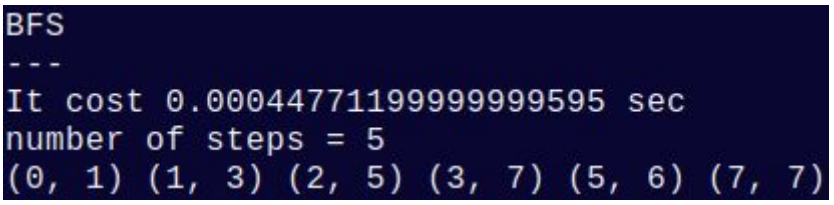
Introduction to Artificial Intelligence Prog. 1 Report

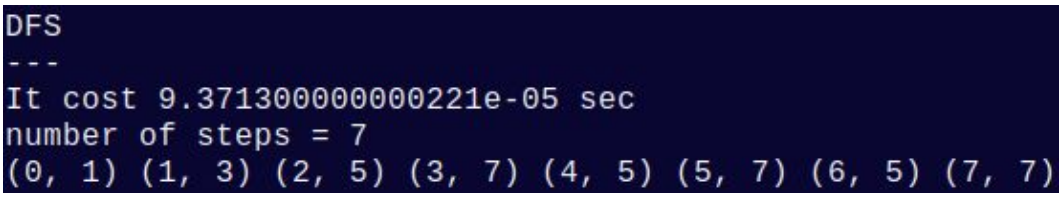
0516003 李智嘉

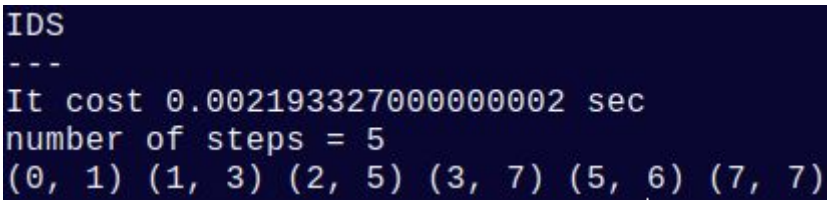
1. Introduction

In this program assignment, we'll implement BFS, DFS, IDS graph search, and A*, IDA* tree search to solve knight moving problem in 8*8 chessboard (or arbitrary chessboard size).

In the following experiment, I'll focus on (1) does it find minimal path (2) how much the time it need (3) other changes and the performance result.

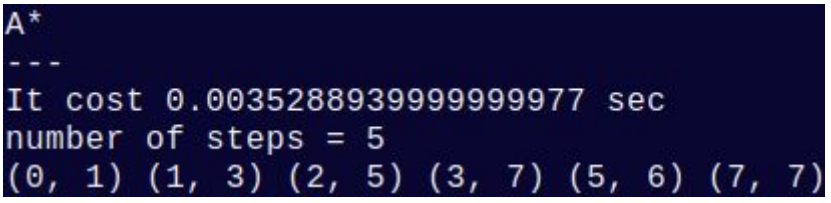
2. Basic Setting Experiments
   - Setting
     - 8*8 chessboard
     - starting position (0, 1)
     - target position (7, 7)
   - BFS
     - 

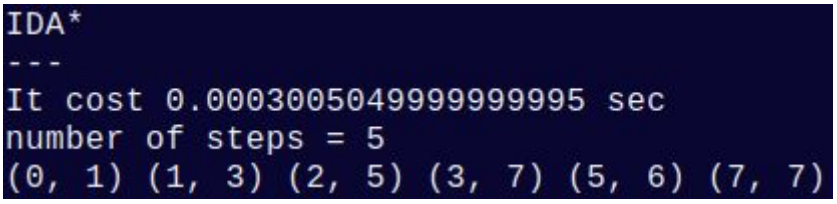
   - DFS
     - 

   - IDS
     - For IDS, we can not just call DFS graph search many times, since the node in the shortest path could exist in other path, and when the node is expanded and could not achieve goal state, this node would not be expanded again.
     - To avoid this problem, I implement two method, one is to use more space to solve it, the other is to use more time to solve it.
     - I left the detail and comparison in the next part (larger size chessboard), and here I use the more space complexity version.
     - 

   - Comparison between BFS, DFS, IDS
     - For running time, DFS spends least time, but it's result is not optimal since once it find one path, it will return it no matter it's shortest or not.

