算，可以說是「憑運氣」在找答案。若測資沒有很大，是有可能在短時間走到 final state 的。但若是要找最短路徑，這個方法就不太適用了。

IDS:

IDS 解決了 DFS 無限制往下搜索的問題，它利用逐步放寬深度限制的方式，讓 DFS 在往深度搜索的過程中，同時也兼顧了廣度的遍歷。此方式的優點在可以找到最佳解，以及可以不太需要消耗空間。但缺點就是執行時間會相當久，因為每次都必須要「從頭來過」。令深度限制為 d，搜尋 d=3 會需要遍歷過 d=2 已經搜尋過的 state。越大的 d 每增加 1，需要遍歷的 state 數量增加得越快。下表是搜尋深度與遍歷 state 數的關係圖:

d	state	d	state
1	3	7	40606
2	12	8	212612
3	57	9	1104474
4	285	10	5686272
5	1472	11	123966
6	7721	total	7177480

可以看到，每增加一層 d，state 數大致會是上一層的 5 倍。也就是說，state 以 5 的次方成長。因此，IDS 消耗的時間量會相當大，10 步解就需要遍歷七百多萬個 states，耗費大約 10 分鐘。

A*:

A*就像是改良版的 BFS，BFS 選擇 queue 的頭作為下一個 node，而 A*選擇有最小 f 值的 state 作為下個 node。A*能夠在搜尋更少 state、使用更少空間的情況下找到答案。但是因為要花時間計算每個 node 的 heuristic function 並找出最小值，因此所花費的時間不一定比 BFS 少。但若 heuristic function 寫得夠好，能節省的 state 夠多，執行時間應該能比 BFS 還短。然而，heuristic function 不一定在所有的 state 都能做出良好的估計，精準的 heuristic function 也有可能需要花費更多的時間計算。因此，如何在 heuristic function 計算量和程式的總執行時間上取得平衡成為 A*的重要課題。

IDA*:

IDA*就像是 IDS 和 A*結合，利用 heuristic function 幫助 IDS 更快的找到路徑。在同樣的測資下，IDS 需要遍歷七百多萬個 states，而 IDA*只需要三十幾萬個。這也明顯的反應在執行時間上，IDS 需要執行六百多秒，IDA*只需要三十幾秒，執行效率上有著相當明顯的提升。下表是搜尋深度與遍歷 state 數的關係圖:

d	state	d	state
1	1	7	751
2	1	8	6015
3	1	9	44083
4	3	10	289209
5	14	11	8895
6	97	total	349070

我認為，IDA*是一個非常好的中庸解法。它必能找到最佳解，也不需要太多記憶體空間，執行時間上也不需要 IDS 這麼久。它結合了 IDS 和 A*的優點，可說是空間和時間上的一個平衡點。

Things I have learned:

學會並熟悉五種搜尋演算法的運作原理、優缺點和實際的運作情況。並了解到在什麼條件下用什麼演算法會比較合適。例如若需要節省空間，時間上較沒有要求的話，則 IDS 和 IDA*較為適合；反之，則執行時間較短但記憶體需求較大的 BFS、A*較為適合。

Remaining questions:

需要找到一個更好的 heuristic function，能夠在任何 state 下都做出良好、穩定的估計，且計算不能太複雜、計算時間不能太長，讓 A*在執行時間上能夠穩定的勝過 BFS。

Ideas of future investigation:

