An Overview on A-Star Search Algorithm

Md. Refaj Hossan

A report submitted in partial fulfillment
of the requirements for the course assessment of
Bachelor of Science
in Computer Science and Engineering

Chittagong University of Engineering and Technology
Department of Computer Science Engineering
Date: 4th Decemeber, 2023

Table of Contents

Objectives

In the era of computer science and artificial intelligence, pathfinding algorithms play a vital role in solving complex problems. Among the several of algorithms designed for this purpose, the A* (A-Star) algorithm stands out for its ability to find optimal paths efficiently. This report delves into the implementation of the A* (A-Star) algorithm. Therefore, the main objectives of this report are as follows-

- To understand the basic concept of path finding algorithms.
- To understand the heuristic functions and the characteristics of a good heuristic function.
- To understand the basic mechanism behind the A-Star algorithm.
- To implement the A-Star search algorithm by using python and understand the concept practically.

Description

A-Star is a widely used pathfinding algorithm in computer science, particularly in artificial intelligence and robotics. It is an informed search algorithm that efficiently finds the shortest path from a starting node to a goal node, taking into account both the cost to reach the current state and a heuristic estimate of the cost from the current state to the goal. For this reason, A-Star search algorithm has "brain."

A-Star search algorithm picks the node according to the value of 'f', a parameter which represents the sum of the two other parameters known as 'g' and 'h'. At each step, it picks the node having the lowest 'f' and process that node. Here, the terms 'g' and 'h' represents the following things-

- **'g':** The movement cost to move from the starting node to a given node in the graph, following the path generated to get there.
- **'h':** The estimated movement cost to move from that given node to the final destination which is known as the heuristic value, a smart guess. There are many ways to calculate the value of this heuristic function. The calculation method of this value is as follows-

    1. **Exact Heuristics:** Here, one can calculate the exact value of the heuristic value but it is generally very time consuming.
    2. **Approximation Heuristics:** There are generally three approximation heuristics to calculate the value of 'h'.

- **Manhattan Distance:** It is nothing but the sum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

$$h = |current\_cell.x - goal\_cell.x| + |current\_cell.y - goal\_cell.y|$$

- **Diagonal Distance:** It is nothing but the maximum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

$$dx = |current\_cell.x - goal\_cell.x|$$
$$dy = |current\_cell.y - goal\_cell.y|$$
$$h = D * (dx + dy) + (D2 - 2 * D) * \min(dx, dy)$$

- **Euclidean Distance:** It is the direct distance between two nodes in a graph that means the distance between the current cell and the goal cell using the following formula-

$$h = SQRT(|current\_cell.x - goal\_cell.x|^2 + |current\_cell.y - goal\_cell.y|^2)$$

However, to implement the A-Star search algorithm, we need to create two lists just like Dijkstra Algorithm known as 'open list' and 'closed list'. The pseudocode for implementing the A-Star search algorithm is as follows-

Algorithm 1 A* Search Algorithm

```
Initialize the open list
Initialize the closed list
Put the starting node on the open list (you can leave its f at zero)
while the open list is not empty do
    Find the node with the least f on the open list, call it "q"
    Pop q off the open list
    Generate q's 8 successors and set their parents to q
    for each successor do
        if successor is the goal then
            Stop search
        else
            Compute both g and h for successor
            successor.g = q.g + distancebetweensuccessorandq
            successor.h = distancefromgoaltosuccessor (This can be done
    using many ways, we will discuss three heuristics - Manhattan, Diagonal,
    and Euclidean Heuristics)
            successor.f = successor.g + successor.h
            if a node with the same position as successor is in the OPEN list
    which has a lower f than successor then
                Skip this successor
            end if
            if a node with the same position as successor is in the CLOSED
    list which has a lower f than successor then
                Skip this successor
            else
                Add the node to the open list
            end if
        end if
    end for
    Push q on the closed list
end while
```

Source Code

D: > Documents_2 > Level_3_Term_2 > Sessional > Artificial Intelligence > Program_Folder > A_Star'.py > astar_search

```python
import heapq

class Node:
    def __init__(self, state, parent=None, cost=0, heuristic=0):
        self.state = state
        self.parent = parent
        self.cost = cost
        self.heuristic = heuristic

    def __lt__(self, other):
        return (self.cost + self.heuristic) < (other.cost + other.heuristic)

def astar_search(start_state, goal_state, get_neighbors, heuristic):
    start_node = Node(state=start_state, cost=0, heuristic=heuristic(start_state))
    frontier = [start_node]
    explored = set()

    while frontier:
        current_node = heapq.heappop(frontier)

        if current_node.state == goal_state:
            path = []
            while current_node:
                path.insert(0, current_node.state)
                current_node = current_node.parent
            return path

        explored.add(current_node.state)

        for neighbor_state, cost in get_neighbors(current_node.state):
            if neighbor_state not in explored:
                neighbor_node = Node(
                    state=neighbor_state,
                    parent=current_node,
                    cost=current_node.cost + cost,
                    heuristic=heuristic(neighbor_state),
                )
                heapq.heappush(frontier, neighbor_node)

    return None  # No path found

def get_neighbors(state):
    x, y = state
    neighbors = [
        ((x + 1, y), 1),
        ((x - 1, y), 1),
        ((x, y + 1), 1),
        ((x, y - 1), 1),
    ]
    return neighbors
```

```python
def heuristic(state):
    goal_state = (0, 3)
    return ((state[0] - goal_state[0]) ** 2 + (state[1] - goal_state[1]) ** 2) ** 0.5

start_state = (0, 0)
goal_state = (0, 3)
path = astar_search(start_state, goal_state, get_neighbors, heuristic)

if path:
    print("Path found:", path)
else:
    print("No path found.")
```

