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

DATA STRUCTURE AND ALGORITHM - Group 4

# TABLE OF CONTENT


01 **Problem**

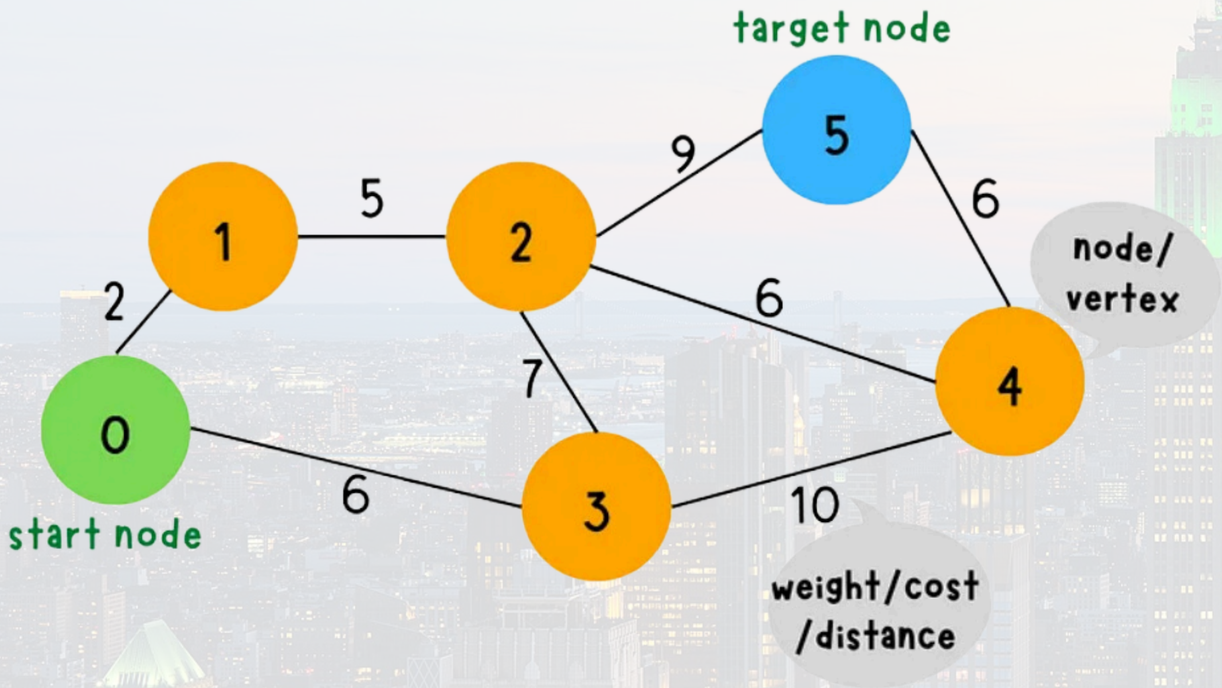
02 **A* search algorithm**

03 **Real-life application**

04 **Comparison with other algorithms**


# I. PROBLEM

# I. PROBLEM



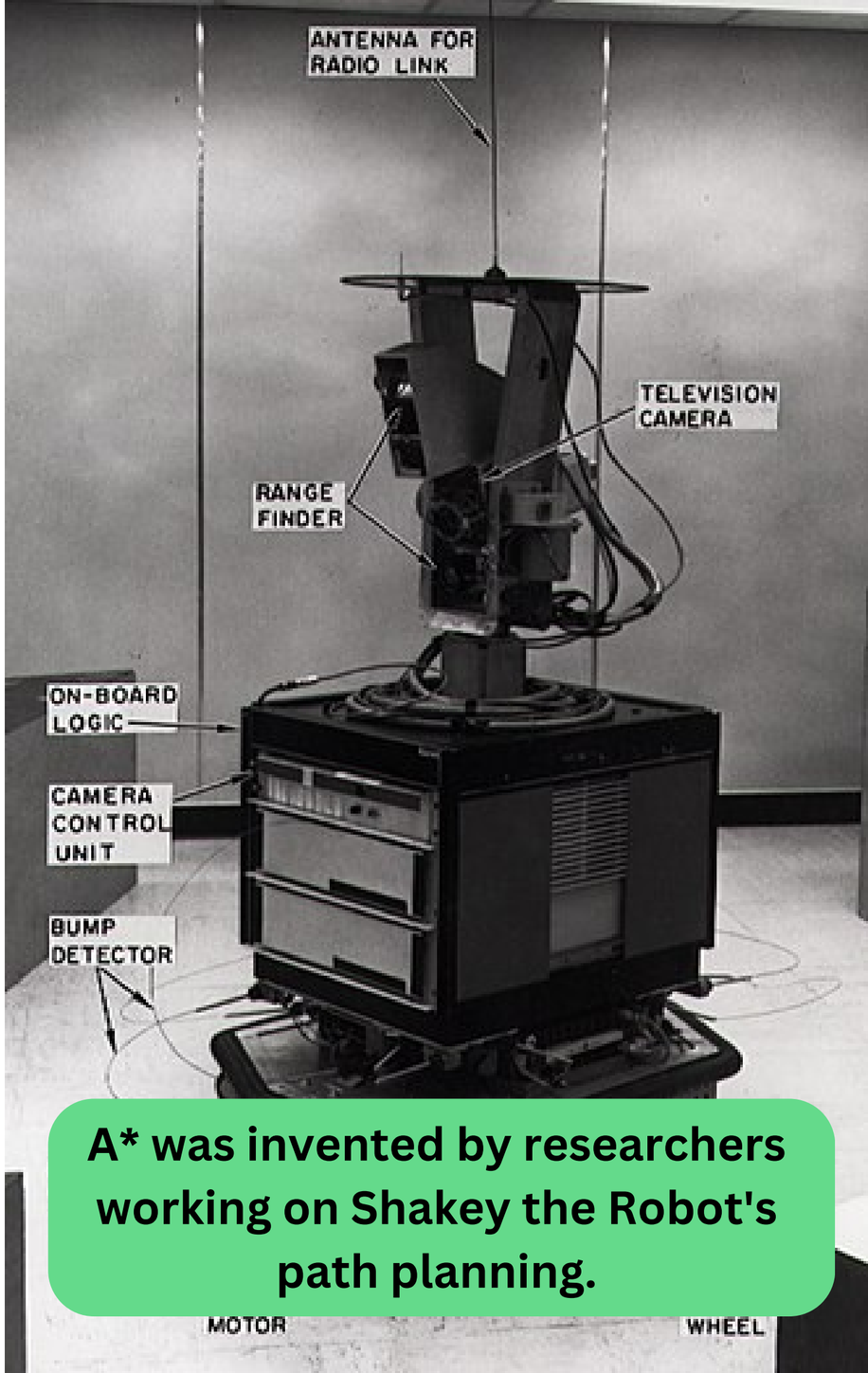

Imagine a map as a web of connections, where cities are represented by nodes and roads are lines. You are in the start node, and you want to go to the target node. So which way is the fastest way?

# II. A* ALGORITHM

# II. A* ALGORITHM

## 1. Definitions:

First invented by Peter Hart, Nils Nilsson, and Bertram Raphael in 1968, A* (A-star) is a graph traversal and pathfinding algorithm. It finds the shortest path (with respect to the given weights) from source to goal.