這些演算法不只能用在這個問題中，另外還有許多問題、遊戲能透過這些方法找到解答。當然，每個遊戲的情況不同，也有可能會有不同的問題或結果等著我們去研究。

Appendix

Initialization, file input, file output:

```
import queue
import time

m=[]
for i in range(6):
    m.append([0]*6)

Input=[]
valid_dir=[] #every car's direction ; 0,2:left,right ; 1,3:up,down
valid_dir.append([-1,-1])
initial_state=""
state_dict={} #dictionary that saves states
car=0

def setting(file): #initialization
    global initial_state
    global car

    f=open("prog1_puzzle/"+file+".txt") #read file
    lines=f.readlines()

    for line in lines:
        I=[]
        for i in line.split():
            I.append(int(i))
        Input.append(I)

    for li in Input: #convert input file to structure the code need
        car+=1
        index=li[0]+1
        y=li[1]
        x=li[2]
        Len=li[3]
        direction=li[4]
```

```
            if direction==1:
                valid_dir.append([0,2])
                x1=x
                for i in range(Len):
                    m[y][x1]=index
                    x1+=1
            else:
                valid_dir.append([1,3])
                y1=y
                for i in range(Len):
                    m[y1][x]=index
                    y1+=1

        for i in m:
            for j in i:
                if j<10:
                    initial_state+="0"
                initial_state+=str(j) #translate initial state to string, easy for dict to save

        state_dict[initial_state]=""

#direct: right=0, up=1, left=2, down=3
def move(ind, state, direct):
        tmp_m=[]
        tmp_s=state
        tmp=[]
        for i in range(0,72,2):
            s=int(tmp_s[i]+tmp_s[i+1])
            tmp.append(s)
            if i%12==10:
                tmp_m.append(tmp)
                tmp=[]

        target=[]
        for i in range(6):
            for j in range(6):
                if tmp_m[i][j]==ind:
                    target.append([i,j])
```

```
if direct==0:        #right
        t=target[-1]
        if t[1]<5 and tmp_m[t[0]][t[1]+1]==0:          #valid move
                t2=target[0]
                tmp_m[t2[0]][t2[1]]=0
                tmp_m[t[0]][t[1]+1]=ind
        else:
                return "0" #invaid move

elif direct==1:    #up
        t=target[0]
        if t[0]>0 and tmp_m[t[0]-1][t[1]]==0:          #valid move
                t2=target[-1]
                tmp_m[t2[0]][t2[1]]=0
                tmp_m[t[0]-1][t[1]]=ind
        else:
                return "0" #invaid move

elif direct==2:    #left
        t=target[0]
        if t[1]>0 and tmp_m[t[0]][t[1]-1]==0:          #valid move
                t2=target[-1]
                tmp_m[t2[0]][t2[1]]=0
                tmp_m[t[0]][t[1]-1]=ind
        else:
                return "0" #invaid move

else:                    #down
        t=target[-1]
        if t[0]<5 and tmp_m[t[0]+1][t[1]]==0:          #valid move
                t2=target[0]
                tmp_m[t2[0]][t2[1]]=0
                tmp_m[t[0]+1][t[1]]=ind
        else:
                return "0" #invaid move

state_new=""
```

```
    for i in tmp_m:
        for j in i:
            if j<10:
                state_new+="0"
            state_new+=str(j) #encode the state to string
    return state_new

def output_route(route): #output all the route to solution.txt
    f=open("solution.txt","w")
    for r in route:
        board=[]
        for i in range(6):
            board.append([0]*6)

        row=0
        col=0
        for i in range(0,72,2):
            board[row][col]=int(r[i]+r[i+1])-1
            col+=1
            if (i+1)%12==11:
                row+=1
                col=0
        '''
        for i in board:
            for j in i:
                print(str(j-1),end=' ',file=f)
            print(file=f)
        print(file=f)
        '''

        for c in range(car):
            ok=False
            for i in range(6):
                for j in range(6):
                    if board[i][j]==c:
                        print(str(c)+' '+str(i)+' '+str(j),file=f)
                        ok=True
                        break
```

```
                    if ok:
                        break
            print(file=f)
        f.close()


def find_route(state_dict,terminal_state): #find route by dictionary mapping
        route=[]                                        #dict[new_state]=now_state
        route.append(terminal_state)
        st=terminal_state
        while st!="":                                   #dict[initial_state]=""
            st=state_dict[st]
            route.append(st)
        route.pop()
        route.reverse()
        output_route(route)
        return route

def depth_state_analyze(depth_state_num): #output number of searched states for each depth
        print()
        m=max(depth_state_num)+1
        for i in range(m):
            print('depth '+str(i)+': '+str(depth_state_num.count(i))+' states')
        print()
```

BFS:

```
#BFS
def sol_BFS(initial_state):

        q=queue.Queue()                 #queue which saves states
        q.put(initial_state)

        q_cnt=queue.Queue()             #queue which saves number of depth
        q_cnt.put(0)
        terminal_state=""
        depth_state_num_BFS=[]          #list which saves number of depth
        depth_state_num_BFS.append(0)
```

```
    while not q.empty():

        now_state=q.get()
        c=q_cnt.get()

        if now_state[34]=="0" and now_state[35]=="1": #finish
            terminal_state=now_state
            for i in range(0,72,2):
                print(now_state[i]+now_state[i+1]+' ',end='')
                if (i+1)%12==11:
                    print()
            break

        for i in range(car):
            new_state1=move(i+1,now_state,valid_dir[i+1][0]) #find valid move in one direction
            new_state2=move(i+1,now_state,valid_dir[i+1][1]) #find valid move in other direction
            if new_state1!="0" and state_dict.get(new_state1)==None: #push valid state in queue
                q.put(new_state1)
                state_dict[new_state1]=now_state #put searched states in dictionary
                                                 (dict[new_state]=now_state)
                q_cnt.put(c+1)
                depth_state_num_BFS.append(c+1)    #depth counter++

            if new_state2!="0" and state_dict.get(new_state2)==None: #push valid state in queue
                q.put(new_state2)
                state_dict[new_state2]=now_state #put searched states in dictionary
                                                 (dict[new_state]=now_state)
                q_cnt.put(c+1)
                depth_state_num_BFS.append(c+1)    #depth counter++

    BFS_route=find_route(state_dict,terminal_state)
    print('BFS route: '+str(len(BFS_route)-1)+' steps.')
    depth_state_analyze(depth_state_num_BFS)

def BFS():
    print('BFS:')
    s=time.time()
```

```
    sol_BFS(initial_state)
    t=time.time()
    print('exe time: '+str(t-s))
    print('dict len: '+str(len(state_dict)))
```