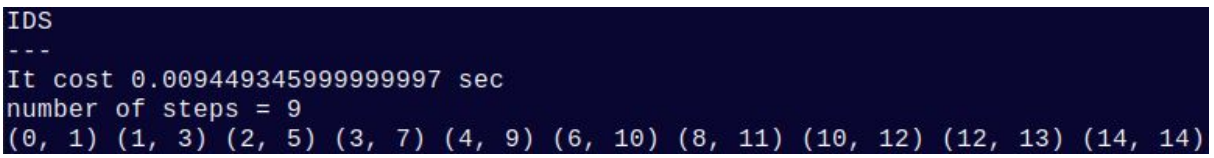
- BFS spends less time than IDS, even though the time complexity are both b^d.
- The reason of IDS is slower than BFS is that IDS need to call the recursive function more times, and IDS will run the same state more times.
- For space complexity, since it's graph search, the space complexity of BFS is bound by chessboard size (just 64 for 8*8 chessboard). Otherwise, for tree search, the space complexity would be b^d which is very huge when the goal state is far from initial state.

- A*
  - 

- IDA*
  - 

- Comparison between A* and IDA*
  - We can see that, both algorithm could find shortest path.
  - But IDA* is much faster than A*, the reason may be that A* is using priority, the process time is huge, so even though IDA* should run iteration many times, it still faster than A*.

3. Larger Setting Experiments
   - Setting
     - 15*15 chessboard
     - starting position (0, 1)
     - target position (14, 14)
   - IDS
     - For more space complexity version, I also store the depth when it expand one node. And if the node hasn't expand or the depth of current path is less than previous depth, then expand it.



- For more time complexity version, I unmark the flag which indicate the state has been expanded when this path could not achieve goal state.

```
IDS
---
It cost 1.1371473820000002 sec
number of steps = 9
(0, 1) (1, 3) (2, 5) (3, 7) (4, 9) (6, 10) (8, 11) (10, 12) (12, 13) (14, 14)
```

- We can see that the last version cost lots of time.

- Time consuming comparing
  - Setting: 25*25 chessboard, from (0, 1) to (24, 24)

Algorithm	Time Consuming	Optimal
BFS graph search	0.00459	Yes
DFS graph search	0.00050	No
IDS graph search (more space version)	0.07716	Yes
IDS graph search (more time version)	> 30s	
A* tree search	> 30s	
IDA* tree search	8.77400	Yes

  - We can see that, DFS is still the fast one, but not optimal.
  - For different IDS graph search, the more time version could not finish searching in 30 seconds, but the time of more space version only 0.077 seconds.
  - For A* tree search, it also can not finish the searching in 30 seconds, the reason may be priority queue is slow, or the searching space is too large.
  - Surprisingly, even though IDA* is tree search, it can still find the optimal solution in 8.77 seconds. The reason may be that heuristic function is helpful (so that IDS more time version can not finish query, but IDA* can)

4. Conclusion

In this homework, I implement five different searching algorithm, and compare the time it need, the optimality and different implementation detail. I also mention the performance when it comes to bigger chessboard.

Actually, I misunderstand what IDA* is when I implement it. I originally thought that it was A* in iterative version, until one student ask in the forum so

that I reimplement IDA*. And for IDS, it’s also until one student ask in the forum so that I find the problem it has.

I learn a lot from this homework and other students. It’s great :)

- main

hw1 > 0516003_hw1.py > ...

```python
import copy

from BFS import BFS_graph
from DFS import DFS_graph
from IDS import IDS
from A_star import A_star
from IDA_star import IDA_star
import utils

## The main running function of this program
def run(algo_type, s_x, s_y, g_x, g_y, board_size=8):
    ## do different search based on algo_type
    if algo_type == 0:
        print("BFS")
        sol_pth, cost_time = BFS_graph(s_x, s_y, g_x, g_y, board_size)
    elif algo_type == 1:
        print("DFS")
        sol_pth, cost_time = DFS_graph(s_x, s_y, g_x, g_y, board_size)
    elif algo_type == 2:
        print("IDS")
        sol_pth, cost_time = IDS(s_x, s_y, g_x, g_y, board_size)
    elif algo_type == 3:
        print("A*")
        sol_pth, cost_time = A_star(s_x, s_y, g_x, g_y, board_size)
    elif algo_type == 4:
        print("IDA*")
        sol_pth, cost_time = IDA_star(s_x, s_y, g_x, g_y, board_size)
    else:
        print("Algo Type Error!")
        exit(1)
    print("---")

    ## print result and cost time
    print(f'It cost {cost_time} sec')
    if sol_pth is None:
        print('Can not find path!')
    else:
        print(f'number of steps = {len(sol_pth)-1}')
        for _, i in enumerate(sol_pth):
            print(f'{i}', end=' ')
        print()

if __name__ == "__main__":
    algo_type = 4      ## (0: BFS), (1: DFS), (2: IDS), (3: A*), (4: IDA*)
    s_x, s_y = 0, 1     ## starting position
    g_x, g_y = 2, 2     ## target position
    board_size = 25      ## the edge size of square board

    run(algo_type, s_x, s_y, g_x, g_y, board_size)
```

