AI Coursework Report

Zhou Jie

31384579

jz5n19@soton.ac.uk

1. **Approach**

1.1, Common components

Language

The 4 algorithms were developed in Python.

Data structure

The state in this problem was built in python dictionary with the names of blocks and agent as keys and the corresponding coordinates as the value. For example, the initial state is: {"A":[4,1],"B":[4,2],"C":[4,3],"Agent":[4,4]}

The coordinates represent the number of rows and columns, like block A is placed in row 4 column 1 in that state.

The node was also built in python dictionary, with a series of keys include 'state', 'path_cost', ' depth',' parent_node', and the corresponding values, for example the root node is: {"state":initial_state,"path_cost":0,"depth":0,"parent_node":None}

Function

Goal test function: By comparing the positions/coordinates of the 3 blocks in current and goal state.

Action function: Consists of the movements in 4 directions. The movements were restricted within the size of grid, and when the agent meet the block, they exchange their position. The parameter of size was included in the function definition for the use of scalability.

1.2, Algorithms

BFS

The BFS was implemented according to the pseudo code(snapshot below) on page82 of the textbook *Artificial Intelligence A Modern Approach* Third Edition*,* but without the line in the red box, because it avoids exploring repeat states that belongs to the graph version:

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

The frontier FIFO queue was implemented by building a list and poping the node with the smallest depth in it.

DFS

The implementation of DFS was the same as BFS except the frontier list. DFS pops the node with the largest depth in every iteration instead.

IDS

I implemented depth limited search first, then I iterate different depth limits to implement IDS. I followed the pseudo code on P88 *A recursive implementation of depth-limited tree search*. in the book *Artificial Intelligence A Modern Approach* Third Edition. Additionally, I added an list to record the explored nodes.

