Exploration and path planning

Autonomous Robotics Labs

Labs 04

Outline

- Path planning
- OccupancyGrid
- Exploration
  - Frontier-based exploration
  - WFD
  - Frontier filtering
  - Frontier selection

Path planning – quick refresh

Path an ordered set of points ∼ locations in either joint or operational space – that the robot should follow

Trajectory is path plus velocities and accelerations at each point

- We will be concerned only with *path*
- Planning on a 2D grid (oocupancy grid – **nav_msgs/OccupancyGrid**)
- Graph based algorithms (other types exist)
  - Breadth-first search (BFS) – "*Grass fire*"
  - Depth-first search (DFS)

- A very simplistic explanations of some well-known algorithms:
  - Dijkstra – explore the node that has the least cost so far
  - A* – explore the node with the lowest expected distance
    - uses a heuristic to estimate the distance to target

      $$\textit{expected cost}(\textit{node}) = \textit{cost so far}\,(\textit{node}) + \textit{heuristic}\,(\textit{node})$$

  - Performance depends on the space (e.g., complexity of obstacles) and the specific task (single vs. multi-target search)
- There are many search algorithms and modifications of the most popular ones (A*, Dijkstra)

Path planning

- Note: *Even the most simplistic approach will (probably) work, the difference is execution time and overall efficiency (...memory)*
- Obstacles:
  - path planning with sufficient "safety" margin
  - easiest approach (only possible with a rigid robot or maybe holonomic):
    - inflate obstacles
    - in case of occupancy grid – morphological dilation can be used, e.g.:

```
from scipy.ndimage import morphology
inflated_grid = morphology.binary_dilation(grid)
OR
inflated_grid = morphology.grey_dilation(grid)
```

Parameters:

```
size=(n, m)  # kernel size
structure=np.ones((n, m))  # the kernel itself
iteration=1  # number of iterations
```

OccupancyGrid

`nav_msgs/OccupancyGrid`

data (int8[]) the occupancy grid itself (flattened)
- 0 == empty cell
- 1 . . . 100 == (probably) occupied cell
- −1 == unknown (unseen) cell

info (nav_msgs/MapMetaData) additional information about the grid

- **resolution** (float23) "size" of cell in meters
- **width** width of the grid (in cells)
- **height** height of the grid (in cells)
- **origin** (geometry_msgs/Pose) the relation of the origin [0, 0] of the grid to the "real world" (e.g. map *tf*)
  - **position** (Point) translation of the origin w.r.t. real world
  - **orientation** (Quaternion) rotation of the XY-axes w.r.t. real world

OccupancyGrid

- Remember:
  - robot position needs to be transformed to fit into the grid coordinates
    - translation and rotation – origin
    - scaling – resolution
  - thresholding and/or recalculation of the values of the grid might be necessary depending on the operation (e.g. dilation, probability threshold, ...)
  - make sure you know where the origin is and which is the X-direction and which is the Y-direction (might be confusing when plotting)
  - It's a good idea to optimize the code – operations on a large grid will be costly

Exploration

- Goal:
  - explore the existing world (e.g., the maze)
  - map currently unknown locations $\rightarrow$ where to focus the search?
    $\sim$ explore the "frontier"

Frontier
: a cell that separates known and unknown regions, possibly hiding new parts of the "maze"

- Cells in the grid:

  Open space: $p(occupied) < threshold$
  Occupied space: $p(occupied) > threshold$
  Known_region: $cell\ value \geq 0$
  Unknown region: $cell\ value = -1$

- Frontier-based exploration (in as few words as possible):
  1. Find frontiers
  2. Select and go to a frontier
  3. Repeat until there are no more frontiers (or some other goal is reached)

Frontier-based exploration

[1] YAMAUCHI, Brian, et al. Frontier-based exploration using multiple robots. In: Agents. 1998. p. 47-53.

[2] TOPIWALA, Anirudh; INANI, Pranav; KATHPAL, Abhishek. Frontier Based Exploration for Autonomous Robot. arXiv preprint arXiv:1806.03581, 2018.

[3] USLU, Erkan, et al. Implementation of frontier-based exploration algorithm for an autonomous robot. In: 2015 International Symposium on Innovations in Intelligent SysTems and Applications (INISTA). IEEE, 2015. p. 1-7.

Slightly more indepth article & more efficient implementations:

[4] KEIDAR, Matan; KAMINKA, Gal A. Efficient frontier detection for robot exploration. The International Journal of Robotics Research, 2014, 33.2: 215-236.

Wave-front detection (WFD)

- A BFS algorithm to search for frontiers

  outer BFS search for frontier points connected to the robot position

  inner BFS "frontier assembly" – search for contiguous frontier points

- Definitions:

pose the robot position in the occupancy grid

Map-Open-List list of points enqueued (selected for processing) by the outer BFS

Map-Close-List list of points dequeued (already processed) by the outer BFS

Frontier-Open-List list of points enqueued by the inner BFS

Frontier-Close-list of points dequeued by the inner BFS

$queue_m$ a queue for the map points (points for the outer BFS)

$queue_f$ a queue for the frontier points (points for the inner BFS)

Wave-front detection [4]

```
queue_m ← ϕ
ENQUEUE(queue_m , pose)
mark pose as "Map-Open-List"

while queue_m is not empty:
        p ← DEQUEUE(queue_m)
        if p is marked as "Map-Close-List":
                continue
        if p is a frontier point:
                queue_f ← ϕ
                NewFrontier ← ϕ
                ENQUEUE(queue_f, p)
                mark p as "Frontier-Open-List"

                while queue_f is not empty:
                        q ← DEQUEUE(queue_f)
                        if q is marked as {"Map-Close-List","Frontier-Close-List"}:
                                continue
                        if q is a frontier point:
                                NewFrontier ← q
                                for all w ∈ neighbors(q):
                                        if w not marked as {"Frontier-Open-List",
                                        "Frontier-Close-List","Map-Close-List"}:
                                                ENQUEUE(queue_f, w)
                                                mark w as "Frontier-Open-List"
                        mark q as "Frontier-Close-List"
                save data of NewFrontier
                mark all points of NewFrontier as "Map-Close-List"
        for all v ∈ neighbors(p):
                if v not marked as {"Map-Open-List","Map-Close-List"}
                and v has at least one "Map-Open-Space" neighbor:
                        ENQUEUE(queue_m, v)
                        mark v as "Map-Open-List"
        mark p as "Map-Close-List"
```

Frontier filtering

- Make sure the frontier is reachable
  - test path planning
  - run WFD on grid with inflated obstacles
    - remember connectivity
    - keep occupancy information (-1/0/>0)
- Discard useless frontiers
  - too small
  - possibly does not conceal explorable areas

Frontier selection

- Simple approach:
  Random frontier
- Better approach:
  Closest frontier
  - direct distance (Euclidean, Manhattan)
  - path distance – costly but more accurate & includes reachability test

  Most promising frontier
  - e.g., expected distance to a goal

Thank you for your attention