- utils

hw1 > utils.py > ...

```python
import copy

moves = [(1, 2), (1, -2), (-1, 2), (-1, -2), (2, 1), (2, -1), (-2, 1), (-2, -1)]

def check_valid_pos(x, y, board_size=8):
    ## if it has been outside the board, then return False, otherwise return True
    if x < 0 or y < 0 or x >= board_size or y >= board_size:
        return False
    return True

## build initla board for BFS and DFS
def build_board(board_size, element):
    board = [
        [
            copy.deepcopy(element) for _ in range(board_size)
        ] for _ in range(board_size)
    ]
    return board

def compute_h(new_x, new_y, g_x, g_y):
    ## compute h score, would be (|dx|+|dy|)/3
    return int((max(0, g_x-new_x)+max(0, g_y-new_y))/3)
```

```python
def compute_g(prev_cost):
    ## compute g score (unit cost)
    return prev_cost

def compute_f(new_x, new_y, g_x, g_y, cur_cost):
    ## compute f score, add by h score and g score
    return compute_h(new_x, new_y, g_x, g_y) + compute_g(cur_cost)
```

- BFS

hw1 > BFS.py > BFS_graph

```python
import time
import utils

def BFS_graph(s_x, s_y, g_x, g_y, board_size=8):
    ## initial process time, to compute the program time
    start_time = time.process_time()

    ## construct the board
    ## the element would be (prev_x, prev_y, current_length)
    board = utils.build_board(board_size, (0, 0, 0))

    frontier = [(s_x, s_y, 0)]  ## frontier list as a queue
    done = False

    while len(frontier) > 0 and not done:
        cur_pos = frontier[0]   ## get one node from frontier list
        frontier.remove(cur_pos) ## pop out the current node from frontier list

        for move in utils.moves:
            new_x, new_y = cur_pos[0]+move[0], cur_pos[1]+move[1]   ## compute new node coordinate

            ## check whether it is valid
            if utils.check_valid_pos(new_x, new_y, board_size=board_size) and board[new_x][new_y][2] == 0:
                board[new_x][new_y] = (cur_pos[0], cur_pos[1], cur_pos[2]+1)    ## update the board values
                frontier.append((new_x, new_y, cur_pos[2]+1))   ## add new node to frontier
                ## if achieve goal state, break it
                if new_x == g_x and new_y == g_y:
                    done = True
                    break

    ## calculate the process time
    cost_time = time.process_time() - start_time

    ## construct the path from initial state to goal state
    if board[g_x][g_y][2] == 0: ## if the goal state hasn't achieve, set sol_pth as None
        sol_pth = None
    else:
        sol_pth = [(g_x, g_y)]
        ## add nodes to sol_pth until the sol_pth has initial state
        while sol_pth[-1][0] != s_x or sol_pth[-1][1] != s_y:
            prev_pos = board[sol_pth[-1][0]][sol_pth[-1][1]]   ## get previous node
            sol_pth.append((prev_pos[0], prev_pos[1]))
        sol_pth.reverse()

    return sol_pth, cost_time
```