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
     cutoff_occurred? ← false
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        result ← RECURSIVE-DLS(child, problem, limit − 1)
        if result = cutoff then cutoff_occurred? ← true
        else if result ≠ failure then return result
     if cutoff_occurred? then return cutoff else return failure
```

Figure 3.17 A recursive implementation of depth-limited tree search.

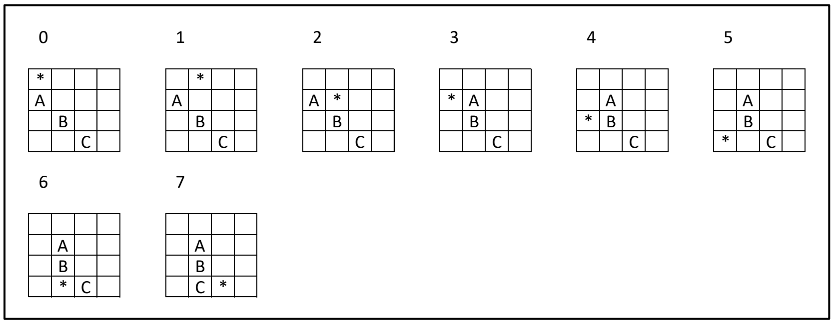
A*

In A*, I built the heuristic function h(n) by using Manhattan distance. It is the sum of the Manhattan distances of every block to the final position in the goal state. It’s no greater than the steps taken by the agent to move every block to the final position e.g. it’s admissible.

The difference between the implementation of A* and BFS was the order of poping the node in frontier list. My code pops the node with the smallest value of the sum of the path cost g(n)+the heuristic h(n) first.

2. **Evidence**
I changed the initial state to make it closer to the goal state, otherwise the algorithms took too much time to get the solution in the tree search version. The initial state I took was:
{"A":[2,1],"B":[3,2],"C":[4,3],"Agent":[1,1]}

2.1, BFS



The solution found by BFS took 7 steps to reach the goal state. Since the step costs in this problem are all identical, it should be the optimal solution.

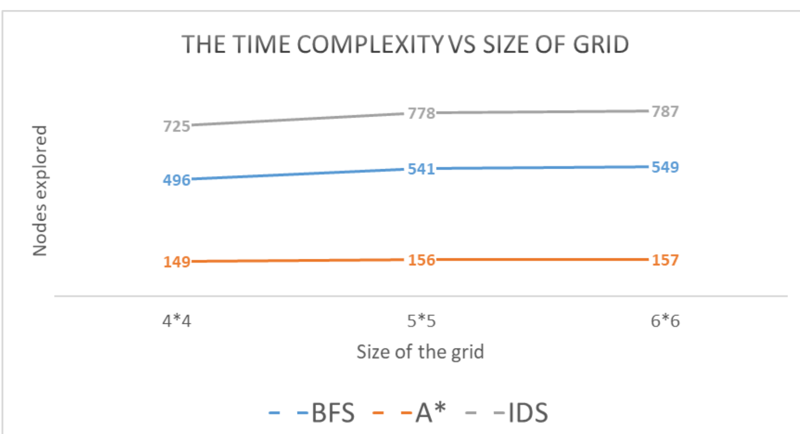
2.2, DFS
Solution not found (took too much time and I interrupted the program)

2.3, IDS
The solution found by IDS was exactly the same as BFS. Since the step costs in this problem are all identical, the solution found by IDS was also the optimal solution.

2.4, A*
The solution found by A* was also the same as BFS. Since my heuristic function is admissible. So, it could get the optimal solution.

3. **Scalability**
I controlled the problem difficulty by changing the size of the grid, I chose 3 sizes : 4*4, 5*5, 6*6 to compare the time complexity.

The graph showed the relationship (No solution found in DFS):



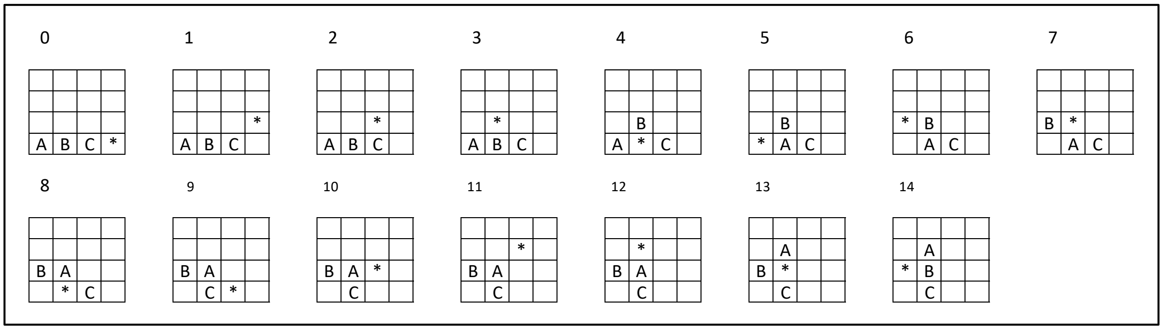
Obviously, the uninformed tree search method (BFS, IDS) took much time to find the solution than the informed one(A*).

4. **Extras and limitations**

4.1, Extras:

Graph search

1, I found the solution by graph search version of **BFS** with original initial state:



2, I found the solution by graph search version of **DFS** with original initial state, but with the depth of 12841 and explored nodes of 13566.

Heuristic function

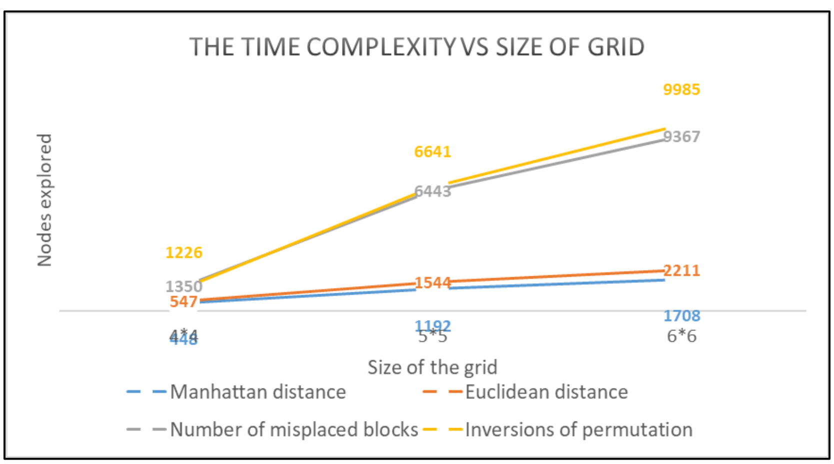
I tried another 3 kinds of heuristic functions in graph version A* algorithms beyond Manhattan distance:

1, Euclidean distance: By calculating the Euclidean distance of the coordinates of each block between current state and goal state and sum them up.

2, Number of misplaced blocks: By counting the number of blocks with different positions compare with the goal state.

3, Inversions of permutation of the blocks' row coordinate: The row coordinates of blocks in the goal state should be in ascending order i.e. {"A":[2,2],"B":[3,2],"C":[4,2]}. By counting the number of inversions of blocks' row coordinates in the current states is a way to quantify the distance to the goal state.

The relationship between time complexity and size of grid in different heuristic functions is like following (with original start state{"A":[4,1],"B":[4,2],"C":[4,3],"Agent":[4,4]}):



According to the result, the Manhattan distance is the best heuristic function in the graph version of A*. I think it's because the Manhattan distance is the most accuracy description of the movements in this problem compare to other heuristic functions.

4.2, Limitations:

1, I couldn't find the reason why there were duplicate states in both explored and frontier list (graph version of BFS) when the optimal solution was found. The following code found the problem:

```
def checkIfDuplicates(listOfElems):
    ''' Check if given list contains any duplicates '''
    for elem in listOfElems:
        if listOfElems.count(elem) > 1:
            return True
    return False
```

```
print(checkIfDuplicates([i['state'] for i in explored ]))
print(checkIfDuplicates([i['state'] for i in frontier ]))
print(checkIfDuplicates([1,1,1,2,3]))
```

```
True
True
True
```

5. **References**

1, P82 *Breadth-first search on a graph* Pseudo code in the book *Artificial Intelligence A Modern Approach Third Edition*

2, P88 *A recursive implementation of depth-limited tree search.* pseudo code in the book *Artificial Intelligence A Modern Approach Third Edition*

3, *Heuristic function* http://ai.stanford.edu/~latombe/cs121/2011/slides/D-heuristic-search.pdf

6. **Code**

```
#goal_test function
def goal_test(state):
    if state['A']==[2,2] and state['B']==[3,2] and state['C']==[4,2]:
        return True
    return False
```

```
#action function
def move_right(state,size):
    import copy
    new_state=copy.deepcopy(state)
    if state['Agent'][1]+1>=1 and state['Agent'][1]+1<=size:  #check if move will be out of box
        agent_position1=state['Agent'] #store the position of agent before moving
        new_state['Agent'][1]=new_state['Agent'][1]+1 #move the agent
        agent_position2=new_state['Agent']#store the position of agent after moving
        for block in {key:value for key,value in new_state.items() if key != 'Agent'}:#iterate every block
            if new_state[block]==agent_position2:#check if the position of block overlap the agent after moving
                new_state[block]=agent_position1#assign the position of agent before moving to the overlap block
    return new_state
def move_left(state,size):
    import copy
    new_state=copy.deepcopy(state)
    if state['Agent'][1]-1>=1 and state['Agent'][1]-1<=size:  #check if move will be out of box
        agent_position1=state['Agent'] #store the position of agent before moving
        new_state['Agent'][1]=new_state['Agent'][1]-1 #move the agent
        agent_position2=new_state['Agent']#store the position of agent after moving
        for block in {key:value for key,value in new_state.items() if key != 'Agent'}:#iterate every block
            if new_state[block]==agent_position2:#check if the position of block overlap the agent after moving
                new_state[block]=agent_position1#assign the position of agent before moving to the overlap block
    return new_state
```

