

Drone control algorithm

The

Algorithms

Input: the start coordinate position of the drone c , and G , a weighted graph of the sensors and their distances to each other, taking into account obstacle evasion.

Output: a list of moves M that visits all sensors and return to the starting location

Function `createPlan(c , G)`

```
|  $T \leftarrow \text{twoOptTSP}(G)$   
|  $T \leftarrow \text{twoOptImprove}(tour)$   
| return constructFlightAlongTour( $T$ )
```

Input: a list of points T starting and ending with the drone start position with the rest containing a tour of all of the sensors

Function `constructFlightAlongTour(T)`

```
|  $c \leftarrow T_0$   
|  $M \leftarrow []$   
| for  $i \in [1, \text{size}(T) - 1]$  do  
|    $t \leftarrow T_i$   
|   if  $i < \text{size}(T) - 1$  then  
|     |  $t \leftarrow \text{cutCorner}(c, T_i, T_{i+1})$   
|      $W \leftarrow \text{getPathBetweenPoints}(c, t)$   
|      $M.\text{add}(\text{navigateAlongWaypoints}(c, W, 1, \text{maxMoveLength}(c, W_1)))$  // start at  
|       1 since the first is the sensor we have already reached  
|      $c \leftarrow \text{last}(M)$  // update current location to the end of the latest move  
| return  $M$ 
```

Algorithm 1: Creates a flight plan for the drone which visits all sensors and returns to the start. It first uses a two stage 2-opt heuristic on a pre-generated sensor graph to generate a tour. It then scans through the tour, constructing a flight plan along a list of waypoints which pathfind around any obstacles using the drone movement rules. Before each step of generating the flight plan, it attempts to “cut the corner”, taking advantage of the fact that sensors have a 0.0002 range to shorten the route. In the implementation there are many bonus features: the algorithm is run a large number of times, and the shortest flight plan is chosen. Depending on the options (default is timed), this continues for a fixed number of iterations or a set amount of time (which can be input as an extra command line argument), after which it stops and chooses the best. Panellisation is also used.

Input: the current coordinate position c , a list of waypoints W , the last of which is the target sensor or end position, the current waypoint number i , the number of moves remaining until we declare a potential path invalid t

Output: a list of moves which move the drone along the waypoints into range of the targets, in reverse order

```

 $D \leftarrow ()$  // the set of points that have been examined so far
Function nagigateAlongWaypoints( $c, W, i, t$ )
| if  $t = 0$  then
| | return null
| foreach  $offset \in [0, 10, -10, \dots, 170, -170]$  do
| |  $\theta \leftarrow$  offset + direction from  $c$  to  $W_i$  rounded to the nearest  $10^\circ$ 
| |  $d \leftarrow$  position after moving 0.0003 in the direction  $\theta$ 
| | if  $d \in D$  then
| | | continue // try the next offset
| |  $D.add(d)$ 
| | if the line  $cd$  collides with an obstacle then
| | | continue // try the next offset
| | if  $W_i$  is not the last waypoint and  $d$  has line of sight to  $W_{i+1}$  then
| | |  $M \leftarrow$  nagigateAlongWaypoints( $d, W, i + 1, \text{maxMoveLength}(d, W_{i+1})$ )
| | | if  $M$  is not null then
| | | | return  $M.add(\text{a move from } c \text{ to } d \text{ in direction } \theta)$ 
| | | else
| | | | continue // try the next offset
| | if  $d$  is within 0.0002 of the target sensor or 0.0003 of the end position then
| | | return  $[Move(c, d, \theta, \text{sensor if it is not the end})]$ 
| |  $M \leftarrow$  nagigateAlongWaypoints( $d, W, i + 1, t - 1$ )
| | if  $M$  is not null then
| | | return  $M.add(\text{a move from } c \text{ to } d \text{ in direction } \theta)$ 
| return null // if none of the offsets worked
Function maxMoveLength( $c, w$ ) // auxiliary function
| return  $2 + \lceil cw \rceil / 0.0003$ 

```

Algorithm 2: Recursively finds moves which navigate from waypoint to waypoint, until the target location is reached. The main idea of this algorithm is to move in the direction of a waypoint until we reach it, and then move to the next waypoint, until we reach the target sensor or end position. If the target is an intermediate waypoint located on the corner of a no-fly zone, we determine that we have reached it not with but a range but when the next waypoint is in sight, since the waypoints exist to find paths around obstacles. We also keep track of which points we have examined so far to avoid looping, and stop examining a branch if it takes far more moves than expected, because it went the wrong way or got stuck somehow. If the move we tried hits an obstacle or gets stuck further down the recursive call, we try a different offset from the direction until we find one that works. In most cases, which do not need to avoid any obstacles, this algorithm will progress directly towards the target and terminate very quickly.

Examples

what, why?



Figure 1: 01/01/2020