DFS:

```
solve=False
depth_state_num_DFS=[]
terminal_state=""
#DFS
def sol_DFS(state,depth): #recursive to get the answer
    global solve
    global terminal_state
    if solve==True:            #if answer already got, return
        return

    depth_state_num_DFS.append(depth)

    if state[34]=="0" and state[35]=="1": #finish
        terminal_state=state
        for i in range(0,72,2):
            print(state[i]+state[i+1]+' ',end='')
            if (i+1)%12==11:
                print()
        solve=True
        return
    for i in range(car):
        new_state1=move(i+1,state,valid_dir[i+1][0])
        new_state2=move(i+1,state,valid_dir[i+1][1])
        if new_state1!="0" and state_dict.get(new_state1)==None:
            state_dict[new_state1]=state #put state in dict
            depth+=1
            sol_DFS(new_state1,depth)       #recursive
            if solve==True:
                return
            del state_dict[new_state1]          #backtracking
            depth-=1                             #backtracking
```

```
            if new_state2!="0" and state_dict.get(new_state2)==None:
                state_dict[new_state2]=state      #put state in dict
                depth+=1
                sol_DFS(new_state2,depth)       #recursive
                if solve==True:
                    return
                del state_dict[new_state2]         #backtracking
                depth-=1                           #backtracking

def DFS():
    print('DFS:')
    state_dict.clear()
    state_dict[initial_state]=""
    s=time.time()
    sol_DFS(initial_state,0)
    t=time.time()
    DFS_route=find_route(state_dict,terminal_state)
    print('DFS route: '+str(len(DFS_route)-1)+' steps.')
    depth_state_analyze(depth_state_num_DFS)
    print('exe time: '+str(t-s))
    print('dict len: '+str(len(state_dict)))
```

IDS:

```
depth_state_num=[]
dict_len=[]
solve=False
terminal_state=""
#IDS
def sol_IDS(state,depth,max_depth):
    global solve
    global terminal_state

    if depth>=max_depth:    #if depth exceeded the limit, forcing return
        dict_len.append(str(len(state_dict)))
        depth_state_num.append(depth)
        return
    if solve==True:
```

```
            return

        if state[34]=="0" and state[35]=="1": #finish
            terminal_state=state
            for i in range(0,72,2):
                print(state[i]+state[i+1]+' ',end='')
                if (i+1)%12==11:
                    print()

            solve=True
            return
        for i in range(car):
            new_state1=move(i+1,state,valid_dir[i+1][0])
            new_state2=move(i+1,state,valid_dir[i+1][1])
            if new_state1!="0" and state_dict.get(new_state1)==None:
                state_dict[new_state1]=state
                depth+=1
                sol_IDS(new_state1,depth,max_depth)    #recursive
                if solve==True:
                    return
                depth-=1                                #backtracking
                del state_dict[new_state1]              #backtracking
            if new_state2!="0" and state_dict.get(new_state2)==None:
                state_dict[new_state2]=state
                depth+=1
                sol_IDS(new_state2,depth,max_depth)    #recursive
                if solve==True:
                    return
                depth-=1                                #backtracking
                del state_dict[new_state2]              #backtracking

def IDS():
    print('IDS:')
    s=time.time()
    for i in range(1,100):                          #gradually add the limit by 1
        state_dict.clear()
        state_dict[initial_state]=""
        sol_IDS(initial_state,0,i)
```

```
        if solve==True:
            break
    t=time.time()
    IDS_route=find_route(state_dict,terminal_state)
    print('IDS route: '+str(len(IDS_route)-1)+' steps.')
    depth_state_analyze(depth_state_num)

    print('exe time: '+str(t-s))
    print('dict len: '+str(max(dict_len)))
```