```python
def move_up(state,size):
    import copy
    new_state=copy.deepcopy(state)
    if state['Agent'][0]-1>=1 and state['Agent'][0]-1<=size:  #check if move will be out of box
        agent_position1=state['Agent'] #store the position of agent before moving
        new_state['Agent'][0]=new_state['Agent'][0]-1 #move the agent
        agent_position2=new_state['Agent']#store the position of agent after moving
        for block in {key:value for key,value in new_state.items() if key != 'Agent'}:#iterate every block
            if new_state[block]==agent_position2:#check if the position of block overlap the agent after moving
                new_state[block]=agent_position1#assign the position of agent before moving to the overlap block
    return new_state
def move_down(state,size):
    import copy
    new_state=copy.deepcopy(state)
    if state['Agent'][0]+1>=1 and state['Agent'][0]+1<=size:  #check if move will be out of box
        agent_position1=state['Agent'] #store the position of agent before moving
        new_state['Agent'][0]=new_state['Agent'][0]+1 #move the agent
        agent_position2=new_state['Agent']#store the position of agent after moving
        for block in {key:value for key,value in new_state.items() if key != 'Agent'}:#iterate every block
            if new_state[block]==agent_position2:#check if the position of block overlap the agent after moving
                new_state[block]=agent_position1#assign the position of agent before moving to the overlap block
    return new_state
def action(state,size):
    right_state=move_right(state,size) if  state!=move_right(state,size) else None
    left_state=move_left(state,size) if  state!=move_left(state,size) else None
    up_state=move_up(state,size) if  state!=move_up(state,size) else None
    down_state=move_down(state,size) if  state!=move_down(state,size) else None
    results=[right_state,left_state,up_state,down_state]
    return [result for result in results if result is not None]#return only legal move results
```

```python
#1,  BFS(tree version)
def BFS(node,size):
    #check if the initial node is the goal node
    if goal_test(node['state']):
        print(node)
    # a FIFO queue with node as the only element
    frontier=[node]
    explored=[]
    goal_found=False
    # while not goal_test(node['state']):
    while goal_found==False:
    #update node and explored state

        #check if the frontier is empty
        if len(frontier)==0:
            print('failure')
        #pop the shallowest node from frontier for expansion
        node=frontier[0]
        for n in frontier:
            if n['depth']<node['depth']:
                node=n
        frontier.pop(frontier.index(node))
        #add the pop node state to the explored list
        explored.append(node)
    #update frontier node list
        for result in action(node['state'],size):
            child={"state":result,"path_cost":node['path_cost']+1,"depth":node['depth']+1,"parent_node":node}
            if goal_test(child['state']):
                goal_found=True
                print('goal found:  '+str(child['state'])+'/'+
                      ' path_cost:'+str(child['path_cost'])+'/'+
                      ' depth:'+str(child['depth'])+'/'+
                      ' parent:'+str(child['parent_node']['state'])+'/'+
                      ' exploredlen:'+str(len(explored))+
                      ' fronlen:'+str(len(frontier))

                     )
            frontier.append(child)
```

```python
#BFS(graph version)
def BFS(node,size):
    #check if the initial node is the goal node
    if goal_test(node['state']):
        print(node)
    # a FIFO queue with node as the only element
    frontier=[node]
    explored=[]
    goal_found=False
    while goal_found==False:
    #update node and explored state

        #check if the frontier is empty
        if len(frontier)==0:
            print('failure')
        #pop the shallowest node from frontier for expansion
        node=frontier[0]
        for n in frontier:
            if n['depth']<node['depth']:
                node=n
        frontier.pop(frontier.index(node))
        #add the pop node state to the explored list
        explored.append(node)
    #update frontier node list
        for result in action(node['state'],size):
            child={"state":result,"path_cost":node['path_cost']+1,"depth":node['depth']+1,"parent_node":node}
            if child['state'] not in [i['state'] for i in explored ] and child not in [i['state'] for i in frontier ]:
                #this need to be check the duplicate of the 'state' rather than the 'node',because the state can be duplica
                if goal_test(child['state']):
                    goal_found=True
                    print('goal found: '+str(child['state'])+'/'+
                          ' path_cost:'+str(child['path_cost'])+'/'+
                          ' depth:'+str(child['depth'])+'/'+
                          ' parent:'+str(child['parent_node']['state'])+'/'+
                          ' exploredlen:'+str(len(explored))+
                          ' fronlen:'+str(len(frontier))

