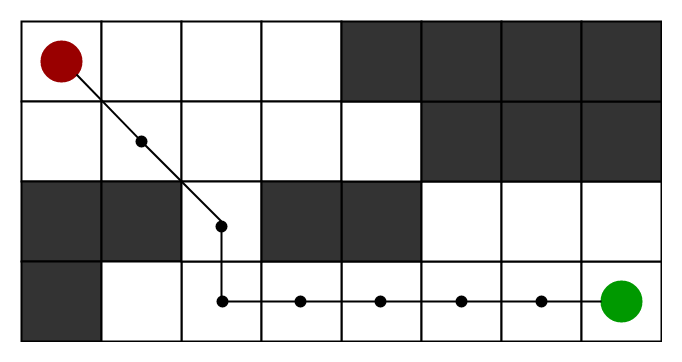
**A\* Pathfinder**Artificial Intelligent ending project, Braude College   
Lior Ohayon and Ron Yalensky (305764490, 305160715)

**Abstract**  
In this project, we demonstrate a simple Pathfinder that finds its' way from 'start' point to 'end' point, through obstacles, by A\* search algorithm. We implemented our A\* Pathfinder with javaScript, [p5js](https://p5js.org/) (a javaScript library) and HTML.

**A\* Search algorithm**

To approximate the shortest path in real-life situations, like in maps, games where there can be many obstacles.

We can consider a two-dimensional grid having several obstacles and we start from a source cell (colored red below) to reach towards a goal cell (colored green)



A\* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.

A\* Search algorithm, unlike other traversal techniques, it has a "brain". What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. It is also worth mentioning that many games and web-based maps use this algorithm to fine the shortest path very efficiently.

Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell 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 **f** which is a parameter equal to the sum of two other parameters – **g** and **h**. At each step it picks the node having the lowest **f** and move to that node.

Definition of **g** and **h** as simply as possible:  
**g** – the movement cost to move (also, the known distance) from the starting point to a given node on the grid, following the path generated to get there.  
**h** – the heuristic rule, the estimated movement cost to move from the given square on the grid to the final destination. This is often referred to a "smart guess". There can be many ways to calculate **h**, a common way to calculate it is by calculating the distance from the start to the end.

Pseudo-Code to describe the algorithm. Our A\* Pathfinder algorithm is very inspired by the pseudo-code below.

**function** A\*(start, end)

closedSet := {} *// The set of nodes already evaluated*

*// The set of currently discovered nodes that are* ***not evaluated yet****.*

openSet := {start} *// Initially, only the start node is known.*

cameFrom := an empty map  *//cameFrom will eventually contain the* *most efficient previous step.*

*// For each node g =: the cost of getting from start to that node.*

*// The cost of going from start to start is zero.*

gScore[start] := 0

*// For the first node, that value is completely heuristic.*

fScore[start] := gScore[start] + heuristic(start, end)

**while** openSet **is** **not** empty

current := *min*(fScore[]) in openSet

**if** current = goal

**return** *reconstruct*\_*path*(cameFrom, current)

openSet.*Remove*(current)

closedSet.*Add*(current)

**for** each neighbor **of** current

**if** neighbor **in** closedSet

**continue** *// Ignore the neighbor which is already evaluated.*

*// The distance from start to a neighbor*

temp\_gScore := gScore[current] + *dist\_between*(current, neighbor)

*//in Pathfinder, the distance between current and neighbor = 1*

**if** neighbor **not** **in** openSet *// Discover a new node*

openSet.*Add*(neighbor)

