ARTIFICIAL INTELLIGENCE & AGENT TECHNOLOGY

Project Synopsis for AAT

Aug-Dec 2018

COURSE CODE: CS562 Course Faculty: Dr. Preeti Satish

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| TITLE OF THE PROJECT | A\* Search Algorithm Implementation | | | | | | | |
|  |  | | | | | | | |
| STUDENT NAME | Mohammed Ataaur Rahaman | | | Nabil Mahtab | | | Nishant Ranjan | |
| USN | 1DS16CS721 | | | 1DS16CS722 | | | 1DS16CS724 | |
| INDIVIDUAL  CONTRIBUTION | Algorithm Designer & Debugger | | | Backend Developer  (Implementation Code) | | | Frontend Developer (A\* Search Code) | |
| GUIDE | **Dr. Preeti Satish**  Associate Professor  CSE, DSCE | | | | | | | |
|  |  | | | | | | | |
| THEORY CONCEPT | A\* Search algorithm is one of the best and popular technique used in path-finding and graph traversals. | | APPLICATION | | | To approximate the shortest path in real-life situations, like- in maps, games where there can be many hindrances. | | |
|  |  | | | | | | | |
| PROJECT ABSTRACT: | This project deals with the implementation of A\* Search using Python. We have made a grid layout of 10x10 (height x width) having a start point(A) and a end point(Z) which is the Goal. The agent has to find out the best path to reach the Goal using the A\* Search Algorithm.  Priority = Cost\_so\_far + Heuristic \_value  Priority is a queue with Heap Queue function. Which helps to find the least cost so far. | | | | | | | |
|  |  | | | | | | | |
| PLATFORM USED | **Python** | | | | | | | |
| PEAS Description | PERFORMANCE | ENVIRONMENT | | | ACTUATORS | | | SENSORS |
| * Finds the Best path | * 10x10 grid * Walls in the grid | | | * Cost estimation | | | * Heuristic values |
|  |  | | | | | | | |
| PROJECT DESCRIPTION | Like Dijkstra, A\* works by making a lowest-cost path tree from the start node to the goal node. What makes A\* different and better for many searches is that for each node, A\* uses a function f(n) that gives an estimate of the total cost of a path using that node. Therefore, A\* is a heuristic function, which differs from an algorithm in that a heuristic is more of an estimate and is not necessarily provably correct.  f(n) = g(n) + h(n)  A\* expands paths that are already less expensive by using this function:  where   * f(n) = total estimated cost of path through node * g(n) = cost so far to reach node * h(n) = estimated cost from  to goal. This is the heuristic part of the cost function, so it is like a guess.   A non-efficient way to find a path    An example of using A* algorithm to find a path  **Heuristics**  Cost to travel to the adjacent node = 10  Cost to travel to the diagonal adjacent node is approximated to be **14** for the purpose of shortest heuristic distance (**The Euclidean Distance Heuristic)**. | | | | | | | |
|  |  | | | | | | | |
| Conclusion /FUTURE ENHANCEMENT | * This heuristic is slightly more accurate than its Manhattan counterpart. If we try run both simultaneously on the same maze, the Euclidean path finder favors a path along a straight line. This is more accurate but it is also slower because it has to explore a larger area to find the path. * Dijkstra is a special case of A\* Search Algorithm, where h = 0 for all nodes. * Future Enhancements: To implement A\* Search in finding the path in Maze puzzle. | | | | | | | |
|  |  | | | | | | | |
