Introduction to AI

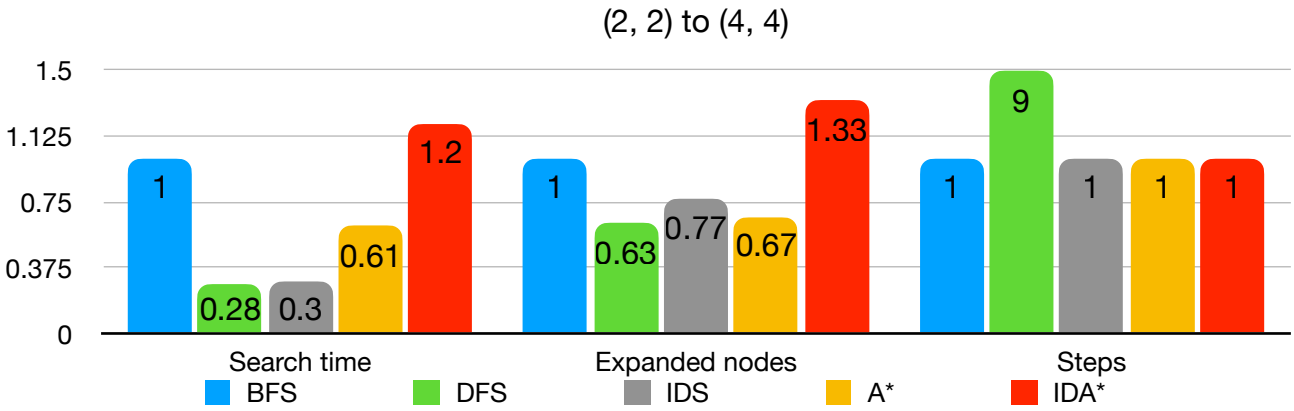
Programming Assignment #1

實驗與結果

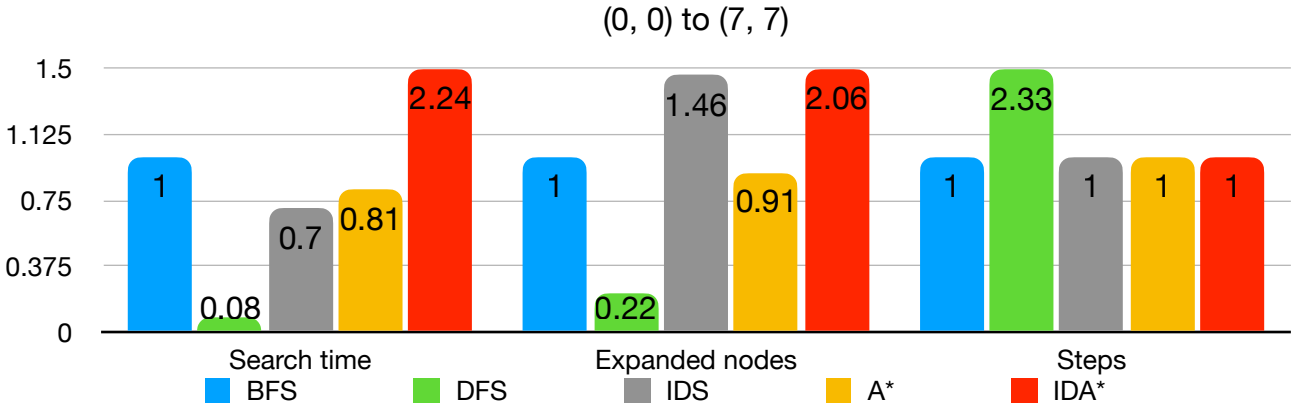
演算法種類比較

以下為在一個 8*8 盤面上，由 5 種演算法移動棋子的統計結果長條圖，其 X 軸依序為「搜尋時間」、「展開的節點數」、「步數」，Y 軸為各項數據相對於 BFS 之數據的比值。

