

# A\* pathfinding implementation

<https://robotics.caltech.edu/wiki/images/e/e0/Astar.pdf>  
<https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2>

## Problem statement

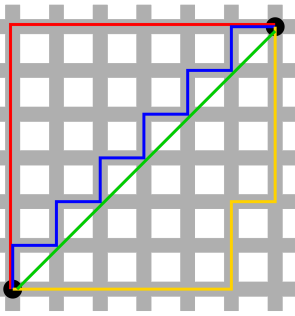
A person needs to find the quickest route between two points. There are obstacles that wholly impede movement and partially slow down movement. The intended solution will implement the *A\* algorithm* and visually display the pathfinding mechanism and obstacles through a *PyGame* display.

## Pathfinding system

### F, G, H values

Implemented via a *Map* class representing a grid of *Nodes*. A *Path* class takes the *Map* and traverses around *Nodes* labelled as obstacles to produce a path.

Each node has an *f*, *g*, and a *h* value. They are combined in  $f = g + h$ .

f	total cost of the node
g	<b>actual distance between current node and start node</b> actual traversed distance from start node <code>node_four.g = 4</code>
h	<b>taxi-cab distance (heuristic)</b> typically the distance can be solved via the Pythagorean theorem but many implementations use taxi-cab distance, which is the distance it takes to traverse blocks in a linear fashion as if you were driving in New York. The heuristic is an <i>underestimation</i> of the path. An <i>overestimation</i> would lead to the algorithm searching for poor h values.  given by $ x1 - x2  +  y1 - y2 $ 

7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	19	20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

With the  $f$  value of all neighbouring nodes to the current node, the algorithm can choose the best neighbouring node without running through all nodes.

## Open and closed lists

Each node maintains a reference to its parent (previously traversed) to 'backtrack' the path.

The open and closed sets are datasets maintained by the algorithm during runtime. The open list keeps track of nodes that need to be examined, and the closed list keeps track of nodes that have already been examined. At the start, the open set contains just the starting node, and the closed list is empty.

## Pseudocode

1. start at *start\_node*
2. add *start\_node* to open list
3. repeat:
  - a. look for lowest cost node on open list
  - b. add that node to closed list
  - c. for all neighbouring nodes:
    - i. if on closed list -> ignore
    - ii. if not traversable -> ignore
    - iii. if not on open list -> add to open list; make current square parent of that list; record  $f$ ,  $g$ , and  $h$  costs
    - iv. if on open list -> see if path to square is better via  $g$  cost; if a better route, change that nodes parent to the current node; recalculate  $g$  and  $f$  scores.
  - d. stop if:
    - i. add the target to the closed list (goal reached);
    - ii. open list is empty (no path).
4. save path; work backwards from each node's parent to reach the starting square

## Alterations

A lot of A-star implementations use weighted graphs which signal the cost of moving along an edge. My implementation is a grid of nodes which can be weighted or adjusted. The

system should still be the same though. The technique of using heuristics, open, closed sets and node weighting will produce the same output on a more expanded 2D field.

## System design

### UML Diagram

