Assignment 2: Path planning

ETH Zurich

Individual Demonstration: Monday, 27.10.2014 at 16:15
Individual Software Due: Monday, 27.10.2014 at 23:00
Group Work Demonstration: Monday, 03.11.2014 at 16:15
Group Work Software Due: Monday, 03.11.2014 at 23:00

1 Path planning

1.1 Background

Path planning is the process of producing continuous motion for a robot such that it can move from a start configuration and a goal configuration while avoiding collision with known obstacles. Path planning often represents these obstacles in a 2D or 3D map and describes the path in robot's configuration space. Configuration space is a set of all possible configurations, i.e., the robot's degrees of freedom. As most robots are nonholonomic, there is a limit to the robot's velocity in each configuration. To simplify this problem, in mobile robot path planning with a differential drive robot, we often assume that the robot is holonomic. The configuration space is then identical to 2D projection of physical space. In addition, we inflate the configuration space by the robot's radius and assume that the robot is a point. These modifications allow path planning algorithms to simply search for a path for point mass between various points in the free space.

One way to search for a path is by applying a graph search algorithm, which requires the configuration space to be converted into a graph. A popular method for converting the space is decomposing it into grid cells. Each cell is then a node in the graph, and two neighboring cells form an edge in the graph. The connectivity between two neighbouring cells can be either four or eight. In four-connected graph, a node forms an edge with the node's top, down, left, and right neighboring nodes; in eight-connected graph, the node forms additional four edges to its diagonal neighbors.

A* search algorithm [1] is a best-first search algorithm that finds a lowest-cost path from a starting node to a goal node in a graph. It explores a path with the lowest expected total cost first using a priority queue. The expected total cost $f(n)$ of a node n is computed by

$$f(n) = g(n) + \epsilon * h(n), \tag{1}$$

where $g(n)$ is the path cost to n from the starting node, $h(n)$ is the heuristic cost to the goal from n, and ϵ determines the weight between $g(n)$ and $h(n)$. Heuristic cost $h(n)$ is often computed as the Euclidean distance between n and the goal node, and the optimal path is found when $\epsilon = 1$. The path cost $g(n)$ is

$$g(n) = g(n') + c(n, n'), \tag{2}$$

where $g(n')$ is the path cost of n's neighboring node n' and $c(n, n')$ is the traversal cost of getting from n' to n. The traversal cost $c(n, n')$ can be computed by the Euclidean distance between two nodes n and n' or approximated by the Manhattan distance.

Using the expected cost $f(n)$, A* algorithm, shown in Algorithm 1, works as follows: Given a starting node n_{start} and a goal node n_{goal}, the algorithm initializes the search by setting

$g(n_{start}) = 0$ and adding the neighboring nodes of n_{start} to a heap S_{open}. S_{open} contains all the neighboring nodes of the visited nodes S_{closed}. Among the nodes in S_{open}, the algorithm searches for the next node n_{next} with the smallest expected cost $f_{cost}(n_{next})$ to the goal and repeatedly adds the neighboring nodes of n_{next} to S_{open}. When the next node n_{next} is the goal node n_{goal} or all the reachable nodes S_{open} in the space have been searched, the algorithm terminates its search. If the algorithm reaches the goal node n_{goal}, we can build an optimal path $P_{optimal}$ by trailing back the previous nodes until we reach the starting node.

1.2 Task

Write A* search algorithm in as a ROS node and control the robot to go from a starting point to an end point in Roboscoop.

2 Practical help

For this assignment, you need to download the data directory and install `map_server`. You can find the installation instruction here: http://www.roboscoop.org/docs.

2.1 Reading a map and publish a path in ROS

`map_server` publishes `map` topic. To read it in a ROS node, the node must subscribe to `map` of type `nav_msgs/OccupancyGrid`. To publish a path and visualize it in RViz, the path must be of type `nav_msgs/Path`.

2.2 Using TF

`tf` is a package that lets the user keep track of multiple coordinate frames over time. This package will be useful when transforming points between the `map` coordinate frame and the `odometry_link` coordinate frame.

3 Grading

3.1 In-class demonstration (20 points)

You will be given a map as a png file and some points in the map. The task is for the robot to plan a path that visits one or more of these points.

3.1.1 Individual In-class demonstration (10 points): Monday, 27.10.2014 at 16:15

For the individual portion, you will publish a path between two or more points as `nav_msgs/Path` and display the path in RViz. If no path exists, you must indicate that no path exists.

- Find a path between two points
  - 4-connected path, point-mass robot: 2 points
  - 8-connected path, point-mass robot: 2 points
  - 4-connected path, robot of radius $6cm$: 2 points
  - 8-connected path, robot of radius $6cm$: 2 points
- Find a path between four points: 2 points
  - Connectivity and robot's radius will be given to you on the day.