**A* was invented by researchers working on Shakey the Robot's path planning.**

# II. A* ALGORITHM

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:

Dijkstra's algorithm

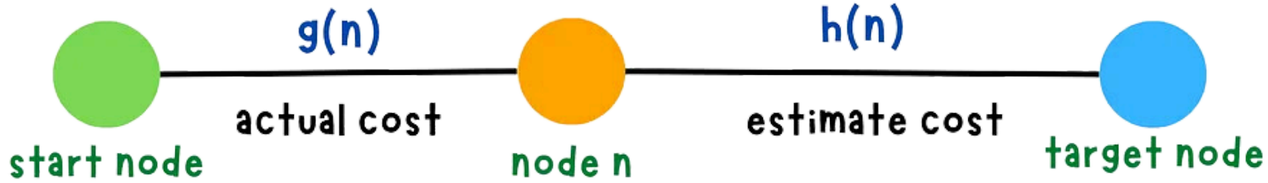
$$f(n) = g(n) + h(n)$$

f(n): the estimate of the total cost from start node to target node through node n

g(n): actual cost from start node to node n

h(n): estimated cost from node n to target node

heuristic function



# II. A* ALGORITHM

## 2. Pseudocode:

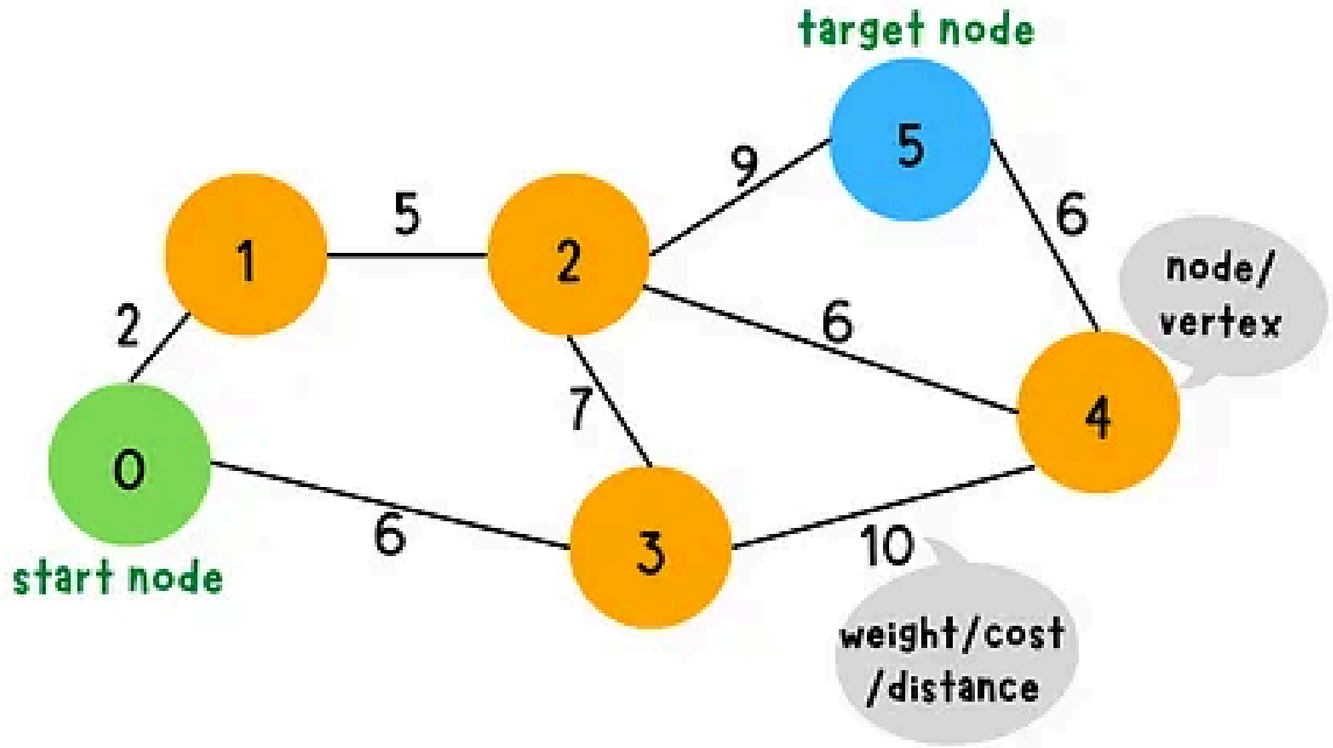
```
# initialize a min Heap contains the start node
minHeap = MinHeap([start])While minHeap is not empty:
# grab the minimum f-value node from minHeap and denote as current
current = minHeap.pop() # check if current is the target node
if current == target:
    break

# 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
```