- DFS

hw1 > DFS.py > recursive_dfs

```python
import time
import utils

def recursive_dfs(cur_x, cur_y, g_x, g_y, board, cur_steps, cur_pth, board_size=8):
    ## if achieve goal state, return the path
    if cur_x == g_x and cur_y == g_y:
        return cur_pth

    for move in utils.moves:
        new_x, new_y = cur_x+move[0], cur_y+move[1] ## compute new node coordinate
        if utils.check_valid_pos(new_x, new_y, board_size=board_size) and board[new_x][new_y]:  ## check whe
            board[new_x][new_y] = False ## mark the state as expanded
            cur_pth.append((new_x, new_y))  ## add this node to path list

            ## do recursion
            res_pth = recursive_dfs(new_x, new_y, g_x, g_y, board, cur_steps+1, cur_pth, board_size=board_si

            ## if find the path, then return it
            if res_pth is not None:
                return res_pth

            ## if this node could not achieve goal state, remove it from path list
            cur_pth.remove(cur_pth[-1])

    return None
```

```python
def DFS_graph(s_x, s_y, g_x, g_y, board_size=8):
    ## initial process time, to compute the program time
    start_time = time.process_time()

    cur_pth = [(s_x, s_y)]

    ## construct board, element is "True"
    ##      to record which state has been reach, True means it hasn't expanded
    board = utils.build_board(board_size, True)

    ## mark the initial state as expanded, and do the recursion
    board[s_x][s_y] = False
    sol_pth = recursive_dfs(s_x, s_y, g_x, g_y, board, 0, cur_pth, board_size=board_size)

    ## calculate the process time
    cost_time = time.process_time() - start_time
    return sol_pth, cost_time
```

- IDS

hw1 > IDS.py > ...

```python
import time
import utils

def recursive_dfs(cur_x, cur_y, g_x, g_y, board, cur_steps, cur_pth, max_steps, board_size=8):
    ## if achieve goal state, return the path
    if cur_x == g_x and cur_y == g_y:
        return cur_pth

    ## check whether next_steps is greater than threshold
    new_steps = cur_steps+1
    if new_steps > max_steps:
        return None

    for move in utils.moves:
        new_x, new_y = cur_x+move[0], cur_y+move[1] ## compute new node coordinate

        ## check whether it is valid and this state hasn't expanded or previous expanded depth is deeper
        if utils.check_valid_pos(new_x, new_y, board_size=board_size)\
                and (board[new_x][new_y][0] or new_steps<board[new_x][new_y][1]):
            board[new_x][new_y] = (False, new_steps) ## mark the state as achieved
            cur_pth.append((new_x, new_y))  ## add this node to path list

            ## do recursive
            res_pth = recursive_dfs(new_x, new_y, g_x, g_y, board, new_steps, cur_pth, max_steps, board_size
```

```python
            ## if it find the path, then return it
            if res_pth is not None:
                return res_pth

            ## if this node could not achieve goal state, remove it from path list
            cur_pth.remove(cur_pth[-1])

    return None

def dfs(s_x, s_y, g_x, g_y, max_steps, board_size=8):
    cur_pth = [(s_x, s_y)]

    ## construct board, element is (hasn't expanded, previous expanded depth)
    board = utils.build_board(board_size, (True, 0))

    ## mark the initial state as expanded, and do the recursion
    board[s_x][s_y] = (False, -1)
    sol_pth = recursive_dfs(s_x, s_y, g_x, g_y, board, 0, cur_pth, max_steps, board_size=board_size)

    return sol_pth
```

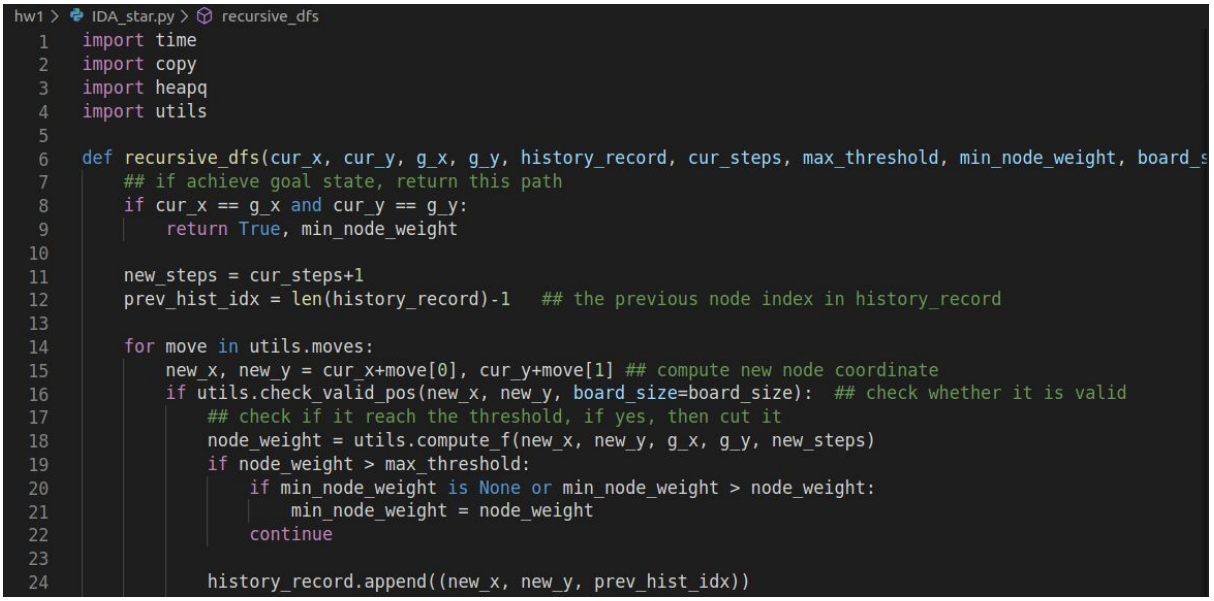
```python
def IDS(s_x, s_y, g_x, g_y, board_size):
    ## initial process time, to compute the program time
    start_time = time.process_time()

    ## do iterative DFS, `i` would be the threshold for current iteration
    for i in range(1, board_size**2):
        sol_pth = dfs(s_x, s_y, g_x, g_y, max_steps=i, board_size=board_size)

        ## if achieve goal state, then break this loop
        if sol_pth is not None:
            break

    ## calculate the process time
    cost_time = time.process_time() - start_time

    return sol_pth, cost_time
```

