Two-way D* algorithm for path planning and replanning

Marija Dakulović[a,*], Ivan Petrović[a]

[a]*Department of Control and Computer Engineering, Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia, fax: +38516129809*

Abstract

Inspired by the Witkowski's algorithm, we introduce a novel path planning and replanning algorithm – the two-way D* (TWD*) algorithm – based on a two-dimensional occupancy grid map of the environment. Unlike the Witkowski's algorithm, which finds optimal paths only in binary occupancy grid maps, the TWD* algorithm can find optimal paths in weighted occupancy grid maps. The optimal path found by the TWD* algorithm is the shortest possible path for a given occupancy grid map of the environment. This path is more natural than the path found by the standard D* algorithm as it consists of straight line segments with continuous headings. The TWD* algorithm is tested and compared to the D* and Witkowski's algorithms by extensive simulations and experimentally on a Pioneer 3DX mobile robot equipped with a laser range finder.

Keywords:
Graph search, path planning, Euclidean shortest path, mobile robotics

1. Introduction

An autonomous mobile robot is expected to perform goal-directed tasks in cramped and unknown environments. The task of the path planning algorithm is to compute an optimal path to the given goal and to replan the path in case the previously planned path is blocked by obstacles.

The majority of path planning algorithms produce a graph of possible paths to the goal [1] and then apply a classical graph search algorithm [2], such as the A* [3], D* [4] or focused D* (FD*) [5] algorithms, to find the optimal path. The D* and FD* algorithms are very often used because of their capabilities of fast replanning in changing environments, i.e. in environments where new obstacles can be added or removed. Two-dimensional (2D) occupancy grid maps are usually used to represent a continuous environment by an equally-spaced grid of discrete points [6]. Grid cells cover the area densely and each grid cell contains information about its occupancy.

The path obtained by a classical graph search algorithm based on uniform resolution grid is a zigzag line with sharp turns, angles of which are limited to increments of 45° [4, 5, 7]. A mobile robot can not follow such a path smoothly due to its kinematic and dynamic constraints. To overcome this limitation a number of algorithms have been developed which produce paths not constrained to a small set of headings [8, 9, 10, 11, 12, 13]. For example, the Field D* algorithm [8] extends the standard D* algorithm by using linear interpolation to produce straight line segments with continuous headings. Although Field D* produces less costly paths than the D* algorithm, the path optimality is not guaranteed. A different approach to produce straight line segments with continuous headings is proposed by Gennery in [9]. His algorithm is based on the Witkowski's algorithm [14], which uses two-way (forward and backward) passes of the breadth-first search (BFS) to find the optimal paths. BFS can find optimal paths only in graphs that has all weights equal [2]. Gennery adapted the Witkowski's algorithm for finding the paths in graphs with arbitrary weights, but his adaptation does not guarantee the path optimality. Besides, the Witkowski's and Gennery algorithms perform poorly in changing environments as the path must be completely replanned each time the environment changes, which significantly enlarges the computational time of the search algorithm.

This paper presents a new path planning algorithm, called two-way D* (TWD*) algorithm, which is actually the Witkowski's algorithm with the two-way breadth-first search replaced by the two-way D* search. By applying the two-way D* search of the graph, an area of minimal path costs from the start node to the goal node in graphs with arbitrary weights is found and an algorithm for optimal path calculation through that area is developed. The calculated path is the shortest possible path in the geometrical space composed of long straight line segments with continuous headings. The proposed algorithm also performs well in changing environments although it is computationally more demanding than the D* algorithm.

The rest of the paper is organized as follows. Section 2

*Corresponding author
Email addresses: marija.dakulovic@fer.hr (Marija Dakulović), ivan.petrovic@fer.hr (Ivan Petrović)

briefly reviews the D* and Witkowski's algorithms. Section 3 presents the new path planning algorithm. In Section 4 test results are given and compared to those obtained by the Witkowski's and D* algorithms under the same conditions. Finally, Section 5 gives the conclusions of the paper.

2. Related algorithms

Path planning and replanning algorithm proposed in this paper is based on the D* algorithm [4] and the Witkowski's algorithm [14], and therefore these two algorithms are briefly surveyed in this section. Since both algorithms as well as our proposed algorithm create and search graphs in occupancy grid maps of the environment, we firstly present used maps and the process of graph creation and search.

The following notation is used for algorithms description throughout the paper. For any variable $x \in \mathbb{R}^n$ norm infinity and Euclidean norm are defined as $\|x\|_\infty := \max_{i\in\{1,\dots,n\}} |x_i|$ and $\|x\| := \sqrt{x^T x}$, respectively. A *set* is an unordered collection of distinct objects (e.g., $\mathcal{S} = \{1, 5, 8\}$). The *cardinality* of a set $\mathcal{S}$, denoted $|\mathcal{S}|$, is the number of elements in $\mathcal{S}$. The *empty set* is denoted by $\emptyset$. A *list* is an ordered collection of objects (e.g., $\mathcal{L} = [8, 1, 1]$). The ordered elements of the list $\mathcal{L}$ are denoted with $\mathcal{L}[1], \dots, \mathcal{L}[|\mathcal{L}|]$, where $|\mathcal{L}|$ is the *length* (the number of elements) of the list $\mathcal{L}$. The *empty list* is denoted by $\emptyset$.

2.1. Used occupancy grid maps and robot representation

An occupancy grid map is created by approximate cell decomposition of the environment [1], [6]. The whole environment is divided into squared cells of equal size e_{cell}, which are abstractly represented as the set of M elements $\mathcal{M} = \{1, \dots, M\}$ with corresponding Cartesian coordinates of cell centers $c_i \in \mathbb{R}^2$, $i \in \mathcal{M}$. Each cell contains occupancy information of the part of the environment that it covers. In this paper two types of occupancy grid maps are used: binary occupancy grid maps and weighted occupancy grid maps [15].

A binary occupancy grid map contains only free and occupied cells. Binary occupancy function $o(i) \in \{1, \infty\}$, $i \in \mathcal{M}$ is used for representing the set of all obstacles in the environment noted as $\mathcal{O} = \{i \in \mathcal{M} \mid o(i) = \infty\}$ and free environment is represented by the set of free cells noted as $\mathcal{N} = \mathcal{M} \setminus \mathcal{O}$, see Fig. 1.

A weighted occupancy grid map contains free cells ($o(i) = 1$), occupied cells ($o(i) = \infty$) and also cells with other occupancy values ($1 < o(i) < \infty$). The cells with other occupancy values belong to the so-called safety cost mask which is introduced in the map around the obstacles to push the path away from the obstacles in order to assure safe robot motion. The size of the safety cost mask is defined by the integer number of cells M_c. The number of cells M_c influence on the distance between the path and

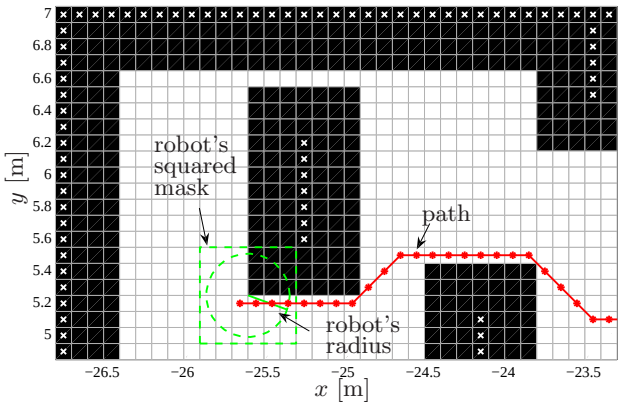

Figure 1: A section of the experimental environment represented by the binary occupancy grid map. For illustration purposes free space is colored white ($o(\cdot) = 1$), and obstacles are colored black ($o(\cdot) = \infty$), where real obstacle position is marked by x. An example of the path produced in this grid map is marked by *. The robot's radius is $r_r = 0.26$ m, and the cell size is $e_{cell} = 0.1$ m.

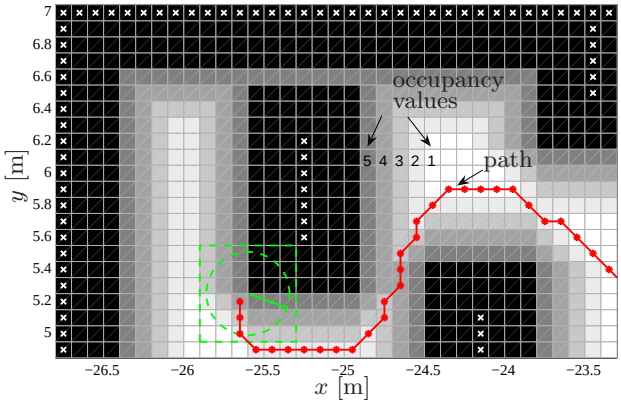

Figure 2: A section of the experimental environment represented by the weighted occupancy grid map with the safety cost mask ($M_c = 4$). Free space is colored white ($o(\cdot) = 1$), unoccupied space within the safety cost mask is colored by shades of gray ($o(\cdot) \in \{2, 3, 4, 5\}$), and obstacles are colored black ($o(\cdot) = \infty$), where real obstacle position is marked by x. An example of the path produced in this grid map is marked by *.

the obstacles and can be determined as in our previous work [15]. Occupancy value of a cell within the safety cost mask depends on its distance from the closest occupied cell. The utmost cells of the safety cost mask obtain the occupancy value for one greater than unoccupied cells out of the safety cost mask ($o(i) = 2$), and occupancy values of inner cells incrementally increase from utmost cells to the occupied cells. In this way defined safety cost mask does not prevent the robot to pass through the narrow passages. Therefore, occupancy function of the occupancy grid maps with safety cost masks is defined as follows:

$$o(i) = \begin{cases} \max\{1, (M_c + 2 - \min\limits_{j\in\mathcal{O}} \|c_i - c_j\|_\infty)\} & \text{if } i \notin \mathcal{O} \\ \infty \quad \text{if } i \in \mathcal{O} \end{cases} \tag{1}$$

Described procedure generates a smooth decrease of oc-

cupancy values from the obstacles towards the free space. Thus, the safety cost mask acts similarly to the artificial potential field [16]. Fig. 2 represents the weighed occupancy grid map with $M_c = 4$ cells wide safety cost mask of the same section of the environment as shown in Fig. 1.

We assume that real shape of the mobile robot can be approximated by a circle of a radius r_r, which is very often used assumption in the literature. In that case the robot is represented by a squared mask in the grid map, within which the circular shape of the robot can be drawn. Thus, all obstacles in the grid map are enlarged for the integer number of cells $\lceil r_r/e_{cell} \rceil$, i.e. the robot is described by a squared mask of size $2 \cdot \lceil r_r/e_{cell} \rceil \cdot e_{cell}$. Real shape of the robot and its squared mask in occupancy grid maps are depicted in Figs. 1 and 2. It can be noticed that at corners the real obstacles are enlarged $\sqrt{2}$ times more than strictly necessary, which additionally confirms the safety of trajectories that go through the free cells. A safety cost mask defined in this way allows the robot to rotate in place at each point within the free space of the grid map and consequently the path planning algorithm needs to plan only the robot positions in the free space. In case of more complex robot shapes (e.g. rectangle or ellipse), the path planning algorithm should also plan the robot orientation [17].

2.2. Graph creation and search

Weighted undirected graph $\mathcal{G}(\mathcal{N}, \mathcal{E}, \mathcal{W})$ is created in the occupancy grid map in such a way that all unoccupied cells $\mathcal{N} = \mathcal{M} \setminus \mathcal{O}$ represent the set of nodes in the graph. Further, we say that two nodes $i, j \in \mathcal{N}$ in the graph are *neighbors* if $\|c_i - c_j\|_\infty = e_{cell}$. The set of edges is defined as $\mathcal{E} = \{e_{i,j} := \{i, j\} \mid i, j \in \mathcal{N}, i \text{ and } j \text{ are neighbors}\}$. The set of edge weights $\mathcal{W} = \{w_{i,j} \mid i, j \in \mathcal{N}, i \text{ and } j \text{ are neighbors}\}$ is defined as the cost of transition between neighbors as

$$w_{i,j} := \|c_i - c_j\| \cdot \max\{o(i), o(j)\}. \tag{2}$$

In binary occupancy grid maps there are two values of transitions between neighbors: straight and diagonal transition. In weighted occupancy grid maps with safety cost mask transitions between neighbors are additionally weighted according to their distances to the obstacles.

