Computer Assignment #1 – Ambulance and Patients

Ghazal Kalhor

810196675

kalhorghazal1378@gmail.com

Abstract **— In this computer assignment, we want to find a suitable solution for "Ambulance and Patients" problem; this is done by using the Uninformed Search and Informed Search algorithms such as BFS, IDS, and A* that we learnt in Artificial Intelligence.**

Keywords **— Uninformed Search, Informed Search, BFS, IDS, A*, Artificial Intelligence**

I. Introduction

The goal of this computer assignment to help our agent to do its job. Our agent is an ambulance that has to transfer the patients to the hospitals of the city in minimum possible time.

II. Downloading the Map of the City

We are given the map of the city in the form of a chess board. In some of the cells of the map, there are barriers that we cannot cross them. Each hospital a specified capacity to accept patients. In this part we read the given file and stored the position of the patients and hospitals in some lists. Also we defined the "Hospital" class that has a position and a capacity.

CODE 1
Downloading the Map of the City

```
char_map = []
file = open('test3.txt', 'r')
for x in file:
    x = list(x)
    if x[len(x)-1] == '\n':
        x = x [:len(x)-1]

    char_map.append(x)
file.close()

ambulance_x = 0
ambulance_y = 0
patients_x = list()
patients_y = list()
hospitals = list()
capacities = list()

for i in range(len(char_map)):
    for j in range(len(char_map[0])):
        if char_map[i][j] == AMBULANCE:
            ambulance_x = i
            ambulance_y = j
            char_map[i][j] = BLANK
        elif char_map[i][j] in Hospitals:
            capacity = ord(char_map[i][j]) - ord('0')
            h = Hospital(i, j)
            hospitals.append(h)
            capacities.append(capacity)
            char_map[i][j] = BLANK
        elif char_map[i][j] == PATIENT:
            patients_x.append(i)
            patients_y.append(j)
            char_map[i][j] = BLANK
```

CODE 2
Hospital Class

```
class Hospital:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def has_position(self, x, y):
        if self.x == x and self.y == y:
            return True
        return False
```

III. Modelling the Problem

In this problem we considered each state consisting of a lists of the positions of the patients, a set of hospital objects and the position of the ambulance. The goal node is be obtained, when the list of patients is empty. It means that the agent has done its job. Each action can be moving the object one unit to the left, right, top and bottom. We defined the "Node" class to for this goal.

CODE 3
Node Class

```
from Hospital import Hospital
import copy
Hospitals = ['0', '1', '2', '3']
PATIENT = 'P'
AMBULANCE = 'A'
WALL = '#'
BLANK = ' '
HOSPITAL = 'H'

LEFT = "Left"
RIGHT = "Right"
TOP = "Top"
BOTTOM = "Bottom"

class Node:
    def __init__(self, status_map, x, y, patients_x, patients_y, hospitals, capacities, parent=None, action=None):
        self.state = status_map
        self.parent = parent
        self.patients_x = patients_x
        self.patients_y = patients_y
        self.capacities = capacities
        self.hospitals = hospitals
        self.path_cost = 0
        self.action = action
        self.x = x
        self.y = y
```