Class node:

```
class state_node:    #node use for A* and IDA*
    def __init__(self,s,n):
        self.state=s
        self.step=n

    def g(self):
        return self.step

    def h(self): #count the number of car that block in front of the red car
        target_row=[]
        s=self.state[24:36]
        for i in range(0,12,2):
            s1=int(s[i]+s[i+1])
            target_row.append(s1)
        cnt=0
        for i in reversed(target_row):
            if i==1:
                break
            if i!=0:
                cnt+=1
        return cnt


    def f(self): #f()=g()+h()
        return self.g()+self.h()
```

A*:

```
def sol_A_Star(initial_state):
    state_dict.clear()
    state_dict[initial_state]=""

    start=state_node(initial_state,0)
    state_list=[]
    state_list.append(start)
    while True:
        li=[]
        for nodes in state_list:
            li.append(nodes.f())

        current_node=state_list[li.index(min(li))] #current_node = node who has minimum f
        state_list.remove(current_node)                    #remove current node in state_list

        now_state=current_node.state

        if now_state[34]=="0" and now_state[35]=="1": #finish
            terminal_state=now_state
            for i in range(0,72,2):
                print(now_state[i]+now_state[i+1]+' ',end='')
                if (i+1)%12==11:
                    print()
            break

        for i in range(car):
            new_state1=move(i+1,now_state,valid_dir[i+1][0])
            new_state2=move(i+1,now_state,valid_dir[i+1][1])
            if new_state1!="0" and state_dict.get(new_state1)==None: #push in state_list
                new_node=state_node(new_state1,current_node.step+1)
                state_list.append(new_node)
                state_dict[new_state1]=now_state

            if new_state2!="0" and state_dict.get(new_state2)==None: #push in state_list
                new_node=state_node(new_state2,current_node.step+1)
                state_list.append(new_node)
```

```
                    state_dict[new_state2]=now_state


    A_star_route=find_route(state_dict,terminal_state)
    print('A* route: '+str(len(A_star_route)-1)+' steps.')

def A_Star():
    print('A*:')
    s=time.time()
    sol_A_Star(initial_state)
    t=time.time()
    print('exe time: '+str(t-s))
    print('dict len: '+str(len(state_dict)))
```

IDA*:

```
terminal_state=""
solve=False
depth_state_num=[]
dict_len=[]
def sol_IDA_star(node,limit):
    global solve
    global terminal_state
    if node.f()>=limit:    #if node's f exceed the limit, forcing return
        depth_state_num.append(limit)
        dict_len.append(str(len(state_dict)))
        return
    if solve==True:
        return
    state=node.state
    if state[34]=="0" and state[35]=="1": #finish
        terminal_state=state
        for i in range(0,72,2):
            print(state[i]+state[i+1]+' ',end='')
            if (i+1)%12==11:
                print()
        solve=True
        return
```

```
    for i in range(car):
        new_state1=move(i+1,state,valid_dir[i+1][0])
        new_state2=move(i+1,state,valid_dir[i+1][1])
        if new_state1!="0" and state_dict.get(new_state1)==None:
            new_node=state_node(new_state1,node.step+1)
            state_dict[new_state1]=state
            sol_IDA_star(new_node,limit)    #recursive
            if solve==True:
                return
            del state_dict[new_state1]          #backtracking
        if new_state2!="0" and state_dict.get(new_state2)==None:
            new_node=state_node(new_state2,node.step+1)
            state_dict[new_state2]=state
            sol_IDA_star(new_node,limit)    #recursive
            if solve==True:
                return
            del state_dict[new_state2]          #backtracking

def IDA_Star():
    print('IDA*:')
    s=time.time()
    for i in range(1,100):                      #gradually add the limit by 1
        state_dict.clear()
        state_dict[initial_state]=""
        start=state_node(initial_state,0)
        sol_IDA_star(start,i)
        if solve==True:
            break
    t=time.time()

    IDA_star_route=find_route(state_dict,terminal_state)
    print('IDA* route: '+str(len(IDA_star_route)-1)+' steps.')
    depth_state_analyze(depth_state_num)
    print('exe time: '+str(t-s))
    print('dict len: '+str(max(dict_len)))
```

Main function:

```
select_algo=int(input('1: BFS    2: DFS    3: IDS    4: A*    5: IDA*    input: '))
select_file=input('file: ')
print()
setting(select_file)
if select_algo==1:
    BFS()
elif select_algo==2:
    DFS()
elif select_algo==3:
    IDS()
elif select_algo==4:
    A_Star()
else:
    IDA_Star()
```