Planning

Planning is a term used in control theory, artificial intelligence, manufacturing, and many other areas including robotics of course. It can be applied for example to solve puzzles, assembly in automotive, for programming virtual humans and smart characters in movies or video games, drug design, scheduling, etc. Dozens of methods were developed during last decades differing in the task they are used, representation of the environment the planning is done as well as movement constrains. Generally, the planning problem is NP-hard, which means that no polynomial algorithm exist and therefore some simplifications of the problem are needed or approximations algorithms should be developed.

In this lecture we describe a practical approach for planing a path for a circular-shaped robot in the environment represented as a binary grid, i.e. a grid, where cells representing an empty space have value 0, while cells corresponding to an obstacle have value 1. A binary grid can be also understood as a binary image, where obstacles are shown as black pixels, while free space is white, see Fig. 1.

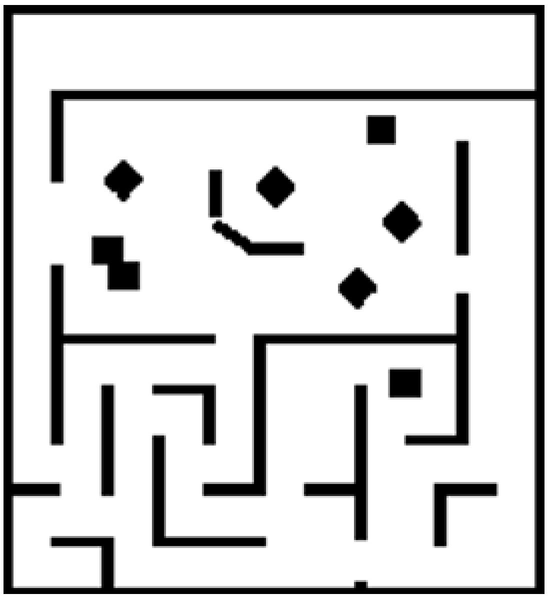

Fig.1.: *A map as a binary grid.*

In this case, an algorithm computing a shortest path between given start and goal positions on a graph can be employed. To do that, some preprocessing is needed: the graph has to be constructed from a binary grid. Notice that an algorithm that plans on a graph assumes that a robot is a point, i.e. it can pass an arbitrary edge in the graph. Fortunately, adjusting the graph for a circular robot is relatively straightforward, it only involves inflating of obstacles by a radius of the robot. A path found this way is optimal on a grid, but can be suboptimal in Euclidean space, see Fig. 2. The path is therefore smoothed and only a vector important nodes of the path is returned as a final path.

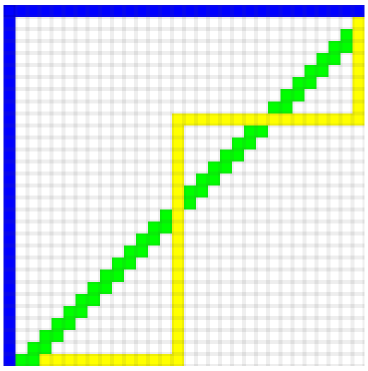
Fig.2.: *All paths have the same length on a grid (they contain the same number of cells).*

Assuming a binary grid, the start position of the robot, the goal position, and robot's radius, output of the algorithm is a vector of nodes (points) to be sequentially traversed and the planning algorithm is a sequence of the following steps:

- Inflate obstacles in the binary grid by a radious of the robot.
- Construct a graph from the inflated binary grid.
- Run a graph searching algorithm.
- Smooth the found path.

The particular steps are described in details in the next paragraphs.

Map inflation

Position of a circular robot in 2D space is typically described by two parameters: *x* and *y* representing coordinates of robot's center in some coordinate system (note that robot orientation is not considered as it has no influence on the space the robot occupies. Moreover, we assume a differential driving of the robot, which means that it can turn on the spot). It is clear that robot's center cannot be placed at an arbitrary empty place in the environment, instead if the center is close to some obstacle, the robot collides with it. Therefore, all positions of robot's center that collide with any obstacle should be determined. Intuition behind this determination is shown in Fig. 3, where the gray area was identified to be colliding with the robot.

