THE NATIONAL ECONOMIC UNIVERSITY

-------***-------

DATA STRUCTURE AND ALGORITHM ASSIGNMENT

Navigating the Frontier: An Exploration of the A Search Algorithm*

Teacher: Vu Duc Minh

Class: DSEB 64A

Member: Le Hoang Minh 11224192

Mai Thanh Loc 11223875

Vu Quoc Huy 11222839

Tran Dam Quoc Khanh 11223079

Nguyen Duy Hung 11222611

Tran The Duy 11221684

HA NOI, 2023

TABLE OF CONTENT

A. INTRODUCTION 3
B. MAIN CONTENT 3
I. Definition 3
II. The algorithm's mechanics 4
1. Algorithm 4
2. Graphical explanation 5
3. Heuristics: speed or accuracy? 11
3.1. Exact Heuristics 11
3.2. Approximation Heuristics 11
4. Limitations 14
5. Time Complexity 14
6. Auxiliary Space 14
III. Real-life application 14
IV. Comparison with Dijkstra's algorithm 14
C. CONCLUSION 16
D. APPENDIX 16
E. REFERENCES 19

A. INTRODUCTION

The quest for efficient pathfinding algorithms lies at the heart of artificial intelligence (AI) and various computer science applications. Within this realm, the **A*** search algorithm stands out as a powerful tool. Unlike simpler search techniques that explore a graph in a piecemeal fashion, **A*** boasts an informed approach, strategically navigating complex networks to find the optimal path between a starting point and a desired goal. This "optimality" can take many forms: shortest distance, fastest travel time, or even minimizing resource expenditure, depending on the specific problem being addressed. **A*** achieves this efficiency by incorporating additional knowledge, encoded as a heuristic function, to prioritize the most promising paths at each step.

This report delves into the intricate workings of **A***. We'll explore the mechanics behind its informed search, examining how it leverages heuristics to make intelligent decisions. We'll also analyze the factors that influence the algorithm's efficiency, providing insights into its strengths and limitations. Finally, we'll showcase the diverse applications of **A***, highlighting its impact across various fields, from game development and robotics to route planning and logistics optimization. Through this exploration, we'll gain a deeper understanding of how **A*** empowers intelligent systems to navigate complex environments and optimally achieve their goals.

B. MAIN CONTENT

I. DEFINITION

A* (A-star) is a graph traversal and pathfinding algorithm, which is used in many fields of computer science due to its completeness, optimality, and optimal efficiency. Given a weighted graph, a source node, and a goal node, the algorithm finds the shortest path (with respect to the given weights) from source to goal.

Peter Hart, Nils Nilsson, and Bertram Raphael of Stanford Research Institute (now SRI International) first published the algorithm in 1968. It can be seen as an extension of Dijkstra's algorithm. A* achieves better performance by using heuristics to guide its search.

Why A* Search Algorithm?

Informally speaking, A* Search algorithms, unlike other traversal techniques, it has “brains”. What it means is that it is really a smart algorithm that separates it from the other conventional algorithms. This fact is cleared in detail in the below sections. It is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

II. THE OPERATION OF THE ALGORITHM

Consider a graph having nodes and we are given a starting node and a target node. We want to reach the target node (if possible) from the starting node as quickly as possible. Here A* Search Algorithm comes to the rescue. What A* Search Algorithm does is that at each step it picks the node according to a value:

$$\mathbf{f(n) = g(n) + h(n)}$$

At each step it picks the node having the lowest '**f(n)**', and processes that node/cell. We define '**g(n)**' and '**h(n)**' as simply as possible below:

- **f(n)** = the estimate of the total cost from start node to target node through node n

- **g(n)** = the movement cost to move from the starting node to a given node n, following the path generated to get there.

- **h(n)** = the estimated movement cost to move from that given node n on the graph to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate this '**h(n)**' which are discussed in the later sections.

If the heuristic function is admissible – meaning that it never overestimates the actual cost to get to the goal – **A*** is guaranteed to return a least-cost path from start to goal.