CODE 4
MOVE TO THE LEFT FUNCTION

```
def add_left_child(self, children):
    i = self.x
    j = self.y

    move = False
    if j == 0:
        move = False
        return
    patients_x = self.patients_x[:]
    patients_y = self.patients_y[:]
    hospitals = self.hospitals
    capacities = self.capacities[:]
    new_state = self.state

    if new_state[i][j-1] == WALL:
        move = False

    elif new_state[i][j-1] == BLANK:
        is_h = False
        is_p = False
        for h in hospitals:
            if h.has_position(i, j-1):
                is_h = True
                break
        for r in range(len(patients_x)):
            if patients_x[r] == i and patients_y[r] == j-1:
                is_p = True
                break
        if is_p == False and is_h == False:
            move = True
        elif is_p == False and is_h == True:
            move = True
        elif is_p == True:
            if j == 1:
                move = False
            elif new_state[i][j-2] == WALL:
                move = False
            elif new_state[i][j-2] == BLANK:
                is_h2 = False
                is_p2 = False
                for h in hospitals:
                    if h.has_position(i, j-2):
                        is_h2 = True
                        break
                for r in range(len(patients_x)):
                    if patients_x[r] == i and patients_y[r] == j-2:
                        is_p2 = True
                        break
                if is_p2 == False and is_h2 == True:
                    capacity = 0
                    for r in range(len(hospitals)):
                        if hospitals[r].has_position(i, j-2):
                            capacity = capacities[r]
                            if capacity != 0:
                                capacities[r] -= 1
                            break

                    if capacity == 0:
                        for r in range(len(patients_x)):
                            if patients_x[r] == i and patients_y[r] == j-1:
                                patients_y[r] -= 1
                                break
                    else:
                        p = 0
                        for r in range(len(patients_x)):
                            if patients_x[r] == i and patients_y[r] == j-1:
                                p = r
                                break
                        patients_x.pop(p)
                        patients_y.pop(p)
                    move = True

                elif is_p2 == False and is_h2 == False:
                    for r in range(len(patients_x)):
                            if patients_x[r] == i and patients_y[r] == j-1:
                                patients_y[r]-=1
                                break
                    move = True
                else:
                    move = False

    if move == True:
        left_node = Node(new_state, i, j-1, patients_x, patients_y, hospitals, capacities, self, LEFT)
        children.append(left_node)
```

IV. BFS ALGORITHM

In this section, we used the BFS algorithm to find a suitable solution to the problem. For this goal, we defined the class BFS that has a frontier set that is implemented by FIFO queue to store the list of unexpanded nodes, an explored set that every time we expand a node, we add its state to this set, Also we stored number of visited states and number of unique visited states in this class that we update them during the execution of the algorithm. The algorithm works when we call the “play” function on its instance. At first we initialize the frontier set by the “initial_node”, then a loop continues until we did not reach a goal node or the frontier set is non-empty. At each step we enqueue a node from the frontier set. If it contains the goal state, we return the solution, else we expand it by adding its children to the frontier set.

CODE 5
BFS CLASS

```
import copy
from FIFO_queue import FIFO_queue
from Node import Node

SUCCESS = True
FAILURE = False

class BFS:
    def __init__(self, status_map, ambulance_x, ambulance_y, patients_x, patients_y, hospitals, capacities):
        self.number_of_unique_states = 0
        self.number_of_states = 0
        self.path_cost = 0
        self.inital_node = Node(status_map, ambulance_x, ambulance_y, patients_x, patients_y, hospitals, capacities)
        self.frontier = FIFO_queue()
        self.explored = set()
        self.unique_states = set()
```

CODE 6
FIFO QUEUE CLASS

```
class FIFO_queue:
    def __init__(self):
        self.dataset = []

    def enqueue(self, data):
        self.dataset.append(data)

    def dequeue(self):
        if self.is_empty() == True:
            return
        data = self.dataset[0]

        self.dataset.pop(0)

        return data

    def initialize(self, data):
        self.enqueue(data)

    def is_empty(self):
        if len(self.dataset) == 0:
            return True
        return False
```

CODE 7
PLAY FUNCTION IN BFS

```
def play(self):
    node = self.inital_node

    if node.contains_goal_state():
        self.path_cost = node.path_cost
        return SUCCESS

    self.frontier.initialize(node)

    while not self.frontier.is_empty():

        node = self.frontier.dequeue()

        self.number_of_states += 1
        if node.get_string() in self.explored:
            continue

        self.explored.add(node.get_string())

        if not node.get_string() in self.unique_states:
            self.unique_states.add(node.get_string())
            self.number_of_unique_states += 1

        if node.contains_goal_state():
            self.path_cost = node.path_cost
            return SUCCESS

        for child in node.get_children():
            if not child.get_string() in self.explored:

                if child.contains_goal_state():
                    self.path_cost= child.path_cost
                    return SUCCESS

                self.frontier.enqueue(child)

    return FAILURE
```

V. IDS ALGORITHM

In this section, we used the IDS algorithm to find a suitable solution to the problem. For this goal, we defined the class IDS that has a frontier set that is implemented by LIFO queue to store the list of unexpanded nodes, an explored set that every time we expand a node, we add its state to this set and a dictionary that has each state as its key and minimum depth that this state has been visited at. Also we stored number of visited states and number of unique visited states in this class that we update them during the execution of the algorithm. The algorithm works when we call the "play" function on its instance. This function calls the subroutine "dls" that executes a depth limited search with the specified depth. We used a recursive approach for implementing this function. At first we check that the given node contains the goal state or not, if yes we return a solution, else we call the function on its children if we did non visit its state or we visited it at the lower depth. The "play" function call its subroutine in a loop by incrementing the depth, until it reaches the success.