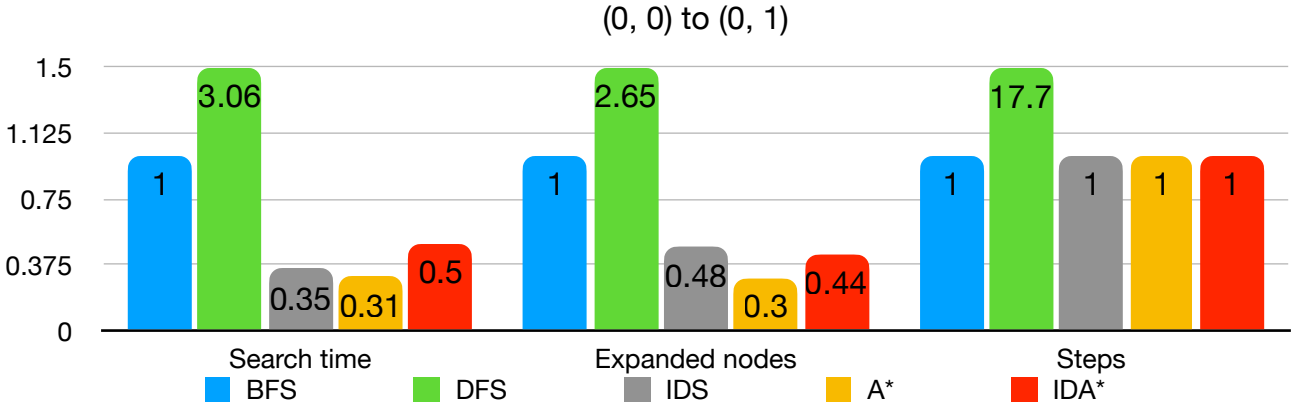
短距離



長距離



相鄰



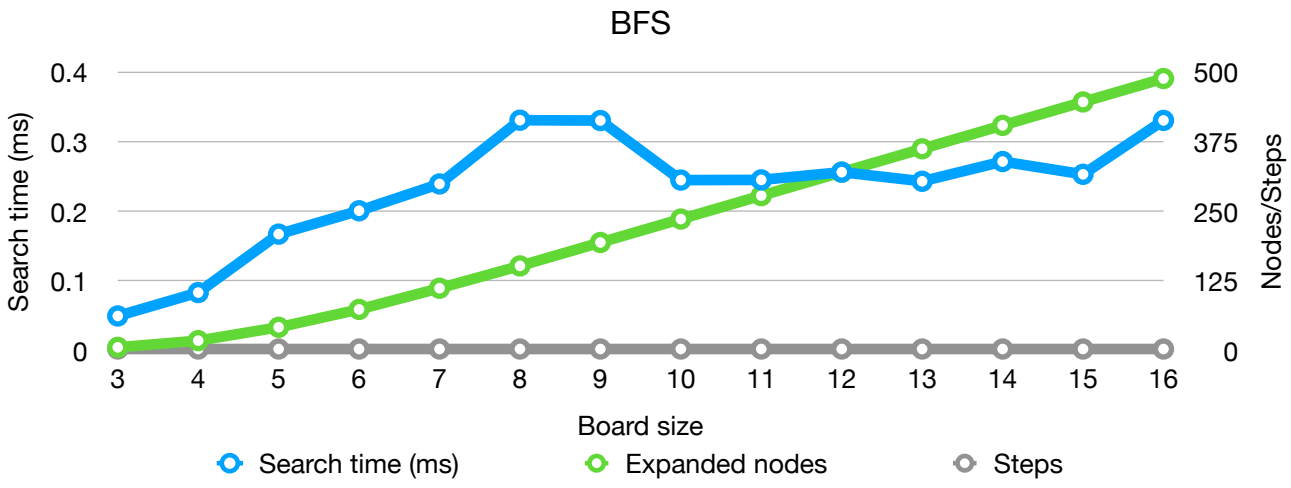
考量到版面問題因此上處僅列出 3 組結果，請注意 Y 軸所列為「比值」，綜合所有結果可以得到以下結論：

1. BFS 所「展開的節點數」通常要比 DFS 以及 A* 來的多，這應是由於 BFS 將每一層的節點皆全部展開後才到下一層搜索，因此展開的節點非常詳盡，搜尋時間也較多。
2. DFS 所找到的路徑其「步數」通常較其他演算法多出許多，歸咎其原因應是由於 DFS 在搜尋路徑時優先展開最深層的節點，因此會有「能走一步就走一步」的表現，導致其往往繞了一大圈才到目標。除了 DFS 外，其餘所有演算發幾乎都能以最短步數找到路徑。
3. IDS 藉由遞迴限制搜索深度來避免搜尋深度過深，從「(0, 0) to (0, 1)」的比較中，可以清楚看到 IDS 明顯優於 DFS 的結果，因為 IDS 有效避免了過深的搜索。然而，當起點與目標距離較遠時， IDS 反而可能不如 DFS ，因需要搜尋至較深的節點才能找到路徑，但 IDS 會先在較淺的節點反覆搜索，造成浪費，這一點由「(0, 0) to (7, 7)」的比較中可以看出。
4. A* 普遍優於 BFS ，應是由於 A* 的 Heuristic function 使得 A* 得以優先搜尋離目標較近的節點，也就是說 A* 能利用先天知識來往目標方向搜尋，以此減少搜尋不必要的節點。
5. IDA* 在所有結果中都要比 A* 的表現來得差，我認為這是因為 A* 中「往目標方向搜尋」的特性必然是正確的，因此 A* 並不會無意義的在較深的節點中搜索，也就是說 IDA* 藉由限制搜索深度來避免搜尋深度過深的措施反而造成其無意義的重複搜索。

盤面大小比較

以下為「盤面大小」對於「搜尋時間」、「展開的節點數」、「步數」的影響統計折線圖，其 X 軸為盤面大小之邊長，左方 Y 軸為搜尋時間，右方 Y 軸為節點數或步數。每組測試都是將棋子從起點 (0, 0) 移動至目標 (2, 2)。