1. Algorithm

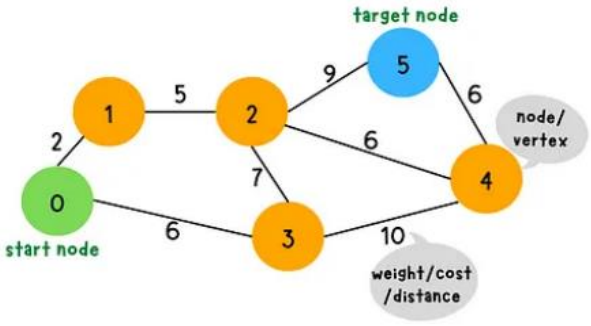
```
    # initialize a min Heap contains the start node
minHeap = MinHeap([start])While minHeap is not empty:
  # grab the minimum f-value node from minHeap and denote as current
  current = minHeap.pop()  # check if current is the target node
  if current == target:
      break

  # populate all current node's neighbors
  for neighbor in current.neighbors:
      compute neighbor's g, h, f value

      if neighbor in minHeap:
         if neighbor.g < neighbor.g in minHeap:
```

```
            update minHeap
    else:
       insert neighbor into minHeap
```

2. Graphical explanation



h(n)

node	Heuristic distance to target node
0	20
1	16
2	6
3	10
4	4
5	0

- ***Step 1***
  - *a*: Set the distance to the **start node** itself to **0** and the distance to **all other nodes** to **infinity**.
  - *b*: Calculate the **f value (** for the start node and set the **previous node** to none/nil/null.

    f(n) = g(n) + h(n)
  - *c*: Initialize an **open list** and **close list** which are empty initially.

- ***Step 2****: Place the* ***start node*** *into the* ***open list***
  Open list: [0]
  Close list: []

node	status	g(n)	h(n)	f(n)	Previous node
0(start)	open	0	20	20	None
1		∞	16		
2		∞	6		
3		∞	10		
4		∞	4		
5(target)		∞	0		

- ***Step 3***
  - *a*: Find the node with the minimum f-value in the open list and removed from the list. Denote this node as the current node.
  - *b*: Check if the current node is the target node or not
  - *c*: Populate all the current node's neighboring nodes and do following checks for each neighboring nodes
  - *d*: Place the current node to the close list because we have expanded this node.

- ***Step 4****: Repeat* Step 3 *until reaches the target node*

Demo iteration 1

- ***Step 3.a****:* Find the node with the lowest f cost in the open list and remove it from the list
  Only one node on the open list. So chose node 0 to remove and explore.
  Denote node 0 as current node.
  Open list: []
  Close list: []
  Current = node 0

- ***Step 3.b:*** Current node == target node?
  No. node 0 != node 5

- ***Step 3.c:*** find all current node's neighboring node.
  Neighboring node = node 1, node 3

  Check node 1

  - ***Step 3.c.1:*** Is node 1 on the close list? – No
  - ***Step 3.c.2:*** Calculate g cost of node 1 and check if we find a better path (lower g-value).

    g(n) = min(∞, 0 + 2) = 2
  - Step 3.c.3: Update g, h and f cost of node 1 and store current node as previous node.
  - Step 3.c.4: Is node 1 on the open list? – No

    Insert node 1 to the open list.

    Open list: [1]
    Close list: []

  Check node 3

  - ***Step 3.c.1:*** Is node 3 on the close list? – No
  - ***Step 3.c.2:*** Calculate g cost of node 3 and check if we find a better path (lower g-value).

g(n) = min(∞, 0 + 6) = 6

- ***Step 3.c.3:*** Update g, h and f cost of node 3 and store current node as previous node.
- ***Step 3.c.4:*** Is node 3 on the open list? – No
  Insert node 3 to the open list.

- ***Step 3.d:*** Put current node that we have expanded to the close list.
  Open list: [1, 3]
  Close list: [0]

node	status	g(n)	h(n)	f(n)	Previous node
0(start)	close	0	20	20	None
1	open	2	16	18	0
2		∞	6		
3	open	6	10	16	0
4		∞	4		
5(target)		∞	0		

Demo iteration 2

- ***Step 3.a:*** Find the node with the lowest f cost in the open list and remove it from the list
  Select node 3 as current node (the lowest f value)
  Open list: [1]
  Close list: [0]
  Current = node 3

- ***Step 3.b:*** Current node == target node?
  No. node 3 != node 5

- ***Step 3.c:*** find all current node's neighboring node.
  Neighboring node = node 0, node 2, node 4
  Check node 0
  - ***Step 3.c.1:*** Is node 0 on the close list? – Yes → Ignore it.