CODE 8
IDS CLASS

```
import copy
from LIFO_queue import LIFO_queue
from Node import Node

SUCCESS = True
FAILURE = False

class IDS:
    def __init__(self, status_map, ambulance_x, ambulance_y, patients_x, patients_y, hospitals, capacities):
        self.number_of_unique_states = 0
        self.number_of_states = 0
        self.path_cost = 0
        self.inital_node = Node(status_map, ambulance_x, ambulance_y, patients_x, patients_y, hospitals, capacities)
        self.frontier = LIFO_queue()
        self.explored = set()
        self.unique_states = set()
        self.goal_node = None
        self.min_depth_explored = {}
```

CODE 9
LIFO QUEUE CLASS

```
import copy

class LIFO_queue:
    def __init__(self):
        self.dataset = []

    def enqueue(self, data):
        self.dataset.append(data)

    def dequeue(self):
        if self.is_empty() == True:
            return
        data = self.dataset[len(self.dataset)-1]

        self.dataset.pop(len(self.dataset)-1)

        return data

    def initialize(self, data):
        self.enqueue(data)

    def is_empty(self):
        if len(self.dataset) == 0:
            return True
        return False
```

CODE 10
PLAY FUNCTION IN IDS

```
def dls(self, node, depth):
    self.number_of_states += 1

    if not node.get_string() in self.unique_states:
        self.unique_states.add(node.get_string())
        self.number_of_unique_states += 1

    if node.contains_goal_state():
        return SUCCESS

    if depth <= 0:
        return FAILURE

    self.min_depth_explored[node.get_string()] = depth

    for child in node.get_children():
        if (not child.get_string() in self.min_depth_explored) or self.min_depth_explored[child.get_string()] < depth-1:

            if self.dls(child, depth-1) == SUCCESS:
                return SUCCESS
    return FAILURE

def play(self):
    node = self.inital_node
    self.min_depth_explored = {}
    depth = 1
    while True:
        if self.dls(node, depth) == SUCCESS:
            self.path_cost = depth
            return SUCCESS
        depth +=1
    return FAILURE
```

VI. A* Algorithm

In this section, we used the A* algorithm to find a suitable solution to the problem. For this goal, we defined the class A* that has a frontier set that is implemented by min-heap queue that its criteria for choosing a node is the value of the evaluating function “f”. The argument of this function is a node and the return value of this function is sum of the path_cost of the node and the value of the heuristic function. We used two heuristic for implementing this function. Our first heuristic is to calculate sum of distances of each patients to nearest hospital. This heuristic is admissible and optimal. It is much more faster than the previous algorithms. Our second heuristic is to calculate the distance of the ambulance to the nearest patients. This heuristic is optimal. Also it has an explored set that every time we expand a node, we add its state to this set, Also we stored number of visited states and number of unique visited states in this class that we update them during the execution of the algorithm. The algorithm works when we call the “play” function on its instance. At first we initialize the frontier set by the “initial_node”, then a loop continues until we did not reach a goal node or the frontier set is non-empty. At each step we enqueue a node from the frontier set. If it contains the goal state, we return the solution, else we expand it by adding its children to the frontier set.

CODE 11
A* Class

```
import copy
import math
from HEAP_queue import HEAP_queue
from Node import Node

SUCCESS = True
FAILURE = False

class A_STAR:
    def __init__(self, status_map, ambulance_x, ambulance_y, patients_x, patients_y, hospitals, capacities):
        self.number_of_unique_states = 0
        self.number_of_states = 0
        self.path_cost = 0
        self.inital_node = Node(status_map, ambulance_x, ambulance_y, patients_x, patients_y, hospitals, capacities)
        self.frontier = HEAP_queue()
        self.explored = set()
        self.unique_states = set()
```

CODE 12
Heap Queue Class

```
import heapq

class HEAP_queue:
    def __init__(self):
        self.dataset = []
        self.top = 0

    def enqueue(self, data, criteria):
        heapq.heappush(self.dataset, (criteria, self.top, data))
        self.top += 1

    def dequeue(self):
        return heapq.heappop(self.dataset)[-1]

    def initialize(self, data, criteria):
        self.enqueue(data, criteria)

    def is_empty(self):
        if len(self.dataset) == 0:
            return True
        return False
```

