COS30019 - Assignment – 1 Report

Submitted by:

Name – Pulkit Pannu

Student ID – 104093910

Tutorial – Tuesday, 4:30 – 6:30 pm

Instructions

Using the program is straight forward. From the command line in the current directory, given that all the files are in the same directory including the test cases, run the command:

>python main.py <filename> <algorithm>

The filename has to be with .txt extension.

The algorithm that you can type is in this format: bfs, dfs, gbfs, astar, iddfs, idastr.

After that you can hit play on the GUI and run the search. Futhermore, the GUI allows you to choose the algorithm manually.

Introduction

The Robot Navigation problem is a simple situation where we define a grid or a maze through which a robot must traverse to find a path to the goal. The grid has a finite dimension and consists of a starting point for the robot, one or multiple end points (i.e. goals), empty squares through which the robot can pass and walls through which the robot cannot pass.

To solve this problem, we are to create various search algorithms which directs the robot from the start position to the goal position. Some basic concepts that are used to solve search problems like this are: Tree search and Graph search.

The basic difference between tree and graph search is that in graph search we us a closed list to store and keep track of the nodes that have already been visited and expanded, whereas in tree search we do not keep a closed list and hence, the same node can be visited multiple times, possibly resulting in infinite loops [1]. In my implementation of the algorithms, I have used graph search rather than tree search for the above reason.

Glossary

Some glossary terms that I have used throughout the report:

- Breadth First Graph Search as **BFS**
- Depth First Graph Search as **DFS**
- Greedy Best First Search as **GBFS**

- A* search as **A***
- Iterative Deepening Depth First Search as **IDDFS**
- Iterative Deepening A* as **IDA***

Search Algorithms

From the Glossary you can see that I have implemented six search algorithms which are a mix of informed and uninformed search algorithms. All the search algorithms are basically different ways of choosing which nodes to expand first and in which order to find the most optimal solution with the least amount of overhead.

Uniformed Search Algorithms

Also known as Blind Search, uninformed search algorithms are strategies which only have the provided problem definition as the information available to carry out the search on their own. All they can do is generate successors and distinguish a goal state from a non-goal state [2].

1. *Breadth First Graph Search*

   BFS is a simple and optimal search strategy. It works on the principle of FIFO, i.e., the nodes are expanded in the order they are explored. Meaning, that the nodes that are added to the frontier first are explored first and checks each node before exploring whether or not it is a goal node.

   BFS will find a path to the solution, given that one exists, if the *b* of the solution is finite. Hence, it is complete. Moreover, it will find the most optimal solution if all the step costs are identical [3].

2. *Depth First Graph Search*

   DFS uses the LIFO rule such that it expands the deepest node in the current frontier first and so on. If we use tree search instead of graph search, we are most likely to get stuck in infinite loops and state spaces as it does not checks for repeated states and redundant paths. Therefore, it is better to use graph search.

   DFS is not an optimal or complete search strategy as it will return the goal state that it reaches first at a certain depth, irrespective of the fact that there is another goal state available at a much shorter depth. And since it can get stuck in infinite loops, it might return that no goal is reachable when it is otherwise.

3. *Iterative Deepening Depth First Search*

   As the name suggests, IDDFS uses DFS in its implementation. It is an optimised version of the Depth Limited search strategy which basically uses DFS but stops

the search down one branch after it reaches a fixed depth limit. IDDFS uses the same concept but it gradually increases the depth limit, i.e. from 0 then 1 and 2 and so on, till the goal is reached. This will occur when the depth limit reaches d, the depth of the shallowest node [4].

IDDFS is a good search strategy as it gets the best of both BFS and DFS. It has the memory requirements of DFS (O(*bd*)) and it is complete and optimal like BFS when *b* of the solution is finite and all the step costs are identical respectively.

Informed Search Algorithms

Also known as Heuristic search strategies, informed search algorithms uses "problem-specific knowledge beyond the definition of the problem itself" [5] and therefore, can find more efficient solutions than uninformed searches. This specific knowledge is known as a heuristic function, f(n).

4. *Greedy Best First Search*

   GBFS uses an admissible heuristic, according to which it tries to expand the node which is closest to the goal thinking that it will lead to the solution quickly. In my implementation of the GBFS, it's using the Manhattan distance of two points as the heuristic, i.e., f(n) = h(n) where h(n) = Manhattan distance.

   GBFS is suboptimal because it might go down a path which at first might seem to be the "cheapest" option but in the end provides a path that is longer than the shortest path. Moreover, GBFS is also incomplete even in finite state space [6].

5. *A* Search*

   A* is a modification of the Uniform-Cost search or Dijkstra's Shortest Path and is a widely known form of the best-first search.

   It improves on the heuristic function of Dijkstra's algorithm which expanded the node n with the lowest path-cost g(n). The problem with this approach is that the if, for instance, for some reason the path-cost of nodes leading away from the goal decrease slightly than the nodes on the correct path, the search algorithm would traverse down the wrong path for some time before the path-cost of a node becomes greater than the node on the correct path.