  Check node 2
  - ***Step 3.c.1:*** Is node 2 on the close list? – No
  - ***Step 3.c.2:*** Calculate g cost of node 2 and check if we find a better path (lower g-value).

    g(n) = min(∞, 6 + 7) = 13

  Step 3.c.3: Update g, h and f cost of node 2 and store current node as previous node.

- Step 3.c.4: Is node 2 on the open list? – No
  Insert node 2 to the open list.

Check node 4

- ***Step 3.c.1:*** Is node 4 on the close list? – No
- ***Step 3.c.2:*** Calculate g cost of node 4 and check if we find a better path (lower g-value).
  g(n) = min(∞, 6 + 10) = 16
- ***Step 3.c.3:*** Update g, h and f cost of node 4 and store current node as previous node.
- ***Step 3.c.4:*** Is node 4 on the open list? – No
  Insert node 4 to the open list.

- ***Step 3.d:*** Put current node that we have expanded to the close list.
  Open list: [1, 2, 4]
  Close list: [0, 3]

node	status	g(n)	h(n)	f(n)	Previous node
0(start)	close	0	20	20	None
1	open	2	16	18	0
2	open	13	6	19	3
3	close	6	10	16	0
4	open	16	4	20	3
5(target)		∞	0		

Demo iteration 3

- ***Step 3.a:*** Find the node with the lowest f cost in the open list and remove it from the list
  Select node 1 as current node (the lowest f value)
  Open list: [2, 4]
  Close list: [0, 3]
  Current = node 1

- ***Step 3.b:*** Current node == target node?
  No. node 1 != node 5

- ***Step 3.c:*** find all current node's neighboring node.
  Neighboring node = node 0, node 2

Check node 0

- ***Step 3.c.1:*** Is node 0 on the close list? – Yes → Ignore it.

Check node 2

- ***Step 3.c.1:*** Is node 2 on the close list? – No

- ***Step 3.c.2:*** Calculate g cost of node 2 and check if we find a better path (lower g-value).

  g(n) = min(13, 2 + 5) = 7
- ***Step 3.c.3:*** Update g, h and f cost of node 2 and store current node as previous node.
- ***Step 3.c.4:*** Is node 2 on the open list? – No

  Insert node 2 to the open list.

- ***Step 3.d:*** Put current node that we have expanded to the close list.

  Open list: [2, 4]

  Close list: [0, 3, 1]

node	status	g(n)	h(n)	f(n)	Previous node
0(start)	close	0	20	20	None
1	close	2	16	18	0
2	open	7	6	13	1
3	close	6	10	16	0
4	open	16	4	20	3
5(target)		∞	0		

Demo iteration 4

- ***Step 3.a:*** Find the node with the lowest f cost in the open list and remove it from the list

  Select node 2 as current node (the lowest f value)

  Open list: [4]

  Close list: [0, 3, 1]

  Current = node 2

- ***Step 3.b:*** Current node == target node?

  No. node 2 != node 5

- ***Step 3.c:*** find all current node's neighboring node.

  Neighboring node = node 1, node 5

  Check node 1

  - ***Step 3.c.1:*** Is node 1 on the close list? – Yes → Ignore it.

  Check node 5

  - ***Step 3.c.1:*** Is node 5 on the close list? – No
  - ***Step 3.c.2:*** Calculate g cost of node 5 and check if we find a better path (lower g-value).

    g(n) = min(∞, 7 + 9) = 16
  - ***Step 3.c.3:*** Update g, h and f cost of node 5 and store current node as previous node.

- ***Step 3.c.4***: Is node 5 on the open list? – No
  Insert node to the open list.