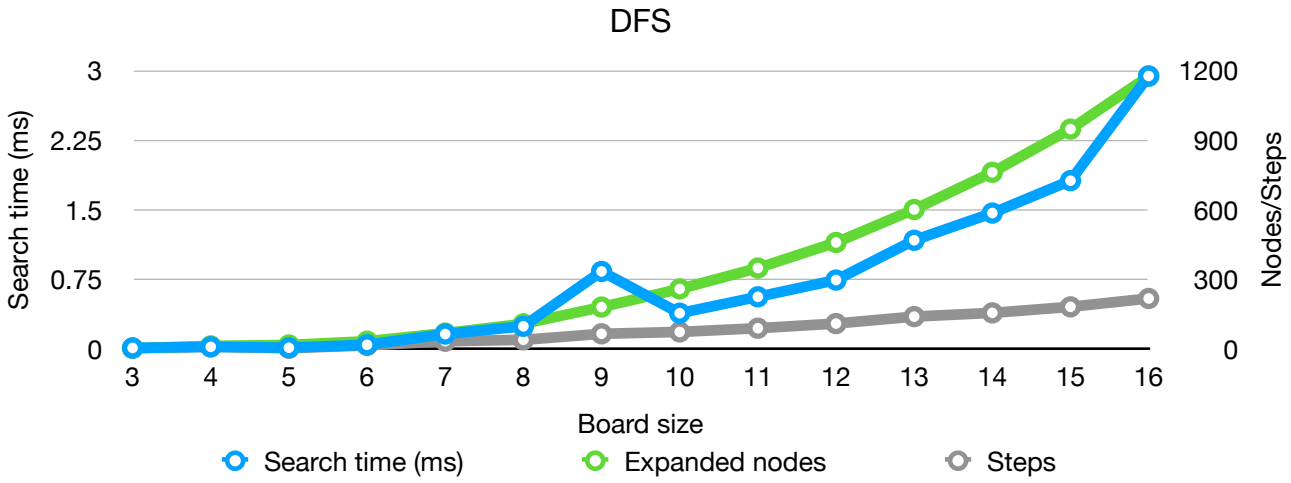
BFS Search



BFS 總是能找出的最短路徑，其搜索時間在盤面大小為 8*8 之後開始趨於穩定，但所展開的節點數量隨著每增加一盤面邊長大約增加 40 個節點個數，這個數量要比 IDS 以及 A* 要

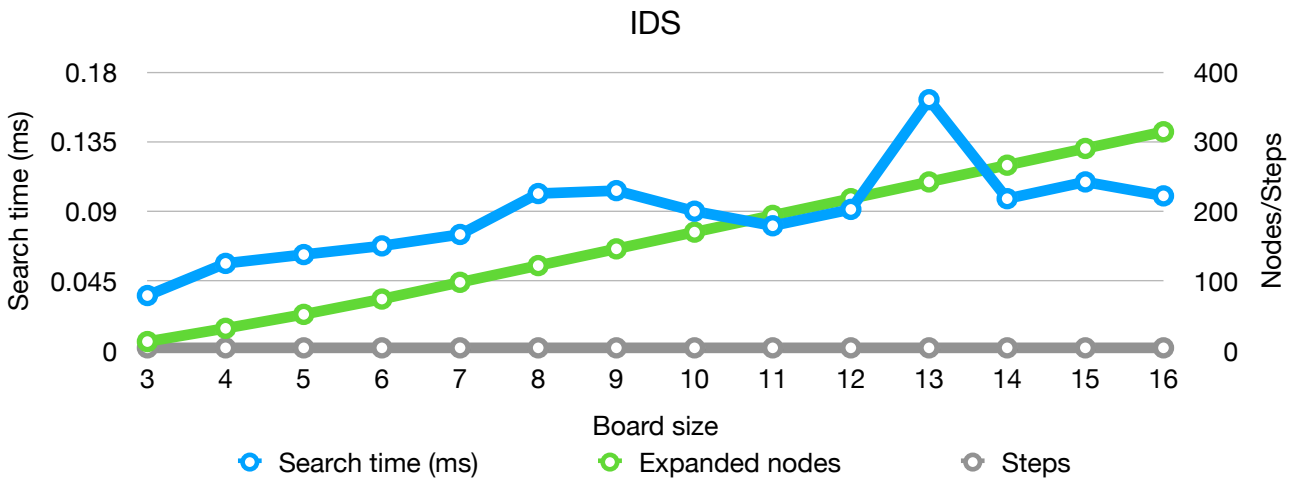
明顯來得高。我認為 BFS 的搜尋時間能趨緩是因為起點至終點的距離隨著盤面增加而始終不變，因此相對來說是縮短所產生的結果。

DFS Search



DFS 的所展開的節點數量在盤面小時表現出色，但大約在盤面大小為 8*8 之後開始超越 BFS ，而後開始呈現指數的增加，其搜索時間隨盤面的增加也有類似的表現。因此我認為 DFS 可能不適合大範圍的搜尋。此外， DFS 並不能找出最短路徑，反而是隨著盤面增加，其自起點走至終點所需之步數亦隨之增加。