   A* uses the sum of g(n), the cost to reach a node from the root, and h(n), the cost to get from the node to the goal state:

   f(n) = g(n) + h(n)

   Therefore, if the algorithm traverses along the wrong path, the heuristic function would start to increase as h(n) for these nodes would be greater than the ones on the correct path and eventually come back to the correct path.

The function h(n) plays a role of great importance in this search strategy. To ensure that our search algorithm is optimal, our heuristic h(n) needs to be admissible and consistent, i.e., h(n) should never overestimate the cost to reach the goal and h(n) should satisfy the general triangle inequality where the three vertices of the triangle are the root node, any node on the path to the goal and the goal node and the length of the sides of the triangle are the path cost from one node to the other.

A*s main drawback is that it runs out of memory as it keeps all the nodes stored so that its easier for it to backtrack it. Hence, A* is not practical for large scale problems.

6. *Iterative Deepening A* Search*

   IDA* reduces the memory requirements of A* search as it uses the f-cost(g+h) as the limit in iterative deepening search rather than the depth. IDA* is practical for problems with unit step-costs, just like the robot navigation problem, but it suffers some difficulties in problems with real-valued costs.

Implementation

In my implementation of the Robot navigation problem, I followed the general procedure provided to us in the week 3 and 4 labs. First, I made an abstract Problem class and my RobotNav class which is derived from the problem class. The RobotNav class takes in a filename in the constructor and creates the grid and initializes all the variables such as start position, goal position, obstacles, actions, etc.

I also have a Node class which is a representation of each square or state that the robot can be in as a Node. A Node has the following attributes:

- State – Its position or co-ordinate.
- Parent – The previous Node from which it got expanded.
- Action – The action performed on the Node.
- Path cost – Step cost of the node.
- Depth – The depth of the Node.

Node class has a function named **expand** which takes in a problem as an argument and returns a list of nodes that can be reached from that node by all the applicable actions available to it. It uses the function named **child node** which provides the node state of each action.

Then I made a file where I defined all the algorithms. It had different functions for different algorithms.

BFS - To implement BFS I made a simple frontier of the data type “deque” initialized with the starting position of the robot. Deque allowed me to pop the first item (“popleft”) from the frontier and add it to the explored list as we check whether it’s a goal state. If it is, then the algorithm returns the node and the explored list otherwise it would expand the node and add its children to the frontier if they are not already in the frontier or the explored list and then repeat again till the frontier is empty.

```
def breadth_first_search(problem):
  frontier = deque([Node(problem.initial)] )
  explored = []
  while frontier:
    node = frontier.popleft()
    explored.append(node.state)

    if problem.goal_test(node.state):
      return node, explored
    for child in node.expand(problem):
      if child.state not in explored and child not in frontier:
        frontier.append(child)
  return None, explored
```

DFS – Implementing DFS is the same as BFS except it explores the nodes in LIFO format. Instead of exploring the first node from the frontier, pops the last on the was added and explores it first.

```
def depth_first_graph_search(problem):
  frontier = [Node(problem.initial)]

  explored = []

  while frontier:
    node = frontier.pop()
    explored.append(node.state)
    if problem.goal_test(node.state):
      return node, explored
    for child in node.expand(problem):
      if child.state not in explored and child not in frontier:
        frontier.append(child)
  return None, explored
```

GBFS – To implement GBFS, the frontier is a data type of Priority Queue which basically prioritizes the element popping with either least or the maximum value of the heuristic function provided to the queue to be applied on each node. So, the heuristic used for GBFS is the Manhattan distance and the frontier is prioritized according to the node with the least heuristic value. My implementation uses the generalized version, i.e., it uses the best first graph search and then provides the heuristic for GBFS to be implemented.

```
# Best First Graph Search Algorithm
def best_first_graph_search(problem, f, display=False):
  f = memoize(f, 'f')
  node = Node(problem.initial)
  frontier = PriorityQueue('min', f)
  frontier.append(node)
  explored = []
  while frontier:
    node = frontier.pop()
    if problem.goal_test(node.state):
        if display:
            print(len(explored), "paths have been expanded and", len(frontier), "paths remain in the frontier")
        return node, explored
    explored.append(node.state)
    for child in node.expand(problem):
        if child.state not in explored and child not in frontier:
            frontier.append(child)
        elif child in frontier:
            if f(child) < frontier[child]:
                del frontier[child]
                frontier.append(child)
  return None, explored

# Greedy Best First Graph Search Algorithm
def greedy_best_first_graph_search(problem, h=None, display=False):
    h = memoize(h or problem.h, 'h')
    return best_first_graph_search(problem, h, display)
```

