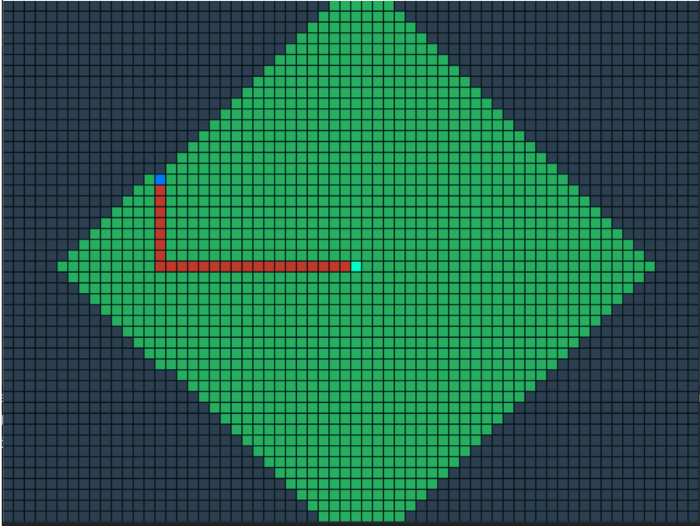
## horizontal line



# VISUALIZATION A\* ALGORITHM Using Python

24.11.2021

**─**

Tun Aung Wai

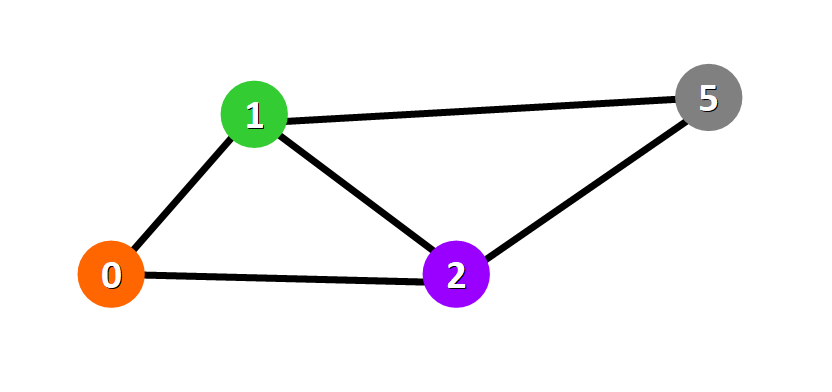
st121774

### **Basic Concepts**

Graphs are data structures used to represent "connections" between pairs of elements.

* These elements are called **nodes**. They represent real-life objects, persons, or entities.
* The connections between nodes are called **edges**.

This is a graphical representation of a graph:

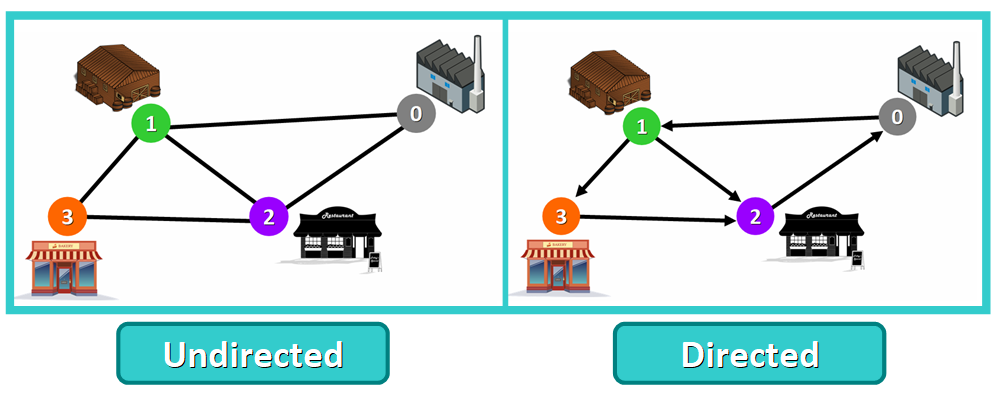


### **Applications**

Graphs are directly applicable to real-world scenarios. For example, we could use graphs to model a transportation network where nodes would represent facilities that send or receive products and edges would represent roads or paths that connect them (see below).

### **Types of Graphs**

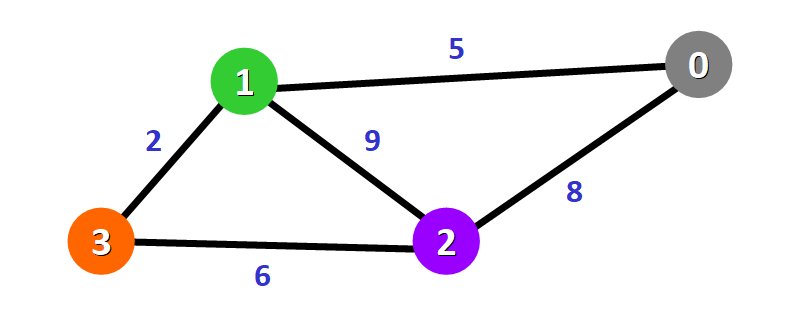
Graphs can be:

* **Undirected:** if for every pair of connected nodes, you can go from one node to the other in both directions.
* **Directed:** if for every pair of connected nodes, you can only go from one node to another in a specific direction. We use arrows instead of simple lines to represent directed edges.