# II. A* ALGORITHM

## 3. Graphical explanation:




h(n)

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

# II. A* ALGORITHM

## 3. Graphical explanation:

**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.
- **c**: Initialize an **open list** and **close list** which are empty initially.

**Step 2**: Place the **start node** into the **open list**

**Step 3:**

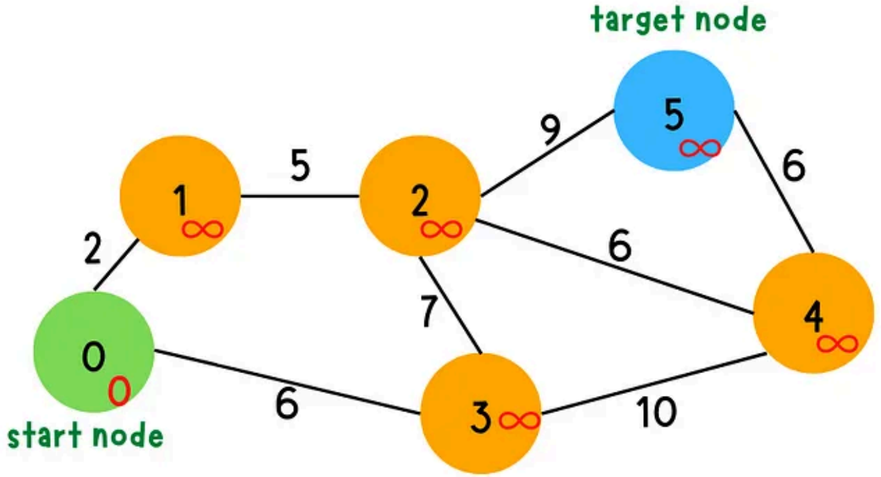
- **a**: Find the node with **the minimum f-value** in the **open list** and **removed** from the list. Denote this node as the current node.
- **b**: Check if the **current node** is the **target node** or not
- **c**: **Populate** all the current node's **neighboring nodes** and do following checks 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

# II. A* ALGORITHM

## 3. Graphical explanation:

**step 1.a:** Set distance from start node itself to 0,
Set distance from start node to other nodes to infinity.




**step 1.b:** Calculate the f value for the start node and set previous node to None.

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

**step 1.c:** Initialize open list and close list

open list: []
close list: []

open list: a collection of nodes that needs to be examined
close list: a collection of visited nodes that are already examined

# II. A* ALGORITHM

## 3. Graphical explanation:

**step 2:** Put the start node to 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 | | |

# II. A* ALGORITHM

## 3. Graphical explanation:

iteration 1

step 3.a: Find the node with the lowest f cost in the open list and remove it from the 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 | | |

open list: [0]
close list: []

# II. A* ALGORITHM

## 3. Graphical explanation:

Only one node on the open list, so choose node 0 to remove and explore.
Denote node 0 as current node.

| node | status | g(n) | h(n) | f(n) | previous node |
|---|---|---|---|---|---|
| 0 (start) | current | 0 | 20 | 20 | None |
| 1 | | ∞ | 16 | | |
| 2 | | ∞ | 6 | | |
| 3 | | ∞ | 10 | | |
| 4 | | ∞ | 4 | | |
| 5 (target) | | ∞ | 0 | | |