Sample Input & Output

Source Node = (0, 0)
Goal Node = (0, 3)
Path found: [(0, 0), (0, 1), (0, 2), (0, 3)]

The graph is as follows for this output-

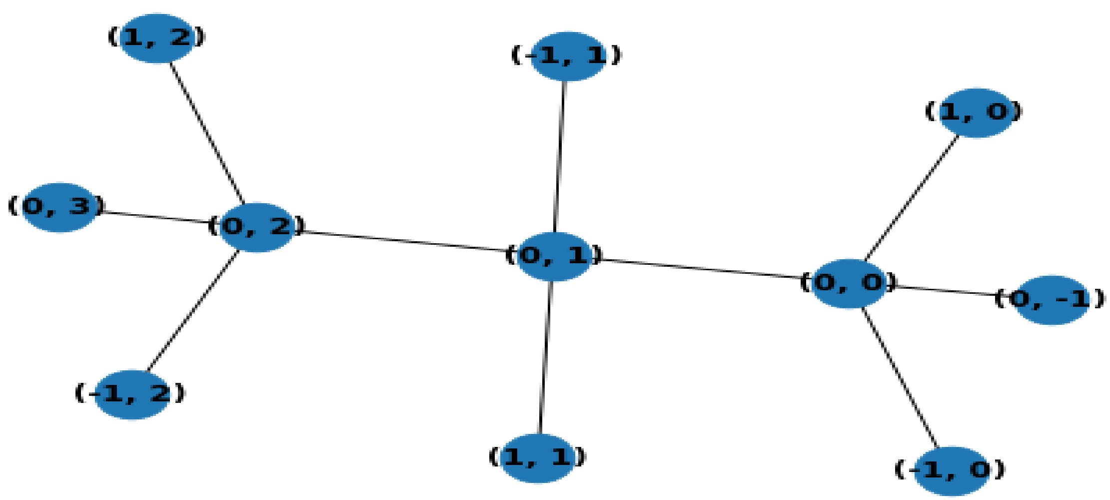

Figure 01: Sample graph for path finding using A* algorithm

Code Analysis

import heapq

This line imports the heapq module, which provides heap queue algorithms. In this context, it's used to implement a priority queue.

class Node:

def __init__(self, state, parent=None, cost=0, heuristic=0):
self.state = state
self.parent = parent

```
        self.cost = cost
        self.heuristic = heuristic
```

Here, a Node class is defined to represent a state in the search space. Each node has a state, a parent node, a cost to reach this state, and a heuristic value.

```
    def __lt__(self, other):
        return (self.cost + self.heuristic) < (other.cost + other.heuristic)
```

This method overloads the less-than (<) operator for comparing nodes based on their total cost, which is the sum of the actual cost and the heuristic. This is crucial for the priority queue to work correctly.

```
def astar_search(start_state, goal_state, get_neighbors, heuristic):
    start_node = Node(state=start_state, cost=0, heuristic=heuristic(start_state))
    frontier = [start_node]
    explored = set()
```

The astar_search function initializes the search with the start state and goal state. It creates the initial node, start_node, with a cost of 0 and the provided heuristic. frontier is a list acting as the priority queue, and explored is a set to store states that have been explored.

```
    while frontier:
        current_node = heapq.heappop(frontier)
```

The algorithm uses a priority queue (implemented as a heap) to keep track of the nodes to be explored. In each iteration, the node with the lowest total cost is popped from the heap.

```
        if current_node.state == goal_state:
            path = []
            while current_node:
                path.insert(0, current_node.state)
                current_node = current_node.parent
            return path
```

If the goal state is reached, the function reconstructs the path from the start state to the goal state using the parent pointers of each node in reverse order.

```
        explored.add(current_node.state)
        for neighbor_state, cost in get_neighbors(current_node.state):
            if neighbor_state not in explored:
                neighbor_node = Node(
                    state=neighbor_state,
                    parent=current_node,
```

```
        cost=current_node.cost + cost,
        heuristic=heuristic(neighbor_state),
    )
    heapq.heappush(frontier, neighbor_node)
```

The neighbors of the current state are generated using the get_neighbors function. For each neighbor, a new node is created. If the neighbor state is not in the explored set, it is added to the priority queue.

Discussion

The report delves into the implementation of A-Star algorithm by using Euclidean distance as heuristic value and the importance of the algorithm in complex navigation problems in the field of CS and AI as well. A-Star is a powerful algorithm that combines the benefits of Dijkstra's algorithm and greedy best-first search. Its ability to find optimal paths with the use of heuristics makes it suitable for various applications, such as robotics, gaming, and route planning. During the implementation, there were several things to handle such as what data structure to use, how to use the selected data structure i.e., heap, priority queue, how to calculate the heuristic value and so on. Therefore, there were some cases at which there was necessary to get help from the several sources from the net. However, the implementation of the algorithm is being successfully completed and the concepts behind the algorithm fascinated to further exploration towards in this field. Hence, the report provides an overview of the A* algorithm, describes its key components, presents pseudocode, and illustrates its usage in a Python implementation.

References

1. A-Star algorithm and its concepts [https://www.geeksforgeeks.org/a-search-algorithm/]

2. Python A*- the simple guide to the A-Star search algorithm [https://blog.finxter.com/python-a-the-simple-guide-to-the-a-star-search-algorithm/]

3. A* search algorithm [https://en.wikipedia.org/wiki/A*_search_algorithm]