3.1.2 Group In-class demonstration (10 points): Monday, 03.11.2014 at 16:15

For the group portion, you will demonstrate how your robot can go from a starting location to a goal location without bumping into obstacles. In addition to known obstacles given to you as a map, the testing environment will contain one or two unknown obstacles.

- Accuracy (4.5 points)
  - Within 2*cm* of the most accurate robot: 4.5 points
  - Every 2*cm* thereafter: -0.5 point
- Speed (4.5 points)
  - Within 30 seconds of the fastest robot: 4.5 points
  - Every 30 seconds thereafter: -0.5 point
- No bumping (1 point)
- You will get 2 tries. Every extra try will cost 1 point.

3.2 Software quality (20 points)

On the due date at 23:00, we will collect your code through your SVN repository. Every file that should be considered for grading must be in the repository at that time. Note that EIFGENs folder in your project contains auxiliary files and binaries after compilation. Please, DO NOT include EIFGENs folder into your svn repository.

3.2.1 Individual evaluation (10 points): Monday, 27.10.2014 at 23:00

- Choice of abstraction and relations (3 points)
- Correctness of implementation (4 points)
- Extendibility and reusability (2 points)
- Comments and documentation, including "README" (1 points)

3.2.2 Group evaluation (10 points): Monday, 03.11.2014 at 23:00

- Choice of abstraction and relations (3 points)
- Correctness of implementation (4 points)
- Extendibility and reusability (2 points)
- Comments and documentation, including "README" (1 points)

References

[1] Hart, P. E., Nilsson, N. J., Raphael, B. 1968. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". IEEE Transactions on Systems Science and Cybernetics SSC4 4 (2): 100107.

Algorithm 1: A* search algorithm [1]

```
Data: $n_{start}$ – A starting node
      $n_{goal}$ – A goal node
      $G := (V, E)$ – A graph of nodes $V$ and edges $E$
Result: $P_{optimal}$ – An optimal path from $n_{start}$ and $n_{goal}$
begin
    $S_{closed} := \emptyset$ – A set of visited nodes
    $S_{open} := \{n_{start}\}$ – A priority queue of nodes to visit
    $g_{cost}(n_{start}) := 0$ – Path cost
    $f_{cost}(n_{start}) := g_{cost}(n_{start}) + h_{cost}(n_{start}, n_{goal})$ – Expected total cost
    $n_{start}^{previous} := \varnothing$ – Previous node from which $n_{start}$ comes
    $P_{optimal} := \emptyset$
    while $S_{open} \neq \emptyset$ or $n_{goal} \notin S_{closed}$ do
        $\{n_{next} \mid n_{next} \in S_{open},\ f_{cost}(n_{next}) \leq f_{cost}(n)\ \forall n \in S_{open}\}$
        – Get a node with the lowest expected cost.
        $S_{open} := S_{open} - \{n_{next}\}$
        $S_{closed} := S_{closed} + \{n_{next}\}$
        if $n_{next} = n_{goal}$ then
            – If the node is at goal, build the path.
            $n_{current} := n_{goal}$
            while $n_{current} \neq \varnothing$ do
                $P_{optimal} := \{n_{current}\} + P_{optimal}$
                $n_{current} := n_{current}^{previous}$
        else
            – If the node is not at goal, update its neighbors' expected cost.
            forall $\{n_{neighbor} \in V \mid n_{neighbor} \notin S_{closed}, \exists(n_{next}, n_{neighbor}) \in E\}$ do
                – Go through all neighboring nodes of the node.
                if $n_{neighbor} \notin S_{open}$ then
                    – If the neighboring node is not in the priority queue, update its
                    expected cost, set its previous to the node, and add it to the queue.
                    $g_{cost}(n_{neighbor}) := g_{cost}(n_{neighbor}) + d(n_{neighbor}, n_{next})$
                    $f_{cost}(n_{neighbor}) := g_{cost}(n_{neighbor}) + h_{cost}(n_{neighbor}, n_{goal})$
                    $n_{neighbor}^{previous} := n_{next}$
                    $S_{open} := S_{open} + \{n_{next}\}$
                else
                    – If the neighboring node is in the priority queue, compute its path
                    cost.
                    $g_{cost,temp}(n_{neighbor}) := g_{cost}(n_{next}) + d(n_{neighbor}, n_{next})$
                    if $g_{cost,temp}(n_{neighbor}) < g_{cost}(n_{neighbor})$ then
                        – If the new cost is smaller, then update the expected cost and set
                        its previous to the node.
                        $g_{cost}(n_{neighbor}) := g_{cost,temp}(n_{neighbor})$
                        $f_{cost}(n_{neighbor}) := g_{cost}(n_{neighbor}) + h_{cost}(n_{neighbor}, n_{goal})$
                        $n_{neighbor}^{previous} := n_{next}$
end
```