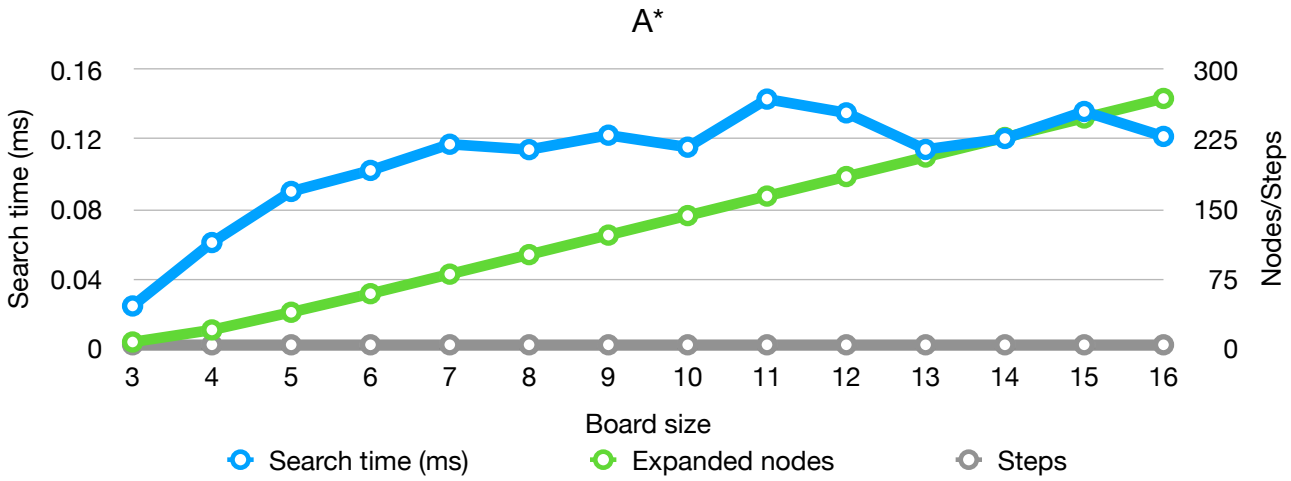
IDS Search



IDS 總是能找出最短路徑，其搜索時間除了 13*13 的盤面之外，在盤面大小為 8*8 之後大致趨於穩定，且所需時間僅約為 BFS 的三成，其所展開的節點數量也要比 BFS 來得少，但我認為這是由於起點至終點的距離較短才產生的結果，如果起點至終點的距離拉長， IDS 則不見得會有優於 BFS 的表現。

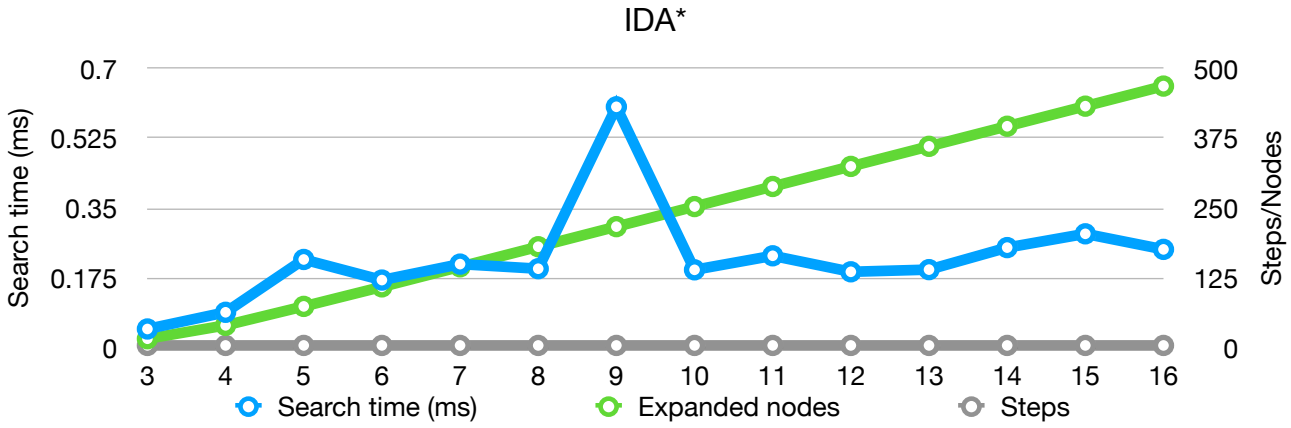
另外，值得一提的是，當盤面為 13* 13 時，搜尋時間突然增加近一倍，而後又減少，我原先認為這是因為作業系統的 Context switch 所造成的結果，但反覆實驗數次，其結果皆大致相同，這是我認為相當疑惑的地方。

A* Search



A* 總是能找出最短路徑，其搜索時間在盤面大小為 7*7 之後亦大致趨於穩定，且所需時間相較於 IDS 更為優越，其所展開的節點數量與 IDS 相比大致相同，在盤面大小為 16*16 時展開約不到 300 個節點。

IDA* Search



IDA* 總是能找出最短路徑，其搜索時間除了 9*9 的盤面之外，在盤面大小為 5*5 之後大致趨於穩定，然而其所需時間以及展開的節點數量與 A* 相比都較差，在盤面大小為 16*16 時展開約 500 個節點比 A* 來得多。

IDA* 在盤面大小為 9*9 時也有時間異常突出的問題。

所學

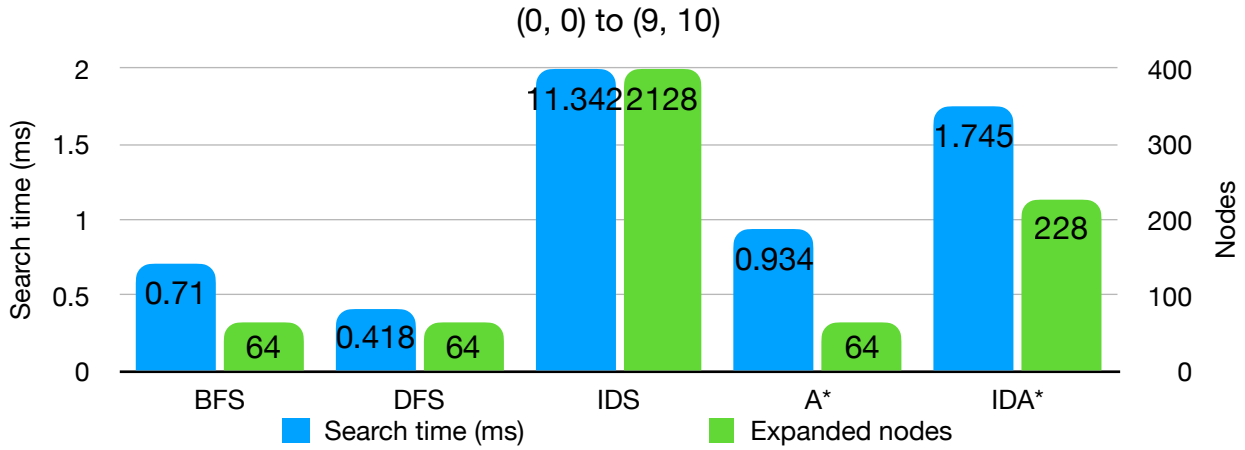
透過本次實驗我所學以下幾點：

1. 利用物件導向程式設計，以 Python 實作棋盤之 Framework。
2. 了解 5 種演算法的運作原理、特性以及優缺點。
3. 透過比較不同的盤面大小來體會時間及空間複雜度對於演算法效能的影響。
4. 將實驗結果整理、列表並使人易於閱讀。