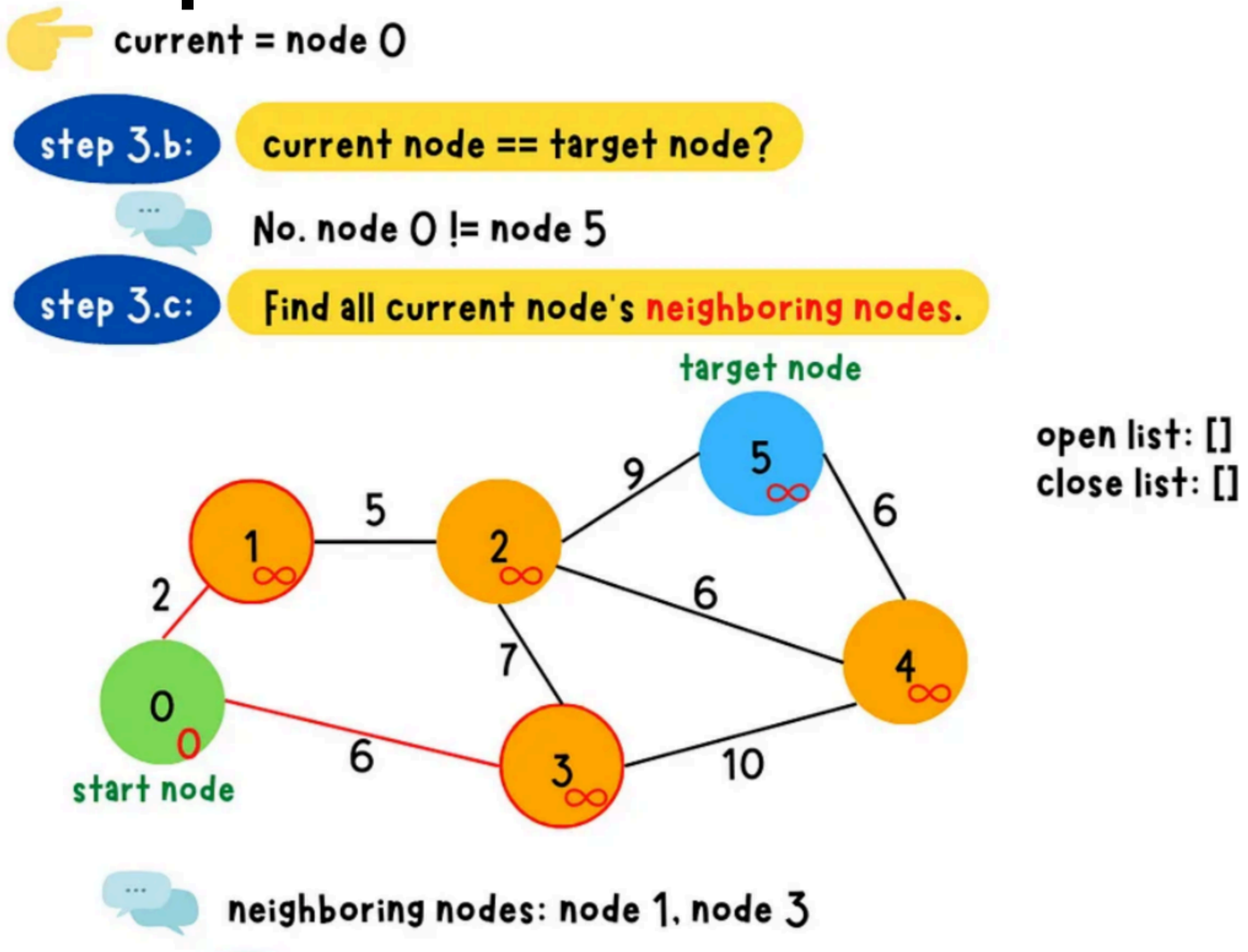
open list: []
close list: []