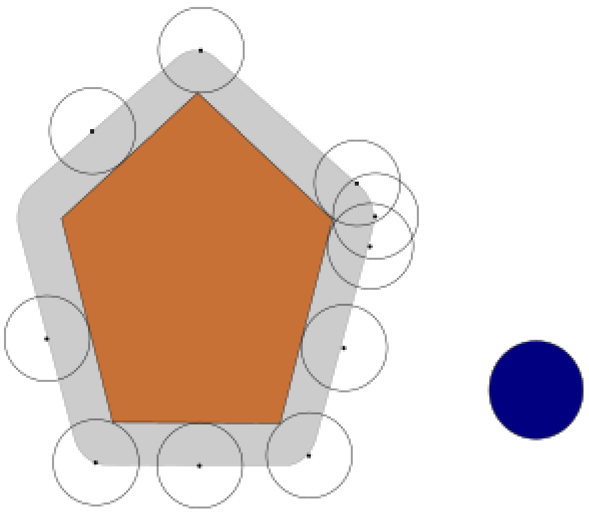
Fig.3.: *The gray color depicts an area of colliding positions of the blue robot with the brown obstacle.*

Mathematical apparatus for computing this area is called **Minkowski sum (or Minkowski addition)**, which is a binary operation over two sets *P* and *Q*:

$$P \oplus Q = \{p + q | p \in P, q \in Q\}$$

.

If the sets are described by convex polygons, Minkowski sum can be computed in $O(m+n)$, where *m* and *n* are number of verticies of *P* and *Q* [1]. In case, only one polygon is convex and another one is not, the complexity increases to $O(nm)$. If both of them are nonconvex, their Minkowski sum complexity is $O((mn)^2)$.

For binary grids, which can be viewed as images, mathematical morphology known used in image processing can be applied. Specifically, dilation is a special case of Minkowski sum for images, which uses *a structuring element* for expanding of an input image. A structuring element is a point set, i.e. a set of point placed relatively to the origin of the element. Typical structuring elements are depicted in Fig. 4, where the origin is highlighted by a cross.

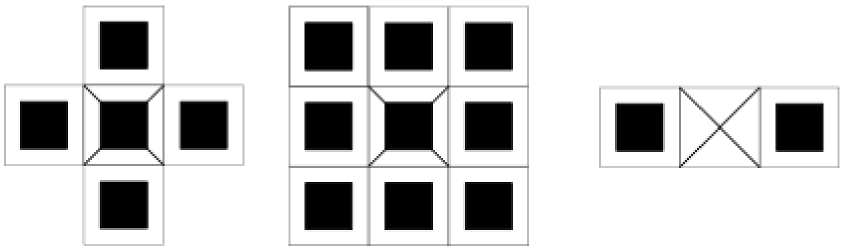
Fig.4.: *Typical structuring elements.*

Dilation traverses the input image (grid) row by row and column by column and whenever it reaches a black pixel, it places a structuring element so, that its origin lies on the pixel. The structuring element is than used as a template for drawing: new black pixels are drawn at pixels covered by the structuring element. An example of dilation is shown in Fig.5

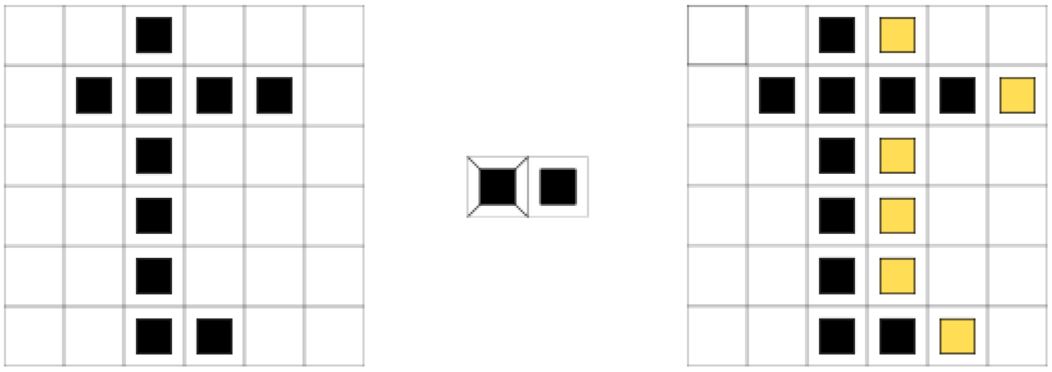

Fig.5.: *Dilation of the image (on the left) by the structuring elements in the middle. The added pixels are in yellow.*

A structuring element for a circular robot is approximation of a circle in a grid (given the radius of the robot and a real size of one cell) as can be seen in Fig. 6.