Assume that the *start* node and the *goal* node are specified in the graph $\mathcal{G}(\mathcal{N}, \mathcal{E}, \mathcal{W})$. The search for optimal path from the start node to the goal node has to be performed. A list $\mathcal{P} = \mathcal{P}(start, goal)$ is called a *path* between the start node and the goal node in a graph $\mathcal{G}(\mathcal{N}, \mathcal{E}, \mathcal{W})$ if:

$$\begin{gathered} \mathcal{P}[1] = start, \quad \mathcal{P}[|\mathcal{P}|] = goal, \\ \mathcal{P}[i] \in \mathcal{N}, \quad i = 1, \ldots, |\mathcal{P}|, \\ \{\mathcal{P}[j], \mathcal{P}[j+1]\} \in \mathcal{E}, \quad j = 1, \ldots, |\mathcal{P}| - 1. \end{gathered} \tag{3}$$

The cost of the path $\mathcal{P}$ is defined as the sum of weights of edges along the path, i.e.,

$$c(\mathcal{P}) := \sum_{i=1}^{|\mathcal{P}|-1} w_{\mathcal{P}[i],\mathcal{P}[i+1]}. \tag{4}$$

Let $\pi(start, goal)$ be the set of all possible paths between the *start* node and the *goal* node in the graph $\mathcal{G}(\mathcal{N}, \mathcal{E}, \mathcal{W})$. The *optimal cost* of the path from the *start* node to the *goal* node that has to be searched for is defined as

$$\begin{aligned} c^*(start, goal) := \quad & \min_{\mathcal{P}} \quad c(\mathcal{P}) \\ & \text{s.t.} \quad \mathcal{P} \in \pi(start, goal) \end{aligned} \tag{5}$$

with implicit assumption that $c^*(start, goal) = \infty$ if $\pi(start, goal) = \emptyset$. The *optimal path* from the *start* node to the *goal* node is the path (or more than one path) that has optimal cost $c^*(start, goal)$. For descriptions of the graph search algorithms in this paper, cost functions $g(n)$ and $h(n)$ are introduced. The cost $g(n)$ denotes the unique cost of the optimal path from the node n to the *goal* node, i.e. $g(n) \equiv c^*(n, goal)$, and the cost $h(n)$ denotes the unique cost of the optimal path from the *start* node to the node n, i.e. $h(n) \equiv c^*(start, n)$.

2.3. The D algorithm*

The D* algorithm is a well known graph search algorithm capable of fast replanning in changing environments [4]. It is also known as dynamic version of the Dijkstra's algorithm or dynamic version of the A* algorithm without the heuristic function [17]. The D* algorithm finds the optimal path in graphs in which weights change during the time.

For every searched node n, the D* algorithm computes the cost value $g(n)$ of the path from the node n to the *goal* node and the value of the key function $k(n)$ for the replanning process, which stores the old values $g(n)$ before changes of weights in the graph happened. The algorithm stores the *backpointers* for every searched node n, which point to the parent node with the smallest cost g. A backpointer is noted by the function $b(\cdot)$, where $b(n) = m$ means that the node n has the smallest cost because it follows the node m. The backpointers ensure that optimal path from any searched node n to the goal node can be extracted according to the function $b(\cdot)$ as follows

$$\begin{gathered} \mathcal{P}_{D^*}[1] = n, \\ \mathcal{P}_{D^*}[i] = b(\mathcal{P}_{D^*}[i-1]), \quad i = 2, \ldots, |\mathcal{P}_{D^*}|, \end{gathered} \tag{6}$$

thus ensuring that the extraction in (6) stops with the goal node ($\mathcal{P}_{D^*}[|\mathcal{P}_{D^*}|] = goal$).

D* handles changes of the path cost during the search process by processing the *raise* and the *lower* nodes. The raise nodes are classified by relation $k < g$ and the lower nodes by relation $k = g$. The raise nodes propagate information about the path cost increases starting at the newly occupied nodes, update the costs to all nodes that are tied by backpointers, and continue towards the start node (robot's current position). They activate neighbor nodes that can lower the path cost – the lower nodes. The lower nodes propagate information about the path cost decreases and re-direct backpointers to compute new optimal paths of the nodes that were previously raised.

The execution of the D* algorithm can be divided into *initial planning* and *replanning* phases. Initial planning is performed if the robot is standstill at the start position ($R = start$) and replanning is performed if the robot detects nodes with changed occupancy values during its motion ($\mathcal{I}_R$ is the set of changed nodes at position R). The pseudocode of the D* algorithm is given by Alg. 2.1 and the pseudocode of the algorithm which invokes the D* algorithm during the robot motion is given by Alg. 2.2. D* uses the set *Open* as a temporary storage for the currently examined nodes. The nodes are added to the set *Open* by Alg. 2.3.

Algorithm 2.1: D*$(\mathcal{I}_R, R)$

```
1:  for ∀n ∈ I_R \ Open | b(n) ≠ n
2:      insert-D*(n, g(n))
3:  end
4:  k_min ← 0
5:  while Open ≠ ∅ and k_min ≤ g(R)
6:      k_min ← min{k(n) | n ∈ Open}
7:      o ← n*   //for n* such that k(n*) = k_min
8:      Open ← Open \ {o}
9:      if k_min < g(o)
10:         for ∀n ∈ N such that {o, n} ∈ E
11:             if g(n) ≤ k_min and g(o) > g(n) + w_{o,n}
12:                 g(o) ← g(n) + w_{o,n}
13:                 b(o) ← n
14:             end
15:         end
16:     end
17:     if k_min = g(o)
18:         for ∀n ∈ N such that {o, n} ∈ E
19:             if (b(n) = o and g(n) ≠ g(o) + w_{n,o})
                or (b(n) ≠ o and g(n) > g(o) + w_{n,o})
20:                 insert-D*(n, (g(o) + w_{n,o}))
21:                 b(n) ← o
22:             end
23:         end
24:     else
25:         for ∀n ∈ N such that {o, n} ∈ E
26:             if b(n) = n
                or (b(n) = o and g(n) ≠ g(o) + w_{n,o})
27:                 insert-D*(n, (g(o) + w_{n,o}))
28:             else if (b(n) ≠ o and g(n) > g(o) + w_{n,o})
                and o ∉ Open
29:                 insert-D*(o, g(o))
30:             else if (b(n) ≠ o and g(o) > g(n) + w_{o,n})
                and n ∉ Open and k_min < g(n)
31:                 insert-D*(n, g(n))
32:             end
33:         end
34:     end
35: end
```

Algorithm 2.2: move-robot-D*$(start, goal)$

```
1:  R ← start
2:  ∀n ∈ N, k(n) ← ∞, g(n) ← ∞, b(n) ← n
3:  Open ← ∅  //Initialization
4:  insert-D*(goal, 0)
5:  while R ≠ goal
6:      I_R ← changed-nodes(R)
7:      if R = start or I_R \ Open ≠ ∅
8:          D*(I_R, R)  //Initial planning or replanning
9:      end
10:     R ← b(R)
11: end
```

Algorithm 2.3: insert-D*(n, g_{min})

```
1:  if n ∈ Open  k(n) ← min{g(n), g_min}
2:  else k(n) ← min{k(n), g_min}
3:  g(n) ← g_min
4:  Open ← Open ∪ {n}
```

Initial planning. The D* algorithm starts by inserting the goal node into the set *Open*. It subtracts the node o with the minimal value k from the set *Open*. In initial planning all searched nodes are lower, i.e. $k_{min} = g(n)$, and therefore only execution of lines 17-24 of Alg. 2.1 is performed, since according to the conditions in lines 6 and 19 a node that will raise its cost g will never be inserted into the set *Open*. The node o is *expanded* with its neighbors by inserting into the set *Open* all nodes which cost g can be lowered by the node o. Every inserted node n in the set *Open* has values $g(n)$ and $k(n)$ calculated according to the values of the parent node o. Also, the backpointer b points towards the o since o is currently the best parent node. The algorithm stops initial planning when the start node is chosen as the best node. The robot can follow the path simply by applying the backpointer function $b(\cdot)$. If we want to perform the exhaustive search by the D* algorithm, then in line 5 of Alg. 2.1 stands only the condition $Open \neq \emptyset$, while the condition $k_{min} \leq g(R)$ is removed.

Replanning. If the robot's sensor detects nodes with changed occupancy values, the D* algorithm performs replanning. The function **changed-nodes**(R) in Alg. 2.2, line 6, returns all nodes with changed occupancy values according to the prior grid map. The weights in the graph $\mathcal{G}(\mathcal{N}, \mathcal{E}, \mathcal{W})$ are updated according to (2). If there are discrepancies between the initial grid map and sensor readings when $R = start$, the D* algorithm performs initial planning with the updated grid map. All changed nodes are inserted in the set *Open* if the algorithm expanded them in the previous executions (lines 1-3 of Alg. 2.1). Among them is the node with the smallest value k, since this node was the best node in the previous execution of the algorithm and minimal cost of nodes in the set *Open* never decreases [3]. Therefore, for the best node is $k = g$.

This node propagates the change of the cost g (increase or decrease) to the parent nodes (nodes tied by backpointers $b(\cdot)$) or lowers the cost of the neighbor nodes that are not tied by backpointers (lines 17-24). If the changed nodes increase occupancy values, the raise nodes will arise, otherwise the lower nodes will arise. The raise nodes further propagate the cost increases to their parent nodes (lines 24-34). At each iteration it is checked if the cost of the best node can be decreased (lines 9-16). Also it is checked if the cost of the neighbor node that is not tied by the backpointer can be decreased (lines 28-30). Then the best node is inserted again in the set *Open*. Cycles in backpointers are avoided by this step. It is also checked if the neighbor node that is not tied by the backpointer can lower the cost of the best node (lines 30-32). Then this neighbor node is inserted in the set *Open* for later examination. When the best node is lower node, it propagates cost decrease to the parent node, and redirects pointers to other nodes (lines 17-24). The iterations are repeated while there exists a node in the set *Open* with the cost k less then the cost $g(R)$. The number of expanded nodes is minimal and consequently the time of execution.

2.4. The Witkowski's algorithm

The Witkowski's algorithm uses two passes (forward and backward) of the breadth-first search (BFS) algorithm. It uses the FIFO queue for node examinations, i.e. the first node inserted into queue (operation push) is firstly examined and taken out of the queue (operation pop). The algorithm is invented for implementation on parallel computers in which each node is examined in parallel. Consequently, the time of execution is proportional to the length of the path. The BFS algorithm finds optimal paths only in graphs that have all weights equal [2]. BFS allways calculates the path with the smallest number of steps (edges) to the goal node. In a weighted graph (with arbitrary weights) such path is not necessarily the optimal one.

Pseudocode of the Witkowski's algorithm is given by Alg. 2.4. Forward and backward queues are represented as lists (which are ordered) $\mathcal{Q}_{start}$ and $\mathcal{Q}_{goal}$, respectively. In the forward pass nodes are searched from the start node to the goal node and minimal cost $h(n)$ of the path from the start node to any expanded node n is memorized. In the backward pass nodes are searched from the goal node to the start node and minimal cost $g(n)$ of the path from the goal node to any expanded node n is memorized. At the end of both BFS passes, costs g and h for every expanded node n give the complete cost of the path calculated as the sum, $f(n) = h(n) + g(n)$ (line 21). Value $f(n)$ is the cost of the path from the start node to the goal node that passes through the node n and which is composed of two optimal paths: from *start* to n and from n to *goal*. The smallest value f is in all optimal paths from the start node to the goal node and is equal to $f^* = f(start) = f(goal)$.