**else** **if** temp\_gScore >= gScore[neighbor]

**continue**

*// This path is the best until now. Record it!*

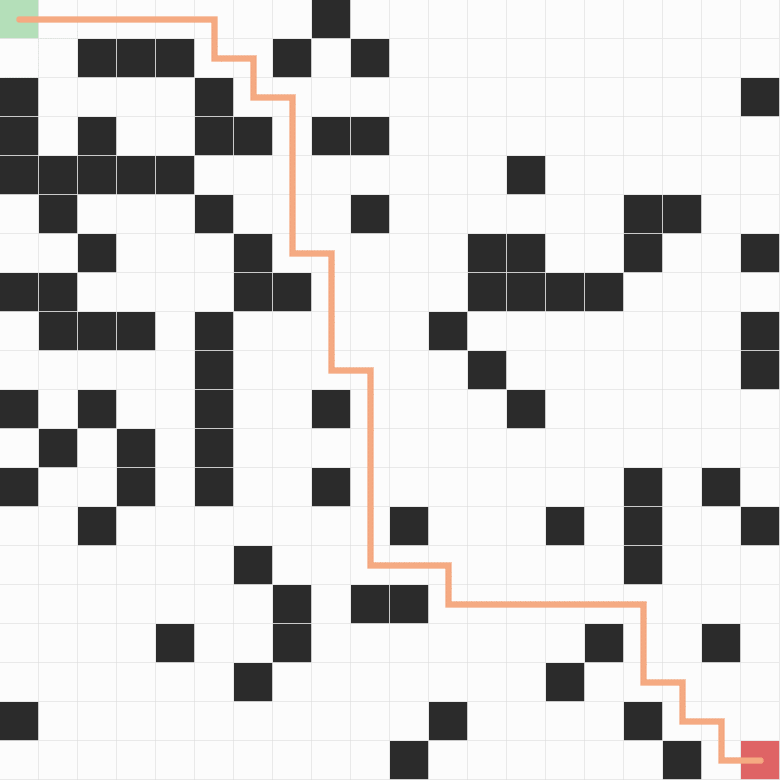
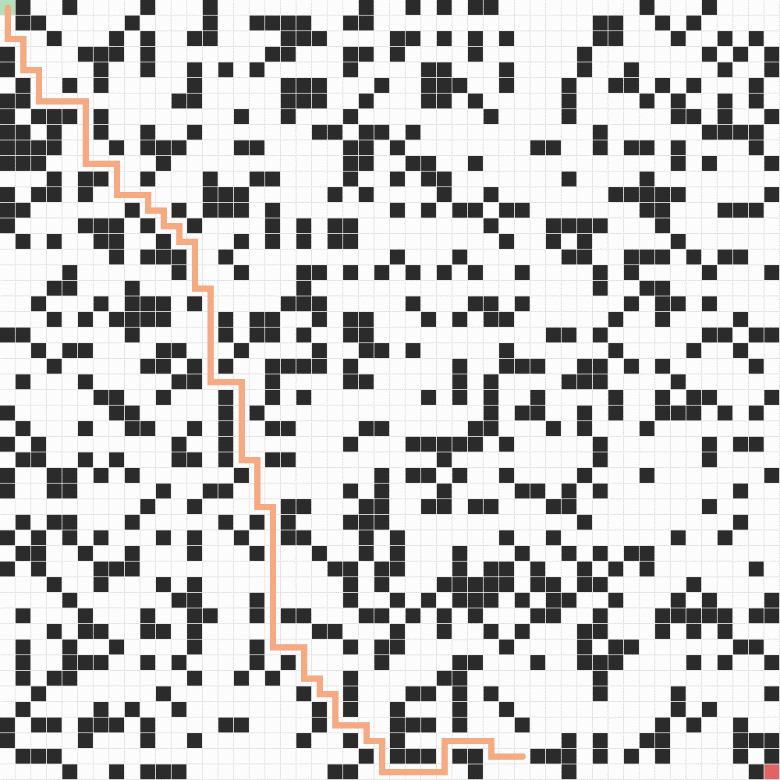
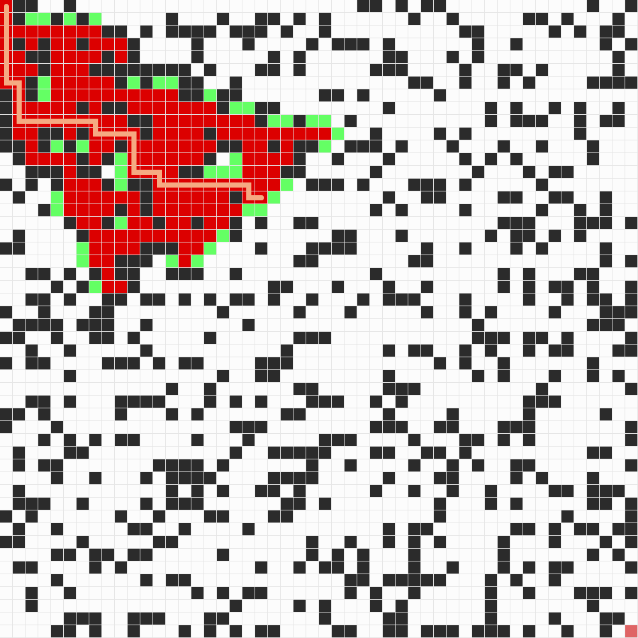
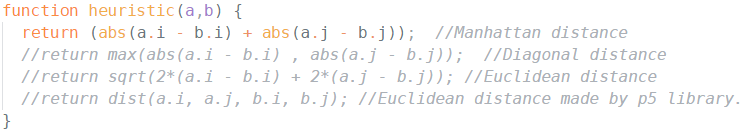
cameFrom[neighbor] := current