current = node 0

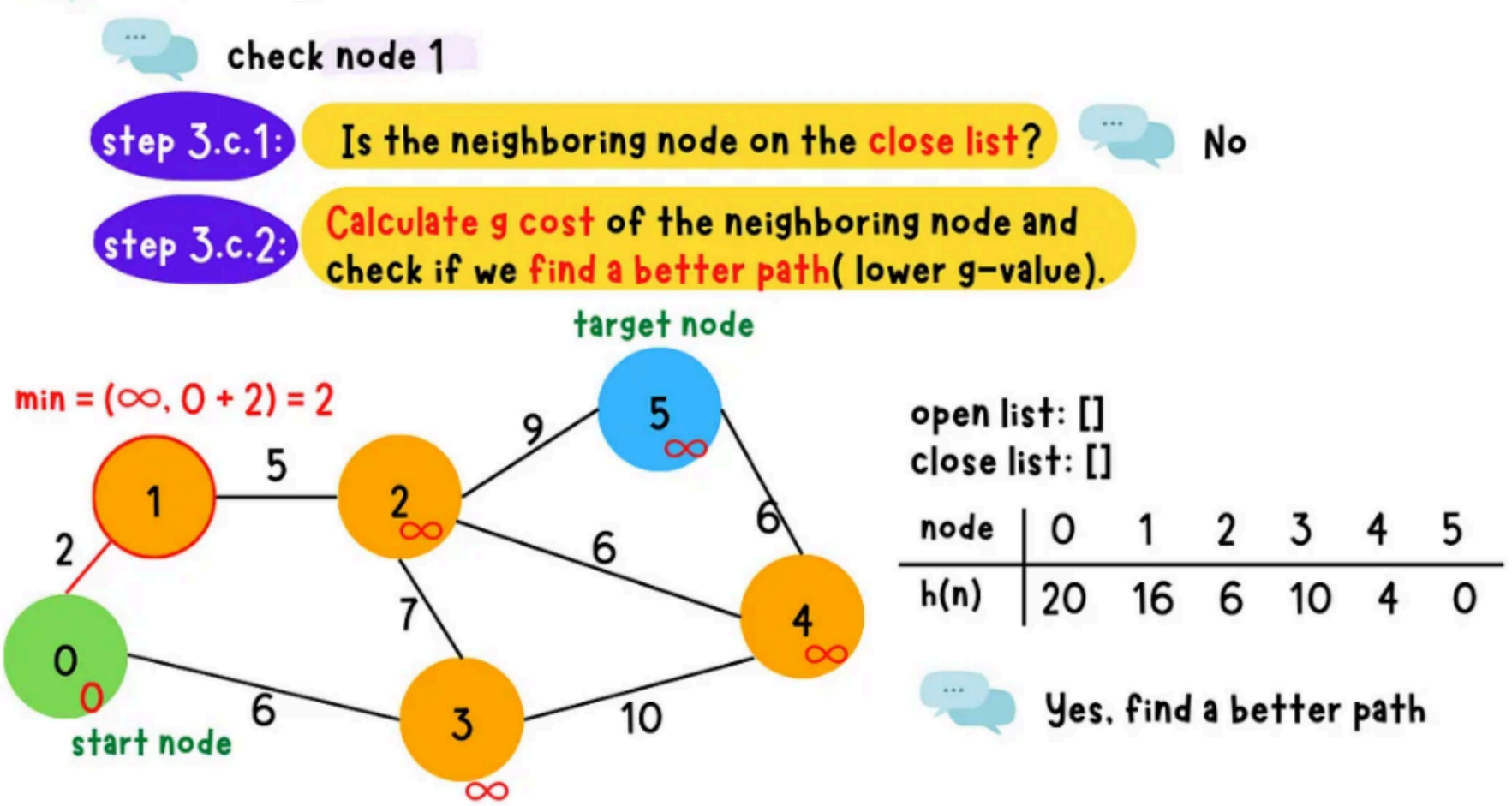
# II. A* ALGORITHM

## 3. Graphical explanation:



# II. A* ALGORITHM

## 3. Graphical explanation:



# II. A* ALGORITHM

## 3. Graphical explanation:

step 3.c.3: Update g, h and f cost of the neighboring node and store current node as previous node

| node | status | g(n) | h(n) | f(n) | previous node |
|---|---|---|---|---|---|
| 1 | | 2 | 16 | 18 | 0 |

step 3.c.4: Is the neighboring node on the open list?

No

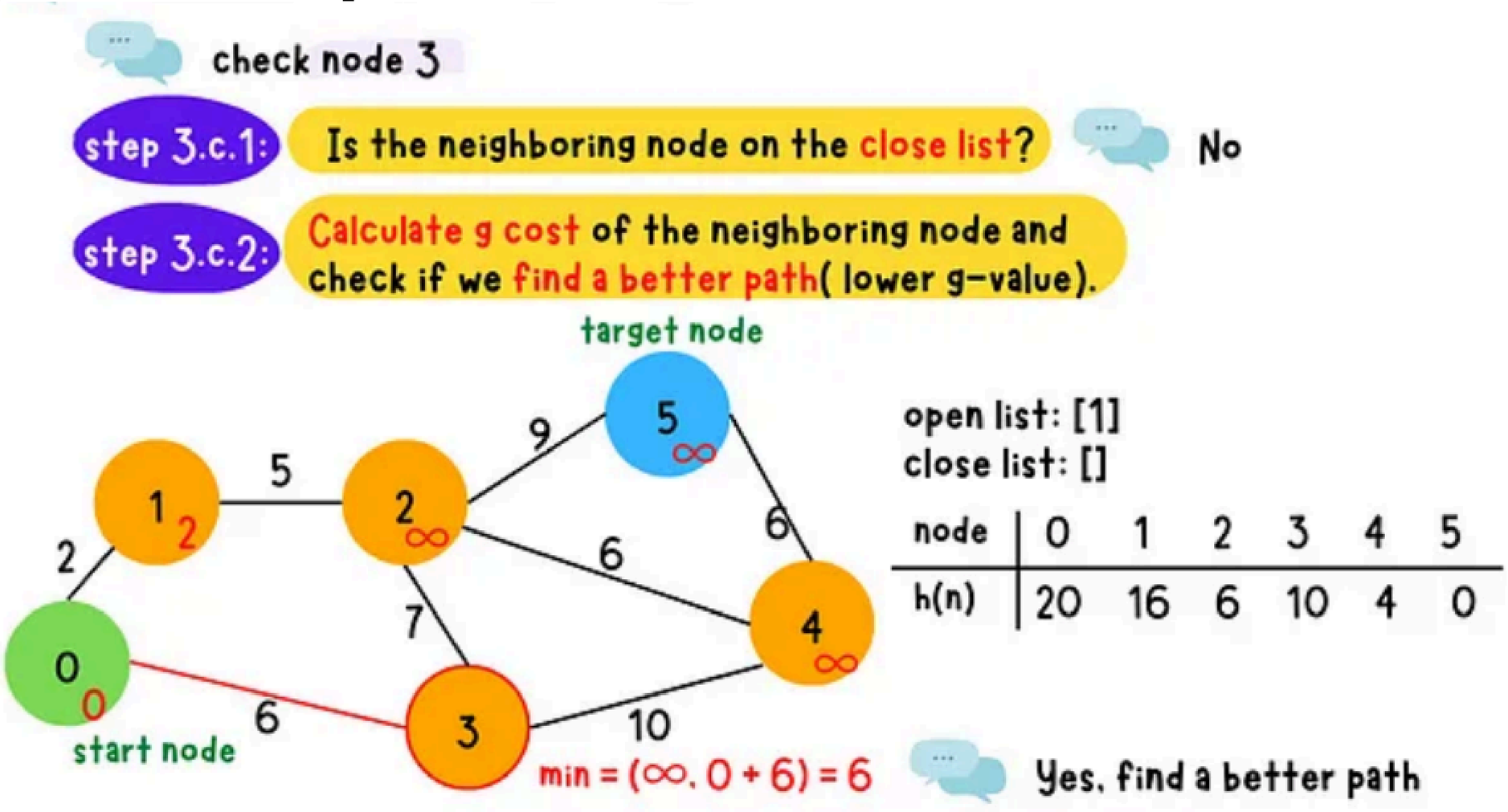
Insert the neighboring node to the open list

open list: [1]
close list: []

| node | status | g(n) | h(n) | f(n) | previous node |
|---|---|---|---|---|---|
| 1 | open | 2 | 16 | 18 | 0 |