疑問與探討

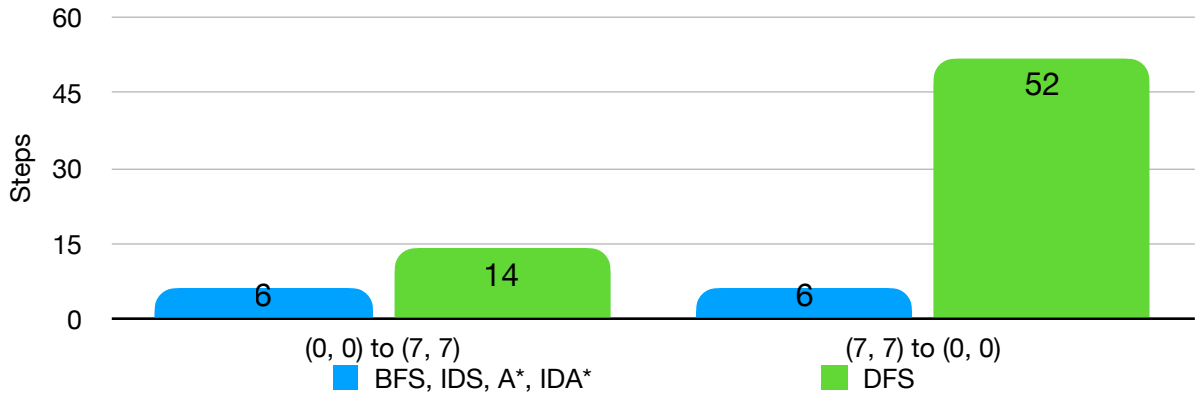
1. IDS 以及 IDA* 在比較盤面大小的影響時，皆存在時間突然增加又減少的問題，我試著以 Context switch 、 Cache 以及演算法本身的特性去思考，都無法想到合理的解釋。

2. 在 8*8 的棋盤上，若不加上任何限制，不可能會存在起點走不到終點的狀況，但我很好奇當路徑不存在時，演算法需要多久才能發現。因此我設計從 (0, 0) 走到 (9, 10)，也就是盤面上不存在的點，並繞過合理位置的檢查，實驗結果如下：



可以看出在沒有解的情況下， IDS 以及 IDA* 兩種演算法需要花費較長的時間才能進行確認，我想這是因為遞迴限制深度的影響使得這兩個演算法需要遞迴多次才能展開全部節點並發現解不存在。

3. DFS 演算法所找出來的路徑之步數似乎也受起終點方向影響：



可以看到左方為自 (0, 0) 到 (7, 7)，右方為自 (7, 7) 到 (0, 0)，其他演算法找出的兩者路徑的步數都是 6 步，但 DFS 所找到的路徑其步數 14 和 52 有明顯的不同，並且都不是最短路徑。

未來發想

以上的做法的是從起點開始不斷展開節點，直到發現終點，我的想法是：如果可以從起點和終點雙邊同時開始展開，中間若發現有相同的葉節點即是找到路徑。如此一來可能可以避免展開很多不必要的節點，或許可以有效減少空間複雜度。但是這個方法需要比較兩邊的所有葉節點或者 Frontier 是否存在相同者，這樣的比較可能需要 $O(n^2)$ 的時間複雜度，因此在時間上或許不具備優勢。