### **Weighted Graphs**

A **weight graph** is a graph whose edges have a "weight" or "cost". The weight of an edge can represent distance, time, or anything that models the "connection" between the pair of nodes it connects.

For example, in the weighted graph below you can see a blue number next to each edge. This number is used to represent the weight of the corresponding edge.



Some prominent examples for the application of graphs are:

* Routing: In this case nodes represent important places (junctions, cities), while edges correspond to roads connecting these places. A one-way road is represented by a directed edge.
* Communication networks: Here, nodes are phones, modems, server, ISPs and more, edges are data lines connecting them.
* Gantt diagrams for sequence planning: Nodes correspond to phases of the projects, directed edges to dependencies between sections.
* Representation of hierarchical structures: Nodes could correspond to people or departments and edges visualize a hierarchy of command.

Types of graphs are:

* Minimum Spanning Trees
* Shortest Paths
* Matching
* Network Flow
* Routing

One of the most common applications of graphs in everyday life is the representation of infrastructure and communication networks. A street map, bus lines in a city or the flights offered by an airline; they can all be represented by a graph.

The search for paths between given nodes in these graphs is of great importance. Of special interest are ‘cost efficient’ paths, where ‘cost efficient’ can have diverse interpretations: Maybe it is the shortest or fastest way, the one that can be traversed with minimum fuel consumption, paying a minimum fare or one that avoids speed traps.

Most popular algorithms for shortest paths are

* A\*Algorithm
* Bellman-Ford Algorithm
* Dijkstra’s Algorithm
* Floyd-Warshall Algorithm

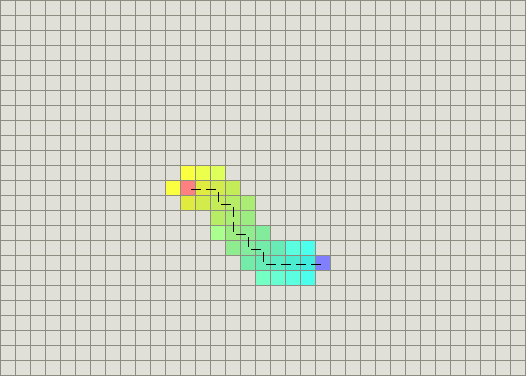
## **A\* Algorithm**

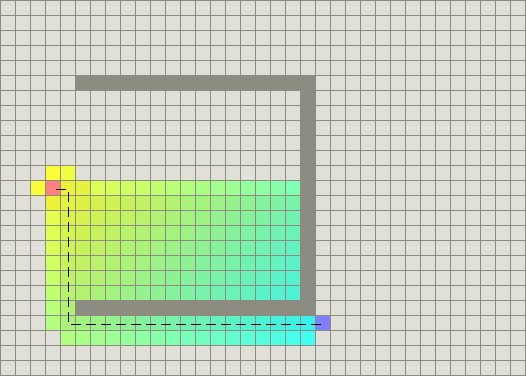
A\* is a relatively simple adjustment to Dijkstra’s algorithm, making it a [Best-First Search](https://en.wikipedia.org/wiki/Best-first_search) instead.

This works by having two scoring mechanisms for each node. One is identical to the one used in Dijkstra’s algorithm. The second is a heuristic score for how close a node is to the target node. We then use these to store two different scores against each node – the score for reaching the node on the current route, and the score for selecting a node to visit next.

This has a subtle but significant impact on the way the algorithm works. It means that we always prefer nodes that are closer to our target and avoid visiting nodes that are further away. That, in turn, makes our algorithm more efficient since we’ll be heading in the correct general direction at each step.

Interestingly, if we have a heuristic that always returns 0 then the A\* Algorithm is identical to Dijkstra’s Algorithm. This one score is the only difference between the two.

****

****

The time complexity of A\* depends on the quality of the heuristic function. In a worst-case, the algorithm can be *O(b^d)*, where *b* is the branching factor – the average number of edges from each node, and *d* is the number of nodes on the resulting path.

The better the heuristic function, the less of these nodes need to be visited, and so the complexity drops. We can describe the result of the heuristic function as the effective branching factor – the average number of edges from each node that we need to visit.

# Pseudocode

// A\* (star) Pathfinding

// Initialize both open and closed list

let the openList equal empty list of nodes

let the closedList equal empty list of nodes

// Add the start node

put the startNode on the openList (leave it's f at zero)

// Loop until you find the end

while the openList is not empty

// Get the current node

let the currentNode equal the node with the least f value

remove the currentNode from the openList

add the currentNode to the closedList

// Found the goal

if currentNode is the goal

Congratz! You've found the end! Backtrack to get path

// Generate children

let the children of the currentNode equal the adjacent nodes

for each child in the children

// Child is on the closedList

if child is in the closedList

continue to beginning of for loop

// Create the f, g, and h values

child.g = currentNode.g + distance between child and current

child.h = distance from child to end

child.f = child.g + child.h

// Child is already in openList

if child.position is in the openList's nodes positions

if the child.g is higher than the openList node's g

continue to beginning of for loop

// Add the child to the openList

add the child to the openList

