

Artificial Intelligence–Fall 2022
Project 01–Search Problems
Ehsan Hosseini–Mohammad Afshari

Mohammad Afshari

Overview

The company that manufactures dining tables has made large tables and the customers are dissatisfied with the size of the tables because it is difficult for them to access the butters on the table.

After consulting with a computer scientist, the company comes to the conclusion that it needs to build robots that deliver the globes to customers.

There exists a robot for carrying butters and delivering them to requesters , but unfortunately, the search segment of the robot does not work correctly. So our duty is to implement multiple search algorithms and develop the search segment of this problem-solving agent.

The objective is to use the optimal search algorithm to find routes in which the robot will deliver the butters to the requesters with the least battery consumption.

Some features and constraints are established on the robot, which are as follows:

1. Robot can only push the butter and cant pull or carry it with itself.
2. Robot can only push one butter at a time and two in a row.
3. Robot is only capable of moving in 4 main directions(up,down,right,left) .
4. The environment is fully observable.
5. There some blocks that cannot be passed.
6. Robot should care not to drop any butter or itself from the table.

7. After delivering butter to the customer, robot cannot move the butter.

Mohammad Afshari

Phase one

Considering that our problem is a search problem, so the first step is to formulate the problem.

In addition In this phase, we will model the problem and implement the functions and general structure of the program that is supposed to be used in the search algorithms.

Problem formulation:

- **Initial state:** Any state can be designated as the initial state
- **State Space:** A state of the world says which objects are in which cells. So the The state space contains all possible states in which we can place the robot, butters and targets on the table.
- **Actions:** In the Butter-Robot world we defined four actions: move Left, move Right, move Up and move Down.
- **Successor:** Right moves the agent ahead one cell in the direction it is facing, Left moves the agent ahead one cell in the direction it is facing, Down moves the agent ahead one cell in the direction it is facing and Up moves the agent ahead one cell in the direction it is facing unless it violates the mentioned constraints on overview section.
- **Goal states:** The states in which every cell is clean.

Problem modeling:

In this part we will explain about the modules and packages which is used in implementation.

- **Input**

This module is used to get inputs from the user.

This module allows the user to enter the input matrix manually in the terminal or read it from the file.

```
def read_input(read_type:int=1) -> list:
    # Read from file
    if read_type == 1:
        with open("test/input2.txt", 'r') as file:
            input_matrix = file.readlines()

            # Extract number of rows and Columns from file
            n_rows = int(input_matrix[0].strip("\n")[0])
            n_columns = int(input_matrix[0].strip("\n")[2])
            del input_matrix[0]

            # Extract gameplay matrix from file
            matrix = [line.strip('\n').split() for line in input_matrix]

        return matrix

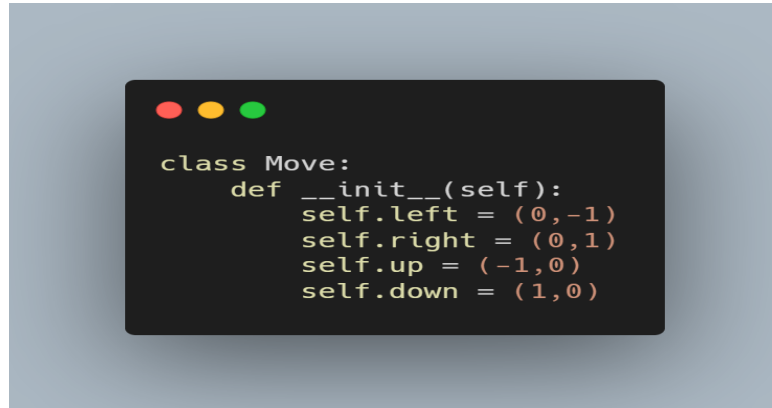
    # Read from command line
    elif read_type == 2:
        while True:
            print("Enter matrix Below:\n")
            input_matrix = input()
            try:
                n_rows = int(input_matrix[0])
                n_columns = int(input_matrix[2:4])
                input_matrix = input_matrix[5:]

                matrix = []
                input_matrix = input_matrix.split()
                for i in range(n_rows):
                    row = []
                    for j in range(n_columns):
                        row.append(input_matrix[j])
                        input_matrix = input_matrix[n_columns:]
                    matrix.append(row)

                return matrix
            except:
                print("Wrong input for matrix!")
    else:
        return []
```

- **Move**

This module contains a Move class that specifies the directions of the robot which are constant.