If there exist more than one optimal path in the graph from the start node to the goal node, all optimal paths are composed of integer number of straight transitions and

Algorithm 2.4: Witkowski($start, goal$)

```
 1: ∀n ∈ 𝒩, h(n) ← ∞, g(n) ← ∞, f(n) ← ∞
    𝒬_start ← ∅, 𝒬_goal ← ∅  //Initialization
 2: push(𝒬_start, start)  //Forward pass
 3: h(start) ← 0
 4: while 𝒬_start ≠ ∅ and ∃o ∈ 𝒬_start | h(o) ≤ h(goal)
 5:     o ← pop(𝒬_start)
 6:     for ∀n ∈ 𝒩 such that {o, n} ∈ ℰ
 7:         if h(n) > h(o) + w_{n,o}
 8:             push(𝒬_start, n)
 9:             h(n) ← h(o) + w_{n,o}
10:         end
11:     end
12: end
13: push(𝒬_goal, goal)  //Backward pass
14: g(goal) ← 0
15: while 𝒬_goal ≠ ∅ and ∃o ∈ 𝒬_goal | g(o) ≤ g(start)
16:     o ← pop(𝒬_goal)
17:     for ∀n ∈ 𝒩 such that {o, n} ∈ ℰ
18:         if g(n) > g(o) + w_{n,o}
19:             push(𝒬_goal, n)
20:             g(n) ← g(o) + w_{n,o}
21:             f(n) ← h(n) + g(n)
22:         end
23:     end
24: end
```

integer number of diagonal transitions. Variations in order of straight and diagonal transitions make different optimal paths. Let $\mathcal{F}$ be the set of all nodes n such that $f(n) = f^*$. In order to get one path, an extra pass through the nodes in the set $\mathcal{F}$ is needed starting from the start node until the goal node is reached. In case of multiple optimal paths, one path is chosen arbitrarily and other paths are pronounced inactive. For example, we can choose always diagonal transition through the nodes in the set $\mathcal{F}$ starting from the start node, and choose straight transition only if diagonal transition is not possible. An example of multiple optimal paths generated by the Witkowski's algorithm in a binary occupancy grid map is shown in Fig. 3.

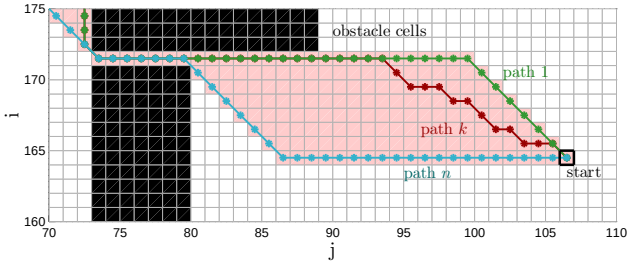

Figure 3: An example of multiple optimal paths produced by the Witkowski's algorithm in a binary occupancy grid map. There are noted only three paths among many.

The Witkowski's algorithm does not have fast replanning capability as the D* algorithm. It must execute

the whole procedure from the beginning each time the change in the environment is detected. The pseudocode of the Witkowski's algorithm execution in changing environments is given by Alg. 2.5. The robot detects the change in the environment by its sensor and builds the new graph noted as $\mathcal{G}_R$. The Witkowski's algorithm is executed again from the robot's current position (node R) to the goal node and new values f are calculated. The new set $\mathcal{F}$ is determined and next node in the optimal path to follow is chosen from that set.

Algorithm 2.5: move-robot-W($start, goal$)

```
1:  R ← start
2:  while R ≠ goal
3:      G_R ← sensor(R)
4:      if G_R ≠ G or R = start
5:          G ← G_R
6:          Witkowski(R, goal)
7:      end
8:      F ← {n | f(n) = f(goal)}
9:      R ← choose-next(R, F)
10: end
```

The main advantage of the Witkowski's algorithm is that it finds all optimal paths that exist in the search space.

If we represent all nodes from the set $\mathcal{F}$ as the group of cells which form a geometrical area, called the area of the set $\mathcal{F}$, it is possible to compose a new path as a sequence of points connected by straight lines each of which lies within that area. Since the path that passes through the border of that area is optimal for the given graph $\mathcal{G}$ created from the binary occupancy grid map, the new path which passes through the middle of the area is shorter than the optimal path in the graph.

3. The Two-Way D* algorithm

The proposed two-way D* algorithm is inspired by the Witkowski's algorithm. It also searches the graph in forward and backward passes. The difference is that the TWD* algorithm uses the D* algorithm for graph search in these two passes instead of the BFS algorithm used in the Witkowski's algorithm. The usage of forward and backward passes of the D* algorithm through the graph enables the TWD* algorithm to find optimal paths also in weighted graphs, i.e. in weighted occupancy grid maps with the cost mask. The optimal path found by the TWD* algorithm consists of straight line segments with continuous headings and is the shortest possible path in geometrical space, which is the major advantage of the TWD* algorithm with respect to the standard D* algorithm. Hereafter we first describe how the TWD* algorithm plans the initial path when the robot is standstill at start position, and then how it replans the path when the robot detects obstacles on its journey to the goal position.

3.1. Initial path planning by the two-way D algorithm*

Initially, the TWD* algorithm does the exhaustive search of the graph $\mathcal{G}(\mathcal{N}, \mathcal{E}, \mathcal{W})$ by applying two passes of the D* algorithm, one pass is executed from the goal node to the start node (standard D*) and another is executed from the start node to the goal node (the reverse D* (RD*)). For every node n in the graph $\mathcal{G}(\mathcal{N}, \mathcal{E}, \mathcal{W})$, D* calculates cost $g(n)$ to the goal node as well as $k(n)$ needed for path replanning and the backpointer $b(n)$ needed for reconstruction of the optimal path to the goal node. Analogously, RD* calculates cost $h(n)$ to the start node and $k_R(n)$ needed for path replanning and the backpointer $b_R(n)$ needed for reconstruction of the optimal path to the start node. The D* algorithm uses the set $Open$ and the RD* algorithm the set $Open_R$ to store currently examined nodes. The RD* algorithm is also given by Alg. 2.1 as the D* algorithm, except that it uses h, k_R, b_R and the set $Open_R$, instead of g, k, b and the set $Open$, respectively. Consequently, the function **insert-D*** (n, g_{min}) given by Alg. 2.3 is changed and named as **insert-RD*** (n, h_{min}).

By applying the above described two-way D* search of the graph, the TWD* algorithm finds an area of minimal path costs (area of the set $\mathcal{F}$) from the start node to the goal node either in a binary or a weighted occupancy grid map. The found area of the set $\mathcal{F}$ possesses the same characteristics as the area of the set $\mathcal{F}$ found by the Witkowski's algorithm in a binary occupancy grid map.

The idea is to partition the set $\mathcal{F}$ into smaller subsets, denoted as $\mathcal{H}_i$, $i = 1, \ldots, H$, and to find line segments inside the area of each $\mathcal{H}_i$ that form the shortest possible path $\mathcal{P}_W$ in the geometrical space, as shown in Figs. 4 and 5. Each subset $\mathcal{H}_i$ contains optimal paths in the graph $\mathcal{G}$, which are composed of exactly two directions of transition through the grid – one straight direction and one diagonal direction. Every subset $\mathcal{H}_i$ contains so called concavity points, which are always at the concavity angle of the border of the area of $\mathcal{H}_i$. Concavity points at which the area of $\mathcal{F}$ is one cell thin and through which pass all optimal

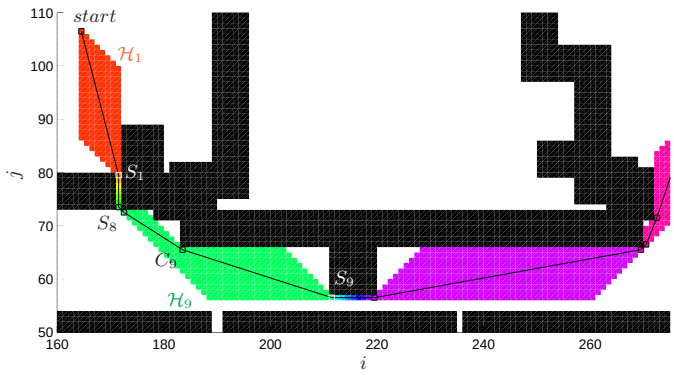

Figure 4: The sets of nodes $\mathcal{H}_i$, concavity points of type C and S and line segments that connect them created in a binary occupancy grid map.

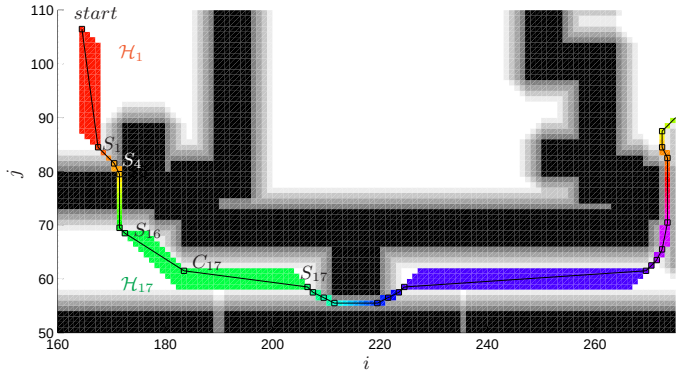

Figure 5: The sets of nodes $\mathcal{H}_i$, concavity points of type C and S and line segments that connect them created in a weighted occupancy grid map with the safety cost mask.

paths are marked as S, while other ones are marked as C. At C points the area of $\mathcal{H}_i$ can be split into convex subsets, noted as CS_k, where $k \in \{1, \ldots, 8\}$, i.e. there exist exactly eight types of convex sets depending on the optimal paths which pass through these sets. All eight possible convex sets CS_k are shown in Fig. 6.a), and splitting the set $\mathcal{H}_i$ into convex subsets is illustrated in Fig. 6.b). Every convex set CS_k is defined by two optimal transitions through the grid. Line segments of the path $\mathcal{P}_W$ connect

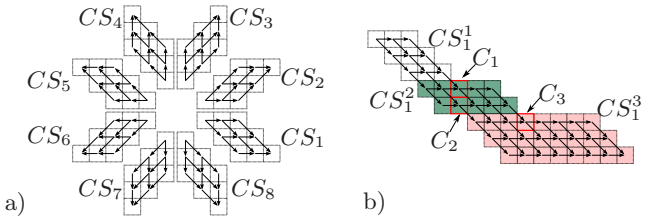

Figure 6: a) Eight possible types of convex sets CS_k, $k = 1, \ldots, 8$; b) An example of splitting the set $\mathcal{H}_i$ into the convex subsets CS_1^j, $j = 1, 2, 3$, with denoted concavity points of type C and S.

only the concavity points and lie inside the area of these convex sets. Procedure of the shortest path determination is composed of two steps: (1) determination of the subsets $\mathcal{H}_i$ and concavity points and (2) calculation of the shortest path $\mathcal{P}_W$.

3.1.1. Determination of the subsets of optimal paths and concavity points

The procedure of partitioning the set $\mathcal{F}$ into subset $\mathcal{H}_i$, $i = 1, \ldots, H$, and determining concavity points is given by Alg. 3.1. An auxiliary set $\mathcal{H}$ is used, which contains all nodes of the set $\mathcal{F}$ not included in the subsets $\mathcal{H}_i$ created in previous iterations. Thus, the set $\mathcal{H}$ is initially equal to the set $\mathcal{F}$ and from iteration to iteration the nodes are subtracted from it and added to the subsets $\mathcal{H}_i$, starting from the start node and for $\mathcal{H}_1$ ($i = 1$). The nodes from the set $\mathcal{H}$ are iteratively added to the set $\mathcal{H}_i$ if they are neighbors to the nodes already in the set $\mathcal{H}_i$ and if their cost g is less or equal to the cost of those neighbor nodes in

Algorithm 3.1: H-subsets($start, goal, \mathcal{F}$)

```
1:  H ← F, i ← 1  //Initialization
2:  H_i ← {start}
3:  S_0 ← start  //Initial S point
4:  C_{H_i} ← ∅  //Set of C points assigned to the H_i
5:  H ← H \ {start}
6:  while H ≠ ∅ or goal ∈ H
7:      N_i ← {n | n ∈ H, m ∈ H_i and {m,n} ∈ E
              and g(n) ≤ g(m)}
8:      H_i ← H_i ∪ N_i
9:      H ← H \ N_i
10:     for ∀n ∈ N_i
11:         N_n ← {m | m ∈ F and {m,n} ∈ E}
12:         if ∄m ∈ N_n such that |g(n) − g(m)| < w_{n,m}
13:             w_max ← max{w_{m,n} | m ∈ N_n}
14:             if ∄o ∈ F such that |g(n) − g(o)| < w_max
15:                 S_i ← n  //New S point
16:                 i ← i + 1
17:                 H_i ← {n}
18:                 C_{H_i} ← ∅
19:             else
20:                 C_{H_i} ← C_{H_i} ∪ {n}  //New C point
21:             end
22:         else if |N_n| ∈ {5,6} and ∃o ∉ F
                    such that ‖o − n‖ = e_cell
23:             N_o ← {r | r ∈ F and ‖r − o‖_∞ = e_cell}
24:             if |N_o| ≤ 4
25:                 C_{H_i} ← C_{H_i} ∪ {n}  //New C point
26:             end
27:         end
28:     end
29: end
```