- A*

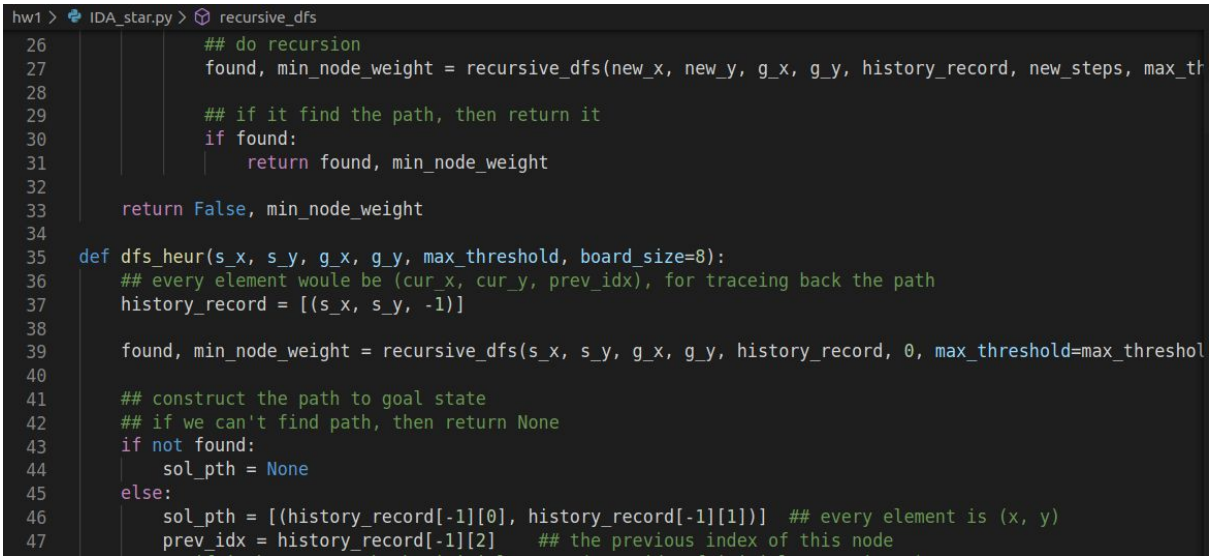
hw1 > A_star.py > A_star

```python
import time
import heapq
import utils

def A_star(s_x, s_y, g_x, g_y, board_size=8):
    ## initial process time, to compute the program time
    start_time = time.process_time()

    history_record = [] ## every element woule be (cur_x, cur_y, prev_idx), for traceing back the path
    frontier = [] ## every element would be (f, new_x, new_y, prev_hist_idx, new_g)

    ## add first node to priority queue
    heapq.heappush(frontier, (0, s_x, s_y, None, 0))
    done = False

    while len(frontier) > 0:
        node = heapq.heappop(frontier)
        history_record.append((node[1], node[2], node[3]))
        prev_hist_idx = len(history_record)-1   ## get the index of last history_record

        ## check if the node is goal state
        if node[1] == g_x and node[2] == g_y:
            done = True
            break

        ## push new nodes to frontier
        for move in utils.moves:
            new_x, new_y = node[1]+move[0], node[2]+move[1] ## compute new node coordinate
            if utils.check_valid_pos(new_x, new_y, board_size=board_size):    ## check if it is valid
                new_g = node[4]+1   ## compute g score
                f = utils.compute_f(new_x, new_y, g_x, g_y, new_g)  ## compute f score of new node

                ## add it to frontier
                heapq.heappush(frontier, (f, new_x, new_y, prev_hist_idx, new_g))

    ## calculate the process time
    cost_time = time.process_time() - start_time

    ## construct the path to goal state
    ## if it didn't find path, then return None
    if not done:
        sol_pth = None
    else:
        sol_pth = [(history_record[-1][0], history_record[-1][1])]  ## every element is (x, y)
        prev_idx = history_record[-1][2]    ## the previous index of this node

        ## if it hasn't reach the initial state (prev_idx of initial state is None)
        while prev_idx is not None:
            sol_pth.append((history_record[prev_idx][0], history_record[prev_idx][1]))
            prev_idx = history_record[prev_idx][2]
        sol_pth.reverse()

    return sol_pth, cost_time
```