# II. A* ALGORITHM

## 3. Graphical explanation:



# II. A* ALGORITHM

## 3. Graphical explanation:

step 3.c.3: Update g, h and f cost of the neighboring node and store current node as previous node

| node | status | g(n) | h(n) | f(n) | previous node |
|---|---|---|---|---|---|
| 3 | | 6 | 10 | 16 | 0 |

step 3.c.4: Is the neighboring node on the open list? No

Insert the neighboring node to the open list

open list: [1, 3]
close list: []

| node | status | g(n) | h(n) | f(n) | previous node |
|---|---|---|---|---|---|
| 3 | open | 6 | 10 | 16 | 0 |

# II. A* ALGORITHM

## 3. Graphical explanation:

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 | | |

# II. A* ALGORITHM

## 3. Graphical explanation:

**Step 4:**

**Repeat Step 3 until reaches the target node**

# II. A* ALGORITHM

## 3. Graphical explanation:

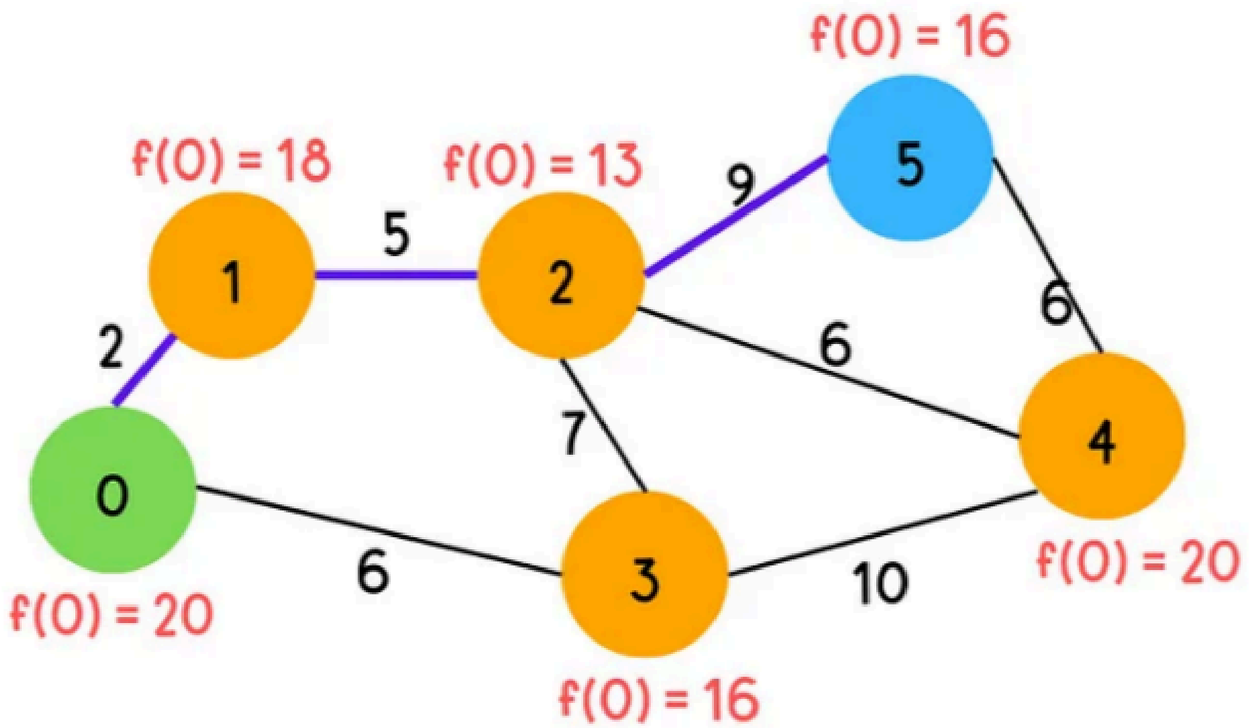
final result

| 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 |

# II. A* ALGORITHM

## 3. Graphical explanation:

The shortest path = 0 -> 1 -> 2 -> 5




backtrack to construct the shortest path

open list: [4]
close list: [0, 3, 1, 2]

| node | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| h(n) | 20 | 16 | 6 | 12 | 4 | 0 |