$\mathcal{H}_i$ (line 7). This condition of decreasing g value while creating the set $\mathcal{H}_i$ assures progressing towards the goal node. This is necessary only in the replanning phase described in Section 3.2 where the start node is replaced by the node R (robot's position) which can be somewhere inside the area of the set $\mathcal{F}$. Adding nodes to the set $\mathcal{H}_i$ stops when a node is added that represents a concavity point of type S (noted as S_i). Then, nodes are added to the new set $\mathcal{H}_{i+1}$ (lines 15-17). The first S point, noted as S_0, is the *start* node, and the last S point, noted as S_H, is the *goal* node. Checking if a node n is S point or not is obtained by observing its neighbor nodes (line 11). All optimal paths pass through the S point and all neighbor nodes have g values that differ from the $g(S)$ exactly for the weight w between neighbor node and the S point. So if there is no any neighbor node m for which $|g(n) - g(m)| < w_{m,n}$ then the node n is the S point candidate. Additionally, it must be checked whether there is a node with similar cost g but distanced from the S point candidate. This occurs in symmetrical configurations in which exist two or more distanced optimal paths (see Fig. 7). For similarity test, the largest weight w_{max} between the S point candidate and

the neighbor node is used (line 13). If there is no node $o \in \mathcal{F}$ for which $|g(n) - g(m)| < w_{max}$ then the node n is stored as the point S_i, otherwise the node n is stored as the C point since not all optimal paths pass through this point. Within the set $\mathcal{H}_i$ can exist one or more concavity points of type C. Checking if a node n is C point or not is obtained by observing its neighbor nodes and the nodes around the closest cell $o \notin \mathcal{F}$ that shares a common edge with the cell n (line 23). If neighbor nodes contain two or three nodes that are not in the set $\mathcal{F}$ and if the cell o has less or equal to four nodes around, which belong to the set $\mathcal{F}$, then the node n is stored as the C point. All C points within the set $\mathcal{H}_i$ are added to the set $C_{\mathcal{H}_i}$ (line 25). The algorithm ends with the empty set $\mathcal{H}$ (line 6). The use of the additional termination condition when *goal* is not in the set $\mathcal{H}$ is for the option of stopping the creation of the subsets to some other goal point inside the set $\mathcal{F}$, as will be the case in the replanning process described in 3.2.

3.1.2. Calculation of the shortest path

The shortest path is calculated by Alg. 3.2. The straight line segments of the shortest path $\mathcal{P}_W$ are created connecting neighbor S points, i.e. S_{i-1} and S_i, for $i = 1, \ldots, H$, if between them does not exist any C point, i.e. if the set $\mathcal{H}_i$ is convex. Otherwise, creation of the path $\mathcal{P}_W$ includes C points from the set $C_{\mathcal{H}_i}$ such that the shortest Euclidean path, which lie within the area of the set $\mathcal{H}_i$, is computed between the points S_{i-1} and S_i. For this task an extra graph is created for every set $\mathcal{H}_i$ noted as $\mathcal{G}_{\mathcal{H}_i}(\mathcal{N}_{\mathcal{H}_i}, \mathcal{E}_{\mathcal{H}_i}, \mathcal{W}_{\mathcal{H}_i})$. The set of nodes is defined as $\mathcal{N}_{\mathcal{H}_i} = \{S_{i-1}, S_i\} \cup C_{\mathcal{H}_i}$, i.e. it is composed of starting and ending S points of the set $\mathcal{H}_i$ and C points between these S points, which are in the set $C_{\mathcal{H}_i}$. Edges are defined only between the visible nodes, $\mathcal{E}_{\mathcal{H}_i} = \{\{m, n\} \mid m, n \in \mathcal{N}_{\mathcal{H}_i},\ m \text{ and } n \text{ are visible}\}$. Two nodes are said to be visible to each other if the line segment that connects them does not intersect any obstacle and lies completely within the area of the set $\mathcal{H}_i$. Every edge $\{m, n\} \in \mathcal{E}_{\mathcal{H}_i}$ has assigned weight $w_{m,n} := \|m - n\|$. The graph $\mathcal{G}_{\mathcal{H}_i}$ needs to be searched to find a path $\mathcal{P}_i = \mathcal{P}_i(S_{i-1}, S_i)$ that has the smallest cost $c(\mathcal{P}_i)$ among all possible paths in the graph $\mathcal{G}_{\mathcal{H}_i}$. The path $\mathcal{P}_i$ is a list defined as:

$$\begin{aligned} \mathcal{P}_i[1] = S_{i-1}, \quad \mathcal{P}_i[|\mathcal{P}_i|] = S_i, \\ \mathcal{P}_i[k] \in \mathcal{N}_{\mathcal{H}_i}, \quad k = 1, \ldots, |\mathcal{P}_i|, \\ \{\mathcal{P}_i[l], \mathcal{P}_i[l+1]\} \in \mathcal{E}_{\mathcal{H}_i}, \quad l = 1, \ldots, |\mathcal{P}_i| - 1. \end{aligned} \tag{7}$$

Every two neighbor points in the path $\mathcal{P}_i$ represent a line segment, which lies within the area of the set $\mathcal{F}$. The cost of the path $\mathcal{P}_i$ is defined as the sum of weights of edges along the path, i.e.,

$$c(\mathcal{P}_i) := \sum_{k=1}^{|\mathcal{P}_i|-1} w_{\mathcal{P}_i[k], \mathcal{P}_i[k+1]}. \tag{8}$$

In most cases the graph $\mathcal{G}_{\mathcal{H}_i}(\mathcal{N}_{\mathcal{H}_i}, \mathcal{E}_{\mathcal{H}_i}, \mathcal{W}_{\mathcal{H}_i})$, which is needed to be searched from S_{i-1} to S_i, has small number of nodes and any graph search algorithm can be applied, e.g. the uniform cost search or the A* algorithm (lines 5-17).

Algorithm 3.2: shortest-path$(S_{i-1}, S_i, C_{\mathcal{H}_i})$

1: $\mathcal{N}_{\mathcal{H}_i} \leftarrow \{S_{i-1}, S_i\} \cup C_{\mathcal{H}_i}$
2: $\mathcal{E}_{\mathcal{H}_i} \leftarrow \{\{m, n\} \mid m, n \in \mathcal{N}_{\mathcal{H}_i}$ if **visible**$(n, m)\}$
3: $\mathcal{W}_{\mathcal{H}_i} \leftarrow \{w_{m,n} := \|m - n\| \mid \{m, n\} \in \mathcal{E}_{\mathcal{H}_i}\}$
4: $\forall n \in \mathcal{N}_{\mathcal{H}_i},\ h(n) \leftarrow \infty$ *//Initialization*
5: $Open_{\mathcal{H}_i} \leftarrow \{S_i\},\ h(S_i) \leftarrow 0$
6: while $Open_{\mathcal{H}_i} \neq \emptyset$
7: $\quad h_{min} \leftarrow \min\{h(n) \mid n \in Open_{\mathcal{H}_i}\}$
8: $\quad o \leftarrow n^*$ *//for n^* such that $h(n^*) = h_{min}$*
9: $\quad Open_{\mathcal{H}_i} \leftarrow Open_{\mathcal{H}_i} \setminus \{o\}$
10: $\quad$ for $\forall n \in \mathcal{N}_{\mathcal{H}_i}$ such that $\{o, n\} \in \mathcal{E}_{\mathcal{H}_i}$
11: $\quad\quad$ if $h(n) > h(o) + w_{n,o}$
12: $\quad\quad\quad Open_{\mathcal{H}_i} \leftarrow Open_{\mathcal{H}_i} \cup \{n\}$
13: $\quad\quad\quad h(n) \leftarrow h(o) + w_{n,o}$
14: $\quad\quad\quad b(n) \leftarrow o$
15: $\quad\quad$ end
16: $\quad$ end
17: end
18: $k \leftarrow 1,\ \mathcal{P}_i[k] \leftarrow S_{i-1}$ *//Determing $\mathcal{P}_i$*
19: while $\mathcal{P}_i[k] \neq S_i$
20: $\quad k \leftarrow k + 1$
21: $\quad \mathcal{P}_i[k] \leftarrow b(\mathcal{P}_i[k-1])$
22: end
23: return $\mathcal{P}_i$

The worst case scenario occurs if only one $\mathcal{H}_i$ set ($\mathcal{H}_1 = \mathcal{F}$) and two S points ($S_0$ and S_H) are created. In that case the graph $\mathcal{G}_{\mathcal{H}_1}$ is the only graph to be searched for calculating the shortest path $\mathcal{P}_W$, and its size will depend on the number of C points. However, the number of nodes in that graph will still be significantly smaller than the number of nodes in the graph created from the occupancy grid map.

An example of calculating the path $\mathcal{P}_i$ in the graph $\mathcal{G}_{\mathcal{H}_i}(\mathcal{N}_{\mathcal{H}_i}, \mathcal{E}_{\mathcal{H}_i}, \mathcal{W}_{\mathcal{H}_i})$ is shown in Fig. 7. The example illus-

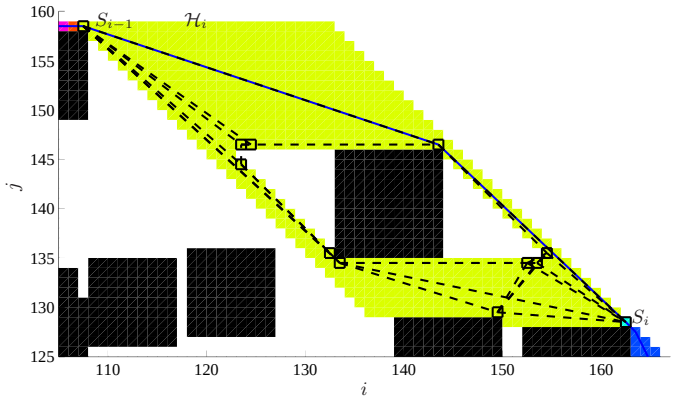

Figure 7: Calculating the path $\mathcal{P}_i$ in the graph $\mathcal{G}_{\mathcal{H}_i}(\mathcal{N}_{\mathcal{H}_i}, \mathcal{E}_{\mathcal{H}_i}, \mathcal{W}_{\mathcal{H}_i})$ for the set $\mathcal{H}_i$.

trates symmetrical obstacle configuration in which there are left and right side paths around the obstacle, which is placed between two S points, i.e. the area of the set $\mathcal{H}_i$ contains a hole. In general, the area of the set $\mathcal{H}_i$ can contain many holes and the path is calculated by the same procedure as in the case of the area with no holes. There are ten concavity points of type C noted by squares. The set $\mathcal{E}_{\mathcal{H}_i}$ contains 29 visible edges (neither goes through the hole) noted by dashed lines. The shortest path (noted by solid line) is calculated by searching the graph $\mathcal{G}_{\mathcal{H}_i}(\mathcal{N}_{\mathcal{H}_i}, \mathcal{E}_{\mathcal{H}_i}, \mathcal{W}_{\mathcal{H}_i})$.

Total path from the start node to the goal node is created by union of all paths calculated for the sets $\mathcal{H}_i$ for $i = 1, \ldots, H$

$$\mathcal{P}_W = \mathcal{P}_1 \cup \cdots \cup \mathcal{P}_H. \tag{9}$$

According to the procedure given by Alg. 3.2 the path $\mathcal{P}_W$ lies within the area of the set $\mathcal{F}$, and changes direction only in concavity points of type S and C – which are in the centers of the cells. The path is shorter than the path calculated by the D* algorithm since line segments of the path $\mathcal{P}_W$ pass through the middle of the area of the set $\mathcal{F}$, and the path calculated by the D* algorithm passes through the border of that area. It is obvious that the path $\mathcal{P}_W$ is the shortest path among all paths that pass only through the area of the set $\mathcal{F}$, under the condition that the points of changing the path direction are in the center of the grid cells.