gScore[neighbor] := temp\_gScore

fScore[neighbor] := gScore[neighbor] + heuristic(neighbor, goal)

**Instructions**

* **Obstacles Density** – Adjust the obstacles density, The chance is from 0 to 1, when 0 is no obstacles, and 1 is full board of obstacles.
* **Frame Rate –** Adjust the frame rate of the board. The frame rate specifies the number of frames to be displayed every second. For example, the function call frameRate(30) will attempt to refresh 30 times a second.
* Select **Size of Board –** Choose the size of the board from the options given.
* **Debug Mode –** Check Debug Mode to see the decisions through the path calculations. As shown in Fig 3.
* **Submit –** Start the program.
* **Refresh –** Refresh the page and start from the beginning.

**Algorithm**  
Let's start with an output examples of the algorithm.  
  
  
**Fig 1.** In this example, the grid size is 20x20, the *start* node is in the top left, the *end* node is in the bottom right. The shortest path is orange colored. Obstacles are randomly putted in the grid by chance of 20%.  
  
 **Fig 2.** In this example, the grid size is 50x50. Obstacles are randomly putted in the grid by chance of 30%. As you can see, the *end* node cannot be reached. In this case there is no solution.   
  
   
**Fig 3.** For debugging, we colored the nodes that we visited (closedSet members) in red and the nodes that need to be evaluated (openSet members) in green. The coloring is done for every "frame" in the draw function.  
  
In this implementation, we wanted to demonstrate the calculation of the path through its way, by that, you can watch every "frame" of calculation (draw() function), thanks to the scripting language javaScript and p5js library.  
  
In our code, we used the *heuristic* function that returns the distance between the *start* node and the *end* node. The distance of those two nodes can be calculated by variant of common distance functions. In our code, we wrote four options to calculate the distance (Manhattan, Diagonal and Euclidean distances). As you can see below.  
  
  
When i,j describes the x,y value of a and b respectively. abs() refer to the absolute value, and sqrt() refers to the square root. Here Manhattan distance is running, and the three other options are in comment.

**Data structure**  
The data structure of this implementation described below.

**Grid**  
The grid in this algorithm is a simple 2-dimensional array of Nodes. At the setup section we initialize the grid.  
*gridi,j = new Node(i, j)*, for each pair of i, j in the grid i is incrementing from 0 to the number of columns and j from 0 to number of rows in the grid.

**Node**  
Each Node has the followings:  
f, g and h values for the A\* algorithm, x and y values (i and j) ,neighbors array, previous node to map the path, Boolean variable isObstacle to define if the node obstacle or not (Obstacle nodes are randomly picked), show(color) function to display the node (a square with a color) and a getNeighbors function.

The implementation code (filename: AStarPathfinder.js) is well documented, so it's easy to understand each step.

\* In this project we used a few sources on the internet - Wikipedia entry of A\* search algorithm, GeekForGeeks website, p5js and youtube videos.