- **Node**

This module contains a Node class. Search algorithms require a data structure to keep track of the search tree. A node in the tree is represented by a data structure with following attributes:

- **State:** the state to which the node corresponds;
- **Parent:** the node in the tree that generated this node;
- **Action:** the action that was applied to the parent's state to generate this node;
- **Action_path:** a string that contains the actions that robot has been done from initial state to state at node n. ex: 'UUDR'
- **Depth:** it indicates number of movements
- **H_n:** estimated cost of the cheapest path from the state at node n to a goal state using manhattan distance.
- **Path_cost:** path cost from the initial state to node n

```

from Utils import manhatan_distance
class Node:
    def __init__(self, state, parent=None, action=None, path='', depth=0, h_n=None,
cost=None):
        self.state = state
        self.parent = parent
        self.action = action
        self.action_path = path
        self.depth = depth
        if h_n == None:
            self.h_n = self.find_h_n_for_initial_State(self.state)

        else:
            self.h_n = h_n

        if cost == None:
            self.path_cost = self.find_initial_cost(self.state)

        else:
            self.path_cost = cost

```

It also contains two utility functions that are used to set `h_n` and `path_cost` for initial state.

```

def find_initial_cost(self, state:list):
    for row in range(len(self.state)):
        for column in range(len(self.state[0])):
            if 'r' in self.state[row][column]:
                r_place = (row,column)

    r_grid = self.state[r_place[0]][r_place[1]]
    cost = 0
    for char in r_grid:
        if char.isdigit():
            cost = int(char)
            break
    return cost

```

```

def find_h_n_for_initial_State(self, state:list):
    b_places = set()
    # find butters places
    for row in range(len(state)):
        for column in range(len(state[0])):
            if 'b' in state[row][column]:
                b_places.add((row,column))

    p_places = set()
    # find goal places
    for row in range(len(state)):
        for column in range(len(state[0])):
            if 'p' in state[row][column]:
                p_places.add((row,column))
    b_places = list(b_places)
    p_places = list(p_places)
    return sum([manhatan_distance(b_places[index], p_places[index]) for index in
range(len(b_places))])

```

- **Queue**

We need a data structure to store the frontier(The frontier is the set of unexpanded nodes).The appropriate choice is a queue of some kind in some algorithms, because the operations on a frontier are:

- **Is_empty:** which check that if queue is empty or not
- **Enqueue:** enqueue an element to queue
- **Dequeue:** dequeue an element from queue
- **Front:** get the element which is on top of queue
- **Some priority enqueues (ucs_enqueue, bfs_enqueues, a_star_enqueues):** these are enqueue methods with one difference, These methods put each element in the right place according to their special evaluation condition so that the queue is sorted

```
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, node:Node):
        self.queue.append(node)

    def dequeue(self):
        try:
            return self.queue.pop(0)
        except:
            return

    def front(self):
        try:
            return self.queue[0]
        except:
            return

    def is_empty(self):
        if len(self.queue) == 0:
            return True
        else:
            return False

    def ucs_enqueue(self, node:Node):
        for i in range(len(self.queue)):
            if node.path_cost < self.queue[i].path_cost:
                self.queue.insert(i, node)
                return
        self.queue.append(node)

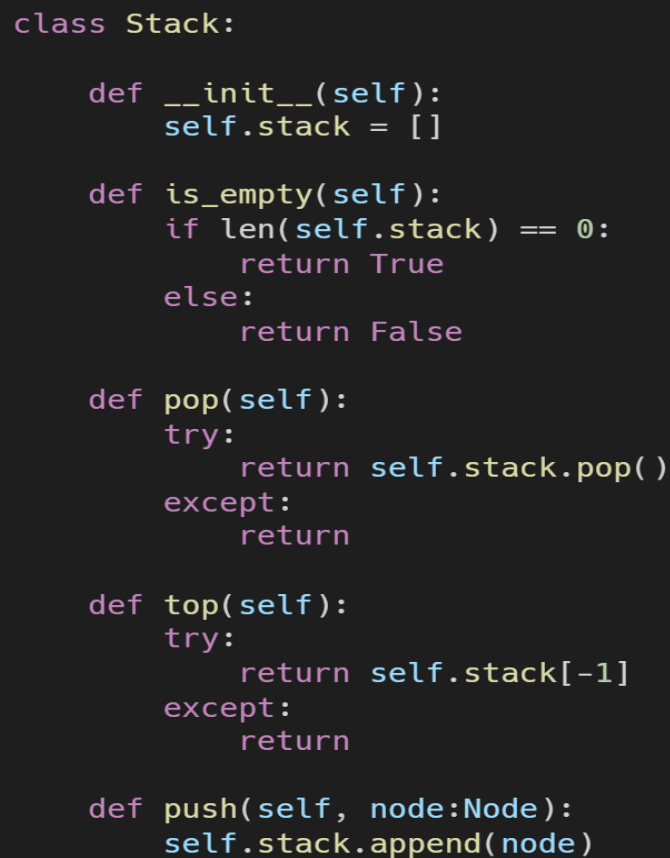
    def bfs_enqueue(self, node:Node):
        for i in range(len(self.queue)):
            if node.h_n < self.queue[i].h_n:
                self.queue.insert(i, node)
                return
        self.queue.append(node)

    def a_star_enqueue(self, node:Node):
        for i in range(len(self.queue)):
            if (node.h_n + node.path_cost) < (self.queue[i].h_n + self.queue[i].path_cost):
                self.queue.insert(i, node)
                return
        self.queue.append(node)
```