CODE 13
Heuristic Functions

```
def f(self, node):
    return self.g(node) + self.h(node)

def g(self, node):
    return node.path_cost

def h(self, node):
    hospitals_x = node.get_state()[0]
    hospitals_y = node.get_state()[1]
    patients_x = node.get_state()[2]
    patients_y = node.get_state()[3]

    sum_distance = 0
    for i in range(len(patients_y)):
        min_distance = math.inf

        for j in range(len(hospitals_x)):
            distance = abs(patients_x[i]-hospitals_x[j]) + abs(patients_y[i]-hospitals_y[j])
            if distance < min_distance:
                min_distance = distance

        sum_distance += min_distance

    return sum_distance


def h2(self, node):
    patients_x = node.get_state()[2]
    patients_y = node.get_state()[3]

    distances = []
    sum_distance = 0
    for i in range(len(patients_y)):
        distance = abs(patients_x[i]-node.x) + abs(patients_y[i]-node.y)

        sum_distance += distance
        distances.append(distance)
    if len(distances) == 0:
        return 0
    return min(distances)
```

CODE 14
Play Function in IDS

```
def play(self):
    node = self.inital_node

    if node.contains_goal_state():
        self.path_cost= node.path_cost
        return SUCCESS

    self.frontier.initialize(node, self.f(node))

    while not self.frontier.is_empty():

        node = self.frontier.dequeue()

        self.number_of_states += 1
        if node.get_string() in self.explored:
            continue

        self.explored.add(node.get_string())

        if not node.get_string() in self.unique_states:
            self.unique_states.add(node.get_string())
            self.number_of_unique_states += 1

        if node.contains_goal_state():
            self.path_cost= node.path_cost
            return SUCCESS

        for child in node.get_children():
            if not child.get_string() in self.explored:

                self.frontier.enqueue(child, self.f(child))

    return FAILURE
```

VII. Comparison between BFS and IDS

When the solution is in higher depth we should choose IDS. But if time and memory is very critical, it is better to use BFS.

VIII. Comparison between BFS and A*

We know that the complexity of FIFO queue is better than the min-heap queue. So if we have an admissible heuristic, we should choose A*. Also the time complexity of A* is better than BFS. And it is faster when we have a higher branching factor. But if our heuristic is not good and optimality and memory is important for us, we should choose BFS.

IX. Comparison between A* and IDS

We know that uninformed search algorithms are slower than informed search algorithms. So if we have an admissible heuristic, we should choose A*. But if our heuristic is not good and optimality and memory is important for us, we should choose IDS.

X. Results

The result of execution of these search algorithms on three given test cases are shown in Table 1, 2, 3.

Table 1
Result of test case 1

Algorithm	Path Cost	Number of Visited States	Number of Unique Visited States	Execution Time
BFS	11	386	309	0.008949513 s
IDS	11	1529	415	0.027928823 s
A* (Heuristic #1)	11	190	153	0.00721327 s
A* (Heuristic #2)	11	320	264	0.011141373 s

Table 2
Result of test case 2

Algorithm	Path Cost	Number of Visited States	Number of Unique Visited States	Execution Time
BFS	27	16701	13216	0.440853193 s
IDS	27	110242	14215	2.77018189 s
A* (Heuristic #1)	27	7127	5689	0.273404116 s
A* (Heuristic #2)	27	16426	13051	0.58872151 s

TABLE 3
RESULT OF TEST CASE 3

Algorithm	Path Cost	Number of Visited States	Number of Unique Visited States	Execution Time
BFS	39	38034	28581	1.007239416 s
IDS	39	260267	31522	6.4436473 s
A* (Heuristic #1)	39	14821	11325	0.52252515 s
A* (Heuristic #2)	39	34064	26135	1.21768482 s

XI. CONCLUSIONS

In this computer assignment we learned that informed search algorithms are much more faster than uninformed algorithms. Also we were introduced one of the applications of Artificial Intelligence in real life.

REFERENCES

[1] COMPUTER ASSIGNMENT MANUAL, Computer Assignment 1, *Search*