- ***Step 3.d:*** Put current node that we have expanded to the close list.
  Open list: [4, 5]
  Close list: [0, 3, 1, 2]

node	status	g(n)	h(n)	f(n)	Previous node
0(start)	close	0	20	20	None
1	close	2	16	18	0
2	close	7	6	13	1
3	close	6	10	16	0
4	open	16	4	20	3
5(target)	open	16	0	16	2

Demo iteration 5

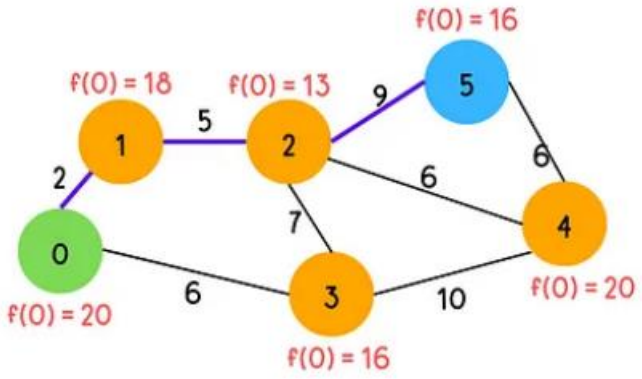
- ***Step 3.a:*** Find the node with the lowest f cost in the open list and remove it from the list
  Select node 5 as current node (the lowest f value)
  Open list: [4]
  Close list: [0, 3, 1, 2]
  Current = node 5

- ***Step 3.b:*** Current node == target node?
  Yes → Stop search

THE FINAL RESULT

The shortest path = 0 → 1 → 2 → 5 (backtracking to construct the shortest path)

node	status	g(n)	h(n)	f(n)	Previous node
0(start)	close	0	20	20	None
1	close	2	16	18	0
2	close	7	6	13	1
3	close	6	10	16	0
4	open	16	4	20	3
5(target)	current	16	0	16	2



3. Heuristics: speed or accuracy?

We can calculate **g(n)** but how to calculate **h(n)**?
We can do things.
A) Either calculate the exact value of h (which is certainly time consuming).

OR

B) Approximate the value of h using some heuristics (less time consuming).
We will discuss both of the methods.

3.1. Exact Heuristics

We can find exact values of h, but that is generally very time consuming.

Below are some of the methods to calculate the exact value of h.

1) Pre-compute the distance between each pair of cells before running the **A*** Search Algorithm.
2) If there are no blocked cells/obstacles then we can just find the exact value of h without any pre-computation using the distance formula/Euclidean Distance

3.2. Approximation Heuristics

There are generally three approximation heuristics to calculate h(n):