- **Stack**

We need a data structure to store the frontier(The frontier is the set of unexpanded nodes).The appropriate choice is a stack in some algorithms, because the operations on a frontier are:

- **Is_empty**: which check that if stack is empty or not
- **push**: push an element to top of the stack
- **pop**: pop an element from the top of the stack
- **top** : get the element which is on top of the stack

A screenshot of a code editor with a dark background and light-colored text. The code defines a Python class named 'Stack'. It includes methods for initialization, checking if the stack is empty, popping an element, getting the top element, and pushing a new element. The code is as follows:

```
class Stack:

    def __init__(self):
        self.stack = []

    def is_empty(self):
        if len(self.stack) == 0:
            return True
        else:
            return False

    def pop(self):
        try:
            return self.stack.pop()
        except:
            return

    def top(self):
        try:
            return self.stack[-1]
        except:
            return

    def push(self, node:Node):
        self.stack.append(node)
```

- **Utils**

This module contains some utility functions which are used in different parts of the program as needed

- **Expand:** this function is used in all algorithms to expand the childs and create nodes of the tree. It returns a generator of child nodes so we can iterate on childs
- **Is_cycle:** this function is used in IDS algorithm to backtrack the tree to parent nodes and look for cycle
- **manhatan_distance:** this function is the implementation of Manhattan distance algorithm which gets the index of one butter and one goal and applies the algorithm on it.
- **Heuristic:** this function uses Manhattan distance to calculate h_n for each node and is used to generate h_n attributes for class Node.

- **Main**

The flow of the program starts from here and all modules will be used here

- **Butter_Robot**

This module contains the ButterRobot class which could be said is the main part of the program in which the main logic of the program is implemented.

It just has one attribute which is initial_state and it corresponds to the given matrix by the user.

- **actions:** this method gets state as input and returns all possible directions that robot can g
- **Successor:** this method gets state and direction as input and moves the robot and mabe butter based on that
- **action_cost:** this method will return how much the action that robot does is cost.

- **is_goal:** this method gets a set of butter places and a set of target places and check if they are equal or not
- **get_direction:** this method gets a tuple which indicates direction of move and then returns a corresponding string to that. ex: (0,1) -> 'R'
- **find_robot_place:** this method gets state as an input and return robot place in tuple form
- **find_butter_places:** this method gets state as an input and return a set of butters places in tuple form
- **find_target_places:** this method gets state as an input and return a set of target places in tuple form
- **is_obstacle:** this method check if there is an obstacle in destination or not.
- **is_out_of_bound:** check that if the movement cause the robot to throw itself down from the table or not.
- **Is_butter:** this method check if there is an butter in destination or not.
- **is_butter_movable:** this method checks if the butter is movable or not, based on conditions on overview.

Phase Two

A search algorithm takes a search problem as input and returns a solution, or an indication of Search algorithm failure. this is the reason that we implemented ButterRobot class as a problem.