Appendix

Structure

- knight.py
- agent.py
- board.py
- tree.py
- test.py

knight.py

```
"""
    This is the main program
    Usage: knight.py <algorithm> <starting_x> <starting_y> <goal_x> <goal_y>
        0   BFS
        1   DFS
        2   IDS
        3   A*
        4   IDA*
"""

import sys
from board import board
from board import position
from agent import agent
from agent import bfs
from agent import dfs
from agent import ids
from agent import astar
from agent import idastar

def usage():
    print("usage: knight.py <algorithm> <starting_x> <starting_y> <goal_x>
<goal_y>")
    print("\t0\tBFS")
    print("\t1\tDFS")
    print("\t2\tIDS")
    print("\t3\tA*")
    print("\t4\tIDA*")
    return

""" This is the main function of the program """
def knight(algorithm_type, starting_x, starting_y, goal_x, goal_y):
```

```
    # Create a board of size 8
    b = board(8)

    # Set starting position and goal position
    p_start = position(starting_x, starting_y)
    p_goal = position(goal_x, goal_y)

    # Check if the positions are available
    if not p_start.is_available_pos(b) or not p_goal.is_available_pos(b):
        print("position out of board range")
        return

    # Create agent
    a = agent(p_start, p_goal)

    # Classify the type of agent
    if algorithm_type == 0:
        a = bfs(p_start, p_goal)
    elif algorithm_type == 1:
        a = dfs(p_start, p_goal)
    elif algorithm_type == 2:
        a = ids(p_start, p_goal)
    elif algorithm_type == 3:
        a = astar(p_start, p_goal)
    elif algorithm_type == 4:
        a = idastar(p_start, p_goal)
    else:
        usage()
        return

    # Search path
    path_list = a.search(b)

    # Print path
    for path in path_list:
        print(path, end="")
    print()

    # Return the number of expanded nodes
    return a.expanded_node_count


if __name__ == '__main__':
```

```
    if len(sys.argv) != 6:
        usage()
    else:
        algorithm_type = int(sys.argv[1])
        starting_x = int(sys.argv[2])
        starting_y = int(sys.argv[3])
        goal_x = int(sys.argv[4])
        goal_y = int(sys.argv[5])

        knight(algorithm_type, starting_x, starting_y, goal_x, goal_y)

def usage():
    print("usage: knight.py <algorithm> <starting_x> <starting_y> <goal_x>
<goal_y>")
    print("\t0\tBFS")
    print("\t1\tDFS")
    print("\t2\tIDS")
    print("\t3\tA*")
    print("\t4\tIDA*")
    return

""" This is the main function of the program """
def knight(algorithm_type, starting_x, starting_y, goal_x, goal_y):
    # Create a board of size 8
    b = board(8)

    # Set starting position and goal position
    p_start = position(starting_x, starting_y)
    p_goal = position(goal_x, goal_y)

    # Check if the positions are available
    if not p_start.is_available_pos(b) or not p_goal.is_available_pos(b):
        print("position out of board range")
        return

    # Create agent
    a = agent(p_start, p_goal)

    # Classify the type of agent
    if algorithm_type == 0:
        a = bfs(p_start, p_goal)
    elif algorithm_type == 1:
        a = dfs(p_start, p_goal)
```

```
    elif algorithm_type == 2:
        a = ids(p_start, p_goal)
    elif algorithm_type == 3:
        a = astar(p_start, p_goal)
    elif algorithm_type == 4:
        a = idastar(p_start, p_goal)
    else:
        usage()
        return

    # Search path
    path_list = a.search(b)

    # Print path
    for path in path_list:
        print(path, end="")
    print()

    # Return the number of expanded nodes
    return a.expanded_node_count


if __name__ == '__main__':
    if len(sys.argv) != 6:
        usage()
    else:
        algorithm_type = int(sys.argv[1])
        starting_x = int(sys.argv[2])
        starting_y = int(sys.argv[3])
        goal_x = int(sys.argv[4])
        goal_y = int(sys.argv[5])

        knight(algorithm_type, starting_x, starting_y, goal_x, goal_y)
```