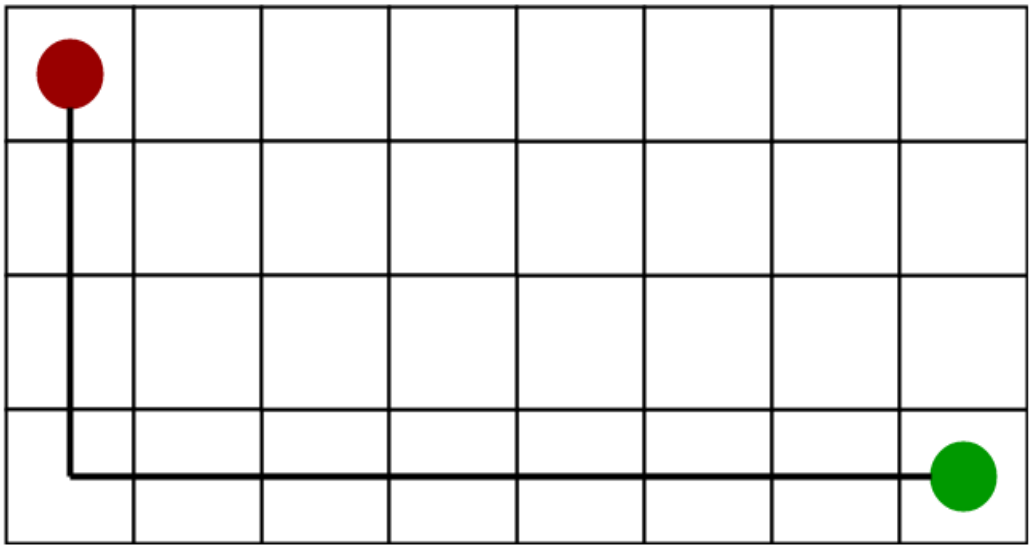
a. Manhattan Distance

- 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 = abs (current_cell.x - goal.x) +
    abs (current_cell.y - goal.y)
```

- When to use this heuristic? – When we are allowed to move only in four directions only (right, left, top, bottom)

The Manhattan Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).



b. Diagonal Distance

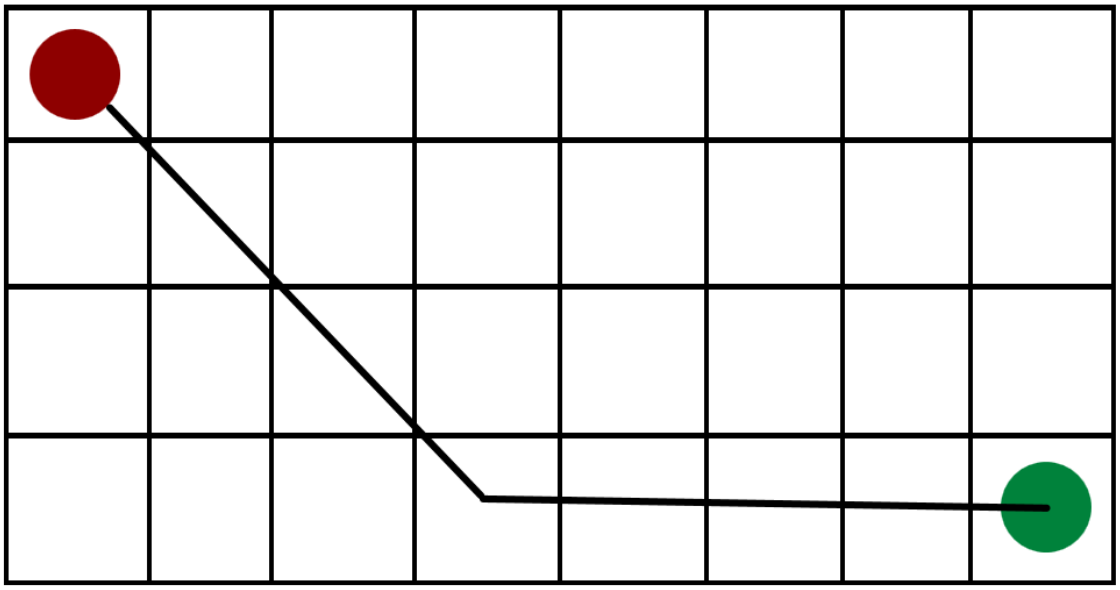
- It is nothing but the maximum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

```
dx = abs(current_cell.x – goal.x)
dy = abs(current_cell.y – goal.y)