                         )


                frontier.append(child)
```

```python
# 2, DFS(tree version)
def DFS(node,size):
    #check if the initial node is the goal node
    if goal_test(node['state']):
        print(node)
    # a LIFO queue with node as the only element
    frontier=[node]
    explored=[]
    goal_found=False
    while goal_found==False:
    #update node and explored state

        #check if the frontier is empty
        if len(frontier)==0:
            print('failure')
        #pop the deepest node from frontier for expansion
        node=frontier[0]
        for n in frontier:
            if n['depth']>node['depth']:
                node=n
        frontier.pop(frontier.index(node))
        #add the pop node state to the explored list
        explored.append(node)
    #update frontier node list
        for result in action(node['state'],size):
            child={"state":result,"path_cost":node['path_cost']+1,"depth":node['depth']+1,"parent_node":node}
            if goal_test(child['state']):
                    goal_found=True
                    print('goal found: '+str(child['state'])+'/'+
                          ' path_cost:'+str(child['path_cost'])+'/'+
                          ' depth:'+str(child['depth'])+'/'+
                          ' parent:'+str(child['parent_node']['state'])+'/'+
                          ' exploredlen:'+str(len(explored))+
                          ' fronlen:'+str(len(frontier))

                         )
                    return child


            frontier.append(child)
```

```python
#DFS(graph version)
def DFS(node,size):
    #check if the initial node is the goal node
    if goal_test(node['state']):
        print(node)
    # a LIFO queue with node as the only element
    frontier=[node]
    explored=[]
    goal_found=False
    #for i in range(4):
    while goal_found==False:
    #update node and explored state
        #check if the frontier is empty
        if len(frontier)==0:
            print('failure')
        #pop the deepest node from frontier for expansion
        node=frontier[0]
        for n in frontier:
            if n['depth']>node['depth']:
                node=n
        frontier.pop(frontier.index(node))
        #add the pop node state to the explored list
        explored.append(node)
    #update frontier node list
        for result in action(node['state'],size):
            child={"state":result,"path_cost":node['path_cost']+1,"depth":node['depth']+1,"parent_node":node}
            if child['state'] not in [i['state'] for i in explored ] and child not in [i['state'] for i in frontier ]:
            #this need to be check the duplicate of the 'state' rather than the 'node',because the state can be duplicate i
                if goal_test(child['state']):
                    goal_found=True
                    print('goal found: '+str(child['state'])+'/'+
                          ' path_cost:'+str(child['path_cost'])+'/'+
                          ' depth:'+str(child['depth'])+'/'+
                          ' parent:'+str(child['parent_node']['state'])+'/'+
                          ' exploredlen:'+str(len(explored))+
                          ' fronlen:'+str(len(frontier))

                         )
                    return child
                frontier.append(child)
```

```python
#3, IDS
def recursive_dls(node,limit,size):
    #check if the initial node is the goal node
    if goal_test(node['state']):
        return node['state'],node['path_cost'],node['depth'],len(explored)
    elif limit==0:
        return 'cut off'
    else:
        explored.append(node)
        cutoff=False

        for i in action(node['state'],size):
            child={"state":i,"path_cost":node['path_cost']+1,"depth":node['depth']+1,"parent_node":node}
            result=recursive_dls(child,limit-1,size)
            if result=='cut off':
                cutoff=True
            elif result!='failure':
                return result
        if cutoff==True:
            return 'cut off'
        else:
            return 'failure'

#Iterative deepening search with recursive_dls
def ids(node,size):
    limit=0
    stop=False
    while stop==False:
        result=recursive_dls(node,limit,size)
        if result!='cut off':
            return result
            stop=True
        limit=limit+1
```