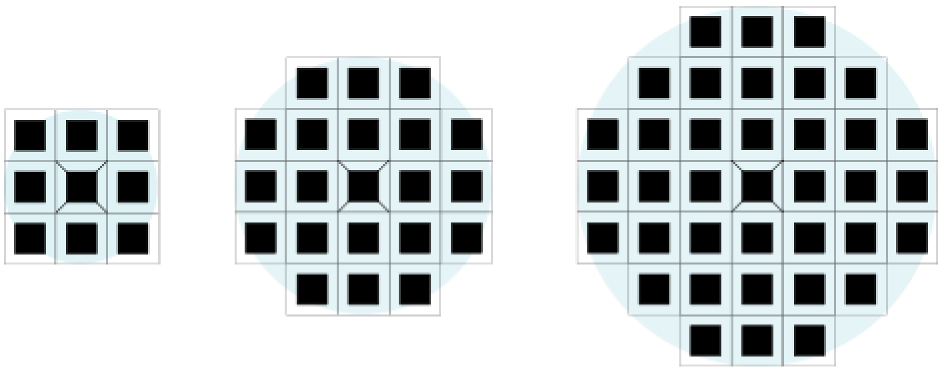

Fig.6.: *A structuring element for a circular robot.*

Fig. 7 shows the inflated map from 1. Notice that narrow corridors disappeared as the robot cannot pass through them. Therefore, it is important to choose appropriate structuring element carefully. If it is small, the generated path can collide with obstacles. On the other hand, a large structuring element may cause that an optimal path (or even any path) is not found.

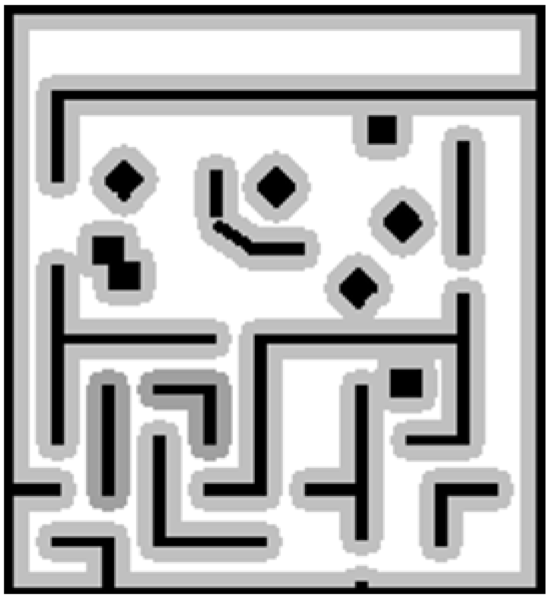

Fig.7.: *The map from Fig. 1 inflated.*

Adjacency graph construction

Once the grid is inflated, adjacency graph of cells in the inflated grid can be constructed. The graph $G(V, E)$ contains a set of nodes *V* and a set of edges *E*, where an edge $e_{ij} \in E$ is a pair of nodes: $e_{ij} = v_i, v_j$. Moreover, each edge has a weight (cost) associated with it expressing effort needed to traverse it. In our case, V is a set of all empty (non occupied) grid cells and we connect two nodes with an edge when their corresponding grid cells share a common edge or point. The cost of the edge is determined as a distance of the centers of the involved cells, i.e. if the cells have a common edge, the cost is 1, while in the case of the common point, the cost is $\sqrt{2}$. Sometimes it is useful to represent distances as integers. A good approximation is to set cost to 5 for edge going vertically or horizontally (i.e. connecting cells with a common edge) and to 7 for diagonal edges, see Fig. 8.