h = D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
where D is length of each node(usually = 1) and D2 is diagonal distance
between each node (usually = sqrt(2) ).
```

- When to use this heuristic? – When we are allowed to move in eight directions only (similar to a move of a King in Chess)

The Diagonal Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).



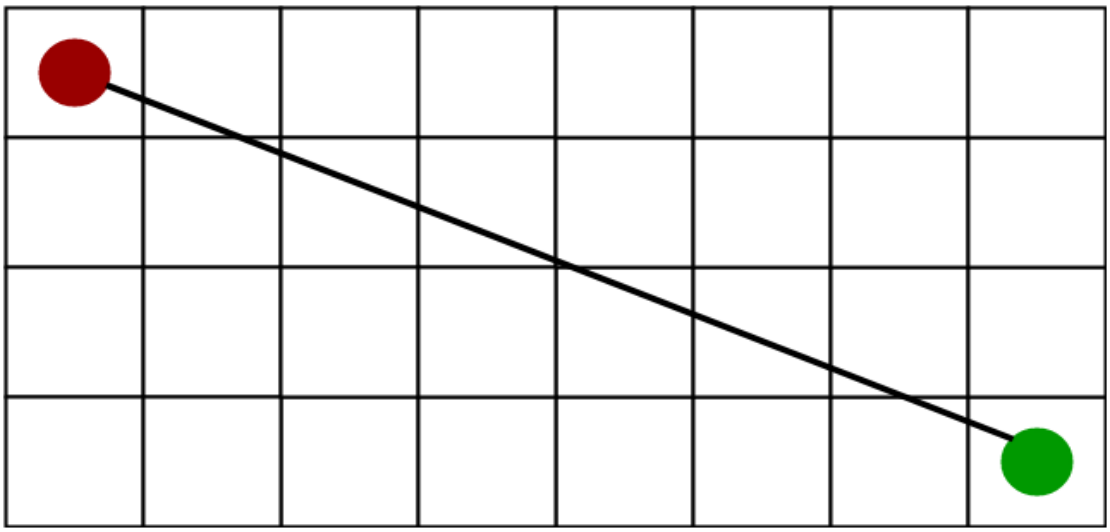
c. Euclidean Distance

- As it is clear from its name, it is nothing but the distance between the current cell and the goal cell using the distance formula