3.2. Path replanning by the TWD algorithm*

If nodes in vicinity of the moving robot change their occupancies, the path replanning process is initiated in order to find the new shortest path $\mathcal{P}_W$ to the goal node. The TWD* algorithm replans the path by sequential execution of the D* and RD* searches of the changed graph $\mathcal{G}(\mathcal{N}, \mathcal{E}, \mathcal{W})$ and by the recalculation of the shortest path from the robot's current position (node R) to the goal node. The D* algorithm searches the graph from the detected changed nodes to the node R producing the new path $\mathcal{P}_{D^*}$ from the node R to the goal node. If the RD* algorithm searches the graph from the changed nodes to the goal node, the area of the set $\mathcal{F}$ from the node R to the goal node can be found and the new shortest path $\mathcal{P}_W$ can be calculated as described in section 3.1.2. However, the computational cost of such RD* search can be very high and can be lowered if the RD* search terminates at a subgoal node which is closer to the robot then the goal node but still enables calculation of the new shortest path $\mathcal{P}_W$ from the robot's current position to the goal node. The procedures of the subgoal node determination and of the shortest path recalculation are presented in sections 3.2.1 and 3.2.2, respectively.

3.2.1. Determination of the subgoal node

The subgoal node is needed for the execution of the RD* algorithm and therefore it is necessary to find a procedure for its determination based only on the new path $\mathcal{P}_{D^*}$ calculated by the D* algorithm. To ensure the path optimality the new shortest path $\mathcal{P}_W$ must also pass through the subgoal node. Since both optimal paths ($\mathcal{P}_{D^*}$ and $\mathcal{P}_W$) must pass through the S concavity points these points are natural candidates for the subgoal node. The S point nearest to the robot, noted as S_R, is chosen as the subgoal node. It is found in the part of the new path $\mathcal{P}_{D^*}$ behind the changed nodes that is completely aligned with the old path $\mathcal{P}_{D^*}$ ($\mathcal{P}_{oldD^*}$) calculated before the change in the environment happened. By finding the S_R node it is necessary to compute only a part of the new path $\mathcal{P}_W$ between the nodes R and S_R and connect it with the remaining part of the old path $\mathcal{P}_W$ between the S_R node and the goal node.

The S points are located between two consecutive sets $\mathcal{H}_i$ and $\mathcal{H}_j$, $i < j \leq H$, that contain more than two nodes. Assume that $\mathcal{H}_i$ contains convex sets of type CS_k, and $\mathcal{H}_j$ contains convex sets of type CS_l, where $k, l \in \{1, \ldots, 8\}$. There are two cases of location of the S point: (1) between two convex sets of different type CS_k and CS_l, $l \neq k$ (e.g. point S_1 in Fig. 8 is between convex sets of types CS_8 and CS_1) and (2) between two convex sets of the same type, CS_k and CS_l, where $k = l$, if the non-free cells ($o(i) > 1$) are at both sides of the path between the sets $\mathcal{H}_i$ and $\mathcal{H}_j$ (e.g. point S_n in Fig. 8 is between two convex sets of type CS_2). Since the path $\mathcal{P}_{D^*}$ is also within the area of the

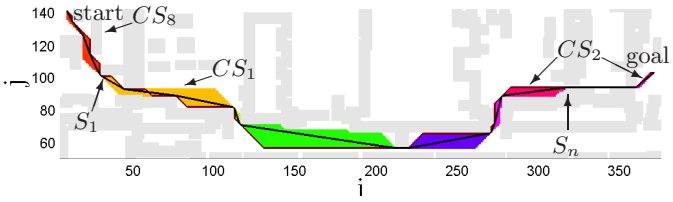

Figure 8: The location of concavity points of type S according to the convex sets CS_k in the set $\mathcal{F}$.

set $\mathcal{F}$, it can be determined to which type of convex set CS_k, $k \in \{1, \ldots, 8\}$ a certain part of the path belongs, by analyzing the transitions in the path. It can also be determined which cells of the path $\mathcal{P}_{D^*}$ are near the edges of the non-free cells by analyzing the surrounding cells of each cell in the path. Therefore, it is possible to determine positions of the S points by using only $\mathcal{P}_{D^*}$ and occupancy grid map.

The procedure of determining the position of S_R point in the path $\mathcal{P}_{D^*}$ is given by Alg. 3.3. A sequence of transitions in the path with the same orientation is called the path line segment. Path line segments are examined from the point after both paths $\mathcal{P}_{D^*}$ and $\mathcal{P}_{oldD^*}$ are aligned (lines 2-6). The type of a convex set CS_k, $k \in \{1, \ldots, 8\}$ to which the part of the path $\mathcal{P}_{D^*}$ belongs is determined by orientations of the three consecutive path line segments (noted as α_1, α_2 and α_3 in lines 8, 14 and 21, respectively). The notation $\measuredangle(a, b)$ is used as a representation of the direction of the line segment defined by points a and

Algorithm 3.3: determine-S_R-point($\mathcal{P}_{D^*}, \mathcal{P}_{oldD^*}$)

```
1:  if P_oldD* = ∅ return P_D*[|P_D*|]  //The goal node
2:  i ← |P_D*|, j ← |P_oldD*|
3:  while P_D*[i] = P_oldD*[j] and i, j > 1
4:      i ← i − 1, j ← j − 1
5:  end
6:  i_align ← i + 1  //The point of path alignment
7:  if i_align = |P_D*| return P_D*[i_align]
8:  α1 ← ∠(P_D*[i_align], P_D*[i_align − 1])
9:  i ← i_align + 1
10: while i < |P_D*| and ∠(P_D*[i], P_D*[i − 1]) = α1
11:     i ← i + 1
12: end
13: if i = |P_D*| return P_D*[i]
14: α2 ← ∠(P_D*[i], P_D*[i − 1])
15: i2 ← i − 1  //Beginning of the 2nd line seqment
16: while i < |P_D*|
17:     while i < |P_D*| and ∠(P_D*[i], P_D*[i − 1]) = α2
18:         i ← i + 1
19:     end
20:     if i = |P_D*| return P_D*[i]
21:     α3 ← ∠(P_D*[i], P_D*[i − 1])
22:     i3 ← i − 1  //Beginning of the 3rd line seqment
23:     x ← x | {x, P_D*[i2]} ∈ E and
          {x, P_D*[i2 − 1]} ∈ E and {x, P_D*[i2 + 1]} ∈ E
24:     β ← ∠(x, P_D*[i2])  //Checked neighbor cell
25:     j ← i2
26:     while o(x) = 1 and j < i3
27:         j ← j + 1
28:         x ← x | ‖x − P_D*[j]‖ = e_cell and
              ∠(x, P_D*[j]) = β
29:     end
30:     if j < i3
31:         if α3 ≠ α1 return P_D*[j]  //The case (1)
32:         k ← i2  //Checking the case (2)
33:         while k < i3
34:             y ← y | ‖y − P_D*[k]‖ = e_cell and
                  ∠(y, P_D*[k]) = −β
35:             if o(y) > 1 return P_D*[min(j, k)]
36:         end
37:     end
38:     α1 ← α2, α2 ← α3, i2 ← i3
39:     i ← i3  //Examine next path line segment
40: end
```

b pointed to the point b according to the positive x axis. According to the orientation of the third path line segment in relation to the first path line segment it is determined whether the third path line segment is still in the convex set of the same type CS_k or not.

Case (1). If the orientation of the third path line segment is not equal to the orientation of the first path line segment, then the third path line segment is within another convex set CS_l, $l \neq k$, and then a concavity point of the type S exists in the second path line segment (line 31). The procedure of finding the S_R point in the second line segment of the path $\mathcal{P}_{D^*}$ is illustrated in Fig. 9. First, the cell marked as x is determined as a neighbor of the path cell in which the path changes direction at the convex side of the angle that form the first and the second path line segments (line 23). The angle β is determined as $\measuredangle(x, \mathcal{P}[i_2])$ (line 24), where index i_2 is the beginning of the second path line segment. Then x is moved along the second path line segment to the other neighbors of the cells in the path at the same side (for example, all left neighbor cells in Fig. 9, i.e. all neighbor cells that form the same angle β with the nodes in the path, where $\beta = \measuredangle(x, \mathcal{P}[i]) = 90°$, line 28). The first found cell of the second path line segment that has non-free checked neighbor cell is the S_R point (lines 25-29).

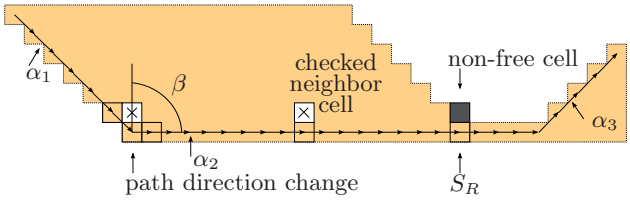

Figure 9: Determination of the point S_R in the path $\mathcal{P}_{D^*}$: the case (1).

Case (2). If the orientation of the third path line segment is equal to the orientation of the first path line segment, then the third segment is within the same convex set CS_k, and a concavity point of the type S may exist in the second path line segment, but does not have to (line 32). The procedure of determining the S_R point in the path $\mathcal{P}_{D^*}$ is based on consecutive checking along the second path line segment is there the non-free cell at both side of the path. In Alg. 3.3, a neighbor cell y at the oposite side of the path with respect to the cell x is discriminated by the angle $-\beta$ ($-90°$ in Fig. 9) that it forms with the node in the second path line segment (line 34). The S_R point is the first point near the edge of the non-free cell in that path line segment (lines 32-36). If there is no S point in the second path line segment then the new search for the S point is started, where the second path line segment becomes the first path line segment and consecutive two path line segments become the second and the third path line segments, respectively (line 38).

3.2.2. Recalculation of the shortest path

The RD* algorithm is invoked as Alg. 2.1 with the argument R set to the determined point S_R. It means that nodes are searched from the newly changed nodes towards the node S_R, until the cost h of the node S_R is less than the cost k_R of the best node in the set $Open_R$. Then, by calculating $f(n)$, the new area of the set $\mathcal{F}$ is determined. In most cases the node R stays in the new area of the set $\mathcal{F}$. However, the change in the environment can be such that the node R is not in the new area of the set $\mathcal{F}$. That

is due the fact that the value of f is optimal according to the start node and the goal node, but not according to the robot's current position and the goal node.

There is no guaranty that the node R will be in the area of the new set $\mathcal{F}$ even if the replanning by D* is extended to the start node and replanning by RD* to the goal node. Therefore, if the node R is not in the new set $\mathcal{F}$ it is necessary to find a way to connect it with the area of the set $\mathcal{F}$ without loss of optimality. Since the D* algorithm has calculated the new optimal path $\mathcal{P}_{D^*}$ from the node R, this problem is solved by finding the closest node in the path $\mathcal{P}_{D^*}$, noted as n_W, which is in the set $\mathcal{F}$. Node n_W becomes the starting node for the new path $\mathcal{P}_W$ calculation. The sets $\mathcal{H}_i$ are formed from n_W to S_R. The new shortest path $\mathcal{P}'_W$ is calculated from n_W to S_R by the procedure described in 3.1.2.

Algorithm 3.4: connect-paths($\mathcal{P}'_W, \mathcal{P}_{oldW}, \mathcal{P}_{D^*}$)

```
1:  if P_oldW = ∅ return P'_W
2:  i ← 1, S_R ← P'_W[1], goal ← P_oldW[|P_oldW|]
3:  while g(P_oldW[i]) > g(S_R)
4:      i ← i + 1
5:  end
6:  n_x ← P_oldW[i]
7:  if visible(S_R, n_x) return
8:      P_D*(R, n_W) ∪ P'_W ∪ P_oldW(n_x, goal)
9:  while P_oldW[i] ∉ P_D*
10:     i ← i + 1
11: end
12: n_y ← P_oldW[i]
13: return P_D*(R, n_W) ∪ P'_W ∪ P_D*(S_R, n_y) ∪
        P_oldW(n_y, goal)
```

The final step is the connection of the paths $\mathcal{P}_{D^*}(R, n_w)$ and $\mathcal{P}'_W$ with the remaining part of the old path $\mathcal{P}_W$, noted as $\mathcal{P}_{oldW}$. The procedure of connecting these paths is given by Alg. 3.4. The first node in the path $\mathcal{P}_{oldW}$ with the cost g less or equal to the cost $g(S_R)$, noted as n_x, is searched (lines 2-6). Before simple concatenation of the paths $\mathcal{P}'_W$ and $\mathcal{P}_W(n_x, goal)$, the visibility between nodes S_R and n_x is checked. If nodes S_R and n_x are not visible then the part of the new path $\mathcal{P}_{D^*}$ is inserted from the node S_R to the first common node on both $\mathcal{P}_{D^*}$ and $\mathcal{P}_{oldW}$ paths, noted as n_y (lines 9-12). Complete path from the robot's position to the goal node is determined as the union of paths: $\mathcal{P}_{D^*}(R, n_W) \cup \mathcal{P}'_W \cup \mathcal{P}_{oldW}(n_x, goal)$ if nodes S_R and n_x are visible (line 8) or $\mathcal{P}_{D^*}(R, n_W) \cup \mathcal{P}'_W \cup \mathcal{P}_{D^*}(S_R, n_y) \cup \mathcal{P}_{oldW}(n_y, goal)$ if nodes S_R and n_x are not visible (line 13).

