Drone control algorithm

The two main complicating factors in the design of the algorithm are the existence of the no-fly zones, and the constraints on the drone move length direction. Individually, these do not pose much of a problem, but it is together that they combine to create significant complexity, especially when not making assumptions. This algorithm attempts to capture the complexity of the situation, and any edge cases that may arise, and to aim for short flight paths by cutting corners literally, but not figuratively.

Algorithms

Input: the start coordinate position of the drone c, and L a list of sensor locations
Output: a list of moves M that visits all sensors and return to the starting location
Function `createPlan(`*c, L*`)`
  $F \leftarrow$ `createSensorGraph(`*c, L*`)`
  T $\leftarrow$ `twoOptTSP(`G`)`
  T $\leftarrow$ `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 $i \in [1, size(T) - 1]$ **do**
    $t \leftarrow T_i$
    if $i < size(T) - 1$ **then**
      $t \leftarrow$ `cutCorner(`*c, T_i, T_{i+1}*`)`
    $W \leftarrow$ `getPathBetweenPoints(`*c, t*`)`
    M.add(`nagigateAlongWaypoints(`*c, W, 1,* `maxMoveLength(`*c,* W_1`)))` `// start at 1 since the first is the sensor we have already reached, see algorithm 5`
    c $\leftarrow$ last(M) `// update current location to the end of the latest move`
  return M

Input: coordinate points c, t, n
Function `cutCorner(`*c, t, n*`)`
  $\theta \leftarrow$ the direction which bisects the angle between lines tn and tc `// new direction`
  $p \leftarrow$ point $0.634 * 0.0002$ degrees away from t in direction θ `// new target`
  $min \leftarrow$ distance ctn
  if $cpn < min$ ***and not*** `lineCollision(`cpn`)` **then**
    return p
  return t

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. The constant ratio 0.634 was chosen as it performed the best in testing: it is a trade-off between cutting more of the corner and being too close to the outside of the range and missing or overshooting (due to the constraints on drone movement). In the implementation there are many bonus features: the algorithm is run a large number of times, in parallel if possible, and the shortest flight plan is chosen. Depending on the options, this continues for a fixed number of iterations or a set amount of time (which can be input as an extra command line argument, default is timed for 1 second), after which it stops and chooses the best.

Input: the start coordinate position of the drone c, and L a list of sensor locations
Output: a graph G with vertices consisting of the start position and all sensors, with edge weight determined by the distance of the shortest path
Function `createSensorGraph(`*c, L*`)`
  $G \leftarrow$ a new weighted graph
  G.addVertices($c + L$)
  foreach *pair of vertices (a, b)* **do**
    G.addEdge((a,b), weight (euclidean distance) of `getPathBetweenPoints(`*a, b*`)`)

Algorithm 2: This simple algorithm creates a graph for use by the TSP algorithms. Instead of using the euclidean distance between points in a straight line, it uses the obstacle pathfinder to calculate the length of the shortest route between them while also going around obstacles.

Input: a list of obstacles represented as polygons
Output: a graph G with vertices consisting of the points surrounding all of the obstacles, and edges wherever they have line of sight with euclidean distance weight
Function `prepareObstacleGraph(`*O*`)`
  $L \leftarrow$ [ ]
  foreach $p \in P$ **do**
    $L \leftarrow$the points of a polygon which forms a very narrow outline around P
  $G \leftarrow$ a new weighted graph
  foreach *pair of vertices (a, b)* **do**
    if *not* `lineCollision(`*ab*`)` **then**
      G.addEdge((a,b), euclidean distance between a and b)

Algorithm 3: Prepare a graph with all points which form an outline around the polygons as vertices, and edges if they have line of sight, which have a weight equal to the distance between them. The graph uses points from outlines of the obstacles since if it used the originals, points on the vertices would occupy the same location and the collision detection would consider any line emerging from the corner of an obstacle to be colliding with it.

Input: coordinate points a and b
Output: a path P of points which is the shortest path from a to b on graph O
Data: The obstacle graph O, as prepared by algorithm 3
Function `getPathBetweenPoints(`*a, b*`)`
  if *not* `lineCollision(`*ab*`)` **then**
    return *path direct from a to b*
  $L \leftarrow$ [ ]
  foreach $p \in P$ **do**
    $L \leftarrow$the points of a polygon which forms a very narrow outline around P

Algorithm 4: a

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* $\in [0, 10, -10, ..., 170, -170]$ **do**
  - $\theta \leftarrow$ offset + direction from c to W_i rounded to the nearest 10°
  - $d \leftarrow$ position after moving 0.0003 in the direction θ
  - **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, $i + 1$, `maxMoveLength` (d, W_{i+1}))
    - **if** M *is not null* **then**
      - **return** M*.add(* *a move from c to d in direction* θ)
    - **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,* θ*, 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(* *a move from c to d in direction* θ)
- **return** *null* `// if none of the offsets worked`

Function `maxMoveLength(`*c, w*`)` `// auxiliary function`

- **return** $2 + \lceil cw \rceil / 0.0003$

Algorithm 5: 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