A* search – Similarly for A*, I have used the best first graph search algorithm and provided it the heuristic to be implemented for A*. In this case the heuristic is the sum of the Manhattan distance and the path cost of the node.

```
# Best First Graph Search Algorithm
def best_first_graph_search(problem, f, display=False):
  f = memoize(f, 'f')
  node = Node(problem.initial)
  frontier = PriorityQueue('min', f)
  frontier.append(node)
  explored = []
  while frontier:
    node = frontier.pop()
    if problem.goal_test(node.state):
        if display:
            print(len(explored), "paths have been expanded and", len(frontier), "paths remain in the frontier")
        return node, explored
    explored.append(node.state)
    for child in node.expand(problem):
        if child.state not in explored and child not in frontier:
            frontier.append(child)
        elif child in frontier:
            if f(child) < frontier[child]:
                del frontier[child]
                frontier.append(child)
  return None, explored

# A* Search Algorithm
def astar_search(problem, h=None, display=False):
    h = memoize(h or problem.h, 'h')
    return best_first_graph_search(problem, lambda n: n.path_cost + h(n), display)
```

IDDFS – IDDFS uses the depth limited search where it recursively performs DFS for the provided limit and increments the limit whenever the cutoff is reached, or the goal node isn't found.

```
def recursive_dls(node, problem, limit, explored):
  if problem.goal_test(node.state):
    return node
  elif limit == 0:
    return 'cutoff'
  else:
    cutoff_occurred = False
    if node.state not in explored :
      for child in node.expand(problem):
        explored.append(node.state)
        result = recursive_dls(child, problem, limit - 1, explored)
        if result == 'cutoff':
            cutoff_occurred = True
        elif result is not None:
          return result
      if cutoff_occurred:
        return 'cutoff'
      else:
        return None

# Depth Limited Search Algorithm
def depth_limited_search(problem, limit=50):
    explored = []
    root = Node(problem.initial)
    result = recursive_dls(root, problem, limit, explored)
    return result, explored

# Iterative Deepening Depth First Search Algorithm
def iterative_deepening_depth_first_search(problem):
    for depth in range(sys.maxsize):
        result, nodes_explored = depth_limited_search(problem, depth)
        if result != 'cutoff':
            return result, nodes_explored
```

IDA* - Finally, for IDA* the recursive function is called based on the sum of the path cost and the heuristic.

```
# Iterative Deepening A* Search algorithm
def iterative_deepening_astar_search(problem):
    def recursive_ida(node, problem, limit, explored):
        if problem.goal_test(node.state):
            return node, 0
        elif node.path_cost + problem.h(node) > limit:
            return None, node.path_cost + problem.h(node)
        else:
            min_cutoff = float('inf')
            for child in node.expand(problem):
                if child.state not in explored:
                    explored.append(child.state)
                    result, child_cost = recursive_ida(child, problem, limit, explored)
                    if result is not None:
                        return result, 0
                    elif child_cost > limit:
                        min_cutoff = min(min_cutoff, child_cost)
            return None, min_cutoff

    explored = []
    root = Node(problem.initial)
    limit = problem.h(root)
    while True:
        explored.clear()
        result, cost = recursive_ida(root, problem, limit, explored)
        if result is not None:
            return result, explored
        elif cost == float('inf'):
            return None, explored
        else:
            limit = cost
```

Testing

The test cases used for this are of the following format:

[5,11] – the first line represents the number of rows and the number of columns of the grid respectively.

(0,1) – the second line represents the starting position of the robot.

(7,0) | (10,3) – the third line represents the goal states separated by "|".

(2,0,2,2) – rest of the lines from this represent a wall or obstacle.

(8,0,1,2)

(10,0,1,1)

(2,3,1,2)

(3,4,3,1)

(9,3,1,1)

(8,4,2,1)

I have used 10 different test cases for my implementation.

Bugs/Missing

- In my implementation of the search algorithms the nodes explored are coming one node less than the example provided in the assignment brief.
- I have implemented a GUI but when I run IDDFS and IDA* the way it shows the path and the nodes explored seems wrong and I was not able to figure out how to fix it.

Research

I have implemented a GUI for my research component. It is very straight forward. One can choose the algorithm from the drop down and type the name of the text file with the extension and then hit play to run the search.

Conclusion

To conclude, the best type of search algorithm is A* as it is optimal and complete and finds the best path in all the test cases as it uses a very accurate heuristic.

Acknowledgement/Resources/References

- https://chat.openai.com ChatGPT. It helped me a lot in understand of the concept and learning new skills such as tkinter and python. Moreover, it helped me a lot to troubleshoot my problems.
- https://people.engr.tamu.edu/guni/csce421/files/AI_Russell_Norvig.pdf Artificial Intelligence: A Modern Approach. This book is very helpful as it provided me pseudo code and very good description of the problems
- https://ai.stackexchange.com/questions/6426/what-is-the-difference-between-tree-search-and-graph-search AI stack exchange has a variety of information related to the topic