- IDA*



```
hw1 > IDA_star.py > recursive_dfs
import time
import copy
import heapq
import utils

def recursive_dfs(cur_x, cur_y, g_x, g_y, history_record, cur_steps, max_threshold, min_node_weight, board_s
    ## if achieve goal state, return this path
    if cur_x == g_x and cur_y == g_y:
        return True, min_node_weight

    new_steps = cur_steps+1
    prev_hist_idx = len(history_record)-1   ## the previous node index in history_record

    for move in utils.moves:
        new_x, new_y = cur_x+move[0], cur_y+move[1] ## compute new node coordinate
        if utils.check_valid_pos(new_x, new_y, board_size=board_size):  ## check whether it is valid
            ## check if it reach the threshold, if yes, then cut it
            node_weight = utils.compute_f(new_x, new_y, g_x, g_y, new_steps)
            if node_weight > max_threshold:
                if min_node_weight is None or min_node_weight > node_weight:
                    min_node_weight = node_weight
                continue

            history_record.append((new_x, new_y, prev_hist_idx))
```



```
hw1 > IDA_star.py > recursive_dfs
            ## do recursion
            found, min_node_weight = recursive_dfs(new_x, new_y, g_x, g_y, history_record, new_steps, max_th

            ## if it find the path, then return it
            if found:
                return found, min_node_weight

    return False, min_node_weight

def dfs_heur(s_x, s_y, g_x, g_y, max_threshold, board_size=8):
    ## every element woule be (cur_x, cur_y, prev_idx), for traceing back the path
    history_record = [(s_x, s_y, -1)]

    found, min_node_weight = recursive_dfs(s_x, s_y, g_x, g_y, history_record, 0, max_threshold=max_threshol

    ## construct the path to goal state
    ## if we can't find path, then return None
    if not found:
        sol_pth = None
    else:
        sol_pth = [(history_record[-1][0], history_record[-1][1])]  ## every element is (x, y)
        prev_idx = history_record[-1][2]    ## the previous index of this node
```

```
        ## if it hasn't reach the initial state (prev_idx of initial state is -1)
        while prev_idx != -1:
            sol_pth.append((history_record[prev_idx][0], history_record[prev_idx][1]))
            prev_idx = history_record[prev_idx][2]
        sol_pth.reverse()

    return sol_pth, min_node_weight

## IDA* algorithm
def IDA_star(s_x, s_y, g_x, g_y, board_size):
    ## initial process time, to compute the program time
    start_time = time.process_time()

    sol_pth = None  ## the final solution path
    ## set the initial f threshold as h score of initial state
    max_f = utils.compute_h(s_x, s_y, g_x, g_y)

    ## if it hasn't find the path, then continue
    while sol_pth is None:
        sol_pth, min_node_weight = dfs_heur(s_x, s_y, g_x, g_y, max_threshold=max_f, board_size=board_size)

        if sol_pth is not None:
            break
```

```
            ## update f threshold as the minimum node weight of drop nodes
            max_f = min_node_weight

        ## calculate the process time
        cost_time = time.process_time() - start_time

        return sol_pth, cost_time
```