agent.py

```
"""
    This program defines agents of each algorithm
"""

import math
import queue
import bisect
```

```python
from board import board
from board import position
from tree import node

def heuristic(pos, goal):
    # This function returns the value of heuristic function
    return int( (abs(pos.x-goal.x) + abs(pos.y-goal.y)) / 3 )

def estimated_cost(g, node, goal):
    # This function returns the value of estimated cost
    return g + heuristic(node.position, goal)

def path_list(goal_node):
    # This function returns a list of positions of the
    # path moving from starting position to goal position
    # by order
    ret = []
    tmp_node = goal_node
    while tmp_node != None:
        ret.append(tmp_node.position)
        tmp_node = tmp_node.parent
    ret.reverse()
    return ret


class agent:
    def __init__(self, start_, goal_):
        self.expanded_node_count = 0
        self.start = start_
        self.goal = goal_

    def add_expanded_node(self):
        self.expanded_node_count += 1


class bfs(agent):
    def __init__(self, start_, goal_):
        super(bfs, self).__init__(start_, goal_)

    def __str__(self):
        return "BFS"

    def __repr__(self):
```

```
        return self.__str__()

    def search(self, b):
        # Create the root node
        root = node(None, self.start)

        # Initial explored set and frontier
        # Frontier: queue
        explorered_set = []
        frontier = queue.Queue(maxsize = -1)
        frontier.put(root)

        while not frontier.empty():
            # Expand shallowest unexpanded node
            cur_node = frontier.get()

            # Return when path is found
            if cur_node.position == self.goal:
                return path_list(cur_node)
            if cur_node.position in explorered_set:
                continue

            possible_moves = cur_node.position.available_moves(b)
            self.add_expanded_node()
            explorered_set.append(cur_node.position)

            for new_pos in possible_moves:
                # Create child node and append to parent
                child = node(cur_node, new_pos)
                cur_node.add_child(child)

                # Set frontier
                frontier.put(child)
        return []


class dfs(agent):
    def __init__(self, start_, goal_):
        super(dfs, self).__init__(start_, goal_)

    def __str__(self):
        return "DFS"
```

```
    def __repr__(self):
        return self.__str__()

    def search(self, b):
        # Create the root node
        root = node(None, self.start)

        # Initial explored set and frontier
        # Frontier: stack
        explorered_set = []
        frontier = [root]

        while len(frontier):
            # Expand the deepest (most recent) unexpanded node
            cur_node = frontier.pop()

            # Return when path is found
            if cur_node.position == self.goal:
                return path_list(cur_node)
            if cur_node.position in explorered_set:
                continue

            possible_moves = cur_node.position.available_moves(b)
            self.add_expanded_node()
            explorered_set.append(cur_node.position)

            for new_pos in possible_moves:
                # Create child node and append to parent
                child = node(cur_node, new_pos)
                cur_node.add_child(child)

                # Set frontier
                frontier.append(child)
        return []


class ids(agent):
    def __init__(self, start_, goal_):
        super(ids, self).__init__(start_, goal_)

    def __str__(self):
        return "IDS"
```

```
def __repr__(self):
    return self.__str__()

def search(self, b):
    # Initial depth limit
    depth_limit = -1

    while True:
        # Set depth limit
        depth_limit += 1

        # Create the root node
        root = node(None, self.start, 0)

        # Initial explored set and frontier
        # Frontier: stack
        explorered_set = []
        frontier = [root]

        while len(frontier):
            # Expand the deepest (most recent) unexpanded node
            # where depth not greater than depth limit
            cur_node = frontier.pop()

            # Return when path is found
            if cur_node.position == self.goal:
                return path_list(cur_node)
            if cur_node.depth == depth_limit:
                continue
            if cur_node.position in explorered_set:
                continue

            possible_moves = cur_node.position.available_moves(b)
            self.add_expanded_node()
            explorered_set.append(cur_node.position)

            for new_pos in possible_moves:
                # Create child node and append to parent
                child = node(cur_node, new_pos, cur_node.depth+1)
                cur_node.add_child(child)

                # Set frontier
                frontier.append(child)
```

```
                    if len(explorered_set) >= math.pow(b.size, 2):
                        return []
        return []


class astar(agent):
    def __init__(self, start_, goal_):
        super(astar, self).__init__(start_, goal_)

    def __str__(self):
        return "A*"

    def __repr__(self):
        return self.__str__()

    def search(self, b):
        # Create the root node
        root = node(None, self.start, 0)

        # Initial explored set and frontier
        # Frontier: priority queue
        explorered_set = []
        frontier = []
        pair = (estimated_cost(root.depth, root, self.goal), root)
        bisect.insort(frontier, pair)

        while len(frontier):
            # Expand the unexpanded node with the lowest
            # estimated total path cost
            cur_node = frontier.pop(0)[1]

            # Return when path is found
            if cur_node.position == self.goal:
                return path_list(cur_node)
            if cur_node.position in explorered_set:
                continue

            possible_moves = cur_node.position.available_moves(b)
            self.add_expanded_node()
            explorered_set.append(cur_node.position)

            for new_pos in possible_moves:
```

```
                # Create child node and append to parent
                child = node(cur_node, new_pos, cur_node.depth+1)
                cur_node.add_child(child)

                # Set frontier
                pair = (estimated_cost(child.depth, child, self.goal), child)
                bisect.insort(frontier, pair)
        return []


class idastar(agent):
    def __init__(self, start_, goal_):
        super(idastar, self).__init__(start_, goal_)

    def __str__(self):
        return "IDA*"

    def __repr__(self):
        return self.__str__()

    def search(self, b):
        # Initial depth limit
        depth_limit = -1

        while True:
            # Set depth limit
            depth_limit += 1

            # Create the root node
            root = node(None, self.start, 0)

            # Initial explored set and frontier
            # Frontier: priority queue
            explorered_set = []
            frontier = []
            pair = (estimated_cost(root.depth, root, self.goal), root)
            bisect.insort(frontier, pair)

            while len(frontier):
                # Expand the unexpanded node with the lowest
                # estimated total path cost
                # where depth not greater than depth limit
                cur_node = frontier.pop(0)[1]
```

```
                # Return when path is found
                if cur_node.position == self.goal:
                    return path_list(cur_node)
                if cur_node.depth == depth_limit:
                    continue
                if cur_node.position in explorered_set:
                    continue

                possible_moves = cur_node.position.available_moves(b)
                self.add_expanded_node()
                explorered_set.append(cur_node.position)

                for new_pos in possible_moves:
                    # Create child node and append to parent
                    child = node(cur_node, new_pos, cur_node.depth+1)
                    cur_node.add_child(child)

                    # Set frontier
                    pair = (estimated_cost(child.depth, child, self.goal),
child)
                    bisect.insort(frontier, pair)


                    if len(explorered_set) >= math.pow(b.size, 2):
                        return []
        return []


if __name__ == '__main__':
    # Examples
    b = board(8)
    p_start = position(0, 0)
    p_goal = position(2, 2)

    a = bfs(p_start, p_goal)
    path = a.search(b)
    b.print_pathway(path)
    print(a.expanded_node_count)

    a = dfs(p_start, p_goal)
    path = a.search(b)
    b.print_pathway(path)
```

```
    print(a.expanded_node_count)

    a = ids(p_start, p_goal)
    path = a.search(b)
    b.print_pathway(path)
    print(a.expanded_node_count)

    a = astar(p_start, p_goal)
    path = a.search(b)
    b.print_pathway(path)
    print(a.expanded_node_count)

    a = idastar(p_start, p_goal)
    path = a.search(b)
    b.print_pathway(path)
    print(a.expanded_node_count)
```

