The algorithm operates on a grid of nodes. Each node contains a number of attributes: the *block* stored at that location (which can be a wall), a list of paths that contain that node, the time interval the object spent at that location, the cost(f), the total distance from *source* (g) using the current path, and the estimated distance to *destination* (h), and the parent node (which is based on position).

1. We use two priority queues: *open list*, which holds a list of nodes which need to be considered, and *closed list*, which holds a list of nodes that have already been considered. Note that unlike Dijkstra’s Algorithm, *open list* is initially empty. We initialize *source* as follows: g is 0 (because the distance from a node to itself is 0), h is abs(*dest.x – source.x)* + abs(*dest.y – source.y*) and f is the sum of the two. The parent of *source* is set to itself. *source* is then added to *open list.*
2. If *open list* is empty, return; there is no path from *source* to *dest*. Otherwise, set *node* to be the node in *open list* with the lowest f value. We want to choose the most promising node. Switch this node to *closed list*. If we added *dest* to the closed list, return; we have found the path.
3. For each node, *child*, that is surrounding *node* and is not a wall and is not in *closed list*,
   1. Calculate the new f, g, and h values for child, but do not store them yet. Note that if we move horizontally or vertically, we add 10 to g, but if we move diagonally, we add 14 (which is ~ 10√2). *time* is set to *parent*.*time*+1
   2. If any path contains this node, check that there is no collision (i.e. they do not meet there at the same time). If they do, ignore the node for now, though it can be reconsidered at a later time when there will be no collision. Continue from step 3. Otherwise, continue to part c.
   3. Add *child* to the *open list*
   4. If *child* is in *open list*, set f, g, and h for *child*. Otherwise, compare the current g value with the new g value. If the new g value is less, set f, g, and h to be the new values. Set the parent of *child* to be *node*.
4. Repeat steps 2-4 until it returns. If there is no path, skip the remaining steps.
5. Now we need to recover the path and update the grid. Start with *dest*. For each node, add it to the list of nodes in the path. To get the next node, look at its parent; continue this until you’ve reached *source*. Next, reverse the list of points, since they’re in reverse order.
6. To update the grid, consider each point *p* along the path. In the list of paths going through the node at point *p*, add the path ID and time that the object reaches that point. This should be 0 for the first point, 1 for the second point, etc.