3.3. Overall TWD algorithm execution*

The pseudocode of the overall TWD* algorithm execution during the robot motion from the start node to the goal node is given by the Alg. 3.5. The TWD* algorithm initial execution is if $R = start$, and further executions are in the case of change in the environment that are detected

Algorithm 3.5: move-robot-TWD*($start, goal$)

```
1:  R ← start
2:  ∀n ∈ N, k(n) ← ∞, g(n) ← ∞, b(n) ← n
    k_R(n) ← ∞, h(n) ← ∞, b_R(n) ← n
3:  Open ← ∅, Open_R ← ∅
    P_oldW ← ∅, P_oldD* ← ∅  //Initialization
4:  insert-D*(goal, 0)
5:  insert-RD*(start, 0)
6:  while R ≠ goal
7:      I_R ←changed-nodes(R)
8:      if R = start or I_R \ Open ≠ ∅
9:          D*(I_R, R)  //Initial planning or replanning
10:         i ← 1, P_D*[i] ← R  //Determing P_D*
11:         while P_D*[i] ≠ goal
12:             i ← i + 1
13:             P_D*[i] ← b(P_D*[i − 1])
14:         end
15:         S_R ←determine-S_R-point(P_D*, P_oldD*)
16:         RD*(I_R, S_R)
17:         F ← {n | g(n) + h(n) = g(S_R) + h(S_R)}
18:         i ← 1
19:         while P_D*[i] ∉ F
20:             i ← i + 1
21:         end
22:         n_W ← P_D*[i]
23:         H-subsets(n_W, S_R, F)
24:         for ∀H_i, i = 1, ..., H
25:             P_i ←shortest-path(S_{i−1}, S_i, C_{H_i})
26:         end
27:         P'_W ← P_1 ∪ ··· ∪ P_H
28:         P_W ← connect-paths(P'_W, P_oldW, P_D*)
29:         P_oldW ← P_W, P_oldD* ← P_D*
30:     end
31:     R ←path-following(P_W)
32: end
```

by the robot's sensors. The function **changed-nodes** in line 7 returns all nodes that have changed occupancy values and those nodes are further handled by the D* and RD* algorithms. When D* finishes the execution the new optimal path $\mathcal{P}_{D^*}$ is determined by the backpointer function b (lines 10-14). The point S_R is determined in the new path $\mathcal{P}_{D^*}$ by the function **determine-S_R-point** given by Alg. 3.3. Then, by the RD* algorithm the path costs h are calculated and the set $\mathcal{F}$ is formed. The node n_W from which the sets $\mathcal{H}_i$ are calculated is determined in lines 18-22. Details of the function **H-subsets** are given by Alg. 3.1. The sets $\mathcal{H}_i$, $i = 1, \ldots, H$ and concavity points of type S and C are treated as hidden variables, which are used in the creation of the shortest paths $\mathcal{P}_i$ within every set $\mathcal{H}_i$ (lines 24-27). Details of the function **shortest-path** are given by Alg. 3.2. The new shortest path $\mathcal{P}'_W$ is connected to the remaining part of the old path $\mathcal{P}_{oldW}$ by the function **connect-paths** given by Alg. 3.4. The complete path $\mathcal{P}_W$ is followed by the robot till the goal node or is calcu-

lated again if a new change in the environment is detected. The function **path-following** can be any path following algorithm that takes into account robot's kinematic and dynamic constraints [18, 19, 20]. We use dynamic window based algorithm described in our previous work [21].

3.4. An illustrative example

Hereafter we illustrate on a simple example how the TWD* algorithm plans and replans the path in weighted occupancy grid maps. Both initial planning and replanning steps are presented graphically in corresponding figures, where only part of the environment around the robot is shown for the sake of clarity of presentation.

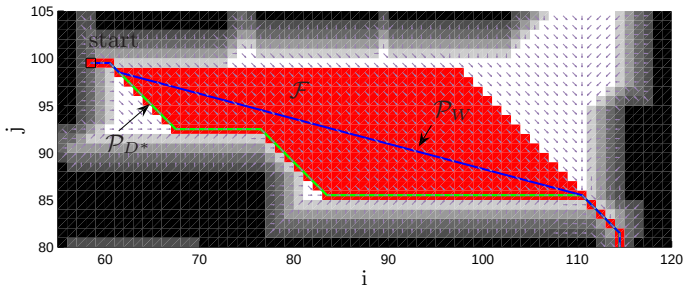

Figure 10: Initial planning: backpointers by D*, the path $\mathcal{P}_{D^*}$, the area of the set $\mathcal{F}$ calculated by the algorithm TWD*, and the path $\mathcal{P}_W$.

Fig. 10 shows results of the initial planning by the TWD* algorithm, where the set $\mathcal{F}$ and the path $\mathcal{P}_W$ were created by Algs. 3.1 and 3.2. The backpointers of the D* algorithm, which determine the optimal paths from every node to the goal node and the optimal path $\mathcal{P}_{D^*}$ are also shown.

Fig. 11 shows replanning by the D* algorithm and detection of the point S_R. The robot follows the initial path

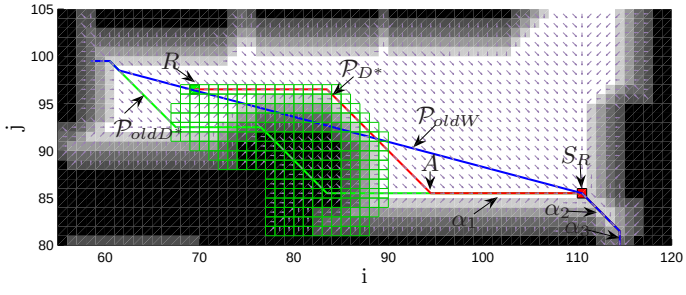

Figure 11: Replanning by D*: backpointers by D*, □ – all searched nodes, the new path $\mathcal{P}_{D^*}$ and determined point S_R.

$\mathcal{P}_W$ (noted as $\mathcal{P}_{oldW}$) and when it reaches the position noted as R it detects a new obstacle shown as a group of black cells. The new obstacle is enlarged to account for the robot's dimensions and the safety cost mask is created around it. All searched nodes by the D* algorithm in the replanning process are presented by squares. The number of searched nodes is determined by the cost of the node R as all nodes with cost k less than $g(R)$ are searched. Path costs g and backpointers directions are corrected for these nodes to form new optimal paths to the goal node. The new path $\mathcal{P}_{D^*}$ is obtained by following the pointers from the node R to the goal node. The point S_R is searched in the new path $\mathcal{P}_{D^*}$ from the point after which the new path $\mathcal{P}_{D^*}$ and the old path $\mathcal{P}_{oldD^*}$ completely align (point A in Fig. 11). Orientations of the path line segments are examined starting from the point A by Alg. 3.3 (α_1, α_2 and α_3 in Fig. 11). The orientation of the third path line segment is not equal to the orientation of the first path line segment (case (1)) so the S point must be in the second path line segment. The neighbor cell at the convex side of the angle that forms the first and the second path line segments is checked for every node in the second path line segment (right neighbor to every cell in the second path line segment). The node S_R is the first node in the second path line segment that has non-free checked neighbor cell.

Fig. 12 shows replanning by the RD* algorithm. All

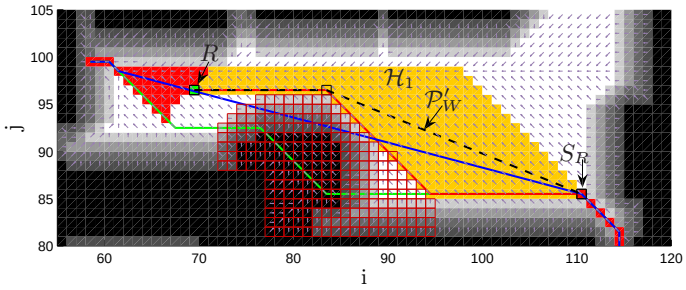

Figure 12: Replanning by RD*: backpointers by RD*, □ – all searched nodes, the area of the set $\mathcal{H}_1$ of the TWD* algorithm and the new path $\mathcal{P}'_W$.

searched nodes by the RD* algorithm in the replanning phase are presented by squares. The number of searched nodes is determined by the cost of the node S_R such that all nodes that have cost k_R less than $h(S_R)$ are searched. Path costs h and backpointers directions are corrected for these nodes to form new optimal paths to the start node. When the RD* is finished, the new set $\mathcal{F}$ is formed and shown by filled squares. The set $\mathcal{H}_1$ is formed from the node R and shown by light gray (orange) color of the filled squares. The shortest path $\mathcal{P}'_W$ is calculated from the node R to the node S_R. The complete shortest path from the robot's current position (node R) to the goal node is created executing Alg. 3.4, which connects the shortest path $\mathcal{P}'_W$ from the node R to the node S_R and the old path $\mathcal{P}_{oldW}$ from the node S_R to the goal node.

In the above described replanning case presented in Figs. 11 and 12, the node R was in the area of the new set $\mathcal{F}$ generated after the replanning by the D* and RD* algorithms had finished and therefore it was straightforward to create the new shortest path $\mathcal{P}_W$ to the goal node. Fig. 13 shows the case when the node R is outside of the new set $\mathcal{F}$ after the replanning by the D* and RD* algorithms. The node R is connected by the new optimal path $\mathcal{P}_{D^*}$ to the node n_W, which is the closest node is in the new set $\mathcal{F}$ found by simply following the backpointers from the

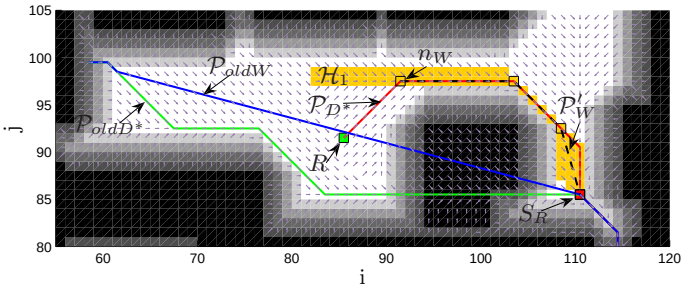

Figure 13: Replanning case: the robot's current position is out of the area of the set $\mathcal{F}$.

node R. Then the set $\mathcal{H}_1$ is formed from the node n_W to the node S_R and the shortest path $\mathcal{P}'_W$ is calculated from the node n_W to the node S_R. The complete path from the robot's position to the goal node is created executing Alg. 3.4, which connects the path $\mathcal{P}_{D^*}$ from the node R to the node n_W, the new path $\mathcal{P}'_W$ from the node n_W to the node S_R, and the remaining part of the old path $\mathcal{P}_{oldW}$ from the node S_R to the goal node.

4. Test Results

In order to validate the proposed two-way D* algorithm, carefully chosen simulation and experimental tests were performed on a Pioneer 3DX mobile robot. The simulation tests were performed in three different-sized randomly generated grid maps and the experimental tests were performed at our Department. The laser range finder SICK LMS200 mounted on the robot was used for environment perception. It scans the environment in radial range of $\pm\frac{\pi}{2}$ with resolution of 1° and sends to the robot 181 uniformly distributed distances to the detected obstacles every 100 ms.

The proposed TWD* algorithm is compared to the standard D* algorithm and Witkowski's algorithm according to the computation time and a number of other important path characteristics. The TWD* and D* algorithms are executed in weighted occupancy grid maps with the safety cost mask, and the Witkowski's algorithm is executed in binary occupancy grid maps, due to its incapability of finding optimal paths in weighted occupancy grid maps. Since the Witkowski's algorithm does not have fast replanning capability as D* and TWD*, it is executed from the beginning each time the unknown nodes are detected (Alg. 2.5). The algorithms were implemented in *Player/Stage* (a free software tool for robot and sensor applications, `www.playerstage.sourceforge.net`) on a notebook with Intel(R) Core(TM)2 Duo 2.00 GHz processor. In all tests, the AMCL algorithm (Adaptive Monte Carlo Localization) was used for robot localization and a dynamic window based algorithm, described in our previous work [21], was used for path following.