board.py

```
"""
    This program defines the framework of the board
"""

import math


class board:
    def __init__(self, size_):
        self.size = size_

    def print_pathway(self, path_list):
        # This function prints the path from the starting
        # position to the goal position on this board
        # with no return value
        ls = []
        for i in range(int(math.pow(self.size, 2))):
            ls.append("  +")
        for i in range(len(path_list)):
            pos = path_list[i]
            if i == 0:
                ls[ pos.y * self.size + pos.x ] = "  S"
            elif i == len(path_list) - 1:
                ls[ pos.y * self.size + pos.x ] = "  G"
```

```
            else:
                ls[ pos.y * self.size + pos.x ] = "{: 3}".format(i)
        print("   ", end="")
        for i in range(self.size):
            print("{: 3}".format(i), end="")
        print()
        for i in range(int(math.pow(self.size, 2))):
            if not i % self.size:
                print("{: 3}".format(int(i/self.size)), end="")
            print(ls[i], end="")
            if not (i+1) % self.size:
                print()


class move:
    def __init__(self, move_x = 0, move_y = 0):
        self.x = move_x
        self.y = move_y


class position:
    def __init__(self, position_x = -1, position_y = -1):
        self.x = position_x
        self.y = position_y

    def __str__(self):
        return ("({}, {})" .format(str(self.x), str(self.y)))

    def __repr__(self):
        return self.__str__

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __ne__(self, other):
        return not ( __eq__(self, other) )

    def __lt__(self, other):
        return self.x < other.x or (self.x == other.x and self.y < other.y)

    def is_available_pos(self, b):
        # This function returns True if this position is
        # available on the given board; otherwise False
```

```
        return 0 <= self.x < b.size and 0 <= self.y < b.size

    def available_moves(self, b):
        # This function returns a list of all possible
        # positions on the given board after taking
        # a single move from this position
        new_pos_list = []
        moves = [move(-1, -2), move(1, -2), move(-2, -1), move(2, -1),
move(-2, 1), move(2, 1), move(-1, 2), move(1, 2)]

        for mv in moves:
            new_x = self.x + mv.x
            new_y = self.y + mv.y
            new_pos = position(new_x, new_y)
            if new_pos.is_available_pos(b):
                new_pos_list.append(new_pos)

        return new_pos_list


if __name__ == '__main__':
    # Examples
    b = board(8)
    p = position(5, 5)
    next_p = p.available_moves(b)
    for n in next_p:
        print(n)
```

tree.py

```
"""
    This program defines the tree
"""

from board import position


class node:
    def __init__(self, parent_, position_, depth_ = -1):
        self.parent = parent_
        self.position = position_
        self.childs = []
```

```
        self.depth = depth_

    def __str__(self):
        return self.position.__str__()

    def __repr__(self):
        return self.position.__str__()

    def __lt__(self, other):
        return self.position < other.position

    def add_child(self, child_node):
        self.childs.append(child_node)
```

test.py

```
"""
    All test results from the report can be shown by running this program
"""
import time
from board import board
from board import position
from agent import agent
from agent import bfs
from agent import dfs
from agent import ids
from agent import astar
from agent import idastar

def algorithm_type_test():
    print("======== Algorithm type comparison ========")

    b = board(8)
    pairs = [(position(2, 2), position(4, 4)),
             (position(4, 4), position(2, 2)),
             (position(0, 0), position(7, 7)),
             (position(7, 7), position(0, 0)),
             (position(0, 0), position(0, 1)),
             (position(0, 1), position(0, 0)),
             (position(0, 0), position(9, 10))]

    for pair in pairs:
```

```
        p_start = pair[0]
        p_goal = pair[1]
        agents = [bfs(p_start, p_goal), dfs(p_start, p_goal), ids(p_start,
p_goal), astar(p_start, p_goal), idastar(p_start, p_goal)]
        bm = [-1, -1, -1]

        print("From", p_start, "to", p_goal)
        print("Algorithm\tSearch time (ms)\tSteps\t\tExpanded nodes")
        for a in agents:
            start_time = time.time()
            path = a.search(b)
            search_time = (time.time() - start_time) * 100
            steps = len(path) - 1

            bm[0] = search_time if bm[0] == -1 else bm[0]
            bm[1] = steps if bm[1] == -1 and steps != 0 else bm[1]
            bm[2] = a.expanded_node_count if bm[2] == -1 and
a.expanded_node_count else bm[2]

            print("{}\t\t{:.6f} ({:.3f})\t{} ({:.3f})\t{} ({:.3f})" .format(a,
search_time, search_time/bm[0], steps, steps/bm[1], a.expanded_node_count,
a.expanded_node_count/bm[2]))
        print()

def board_size_test():
    print("======== Board size comparison ========")

    p_start = position(0, 0)
    p_goal = position(2, 2)
    agents = [bfs(p_start, p_goal), dfs(p_start, p_goal), ids(p_start,
p_goal), astar(p_start, p_goal), idastar(p_start, p_goal)]

    for a in agents:
        print(a)
        print("Board size\tSearch time (ms)\tSteps\t\tExpanded nodes")
        bm = [-1, -1, -1, -1]

        for b_size in range(3, 17):
            b = board(b_size)

            start_time = time.time()
            path = a.search(b)
            search_time = (time.time() - start_time) * 100
```

```
            steps = len(path) - 1

            bm[0] = b_size if bm[0] == -1 else bm[0]
            bm[1] = search_time if bm[1] == -1 else bm[1]
            bm[2] = steps if bm[2] == -1 and steps != 0 else bm[2]
            bm[3] = a.expanded_node_count != 0 if bm[3] == -1 and
a.expanded_node_count else bm[3]

            print("{} ({:.3f})\t{:.6f} ({:.3f})\t{} ({:.3f})\t{}
({:.3f})" .format(b_size, b_size/bm[0], search_time, search_time/bm[1], steps,
steps/bm[2], a.expanded_node_count, a.expanded_node_count/bm[3]))
        print()


if __name__ == '__main__':
    board_size_test()
    algorithm_type_test()
```