| Project Source Code Link (Github/ Google DRive) | **Link to the Project:**  <https://github.com/Ataago/AI-A-Search>  **Implementation.py**  import heapq  def from\_id\_width(id, width):  return (id % width, id // width)  def draw\_tile(graph, id, style, width):  r = "."  if 'number' in style and id in style['number']: r = "%d" % style['number'][id]  if 'point\_to' in style and style['point\_to'].get(id, None) is not None:  (x1, y1) = id  (x2, y2) = style['point\_to'][id]  if x2 == x1 + 1: r = ">"  if x2 == x1 - 1: r = "<"  if y2 == y1 + 1: r = "v"  if y2 == y1 - 1: r = "^"  if 'start' in style and id == style['start']: r = "A"  if 'goal' in style and id == style['goal']: r = "Z"  if 'path' in style and id in style['path']: r = "@"  if id in graph.walls: r = "#" \* width  return r  def draw\_grid(graph, width=2, \*\*style):  for y in range(graph.height):  for x in range(graph.width):  print("%%-%ds" % width % draw\_tile(graph, (x, y), style, width), end="")  print()  class SquareGrid:  def \_\_init\_\_(self, width, height):  self.width = width  self.height = height  self.walls = []    def in\_bounds(self, id):  (x, y) = id  return 0 <= x < self.width and 0 <= y < self.height    def passable(self, id):  return id not in self.walls    def neighbors(self, id):  (x, y) = id  results = [(x+1, y), (x, y-1), (x-1, y), (x, y+1)]  if (x + y) % 2 == 0: results.reverse() # aesthetics  results = filter(self.in\_bounds, results)  results = filter(self.passable, results)  return results  class GridWithWeights(SquareGrid):  def \_\_init\_\_(self, width, height):  super().\_\_init\_\_(width, height)  self.weights = {}    def cost(self, from\_node, to\_node):  return self.weights.get(to\_node, 1)  diagram4 = GridWithWeights(10, 10)  diagram4.walls = [(1, 7), (1, 8), (2, 7), (2, 8), (3, 7), (3, 8),(8,0),(8,1),(8,2)]  diagram4.weights = {loc: 5 for loc in [(3, 4), (3, 5), (4, 1), (4, 2),  (4, 3), (4, 4), (4, 5), (4, 6),  (4, 7), (4, 8), (5, 1), (5, 2),  (5, 3), (5, 4), (5, 5), (5, 6),  (5, 7), (5, 8), (6, 2), (6, 3),  (6, 4), (6, 5), (6, 6), (6, 7),  (7, 3), (7, 4), (7, 5)]}  class PriorityQueue:  def \_\_init\_\_(self):  self.elements = []    def empty(self):  return len(self.elements) == 0    def put(self, item, priority):  heapq.heappush(self.elements, (priority, item))    def get(self):  return heapq.heappop(self.elements)[1]  **A\_Search.py**  from implementation import \*  def heuristic(a, b):  (x1, y1) = a  (x2, y2) = b  distX = abs(x1 - x2)  distY = abs(y1 - y2)  if distX > distY:  return distY \* 14 + 10 \* (distX - distY)  return distX \* 14 + 10 \* (distY - distX)  def a\_star\_search(graph, start, goal):  frontier = PriorityQueue()  frontier.put(start, 0)  came\_from = {}  cost\_so\_far = {}  came\_from[start] = None  cost\_so\_far[start] = 0    while not frontier.empty():  current = frontier.get()    if current == goal:  break    for next in graph.neighbors(current):  new\_cost = cost\_so\_far[current] + graph.cost(current, next)  if next not in cost\_so\_far or new\_cost < cost\_so\_far[next]:  cost\_so\_far[next] = new\_cost  priority = new\_cost + heuristic(goal, next)  frontier.put(next, priority)  came\_from[next] = current    return came\_from, cost\_so\_far  start, goal = (0, 9), (9, 0)  came\_from, cost\_so\_far = a\_star\_search(diagram4, start, goal)  print()  draw\_grid(diagram4, width=3, point\_to=came\_from, start=start, goal=goal)  print()  draw\_grid(diagram4, width=3, number=cost\_so\_far, start=start, goal=goal)  print() | | | | | | | |
|  |  | | | | | | | |
| Ui sCreenshots | Grid:  Wall  START  END  Path Matrix:    Cost Matrix: | | | | | | | |