4.1. Simulation results

The simulations were performed in three different-sized randomly generated square-shaped environments. The geometrical sizes of the environments were 20×20 m^2, 50×50 m^2 and 150×150 m^2 and they consist of 10^4, 10^5, and 10^6 searchable nodes (cells), respectively. Each environment was an occupancy grid map with the cell size of $e_{cell} = 0.1$ m and of about 50% occupied cells. Obstacles were created by drawing the squares at random places and of random sizes. The center of the obstacle square could be at any cell of the grid map and the half of length of the obstacle edge could be any number of cells less or equal to the predefined value that we chose to be 5 cell sizes. Obstacles were also enlarged to account for the robot's dimensions. Further, 50% occupied cells were chosen randomly to be unknown obstacles by the same procedure as for drawing the squares at random places and of random sizes and all occupied cells within the squares were set to be free. The robot could detect by it's sensor that these nodes were occupied when it reached their vicinity. The start and goal nodes were placed always in the center of the left and right border of the environment, respectively. Fig. 14 shows an example of obstacle configurations in the environment of 20×20 m^2 with approximately 10^4 nodes and initial paths computed by the TWD* and D* algorithms. For each environment size, the algorithms were executed and compared on ten randomly-generated obstacle configurations, and the best (min), the worst (max) and the averaged (ave) results are given hereafter.

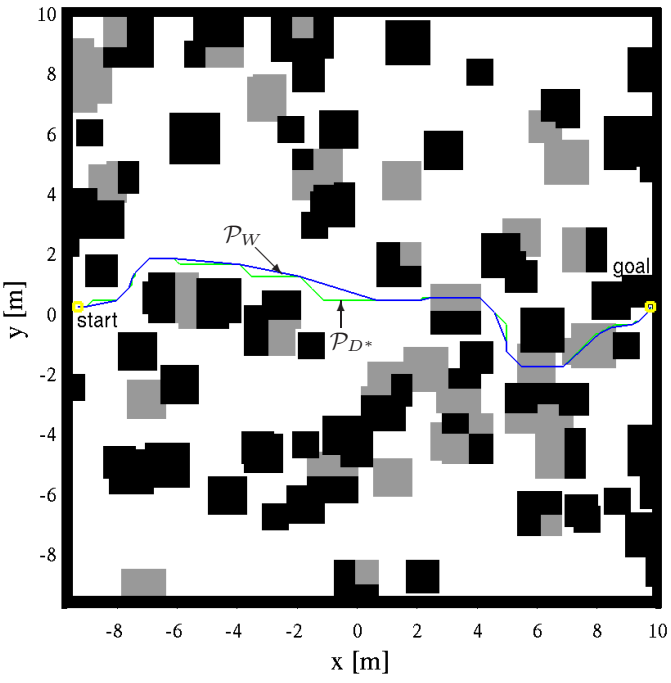

Figure 14: A random environment of 20×20 m^2 with 10^4 nodes and the initial paths computed by the TWD* and D* algorithms. The known obstacles are shown in black squares and the unknown obstacles in gray squares.

Table 1: Planning times in milliseconds and numbers of explored nodes for the Witkowski's, D* and TWD* algorithms in three different-sized randomly generated environments comprising of about 10^4, 10^5 and 10^6 nodes. t_{init}, t_{pmax} and $\sum t_p$ denote initial planning time, maximal replanning time and the sum of all replanning times, respectively. The number of explored nodes E is given in round brackets next to the corresponding planning time.

			Witkowski (Alg. 2.4)	D* (Alg. 2.1)	TWD*=D*+RD*+$\mathcal{P}_W$		
					RD* (Alg. 2.1)	$\mathcal{P}_W$ (Alg. 3.2)	TWD* (Alg. 3.5)
Environment of 10^4 nodes	t_{init} (E)	min	34 (82,987)	180 (33,991)	213 (33,891)	0	405 (67,882)
		max	57 (92,807)	259 (34,944)	326 (34,844)	2	585 (69,768)
		ave	43 (86,156)	195 (34,598)	253 (34,585)	1	449 (69,182)
	t_{pmax} (E)	min	41 (80,618)	2 (1,207)	5 (3,348)	1	16 (5,633)
		max	62 (92,405)	50 (7,003)	60 (11,320)	9	86 (18,306)
		ave	50 (85,576)	17 (3,887)	29 (7,713)	4	43 (10,304)
	$\sum t_p$ (E)	min	1,235 (2,225,608)	13 (14,474)	17 (14,008)	3	85 (28,482)
		max	2,870 (5,385,873)	128 (55,657)	107 (41,720)	65	247 (82,314)
		ave	1,815 (3,445,245)	55 (26,438)	67 (26,565)	31	153 (53,003)
Environment of 10^5 nodes	t_{init} (E)	min	228 (564,395)	3,222 (222,056)	3,117 (221,769)	3	6,447 (443,842)
		max	270 (610,416)	3,897 (224,486)	3,403 (224,582)	8	7,250 (449,068)
		ave	246 (584,005)	3,746 (223,277)	3,258 (223,262)	5	7,010 (446,539)
	t_{pmax} (E)	min	263 (555,850)	5 (2,535)	20 (4,710)	3	25 (8,530)
		max	375 (612,602)	40 (7,317)	450 (43,574)	10	468 (47,361)
		ave	298 (581,584)	17 (4,656)	141 (19,062)	6	150 (21,462)
	$\sum t_p$ (E)	min	10,819 (19,344,236)	44 (38,670)	61 (33,869)	38	242 (92,204)
		max	41,483 (71,706,671)	215 (107,873)	944 (167,815)	330	1,089 (275,688)
		ave	23,888 (45,342,383)	129 (69,920)	380 (93,247)	137	646 (163,167)
Environment of 10^6 nodes	t_{init} (E)	min	2,251 (5,133,238)	85,695 (1,906,344)	84,563 (1,906,297)	11	171,901 (3,812,641)
		max	3,205 (7,270,552)	92,859 (1,917,616)	91,741 (1,916,570)	33	183,732 (3,834,186)
		ave	2,516 (5,597,236)	88,839 (1,912,235)	88,554 (1,912,206)	18	177,412 (3,824,441)
	t_{pmax} (E)	min	3,476 (5,284,687)	28 (6,449)	798 (70,254)	17	819 (72,307)
		max	10,346 (7,270,433)	234 (14,671)	15,824 (516,423)	87	15,891 (518,898)
		ave	5,541 (5,710,197)	85 (9,070)	5,077 (232,362)	52	5,126 (235,292)
	$\sum t_p$ (E)	min	429,991 (612,778,595)	690 (276,607)	5,720 (670,570)	646	7,056 (965,882)
		max	722,278 (895,919,304)	1,352 (389,464)	25,801 (1,477,285)	18,598	42,826 (1,866,492)
		ave	541,596 (755,269,018)	1,102 (346,178)	14,402 (1,113,905)	5,268	20,772 (1,460,083)

The planning times in milliseconds and numbers of explored nodes of the Witkowski's, D* and TWD* algorithms in all three different-sized randomly generated environments are shown in Table 1. The initial planning time t_{init} is the time needed to calculate the path when the robot is standstill at the start node. The maximal replaning time t_{pmax} is the longest planning time during the robot motion from the start node to the goal node. The sum of all replanning times $\sum t_p$ is the total planning time during the robot motion from the start node to the goal node. The number of explored nodes E is given in round brackets next to the corresponding planning time. According to the total replanning times $\sum t_p$ the D* and TWD* algorithms outperform the Witkowski's algorithm for a factor that increases as the size of the environment increases. But, the Witkowski's algorithm has lower initial planning time than the D* and TWD* algorithms, although the Witkowski's algorithm has much higher number of explored nodes than the both D* and TWD* algorithms. This result is expected since the Witkowski's algorithm does not have extra computations in ordering the nodes to find the best one which slows down computations in D* and TWD* (Note: Witkowski's algorithm is executed in binary occupancy grid maps). All planning times together with the number of explored nodes clearly indicate that the TWD* algorithm is computationally more complex than the D* algorithm. This is expected since TWD* planning includes D* planning followed by RD* planning and calculation of the shortest path $\mathcal{P}_W$ through the area of the set $\mathcal{F}$. For environments of 10^4 nodes D* and RD* have similar performances. But for larger environments RD* calculates much longer than D* since the

Table 2: Characteristics of the paths calculated by the D* and TWD* algorithms in three different-sized randomly generated environments comprising of about 10^4, 10^5 and 10^6 nodes. L_{init}, $n\alpha_{init}$ and $\sum \alpha_{init}$ denote lengths of the initial paths, the number of points in which the initial path changes direction, and the sum of all angles of the initial path direction changes, respectively.

		Environment of 10^4 nodes		Environment of 10^5 nodes		Environment of 10^6 nodes	
		D*	TWD*	D*	TWD*	D*	TWD*
L_{init} [m]	min	20.33	19.97	51.95	50.37	162.32	157.73
	max	23.71	23.13	57.74	55.25	177.68	169.43
	ave	22.21	21.65	54.75	52.64	172.59	165.29
$n\alpha_{init}$	min	8	6	27	19	87	64
	max	32	23	48	29	132	101
	ave	17	14	36	25	118	84
$\sum \alpha_{init}$ [°]	min	360	154.5	1,260	383.2	3,960	1,290.2
	max	1,440	575.3	2,250	790.7	5,940	2,370.5
	ave	801	347.4	1,670	581.2	5,310	1,803.2

changed nodes in the map from which D* searches are always within the sensor range from the robot, but the subgoal point S_R from which RD* searches can be much further in the map which indicates much longer calculations for RD*.

Characteristics of the paths calculated by the D* and TWD* algorithms in all three different-sized randomly generated environments are shown in Table 2. The length of the initial path noted as L_{init} is calculated as the sum of Euclidean distances between consecutive points in the path. The path computed by the TWD* algorithm is always shorter than the path computed by the D* algorithm for approximately 4%, not only initially as can be seen in Table 2, but also in all replanning phases as can be seen in Fig. 15 for a simulation environment of 10^4 nodes. The values $n\alpha_{init}$ and $\sum \alpha_{init}$ show that the TWD* algorithm has approximately 30% lower number of points in which the path changes direction and 60% lower the sum of all absolute angles of path direction changes than the D* algorithm, respectively. These facts indicate that a mobile robot can follow the path computed by TWD* algorithm faster then the path computed by D* algorithm. With our path following implementation described in [21] the robot follows the path computed by the TWD* algorithm approximately 20% faster than the path computed by the D* algorithm.

4.2. Experimental results

The experiments show a case of robot navigation in a completely known environment with eight strategically placed unknown obstacles (O_1-O_8), see Fig. 16. The CAD model of the environment is used to create the weighted occupancy grid map of the environment. The size of the grid map is 539 x 164 cells, where the size of the cell is $e_{cell} = 0.1$ m. The environment is comprised of approximately 10^4 searchable nodes.

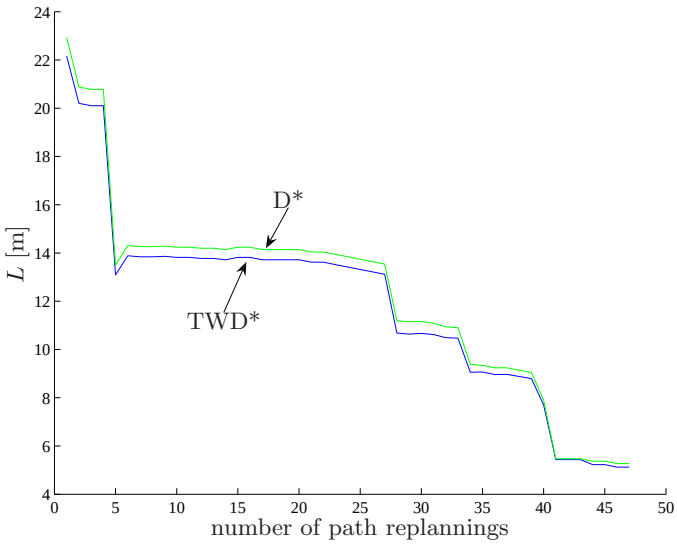

Figure 15: The lengths of the path during all replanning phases from the start to the goal in a simulation environment of 10^4 nodes.

