Solving Problems By Searching

Yashovardhan Srivastava

October 27, 2021

Introduction

In this discussion,we will solve problems by classical-searching.Think of searching as a way of brute-forcing our way through the problem.We will start with this brute force approach and work our way upto better searching strategies.This chapter,for the most part deals with problems with finite number of cases which could be mapped out using some of the known data structures, with some intuition for cases that are not covered. Problems then are solved using search algorithms to find a solution(s).

Searching Process and Parameters

Problem Solving through searching for an agent to solve involves discretizing the problem to state clearly what the problem is,what are the different **States** an agent could be in,what are the **Actions** allowed,what is the **Goal** and what is the **Path Cost** to them ?. It is very important to know what these parameters mean in order to solve the problem appropriately:

States: States represent the different situations an agent encounters.For a problem, States should be well-defined with the goal state being one of them.The agent's task is to consider how to act in the current state in order to reach the goal.All of the different states an agent could be in makes up the State-Space .Example: For an agent solving a Rubik's Cube, all of the different permutations of the cube consists of the State-Space. The agent's current state is one of the states in the States-Space.

Actions: Actions are the steps that an agent takes to move from one state to another to move towads a goal.Actions should be general rather than specific so as to reduce number of steps to the solution. Continuing our Rubik's Cube example,if our action are like: *'Turn side by 90 degrees'* or *'Rotate Cube by 180 degrees'* then the majority of our computation is wasted in being precise when we could have used the same time to find solutions.Level of detail increases the number of steps in the solution.

Goal: Goal refers to the solution of the problem.Goal should be clearly defined and reachable from states to avoid wrong solutions or to avoid possible dead ends(we will encounter this issue later in thr chapter). In our Rubik's Cube example,it refers to the final state that is the cube being solved(all six faces have matching colors on all six sides).

Path Cost: Path Cost refers to the *'cost'* to reach the goal.Here *Path* refers to sequence between two states connected via actions and *Cost* is the numeric value given to a path-also called step-costs.We assume that step-costs are non-negative(we will see later why). Strategies that have a lower Path Cost are more optimal than their counterparts.

A problem can be defined formally by the following components,namely: Initial State,Actions,Transition Model,State and State-Space,Goal Test,Path-Cost.

This would be our basic strategy to tackle problem solving by searching.We would start with defining precisely States and Actions keeping in mind the limitations of the problem.Goal(s) should me mentioned clearly and should be rechable from the States. The solution to the problem is found out by searching through the State -Space,moving between States via Actions which limits the objectives that an agent has to follow to reach the Goal(s).Path to goal are subjected to lower Path-Costs.Implementing these strategies would ease the process of implementing them in computer programs.They basically provides the *pseudo-code* for the program.

Searching Strategies

Having formed some problems,we need some strategies to reach to a solution. A solution is basically a sequence of actions to reach to the goal state(s). A common way to approach this problem is to make use of data-structure *Tree to form* a search tree for the problem.The search-tree consists of a *root* node, representing the **Initial State** of the problem . Other states (representing the *branch* node of the search tree) could be reached from the initial state via **Actions**.The Goal(solution) represents the *leaf* node. The **Path Cost** determines the success of our search algorithm.

Implementing Search Tree

Implementation of Search Tree is in *Python*,but the idea is same for any language. Our strategy to implement the Search-Tree would be same as we described above: Start with a root node,go to different branch nodes via actions to ultimately reach the Goal (leaf node).Nodes have certain properties associated with them such as state,parent,actions,path cost,depth etc. Some tasks associated with our problems is calculating path costs,extending the tree,generating child of the nodes and keeping track of them. Our necessity for such functionality hints us that implementing a `class Node` is the right way to proceed. What do we need for `class Node`?

- `__init__`: **Inputs-**`state,parent node,action,path_cost`.
  Class constructor to define Node's properties.It consists of state,parent,actions,path cost and depth.
- `path`: **Inputs-**`self`.
  Return a list of nodes from initial to the current state(top-down list).
- `solution`:**Inputs-**`self`.
  A sequence of actions leading to the current state from the initial state.