```
h = sqrt ( (current_cell.x - goal.x)^2 +
           (current_cell.y - goal.y)^2 )
```

- When to use this heuristic? – When we are allowed to move in any directions.

The Euclidean Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).



4. Limitations

Despite being the best path finding algorithm around, A* Search Algorithm doesn't always produce the shortest path, as it relies heavily on heuristics/ approximations to calculate – h(n)

5. Time Complexity

Considering a graph, it may take us to travel all the edge to reach the destination cell from the source cell .For example, consider a graph where source and destination nodes are connected by a series of edges, like – 0(source) →1 → 2 → 3 (target)

So the worst case time complexity is **O(E)**, where **E** is the number of edges in the graph

The time complexity is polynomial when the search space is a tree, there is a single goal state, and the heuristic function h meets the following condition:

$$|h(x) - h^*(x)| = O(\log h^*(x))$$

Where h* is the optimal heuristic, the exact cost to get from x to the goal. In other words, the error of h will not grow faster than the logarithm of the "perfect heuristic" h* that returns the true distance from x to the goal.

6. Auxiliary Space

In the worst case, we can have all the edges inside the open list, so the required auxiliary space in the worst case is **O(V),** where **V** is the total number of vertices/nodes.

The space complexity of **A*** is roughly the same as that of all other graph search algorithms, as it keeps all generated nodes in memory. In practice, this turns out to be the biggest drawback of the **A*** search, leading to the development of memory-bounded heuristic searches, such as **Iterative deepening A***, **memory-bounded A***, and **SMA***.

III. Real-life applications

The **A*** algorithm, while famous for video game pathfinding, is a versatile graph traversal tool. It goes beyond games, finding applications in:

- **NLP Parsing: A*** efficiently navigates potential sentence structures in NLP, guided by a heuristic function to find grammatically correct sentences.
- **Informational Search: A*** excels in online learning searches, dynamically adjusting the search path based on relevance to find the most valuable information faster.

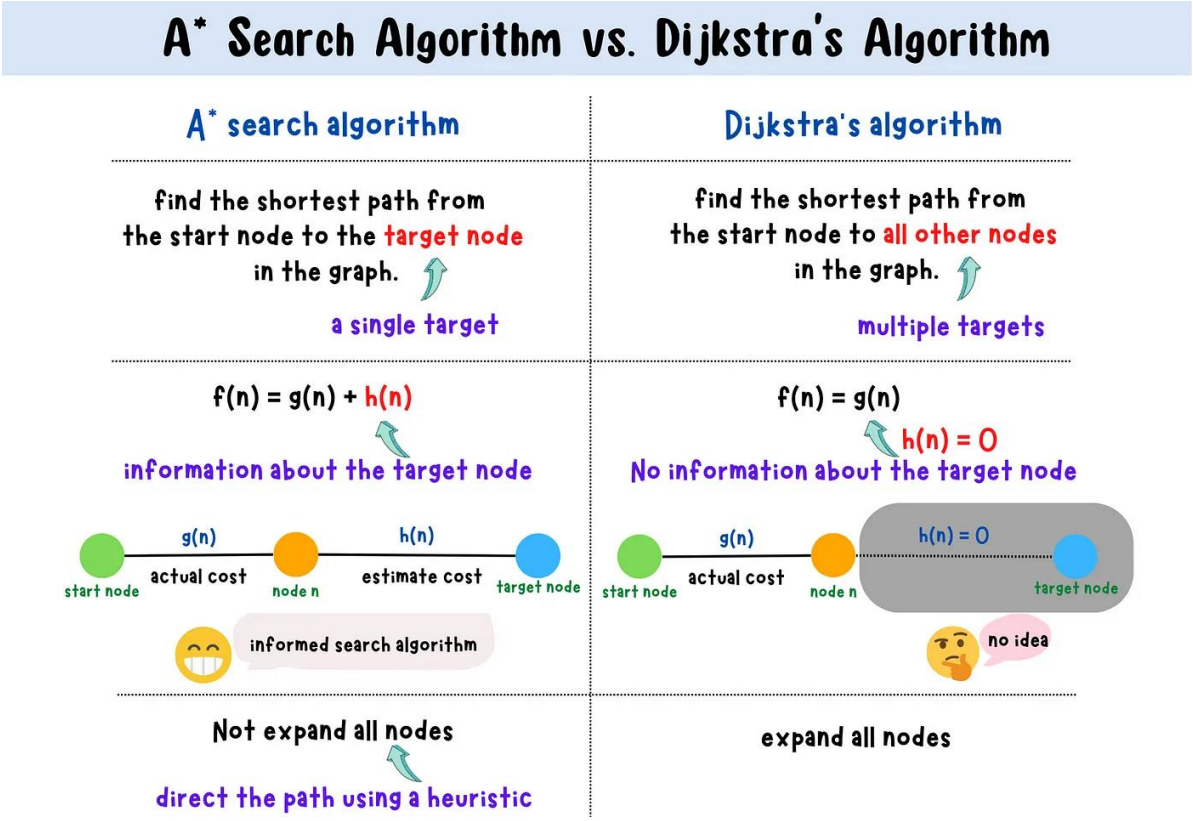
A*'s strength lies in "informed search." It prioritizes paths likely to lead to the goal using a heuristic function. This makes **A*** efficient (avoids irrelevant paths) and optimal (guarantees the shortest path with an admissible heuristic).

Beyond games, **A*** is used in robot navigation, route planning, and even machine learning. **A***'s adaptability and informed search make it a cornerstone of problem-solving across various domains.

IV. Compare with Dijkstra's algorithm

A* Algorithm only finds the shortest path between the start node and the target node, whereas **Dijkstra's** algorithm finds the shortest path from the start node to all other nodes because every node in a graph is a target node for it.

A* algorithm runs faster than **Dijkstra's** algorithm because it uses a heuristic to direct in the correct direction towards the target node. However, **Dijkstra's** algorithm expands evenly in all of the different available directions because it has no knowledge regarding the target node before hand, it always processes the node that is the closest to the start node based on the cost or distance it already took.



C. CONCLUSION

In conclusion, the A* algorithm emerges as a cornerstone in search and pathfinding applications, offering a balanced approach that considers both actual and estimated costs through heuristic evaluation. This implementation exemplifies its efficacy in navigating graphs efficiently, as demonstrated by its ability to find the shortest path while adapting to various problem domains. Beyond its fundamental role in pathfinding, A* finds widespread application across fields such as robotics, gaming, and network routing, showcasing its adaptability and effectiveness in diverse scenarios. Its continued relevance and versatility underscore its status as a go-to solution for engineers and researchers seeking optimal, computationally feasible solutions to complex search challenges.

D. APPENDIX