In this phase we will implement some uninformed algorithms which we can see below. Informed algorithms will be discussed in next phase.

Mohammad Afshari

Bfs

Like other algorithms the bfs gets a butter robot object as an input

then it creates the root node from the initial_state

It checks that is that state a goal state or not. If yes return it and the function will be finish else put that node to a queue and mark the state as reached state while the queue is not empty the algorithm will run.

In each loop iteration a node will be pop from the queue and we pass it to expand function.

expand yield the childs of that node.

Then for each child we check that if it is goal or not. If yes return it and the function will be finish else mark child as visited and add it to queue

If none of the queue elements are target nodes, the function returns the value none, which means that the answer is not found.

```
def breadth_first_search(problem:ButterRobot):
    node = Node(state=problem.initial_state)

    if problem.is_goal(node.state):
        return node

    frontier = Queue()
    frontier.enqueue(node)
    reached = [problem.initial_state]

    while not frontier.is_empty():
        node = frontier.dequeue()

        for child in expand(problem, node):
            state = child.state

            if problem.is_goal(state):
                return child

            if state not in reached:
                reached.append(state)
                frontier.enqueue(child)

    return
```

Time complexity:

Suppose searching a uniform tree where every state has b successors. The root of the search tree generates b nodes, each of which generates b more nodes, for a total of b^2 at the second level. Each of these generates b more nodes, yielding b^3 nodes at the third level, and so on. Now suppose that the solution is at depth d . Then the total number of nodes generated is $1 + b + b^2 + b^3 + \dots + b^d = O(b^d)$

Space complexity:

All the nodes remain in memory, so both time and space complexity are $O(b^d)$.

Mohammad Afshari

DFS

Its as same as bfs and the only difference is that we use stack instead of queue because we traverse tree in depth and not breadth

```
def depth_first_search(problem: ButterRobot):
    node = Node(state=problem.initial_state)

    if problem.is_goal(node.state):
        return node

    frontier = Stack()
    frontier.push(node)
    reached = [problem.initial_state]

    while not frontier.is_empty():
        node = frontier.pop()

        for child in expand(problem, node):
            state = child.state

            if problem.is_goal(state):
                return child

            if state not in reached:
                reached.append(state)
                frontier.push(child)

    return
```

Time complexity:

The time complexity is as same as bfs.

Space complexity:

It has memory complexity of $O(bm)$, where b is the branching factor and m is the maximum depth of the tree.

ehosseini8001@gmail.com

IDS

IDS algorithm is a combination of BFS(breadth-first search) and DFS(depth-first search),we take the space-efficiency of DFS and the time-efficient behavior of BFS.

How it works:in the IDS algorithm instead of running DFS on the whole tree we only run it till it reaches a certain depth in the tree which increases each time. We know that space and time have always been a trade off, meaning that if we try to minimize the space required ,time complexity will change and it will be a higher order.in Ids algorithm we may visit certain nodes multiple times(worth knowing that the bottom nodes are visited less

Time complexity:

just like DFS ($O(v+e)$)

Space complexity:

$O(bm)$, b being the branching factor and m being the maximum depth

ehosseini8001@gmail.com

UCS

UCS is very much like BFS(best-first search),in each time we try to visit a node we choose the least costliest among all the nodes in fringe.Every time a node is visited all the adjacent nodes are checked compared by their costs(g_n) and the least is chosen.

Time complexity:

its exponential with respect to number of children at each node

Space complexity:

exponential having to hold all the nodes (for exponential growth of nodes)

ehosseini8001@gmail.com

Phase three

In this phase informed algorithms are implemented and added to the project including:

BFS(best-first-search)

Best-first search(greedy) algorithm mainly runs on h_n or heuristic function for every node and chooses the least, being 100% dependent on the heuristic function we have. In this project the heuristic function is Manhattan-distance which gives us a fair result.

Time complexity:

$O(b^m)$, 'b' being the branching factor or the number of children node at each node

Space complexity:

it's like DFS being $O(bm)$.

A_Star

A_star algorithm is combination of BFS(best-first search) and UCS comparing both h_n and g_n for nodes and then visit the best offered in the fringe

Time complexity:

time complexity of this algorithm is very dependent on the heuristic offered ranging from $O(n)$ having an optimal heuristic to normal BFS time complexity($O(v)$)

Space complexity:

space complexity is exponential (having to hold exponential number of node)