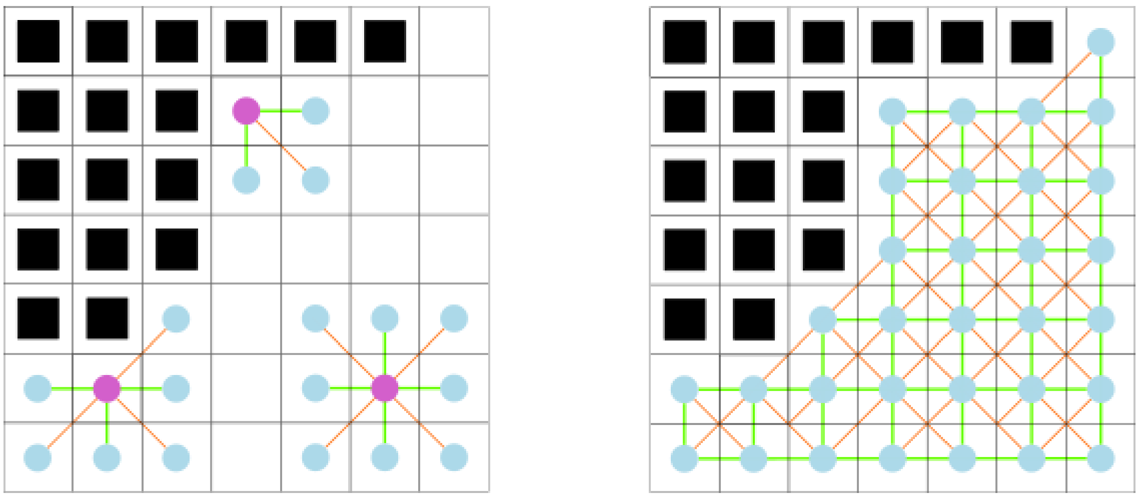
Fig.8.: *Left: Neighbours (in blue) of the violet nodes. Horizontal and vertical edges are in green, while diagonal edges are in orange. Right: the full adjacency graph.*

Dijkstra's algorithm

We can employ some graph searching algorithm to find a path from the current robot position to a given goal. An ideal candidate (at first sight) is A* algorithm [2] introduced by Nils Nilsson in 1968, which is widely used as it is easily implementable, fast, and it provides optimal solutions. As we will need to evaluate costs of (path lengths to) several nodes in the exploration instead of just the single goal position we will use Dijkstra's algorithm [3], which computes shortest paths to all nodes in the graph from a given start position.

Dijkstra's algorithm, as a predecessor of A* was introduces by Edsger W. Dijkstra in 1956. Its key idea is to manage a set of nodes for which a shortest path has been determined and add the nearest node to this set in each iteration. To do that, the algorithm stores actual values of the length of the shortest path v_{dist} and predecessor of each vertex along this path v_{prev} for each vertex $v \in \mathcal{G}$. The shortest path for the given vertex v can be then easily determined by walking consecutively over predecessors starting in v. In the initialization stage, distances v_{dist} of all the vertices are set to infinity and their predecessor v_{prev} to a fictive vertex v_{∞}, what means that the shortest path has not been found yet (lines 1-4). The only exception is the starting vertex, whose distance value is set to 0 (line 5). The start vertex is then put into a priority queue, which is sorted according to v_{dist} (line 6).

The algorithm then consecutively takes the vertices from the priority queue (line 8) and for the current vertex u it processes its neighbours (line 9). If u is reached first time with the algorithm, it is updated accordingly (lines 11 and 12) and added into the priority queue (line 13). If it is not the case, a tentative distance to the start is computed (line 15) for each neighbour v and if it is smaller than the current one, v_{dist} is updated to the tentative distance (line 17) and v_{prev} to u (line 18) and v is updated in the priority queue (line 19).

Alg 1. Dijkstra's algorithm

```
1. **foreach** vertex $v \in \mathcal{G}$ **do**
2.     $v_{dist} = \infty$
3.     $v_{prev} = v_{\infty}$
4. **end**
5. $start_{dist} = 0$;
6. $\mathcal{G}.add(start, 0)$
7. **while**($Q \neq \emptyset$) **do**
8. u = $\mathcal{Q}.pop()$;
9. **foreach** neighbour $v$ of $u$ **do**
10.     **if** ($v_{dist} == \infty$)
11.         $v_{dist} = u_{dist} + d(u, v)$
12.         $v_{prev} = u$
13.         $\mathcal{Q}.add(v)$
14.     **else**
15.         $tentative = u_{dist} + d(u, v)$
16.         **if**($tentative < v_{dist}$)
17.             $v_{dist} = tentative$
18.             $v_{prev} = u$;
19.             $\mathcal{Q}.update(v, v_{dist})$
20.         **end**
```

```
21.     end
22. end
```

Note employment of a binary heap. Heap [4] is a structure based on a binary tree that allows fast (in a constant time) answers to questions on the highest number (given some comparison predicate) in a set of numbers. Moreover, It supports inserting new nodes into the tree, updating nodes in the tree, and deleting the highest number from the tree. All these operations are done in $O(log(n))$.