- `child_node`:**Inputs-**`self,problem,action`
  Returns the next node in the tree.It takes input a `problem` whose result from our `action` gives the next state.We then use this next state to create a child of our current node.

- `expand`:**Inputs-**`self,problem,action`
  Returns the list of child nodes possible from our current state for a problem.

- `__repr__,__eq__` :**Inputs-**`self`
  Magic methods to define class object's behaviour.

Python Code:

```
class Node:
    #class constructor consists of properties of the Node
    def __init__(self,state,parent=None,action=None,path_cost=0)->None:
        self.state=state
        self.parent=parent
        self.action=action
        self.path_cost=path_cost
        self.depth=0
        if parent:
            self.depth=parent.depth +1
    def __repr__(self) -> str:
            return "<Node {}>".format(self.state)
    def __eq__(self, other) -> bool:
        return isinstance(other,Node) and self.state==other.state
    #Path List from root to current state
    def path(self)->list:
        node,path_back=self,[]
        while node: #Till root node is not reached
            path_back.appen(node)
            node=node.parent
        return list(reversed(path_back))
    #Sequence of actions from root to current state
    def solution(self)->list:
        return [node.action for node in self.path()[1:]]
    #Generating child node of the state
    def child_node(self,problem,action):
        next_state=problem.result(self.state,action)
        next_node=Node(next_state,
        self,
        action,
        problem.path_cost(self.path_cost,self.state,action,next_state))
        return next_node
    #List of possible child nodes from current state.
    def expand(self,problem)->list:
        return [self.child_node(problem,action)
        for action in problem.actions(self.state)]
```

Searching for Solutions

Having formed basic structure and implementation of search tree ,we turn our focus towards Searching Algorithms.These are algorithms that move through the search tree to find solution to a problem.In this chapter,we will see Classic Searching-later we will tackle other searching strategies.Searching algorithm requires a data structure to keep track of the search tree of the problem,which should be stored in a container called **frontier** such that the next node for expansion is easily available.It is to be noted that searching strategies that work for one problem may not work well for other problems.
Problems determine searching techniques,not the other way around.

Classic Search(or Search),and therefore Searching algorithms are broadly divided into two categories:**Uninformed** and **Informed**(Heuristic) Searches:
Uninformed Searching involves finding solutions to the problem when the only information provided are problem statement and the goal state.Think of it as checking each state in the search tree and check whether it is a goal state or not.Uninformed strategies work well for many problems.We now look at some uninformed searching strategies:

Breadth First Search

Breadth First Search is a simple strategy in which all of the successors of a node at a particular level (or depth) are expanded first before any nodes at next level to check whether goal state is reached or not.The algorithm stops when a solution is encounter at the shallowest level.The data structure used to store nodes at the frontier is a **FIFO Queue** (First in First Out is the general way in which our algorithm work- so it makes sense to use FIFO Queue for the frontier).Breadth First Search is useful for problems where path cost is prioritized more than other factors. Python code for Breadth First Search is given below.

```
def breadth_first_graph_search(problem):
    node = Node(problem.initial)
    if problem.goal_test(node.state):
        return node
    frontier = deque([node])
    explored = set()
    while frontier:
        node = frontier.popleft()
        explored.add(node.state)
        for child in node.expand(problem):
            if child.state not in explored and child not in frontier:
                if problem.goal_test(child.state):
                    return child
                frontier.append(child)
    return None
```

Having discovered the algorithm,let us evaluate how it performs:
Time Complexity: A uniform search tree that generates b nodes at each level upto a maximum depth of d,the time complexity is given by:

$$b + b^2 + b^3 + b^3..... = O(b^d)$$

Space Complexity: Space complexity for a uniform search tree,which keeps track of all the nodes in memory is $O(b^d)$,which is exponential in nature.
Exponential Complexity indicates that BFS is not suitable for problems that have greater depth.