```
1   import random
2   import math
3   import networkx as nx
4   import matplotlib.pyplot as plt
5   import heapq
6
7   def heuristic_mahattan(point1, point2):
8       x1, y1 = point1
9       x2, y2 = point2
10      dx = abs(x1 - x2)
11      dy = abs(y1 - y2)
12      return dx + dy
13
14  def heuristic_diagonal(point1, point2):
15      x1, y1 = point1
16      x2, y2 = point2
17      dx = abs(x1 - x2)
18      dy = abs(y1 - y2)
19      return (dx + dy) + (math.sqrt(2) - 2) * min(dx, dy)
20
21  def heuristic_euclidean(point1, point2):
22      x1, y1 = point1
23      x2, y2 = point2
24      dx = abs(x1 - x2)
25      dy = abs(y1 - y2)
26      return math.sqrt(dx**2 + dy**2)
27
28  class Graph:
29      def __init__(self):
```

```
30          self.start = 0
31          self.end = 0
32          self.points = []
33          self.edges = {}
34          for i in range(len(self.points)):
35              self.edges[i] = set()
36
37      def generate_points(self, n):
38          self.points = [[random.randint(0, 100), random.randint(0,
100)] for i in range(n)]
39          for i in range(len(self.points)):
40              self.edges[i] = set()
41
42      def distance(self, point1, point2):
43          return math.sqrt((self.points[point1][0] -
self.points[point2][0])**2 +
44                           (self.points[point1][1] -
self.points[point2][1])**2)
45
46      def generate_edges(self, density):
47          for i in range(len(self.points)):
48              distance_from_point_i = sorted([[k, self.distance(i, k)]
for k in range(len(self.points))],
49                                             key=lambda x: x[1])[1:]
50              for j in range(density):
51                  self.edges[i].add(distance_from_point_i[j][0])
52                  self.edges[distance_from_point_i[j][0]].add(i)
53
54  def draw_graph(graph, path=[]):
55      G = nx.Graph()
56      edge_list = list(zip(path, path[1:]))
57      for i in range(len(graph.points)):
58          G.add_node(i, pos=graph.points[i])
59      node_color_map = ['red' if node in path else 'blue' for node in
G]
60      for point1, point2s in graph.edges.items():
61          for point2 in point2s:
62              G.add_edge(point1, point2, color='red' if ((point1,
point2) in edge_list) else 'black',
63                         label=round(graph.distance(point1, point2),
2))
64      edge_color_map = 'black'
65      if len(path) > 0:
66          edge_color_map = [G[u][v]['color'] for u, v in G.edges]
67
68      fig, ax = plt.subplots(figsize=(10, 5))
69      pos = nx.get_node_attributes(G, 'pos')
70      nx.draw(G, pos, node_color=node_color_map,
edge_color=edge_color_map, node_size=100, ax=ax)
71      nx.draw_networkx_labels(G, pos, font_size=6)
```

```
72      nx.draw_networkx_edge_labels(G, pos, font_size=5, ax=ax,
edge_labels=nx.get_edge_attributes(G, 'label'))
73      ax.set_axis_on()
74      ax.tick_params(left=True, bottom=True, labelleft=True,
labelbottom=True)
75
76  random.seed(4)
77  graph1 = Graph()
78  graph1.generate_points(10)
79  graph1.generate_edges(2)
80  draw_graph(graph1)
81
82  def astar(graph, start, end):
83      open_list = []
84      closed_set = set()
85      g_values = {node: float('inf') for node in
range(len(graph.points))}
86      g_values[start] = 0
87      parent = {}
88
89      heapq.heappush(open_list, (0, start))
90
91      while open_list:
92          current_f, current_node = heapq.heappop(open_list)
93
94          if current_node == end:
95              path = []
96              while current_node is not None:
97                  path.append(current_node)
98                  current_node = parent.get(current_node)
99              return path[::-1]
100
101         closed_set.add(current_node)
102
103         for neighbor in graph.edges[current_node]:
104             if neighbor in closed_set:
105                 continue
106
107             tmp = g_values[current_node] +
graph.distance(current_node, neighbor)
108
109             if tmp < g_values[neighbor]:
110                 g_values[neighbor] = tmp
111                 parent[neighbor] = current_node
112                 h_value = heuristic_euclidean(graph.points[neighbor],
graph.points[end])
113                 heapq.heappush(open_list, (tmp + h_value, neighbor))
114
115     return None  # algorithm cant find the path
116
```

```
117 path = astar(graph1, 3, 6)
118 if path:
119     print(path)
120 else:
121     print("path can't be found")
```

E. REFERENCES

Time-efficient A* Algorithm for Robot Path Planning - ScienceDirect

A Systematic Literature Review of A* Pathfinding - ScienceDirect

A* Search - Codecademy

A* Search Algorithm - Geeksforgeeks

A* Search Algorithm - Medium

Introduction to A* - Theory.stanford.edu