Path smoothing

As we already mentioned, the path generated by Dijkstra's algorithm is optimal on a graph, but can be suboptimal in Euclidean space. It can be proven that an optimal path for a point robot (which is our case as we reformulated the task by inflating obstacles) is a polyline. In contrast, the generated path contains many nodes that lie on a line and thus bring no information. These unnecessary nodes will be removed from the path in the final step, which, as a side effect will lead to a shorter path in Euclidean space.

The algorithm starts in the start node and tests, whether the next node on the path is visible from the start point. If yes, it goes to the next node and performs the visibility test again. This is consecutively repeated until the first invisible node is reached. The previous note is taken in that case, pushed into the final path and set as a new starting point. This procedure is repeated until the goal node is reached.

The whole process is illustrated in Fig. 9. The start node is represented by the read color, while the goal is green. The yellow lines are those not intersecting obstacles and thus allowing the process to proceed to a next point. The read lines, in contrary, hit obstacles and the process steps back to the previous node (filled in pink). The pink nodes determine (together with the goal) the final path, while the cyan ones are discarded.

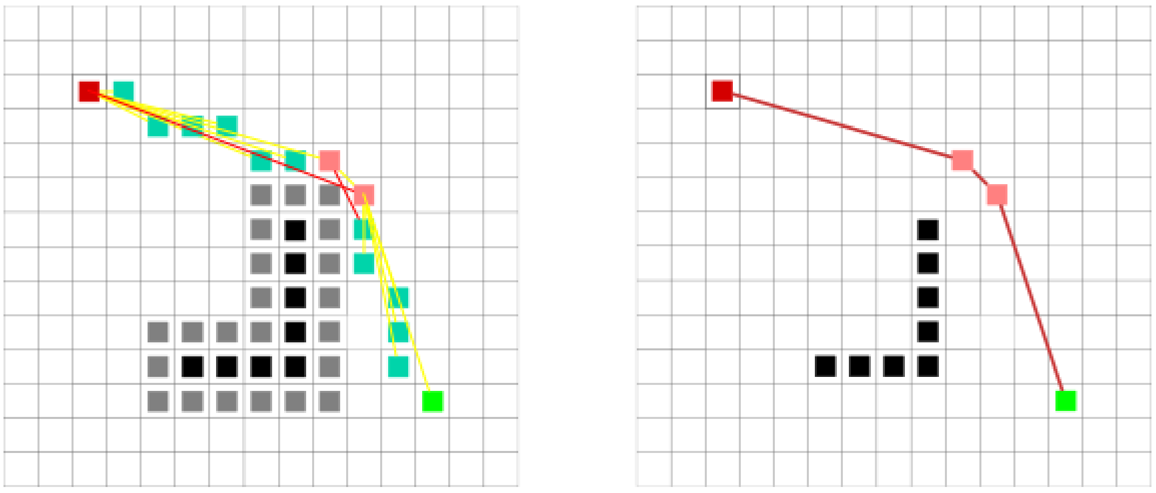

Fig.8.: *Left: the path smoothing process by node filtering. Right: the final path.*

Bresenham's line algorithm [5] is employed to perform visibility test. This algorithm is widely used in computer graphics for line drawing. It consecutively computes coordinates of pixels to be drawn. Instead of drawing a pixel, we will check, whether the pixel represents inflated obstacle and if yes, we report that end points of the line are not visible.

References

[1] Minkowski sum https://en.wikipedia.org/wiki/Minkowski_addition
[2] A* algorithm https://en.wikipedia.org/wiki/A*_search_algorithm
[3] Dijkstra algorithm https://en.wikipedia.org/wiki/Dijkstra's_algorithm
[4] Binary heap https://en.wikipedia.org/wiki/Binary_heap
[5] Brensenham line drawing algorithm https://en.wikipedia.org/wiki/Bresenham's_line_algorithm