```
#4, A*
#hueristic function:
def h(state,htype):
    #manhattan distance from current state to goal state of each block
    if htype=='md2goal':
        a=abs(state['A'][0]-2)+abs(state['A'][1]-2)
        b=abs(state['B'][0]-3)+abs(state['B'][1]-2)
        c=abs(state['C'][0]-4)+abs(state['C'][1]-2)
        return a+b+c

    #euclidean distance from current state to goal state of each block
    if htype=='ed2goal':
        a=(abs(state['A'][0]-2)**2+abs(state['A'][1]-2)**2)**0.5
        b=(abs(state['B'][0]-3)**2+abs(state['B'][1]-2)**2)**0.5
        c=(abs(state['C'][0]-4)**2+abs(state['C'][1]-2)**2)**0.5
        return a+b+c

    #misplaced blocks
    if htype=='misplaced':
        l1=[state['A'],state['B'],state['C']]
        l2=[[2,2],[3,2],[4,2]]
        m=0
        for i in range(3):
            if l1[i]!=l2[i]:
                m=m+1
        return m

    #inversions of permutation of the blocks' row coordinate
    if htype=='inver_p':
        l=[[state['A'][0],state['B'][0]],[state['A'][0],state['C'][0]],[state['B'][0],state['C'][0]]]
        inv=0
        for i in l:
            if i[0]>=i[1]:
                inv=inv+1
        return inv
```

```
# A*(tree version)
def A_star(node,htype,size):
    #check if the initial node is the goal node
    if goal_test(node['state']):
        print(node)
    # a queue with node as the only element
    frontier=[node]
    explored=[]
    goal_found=False
    while goal_found==False:
    #update node and explored state

        #check if the frontier is empty
        if len(frontier)==0:
            print('failure')
        #pop the node with the lowest evaluation function from frontier for expansion
        node=frontier[0]
        for n in frontier:
            if n['path_cost']+n['heuristic']<node['path_cost']+node['heuristic']:
                node=n
        frontier.pop(frontier.index(node))
        #add the pop node state to the explored list
        explored.append(node)
        for result in action(node['state'],size):
            child={"state":result,"path_cost":node['path_cost']+1,"depth":node['depth']+1,"parent_node":node,"heuristic":h(
            if goal_test(child['state']):
                goal_found=True
                print('goal found: '+str(child['state'])+'/'+
                      ' path_cost:'+str(child['path_cost'])+'/'+
                      ' depth:'+str(child['depth'])+'/'+
                      ' parent:'+str(child['parent_node']['state'])+'/'+
                      ' exploredlen:'+str(len(explored))+
                      ' fronlen:'+str(len(frontier))

                     )



            frontier.append(child)
```

```python
#A*(graph version)
def A_star(node,htype,size):
    #check if the initial node is the goal node
    if goal_test(node['state']):
        print(node)
    # a queue with node as the only element
    frontier=[node]
    explored=[]
    goal_found=False
    while goal_found==False:
    #update node and explored state
        #check if the frontier is empty
        if len(frontier)==0:
            print('failure')
        #pop the node with the lowest evaluation function from frontier for expansion
        node=frontier[0]
        for n in frontier:
            if n['path_cost']+n['heuristic']<node['path_cost']+node['heuristic']:
                node=n
        frontier.pop(frontier.index(node))
        #add the pop node state to the explored list
        explored.append(node)
        for result in action(node['state'],size):
            child={"state":result,"path_cost":node['path_cost']+1,"depth":node['depth']+1,"parent_node":node,"heuristic":h(
            if child['state'] not in [i['state'] for i in explored ] and child not in [i['state'] for i in frontier ]:
                #this need to be check the duplicate of the 'state' rather than the 'node',because the state can be duplica
                if goal_test(child['state']):
                    goal_found=True
                    print('goal found: '+str(child['state'])+'/'+
                          ' path_cost:'+str(child['path_cost'])+'/'+
                          ' depth:'+str(child['depth'])+'/'+
                          ' parent:'+str(child['parent_node']['state'])+'/'+
                          ' exploredlen:'+str(len(explored))+
                          ' fronlen:'+str(len(frontier))

                         )


                frontier.append(child)
```