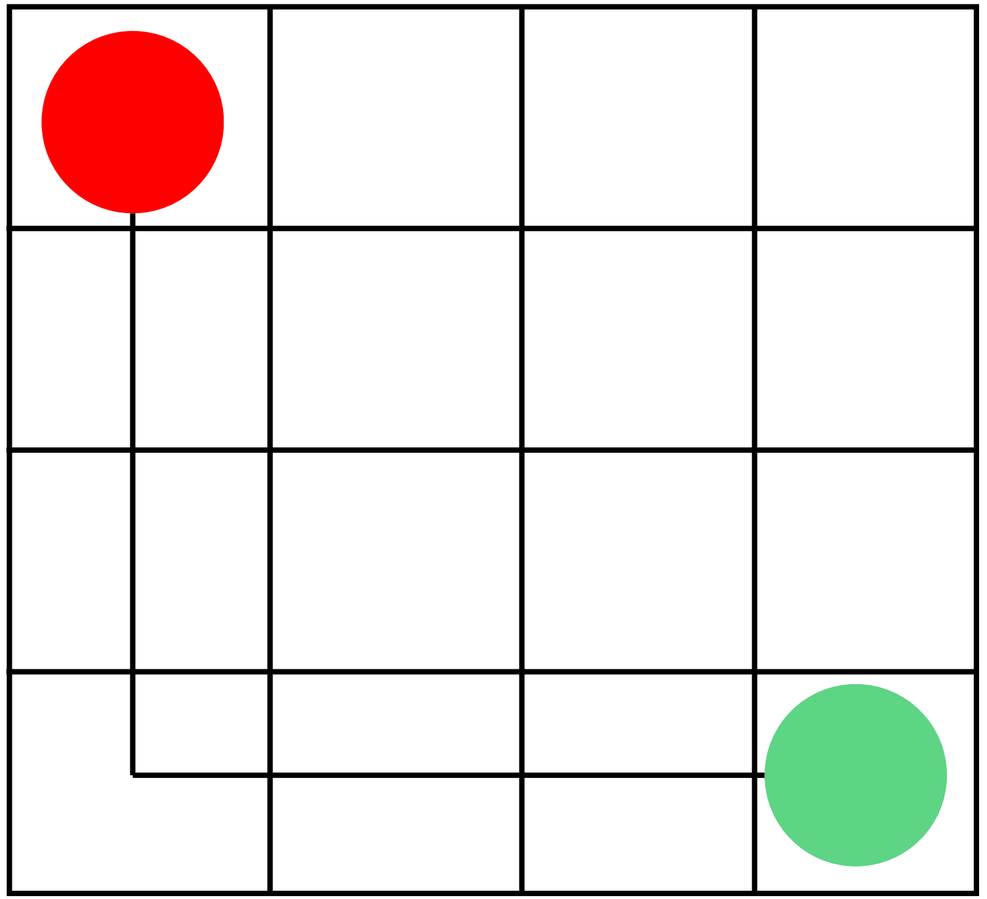
# II. A* ALGORITHM

## 4. Heuristics:

### a. Manhattan Distance

**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)



# II. A* ALGORITHM

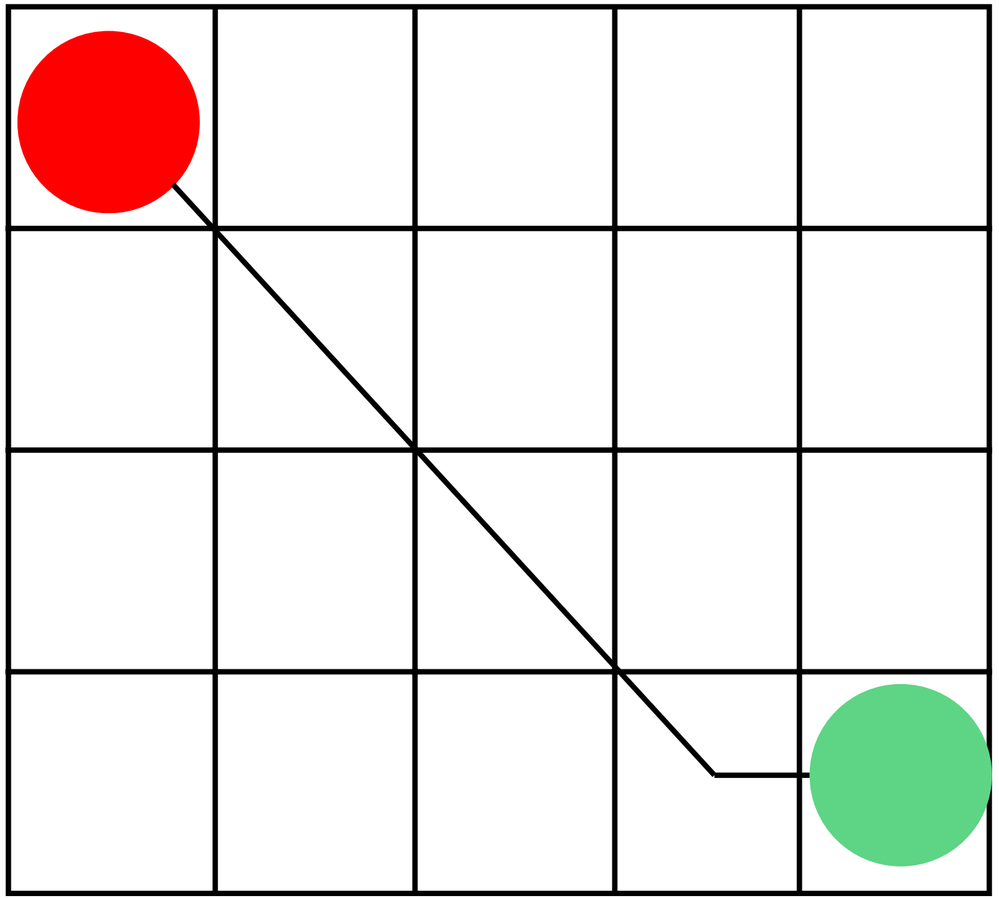
## 4. Heuristics:

### b. Diagonal Distance

**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)

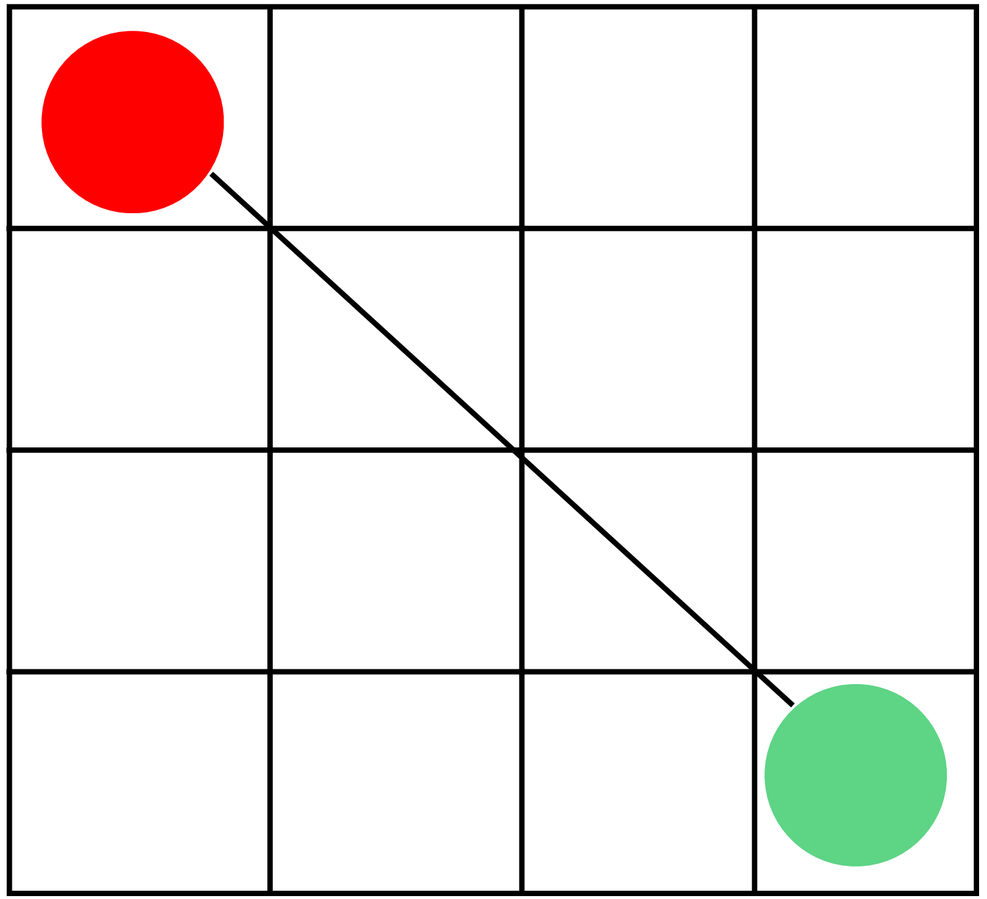
# II. A* ALGORITHM

## 4. Heuristics:

### c. Euclidean Distance

**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.**



# II. A* ALGORITHM

## 5. Time Complexity

Considering a graph, it may take us to travel all the edge to reach the destination cell from the source cell. The worst case time complexity is **O(E)**, where **E is the number of edges in the graph**.

## 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**.

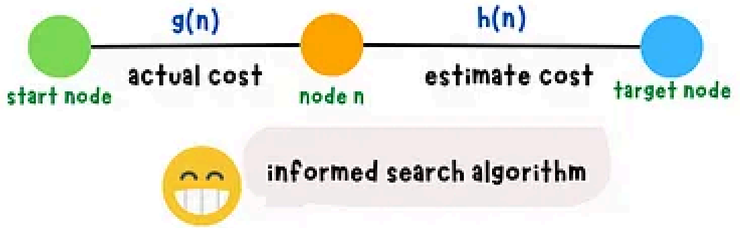
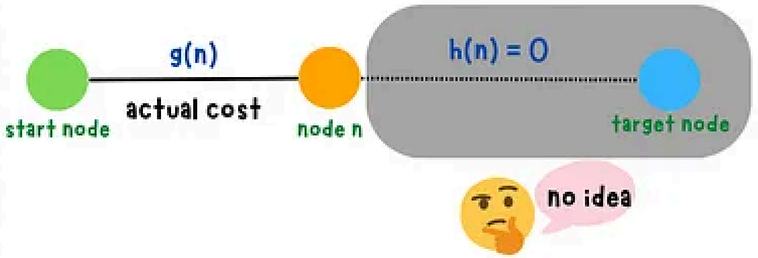
# III. REAL-LIFE APPLICATION

# III. REAL-LIFE APPLICATION

# IV. COMPARE WITH ANOTHER ALGORITHM

# IV. COMPARE WITH ANOTHER ALGORITHM

## A* Search Algorithm vs. Dijkstra's Algorithm

| A* search algorithm | Dijkstra's algorithm |
| --- | --- |
| find the shortest path from the start node to the target node in the graph.<br>a single target | find the shortest path from the start node to all other nodes in the graph.<br>multiple targets |
| f(n) = g(n) + h(n)<br>information about the target node | f(n) = g(n)<br>h(n) = 0<br>No information about the target node |
|  <br>informed search algorithm |  |
| Not expand all nodes<br>direct the path using a heuristic | expand all nodes |


Thank
you