Fig. 16 shows a part of the experimental environment where the start and the goal node are specified. The initial paths calculated by the TWD* and D* algorithms both go through obstacles O_1 and O_2. The paths $\mathcal{P}_W$ and $\mathcal{P}_{D^*}$ change as the robot detects obstacles O_1-O_8. The resulting robot's trajectory is shown, which is produced by following the path $\mathcal{P}_W$.

The initial planning time and replanning times in milliseconds t_p for the D* and TWD* algorithms in avoiding of all eight obstacles with corresponding numbers of explored nodes E in the experimental environment are shown in Table 3. Planning times clearly indicate that the TWD* algorithm is computationally more complex than the D* algorithm since TWD* planning includes D* planning followed by RD* planning and calculation of the shortest path $\mathcal{P}_W$ through the area of the set $\mathcal{F}$. The worst case

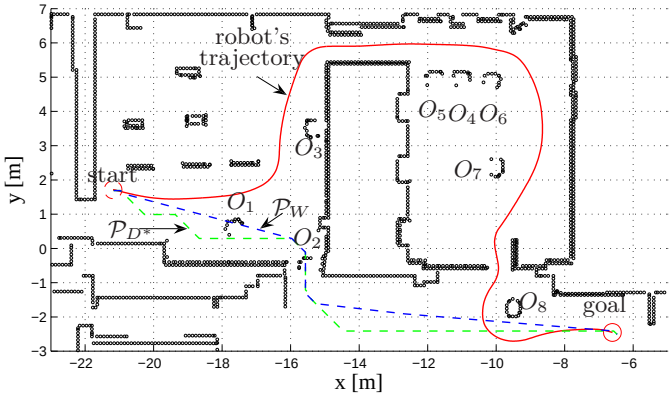

Figure 16: Robot navigation in the experimental environment avoiding eight unknown obstacles (O_1-O_8) placed on its way to the goal node.

Table 3: Initial planning time and replanning times in milliseconds for the D* and TWD* algorithms with corresponding numbers of explored nodes E in the experimental environment avoiding eight unknown obstacles (O_1-O_8).

	D* t_p (E)	RD* t_p (E)	$\mathcal{P}_W$ t_p	TWD* t_p
Initially	206 (42420)	200 (42485)	1	407
O_1	8 (1437)	3 (823)	1	12
O_2	48 (9392)	91 (17533)	3	142
O_3	2 (143)	0 (247)	1	3
O_4	6 (1056)	23 (6396)	2	31
O_5	2 (381)	0 (157)	0	2
O_6	7 (1126)	11 (3860)	0	18
O_7	6 (1248)	2 (846)	1	9
O_8	11 (2783)	27 (7602)	0	38

replanning scenario occurred in avoiding the obstacle O_2 because it is placed in the doorway and the robot could not simply go around it. Instead, the robot took the path through another door of the room. The expanded nodes are shown in Fig. 17. Big number of expanded nodes and

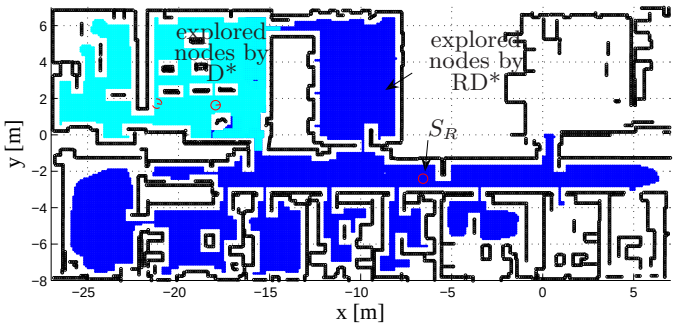

Figure 17: The nodes expanded by the D* and RD* algorithms in avoiding obstacle O_2.

resulting long replanning time by the RD* algorithm are caused by the fact that the node S_R is far away from the obstacle O_2, i.e., almost at the goal position.

4.3. Shortening the replanning times of the TWD algorithm*

The replanning time of the TWD* algorithm can be limited by hierarchical arrangement of the search graph, which enables hierarchical path planning. A suitable choice is hierarchical decomposition of the map based on hierarchical graphs [22, 23]. Hierarchies of abstraction can reduce exponential complexity problems to linear ones [24]. In our previous work [25] optimal hierarchical planning is proposed as an extension of the hierarchical D* algorithm of Cagigas [26]. The optimality of the global path is obtained by optimal placement of the so-called bridge nodes needed for hierarchy creation. A set of optimal precalculated (i.e. partial) paths is stored in the bridge nodes, which define connections and costs between bridge nodes. The set of pre-calculated paths $\mathcal{P}_W$ between neighbor bridge nodes ensures fast replanning by searching only bridge nodes at higher level of the hierarchy. Fig. 18 shows the optimal placement of the bridge nodes for the experimental environment (in the middle of the doorways) connected by the precalculated paths. The paths

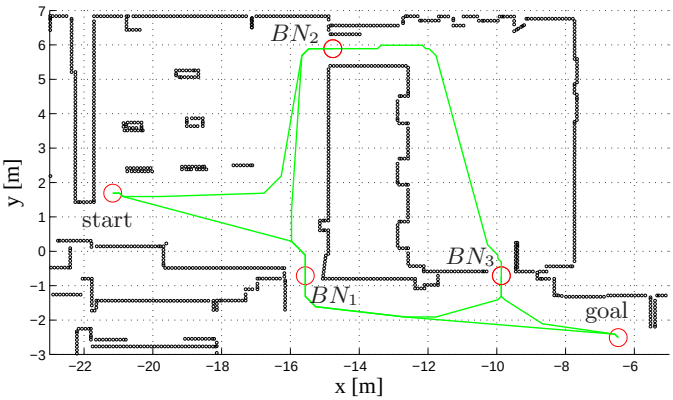

Figure 18: The set of pre-calculated paths between bridge nodes.

from the start node to its neighbor bridge nodes (BN_1 and BN_2) and from the goal's neighbor bridge nodes (BN_1 and BN_3) to the goal node are also shown. In the replanning process, the robot needs only to search the bridge nodes assigned to the room of its current position and then to expand some nodes inside the room. The D* algorithm first searches the bridge nodes and then searches the part of the room expanding from the best bridge node to the robots current position. Then the best bridge node is used as the point S_R for the RD* algorithm. Obviously, by introducing the hierarchical decomposition of the graph the size of the room limits the computational time. For example, in case of avoiding obstacle O_2, the replanning time is reduced from 142 ms to 70 ms and number of expanded nodes is decreased from 26925 to 11644.

5. Conclusion

In this paper a new path planning algorithm – the so-called two-way D* (TWD*) algorithm – is proposed that finds optimal paths in weighted graphs, i.e. in occupancy grid maps with the safety cost mask around the obstacles. The path consists of straight line segments with continuous headings and is the shortest possible path in geometrical space. Similarly to the Witkowski's algorithm, which was used as an inspiration, the TWD* algorithm searches the graph forward and backward. However, unlike the Witkowski's algorithm, which finds the optimal path only in binary occupancy grid maps, the TWD* algorithm finds optimal paths also in weighted occupancy grid maps. This capability of the TWD* algorithm is achieved by applying the D* algorithm in both forward and backward searches of the graph and then calculating the shortest path through the area of optimal paths created by these searches. Unlike the original D* algorithm, the TWD* algorithm produces a shorter and more natural low-cost paths through the grid with a range of continuous headings instead of headings limited to increments of 45°. The proposed algorithm also performs well in changing environments although it is computationally more demanding than the D* algorithm. However, the replanning times can be limited by hierarchical arrangement of the searched graphs. The proposed two-way D* algorithm was tested and compared to the standard D* and Witkowski's algorithms by simulation and experimentally under the same conditions. The results of the tests confirm expected advantages of the proposed path planning algorithm.

Acknowledgements

This research has been supported by the Ministry of Science, Education and Sports of the Republic of Croatia under grant No. $036-0363078-3018$.

References

[1] J. Latombe, Robot Motion Planning, Kluwer Academic Publishers, Dodrecht, Netherlands, 1991.

[2] S. Russell, P. Norvig, J. Canny, J. Malik, D. Edwards, Artificial intelligence: a modern approach, Prentice Hall Englewood Cliffs, NJ, 1995.

[3] P. Hart, N. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE transactions on Systems Science and Cybernetics 4 (2) (1968) 100–107.

[4] A. Stentz, Optimal and efficient path planning for partially-known environments, Robotics and Automation, 1994. Proceedings., 1994 IEEE International Conference on (1994) 3310–3317.

[5] A. Stentz, The focussed D* algorithm for real-time replanning, in: International Joint Conference on Artificial Intelligence, Vol. 14, 1995, pp. 1652–1659.

[6] S. Thrun, W. Burgard, D. Fox, Probabilistic robotics, Cambridge, Massachusetts: MIT Press, 2005.

[7] S. Koenig, M. Likhachev, Fast replanning for navigation in unknown terrain, IEEE Transactions on Robotics 21 (3) (2005) 354–363.

[8] D. Ferguson, A. Stentz, Field D*: An Interpolation-Based Path Planner and Replanner, Robotics Research: Results of the 12 th International Symposium ISRR(STAR: Springer Tracts in Advanced Robotics Series Volume 28) 28 (2007) 239–253.

[9] D. B. Gennery, Traversability analysis and path planning for a planetary rover, Autonomous Robots 6 (2) (1999) 131–146.

[10] D. Ferguson, A. Stentz, Multi-resolution Field D, Intelligent Autonomous Systems 9: IAS-9.

[11] K. Konolige, A gradient method for realtime robot control, in: Proc. of the IEEE/RSJ Interational Conference on Intelligent Robotic Systems (IROS), 2000, pp. 639–646.

[12] H. Samet, Neighbor Finding Techniques for Images Represented by Quadtrees., Computer Graphics and Image Processing 18 (1982) 37–57.

[13] R. Philippsen, R. Siegwart, An interpolated dynamic navigation function, in: Proceedings of the 2005 IEEE International Conference on Robotics and Automation (ICRA), Vol. 4, Citeseer, 2005, pp. 3782–3789.

[14] C. Witkowski, A parallel processor algorithm for robot route planning, Int. Joint Conf. on Artificial Intelligence (IJCAI), Karlsruhe, West Germany (1983) 827–829.

[15] M. Seder, I. Petrović, Dynamic window based approach to mobile robot motion control in the presence of moving obstacles, Robotics and Automation, 2007 IEEE International Conference on (2007) 1986–1991.

[16] O. Khatib, Real-Time Obstacle Avoidance for Manipulators and Mobile Robots, The International Journal of Robotics Research 5 (1) (1986) 90–98.

[17] S. LaValle, Planning algorithms, Cambridge Univ Pr, 2006.

[18] C. Stachniss, W. Burgard, An Integrated Approach to Goal-directed Obstacle Avoidance under Dynamic Constraints for Dynamic Environments, IROS'02, IEEE/RSJ International Conference on Intelligent Robots and Systems.

[19] K. Arras, J. Persson, N. Tomatis, R. Siegwart, Real-Time Obstacle Avoidance for Polygonal Robots with a Reduced Dynamic Window, ICRA'02, IEEE International Conference on Robotics and Automation.

[20] O. Brock, O. Khatib, High-speed navigation using the Global Dynamic Window Approach, ICRA'99, IEEE International Conference on Robotics and Automation.

[21] M. Seder, K. Maček, I. Petrović, An integrated approach to real-time mobile robot control in partially known indoor environments, Industrial Electronics Society, 2005. IECON 2005. 32nd Annual Conference of IEEE (2005) 1785–1790.

[22] J. Fernandez, J. Gonzalez, Hierarchical graph search for mobile robot path planning, in: IEEE International Conference on Robotics and Automation, 1998, pp. 656–661.

[23] J.-A. Fernández-Madrigal, J. González, Multihierarchical graph search, IEEE Transactions on Pattern Analysis and Machine Intelligence 24 (1) (2002) 103–113. doi:http://dx.doi.org/10.1109/34.982887.

[24] R. Korf, Planning as search: a quantitative approach, Artificial Intelligence 33 (1) (1987) 65–68.

[25] M. Seder, P. Mostarac, I. Petrović, Hierarchical path planning of mobile robots in complex indoor environments, Transactions of the Institute of Measurement and Control, doi:10.1177/0142331208100107.

[26] D. Cagigas, Hierarchical D* algorithm with materialization of costs for robot path planning, Robotics and Autonomous Systems 52 (2-3) (2005) 190–208.